Analyzing the Cerber Ransomware

April 18, 2025

Kollin Labowski

klabowski@ufl.edu

Malware Reverse Engineering: Practical 4

Section 1: Executive Summary

This report will analyze the ransomware sample cerber.exe. This malware, part of the Cerber (https://www.proofpoint.com/us/threat-reference/cerber-ransomware) family of ransomware, is known to be distributed as a ransomware-as-a-service. As is typical for ransomware, cerber.exe makes no attempt to hide itself from the user. Instead, it changes the desktop background, and creates two README files (one of type .txt and another of type .hta) explaining to the user that their files have been encrypted and instructing them to pay a ransom. The source code of the malware is packed using what appears to be a sophisticated custom packing scheme.

The malware's behavior is customizable via a JSON configuration files that is encrypted inside the file. Some customizable parameters include the public key for key exchange, blacklisted/whitelisted files, file extensions, folders, languages, subnets to search for (likely for a C2 server), ransom messages, and more. As configured currently, the malware seaches the subnets 178.33.158.0/27, 178.33.159.0/27 and 178.33.160.0/22. Four cryptocurrency domains are also accessed via embedded JavaScript in the .hta file.

The malware can be detected by looking for changes to the desktop background, new README files on the desktop and other directories, and communication to IP addresses in the previously mentioned subnets.

Link to presentation: https://uflorida-my.sharepoint.com/personal/klabowski_ufl_edu/_layouts/15/stream.aspx?id=%2Fpersonal%2Fklabowski%5Fufl%5Fedu%2FDocuments%2FDesktop%2FPMA%20Practical%204%20Presentation%2Emp4&nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0&ga=1&referrer=StreamWebApp%2EWeb&referrerScenario=AddressBarCopied%2Eview%2Edcd0cec3%2Defe9%2D4ff8%2D9cb8%2D522f9a13ec00

Section 2: Static Analysis

Section 2A: Basic static analysis of cerber.exe

property	value
md5	8B6BC16FD137C09A08B02BBE1BB7D670
sha1	C69A0F6C6F809C01DB92CA658FCF1B643391A2B7
sha256	E67834D1E8B38EC5864CFA101B140AEABA8F1900A6E269E6A94C90FCBFE56678
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z . . . . . @ . . . . .
file-size	619008 (bytes)
size-without-overlay	n/a
entropy	5.092
imphash	9D6ED8D049BC10BC45B1995CB6F7F4B6
signature	n/a
entry-point	55 8B EC 83 EC 64 FF 15 DC 03 45 00 8B 55 08 89 15 7C C1 48 00 89 2D 5C C1 48 00 C6 45 A3 00 C7 45
file-version	n/a
description	n/a
file-type	**executable**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x5925F1A2 (Wed May 24 16:48:34 2017)
debugger-stamp	n/a
resources-stamp	
exports-stamp	n/a
version-stamp	empty
certificate-stamp	n/a

Figure 1: The output of PEStudio for cerber.exe

We can see that cerber.exe was compiled on May 24, 2017, for 32-bit systems. It has GUI capabilities. Its hashes are 8b6bC16fd137c09a08b02bbe1bb7d670 (MD5), c69a0f6c6f809c01db92ca658fcf1b643391a2b7 (SHA-1), and e67834d1e8b38ec5864cfa101b140aeaba8f1900a6e269e6a94c90fcbfe56678 (SHA-256). The entropy of the file is 5.092, which is quite low.

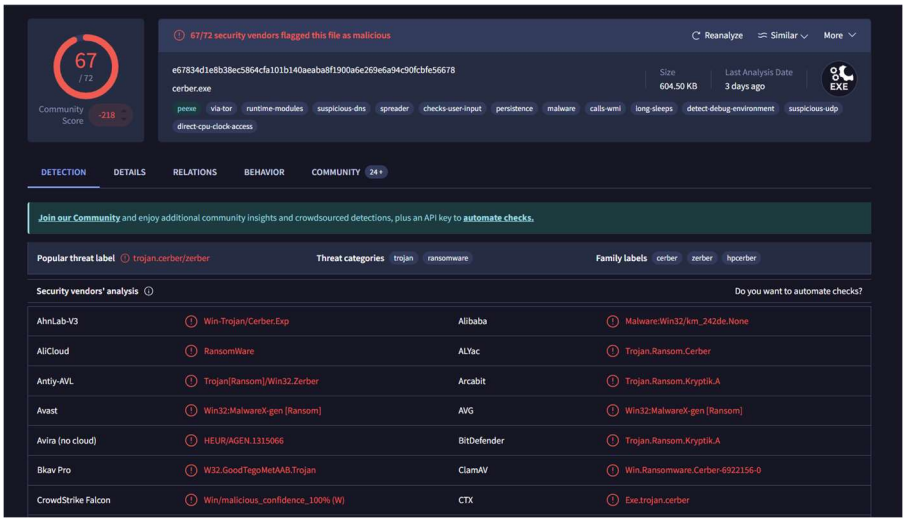


Figure 2: The output of VirusTotal for cerber.exe

Uploading the hash to VirusTotal, we can see that, unsurprisingly, the malware has been flagged as malicious by 67/72 vendors.

property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	AB79238DB7074FEBAB12699...	059D8D7B7A9B5D497836734...	DC35DD1C8E7FF8BBFBFD5B...	3CF29FE934D205800709B2A...
entropy	6.093	1.344	4.976	6.553
file-ratio (99.83%)	52.19 %	38.79 %	0.74 %	8.11 %
raw-address	0x00000400	0x0004F200	0x00089C00	0x0008AE00
raw-size (617984 bytes)	0x0004EE00 (323072 bytes)	0x0003AA00 (240128 bytes)	0x00001200 (4608 bytes)	0x0000C400 (50176 bytes)
virtual-address	0x00401000	0x00450000	0x0048B000	0x0048D000
virtual-size (617296 bytes)	0x0004ED6E (322926 bytes)	0x0003A87A (239738 bytes)	0x000011C0 (4544 bytes)	0x0000C3A8 (50088 bytes)
entry-point	0x0004F4E0	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-
discardable	-	-	-	-
initialized-data	-	x	x	x
uninitialized-data	-	-	-	-
unreadable	-	-	-	-
self-modifying	-	-	-	-
virtualized	-	-	-	-
file	n/a	n/a	n/a	n/a

Figure 3: The sections tab in PEStudio for cerber.exe

We can see that cerber.exe has four sections: .text, .rdata, .data, and .rsrc. The .text section contains the code for the main program. The .rdata section contains read-only constants. The .data section contains global variables that are assigned before compilation. The .rsrc section contains objects such as images and other self-contained files used by the program. The entropy of all sections is normal, as is the virtual-to-raw size ratio for the sections. We will see in a moment that this program also imports many functions and contains many blacklisted strings. Therefore, all traditional signs indicate the program is not packed. However, we will see later that the program actually does use a sophisticated packing scheme, and we will discuss it in detail.

name (317)	group (17)	type (1)	ordinal (0)	blacklist (85)	anti-debug (0)	undocumented (0)	deprecated (23)	library (8)
MonitorFromWindow	windowing	implicit	-	x	-	-	-	user32.dll
IsSystemResumeAutoma...	system-information	implicit	-	x	-	-	-	kernel32.dll
SetSystemTime	system-information	implicit	-	x	-	-	-	kernel32.dll
QueueUserAPC	synchronization	implicit	-	x	-	-	-	kernel32.dll
GetVolumeInformationW	storage	implicit	-	x	-	-	-	kernel32.dll
SearchPathW	storage	implicit	-	x	-	-	-	kernel32.dll
SetCurrentDirectoryW	storage	implicit	-	x	-	-	-	kernel32.dll
SetVolumeLabelA	storage	implicit	-	x	-	-	-	kernel32.dll
WinHelpA	shell	implicit	-	x	-	-	-	user32.dll
SHChangeNotify	shell	implicit	-	x	-	-	-	shell32.dll
SHEmptyRecycleBinA	shell	implicit	-	x	-	-	-	shell32.dll
SaferIdentifyLevel	security	implicit	-	x	-	-	-	advapi32.dll
SaferComputeTokenFro...	security	implicit	-	x	-	-	-	advapi32.dll
RevertToSelf	security	implicit	-	x	-	-	-	advapi32.dll
LookupAccountSidW	security	implicit	-	x	-	-	-	advapi32.dll
ImpersonateLoggedOnU...	security	implicit	-	x	-	-	-	advapi32.dll
GetSecurityDescriptorOw...	security	implicit	-	x	-	-	-	advapi32.dll
FreeSid	security	implicit	-	x	-	-	-	advapi32.dll
WritePrivateProfileSectio...	registry	implicit	-	x	-	-	x	kernel32.dll
RegSetValueW	registry	implicit	-	x	-	-	x	advapi32.dll

Figure 4: The import tab of PEStudio for cerber.exe

We can see that cerber.exe imports 317 functions, 85 of which are labeled as blacklisted by PEStudio. Here are a few of the most interesting:

- **SetCurrentDirectoryW:** This can be used by the malware to traverse the file system and encrypt files in each directory.
- **LookupAccountSidW:** This may be used to generate a unique identifier for the infected machine so the correct decryption key can be supplied if the ransom is paid.
- **SHEmptyRecycleBinA:** This can be used to empty all items in the recycle bin so the user cannot recover any deleted files.
- **RegSetValueW (and related functions):** This indicates that the malware will modify the registry, probably for maintaining persistence.
- **FindFirstFileW (and related functions):** The sequence of FindFirstFile and FindNextFile is commonly used in ransomware to encrypt all files in a directory.
- **SleepEx:** The program may perform certain actions at set time intervals.
- **OpenProcess (and related functions):** This suggests the malware may be creating new processes, possibly splitting encryption of different directories into separate processes.
- **ShellExecuteA (and related functions):** This could be used to execute shell commands, and possibly start more processes.

type (2)	size (bytes)	file-offset	blacklist (85)	hint (12)	group (19)	value (2976)
ascii	26	0x000893E4	x	-	security	GetSecurityDescriptorOwner
ascii	26	0x000894F6	x	-	security	SaferComputeTokenFromLevel
ascii	26	0x00088F63	x	-	registry	WritePrivateProfileSection
ascii	26	0x00088678	x	-	console	FillConsoleOutputAttribute
ascii	26	0x00088697	x	-	console	FillConsoleOutputCharacter
ascii	26	0x000887D4	x	-	console	GetConsoleScreenBufferInfo
ascii	24	0x0008952A	x	-	diagnostic	SaferRecordEventLogEntry
ascii	24	0x000890C5	x	-	desktop	GetUserObjectInformation
ascii	24	0x00089195	x	-	desktop	SetUserObjectInformation
ascii	24	0x00088D1E	x	-	console	SetConsoleCursorPosition
ascii	23	0x00088BB0	x	-	system-information	IsSystemResumeAutomatic
ascii	23	0x00089402	x	-	security	ImpersonateLoggedOnUser
ascii	23	0x00089072	x	-	desktop	GetProcessWindowStation
ascii	23	0x00088704	x	-	console	FlushConsoleInputBuffer
ascii	22	0x000888C5	x	-	execution	GetEnvironmentVariable
ascii	22	0x00088D93	x	-	execution	SetEnvironmentVariable
ascii	21	0x00088793	x	-	file	GetCompressedFileSize
ascii	21	0x000888AB	x	-	execution	GetEnvironmentStrings
ascii	21	0x00088D06	x	-	console	SetConsoleCtrlHandler
ascii	20	0x00088AF1	x	-	storage	GetVolumeInformation
ascii	19	0x00088D7B	x	-	storage	SetCurrentDirectory
ascii	19	0x00088832	x	-	execution	GetCurrentProcessId
ascii	19	0x000893AF	x	-	execution	CreateProcessAsUser
ascii	19	0x00088FED	x	-	desktop	CreateWindowStation
ascii	18	0x00089514	x	-	security	SaferIdentifyLevel
ascii	18	0x0008885C	x	-	execution	GetCurrentThreadId
ascii	18	0x000888DE	x	-	execution	GetExitCodeProcess
ascii	18	0x00088FC8	x	-	desktop	CloseWindowStation
ascii	17	0x0008913C	x	-	windowing	MonitorFromWindow
ascii	17	0x00089613	x	-	shell	SHEmptyRecycleBin
ascii	17	0x00088CB2	x	-	memory	ReadProcessMemory
ascii	17	0x00088DBD	x	-	file	SetFileAttributes
ascii	17	0x00089655	x	-	file	SHBrowseForFolder
ascii	17	0x00088985	x	-	dynamic-library	GetModuleFileName

Figure 5: The strings tab of PEStudio for cerber.exe

PEStudio identified nearly 3,000 strings in the malware sample, 85 of which were labeled as blacklisted (these strings appear to be the names of the 85 blacklisted imports). Here are some of the interesting strings identified:

- **"Elaborate Bytes AG":** This seems to refer to a company that sells software to allow for DVDs and other storage mediums to be cloned (https://www.elby.ch/en/). In the version tab of PEStudio, this is listed as the company name, indicating this malware may be disguising itself as a legitimate product from this company.
- **The manifest XML:** The manifest XML was identified in strings (but not in the manifest section) by PEStudio. The main interesting part of this manifest is that the name is listed as "Siber.Systems.roboform", which appears to reference this password manager (https://www.roboform.com/). Perhaps this is another method the malware is using to appear legitimate.

Section 2B: Decompiling cerber.exe

In this section, we will decompile cerber.exe using Ghidra to take a look at the actual code running.

```
11  GetKBCodePage();
12  loaded_dll_location = location_for_libraries;
13  ebp_reference = &stack0xfffffffc;
14  get_min(0x20,0x6f);
15  reg_sub_key = 0x69;
16  _DAT_0048c0ca = 0x6e;
17  _DAT_0048c0cc = 0x74;
18  _DAT_0048c0ce = 0x65;
19  _DAT_0048c0d0 = 0x72;
20  _DAT_0048c0d2 = 0x66;
21  _DAT_0048c0d8 = 0x65;
22  _DAT_0048c0da = 0x5c;
23  _DAT_0048c0dc = 0x7b;
24  _DAT_0048c126 = 0x7d;
25  _DAT_0048c0ee = 0x2d;
26  _DAT_0048c0f8 = 0x2d;
27  _DAT_0048c102 = 0x2d;
28  _DAT_0048c10c = 0x2d;
29  _DAT_0048c0d4 = 0x61;
30  RegOpenKeyExW = RegOpenKeyExW_exref;
31  if (true) {
32    do {
33      reg_key_handle = reg_key_handle + -2;
34      other_shellcode_base_address =
35           (*RegOpenKeyExW)(reg_key_handle,&reg_sub_key,0,0x20019,&output_reg_handle);
36      if (other_shellcode_base_address == 0) break;
37    } while (true);
38  }
```

Figure 6: The top of the entry function for cerber.exe

The beginning of the entry function of cerber.exe is shown above. The code, as decompiled by Ghidra, appears very strange. For example, we can see the Windows API function GetKBCodePage being called, but its output doesn't appear to be used anywhere. (In fact, even in the assembly view, we can see that this function's output is not used anywhere, meaning it might just be here to make analysis more confusing.)

```
                    get_min                                          X

0044f1b0 55             PUSH    EBP
0044f1b1 8b ec          MOV     EBP, ESP
0044f1b3 83 ec 0c       SUB     ESP, 0xc
0044f1b6 8b 45 08       MOV     EAX, dword ptr [EBP + val_1]
0044f1b9 89 45 fc       MOV     dword ptr [EBP + local_8], EAX
0044f1bc 8b 4d 0c       MOV     ECX, dword ptr [EBP + val_2]
0044f1bf 89 4d f4       MOV     dword ptr [EBP + local_10], ECX
0044f1c2 c7 45 f8 ...   MOV     dword ptr [EBP + local_c], 0xb4fd
0044f1c9 8b 55 fc       MOV     EDX, dword ptr [EBP + local_8]
0044f1cc 3b 55 f4       CMP     EDX, dword ptr [EBP + local_10]
0044f1cf 73 07          JNC     LAB_0044f1d8
0044f1d1 8b 45 fc       MOV     EAX, dword ptr [EBP + local_8]
0044f1d4 eb 05          JMP     LAB_0044f1db
```

