Let’s play (again) with Predator the thief

fumik0.com/2019/12/25/lets-play-again-with-predator-the-thief/

fumko December 25, 2019

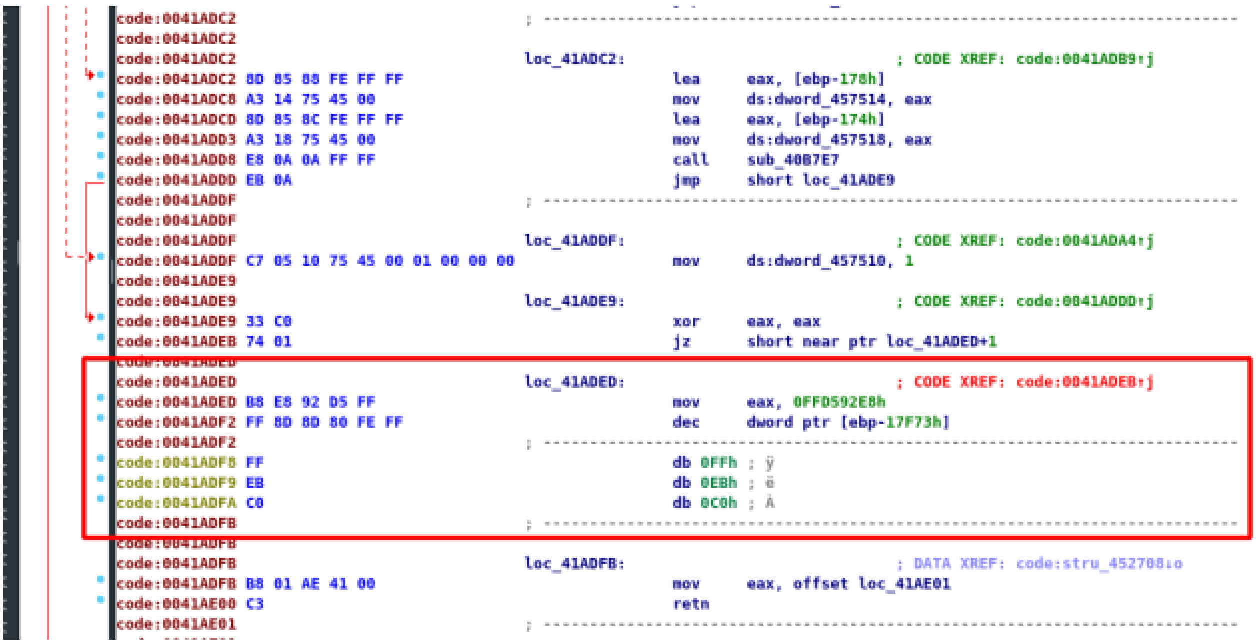
Whenever I reverse a sample, I am mostly interested in how it was developed, even if in the end the techniques employed are generally the same, I am always curious about what was the way to achieve a task, or just simply understand the code philosophy of a piece of code. It is a very nice way to spot different trending and discovering (sometimes) new tricks that you never know it was possible to do. This is one of the main reasons, I love digging mostly into stealers/clippers for their accessibility for being reversed, and enjoying malware analysis as a kind of game (unless some exceptions like Nymaim that is literally hell).

It’s been 1 year and a half now that I start looking into “Predator The Thief”, and this malware has evolved over time in terms of content added and code structure. This impression could be totally different from others in terms of stealing tasks performed, but based on my first in-depth analysis,, the code has changed too much and it was necessary to make another post on it.

This one will focus on some major aspects of the 3.3.2 version, but will not explain everything (because some details have already been mentioned in other papers, some subjects are known). Also, times to times I will add some extra commentary about malware analysis in general.

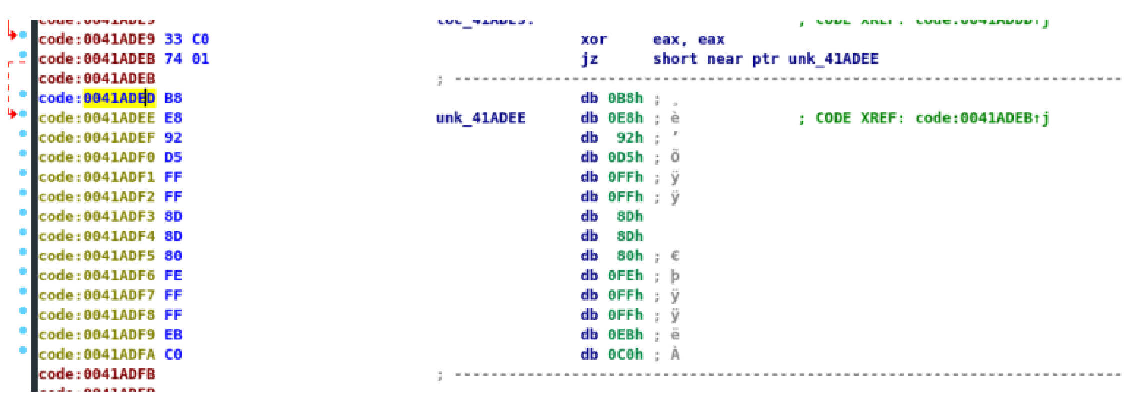
Anti-Disassembly

When you open an unpacked binary in IDA or other disassembler software like GHIDRA, there is an amount of code that is not interpreted correctly which leads to rubbish code, the incapacity to construct instructions or showing some graph. Behind this, it’s obvious that an anti-disassembly trick is used.

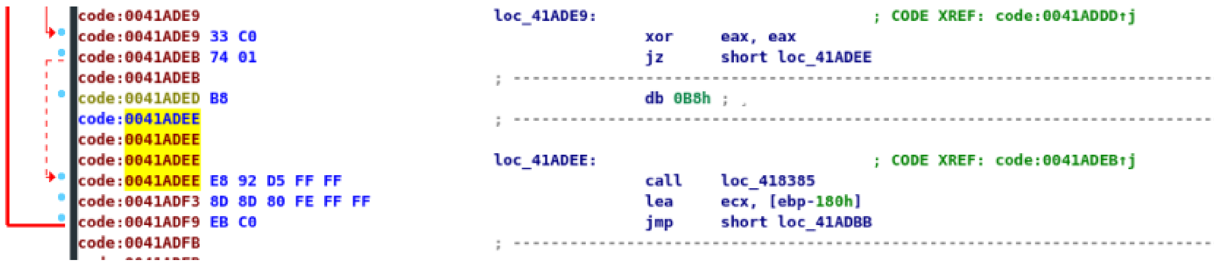


The technique exploited here is known and used in the wild by other malware, it requires just a few opcodes to process and leads at the end at the creation of a false branch. In this case, it begins with a simple xor instruction that focuses on configuring the zero flag and forcing the JZ jump condition to work no matter what, so, at this stage, it’s understandable that something suspicious is in progress. Then the MOV opcode (0xB8) next to the jump is a 5 bytes instruction and disturbing the disassembler to consider that this instruction is the right one to interpret beside that the correct opcode is inside this one, and in the end, by choosing this wrong path malicious tasks are hidden.

Of course, fixing this issue is simple, and required just a few seconds. For example with IDA, you need to undefine the MOV instruction by pressing the keyboard shortcut “U”, to produce this pattern.



Then skip the 0xB8 opcode, and pushing on “C” at the 0xE8 position, to configure the disassembler to interpret instruction at this point.



Replacing the 0xB8 opcode by 0x90. with a hexadecimal editor, will fix the issue. Opening again the patched PE, you will see that IDA is now able to even show the graph mode.

After patching it, there are still some parts that can’t be correctly parsed by the disassembler, but after reading some of the code locations, some of them are correct, so if you want to create a function, you can select the “loc” section then pushed on “P” to create a sub-function, of course, this action could lead to some irreversible thing if you are not sure about your actions and end to restart again the whole process to remove a the ant-disassembly tricks, so this action must be done only at last resort.

Code Obfuscation

Whenever you are analyzing Predator, you know that you will have to deal with some obfuscation tricks almost everywhere just for slowing down your code analysis. Of course, they are not complicated to assimilate, but as always, simple tricks used at their finest could turn a simple fun afternoon to literally “welcome to Dark Souls”. The concept was already there in the first in-depth analysis of this malware, and the idea remains over and over with further updates on it. The only differences are easy to guess :

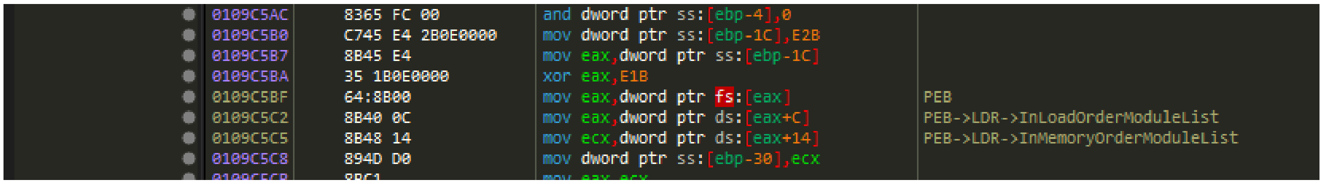
- More layers of obfuscation have been added
- Techniques already used are just adjusted.
- More dose of randomness

As a reversing point of view, I am considering this part as one the main thing to recognized this stealer, even if of course, you can add network communication and C&C pattern as other ways for identifying it, inspecting the code is one way to clarify doubts (and I understand that this statement is for sure not working for every malware), but the idea is that nowadays it’s incredibly easy to make mistakes by being dupe by rules or tags on sandboxes, due to similarities based on code-sharing, or just literally creating false flag.

GetModuleAddress

Already there in a previous analysis, recreating the GetProcAddress is a popular trick to hide an API call behind a simple register call. Over the updates, the main idea is still there but the main procedures have been modified, reworked or slightly optimized.

First of all, we recognized easily the PEB retrieved by spotting fs[0x30] behind some extra instructions.



then from it, the loader data section is requested for two things:

- Getting the InLoadOrderModuleList pointer
- Getting the InMemoryOrderModuleList pointer

For those who are unfamiliar by this, basically, the PEB_LDR_DATA is a structure is where is stored all the information related to the loaded modules of the process.

Then, a loop is performing a basic search on every entry of the module list but in “memory order” on the loader data, by retrieving the module name, generating a hash of it and when it’s done, it is compared with a hardcoded obfuscated hash of the kernel32 module and obviously, if it matches, the module base address is saved, if it’s not, the process is repeated again and again.

The XOR kernel32 hashes compared with the one created

Nowadays, using hashes for a function name or module name is something that you can see in many other malware, purposes are multiple and this is one of the ways to hide some actions. An example of this code behavior could be found easily on the internet and as I said above, this one is popular and already used.

GetProcAddress / GetLoadLibrary

Always followed by GetModuleAddress, the code for recreating GetProcAddress is by far the same architecture model than the v2, in term of the concept used. If the function is forwarded, it will basically perform a recursive call of itself by getting the forward address, checking if the library is loaded then call GetProcAddress again with new values.

Xor everything

It’s almost unnecessary to talk about it, but as in-depth analysis, if you have never read the other article before, it’s always worth to say some words on the subject (as a reminder). The XOR encryption is a common cipher that required a rudimentary implementation for being effective :

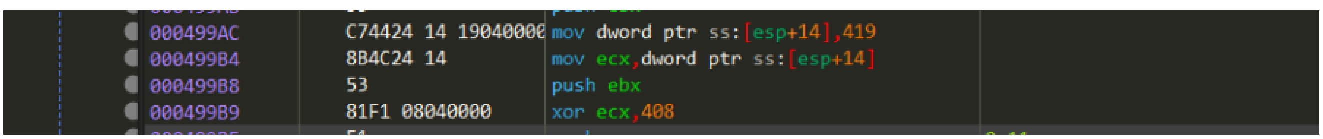
- Only one operator is used (XOR)
- it’s not consuming resources.
- It could be used as a component of other ciphers

This one is extremely popular in malware and the goal is not really to produce strong encryption because it’s ridiculously easy to break most of the time, they are used for hiding information or keywords that could be triggering alerts, rules…

- Communication between host & server
- Hiding strings
- Or… simply used as an absurd step for obfuscating the code
- etc…

A typical example in Predator could be seeing huge blocks with only two instructions (XOR & MOV), where stacks strings are decrypted X bytes per X bytes by just moving content on a temporary value (stored on EAX), XORed then pushed back to EBP, and the principle is reproduced endlessly again and again. This is rudimentary, In this scenario, it’s just part of the obfuscation process heavily abused by predator, for having an absurd amount of instruction for simple things.

Also for some cases, When a hexadecimal/integer value is required for an API call, it could be possible to spot another pattern of a hardcoded string moved to a register then only one XOR instruction is performed for revealing the correct value, this trivial thing is used for some specific cases like the correct position in the TEB for retrieving the PEB, an RVA of a specific module, …



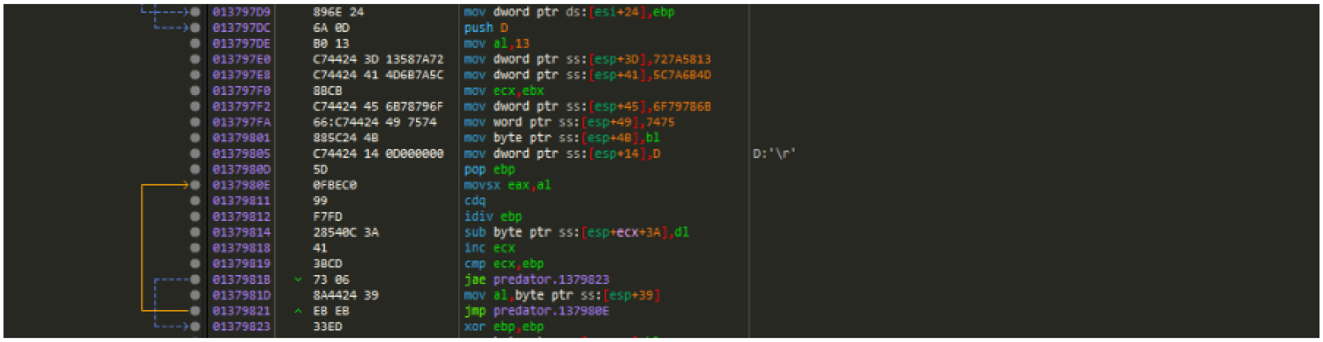
Finally, the most common one, there is also the classic one used by using a for loop for a one key length XOR key, seen for decrypting modules, functions, and other things…

```
str = ... # encrypted string

for i, s in enumerate(str):
  s[i] = s[i] ^ s[len(str)-1]
```

Sub everything

Let's consider this as a perfect example of "let's do the same exact thing by just changing one single instruction", so in the end, a new encryption method is used with no effort for the development. That's how a SUB instruction is used for doing the substitution cipher. The only difference that I could notice it's how the key is retrieved.



