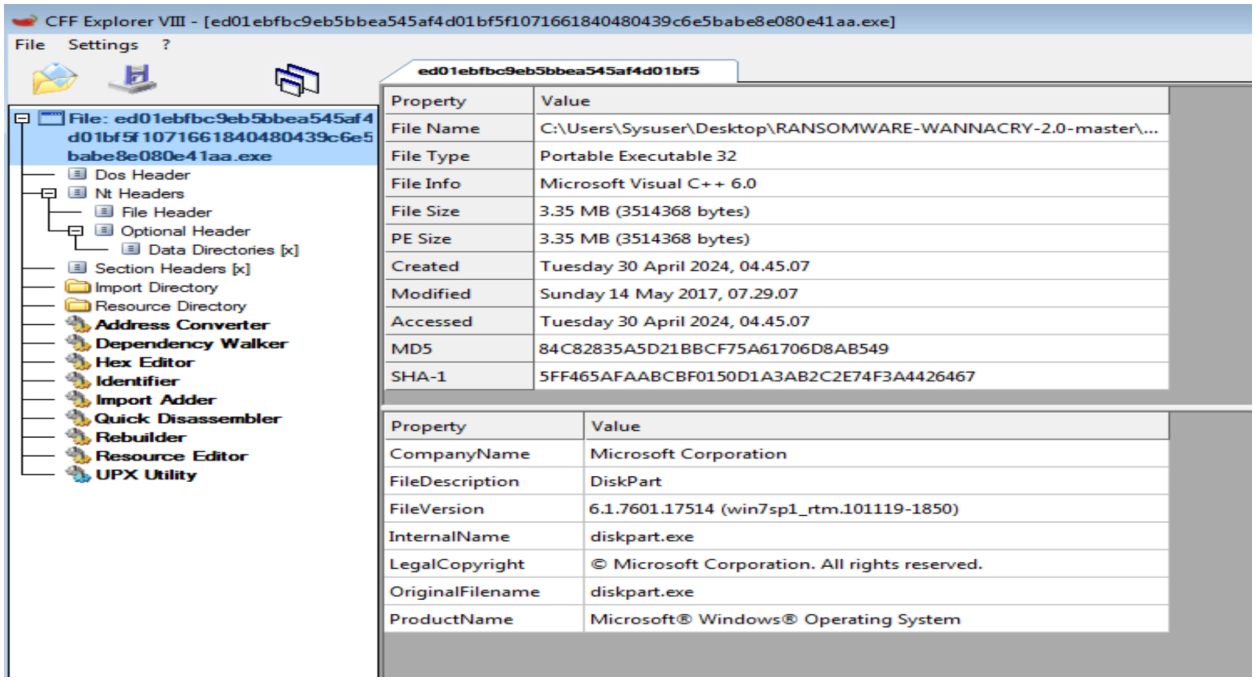
For this CSCE 451 reverse engineering project, I downloaded and analyzed WannaCry, one of the most infamous and destructive pieces of ransomware in recent history. WannaCry was initially created and spread in 2017, affecting networks throughout the world. This paper will explain the process I used to analyze the malware using static and dynamic analysis.