Figure 7: The disassembly of the get_min function

We also see the first function call to a function we have labeled as “get_min”. It takes two integers, and returns the minimum of the two. The decompiled view is very simple, but the assembly shows some values stored in memory addresses, including a constant number 0xb4fd. It is unclear at this point if this number is being used somewhere else, or if this is just a side effect of the compilation process.

```
0044f512 c7 45 a4 ...   MOV     dword ptr [EBP + local_60], 0x0
0044f519 0f b6 45 a3    MOVZX   EAX, byte ptr [EBP + local_61]
0044f51d 83 c0 48       ADD     EAX, 0x48
0044f520 8b 4d a4       MOV     ECX, dword ptr [EBP + local_60]
0044f523 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f526 66 89 04 4a    MOV     word ptr [EDX + ECX*0x2]=>reg_sub_key, AX
0044f52a 8b 45 9c       MOV     EAX, dword ptr [EBP + local_68]
0044f52d 0f b7 08       MOVZX   ECX, word ptr [EAX]=>reg_sub_key
0044f530 83 c1 21       ADD     ECX, 0x21
0044f533 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f536 66 89 0a       MOV     word ptr [EDX]=>reg_sub_key, CX
0044f539 0f b6 45 a3    MOVZX   EAX, byte ptr [EBP + local_61]
0044f53d 83 c0 6e       ADD     EAX, 0x6e
0044f540 8b 4d a4       MOV     ECX, dword ptr [EBP + local_60]
0044f543 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f546 66 89 44 ...   MOV     word ptr [EDX + ECX*0x2 + 0x2]=>DAT_0048c0ca, AX
0044f54b 0f b6 45 a3    MOVZX   EAX, byte ptr [EBP + local_61]
0044f54f 83 c0 74       ADD     EAX, 0x74
0044f552 8b 4d a4       MOV     ECX, dword ptr [EBP + local_60]
0044f555 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f558 66 89 44 ...   MOV     word ptr [EDX + ECX*0x2 + 0x4]=>DAT_0048c0cc, AX
0044f55d 0f b6 45 a3    MOVZX   EAX, byte ptr [EBP + local_61]
0044f561 83 c0 2d       ADD     EAX, 0x2d
0044f564 8b 4d a4       MOV     ECX, dword ptr [EBP + local_60]
0044f567 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f56a 66 89 44 ...   MOV     word ptr [EDX + ECX*0x2 + 0x3a]=>DAT_0048c102,...
0044f56f 0f b6 45 a3    MOVZX   EAX, byte ptr [EBP + local_61]
0044f573 83 c0 65       ADD     EAX, 0x65
0044f576 8b 4d a4       MOV     ECX, dword ptr [EBP + local_60]
0044f579 8b 55 9c       MOV     EDX, dword ptr [EBP + local_68]
0044f57c 66 89 44 ...   MOV     word ptr [EDX + ECX*0x2 + 0x6]=>DAT_0048c0ce, AX
```

Figure 8: The assembly view of lines 16-29 from Figure 6

In *Figure 6*, we saw that Ghidra identified 14 constants being set to specific values. Looking at the assembly view, we can see that there is more going on. Values are being loaded into the ECX and EDX registers, and these are being used to store bytes as specific memory locations. We will see later that this code is decoding a string stored in memory.

We also saw in *Figure 6* that the RegOpenKeyExW function call was not directly identified by Ghidra. Instead, a variable was created that was used to store a reference to this function, and the reference was used to call the function. We will notice a similar technique used throughout this program.

```
0044f651 c7 45 a8 ...   MOV     dword ptr [EBP + local_5c], 0x2
0044f658 83 7d a8 00    CMP     dword ptr [EBP + local_5c], 0x0
0044f65c 74 58          JZ      LAB_0044f6b6
```

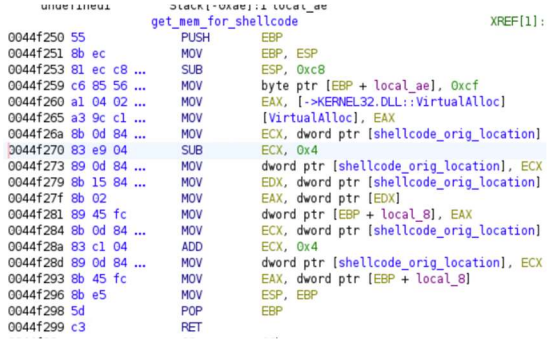
Figure 9: A conditional jump that is never taken

At line 31 in *Figure 6*, we saw that Ghidra identified an if statement that is always entered. In the above screenshot, we show the specific assembly instructions corresponding to this conditional. The value 0 is compared with the value 2, and since these are different numbers, the conditional jump will never be taken, meaning the Ghidra analysis was accurate. This is an odd behavior that was possibly added by the malware authors to make analysis more confusing, or perhaps it is a compiler artifact.

```
23  RegQueryValueExW = RegQueryValueExW_exref;
24  local_29 = 0x2b;
25  cursor_handle = LoadCursorW((HINSTANCE)0x0,(LPCWSTR)0x1402);
26  if (cursor_handle == (HCURSOR)0x0) {
27    buf_size = 300;
28    data_type[0] = 1;
29    local_10c = &DAT_0048c190;
30    module_handle = GetModuleHandleW((LPCWSTR)0x0);
31    _DAT_0048c1b0 = module_handle[0xf].unused;
32    while (true) {
33      reg_key_handle = output_reg_handle;
34      error_code = (*RegQueryValueExW)(output_reg_handle,&registry_value,0,data_type,reg_data,
35                                       &buf_size);
36      if (error_code == 0) {
37        if (local_368 == 0x6b) {
38          return (undefined *)0x0;
39        }
40        if (local_36a == 0x69) {
41          if (false) {
42            return (undefined *)0x0;
43          }
44          if (local_366 == 0x71) {
45            local_10 = &DAT_0040ff27;
46          }
47          if (local_366 == 0x70) {
48            local_10 = &DAT_004017d1;
49          }
50          if (local_362 == 0x74) {
51            local_10 = local_10 + 0x1bc;
52          }
```

Figure 10: The beginning of a function that gets values from the registry

Immediately after the code segment in *Figure 6* is called, a function stored at 0x44f2a0 is called. The beginning of this function is shown above. The function LoadCursorW is called with lpCursorName set to 0x1402. This doesn't appear to correspond with any known cursor resource. GetModuleHandleW is used to get the base address of the current function, which will be used later. RegQueryValueExW is used to query values from the registry (we will check the values being queries in the dynamic analysis section). We can also see a collection of if statements that are used to determine the return value.



```
                        get_mem_for_shellcode                          XREF[1]:
0044f250 55              PUSH    EBP
0044f251 8b ec           MOV     EBP, ESP
0044f253 81 ec c8 ...    SUB     ESP, 0xc8
0044f259 c6 85 56 ...    MOV     byte ptr [EBP + local_ae], 0xcf
0044f260 a1 04 02 ...    MOV     EAX, [->KERNEL32.DLL::VirtualAlloc]
0044f265 a3 9c c1 ...    MOV     [VirtualAlloc], EAX
0044f26a 8b 0d 84 ...    MOV     ECX, dword ptr [shellcode_orig_location]
0044f270 83 e9 04        SUB     ECX, 0x4
0044f273 89 0d 84 ...    MOV     dword ptr [shellcode_orig_location], ECX
0044f279 8b 15 84 ...    MOV     EDX, dword ptr [shellcode_orig_location]
0044f27f 8b 02           MOV     EAX, dword ptr [EDX]
0044f281 89 45 fc        MOV     dword ptr [EBP + local_8], EAX
0044f284 8b 0d 84 ...    MOV     ECX, dword ptr [shellcode_orig_location]
0044f28a 83 c1 04        ADD     ECX, 0x4
0044f28d 89 0d 84 ...    MOV     dword ptr [shellcode_orig_location], ECX
0044f293 8b 45 fc        MOV     EAX, dword ptr [EBP + local_8]
0044f296 8b e5           MOV     ESP, EBP
0044f298 5d              POP     EBP
0044f299 c3              RET
```

Figure 11: A function used to get the size of a shellcode

We will see in the dynamic analysis section that this program will load a shellcode. In a function at 0x44f250, the original location of the shellcode is loaded in, and a reference to the VirtualAlloc function is saved for later use. (We show the assembly view here because the decompiled view doesn't make the true functionality clear.)

```
undefined4 allocate_memory_for_shellcode(void)

{
  undefined4 base_allocated_addr;

  VirtualAlloc2 = VirtualAlloc;
  base_allocated_addr = (*VirtualAlloc)(0,shellcode_size,0x3000,0x40);
  return base_allocated_addr;
}
```

Figure 12: A function used to allocate memory for a shellcode program

Immediately after getting the shellcode size, a function at 0x44f810 is called to allocate the actual memory region for the shellcode. The memory region is allocated using flAllocationType = 0x3000, which corresponds to MEM_COMMIT and MEM_RESERVE. flProtect is set to 0x40, or PAGE_EXECUTE_READWRITE, which is typical of shellcode. Interestingly, another function at 0x44f890 is called next, and it allocates another page region with the same flAllocationType and flProtect.

```
  shellcode_orig_location = load_reg_values();
  shellcode_size = get_mem_for_shellcode();
  shellcode_base_address = allocate_memory_for_shellcode();
  uVar2 = 0x10a;
  puVar1 = &DAT_0048bbb8;
  other_shellcode_base_address = allocate_more_memory(300);
  load_extra_data(other_shellcode_base_address,(int)puVar1,uVar2);
  another_memory_location_ref = shellcode_size;
  new_shellcode_offset = 0;
  orig_shellcode_offset = 0;
  DAT_0048c164 = 0x2a;
  do {
    uVar2 = get_min(DAT_0048c0bc,another_memory_location_ref);
    if (shellcode_size <= new_shellcode_offset) break;
    load_shellcode(0x1405,0x298,uVar2);
    orig_shellcode_offset = orig_shellcode_offset + DAT_0048c164 + DAT_0048c0bc;
    new_shellcode_offset = new_shellcode_offset + DAT_0048c0bc;
    another_memory_location_ref = another_memory_location_ref - uVar2;
    _DAT_0048c14c = uVar2;
  } while (shellcode_base_address != 0);
  _shellcode_entry_address = shellcode_base_address + 0x30b20;
  call_xor_decode_all();
  return;
}
```

Figure 13: The second half of the entry function for cerber.exe

The rest of the code for cerber.exe focuses on calling functions that load both memory regions, and decode the shellcode.

```
void __cdecl load_shellcode(undefined4 param_1,undefined4 param_2,uint param_3)

{
  int iVar1;
  int iVar2;
  uint local_c;

  iVar1 = shellcode_base_address + new_shellcode_offset;
  iVar2 = shellcode_orig_location + orig_shellcode_offset;
  for (local_c = 0; (true && (local_c < param_3)); local_c = local_c + 1) {
    *(undefined1 *)(iVar1 + local_c) = *(undefined1 *)(iVar2 + local_c);
  }
  return;
}
```

Figure 14: The function used to load the shellcode

We can see in the above function that data is being copied from the shellcode's original location in the program to the newly allocated region of memory. The shellcode is loaded in individual segments, with the offset global variables coming from the outer do-while loop shown in *Figure 13*. This means that the encoded shellcode does not exist in one contiguous location in cerber.exe, making it harder to write a decoder to extract the shellcode manually.

```
2  void xor_decode_all(void)
3
4  {
5    key_offset = 0;
6    do {
7      if (shellcode_size <= key_offset) {
8        return;
9      }
10     shellcode_idx = (int *)(shellcode_base_address + key_offset);
11     *shellcode_idx = *shellcode_idx + key_offset;
12     full_xor_key = key_offset + 0x240340;
13     call_xor_encoding();
14     key_offset = key_offset + 4;
15   } while (true);
16   return;
17 }
18
```

Figure 15: A function used for decoding the shellcode

The above function is (apparently) the last code segment called by cerber.exe. It calls a short function that XORs a four byte memory segment with an XOR key. From the decompilation, we can see that the first four bytes are XORed with 0x240340. For every subsequent group of four bytes, 4 is added to this base key, so the second four bytes are XORed with 0x240344, then 0x240348, and so on. (This will be verified via dynamic analysis later.)

```
0044f7fb b8 f0 fb ...     MOV       other_shellcode_base_address, call_shellcode
0044f800 50               PUSH      other_shellcode_base_address=>call_shellcode
0044f801 c3               RET
```

Figure 16: A technique used to covertly call another function

Despite reaching the end of the entry function, we can see that another function will actually be called before the program exits. Just before returning from the entry function, the program pushes the address to another function on the stack. This means that when RET is called, it pops this new function address from the stack and the program "returns" to the head of this other function.

```
                    call_shellcode                                  XREF[2]:

0044fbf0 55             PUSH        EBP
0044fbf1 8b ec          MOV         EBP, ESP
0044fbf3 56             PUSH        ESI
0044fbf4 eb 00          JMP         LAB_0044fbf6

                    LAB_0044fbf6                                    XREF[1]:
0044fbf6 8b f6          MOV         ESI, ESI
0044fbf8 8b 25 5c ...   MOV         ESP, dword ptr [ebp_reference]
0044fbfe 8b f6          MOV         ESI, ESI
0044fc00 5d             POP         EBP
0044fc01 8b c0          MOV         EAX, EAX
0044fc03 ff 35 78 ...   PUSH        dword ptr [module_handle]
0044fc09 8b c0          MOV         EAX, EAX
0044fc0b ff 35 50 ...   PUSH        dword ptr [interesting_flag]
0044fc11 8b c0          MOV         EAX, EAX
0044fc13 33 d2          XOR         EDX, EDX
0044fc15 03 15 54 ...   ADD         EDX, dword ptr [shellcode_entry_address]
0044fc1b 8b c9          MOV         ECX, ECX
0044fc1d 8b c9          MOV         ECX, ECX
0044fc1f 8b c9          MOV         ECX, ECX
0044fc21 52             PUSH        EDX
0044fc22 8b c0          MOV         EAX, EAX
0044fc24 8b d2          MOV         EDX, EDX
0044fc26 c3             RET
```

Figure 17: A function used to call the loaded shellcode

The purpose of this final function is to call the shellcode. This function contains several instructions that are essentially NOPs, such as an unconditional jump to the next instruction, or instructions of the form MOV X, X. Most importantly, we can see that the address of the entry function in the shellcode is added to EDX, and this value is pushed to the stack just before returning, using the same sneaky trick as before to call the shellcode.