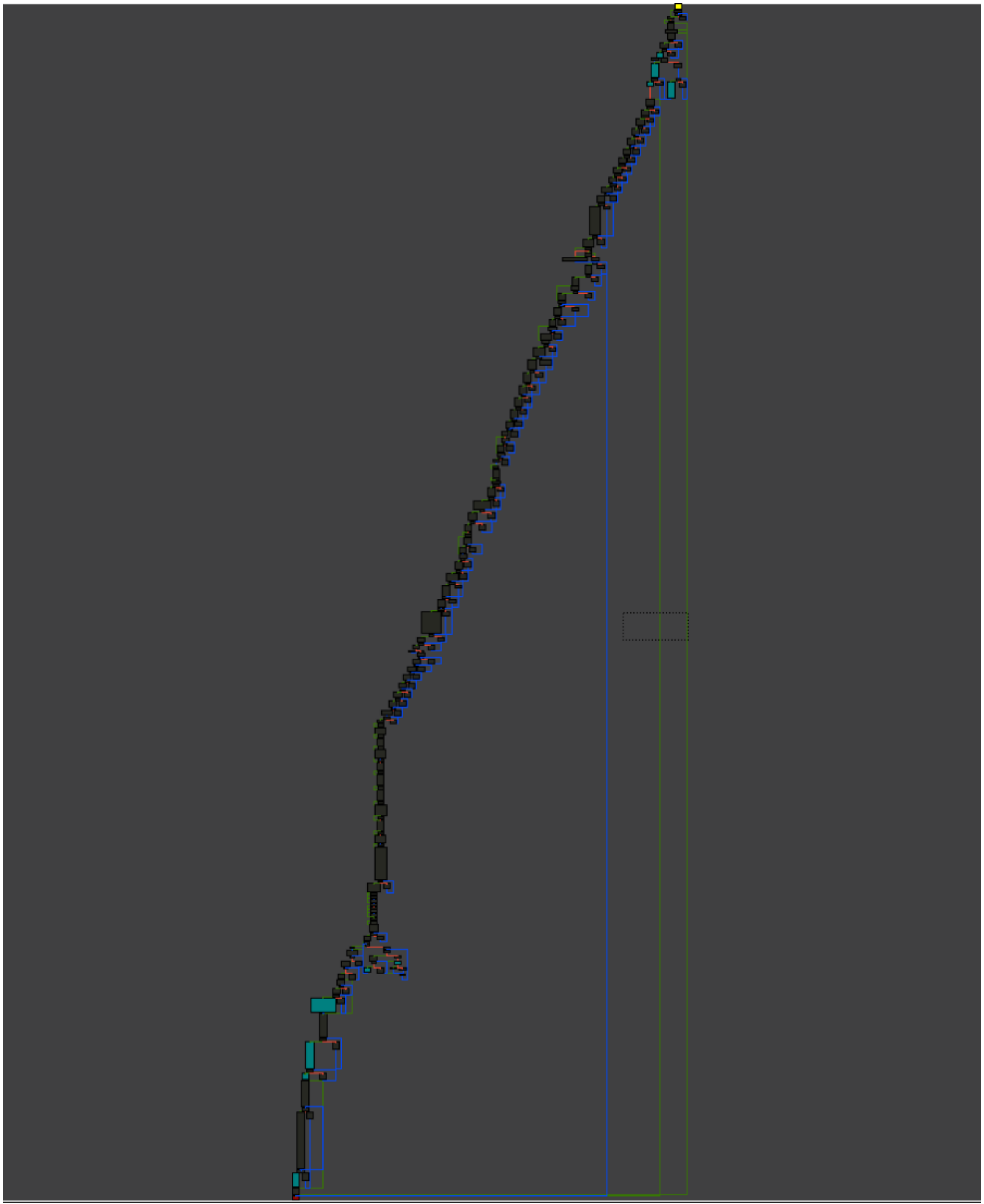
Besides having something hardcoded directly, a signed 32-bit division is performed, easily noticeable by the use of cdq & idiv instructions, then the dl register (the remainder) is used for the substitution.

Stack Strings

```
code:0041A5F8                 mov     byte ptr [ebp-30h], 27h
code:0041A5FC                 push    'R'
code:0041A5FE                 lea     ecx, [ebp-30h]
code:0041A601                 call    func_xor
code:0041A606                 mov     [ebp-2Fh], al
code:0041A609                 push    'e'
code:0041A60B                 lea     ecx, [ebp-30h]
code:0041A60E                 call    func_xor
code:0041A613                 mov     [ebp-2Eh], al
code:0041A616                 push    'l'
code:0041A618                 lea     ecx, [ebp-30h]
code:0041A61B                 call    func_xor
code:0041A620                 mov     [ebp-2Dh], al
code:0041A623                 push    'e'
code:0041A625                 lea     ecx, [ebp-30h]
code:0041A628                 call    func_xor
code:0041A62D                 mov     [ebp-2Ch], al
code:0041A630                 push    'a'
code:0041A632                 lea     ecx, [ebp-30h]
code:0041A635                 call    func_xor
code:0041A63A                 mov     [ebp-2Bh], al
code:0041A63D                 push    's'
code:0041A63F                 lea     ecx, [ebp-30h]
code:0041A642                 call    func_xor
code:0041A647                 mov     [ebp-2Ah], al
code:0041A64A                 push    'e'
code:0041A64C                 lea     ecx, [ebp-30h]
code:0041A64F                 call    func_xor
code:0041A654                 mov     [ebp-29h], al
code:0041A657                 push    'M'
code:0041A659                 lea     ecx, [ebp-30h]
code:0041A65C                 call    func_xor
code:0041A661                 mov     [ebp-28h], al
code:0041A664                 push    'u'
code:0041A666                 lea     ecx, [ebp-30h]
code:0041A669                 call    func_xor
code:0041A66E                 mov     [ebp-27h], al
code:0041A671                 push    't'
code:0041A673                 lea     ecx, [ebp-30h]
code:0041A676                 call    func_xor
code:0041A67B                 mov     [ebp-26h], al
code:0041A67E                 push    'e'
code:0041A680                 lea     ecx, [ebp-30h]
code:0041A683                 call    func_xor
code:0041A688                 mov     [ebp-25h], al
code:0041A68B                 push    'x'
code:0041A68D                 lea     ecx, [ebp-30h]
code:0041A690                 call    func_xor
code:0041A695                 mov     [ebp-24h], al
code:0041A698                 mov     [ebp-23h], bl
code:0041A69B                 mov     esi, ebx
```

What’s the result in the end?

Merging these obfuscation techniques leads to a nonsense amount of instructions for a basic task, which will obviously burn you some hours of analysis if you don’t take some time for cleaning a bit all that mess with the help of some scripts or plenty other ideas, that could trigger in your mind. It could be nice to see these days some scripts released by the community.



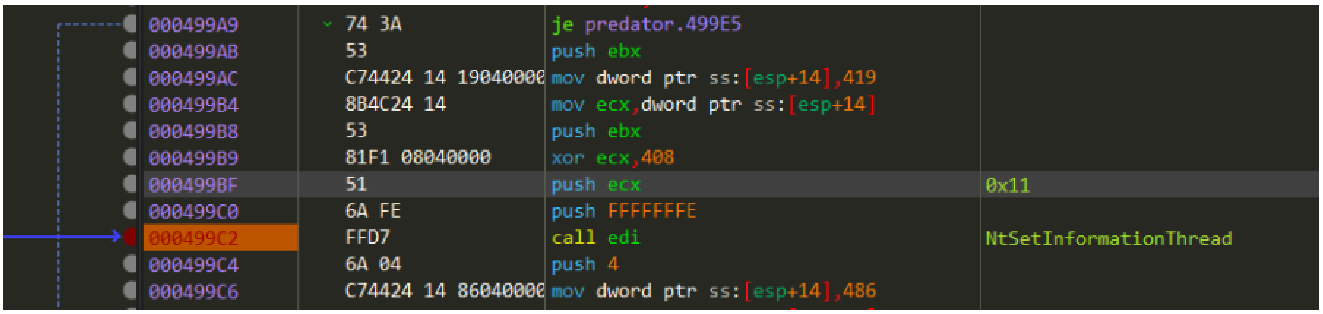
Simple tricks lead to nonsense code

Anti-Debug

There are plenty of techniques abused here that was not in the first analysis, this is not anymore a simple PEB.BeingDebugged or checking if you are running a virtual machine, so let's dig into them. one per one except CheckRemoteDebugger! This one is enough to understand by itself :’)

NtSetInformationThread

One of the oldest tricks in windows and still doing its work over the years. Basically in a very simple way (because there is a lot thing happening during the process), NtSetInformationThread is called with a value (0x11) obfuscated by a XOR operator. This parameter is a ThreadInformationClass with a specific enum called ThreadHideFromDebugger and when it’s executed, the debugger is not able to catch any debug information. So the supposed pointer to the corresponding thread is, of course, the malware and when you are analyzing it with a debugger, it will result to detach itself.



CloseHandle/NtClose

Inside WinMain, a huge function is called with a lot of consecutive anti-debug tricks, they were almost all indirectly related to some techniques patched by TitanHide (or strongly looks like), the first one performed is a really basic one, but pretty efficient to do the task.

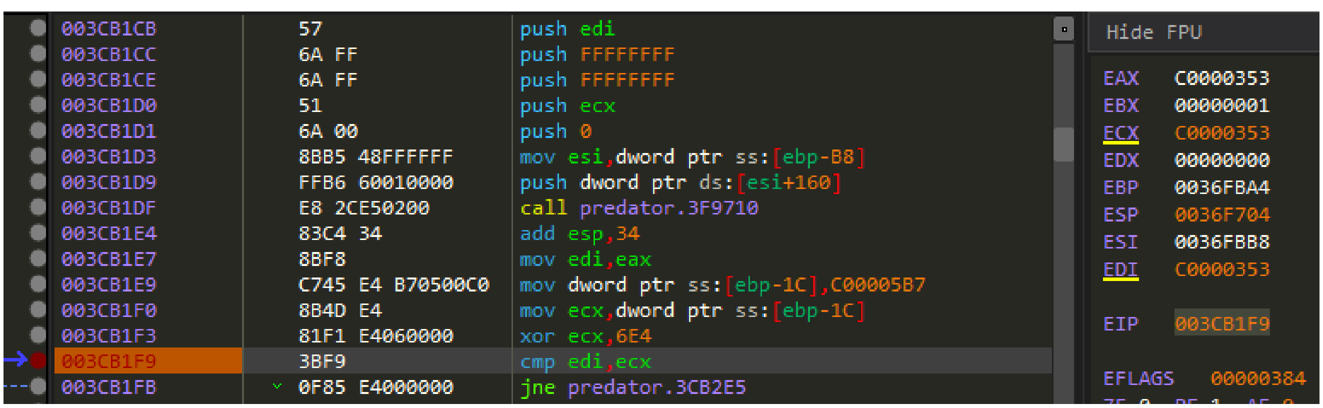
Basically, when CloseHandle is called with an inexistent handle or an invalid one, it will raise an exception and whenever you have a debugger attached to the process, it will not like that at all. To guarantee that it’s not an issue for a normal interaction a simple __try / __except method is used, so if this API call is requested, it will safely lead to the end without any issue.

The invalid handle used here is a static one and it’s L33T code with the value *0xBAADAA55* and makes me bored as much as this face.

That’s not a surprise to see stuff like this from the malware developer. Inside jokes, l33t values, animes and probably other content that I missed are something usual to spot on Predator.

ProcessDebugObjectHandle

When you are debugging a process, Microsoft Windows is creating a “Debug” object and a handle corresponding to it. At this point, when you want to check if this object exists on the process, NtQueryInformationProcess is used with the ProcessInfoClass initialized by 0x1e (that is in fact, ProcessDebugObjectHandle).

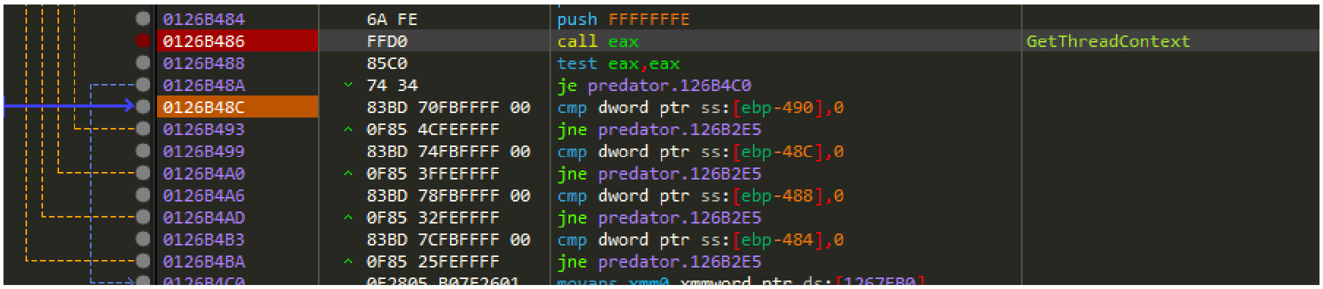


In this case, the NTStatus value (returning result by the API call) is an error who as the ID 0xC0000353, aka STATUS_PORT_NOT_SET. This means, *“An attempt to remove a process’s DebugPort was made, but a port was not already associated with the process.”.* The anti-debug trick is to verify if this error is there, that’s all.

NtGetContextThread

This one is maybe considered as pretty wild if you are not familiar with some hardware breakpoints. Basically, there are some registers that are called “**D**ebug **R**egister” and they are using the **DR**X nomenclature (DR0 to DR7). When GetThreadContext is called, the function will retrieve al the context information from a thread.

For those that are not familiar with a context structure, it contains all the register data from the corresponding element. So, with this data in possession, it only needs to check if those DRX registers are initiated with a value not equal to 0.

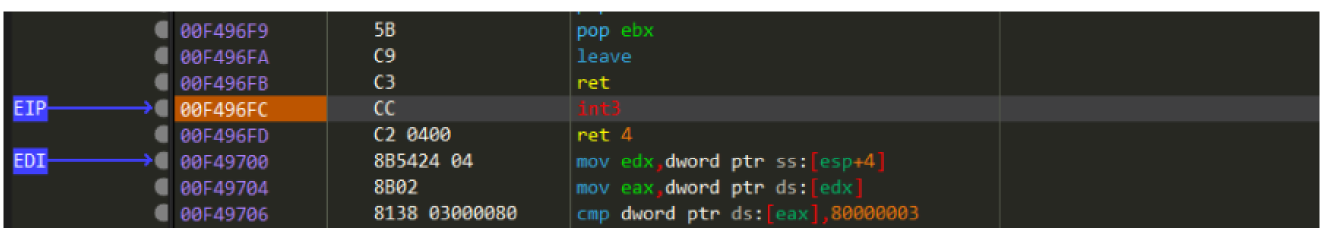


On the case here, it’s easily spottable to see that 4 registers are checked

```
if (ctx->Dr0 != 0 || ctx->Dr1 != 0 || ctx->Dr2 != 0 || ctx->Dr3 != 0)
```

Int 3 breakpoint

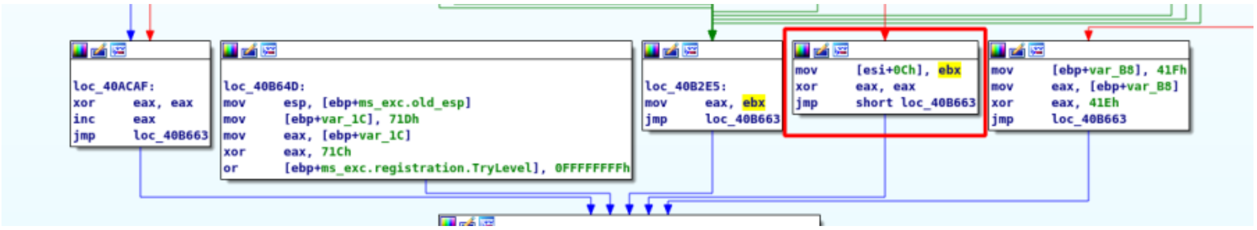
int 3 (or Interrupt 3) is a popular opcode to force the debugger to stop at a specific offset. As said in the title, this is a breakpoint but if it’s executed without any debugging environment, the exception handler is able to deal with this behavior and will continue to run without any issue. Unless I missed something, here is the scenario.