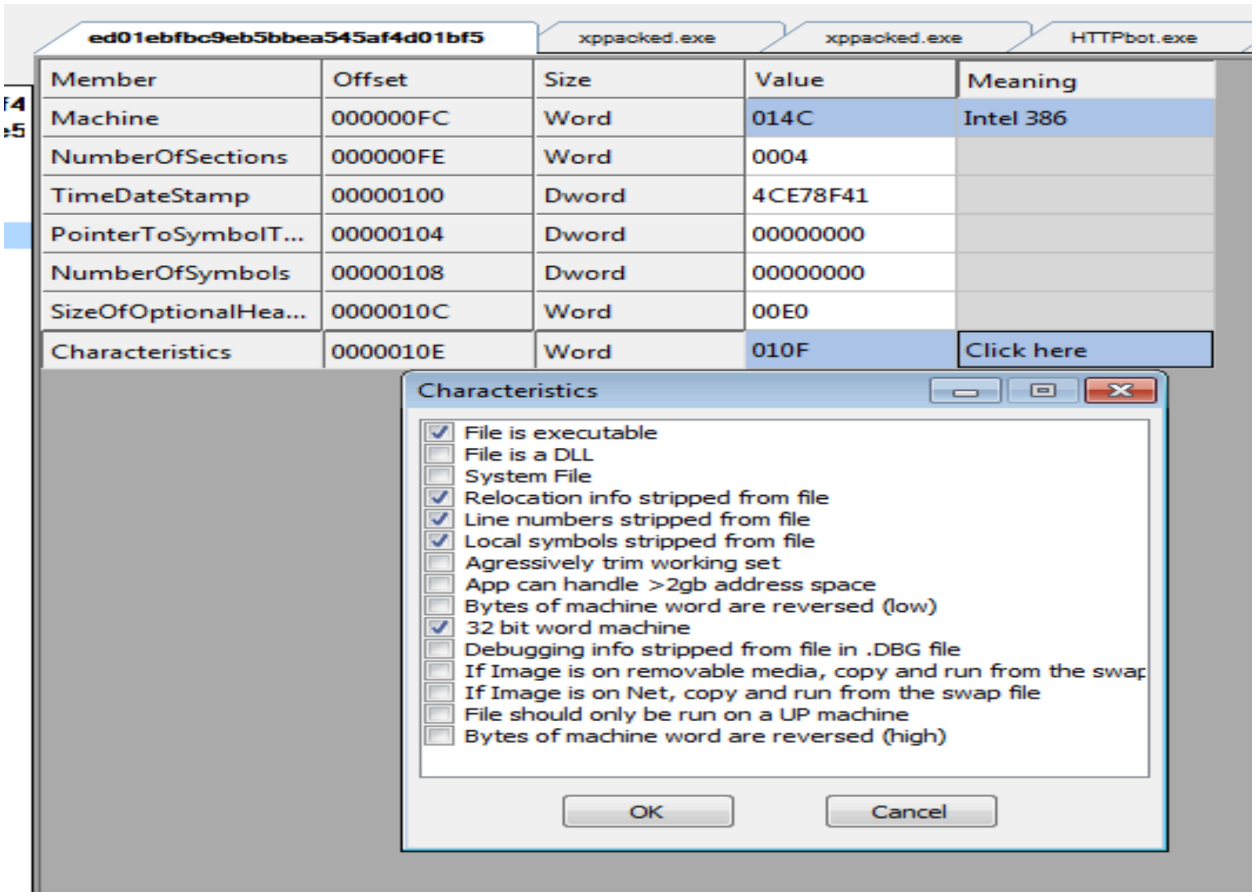
Initial Setup:
The WannaCry sample which I downloaded was from the following URL: https://github.com/chronosmiki/RANSOMWARE-WANNACRY-2.0. I then loaded the .zip file into my VM and unzipped it there, inside which was an executable.

Static Analysis:
First, to check the properties of the file, including whether any packing utility was used to pack or obfuscate the malware, I started with some static analysis using CFF Explorer.



It looks like no packing utility was used that appears in “File Info”. This seems to fit with the intent of a ransomware file like this, as packing a piece of malware usually helps with avoiding detection and making reverse engineering harder, while the primary goal of a file like this is to deny availability and damage file systems quickly, not evade detection for a long period of time.

We can also see that this file was created using C++ in May 2017, which was shortly before it attacked networks worldwide. Additionally, we can see in the lower window that the file was originally called diskpart.exe.



We also see that this is a 32-bit executable.

For further analysis, we will load this file into Ghidra. Immediately on analyzing the file and looking at significant strings, we see several file extension names:

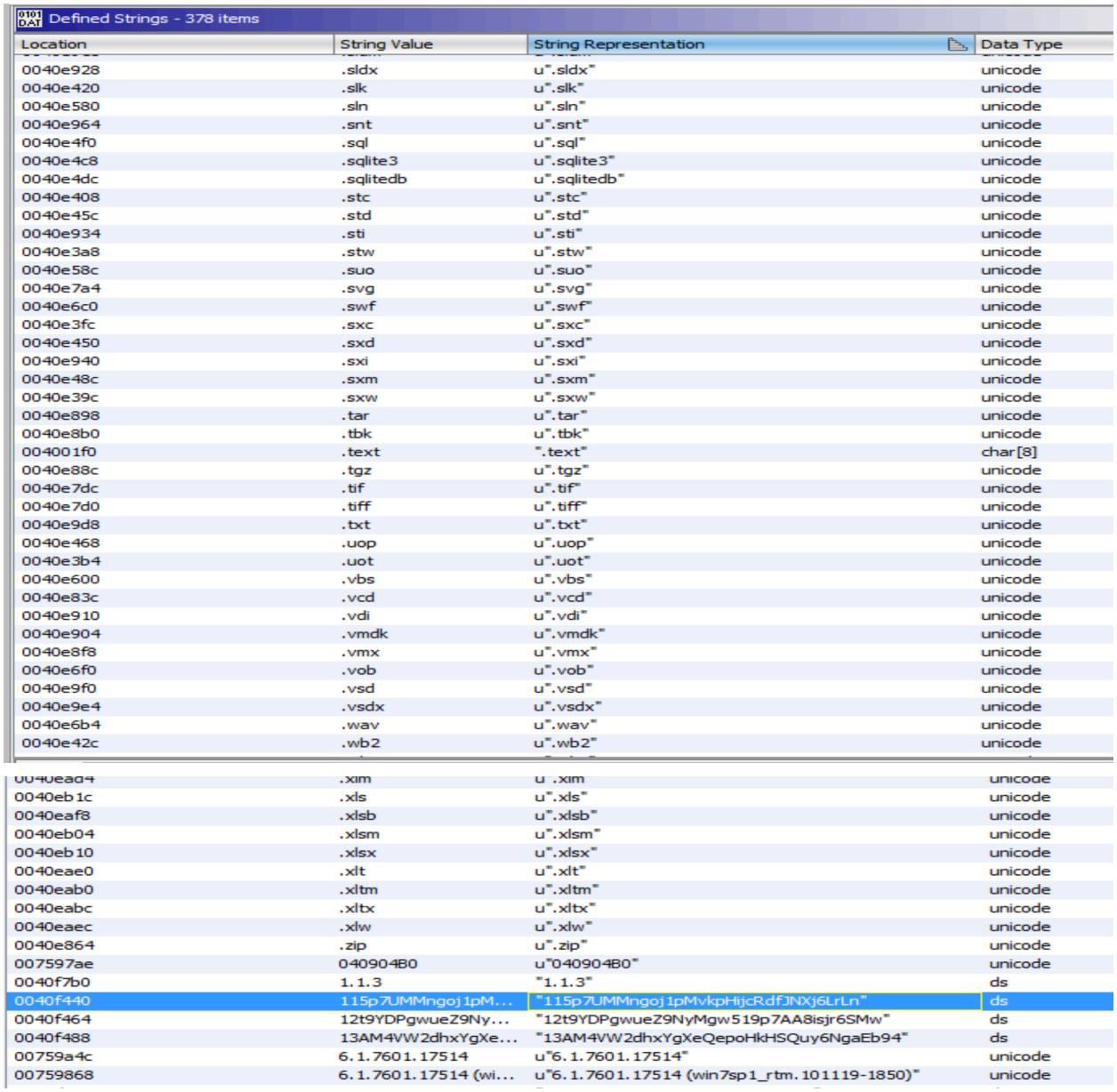
0040d453	- unzip 0.15 Copyrig...	"- unzip 0.15 Copyright 1998 Gilles Vollant "	ds
0040e9b4	.123	u".123"	unicode
0040e3d8	.3dm	u".3dm"	unicode
0040e3c0	.3ds	u".3ds"	unicode
0040e750	.3g2	u".3g2"	unicode
0040e738	.3gp	u".3gp"	unicode
0040e94c	.602	u".602"	unicode
0040f588	.?AVexception@@	".?AVexception@@"	ds
0040f858	.?AVtype_info@@	".?AVtype_info@@"	ds
0040e4fc	.accdb	u".accdb"	unicode
0040e8e0	.aes	u".aes"	unicode
0040e8d4	.ARC	u".ARC"	unicode
0040e4bc	.asc	u".asc"	unicode
0040e708	.asf	u".asf"	unicode
0040e5c8	.asm	u".asm"	unicode
0040e664	.asp	u".asp"	unicode
0040e714	.avi	u".avi"	unicode
0040e854	.backup	u".backup"	unicode
0040e8a4	.bak	u".bak"	unicode
0040e5e8	.bat	u".bat"	unicode
0040e818	.bmp	u".bmp"	unicode
0040e640	.brd	u".brd"	unicode
0040e8bc	.bz2	u".bz2"	unicode
0040e7e8	.cgm	u".cgm"	unicode
0040e690	.class	u".class"	unicode
0040e5dc	.cmd	u".cmd"	unicode
0040e5a8	.cpp	u".cpp"	unicode
0040e354	.crt	u".crt"	unicode
0040e360	.csr	u".csr"	unicode
0040e9cc	.csv	u".csv"	unicode
00400240	.data	".data"	char[8]
0040e520	.dbf	u".dbf"	unicode
0040e628	.dch	u".dch"	unicode
0040e330	.der	u".der"	unicode
0040e414	.dif	u".dif"	unicode
0040e61c	.dip	u".dip"	unicode
0040e798	.djvu	u".djvu"	unicode
0040eb70	.doc	u".doc"	unicode
0040eb58	.doch	u".doch"	unicode