Section 2C: Decompiling the shellcode

We have seen that the shellcode is loaded from an offset address in cerber.exe. While manually decoding this file would be a difficult process due to the sophisticated loading scheme to bring the shellcode into memory, we don't actually need to do this. Using dynamic analysis, we can find the location the decoded shellcode is saved in, and dump that memory section to a file. We can then decompile this program using Ghidra. Because this is a shellcode that is meant to be run in memory, the decompilation may not be as helpful as using dynamic analysis. However, it will still help us to categorize the primary behaviors of the program a bit more effectively.

type (1)	size (bytes)	file-offset	blacklist (2)	hint (2)	group (3)	value (1747)
ascii	15	0x00000055	x	-	file	UnmapViewOfFile
ascii	15	0x00000012	-	-	dynamic-library	GetModuleHandle
ascii	14	0x000000BB	x	-	memory	VirtualProtect
ascii	14	0x00000099	-	-	file	SetFilePointer
ascii	14	0x00000066	-	-	dynamic-library	GetProcAddress
ascii	13	0x00000089	-	-	dynamic-library	LoadLibraryEx
ascii	12	0x00000077	-	-	memory	VirtualAlloc
ascii	11	0x00000044	-	-	memory	VirtualFree
ascii	11	0x000000AB	-	-	file	GetTempPath
ascii	11	0x00000033	-	-	-	CloseHandle
ascii	10	0x000000CD	-	-	file	CreateFile
ascii	10	0x00001773	-	-	-	P6X$, X$,&
ascii	10	0x0001B29B	-	-	-	(v~4}]f~)
ascii	10	0x0001CBFF	-	-	-	RX!V!%vONN
ascii	9	0x00000022	-	-	file	WriteFile

Figure 18: The strings tab in PEStudio for the shellcode

Before decompiling the shellcode, we can look for any interesting strings it contains. The most interesting strings appear to be Windows API function calls. Functions like VirtualProtect, VirtualAlloc, and VirtualFree indicate that the shellcode could be loading more code into memory. LoadLibraryEx and GetProcAddress will certainly be used to load in specific functions. We also see functions suggesting the shellcode can create and modify files.

```
21  local_10 = 0;
22  decode_memory(local_6c,0,0x58);
23  local_6c[0] = &stack0xfffffffc;
24  local_c = unaff_retaddr;
25  load_kernel32_functions(local_6c);
26  uVar1 = get_eip();
27  local_64 = local_c;
28  local_58 = param_1;
29  local_14 = param_1 + *(int *)(param_1 + 0x3c);
30  local_60 = (uint)*(ushort *)(local_14 + 0x16);
31  iVar2 = get_eip();
32  uVar3 = *(undefined4 *)(iVar2 + 0x401100);
33  iVar2 = get_eip(uVar3,uVar1);
34  iVar2 = iVar2 + 0x401104;
35  uVar1 = allocate_mem_for_program(uVar3,uVar3,uVar1,iVar2);
36  load_encoded_program(uVar1,iVar2,uVar3);
37  decode_program(uVar1,uVar3);
38  iVar2 = rearrange_and_reload_program(local_6c,uVar1);
39  if (iVar2 == 0) {
40    return 0;
41  }
42  if (local_64 != 0) {
43    *(undefined4 *)(local_6c[0] + 0x10) = local_54;
44  }
45  uVar3 = *(undefined4 *)(local_6c[0] + 8);
46  *(undefined4 *)(local_6c[0] + 8) = local_5c;
47  return uVar3;
48 }
```

Figure 19: The main code of the shellcode's entry function

Putting the shellcode file into Ghidra, we see the entry function contains the code in the above screenshot. (Because this is not a standard PE file, there is no easy way to determine this is the entry function statically. We know this is the entry function because of the dynamic analysis, which we will elaborate on later. Function names in Ghidra were also added after dynamic analysis.) As expected, the decompilation doesn't seem to help much with understanding the code.

```
2 void load_kernel32_functions(int param_1)
3
4 {
5   int iVar1;
6   undefined4 uVar2;
7   undefined4 LoadLibraryExA;
8   char local_2c [15];
9   char local_1c [13];
10  uint local_8;
11
12  builtin_strncpy(local_2c,"LoadLibraryExA",0xf);
13  builtin_strncpy(local_1c,"kernel32.dll",0xd);
14  LoadLibraryExA = unknown_1();
15  iVar1 = get_eip(LoadLibraryExA);
16  uVar2 = get_proc_address_stuff(LoadLibraryExA);
17  *(undefined4 *)(param_1 + 0x34) = uVar2;
18  LoadLibraryExA = (**(code **)(param_1 + 0x34))(LoadLibraryExA,local_2c);
19  *(undefined4 *)(param_1 + 0x3c) = LoadLibraryExA;
20  LoadLibraryExA = (**(code **)(param_1 + 0x3c))(local_1c,0,0);
21  for (local_8 = 0; local_8 < 0xf; local_8 = local_8 + 1) {
22    uVar2 = (**(code **)(param_1 + 0x34))(LoadLibraryExA,local_8 * 0x11 + iVar1 + 0x401000);
23    *(undefined4 *)(param_1 + 0x1c + local_8 * 4) = uVar2;
24  }
25  return;
26 }
```

Figure 20: A shellcode function used to load libraries

One function we can learn a bit about is shown above. We can see references to the strings “LoadLibraryExA” and “kernel32.dll”. In the assembly, these strings are loaded one byte at a time. Using static analysis alone, we can infer that this function is used to load in certain functions from the kernel32 library. Using dynamic analysis, we can confirm that this is, in fact, the case.

```
void allocate_mem_for_program(undefined4 param_1)

{
  undefined4 uVar1;
  code *pcVar2;
  char local_1c [13];

  builtin_strncpy(local_1c,"VirtualAlloc",0xd);
  uVar1 = unknown_1();
  pcVar2 = (code *)get_proc_address_stuff(uVar1);
  pcVar2 = (code *)(*pcVar2)(uVar1,local_1c);
  (*pcVar2)(0,param_1,0x3000,0x40);
  return;
}
```

Figure 21: A shellcode function used to allocate memory

Another function we can make reasonable inferences about via static analysis is shown above. This function references the string “VirtualAlloc”, and calls a function that references a string “GetProcAddress”. We can infer that this program is getting a reference to VirtualAlloc, and using it to allocate a new block of memory with PAGE_EXECUTE_READWRITE permission. Again, we can confirm this is the case using dynamic analysis.

```
void load_encoded_program(int param_1,int param_2,uint param_3)

{
  undefined4 local_c;

  for (local_c = 0; local_c < param_3; local_c = local_c + 1) {
    *(undefined1 *)(param_1 + local_c) = *(undefined1 *)(param_2 + local_c);
  }
  return;
}
```

Figure 22: A shellcode function used to load a program into memory

After the apparent memory allocation function returns, the above function is called. Its structure is similar to the code in *Figure 14*, indicating it is likely loading some data into memory. Dynamic analysis will reveal this data to be a full PE file. This function is also called multiple times later in the shellcode, and we will see this is because the loading of the PE file is being split between three sections.

```
void decode_program(int param_1,uint param_2)

{
  undefined4 local_8;

  for (local_8 = 0; local_8 < param_2; local_8 = local_8 + 4) {
    *(uint *)(param_1 + local_8) = *(int *)(param_1 + local_8) + local_8;
    *(uint *)(param_1 + local_8) = local_8 + 0x3e9 ^ *(uint *)(param_1 + local_8);
  }
  return;
}
```

Figure 23: A function used to decode a program

The next function called is shown above, and we can see what is most likely XOR decoding. We will see that this function is being used to decode the encoded program that was previously loaded into memory.

Following this point in the program, we can see several functions are called, but without running the program, it will be difficult to determine what those functions are.

Section 2D: Decompiling the new PE file

In the previous section, we implied that the shellcode loads a full PE file into memory. Later, we will see how this is done, and how the PE file gets run. Since this is a standard PE file, we can load it into PEStudio.

property	value
md5	325829FBF115A3E5D4718ADF5468846B
sha1	0CD39E096BB17522450852898504D429933BCF1F
sha256	660D14F5F817861EAAA62A55D5BF3B19BCFE1305C989912D99A33359CE6993BA
md5-without-overlay	AD34ADF8C6B7DB812E599BFFCAA8A8C2
sha1-without-overlay	0791B3243953ED0EE85180208534985 7BD4F8AE6
sha256-without-overlay	8EE93618A9D0B552D5FCCA9636AF6C08816CD72762E25164B882C062B585C850
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
first-bytes-text	M Z ..... @ .....
file-size	217088 (bytes)
size-without-overlay	196608 (bytes)
entropy	6.775
imphash	n/a
signature	n/a
entry-point	07 20 47 48 75 F5 89 7C 24 14 8B 75 0C 80 3E 00 0F 85 08 FE FF FF 83 7C 24 10 00 74 0F 8B 45 08 85
file-version	n/a
description	n/a
file-type	**executable**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x591D6AC7 (Thu May 18 05:35:03 2017)

Figure 24: The output in PEStudio for the new PE file

This program was apparently compiled for 32-bit systems on May 18, 2017 (about a week before cerber.exe was compiled). It has GUI capabilities.

property	value	value	value	value
name	.text	.rdata	.data	.reloc
md5	B3B491A76E3916D66AB4759...	A32CCF8117E59D3A87FB268...	EC485017C8CEBBB45995C83...	B1E27AA018409DE6BFD73F8...
entropy	6.441	4.815	7.419	0.000
file-ratio (90.09%)	25.47 %	2.12 %	60.38 %	2.12 %
raw-address	0x00000400	0x0000DC00	0x0000EE00	0x0002EE00
raw-size (195584 bytes)	0x0000D800 (55296 bytes)	0x00001200 (4608 bytes)	0x00020000 (131072 bytes)	0x00001200 (4608 bytes)
virtual-address	0x00401000	0x0040F000	0x00411000	0x00433000
virtual-size (200944 bytes)	0x0000D794 (55188 bytes)	0x000010DC (4316 bytes)	0x00021730 (137008 bytes)	0x00001150 (4432 bytes)
entry-point	0x0000948E	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x42000040
writable	-	-	x	-
executable	x	-	-	-

Figure 25: The sections table in PEStudio for the new PE file

This PE file contains four sections: .text, .rdata, .data, and .reloc. Of these, the most interesting is the .data section, since it has high entropy. While the raw-to-virtual size ratio is normal for .data, its entropy may indicate that it contains packed or encrypted information.

type (2)	size (bytes)	file-offset	blacklist (1)	hint (20)	group (5)	value (1839)
ascii	49	0x00033AE9	-	-	-	1 1$1(1.1014181<1@1D1H1L1P1T1X1\1`1d1h1l1p1t1x1|1
ascii	49	0x00033BFD	-	-	-	6 6$6(6.6064686<6@6D6H6L6P6T6X6\6`6d6h6l6p6t6x6|6
ascii	49	0x00033C7D	-	-	-	7 7$7(7.7074787<7@7D7H7L7P7T7X7\7`7d7h7l7p7t7x7|7
ascii	49	0x00033D05	-	-	-	9 9$9(9.9094989<9@9D9H9L9P9T9X9\9`9d9h9l9p9t9x9|9
ascii	43	0x00033B85	-	-	-	3(3.3034383<3D3H3L3P3T3X3\3`3d3h3l3p3t3x3|3
ascii	40	0x0000004D	-	dos-message	-	!This program cannot be run in DOS mode.
ascii	39	0x00033B31	-	-	-	2 2$2(2.2024282<2@2D2H2L2P2T2X2\2`2d2h2
ascii	34	0x00033AB8	-	-	-	(0.0004080<0@0D0H0L0P0X0h0l0p0t0|0
ascii	33	0x00033CD3	-	-	-	7.8084888<8@8D8H8L8P8T8X8\8`8h8|8
ascii	25	0x00033A6B	-	-	-	?$?(?4?8?D?H?T?X?d?h?t?x?
unicode	24	0x0000F5D5	-	-	-	Broken file found!\r\n%s
ascii	20	0x000100C6	x	-	memory	NtQueryVirtualMemory
ascii	20	0x00033390	-	-	-	b0l0`1d1h1l1p1t1x1|1

Figure 26: The strings tab in PEStudio for the new PE file

The strings tab doesn't contain too much interesting information, likely because most strings have been obfuscated in some way. One string we can find is "Broken file found\r\n%s", which likely is used to indicate that a file was unable to be encrypted properly. We can also see the string "NtQueryVirtualMemory" which references a function that could be used by the malware to get information about segments of memory. Another string we can find is "memmove" which references a C function that can be used to move data in memory.

```
13  cVar1 = load_libraries();
14  if (cVar1 != '\0') {
15    load_mutexes(local_2b0);
16    (*CreateMutexW)(0,0,local_2b0);
17    iVar2 = (*GetLastError)();
18    if (iVar2 != 0xb7) {
19      (*SetErrorMode)(0x8007);
20      base_address = (*GetModuleHandleA)(0);
21      (*GetModuleFileNameW)(0,&file_name,0x104);
22      pvVar3 = (void *)decrypt_config_file(extraout_ECX);
23      if (pvVar3 != (void *)0x0) {
24        parse_config_file((int)pvVar3);
25        cVar1 = create_temporary_files(pvVar3);
26        if (cVar1 != '\0') {
27          DAT_0043133c = check_privileges();
28          DAT_00431340 = DAT_0043133c;
29          if (0x2fff < DAT_0043133c) {
30            check_firewall_and_antivirus((int)pvVar3);
31          }
32          (*WSAStartup)(0x202,auStack_1a4);
33          DAT_0043111c = (*CreateEventW)(0,1,0,0);
34          uVar4 = (*CreateThread)(0,0,&LAB_0040925f,0,0,0);
35          encrypt_all_files();
36          (*WaitForSingleObject)(uVar4,0xffffffff);
37          (*CloseHandle)(uVar4);
38          (*CloseHandle)(DAT_0043111c);
39          spawn_a_new_process((int)pvVar3);
40          clean_up();
41        }
42      }
43    }
44    end_current_process();
45  }
46  return 0;
47}
```

Figure 27: The main code for the entry function of the new PE file

The entry function on the new PE file is shown above. We will discuss an overview of the most interesting behaviors from the functions shown in the above screenshot.

```
undefined1 load_library_functions(void)

{
  undefined4 *puVar1;
  code *pcVar2;
  short *loaded_library;
  void *pvVar3;
  uint uVar4;
  undefined4 *unaff_EDI;
  undefined1 local_5;

  pcVar2 = LoadLibraryExA;
  uVar4 = 0;
  local_5 = 0;
  loaded_library = (short *)(*LoadLibraryExA)(*unaff_EDI,0,0);
  if ((loaded_library != (short *)0x0) && (local_5 = 1, unaff_EDI[1] != 0)) {
    do {
      puVar1 = *(undefined4 **)(unaff_EDI[2] + uVar4 * 4);
      pvVar3 = load_functions(loaded_library,(char *)*puVar1,1,pcVar2);
      *puVar1 = pvVar3;
      if (pvVar3 == (void *)0x0) {
        return 0;
      }
      uVar4 = uVar4 + 1;
    } while (uVar4 < (uint)unaff_EDI[1]);
  }
  return local_5;
}
```

Figure 28: A function that loads libraries for use by the new PE file

The load_libraries function contains calls to many other functions, but its main purposes seems to be to load in Windows library functions to global variables (much like the Darkside ransomware). (Throughout this static analysis, we will assume the functions have already been linked to their proper global variables in Ghidra.) Above we can see that LoadLibraryExA (which is the first function loaded) is used to get a library. Then, a do-while loop is entered to load the functions from that library into global variables (in the "load_functions" function). This function is called for each library used by the malware.

```
void __fastcall
murmur_hash_encrypt(undefined4 param_1,byte *param_2,uint *param_3,uint *param_4,int param_5)

{
  byte *pbVar1;
  uint uVar2;
  uint uVar3;
  int iVar4;
  uint uVar5;
  uint uVar6;
  uint local_8;

  uVar5 = *param_3;
  uVar3 = *param_4;
  uVar6 = uVar3 & 3;
  local_8 = -uVar6 & 3;
  if ((local_8 != 0) && ((int)local_8 <= param_5)) {
    param_5 = param_5 - local_8;
    do {
      local_8 = local_8 - 1;
      uVar2 = uVar3 >> 8;
      uVar3 = uVar2 | (uint)*param_2 << 0x18;
      param_2 = param_2 + 1;
      uVar6 = uVar6 + 1;
      if (uVar6 == 4) {
        uVar3 = (uVar2 * 0x16a88000 | uVar3 * -0x3361d2af >> 0x11) * 0x1b873593;
        uVar5 = ((uVar5 ^ uVar3) << 0xd | (uVar5 ^ uVar3) >> 0x13) * 5 + 0xe6546b64;
        uVar6 = 0;
      }
    } while (local_8 != 0);
  }
  uVar2 = param_5 + (param_5 >> 0x1f & 3U) & 0xfffffffc;
  pbVar1 = param_2 + uVar2;
```