By the way, as another scenario used for this one (the int 3), the number of this specific opcode triggered could be also used as an incremented counter, if the counter is above a specific value, a simplistic condition is sufficient to check if it’s executed into a debugger in that way.

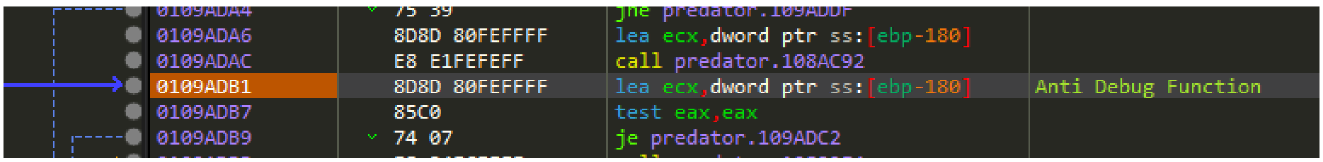
Debug Condition

With all the techniques explained above, in the end, they all lead to a final condition step if of course, the debugger hasn’t crashed. The checking task is pretty easy to understand and it remains to a simple operation: “setting up a value to EAX during the anti-debug function”, if everything is correct this register will be set to zero, if not we could see all the different values that could be possible.



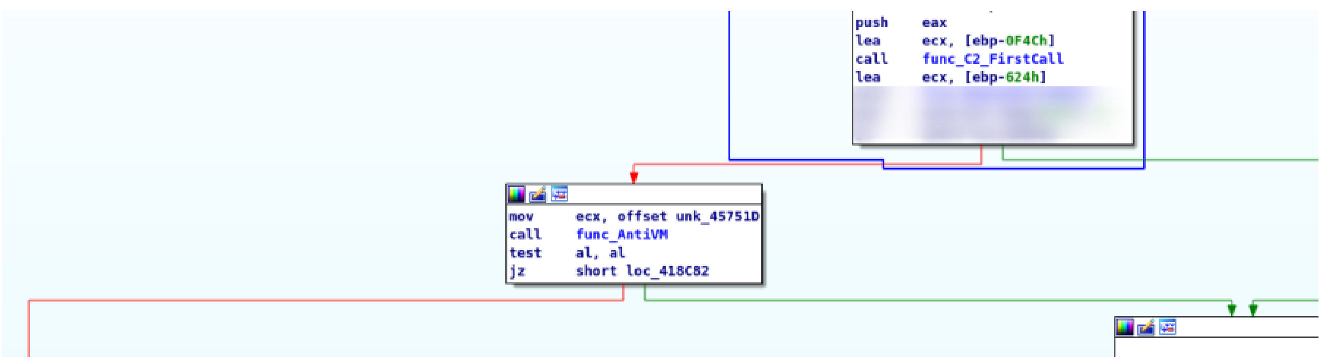
bloc in red is the correct condition over all the anti-debug tests

…And when the Anti-Debug function is done, the register EAX is checked by the test operator, so the ZF flag is determinant for entering into the most important loop that contains the main function of the stealer.



Anti-VM

The Anti VM is presented as an option in Predator and is performed just after the first C&C requests.



Tricks used are pretty olds and basically using Anti-VM Instructions

- SIDT
- SGDT
- STR
- CPUID (Hypervisor Trick)

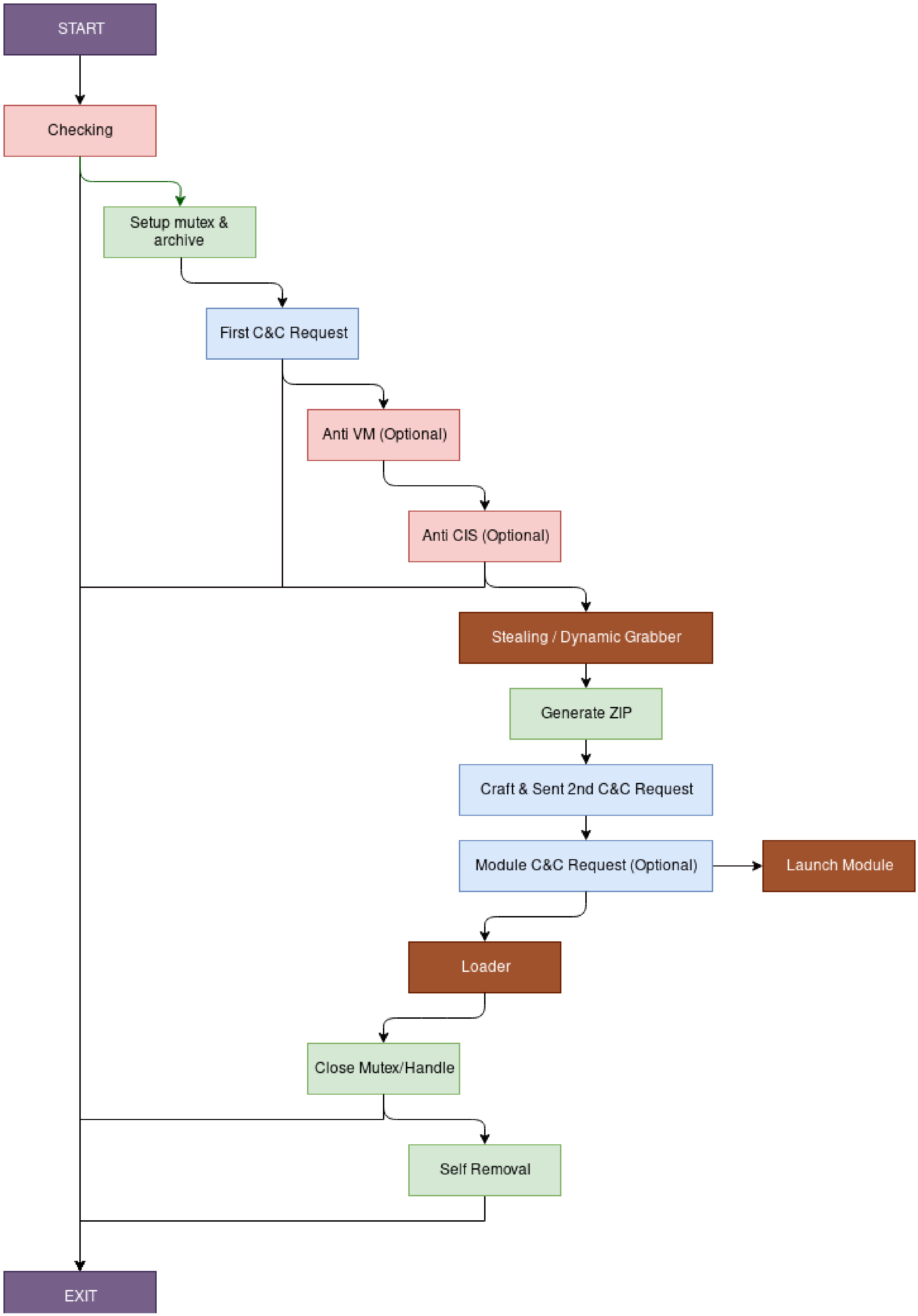
By curiosity, this option is not by default performed if the C&C is not reachable.

Paranoid & Organized Predator

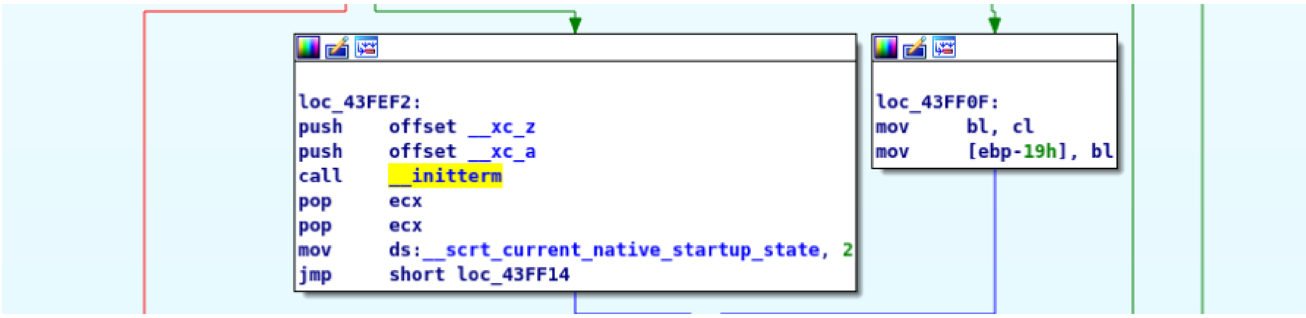
When entering into the “big main function”, the stealer is doing “again” extra validations if you have a valid payload (and not a modded one), you are running it correctly and being sure again that you are not analyzing it.

This kind of paranoid checking step is a result of the multiple cases of cracked builders developed and released in the wild (mostly or exclusively at a time coming from XakFor.Net). Pretty wild and fun to see when Anti-Piracy protocols are also seen in the malware scape.

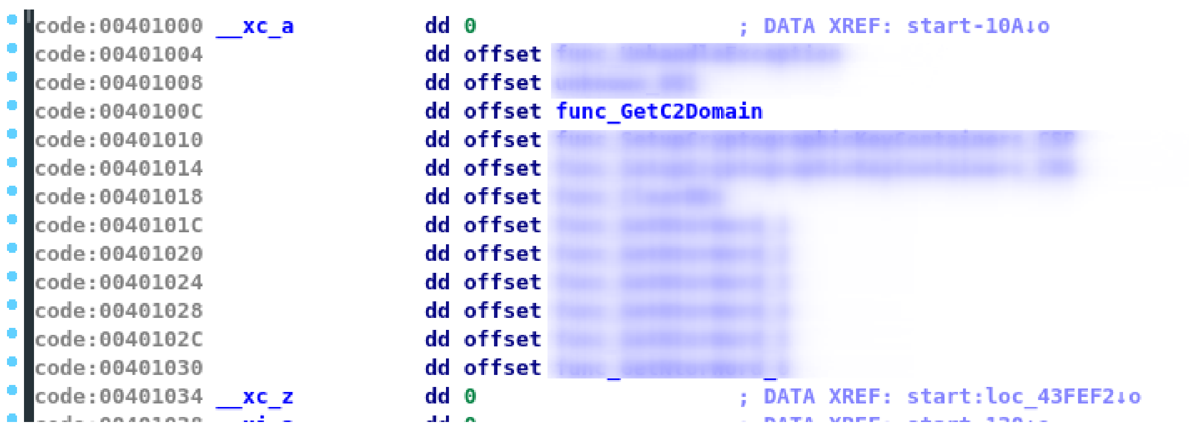
Then the malware is doing a classic organized setup to perform all the requested actions and could be represented in that way.


START
Checking
Setup mutex & archive
First C&C Request
Anti VM (Optional)
Anti CIS (Optional)
Stealing / Dynamic Grabber
Generate ZIP
Craft & Sent 2nd C&C Request
Module C&C Request (Optional)
Launch Module
Loader
Close Mutex/Handle
Self Removal
EXIT

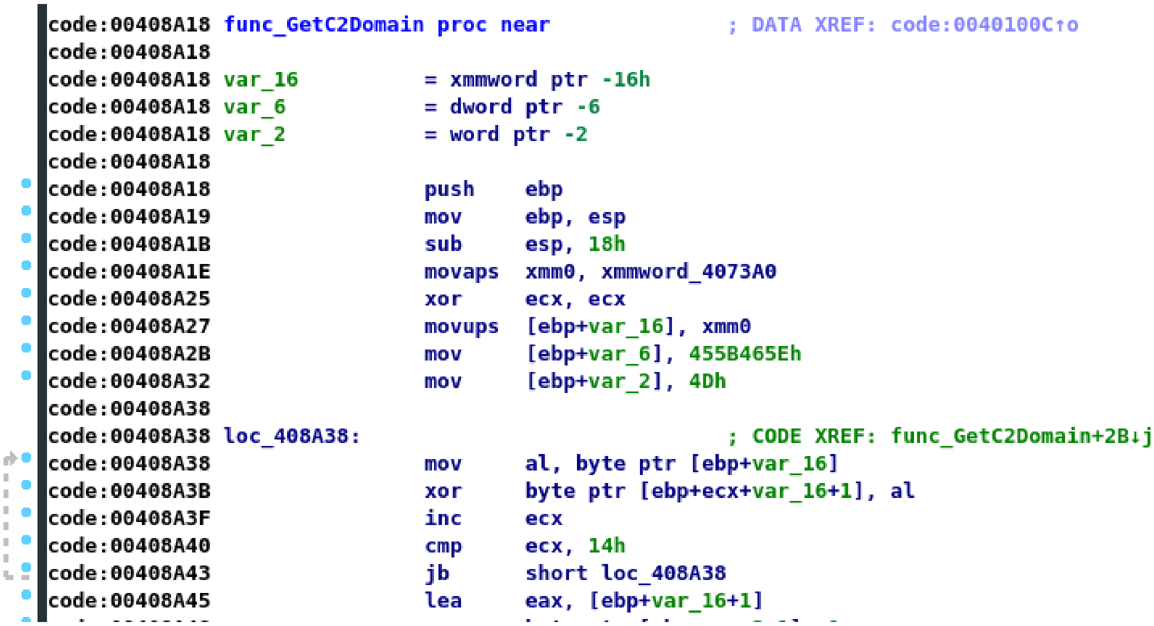
Of course as usual and already a bit explained in the first paper, the C&C domain is retrieved in a table of function pointers before the execution of the WinMain function (where the payload is starting to do tasks).



You can see easily all the functions that will be called based on the starting location (__xc_z) and the ending location (__xc_z).



Then you can spot easily the XOR strings that hide the C&C domain like the usual old predator malware.

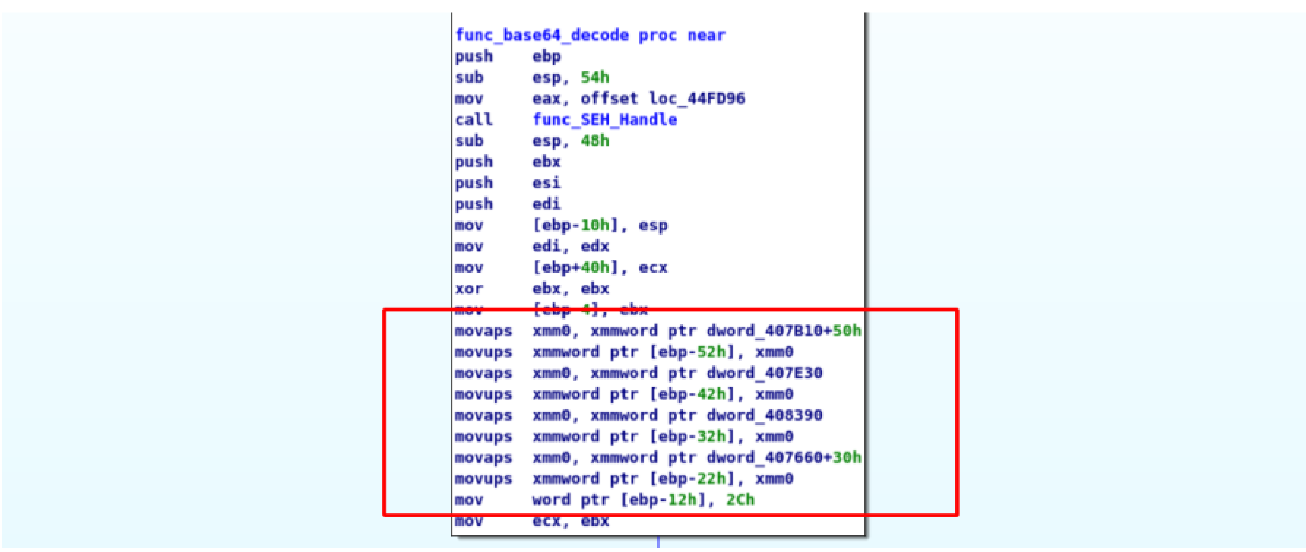


Data Encryption & Encoding

Besides using XOR almost absolutely everywhere, this info stealer is using a mix of RC4 encryption and base64 encoding whenever it is receiving data from the C&C. Without using specialized tools or paid versions of IDA (or whatever other software), it could be a bit challenging to recognize it (when you are a junior analyst), due to some modification of some part of the code.