Figure 29: An implementation of the Murmur hash algorithm

We have labeled the next function in the entry function as "load_mutexes", and its primary function seems to be to load strings to use as mutexes for the program and any threads it creates. (We will see some examples later.) While loading mutexes, the above function is called many times. Based on the magic numbers 0x1b873593 and 0xe6546b64,

this appears to be an implementation of the MurmurHash3 (https://en.wikipedia.org/wiki/MurmurHash) algorithm. It seems this algorithm is being used to create unique identifiers for the mutexes.

```
bool __cdecl crypt_decrypt_config(void *param_1,void *param_2,void *param_3)

{
  int iVar1;
  int iVar2;
  size_t unaff_EDI;
  bool bVar3;

  bVar3 = false;
  iVar1 = get_decryption_key(param_1);
  if (iVar1 != 0) {
    if (param_3 == (void *)0x0) {
      param_3 = param_2;
    }
    else {
      memcpy(param_3,param_2,unaff_EDI);
    }
    iVar2 = (*CryptEncrypt)(iVar1,0,1,0,param_3,&stack0xffffffff8);
    bVar3 = iVar2 != 0;
    (*CryptDestroyKey)(iVar1);
  }
  return bVar3;
}
```

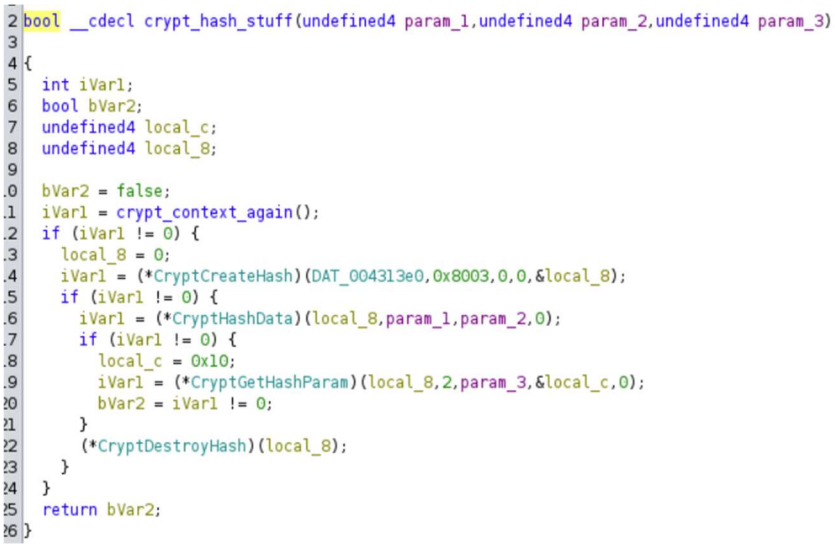
Figure 30: A function that appears to decrypt some data

The next function called by the entry function of the program is labeled in as "decrypt_config_file" in *Figure 27*. That the program does this is not obvious at all using static analysis, but we can see that Windows encryption functions are being called.

```
  local_5 = '\0';
  local_14 = (*CreateFileW)(&file_name,0x80000000,1,0,3,0,0);
  if (local_14 != -1) {
    local_18 = (*CreateFileMappingW)(local_14,0,2,0,0,0);
    if (local_18 != 0) {
      iVar3 = (*MapViewOfFile)(local_18,4,0,0,0);
      if (iVar3 != 0) {
        iVar4 = *(int *)(iVar3 + 0x3c) + iVar3;
        uVar9 = 0;
        local_10 = 0;
        if (*(ushort *)(iVar4 + 6) != 0) {
          puVar5 = (uint *)((uint)*(ushort *)(iVar4 + 0x14) + iVar4 + 0x2c);
          local_c = (uint)*(ushort *)(iVar4 + 6);
          do {
            uVar6 = *puVar5;
            if (local_10 < uVar6) {
              uVar9 = puVar5[-1] + uVar6;
              local_10 = uVar6;
            }
            puVar5 = puVar5 + 10;
            local_c = local_c - 1;
          } while (local_c != 0);
        }
        uVar6 = (*GetFileSize)(local_14,0);
        if (uVar9 < uVar6) {
          local_5 = visited88((void *)(uVar6 + iVar3),param_1);
        }
        (*UnmapViewOfFile)(iVar3);
      }
      (*CloseHandle)(local_18);
```

Figure 31: A function that seems to be accessing files

Then, a function we have labeled "parse_config_file" is called. We can see above that it appears certain files are created. We will see later that these functions are used by the program to read its own code.



```
bool __cdecl crypt_hash_stuff(undefined4 param_1,undefined4 param_2,undefined4 param_3)

{
  int iVar1;
  bool bVar2;
  undefined4 local_c;
  undefined4 local_8;

  bVar2 = false;
  iVar1 = crypt_context_again();
  if (iVar1 != 0) {
    local_8 = 0;
    iVar1 = (*CryptCreateHash)(DAT_004313e0,0x8003,0,0,&local_8);
    if (iVar1 != 0) {
      iVar1 = (*CryptHashData)(local_8,param_1,param_2,0);
      if (iVar1 != 0) {
        local_c = 0x10;
        iVar1 = (*CryptGetHashParam)(local_8,2,param_3,&local_c,0);
        bVar2 = iVar1 != 0;
      }
      (*CryptDestroyHash)(local_8);
    }
  }
  return bVar2;
}
```

Figure 32: A function used to create a cryptographic hash

Another function is called by the entry function labeled as “create_temporary_files”. It appears this function creates logs from running the program. In the process, the above function is called, which likely creates a hash to uniquely identify the infected machine.

```
undefined4 check_privileges(void)

{
  undefined4 uVar1;
  int iVar2;
  undefined4 *puVar3;
  char *pcVar4;
  undefined4 *puVar5;
  undefined4 uVar6;
  undefined4 local_c;
  undefined4 local_8;

  uVar6 = 0;
  local_c = 0;
  uVar1 = (*GetCurrentProcess)(8,&local_c);
  iVar2 = (*OpenProcessToken)(uVar1);
  if (iVar2 != 0) {
    local_8 = 0;
    iVar2 = (*GetTokenInformation)(local_c,0x19,0,0,&local_8);
    if (((iVar2 != 0) || (iVar2 = (*GetLastError)(), iVar2 == 0x7a)) &&
       (puVar3 = (undefined4 *)(*LocalAlloc)(0x40,local_8), puVar3 != (undefined4 *)0x0)) {
      iVar2 = (*GetTokenInformation)(local_c,0x19,puVar3,local_8,&local_8);
      if (iVar2 != 0) {
        pcVar4 = (char *)(*GetSidSubAuthorityCount)(*puVar3);
        puVar5 = (undefined4 *)(*GetSidSubAuthority)(*puVar3,*pcVar4 + -1);
        uVar6 = *puVar5;
      }
      (*LocalFree)(puVar3);
    }
    (*CloseHandle)(local_c);
  }
  return uVar6;
}
```

Figure 33: A function used to check the privileges of the running process

In the function we have labeled “check_privileges”, it appears that the malware may be checking if it has administrative permissions.

```
void create_new_process(void)

{
  byte *pbVar1;
  int iVar2;
  undefined4 local_68;
  undefined1 local_64 [72];
  undefined4 local_1c;
  undefined4 local_18;
  undefined4 local_c [2];

  local_c[0] = 0;
  pbVar1 = (byte *)visited12(&DAT_0040f24c,0x11,0xb23844ee);
  visited20(local_c,pbVar1);
  memset(local_64,0,0x40);
  local_68 = 0x44;
  iVar2 = (*CreateProcessA)(0,local_c[0],0,0,0,0x8000000,0,0,&local_68,&local_1c);
  if (iVar2 != 0) {
    iVar2 = (*WaitForSingleObject)(local_1c,5000);
    if (iVar2 == 0x102) {
      (*TerminateProcess)(local_1c,0);
    }
    (*CloseHandle)(local_1c);
    (*CloseHandle)(local_18);
  }
  free_heap();
  return;
}
```

Figure 34: A function that appears to create a new process

The next function in the entry function was labeled as "check_firewall_and_antivirus". Again, this functionality was not clear using static analysis alone. We can see some code, such as the function shown above, that creates a new process.

```
uint __cdecl encrypt_files(uint param_1)

{
  int in_EAX;
  undefined4 uVar1;
  int iVar2;
  int iVar3;
  uint unaff_EBX;
  uint uVar4;
  uint local_8;

  uVar4 = 0;
  if (((char)param_1 != '\0') && (1 < unaff_EBX)) {
    uVar1 = visited7(in_EAX);
    if ((char)uVar1 != '\0') {
      iVar2 = get_config_parameters(param_1,&local_8);
      if (iVar2 != 0) {
        if (unaff_EBX != 1) {
          do {
            iVar3 = something_encryption();
            *(undefined2 *)(in_EAX + uVar4 * 2) = *(undefined2 *)(iVar2 + iVar3 * 2);
            uVar4 = uVar4 + 1;
          } while (uVar4 < unaff_EBX - 1);
        }
        *(undefined2 *)(in_EAX + -2 + unaff_EBX * 2) = 0;
        free_heap();
        uVar4 = 1;
      }
    }
  }
  return uVar4;
}
```

Figure 35: A function that seems to perform encryption

Next in the entry function, a new thread is created with CreateThread. In the dynamic analysis section, we will take a closer look at the code this thread is running. We will see that it is calling the above function, which appears to be used for encryption, possibly for encrypting the file system.

```
local_38 = 2;
local_36 = (*htons)((uint)uVar7 & 0xffff);
local_24 = (*socket)(2,2,0x11);
if (local_24 != -1) {
```

Figure 36: Socket communication functions

Just before encrypting files, the program performs some sort of socket communication, as can be seen above.

```
33  local_5 = 0;
34  visited12(&DAT_0040fb60,9,0xb218bce3);
35  local_c = visited78(param_1);
36  if (local_c != 0) {
37    piVar7 = &local_30;
38    visited12(&DAT_0040fb6c,4,0x6e758680);
39    visited78(local_c);
40    visited87();
41    local_18 = lots_of_string_comparing(piVar7);
42    if ((local_18 != (undefined1 *)0x0) && (iVar1 = (*GetDC)(0), iVar1 != 0)) {
43      local_30 = iVar1;
44      iVar2 = (*CreateCompatibleDC)(iVar1);
45      if (iVar2 != 0) {
46        local_10 = (*GetDeviceCaps)(iVar1,8);
47        local_14 = (*GetDeviceCaps)(iVar1,10);
48        local_1c = (*CreateCompatibleBitmap)(iVar1,local_10,local_14);
49        if (local_1c != 0) {
50          local_2c = (*SelectObject)(iVar2,local_1c);
51          uVar3 = (*GetDeviceCaps)(iVar1,0x5a,0x48);
52          visited12(&DAT_0040fb74,4,0x9516b2f4);
53          visited78(local_c);
54          uVar6 = visited79();
55          iVar1 = (*MulDiv)((int)uVar6,uVar3);
56          uVar3 = visited6(&DAT_0040fb7c,8,0xa85a521b);
57          local_28 = (*CreateFontW)(-iVar1,0,0,0,0,0,0,0,1,0,0,4,0,uVar3);
58          if (local_28 != 0) {
59            local_24 = (*SelectObject)(iVar2,local_28);
60            visited12(&DAT_0040fb88,10,0x6bfcbc7b);
61            visited78(local_c);
62            uVar6 = visited79();
63            (*SetBkColor)(iVar2,(int)uVar6);
64            visited12(&DAT_0040fb94,5,0x2412ab51);
65            visited78(local_c);
66            uVar6 = visited79();
67            (*SetTextColor)(iVar2,(int)uVar6);
```

Figure 37: Windows calls likely used to create the desktop background

Once the encryption has completed, the above function gets called. This function contains many Windows API calls that are related to GUIs. Therefore, it is reasonable to assume this function is where the desktop background is created.

Section 3: Dynamic Analysis

Section 3A: Basic dynamic analysis of cerber.exe

In this section, we will run cerber.exe to analyze its behavior. We will run the malware by double-clicking its icon on a Windows 7 machine.

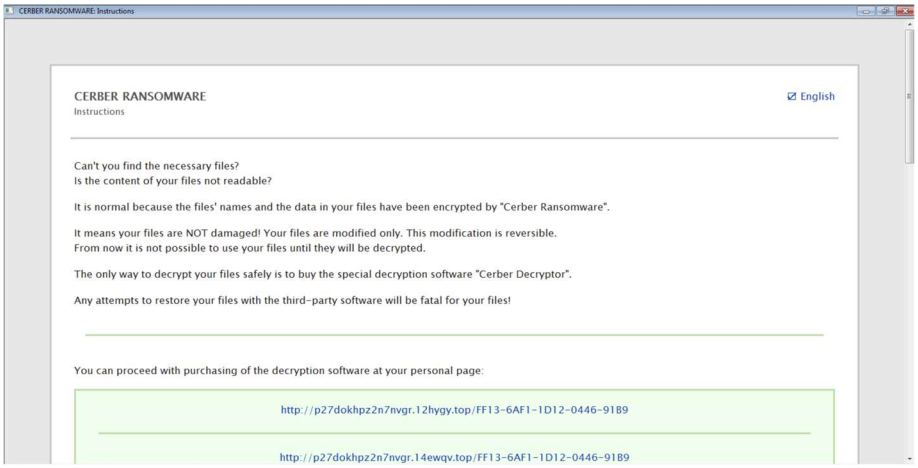


Figure 38: A .hta file instructing the user to purchase decryption software

A few seconds after starting the malware, a .hta file displaying the contents in the above screenshot will open. It explains that the files on the infected machine have been encrypted, and instructs the user to visit a website to purchase decryption software from the malware authors.

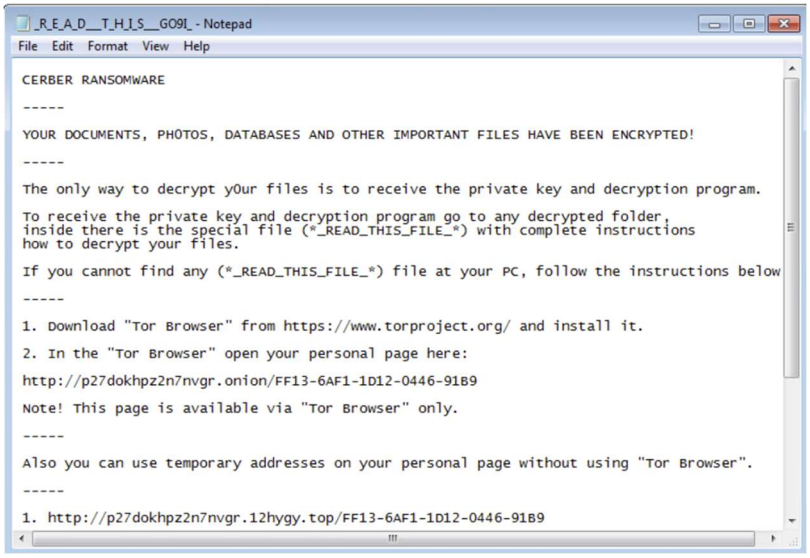


Figure 39: A .txt file added to the infected machine with more instructions for the ransom

The above .txt file is also added to most directories throughout the file system. It contains more instructions for the user, telling them to access a website using Tor. Alternatively, temporary links are also provided that can be accessed without Tor.