Base64

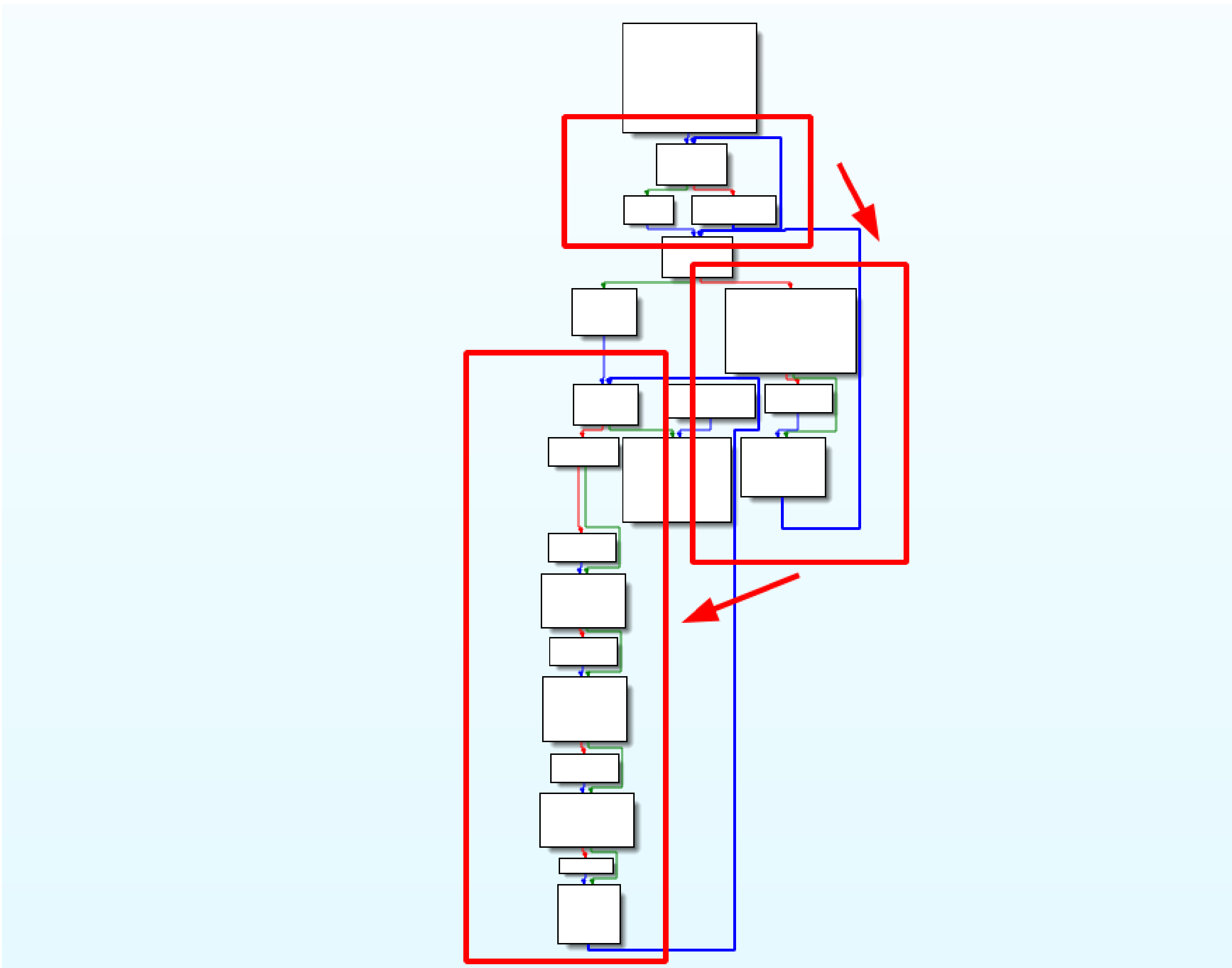
For the Base64 functions, it's extremely easy to spot them, with the symbol values on the register before and after calls. The only thing to notice with them, it's that they are using a typical signature… A whole bloc of XOR stack strings, I believed that this trick is designed to hide an eventual Base64 alphabet from some Yara rules.



By the way, the rest of the code remains identical to standard base64 algorithms.

RC4

For RC4, things could be a little bit messy if you are not familiar at all with encryption algorithm on a disassembler/debugger, for some cases it could be hell, for some case not. Here, it's, in fact, this amount of code for performing the process.



Blocs are representing the Generation of the array S, then performing the Key-Scheduling Algorithm (KSA) by using a specific secret key that is, in fact, the C&C domain! (if there is no domain, but an IP hardcoded, this IP is the secret key), then the last one is the Pseudo-random generation algorithm (PRGA).

For more info, some resources about this algorithm below:

- Stack Overflow example
- RC4 Algorithm (Wikipedia)

Mutex & Hardware ID

The Hardware ID (HWID) and mutex are related, and the generation is quite funky, I would say, even if most of the people will consider this as something not important to investigate, I love small details in malware, even if their role is maybe meaningless, but for me, every detail counts no matter what (even the stupidest one).

Here the hardware ID generation is split into 3 main parts. I had a lot of fun to understand how this one was created.

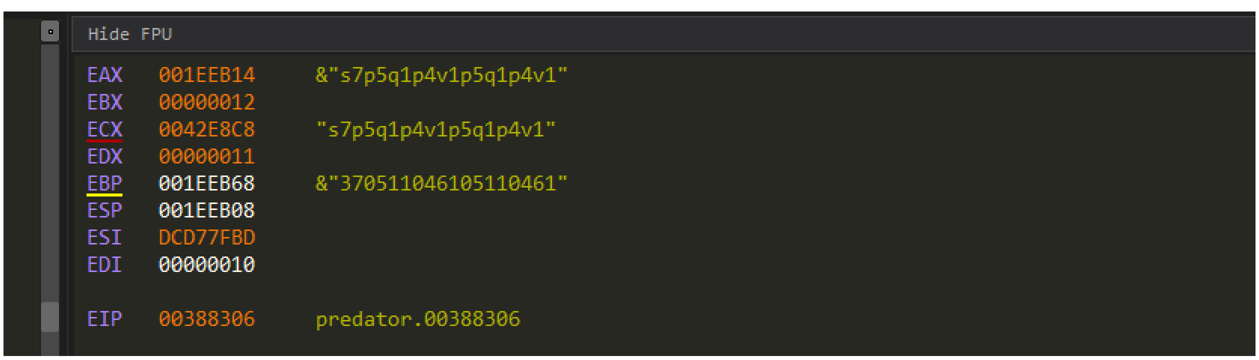
First, it will grab all the available logical drives on the compromised machine, and for each of them, the serial number is saved into a temporary variable. Then, whenever a new drive is found, the hexadecimal value is added to it. so basically if the two drives have the serial number “44C5-F04D” and “1130-DDFF”, so ESI will receive 0x44C5F04D then will add 0x1130DFF.

When it’s done, this value is put into a while loop that will divide the value on ESI by 0xA and saved the remainder into another temporary variable, the loop condition breaks when ESI is below 1. Then the results of this operation are saved, duplicated and added to itself the last 4 bytes (i.e 1122334455 will be 1122334455**22334455**).

If this is not sufficient, the value is put into another loop for performing this operation.

```
for i, s in enumerate(str):
  if i & 1:
    a += chr(s) + 0x40
  else:
    a += chr(s)
```

It results in the creation of an alphanumeric string that will be the archive filename used during the POST request to the C&C.

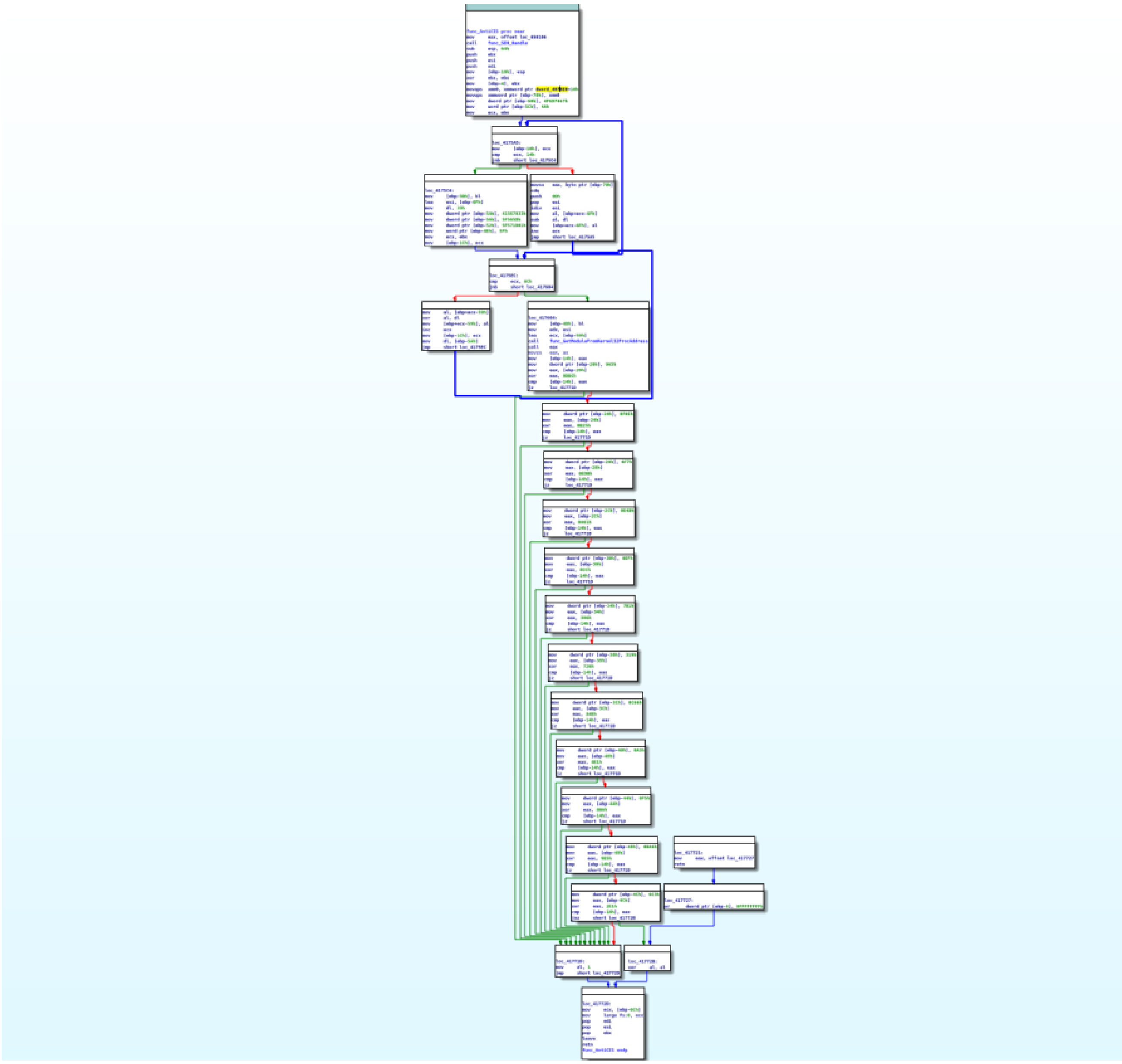


the generated hardware ID based on the serial number devices

But wait! there is more… This value is in part of the creation of the mutex name… with a simple base64 operation on it and some bit operand operation for cutting part of the base64 encoding string for having finally the mutex name!

Anti-CIS

A classic thing in malware, this feature is used for avoiding infecting machines coming from the Commonwealth of Independent States (CIS) by using a simple API call GetUserDefaultLangID.



The value returned is the language identifier of the region format setting for the user and checked by a lot of specific language identifier, of courses in every situation, all the values that are tested, are encrypted.

Language ID	SubLanguage Symbol	Country
0x0419	SUBLANG_RUSSIAN_RUSSIA	Russia
0x042b	SUBLANG_ARMENIAN_ARMENIA	Armenia

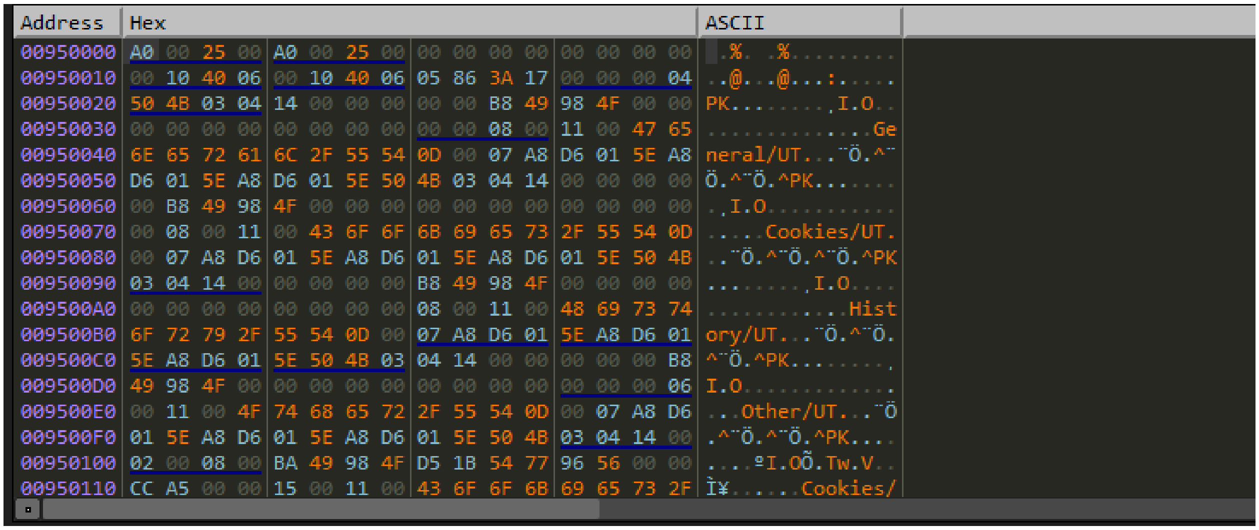
0x082c	SUBLANG_AZERI_CYRILLIC	Azerbaijan
0x042c	SUBLANG_AZERI_LATIN	Azerbaijan
0x0423	SUBLANG_BELARUSIAN_BELARUS	Belarus
0x0437	SUBLANG_GEORGIAN_GEORGIA	Georgia
0x043f	SUBLANG_KAZAK_KAZAKHSTAN	Kazakhstan
0x0428	SUBLANG_TAJIK_TAJIKISTAN	Tajikistan
0x0442	SUBLANG_TURKMEN_TURKMENISTAN	Turkmenistan
0x0843	SUBLANG_UZBEK_CYRILLIC	Uzbekistan
0x0443	SUBLANG_UZBEK_LATIN	Uzbekistan
0x0422	SUBLANG_UKRAINIAN_UKRAINE	Ukraine

Files, files where are you?

When I reversed for the first time this stealer, files and malicious archive were stored on the disk then deleted. But right now, this is not the case anymore. Predator is managing all the stolen data into memory for avoiding as much as possible any extra traces during the execution.

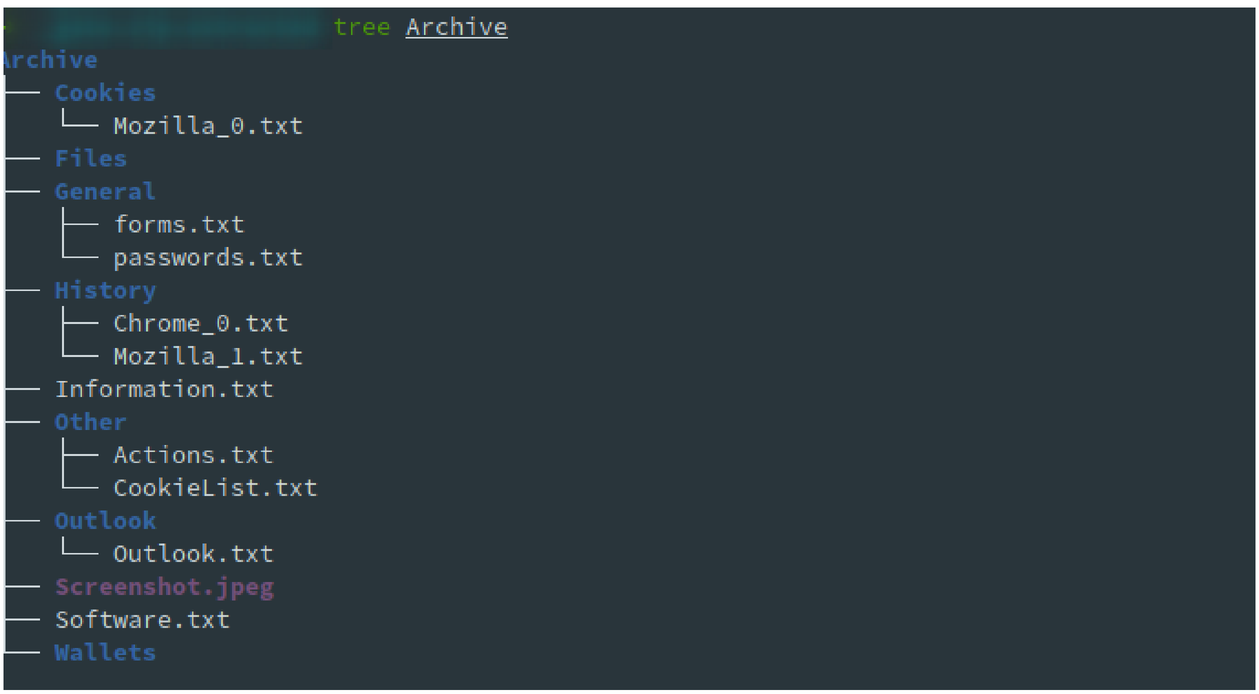
Predator is nowadays creating in memory a lot of allocated pages and temporary files that will be used for interactions with real files that exist on the disk. Most of the time it's basically getting handles, size and doing some operation for opening, grabbing content and saving them to a place in memory. ***This explanation is summarized in a "very" simplify way because there are a lot of cases and scenarios to manage this.***

Another point to notice is that the archive (using ZIP compression), is also created in memory by selecting folder/files.



The generated archive in memory

It doesn't mean that the whole architecture for the files is different, it's the same format as before.



an example of archive intercepted during the C&C Communication

Stealing

After explaining this many times about how this stuff, the fundamental idea is boringly the same for every stealer:

- Check
- Analyzing (optional)
- Parsing (optional)
- Copy

- Profit
- Repeat

What could be different behind that, is how they are obfuscating the files or values to check… and guess what… every malware has their specialties (whenever they are not decided to copy the same piece of code on Github or some whatever generic .NET stealer) and in the end, there is no black magic, just simple (or complex) enigma to solve. As a malware analyst, when you are starting into analyzing stealers, you want literally to understand everything, because everything is new, and with the time, you realized the routine performed to fetch the data and how stupid it is working well (as reminder, it might be not always that easy for some highly specific stuff).

In the end, you just want to know the targeted software, and only dig into those you haven't seen before, but every time the thing is the same:

- Checking dumbly a path
- Checking a register key to have the correct path of a software
- Checking a shortcut path based on an icon
- etc…

Beside that Predator the Thief is stealing a lot of different things:

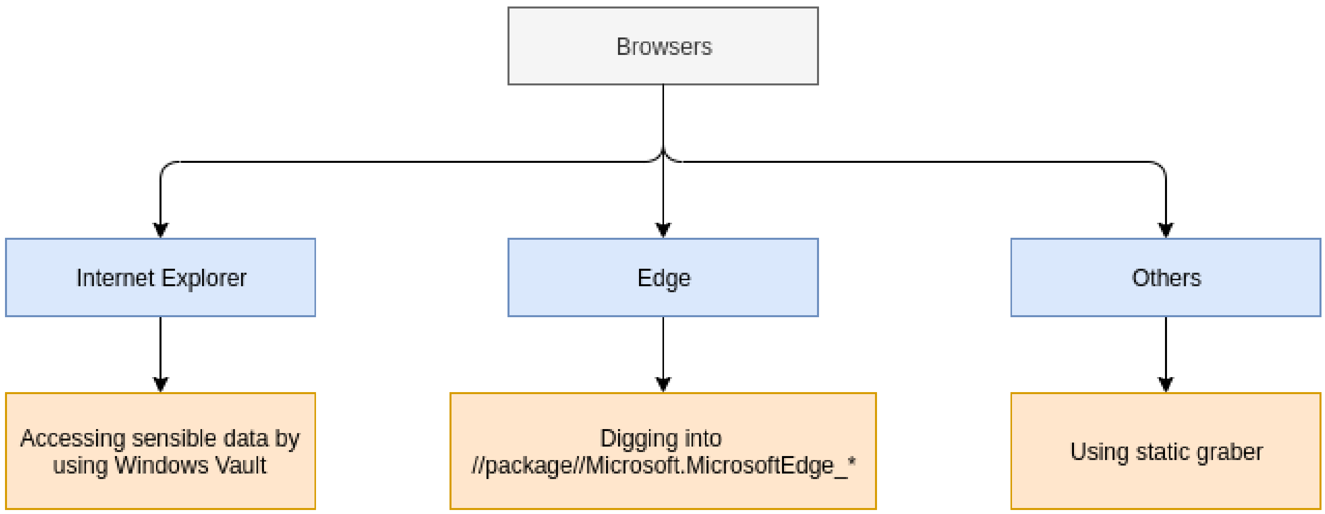
1. Grabbing content from Browsers (Cookies, History, Credentials)
2. Harvesting/Fetching Credit Cards
3. Stealing sensible information & files from Crypto-Wallets
4. Credentials from FTP Software
5. Data coming from Instant communication software
6. Data coming from Messenger software
7. 2FA Authenticator software
8. Fetching Gaming accounts
9. Credentials coming from VPN software
10. Grabbing specific files (also dynamically)
11. Harvesting all the information from the computer (Specs, Software)
12. Stealing Clipboard (if during the execution of it, there is some content)
13. Making a picture of yourself (if your webcam is connected)
14. Making screenshot of your desktop
15. It could also include a Clipper (as a modular feature).
16. And… due to the module manager, other tasks that I still don't have mentioned there (that also I don't know who they are).

Let's explain just some of them that I found worth to dig into.

Browsers

Since my last analysis, things changed for the browser part and it’s now divided into three major parts.

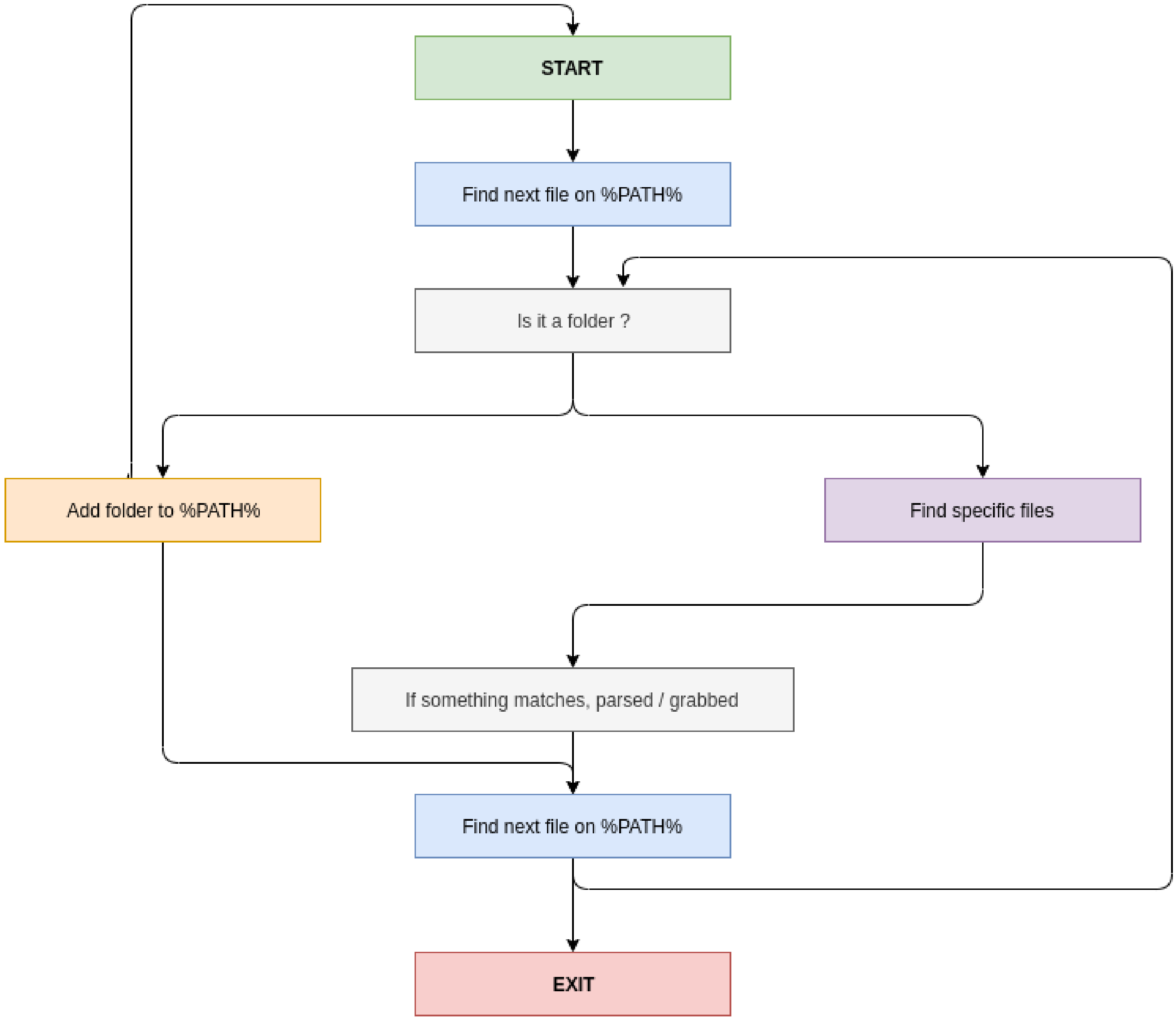
- Internet Explorer is analyzed in a specific function developed due that the data is contained into a “Vault”, so it requires a specific Windows API to read it.
- Microsoft Edge is also split into another part of the stealing process due that this one is using unique files and needs some tasks for the parsing.
- Then, the other browsers are fetched by using a homemade static grabber



Grabber n°1 (The generic one)

It’s pretty fun to see that the stealing process is using at least one single function for catching a lot of things. This generic grabber is pretty “cleaned” based on what I saw before even if there is no magic at all, it’s sufficient to make enough damages by using a recursive loop at a specific place that will search all the required files & folders.

By comparing older versions of predator, when it was attempting to steal content from browsers and some wallets, it was checking step by step specific repositories or registry keys then processing into some loops and tasks for fetching the credentials. Nowadays, this step has been removed (for the browser part) and being part of this raw grabber that will parse everything starting to %USERS% repository.



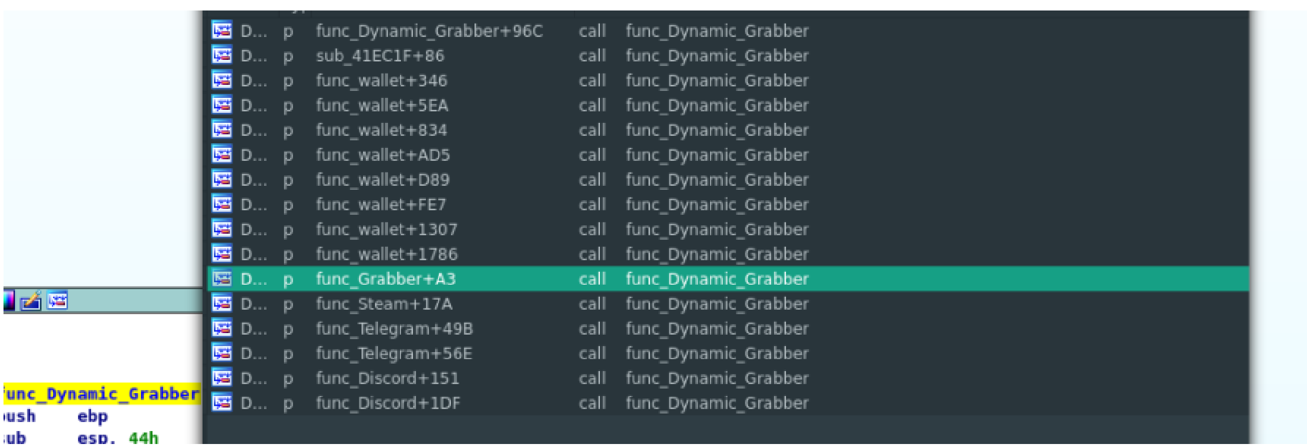
As usual, all the variables that contain required files are obfuscated and encrypted by a simple XOR algorithm and in the end, this is the “static” list that the info stealer will be focused

File grabbed	Type	Actions
Login Data	Chrome / Chromium based	Copy & Parse
Cookies	Chrome / Chromium based	Copy & Parse
Web Data	Browsers	Copy & Parse
History	Browsers	Copy & Parse
formhistory.sqlite	Mozilla Firefox & Others	Copy & Parse
cookies.sqlite	Mozilla Firefox & Others	Copy & Parse

wallet.dat	Bitcoin	Copy & Parse
.sln	Visual Studio Projects	Copy filename into Project.txt
main.db	Skype	Copy & Parse
logins.json	Chrome	Copy & Parse
signons.sqlite	Mozilla Firefox & Others	Copy & Parse
places.sqlite	Mozilla Firefox & Others	Copy & Parse
Last Version	Mozilla Firefox & Others	Copy & Parse

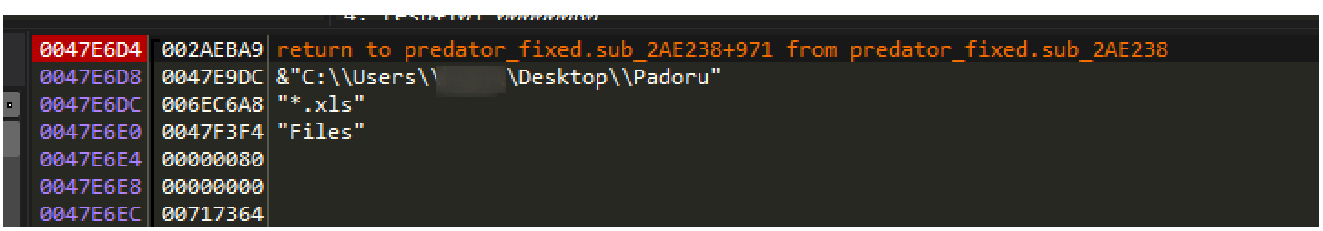
Grabber n°2 (The dynamic one)

There is a second grabber in Predator The Thief, and this not only used when there is available config loaded in memory based on the first request done to the C&C. In fact, it's also used as part of the process of searching & copying critical files coming from wallets software, communication software, and others…



The "main function" of this dynamic grabber only required three arguments:

- The path where you want to search files
- the requested file or mask
- A path where the found files will be put in the final archive sent to the C&C



When the grabber is configured for a recursive search, it's simply adding at the end of the path the value ".." and checking if the next file is a folder to enter again into the same function again and again.