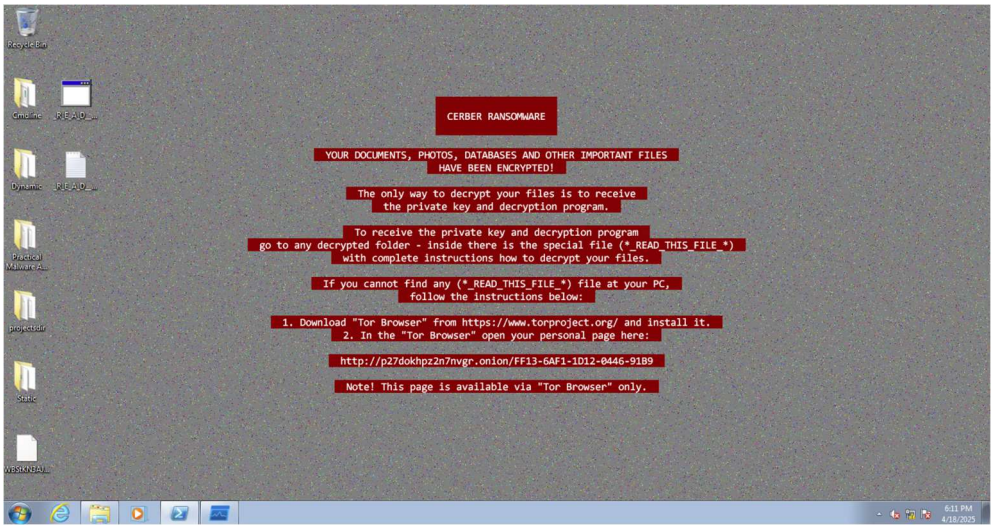


Figure 40: The desktop after the malware has finished running

Once the malware has finished running, the desktop background will also be changed to give the user the same instructions from the two other files.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:01:4...	cerber.exe	3264	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\	SUCCESS	Desired Access: R...
6:02:0...	cerber.exe	3264	RegCreateKey	HKLM\System\CurrentControlSet\Control\Session Manager	REPARSE	Desired Access: R...
6:02:0...	cerber.exe	3264	RegCreateKey	HKLM\System\CurrentControlSet\Control\Session Manager	ACCESS DENIED	Desired Access: R...
6:02:0...	cerber.exe	3264	RegCreateKey	HKLM\System\CurrentControlSet\Control\Session Manager	REPARSE	Desired Access: R...
6:02:0...	cerber.exe	3264	RegCreateKey	HKLM\System\CurrentControlSet\Control\Session Manager	ACCESS DENIED	Desired Access: R...

Figure 41: Registry keys created by cerber.exe

Only one registry key related to internet settings appears to have been successfully created by cerber.exe. It also tries to create a key HKLM\System\CurrentControlSet\Control\Session Manager, but it seems to have insufficient permissions to do so.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Control Panel\Desktop\Wallpaper	SUCCESS	Type: REG_SZ, Le...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet	SUCCESS	Type: REG_DWO...
6:01:4...	cerber.exe	3264	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect	SUCCESS	Type: REG_DWO...

Figure 42: Registry key values modified by cerber.exe

The HKCU\Control Panel\Desktop\Wallpaper key was modified, which makes sense since we have observed that the desktop wallpaper has changed. We can also see several values in the newly created Internet Settings key were modified by cerber.exe.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\Prefetch\CERBER.EXE-C77A6573.pf	NAME NOT FOUND	Desired Access: Generic Read, Disposition: Open, Options: Synchronous IO Non-Alert
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop	SUCCESS	Desired Access: Execute/Traverse, Synchronize, Disposition: Open, Options: Directory
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\aechost.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\aechost.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\cerber.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.7601.18837_none_ec86b8d6858ec0bc	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.7601.18837_none_ec86b8d6858ec0bc	SUCCESS	Desired Access: Execute/Traverse, Synchronize, Disposition: Open, Options: Directory
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.7601.18837_none_ec86b8d6858ec0bc\comctl32.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.7601.18837_none_ec86b8d6858ec0bc\comctl32.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open, Options: S
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Data/List Directory, Synchronize, Disposition: Open, Options: S
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\imm32.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\cerber.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.24203_none_5c030043a0118fbf	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.24203_none_5c030043a0118fbf	SUCCESS	Desired Access: Execute/Traverse, Synchronize, Disposition: Open, Options: Directory
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.24203_none_5c030043a0118fbf\GdiPlus.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.24203_none_5c030043a0118fbf\GdiPlus.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\mpr.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\mpr.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\mpr.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\netapi32.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\netapi32.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\netapi32.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\netutils.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\netutils.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\netutils.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\srvcli.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\srvcli.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\srvcli.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\wkscli.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\wkscli.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\wkscli.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\SAMCLI.DLL	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\samcli.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\samcli.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition
6:01:2...	cerber.exe	3264	CreateFile	C:\Users\malware\Desktop\powprof.dll	NAME NOT FOUND	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\powprof.dll	SUCCESS	Desired Access: Read Attributes, Disposition: Open, Options: Open Reparse Point, Att
6:01:2...	cerber.exe	3264	CreateFile	C:\Windows\System32\powprof.dll	SUCCESS	Desired Access: Read Data/List Directory, Execute/Traverse, Synchronize, Disposition

Figure 43: Files created or accessed by cerber.exe

Many files were created or accessed by this ransomware. This is expected, since we know that most files will be encrypted. A few interesting modifications do stand out, however.

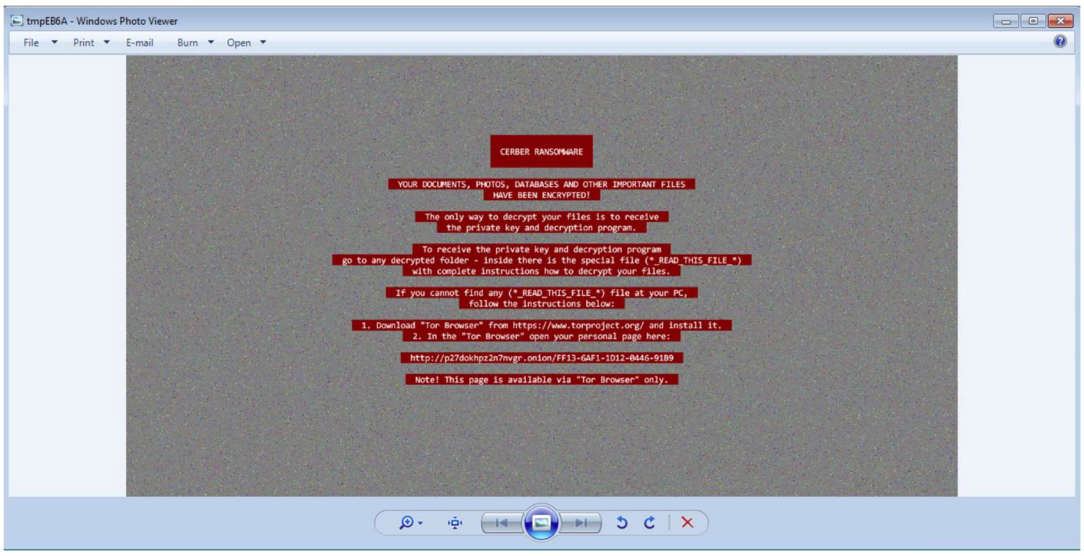


Figure 44: A saved image that was set as the desktop background

We can see that a new image was created in C:\Users\malware\AppData\Local\Temp. On closer inspection, this is the image that was used to replace the desktop background.

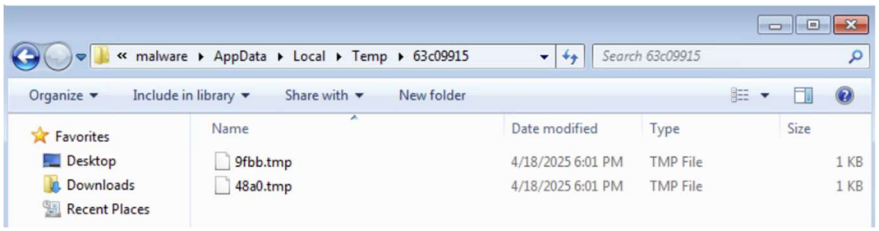


Figure 45: Two temporary files created by cerber.exe

We can also see that two new temporary files were created in a folder called 63c09915 in the same directory.

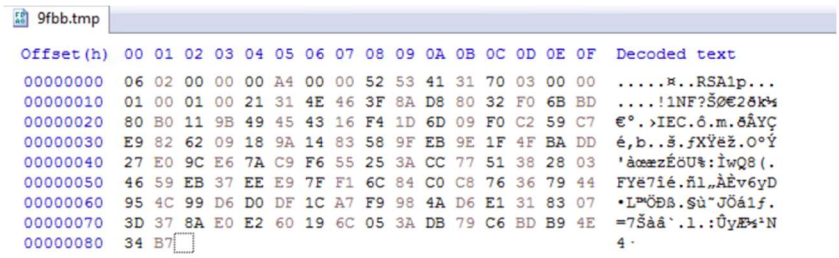


Figure 46: The contents of the file 9fbb.tmp

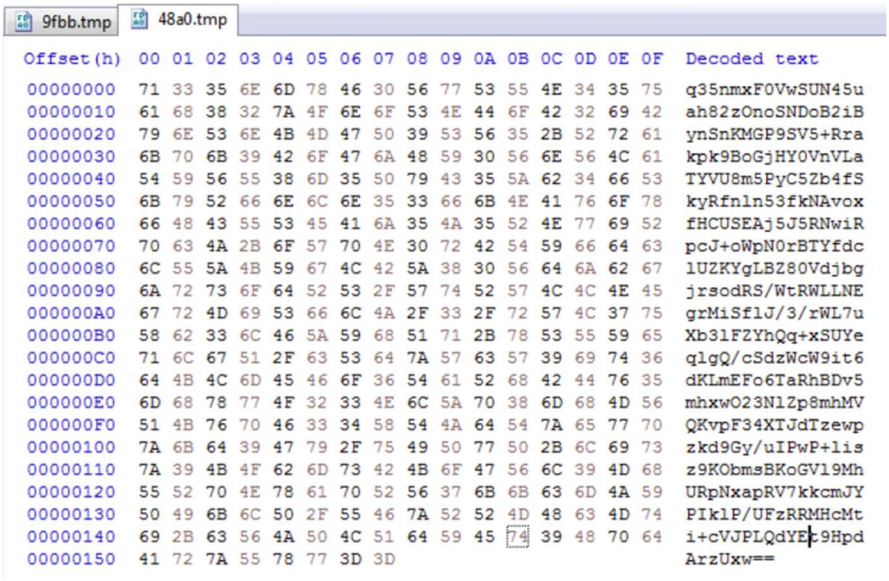


Figure 47: The contents of the file 48a0.tmp

The first of these files appears to be some information related to an RSA key exchange. The second file seems to include a string that has been encoded in base 64.

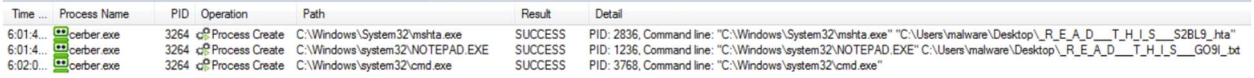


Figure 48: Processes created by cerber.exe

During its execution, it appears that cerber.exe creates three new processes. The first process is mshta.exe. Based on the detail section, it appears that this process is being used to open the .hta file. The second process is NOTEPAD.EXE, which seems to be used to open the .txt file after the program has encrypted the file system. Finally, a cmd.exe process is created. To understand what this process is being used for, we will need to take a closer look at it.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:01:2...	cerber.exe	3264	RegOpenKey	HKLM\SYSTEM\CurrentControlSet\Services\crypt32	REPARSE	Desired Access: Read
6:01:2...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\crypt32	SUCCESS	Desired Access: Read
6:01:2...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\LanmanWorkstation\Parameters	REPARSE	Desired Access: Query Value
6:01:2...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\LanmanWorkstation\Parameters	SUCCESS	Desired Access: Query Value
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\WinSock2\Parameters	REPARSE	Desired Access: All Access
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\WinSock2\Parameters	ACCESS DE...	Desired Access: All Access
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\WinSock2\Parameters	REPARSE	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\Services\WinSock2\Parameters	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\AppId_Catalog	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\AppId_Catalog\0355E90E	NAME NOT ...	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9	SUCCESS	Desired Access: Maximum Allowed, Granted Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\00000043	NAME NOT ...	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries	SUCCESS	Desired Access: Maximum Allowed, Granted Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000001	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000002	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000003	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000004	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000005	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000006	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000007	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000008	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000009	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000010	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000011	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000012	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000013	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000014	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000015	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000016	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000017	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000018	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000019	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000020	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000021	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000022	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000023	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000024	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\Protocol_Catalog9\Catalog_Entries\000000000025	SUCCESS	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\NameSpace_Catalog5	SUCCESS	Desired Access: Maximum Allowed, Granted Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\NameSpace_Catalog5\0000000D	NAME NOT ...	Desired Access: Read
6:01:3...	cerber.exe	3264	RegOpenKey	HKLM\System\CurrentControlSet\services\WinSock2\Parameters\NameSpace_Catalog5\Catalog_Entries	SUCCESS	Desired Access: Maximum Allowed, Granted Access: Read

Figure 49: Registry key operations related to service creation

cerber.exe does not appear to create any new services. It does still read many service-related registry keys, meaning it might be looking for particular services.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Internet Explorer\Main	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Internet Settings	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Content\CachePrefix	SUCCESS	Type: REG_SZ, Length: 2, Data:
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Cookies\CachePrefix	SUCCESS	Type: REG_SZ, Length: 16, Data: Cookie:
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\History\CachePrefix	SUCCESS	Type: REG_SZ, Length: 18, Data: Visited:
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections	SUCCESS	Desired Access: Query Value, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections	SUCCESS	Desired Access: Query Value, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections	SUCCESS	Desired Access: Query Value, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings	SUCCESS	Desired Access: Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyEnable	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:01:4...	mshta.exe	2836	RegDeleteValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyServer	NAME NOT ...	
6:01:4...	mshta.exe	2836	RegDeleteValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyOverride	NAME NOT ...	
6:01:4...	mshta.exe	2836	RegDeleteValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\AutoConfigURL	NAME NOT ...	
6:01:4...	mshta.exe	2836	RegDeleteValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\AutoDetect	NAME NOT ...	
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections	SUCCESS	Desired Access: Query Value, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections	SUCCESS	Desired Access: Set Value, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\SavedLegacySettings	SUCCESS	Type: REG_BINARY, Length: 184, Data: 46 00 00 00 10 00 00 00 09 00 00 00 00 00 00 00
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	REPARSE	Desired Access: Read
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	SUCCESS	Desired Access: Read, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	REPARSE	Desired Access: Read
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	SUCCESS	Desired Access: Read, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	REPARSE	Desired Access: Read
6:01:4...	mshta.exe	2836	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters	SUCCESS	Desired Access: Read, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad	SUCCESS	Desired Access: Read/Write, Disposition: REG_OPENED_EXISTING_KEY
6:01:4...	mshta.exe	2836	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}	SUCCESS	Desired Access: Set Value, Create Sub Key, Enumerate Sub Keys, Disposition: REG_CREATED_NEW_KEY
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}\...	SUCCESS	Type: REG_DWORD, Length: 4, Data: 1
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}\...	SUCCESS	Type: REG_BINARY, Length: 8, Data: 60 34 F4 7B AD B0 DB 01
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}\...	SUCCESS	Type: REG_DWORD, Length: 4, Data: 0
6:01:4...	mshta.exe	2836	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}\...	SUCCESS	Type: REG_SZ, Length: 22, Data: Network 6
6:01:4...	mshta.exe	2836	RegDeleteValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{817D6D99-BC25-4A24-B983-4236CC69D342}\...	NAME NOT ...	

Figure 50: Registry modifications be mshta.exe

The mshta.exe process makes a lot of modifications to the registry, but none seem particularly suspicious or interesting. At least from the basic dynamic analysis, it does not appear that this process is doing anything besides opening the .hta file. Basic dynamic analysis doesn't seem to reveal anything interesting about the NOTEPAD.EXE process, either.

Time ...	Process Name	PID	Operation	Path	Result	Detail
6:02:0...	cmd.exe	3768	Process Start		SUCCESS	Parent PID: 3264, Command line: "C:\Windows\system32\cmd.exe"
6:02:0...	cmd.exe	3768	Process Create	C:\Windows\system32\taskkill.exe	SUCCESS	PID: 3708, Command line: taskkill /f /im "cerber.exe"
6:02:0...	cmd.exe	3768	Process Create	C:\Windows\system32\PING.EXE	SUCCESS	PID: 2752, Command line: ping -n 1 127.0.0.1

Figure 51: Processes created by the cmd.exe process

The cmd.exe process is a bit more interesting. We can see that it creates two new processes. The first process uses taskkill (https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/taskkill), which seems to force the cerber.exe process to end. The second process simply pings 127.0.0.1 a single time. It is unclear at this point in the analysis why the malware would do this, but it could be a method of detecting whether the program is running in a sandbox.

```
fakedns[INFO]: Response: 0.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 1.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 2.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 3.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 4.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 5.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 6.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 7.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 8.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 9.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 10.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 11.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 12.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 13.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 14.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 15.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 16.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 17.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 18.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 19.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 20.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 21.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 22.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 23.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 24.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 25.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 26.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 27.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 28.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 29.158.33.178.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 30.158.33.178.in-addr.arpa -> 192.168.245.133
```

Figure 52: Reverse DNS lookups made by cerber.exe

We can see in the above screenshot that the malware makes several reverse DNS lookups for IP addresses in the subnets 178.33.158.0/27, 178.33.159.0/27 and 178.33.160.0/22. Perhaps the malware is looking to connect with a C2 server, and the server it needs to connect to is in one of the checked ranges.

```
fakedns[INFO]: Response: api.blockcypher.com -> 192.168.245.133
fakedns[INFO]: Response: btc.blockr.io -> 192.168.245.133
fakedns[INFO]: Response: bitaps.com -> 192.168.245.133
fakedns[INFO]: Response: ctldl.windowsupdate.com -> 192.168.245.133
fakedns[INFO]: Response: chain.so -> 192.168.245.133
fakedns[INFO]: Response: ctldl.windowsupdate.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: ctldl.windowsupdate.com -> 192.168.245.133
fakedns[INFO]: Response: fs.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: g.live.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: config.edge.skype.com -> 192.168.245.133
fakedns[INFO]: Response: config.edge.skype.com -> 192.168.245.133
fakedns[INFO]: Response: g.live.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: g.live.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: config.teams.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: settings-win.data.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: config.edge.skype.com -> 192.168.245.133
fakedns[INFO]: Response: config.edge.skype.com -> 192.168.245.133
fakedns[INFO]: Response: api.blockcypher.com -> 192.168.245.133
fakedns[INFO]: Response: btc.blockr.io -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: time.windows.com -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: tlu.dl.delivery.mp.microsoft.com -> 192.168.245.133
```

Figure 53: Domains accessed by cerber.exe

We can also see that the malware has attempted to access several domains related to cryptocurrency. These domains include api.blockcypher.com, btc.blockr.io, bitaps.com, and chain.so. Perhaps the malware is attempting to steal cryptocurrency, or it is providing a means of paying the ransom.

```
2025-04-18 14:01:49  HTTP connection, method: GET, URL: http://api.blockcypher.com/
v1/btc/main/addrs/17gd1msp5FnMcEMF1MitTNSsYs7w7AQyCt?_=1745013709049, file name: /v
ar/lib/inetsim/http/fakefiles/sample.html
2025-04-18 14:01:49  HTTP connection, method: GET, URL: http://btc.blockr.io/api/v1
/address/txs/17gd1msp5FnMcEMF1MitTNSsYs7w7AQyCt?_=1745013709224, file name: /var/li
b/inetsim/http/fakefiles/sample.html
```

Figure 54: HTTP requests sent by cerber.exe

Above we can see HTTP requests sent to two of the identified cryptocurrency domains. It appears the malware may be sending the address of a specific cryptocurrency wallet to each domain.

```
function server40(address, callback) {
	getUrlContent("https://cha"+""+"in.so/api/v2/address/btc/" + address, 0, function(result, error) {
		if (!error) {
			var txs = result.match(/"outputs":[{"output_no":[\w]+,"address":"[\w]+"/g);
			if (txs) {
				var regExp = /"address":"([\w]+)"/;
				var address = regExp.exec(txs[0]);
				if (address) {
					return callback(address[1], null);
				}
				else {
					return callback(null, true);
				}
			}
			else {
				return callback(null, true);
			}
		}
		else {
			return callback(null, true);
		}
	});
}
```

Figure 55: A JavaScript function used to communicate with chain.so

On closer inspection of the .hta file, we can see embedded JavaScript that includes 4 functions: server10, server20, server30, and server40. Each of these functions communicates with one of the cryptocurrency domains. Most likely, the .hta file is intended to have links to specific wallets from these domains, but the feature is broken because the machine is not connected to the internet.

Section 3B: Debugging cerber.exe

In this section, we will use x32dbg to debug cerber.exe. Our first goal in this section will be to take a closer look at how the program is being deobfuscated.

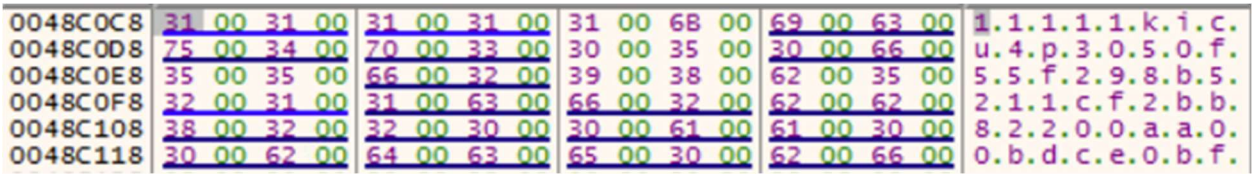


Figure 56: A section of memory with encoded text

Recall that in *Figure 7* we showed a sequence of instructions that seem to be decoding some data at the start of the program execution. We can see that the data being decoded is stored at 0x48c0c8, and it is decoded in place.

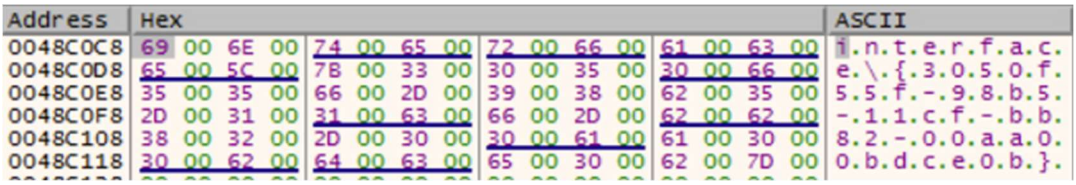


Figure 57: The memory section from above after being decoded

After the data has been decoded, we see that it contains the string "interface\{3050f55f-98b5-11cf-bb82-00aa00bdce0b}", which corresponds to the DispHTMLDocument interface of mshtml.

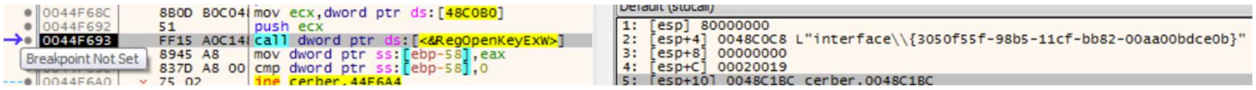


Figure 58: The inputs to RegOpenKeyExW, including the DispHTMLDocument string

In the screenshot above, we can see that this string is being input as the sub key to RegOpenKeyExW (which we can see called in *Figure 6*).

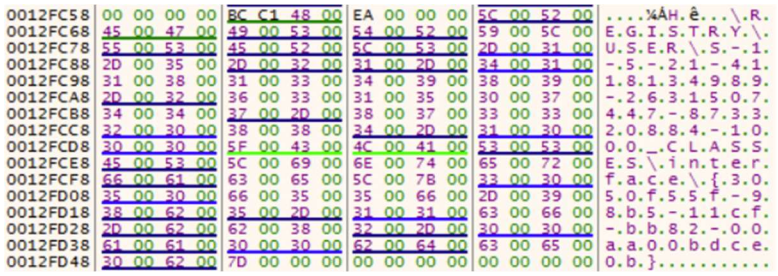


Figure 59: Data added to the stack during execution of RegOpenKeyExW

While the registry key function is executing, new data is added to the stack (as seen in the above screenshot). Of particular interest is the string "S-1-5-21-4118134989-2631507447-873320884-1000", which seems to be a Microsoft security identifier (SID).

Figure 60: Arguments passed to the RegQueryValueExW function

We can see the malware is then using this SID value to query the DispHTMLDocument key.

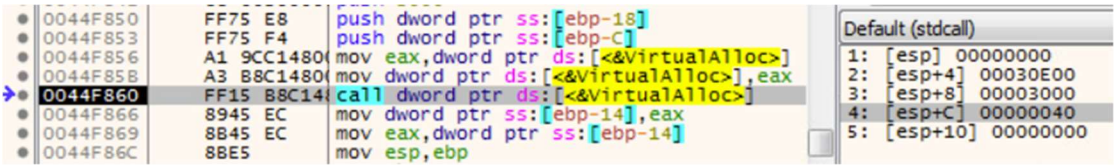


Figure 61: The parameters passed to VirtualAlloc for the shellcode

We showed in *Figure 12* that VirtualAlloc is called to allocate memory for the shellcode. Above, we can see that the size of this shellcode is 0x30E00 bytes. The second memory region allocated in the main program code for cerber.exe only contains 0x12C bytes.

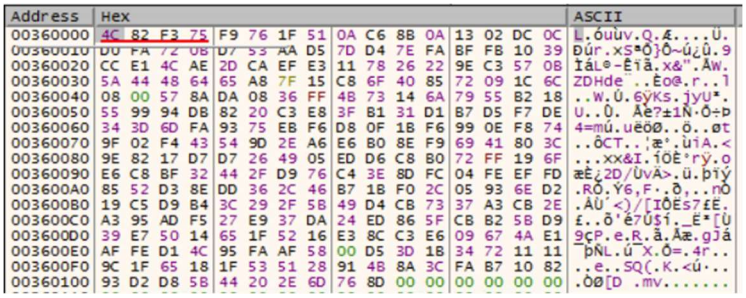
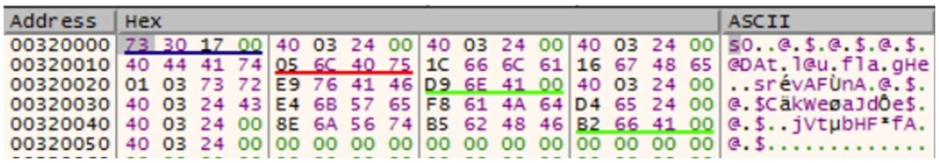


Figure 62: The contents of a newly allocated memory segment

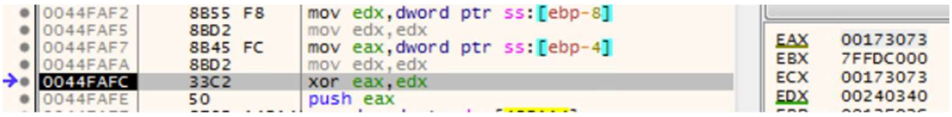
The smaller memory segment is loaded with data first, and this data can be seen above. It appears to be encoded, so it is not clear what it contains.



Address	Hex												ASCII
00320000	73 30 17 00	40 03 24 00	40 03 24 00	40 03 24 00	s0..@.$.@.$.@.$.
00320010	40 44 41 74	05 6C 40 75	1C 66 6C 61	16 67 48 65	@DAt.l@u.fla.gHe
00320020	01 03 73 72	E9 76 41 46	D9 6E 41 00	40 03 24 00	..srévAFÙnA.@.$.
00320030	40 03 24 43	E4 6B 57 65	F8 61 4A 64	D4 65 24 00	@.$CäkWeøaJdÔe$.
00320040	40 03 24 00	8E 6A 56 74	B5 62 48 46	B2 66 41 00	@.$..jVtµbHF²fA.
00320050	40 03 24 00	00 00 00 00	00 00 00 00	00 00 00 00	@.$.............

Figure 63: The first block of the shellcode prior to decoding

We mentioned in the static analysis section that this program is loaded in from different memory locations one block at a time. We show the first encoded block above. Each block is 163 bytes long.

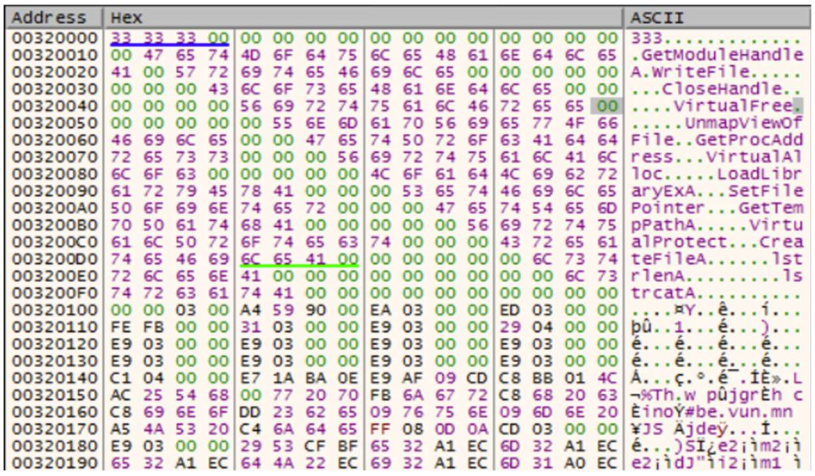


```
0044FAF2   8B55 F8   mov edx,dword ptr ss:[ebp-8]
0044FAF5   8BD2      mov edx,edx
0044FAF7   8B45 FC   mov eax,dword ptr ss:[ebp-4]
0044FAFA   8BD2      mov edx,edx
0044FAFC   33C2      xor eax,edx
0044FAFE   50        push eax

EAX  00173073
EBX  7FFDC000
ECX  00173073
EDX  00240340
```

Figure 64: The initial XOR key used for decode the shellcode

As mentioned in the discussion for *Figure 15*, the initial key for the XOR decoding is 0x240340 (in EDX), and it is used to decode 4 bytes. Each subsequent 4 bytes are decoded by incrementing this key by 4 each time.



Address	Hex												ASCII
00320000	33 33 33 00	00 00 00 00	00 00 00 00	00 00 00 00	333............
00320010	00 47 65 74	4D 6F 64 75	6C 65 48 61	6E 64 6C 65	.GetModuleHandle
00320020	41 00 57 72	69 74 65 46	69 6C 65 00	00 00 00 00	A.WriteFile.....
00320030	00 00 00 43	6C 6F 73 65	48 61 6E 64	6C 65 00 00	...CloseHandle..
00320040	00 00 00 00	56 69 72 74	75 61 6C 46	72 65 65 00	....VirtualFree.
00320050	00 00 00 00	00 55 6E 6D	61 70 56 69	65 77 4F 66	.....UnmapViewOf
00320060	46 69 6C 65	00 00 47 65	74 50 72 6F	63 41 64 64	File..GetProcAdd
00320070	72 65 73 73	00 00 00 56	69 72 74 75	61 6C 41 6C	ress...VirtualAl
00320080	6C 6F 63 00	00 00 00 00	4C 6F 61 64	4C 69 62 72	loc.....LoadLibr
00320090	61 72 79 45	78 41 00 00	00 53 65 74	46 69 6C 65	aryExA...SetFile
003200A0	50 6F 69 6E	74 65 72 00	00 00 47 65	74 54 65 6D	Pointer...GetTem
003200B0	70 50 61 74	68 41 00 00	00 00 00 56	69 72 74 75	pPathA.....Virtu
003200C0	61 6C 50 72	6F 74 65 63	74 00 00 00	43 72 65 61	alProtect...Crea
003200D0	74 65 46 69	6C 65 41 00	00 00 00 00	00 6C 73 74	teFileA......lst
003200E0	72 6C 65 6E	41 00 00 00	00 00 00 00	00 00 6C 73	rlenA.........ls
003200F0	74 72 63 61	74 41 00 00	00 00 00 00	00 00 00 00	trcatA.........
00320100	00 00 03 00	A4 59 90 00	EA 03 00 00	ED 03 00 00	....¤Y..ê...í...
00320110	FE FB 00 00	31 03 00 00	E9 03 00 00	29 04 00 00	þû..1...é...)...
00320120	E9 03 00 00	E9 03 00 00	E9 03 00 00	E9 03 00 00	é...é...é...é...
00320130	E9 03 00 00	E9 03 00 00	E9 03 00 00	E9 03 00 00	é...é...é...é...
00320140	C1 04 00 00	E7 1A BA 0E	E9 AF 09 CD	C8 BB 01 4C	Á...ç.°.é¯.ÍÈ».L
00320150	AC 25 54 68	00 77 20 70	FB 6A 67 72	C8 68 20 63	¬%Th.w pûjgrÈh c
00320160	C8 69 6E 6F	DD 23 62 65	09 76 75 6E	09 6D 6E 20	ÈinoÝ#be.vun.mn
00320170	A5 4A 53 20	C4 6A 64 65	FF 08 0D 0A	CD 03 00 00	¥JS Ájdeÿ...Í...
00320180	E9 03 00 00	29 53 CF BF	65 32 A1 EC	6D 32 A1 EC	é...)SÏ¿e2¡ìm2¡ì
00320190	65 32 A1 EC	64 4A 22 EC	69 32 A1 EC	6D 31 A0 EC	e2¡ìdJ"ìi2¡ìm1 ì

Figure 65: The first several bytes of the decoded shellcode

We show the beginning of the fully-decoded shellcode above. Already, we can see strings indicating some of the Windows API functions the code will use. We can also dump the shellcode to a file from the memory map tab in x32dbg. We used this is Section 2C to decompile the shellcode.