In the end, in the fundamentals, this is almost the same pattern as the first grabber with the only difference that in this case, there are no parsing/analyzing files in an in-depth way. It's simply this follow-up

1. Find a matched file based on the requested search
2. creating an entry on the stolen archive folder
3. setting a handle/pointer from the grabbed file
4. Save the whole content to memory
5. Repeat

Of course, there is a lot of particular cases that are to take in consideration here, but the main idea is like this.

What Predator is stealing in the end?

If we removed the dynamic grabber, this is the current list (for 3.3.2) about what kind of software that is impacted by this stealer, for sure, it's hard to know precisely on the browser all the one that is impacted due to the generic grabber, but in the end, the most important one is listed here.

VPN

NordVPN

Communication

- Jabber
- Discord
- Skype

FTP

- WinSCP
- WinFTP
- FileZilla

Mails

Outlook

2FA Software

- Authy (Inspired by Vidar)

Games

- Steam
- Battle.net (Inspired by Kpot)
- Osu

Wallets

- Electrum
- MultiBit
- Armory
- Ethereum
- Bytecoin
- Bitcoin
- Jaxx
- Atomic
- Exodus

Browser

- Mozilla Firefox (also Gecko browsers using same files)
- Chrome (also Chromium browsers using same files)
- Internet Explorer
- Edge
- Unmentioned browsers using the same files detected by the grabber.

Also beside stealing other actions are performed like:

- Performing a webcam picture capture
- Performing a desktop screenshot

Loader

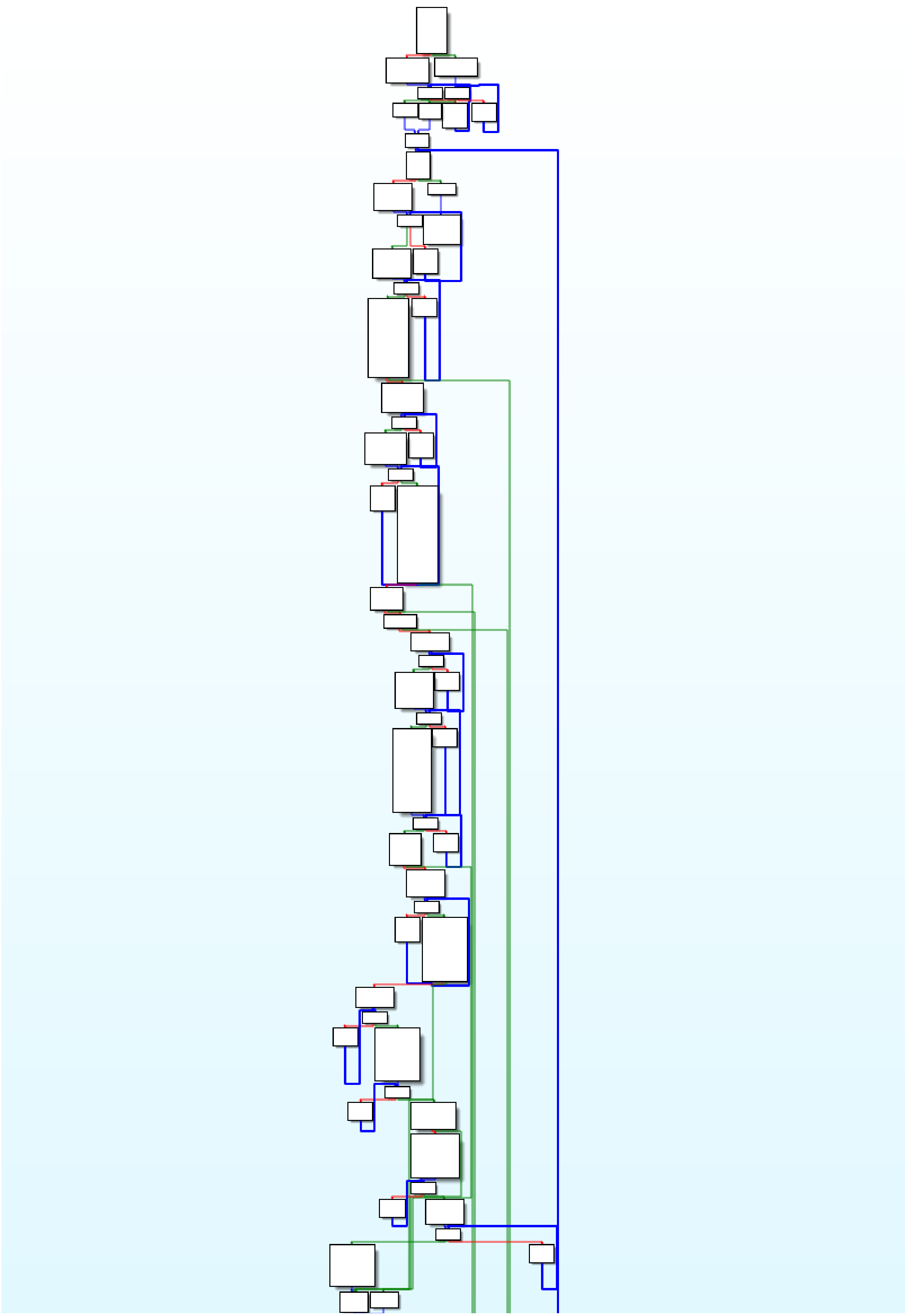
There is currently 4 kind of loader implemented into this info stealer

1. RunPE
2. CreateProcess
3. ShellExecuteA
4. LoadPE
5. LoadLibrary

For all the cases, I have explained below (on another part of this analysis) what are the options of each of the techniques performed. There is no magic, there is nothing to explain more about this feature these days. There are enough articles and tutorials that are talking about this. The only thing to notice is that Predator is designed to load the payload in different ways, just by a simple process creation or abusing some process injections (i recommend on this part, to read the work from endgame).

Module Manager

Something really interesting about this stealer these days, it that it developed a feature for being able to add the additional tasks as part of a module/plugin package. Maybe the name of this thing is wrongly named (i will probably be fixed soon about this statement). But now it’s definitely sure that we can consider this malware as a modular one.



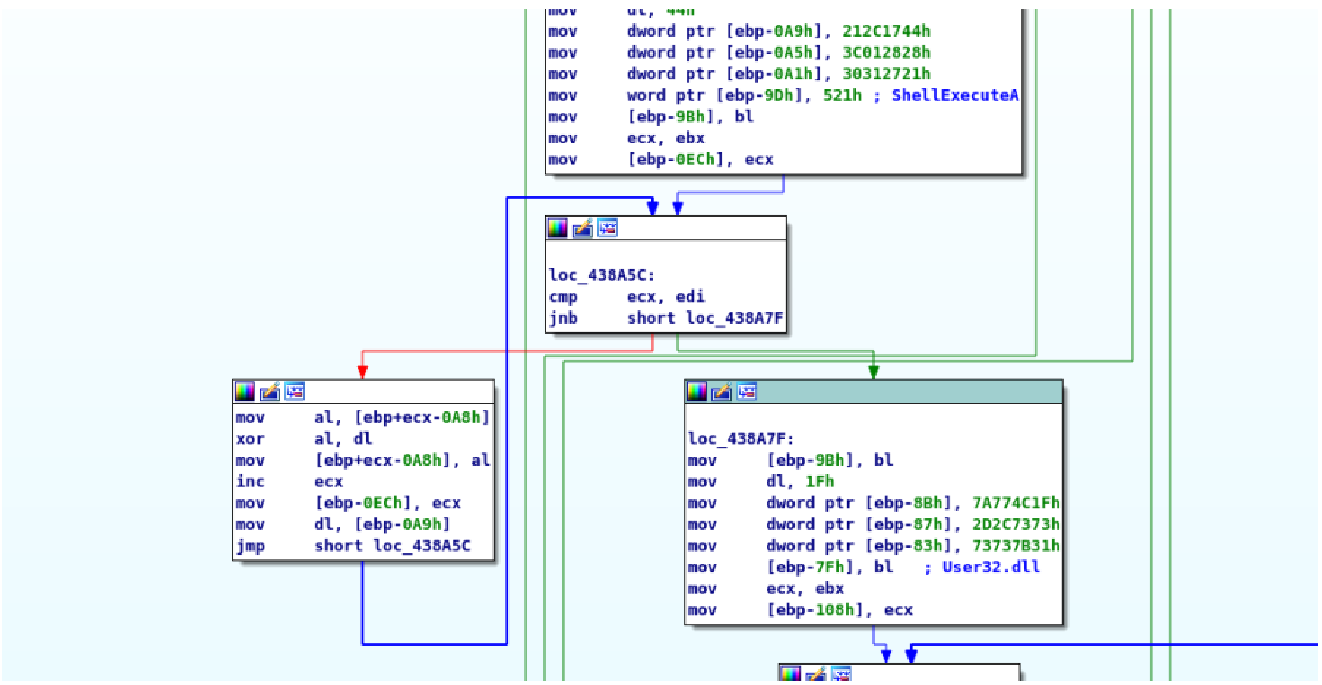
When decrypting the config from check.get, you can understand fast that a module will be launched, by looking at the last entry…

[PREDATOR_CONFIG]#[GRABBER]#[NETWORK_INFO]#[LOADER]#[**example**]

This will be the name of the module that will be requested to the C&C. (this is also the easiest way to spot a new module).

- **example**.get
- **example**.post

The first request is giving you the config of the module (on my case it was like this), it's saved but NOT decrypted (looks like it will be dealt by the module on this part). The other request is focused on downloading the payload, decrypting it and saving it to the disk in a random folder in %PROGRAMDATA% (also the filename is generated also randomly), when it's done, it's simply executed by ShellExecuteA.

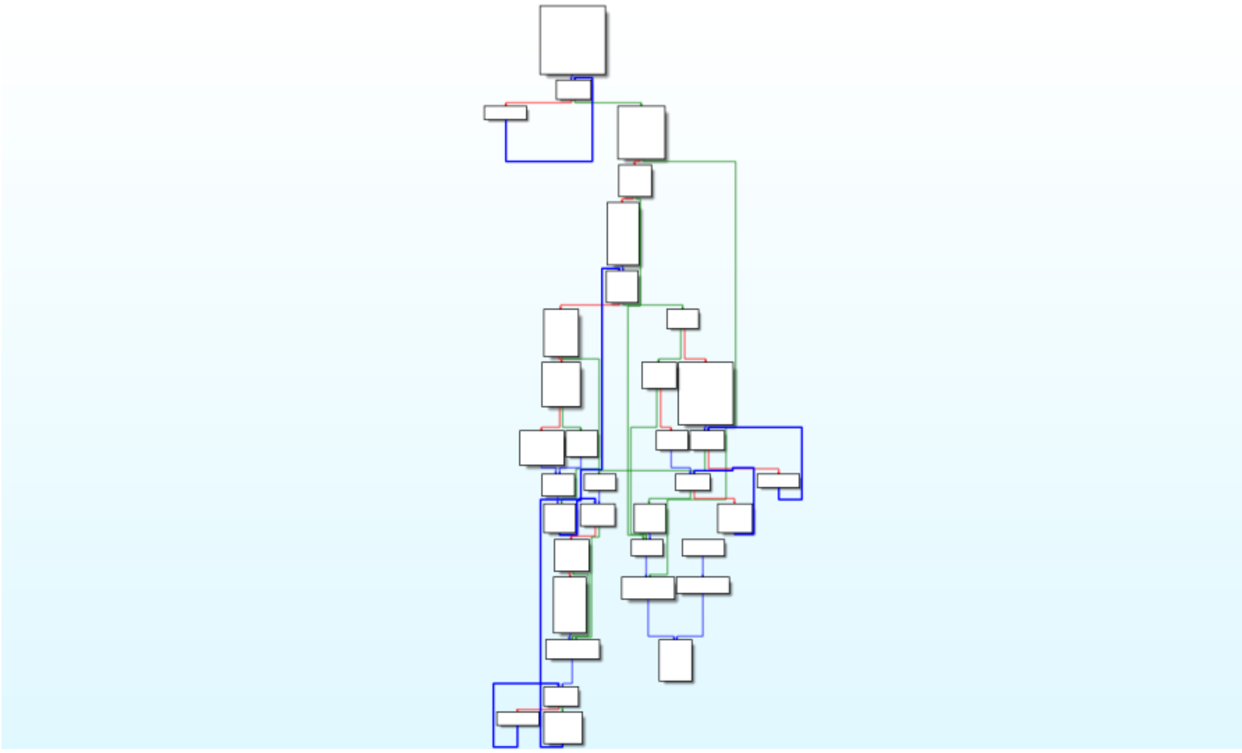


Also, another thing to notice, you know that it's designed to launch multiple modules/plugins.

Clipper (Optional module)

The clipper is one example of the Module that could be loaded by the module manager. As far as I saw, I only see this one (maybe they are other things, maybe not, I don't have the visibility for that).

Disclaimer: Before people will maybe mistaken, the clipper is proper to Predator the Thief and this is NOT something coming from another actor (if it's the case, the loader part would be used).



Clipper WinMain function

This malware module is developed in C++, and like Predator itself, you recognized pretty well the obfuscation proper to it (Stack strings, XOR, SUB, Code spaghetti, GetProcAddress recreated…). Well, everything that you love for slowing down again your analysis.

As detailed already a little above, the module is designed to grab the config from the main program, decrypting it and starting to do the process routine indefinitely:

1. Open Clipboard
2. Checking content based on the config loaded
3. If something matches put the malicious wallet
4. Sleep
5. Repeat

The clipper config is rudimentary using "|" as a delimiter. Mask/Regex on the left, malicious wallet on the right.

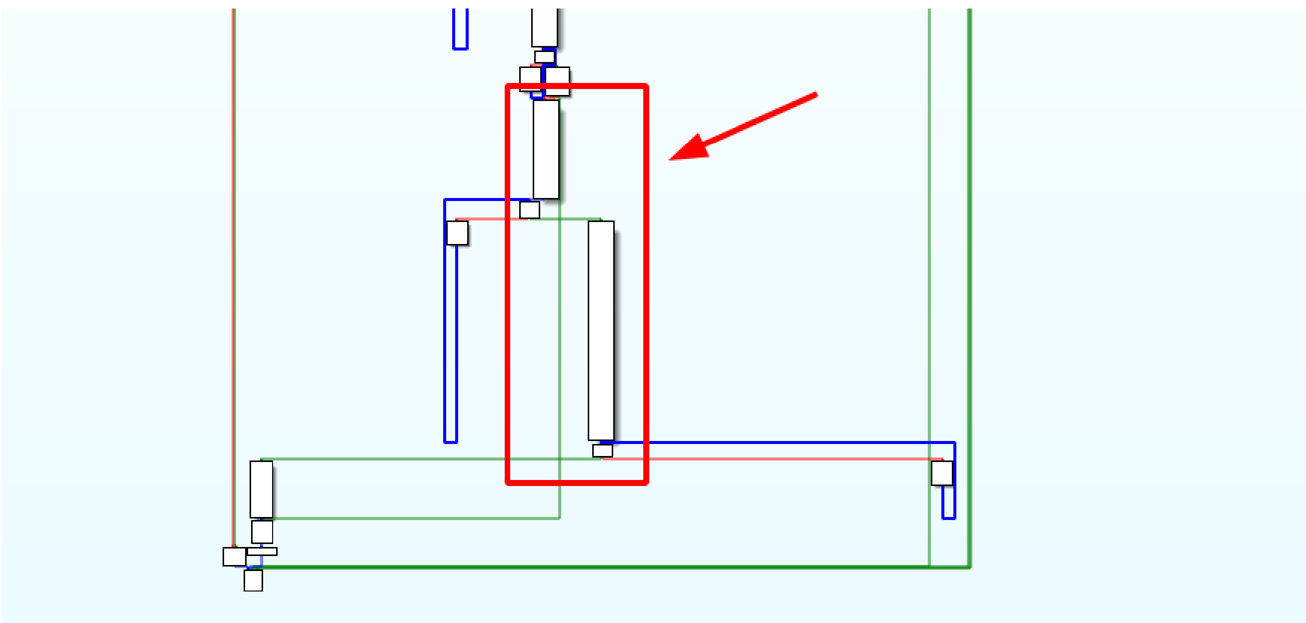
```
1*:1Eh8gHDVCS8xuKQNhCtZKiE1dVuRQiQ58H|
3*:1Eh8gHDVCS8xuKQNhCtZKiE1dVuRQiQ58H|
0x*:0x7996ad65556859C0F795Fe590018b08699092B9C|
q*:qztrpt42h78ks7h6jlgtqtvhp3q6utm7sqrsupgwv0|
G*:GaJvoTcC4Bw3kitxHWU4nrdDK3izXCTmFQ|
X*:XruZmSaEYPX2mH48nGkPSGTzFiPfKXDLWn|
L*:LdPvBrWvimse3WuVNg6pjH15GgBUtSUaWy|
t*:t1dLgBbvV6sXNCMUSS5JeLjF4XhhbJYSDAe|
4*:44tLjmXrQNrWJ5NBsEj2R77ZBEgDa3fEe9GLpSf2FRmhexPvfYDUAB7EXX1Hdb3aMQ9FLqdJ56yaAhiXoRs

D*:DUMKwVVAaMcbtdWipMkXoGfRistK1cC26C|
A*:AaUgfMh5iVkGKLVpMUZW8tGuyjZQNViwDt|
```

There is no communication with the C&C when the clipper is switching wallet, it’s an offline one.

Self Removal

When the parameters are set to 1 in the Predator config got by check.get, the malware is performing a really simple task to erase itself from the machine when all the tasks are done.



By looking at the bottom of the main big function where all the task is performed, you can see two main blocs that could be skipped. these two are huge stack strings that will generate two things.

- the API request “ShellExecuteA”
- The command “ping 127.0.0.1 & del %PATH%”

When all is prepared the thing is simply executed behind the classic register call. By the way, doing a ping request is one of the dozen way to do a sleep call and waiting for a little before performing the deletion.

```
code:0041AACE                push    ebx
code:0041AACF                push    ebx
code:0041AAD0                push    eax
code:0041AAD1                push    dword ptr [ebp-60h]
code:0041AAD4                push    ebx
code:0041AAD5                push    ebx
code:0041AAD6                call    edi             ; kernel32.ShellExecuteA
```

This option is not performed by default when the malware is not able to get data from the C&C.

Telemetry files

There is a bunch of files that are proper to this stealer, which are generated during the whole infection process. Each of them has a specific meaning.

Information.txt

1. Signature of the stealer
2. Stealing statistics
3. Computer specs
4. Number of users in the machine
5. List of logical drives
6. Current usage resources
7. Clipboard content
8. Network info
9. Compile-time of the payload

Also, this generated file is literally "hell" when you want to dig into it by the amount of obfuscated code.

I can quote these following important telemetry files:

Software.txt

- Windows Build Version
- Generated User-Agent
- List of software installed in the machine (checking for x32 and x64 architecture folders)

Actions.txt

List of actions & telemetry performed by the stealer itself during the stealing process

Projects.txt

List of SLN filename found during the grabber research (the static one)

CookeList.txt

List of cookies content fetched/parsed

Network

User-Agent “Builder”

Sometimes features are fun to dig in when I heard about that predator is now generating dynamic user-agent, I was thinking about some things but in fact, it’s way simpler than I thought.

The User-Agent is generated in 5 steps

1. Decrypting a static string that contains the first part of the User-Agent
2. Using GetTickCount and grabbing the last bytes of it for generating a fake builder version of Chrome
3. Decrypting another static string that contains the end of the User-Agent
4. Concat Everything
5. Profit

Tihs User-Agent is shown into the software.txt logfile.

C&C Requests

There is currently 4 kind of request seen in Predator 3.3.2 (it's always a POST request)

Request	*Meaning*
api/check.get	Get dynamic config, tasks and network info
api/gate.get ?……	Send stolen data
api/.get	Get modular dynamic config
api/.post	Get modular dynamic payload (was like this with the clipper)

The first step – Get the config & extra Infos

For the first request, the response from the server is always in a specific form :

- String obviously base64 encoded
- Encrypted using RC4 encryption by using the domain name as the key

When decrypted, the config is pretty easy to guess and also a bit complex (due to the number of options & parameters that the threat actor is able to do).

```
[0;1;0;1;1;0;1;1;0;512;]#
[[%userprofile%\Desktop|%userprofile%\Downloads|%userprofile%\Documents;*.xls,*.xlsx,*
[Trakai;Republic of Lithuania;54.6378;24.9343;85.206.166.82;Europe/Vilnius;21001]#[]#
[Clipper]
```

It's easily understandable that the config is split by the "#" and each data and could be summarized like this

1. The stealer config
2. The grabber config
3. The network config
4. The loader config
5. The dynamic modular config (i.e Clipper)

I have represented each of them into an array with the meaning of each of the parameters (when it was possible).

Predator config

Args	Meaning
Field 1	Webcam screenshot
Field 2	Anti VM
Field 3	Skype

Field 4	Steam
Field 5	Desktop screenshot
Field 6	Anti-CIS
Field 7	Self Destroy
Field 8	Telegram
Field 9	Windows Cookie
Field 10	Max size for files grabbed
Field 11	Powershell script (in base64)

Grabber config

[]#[GRABBER]#[]#[]#[]

Args	Meaning
Field 1	%PATH% using "|" as a delimiter
Field 2	Files to grab
Field 3	Max sized for each file grabbed
Field 4	Whitelist
Field 5	Recursive search (0 – off | 1 – on)

Network info

[]#[]#[NETWORK]#[]#[]

Args	Meaning
Field 1	City
Field 2	Country
Field 3	GPS Coordinate
Field 4	Time Zone
Field 5	Postal Code

Loader config

[]#[]#[]#[LOADER]#[]

Format

[[URL;3;2;;;;1;amazon.com;0;0;1;0;0;5]]

Meaning

1. Loader URL
2. Loader Type
3. Architecture
4. Targeted Countries (“,” as a delimiter)
5. Blacklisted Countries (“,” as a delimiter)
6. Arguments on startup
7. Injected process OR Where it’s saved and executed
8. Pushing loader if the specific domain(s) is(are) seen in the stolen data
9. Pushing loader if wallets are presents
10. Persistence
11. Executing in admin mode
12. Random file generated
13. Repeating execution
14. ???

Loader type (argument 2)

Value	Meaning
1	RunPE
2	CreateProcess
3	ShellExecute
4	LoadPE
5	LoadLibrary

Architecture (argument 3)

Value	Meaning
1	x32 / x64
2	x32 only
3	x64 only

If it’s RunPE (argument 7)

Value	Meaning
1	Attrib.exe
2	Cmd.exe
3	Audiodg.exe

If it’s CreateProcess / ShellExecuteA / LoadLibrary (argument 7)

Value	Meaning
1	%PROGRAMDATA%
2	%TEMP%
3	%APPDATA%

The second step – Sending stolen data

Format

```
/api/gate.get?p1=X&p2=X&p3=X&p4=X&p5=X&p6=X&p7=X&p8=X&p9=X&p10=X
```

Goal

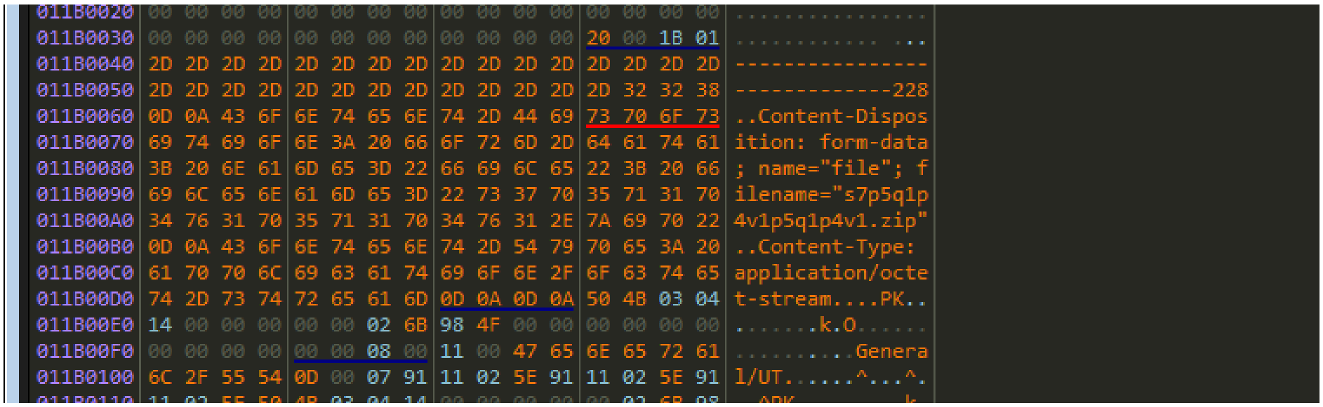
1. Sending stolen data
2. Also victim telemetry

Meaning

Args	Field
p1	Passwords
p2	Cookies
p3	Credit Cards
p4	Forms
p5	Steam
p6	Wallets
p7	Telegram

p8	???
p9	???
p10	OS Version (encrypted + encoded)*

This is an example of crafted request performed by Predator the thief



Third step – Modular tasks (optional)

`/api/Clipper.get`

Give the dynamic clipper config

`/api/Clipper.post`

Give the predator clipper payload

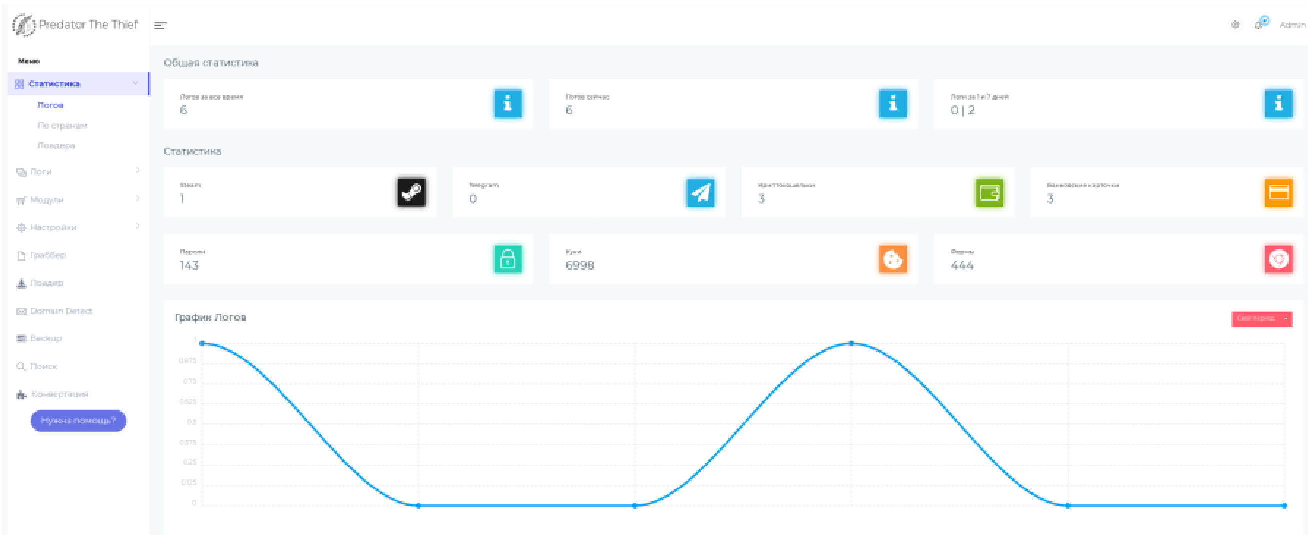
Server side

The C&C is nowadays way different than the beginning, it has been reworked with some fancy designed and being able to do some stuff:

1. Modulable C&C
2. Classic fancy index with statistics
3. Possibility to configure your panel itself
4. Dynamic grabber configuration
5. Telegram notifications
6. Backups
7. Tags for specific domains

Index

The predator panel changed a lot between the v2 and v3. This is currently a fancy theme one, and you can easily spot the whole statistics at first glance. the thing to notice is that the panel is fully in Russian (and I don’t know at that time if there is an English one).