Section 3C: Debugging the shellcode

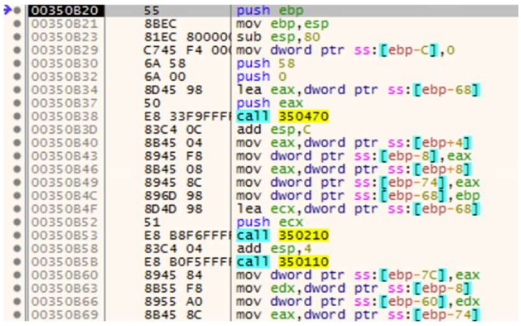


Figure 66: The entry function of the shellcode

Next, the shellcode is called by pushing the address of the entry function to the stack just before calling RET. We will see how this shellcode loads in the PE file containing the main payload of the program.

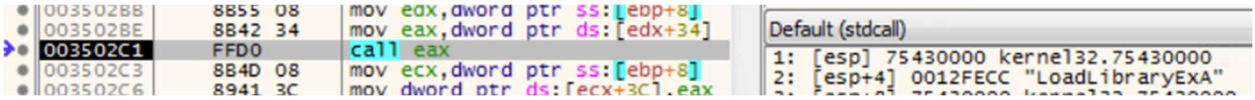


Figure 67: A call to GetProcAddress to get LoadLibraryExA

In *Figure 20* we saw a function that appeared to be used to load functions from kernel32.dll. We can see above that GetProcAddress is used to get a reference to the LoadLibraryExA function, which is then used to load kernel32.dll. Certain functions from this library are loaded in a loop afterwards.

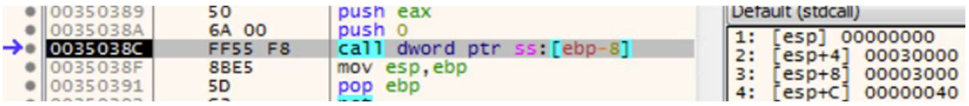


Figure 68: A call to VirtualAlloc by the program for loading a PE file

We mentioned in Section 2C that the shellcode allocates three memory regions for PE files. We show the first call to VirtualAlloc for memory allocation above, with the permissions set to PAGE_EXECUTE_READWRITE. This allocated section is filled with encoded data by the function in *Figure 22*. Then, the data is encoded in place using the function in *Figure* 23. The initial XOR key is 0x3E9, and, like the shellcode decoding, this initial key is incremented by 4 to encode each subsequent group of 4 bytes.

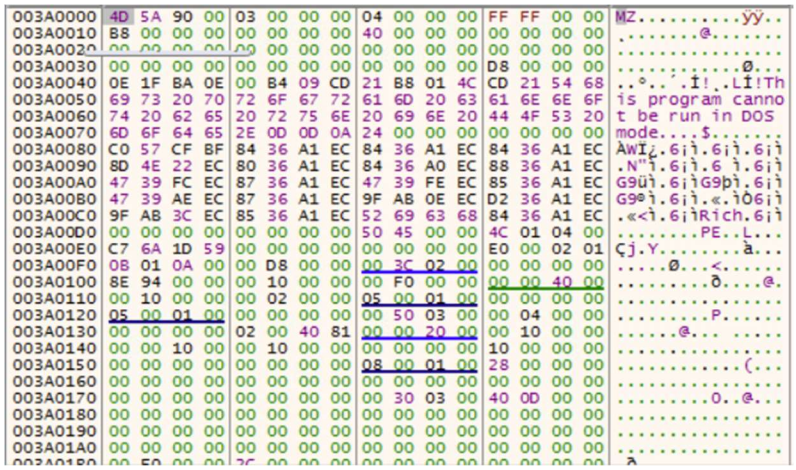


Figure 69: The header of the decoded PE file

We can see an MZ header, and what appears to be a complete program loaded into memory. If we dump this file from memory and load it into PEStudio, we can learn some information from its header contents. However, if we try to decompile it in Ghidra, the decompilation will fail. This is because the program is malformed. In the subsequent function, two additional memory sections are allocated. The first of these new memory sections is used to copy parts of the loaded PE file in a different order. The second of these sections copies from the first, again in a different order. Only the PE file in the last memory section will properly decompile, because the program is in a valid format.

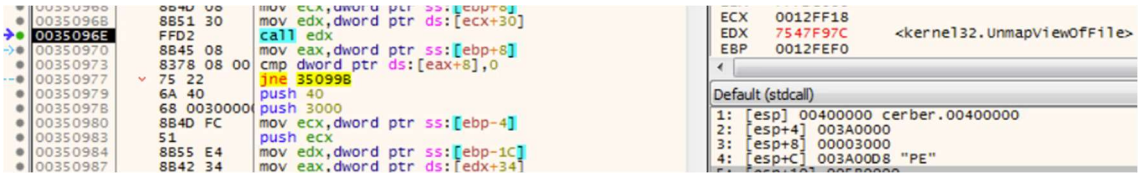


Figure 70: A call to UnmapViewOfFile to unload the original program

Before completing, the shellcode will call UnmapViewOfFile for the base address of the original cerber.exe program (which was found when it called GetModuleHandleW in the cerber.exe code). Immediately after, the valid PE file is loaded into a new memory section created at the original base address.

Section 3D: Debugging the main program

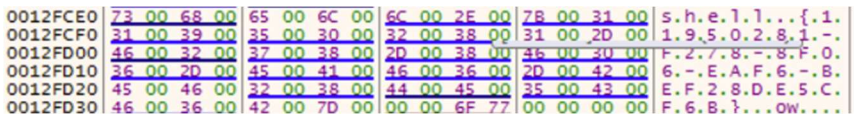


Figure 71: A mutex that was decoded in memory

Once the Windows API functions have been loaded in (which happens in the function in *Figure 28*), a function at 0x40d98b is called. This function decrypts and decodes a lot of miscellaneous data for the function to use. Among this data is the string in the above screenshot.

Figure 72: A call to CreateMutexW for the program

The string we saw loaded into memory previously is used as the mutex for the program. If this mutex is already in use by another process, the program will immediately exit.

Figure 73: A call to CryptImportKey for decryption

Figure 74: A call to CryptEncrypt to decrypt a section of memory

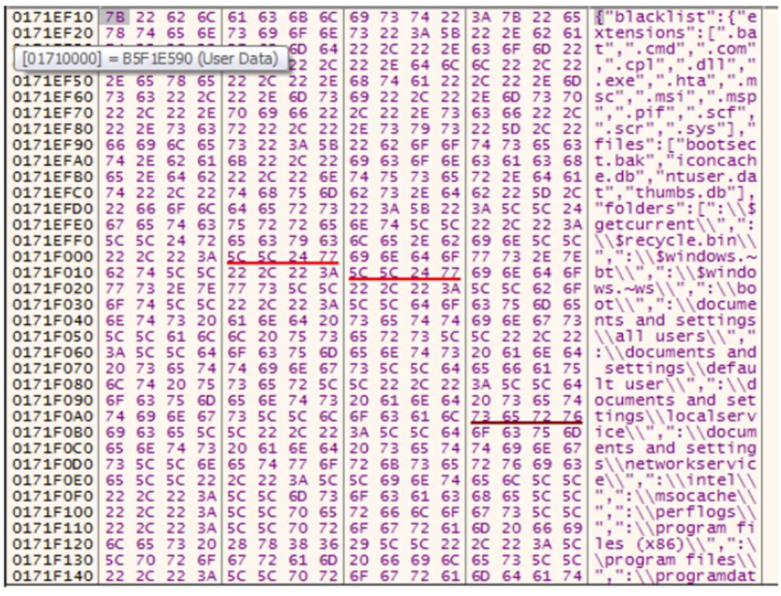


Figure 75: The config file for the program decrypted in memory

In the above three screenshots, we can see the API calls used to import a key and decrypt a region of memory (these API calls are shown in *Figure 30*). Once the memory

region has been decrypted, we can see it contains the configuration file for the malware. This file contains a lot incredibly interesting information, and we will discuss it in the next section.

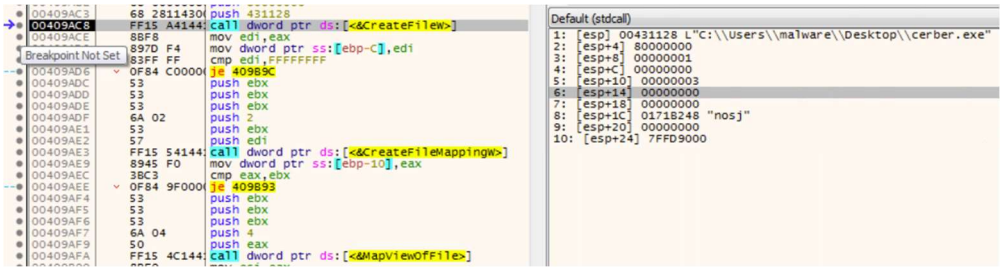


Figure 76: A call to CreateFile by the program on its own code

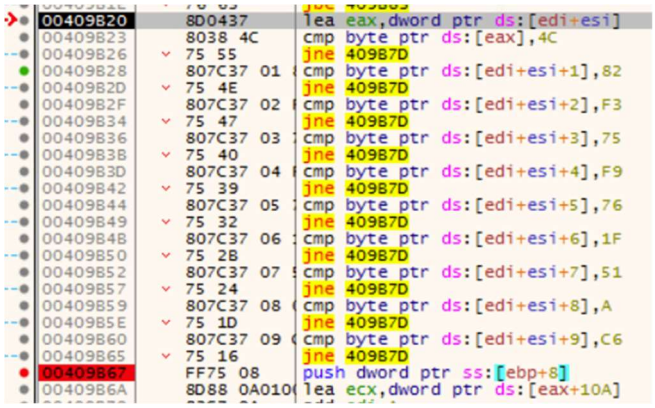


Figure 77: A loop used to search for a particular byte sequence

We mentioned in the discussion of *Figure 31* that the program accesses its own code at one point, and we can verify this above. We can also see that, once the program begins reading its own code, it will look for a specific sequence of bytes: 0x4c82f375f9761f51ac3.

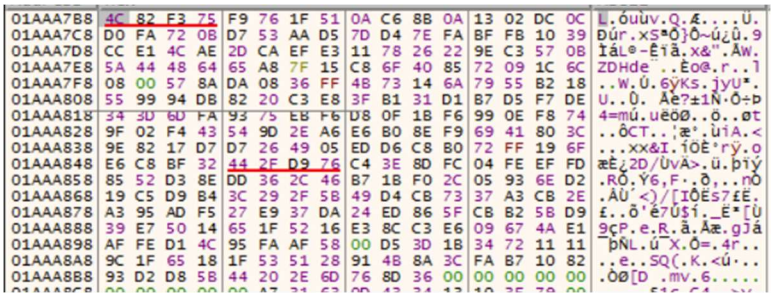


Figure 78: The sequence of code the malware is looking for

As it turns out, this sequence of bytes is the first several bytes of that other allocated file we saw earlier in *Figure 62*. Based on the behaviors of this function, it appears that this collection of bytes contains some sort of instructions for parsing the config file.

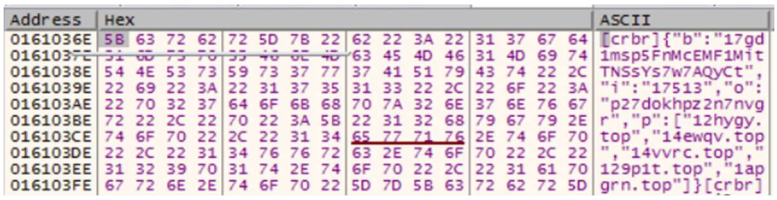


Figure 79: Some information from the configuration file loaded into memory

In this same function, we can see that certain data is loaded into memory using a "[crbr]" tag to denote it. Above we can see some of the URLs for the ransom payment sites.

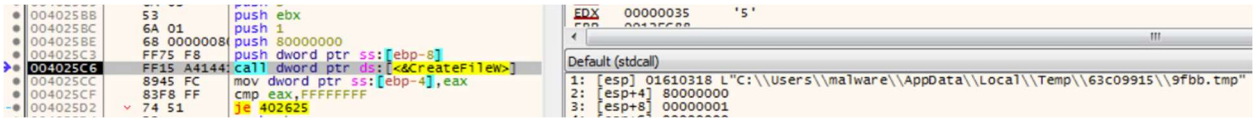


Figure 80: A call to create a new temporary file

While discussing *Figure 32*, we described a function that appears to create some temporary files. We can see above one of the files that is created: "9fbb.tmp". We saw earlier in *Figure 46* that this file appears an RSA key, or some other data replated to the key exchange protocol.

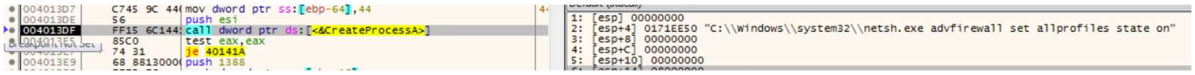


Figure 81: A call to create a process to run "netsh.exe advfirewall set allprofiles state on"

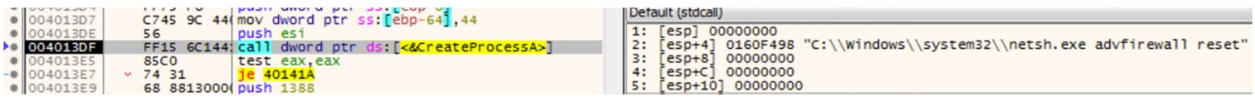


Figure 82: A call to create a process to run "netsh.exe advfirewall reset"

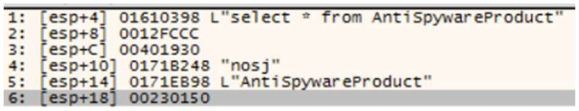


Figure 83: Parameters to a function that appears to be looking for antispyware

In *Figure 34*, we saw code used for creating processes. We can see above some of these processes that are used for performing functionality related to the firewall. We can also see what appears to be WMI queries to learn more about the firewall, and any antivirus products installed on the system.

Figure 84: A call to create a new thread running code stored at 0x40925f

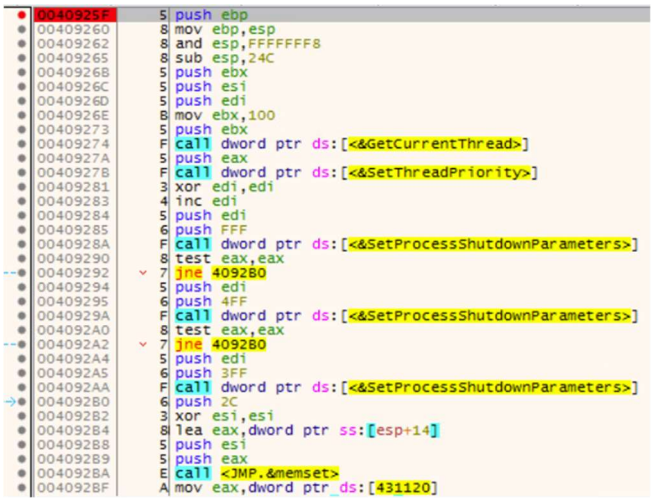


Figure 85: The start of the code being run by the thread

We saw that a new thread is created in the entry function, and using dynamic analysis we can also see where the code is that this thread is running.

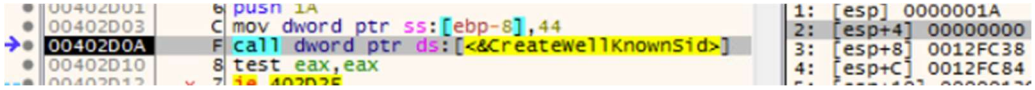


Figure 86: A call used to check if the program has admin permissions before encrypting

Before encrypting the files, the program will run the above Windows API call to check if it is running with admin permissions.

Figure 87: A communication socket opened at port 6893

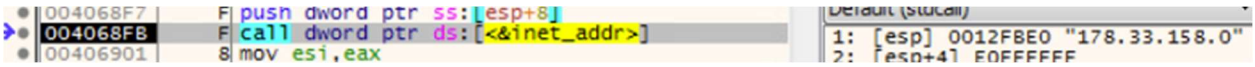


Figure 88: The program getting information for communicating with address 178.33.158.0

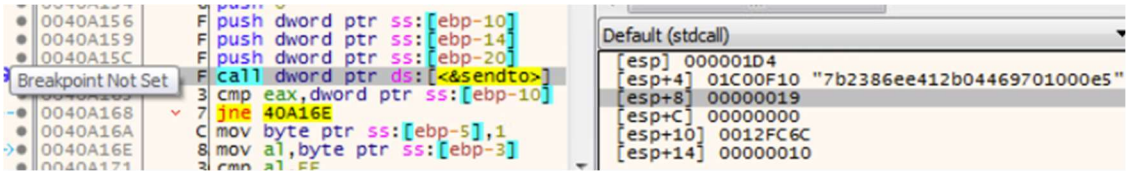


Figure 89: A string of data being sent from the opened socket

The above three functions are used for establishing socket communication. We can see that port 6893 is being used for communication. 178.33.158.0 is the first of many IP addresses the program attempts to communicate with, all in the range mentioned earlier.

The data being sent is created by using the MurmurHash3 algorithm, so this is likely a unique identifier for the infected machine.

Section 3E: The configuration file

In the previous section, we found where the malware's config file is being decrypted in memory. We include screenshots of some of the config file sections in the appendix. Here is a summary of the contents:

- **Blacklisted files:** bootsect.bak, iconcache.db, ntuser.dat, thumbs.db
- **Blacklisted folders:** $recycle.bin, boot, documents and settings\all users, etc.
- **Blacklisted languages:** Russian, Ukranian, Belarusian, Tajik, Armenian, etc.
- **Blacklisted file extensions:** .bat, .cmd, .dll, .exe, .hta, .com, .msi, etc.
- **Debug mode:** Disabled in this version of the malware
- **Ransom payment sites:** tor2web.org, onion.link, onion.cab, etc.
- **Option to enable encryption:** Set to enabled in this version of the malware
- **Encryption parameters:** Bytes to skip (1792), divider (262144)
- **Global public key:** Included in the appendix as a screenshot
- **Encoded files:** These include the data for the .txt and .hta README files
- **Launch README files:** Files are set to automatically open after encrypting files
- **Whitelisted folders:** bitcoin, excel, office, steam, outlook, word, etc.
- **Self-deleting:** Malware is set to delete its source code when it completes
- **Desktop background:** Can change message displayed, color, size, etc.
- **IP addresses:** Subnets 178.33.158.0/27, 178.33.159.0/27 and 178.33.160.0/22
- **Other socket information:** Port 6893, timeout 255

Section 4: Indicators of Compromise and YARA Rule

Host-based indicators:

- Desktop background changes to look like *Figure 44*.
- __R_E_A_D__T_H_I_S__ .txt and .hta files in most folders.
- Temporary files in C:\Users\{username}\AppData\Local\Temp as shown in *Figure 44*, *Figure 46*, and *Figure 47*.

Network-based indicators:

- DNS requests for the following domains: api.blockcypher.com, btc.blockr.io, bitaps.com, and chain.so.
- Communication with the subnets 178.33.158.0/27, 178.33.159.0/27 and 178.33.160.0/22 from port 6893 via TCP

YARA rule:

```
rule cerber
{
        strings:
        $HTMLStr = {31 00 31 00 31 00 31 00 31 00 6B 00 69 00 63 00 75 00 34 00 70
00 33 00 30 00 35 00 30 00 66 00 35 00 35 00 66 00 32 00 39 00 38 00 62 00 35 00 32
 00 31 00 31 00 63 00 66 00 32 00 62 00 62 00 38 00 32 00 32 00 30 00 30 00 61 00 6
1 00 30 00 30 00 62 00 64 00 63 00 65 00 30 00 62 00 66 00}
        $siber = "Siber.Systems.roboform"

        condition:
        $HTMLStr and $siber
}
```

Figure 90: A YARA rule that recognizes cerber.exe

We can use the above YARA rule to identify cerber.exe. The HTMLStr string is the encoded string that we showed in *Figure 56*, and the siber string is from the XML manifest. Together, checking these strings ensures we do not falsely flag any malware samples from the PMA labs.

Appendix

```
"blacklist": {
    "extensions": [
        ".bat",
        ".cmd",
        ".com",
        ".cpl",
        ".dll",
        ".exe",
        ".hta",
        ".msc",
        ".msi",
        ".msp",
        ".pif",
        ".scf",
        ".scr",
        ".sys"
    ],
    "files": [
        "bootsect.bak",
        "iconcache.db",
        "ntuser.dat",
        "thumbs.db"
    ],
    "folders": [
        ":\\$getcurrent\\",
        ":\\$recycle.bin\\",
        ":\\$windows.~bt\\",
        ":\\$windows.~ws\\",
        ":\\boot\\",
        ":\\documents and settings\\all users\\",
        ":\\documents and settings\\default user\\",
        ":\\documents and settings\\localservice\\",
        ":\\documents and settings\\networkservice\\",
        ":\\intel\\",
        ":\\msocache\\",
```

Figure 91: Blacklisted file extensions, files, and folders in the config file

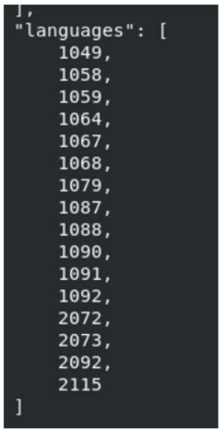


Figure 92: Blacklisted languages in the config file

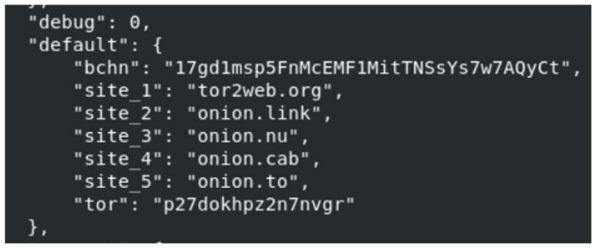


Figure 93: Ransom payment links in the config file

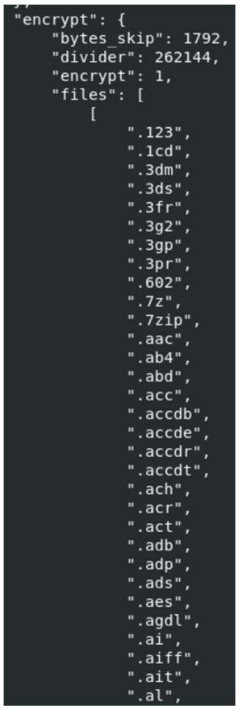


Figure 94: Whitelisted file extensions in the config file

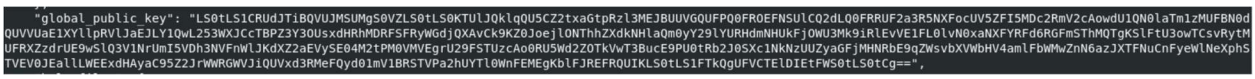
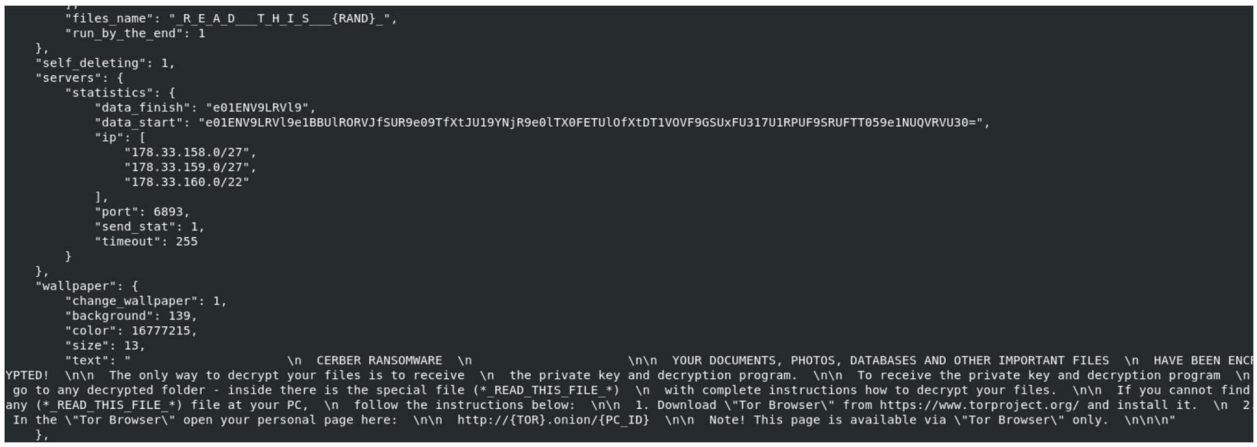


Figure 95: The public key used for key exchange in the config file

```
            "file_body": "IA0KIENFUkJFUiBSQU5TT01XQVJFDQogDQogLS0tLS0NCiANCiBZT1VSIERPQ1VNRU5UUywgUEgwVE9TLCBEQVRBQkFTRVMgQU5EIE9USEVSIElNUE9SVEFOVCBGSUxFUyBIQVZFIEJ
FRU4gRU5DUllQVEVEIQ0KIA0KIC0tLS0tDQogDQogVGhlIG9ubHkgd2F5IHRvIGRlY3J5cHQgeTB1ciBmaWxlcyBpcyB0byByZWNlaXZlIHRoZSBwcml2YXRlIGtleSBhbmQgZGVjcnlwdGlvbiBwcm9ncmFtLg0KIA0KIFRv
IHJlY2VpdmUgdGhlIHByaXZhdGUga2V5IGFuZCBkZWNyeXB0aW9uIHByb2dyYW0gZ28gdG8gYW55IGRlY3J5cHRlZCBmb2xkZXIgLSBpbnNpZGUgaXMgdGhlIHNwZWNpYWwgZmlsZSAoKl9SRUFEX1RISVNfRklMRV8qKSAgXG
UxFXyopIHdpdGggY29tcGxldGUgaW5zdHJlY3Rpb25zIA0KIGhvdyB0byBkZWNyeXB0IHlvdXIgZmlsZXMuICBcbiAgZm9sbG93IHRoZSBpbnN0cnVjdGlvbnMgYmVsb3c6ICAxLiBEb3dubG9hZCBcIlRvciBCcm93c2VyXCIgZnJvbSBodHRwczovL3d3dy50b3Jwcm9qZWN0Lm9yZy8gYW5kIGluc3RhbGwgaXQuICBcbiAgMi
MsIGZvbGxvdyB0aGUgaW5zdHJ1Y3Rpb25zIGJlbG93Og0KIA0KIC0tLS0tDQogDQogMS4gRG93bmxvYWQgIlRvciBCcm93c2VyIiBmcm9tIGh0dHBzOi8vd3d3LnRvcnByb2plY3Qub3JnLyBhbmQgaW5zdGFsbCBpdC4NCiA
NCiAyLiBJbiB0aGUgIlRvciBCcm93c2VyIiBvcGVuIHlvdXIgcGVyc29uYWwgcGFnZSBoZXJlOg0KIA0KIGh0dHA6Ly97VE9SfS5vbmlvbi97UEVfSUR9DQogDQogTm90ZSEgVGhpcyBwYWdlIGlzIGF2YWlsYWJsZSBiaWE
IlRvciBCcm93c2VyIiBvbmx5Lg0KIA0KIC0tLS0tDQogDQogWxzbyB5b3UgY2FuIHVzZS0tZW1wb3JhcnkgYWRkcmVzc2VzIG9uIHlvdXIgcGVyc29uYWwgcGFnZSB3aXRob3V0IHVzaW5nIFRvciBCcm93c2VyIC0tDQogD
QogLS0tLS0NCiANCiAxLiBodHRwOi8ve1RPUn0ue1NJVEVfMX0ve1BDX0lEfQ0KIA0KIDIuIGh0dHA6Ly97VE9SfS57U0lURV8yfS97UENfSUR9DQogDQogLS0tLS0NCiANCiBOb3RlISBUaGVzZSBhcmUgdGVtcG9yYXJ5IGFkZHJlc3NlcyEgVGh
leSB3aWxsIGJlIGF2YWlsYWJsZSBmb3IgYSBsaW1pdGVkIGFtb3VudCBvZiB0aW1lIQ0KIA0KIC0tLS0tDQog",
            "file_extension": ".txt"
```


Figure 96: One of the two files stored in the config file



```
        "files_name": "_R_E_A_D___T_H_I_S___{RAND}_",
        "run_by_the_end": 1
    },
    "self_deleting": 1,
    "servers": {
        "statistics": {
            "data_finish": "e01ENV9LRVl9",
            "data_start": "e01ENV9LRVl9e1BBUlRORVJfSUR9e09TfXtJU19YNjR9e0lTX0FETUlOfXtDT1VOVF9GSUxFU317U1RPUF9SRUFTT059e1NUQVRVU30=",
            "ip": [
                "178.33.158.0/27",
                "178.33.159.0/27",
                "178.33.160.0/22"
            ],
            "port": 6893,
            "send_stat": 1,
            "timeout": 255
        }
    },
    "wallpaper": {
        "change_wallpaper": 1,
        "background": 139,
        "color": 16777215,
        "size": 13,
        "text": "                    \n  CERBER RANSOMWARE  \n                    \n\n  YOUR DOCUMENTS, PHOTOS, DATABASES AND OTHER IMPORTANT FILES  \n  HAVE BEEN ENCR
YPTED!  \n\n  The only way to decrypt your files is to receive  \n  the private key and decryption program.  \n\n  To receive the private key and decryption program  \n
 go to any decrypted folder - inside there is the special file (*_READ_THIS_FILE_*)  \n  with complete instructions how to decrypt your files.  \n\n  If you cannot find
any (*_READ_THIS_FILE_*) file at your PC,  \n  follow the instructions below:  \n\n  1. Download \"Tor Browser\" from https://www.torproject.org/ and install it.  \n  2
 In the \"Tor Browser\" open your personal page here:  \n\n  http://{TOR}.onion/{PC_ID}  \n\n  Note! This page is available via \"Tor Browser\" only.  \n\n\n"
    },
```


Figure 97: Additional information in the config file

```
"whitelist": {
    "folders": [
        "\\bitcoin\\",
        "\\excel\\",
        "\\microsoft sql server\\",
        "\\microsoft\\excel\\",
        "\\microsoft\\microsoft sql server\\",
        "\\microsoft\\office\\",
        "\\microsoft\\onenote\\",
        "\\microsoft\\outlook\\",
        "\\microsoft\\powerpoint\\",
        "\\microsoft\\word\\",
        "\\office\\",
        "\\onenote\\",
        "\\outlook\\",
        "\\powerpoint\\",
        "\\steam\\",
        "\\the bat!\\",
        "\\thunderbird\\",
        "\\word\\"
    ]
}
```


Figure 98: Whitelisted folders in the config file