Menu on the left is divide like this (but I’m not really sure about the correct translation)

Меню (Menu)
Статистика (Stats)

- Логов (Logs)
- По странам (Country stats)
- Лоадера (Loader Stats)

Логи (Logs)

Обычная

Модули (Modules)

Загрузить модуль (Download/Upload Module)

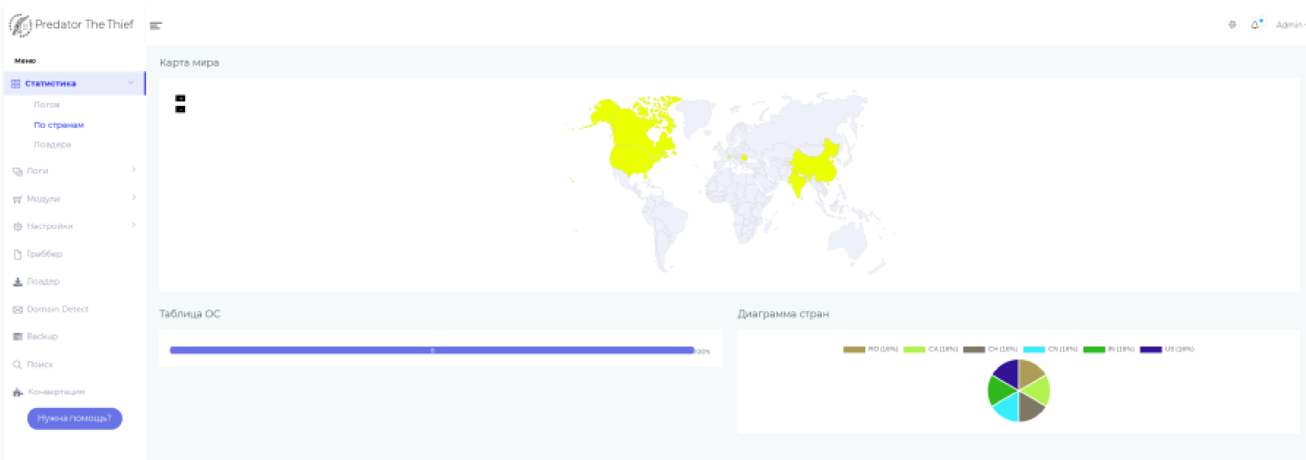
Настройки (Settings)

- Настройки сайта (Site settings)
- Телеграм бот (Telegram Bot)
- Конфиг (Config)

Граббер (Grabber)
Лоадер (Loader)
Domain Detect

Backup
Поиск (Search)
Конвертация (Converter => Netscape Json converter)

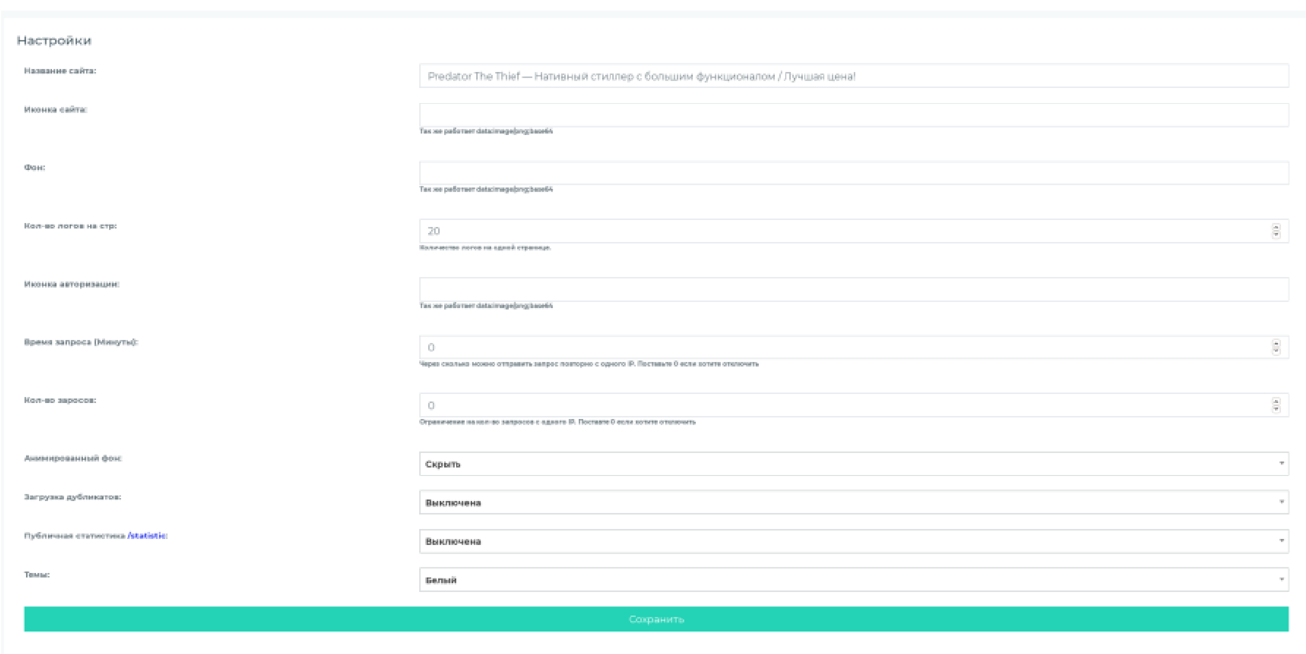
Statistics / Landscape



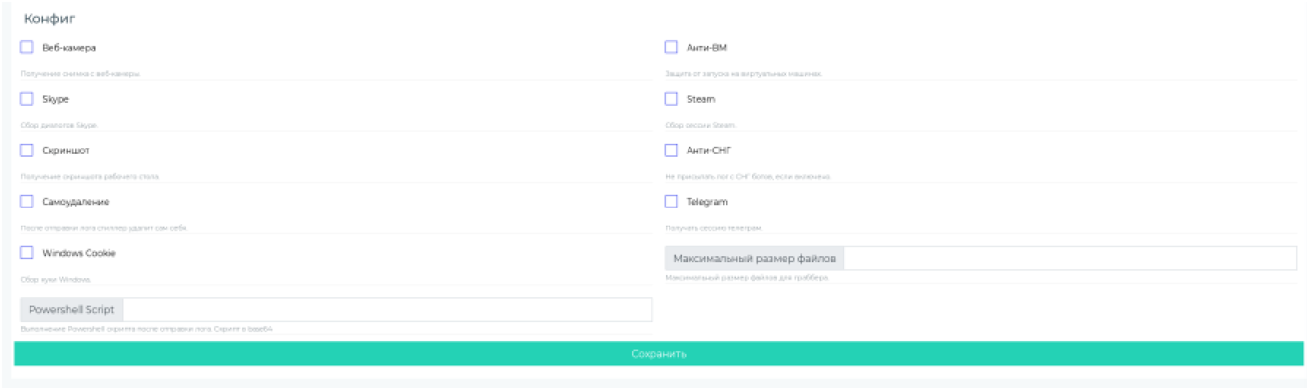
Predator Config

In term of configuring predator, the choices are pretty wild:

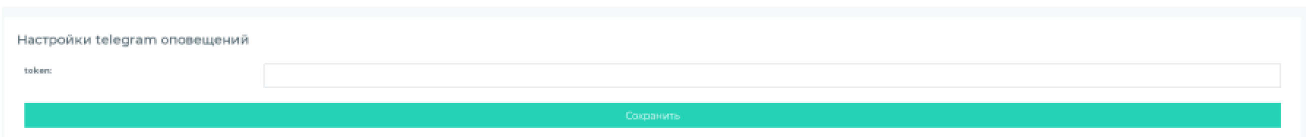
> The actor is able to tweak its panel, by modifying some details, like the title and detail that made me laugh is you can choose a dark theme.



> There is also another form, the payload config is configured by just ticking options. When done, this will update the request coming from check.get

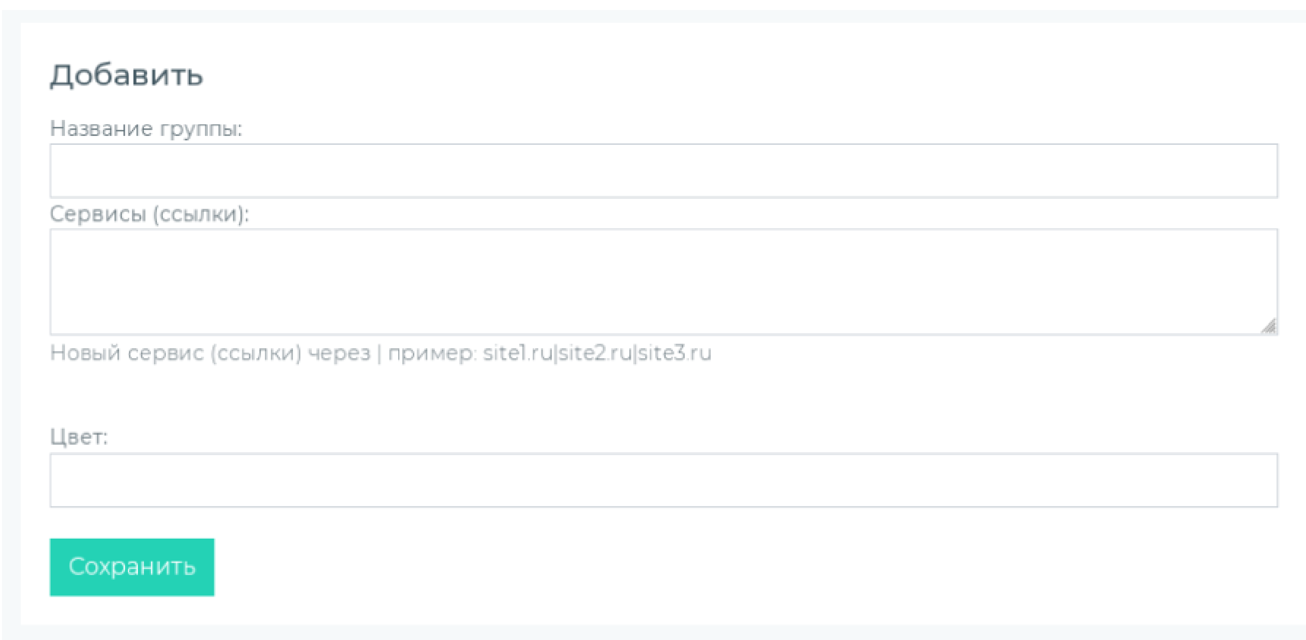


As usual, there is also a telegram bot feature



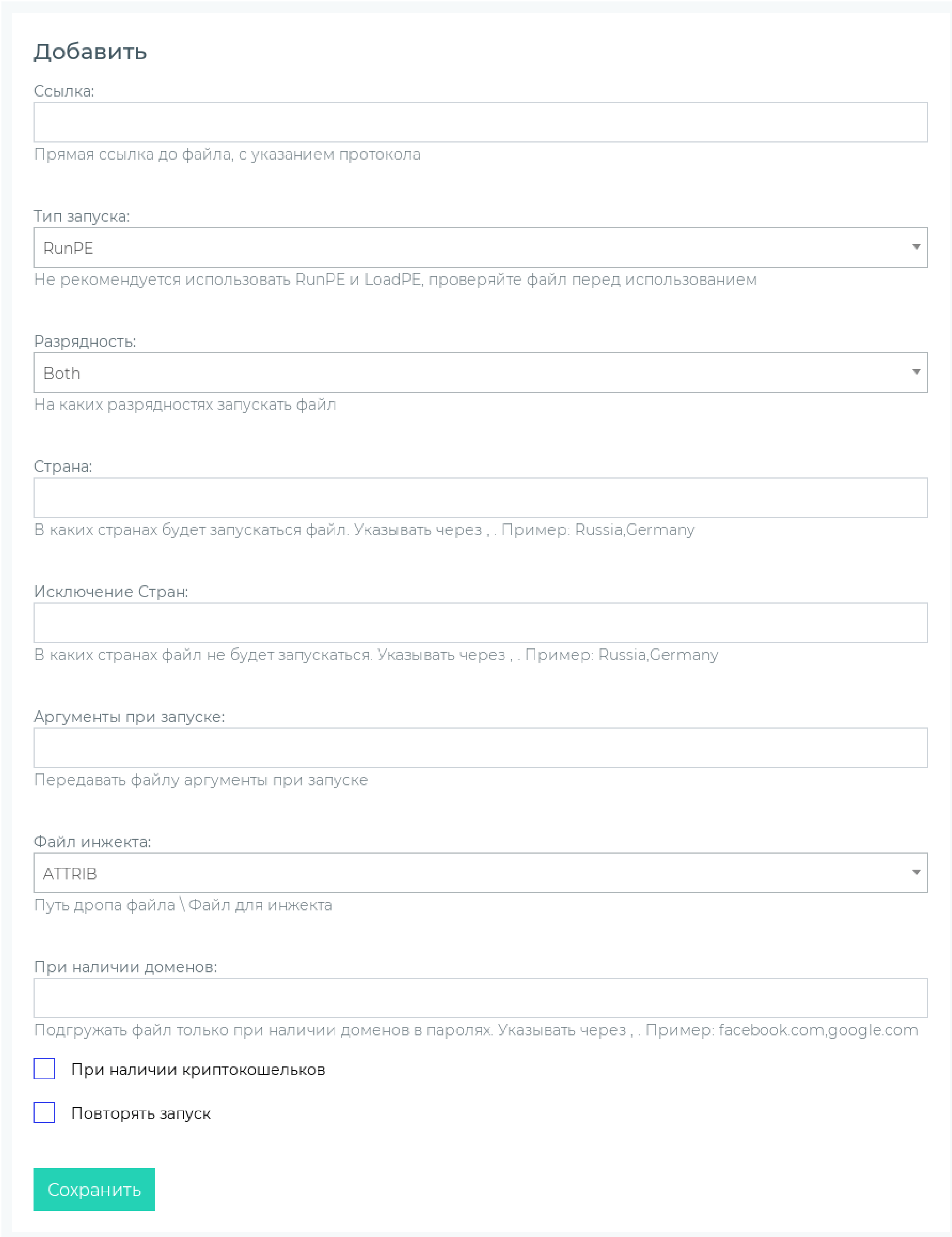
Creating Tags for domains seen

Small details which were also mentioned in Vidar, but if the actor wants specific attention for bots that have data coming from specific domains, it will create a tag that will help him to filter easily which of them is probably worth to dig into.



Loader config

The loader configuration is by far really interesting in my point of view and even it has been explained totally for its functionalities, I considered it pretty complete and user-friendly for the Threat Actor that is using it.



IoCs

Hashes for this analysis

p_pckd.exe – 21ebdc3a58f3d346247b2893d41c80126edabb060759af846273f9c9d0c92a9a
p_upkd.exe – 6e27a2b223ef076d952aaa7c69725c831997898bebcd2d99654f4a1aa3358619
p_clipper.exe – 01ef26b464faf08081fceeeb2cdff7a66ffdbd31072fe47b4eb43c219da287e8

C&C

cadvexmail19mn.world

Other predator hashes

- 9110e59b6c7ced21e194d37bb4fc14b2
- 51e1924ac4c3f87553e9e9c712348ac8
- fe6125adb3cc69aa8c97ab31a0e7f5f8
- 02484e00e248da80c897e2261e65d275
- a86f18fa2d67415ac2d576e1cd5ccad8
- 3861a092245655330f0f1ffec75aca67
- ed3893c96decc3aa798be93192413d28

Conclusion

Infostealer is not considered as harmful as recent highly mediatize ransomware attacks, but they are enough effective to perform severe damage and they should not be underrated, furthermore, with the use of cryptocurrencies that are more and more common, or something totally normal nowadays, the lack of security hygiene on this subject is awfully insane. that I am not surprised at all to see so much money stolen, so they will be still really active, it's always interesting to keep an eye on this malware family (and also on clippers), whenever there is a new wallet software or trading cryptocurrency software on the list, you know easily what are the possible trends (if you have a lack of knowledge in that area).

Nowadays, it's easy to see fresh activities in the wild for this info stealer, it could be dropped by important malware campaigns where notorious malware like ISFB Gozi is also used. It's unnecessary (on my side) to speculate about what will be next move with Predator, I have clearly no idea and not interested in that kind of stuff. The thing is the malware scene nowadays is evolving really fast, threat actor teams are moving/switching easily and it could take only hours for new updates and rework of malware by just modifying a piece of code with something already developed on some GitHub repository, or copying code from another malware. Also, the price of the malware has been adjusted, or the support communication is moved to something else.

Due to this, I am pretty sure at that time, this current in-depth analysis could be already outdated by some modifications. it's always a risk to take and on my side, I am only interested in the malware itself, the main ideas/facts of the major version are explained and it's plenty sufficient. There is, of course, some topics that I haven't talk like nowadays predator is now being to work as a classic executable file or a DLL, but it was developed some times ago and this subject is now a bit popular. Also, another point that I didn't find any explanation, is that seeing some decrypting process for strings that leads to some encryption algorithm related to Tor.

This in-depth analysis is also focused on showing that even simple tricks are an efficient way to slow down analysis and it is a good exercise to practice your skills if you want to improve yourself into malware analysis. Also, reverse engineering is not as hard as people could think when the fundamental concepts are assimilated, It's just time, practice and motivation.

On my side, I am, as usual, typically irregular into releasing stuff due to some stuff (again…). By the way, updating projects are still one of my main focus, I still have some things that I would love to finish which are not necessarily into malware analysis, it's cool to change topics sometimes.

#HappyHunting