Defined Strings - 378 items

Location	String Value	String Representation	Data Type
0040e984	.dwg	u".dwg"	unicode
0040e9fc	.edb	u".edb"	unicode
0040ea08	.eml	u".eml"	unicode
0040e6cc	.fla	u".fla"	unicode
0040e75c	.flv	u".flv"	unicode
0040e538	.frm	u".frm"	unicode
0040e800	.gif	u".gif"	unicode
0040e8ec	.gpg	u".gpg"	unicode
0040e958	.hwp	u".hwp"	unicode
0040e55c	.ibd	u".ibd"	unicode
0040e848	.iso	u".iso"	unicode
0040e684	.jar	u".jar"	unicode
0040e678	.java	u".java"	unicode
0040e830	.jpeg	u".jpeg"	unicode
0040e824	.jpg	u".jpg"	unicode
0040e64c	.jsp	u".jsp"	unicode
0040e348	.key	u".key"	unicode
0040e4a4	.lay	u".lay"	unicode
0040e4b0	.lay6	u".lay6"	unicode
0040e574	.ldf	u".ldf"	unicode
0040e780	.m3u	u".m3u"	unicode
0040e78c	.m4u	u".m4u"	unicode
0040e3cc	.max	u".max"	unicode
0040e50c	.mdb	u".mdb"	unicode
0040e568	.mdf	u".mdf"	unicode
0040e774	.mid	u".mid"	unicode
0040e744	.mkv	u".mkv"	unicode
0040e498	.mml	u".mml"	unicode
0040e720	.mov	u".mov"	unicode
0040e6a8	.mp3	u".mp3"	unicode
0040e72c	.mp4	u".mp4"	unicode
0040e6fc	.mpeg	u".mpeg"	unicode
0040e6e4	.mpg	u".mpg"	unicode
0040ea14	.msg	u".msg"	unicode
0040e544	.myd	u".myd"	unicode
0040e550	.myi	u".myi"	unicode
0040e7c4	.nef	u".nef"	unicode
0040e52c	.odb	u".odb"	unicode

Defined Strings - 378 items

Location	String Value	String Representation	Data Type
0040e474	.odg	u".odg"	unicode
0040e438	.odp	u".odp"	unicode
0040e3e4	.ods	u".ods"	unicode
0040e384	.odt	u".odt"	unicode
0040e970	.onetoc2	u".onetoc2"	unicode
0040ea20	.ost	u".ost"	unicode
0040e480	.otg	u".otg"	unicode
0040e444	.otp	u".otp"	unicode
0040e3f0	.ots	u".ots"	unicode
0040e390	.ott	u".ott"	unicode
0040e36c	.p12	u".p12"	unicode
0040e8c8	.PAQ	u".PAQ"	unicode
0040e5b4	.pas	u".pas"	unicode
0040e990	.pdf	u".pdf"	unicode
0040e378	.pem	u".pem"	unicode
0040e33c	.pfx	u".pfx"	unicode
0040e658	.php	u".php"	unicode
0040e80c	.png	u".png"	unicode
0040ea80	.pot	u".pot"	unicode
0040ea38	.potm	u".potm"	unicode
0040ea44	.potx	u".potx"	unicode
0040ea50	.ppam	u".ppam"	unicode
0040ea74	.pps	u".pps"	unicode
0040ea68	.ppsm	u".ppsm"	unicode
0040ea5c	.ppsx	u".ppsx"	unicode
0040eaa4	.ppt	u".ppt"	unicode
0040ea8c	.pptm	u".pptm"	unicode
0040ea98	.pptx	u".pptx"	unicode
0040e5f4	.ps1	u".ps1"	unicode
0040e7b8	.psd	u".psd"	unicode
0040ea2c	.pst	u".pst"	unicode
0040e870	.rar	u".rar"	unicode
0040e7f4	.raw	u".raw"	unicode
00400218	.rdata	".rdata"	char[8]
00400268	.rsrc	".rsrc"	char[8]
0040e9c0	.rtf	u".rtf"	unicode
0040e634	.sch	u".sch"	unicode
0040e91c	.sldm	u".sldm"	unicode
0040e928	.sldx	u".sldx"	unicode

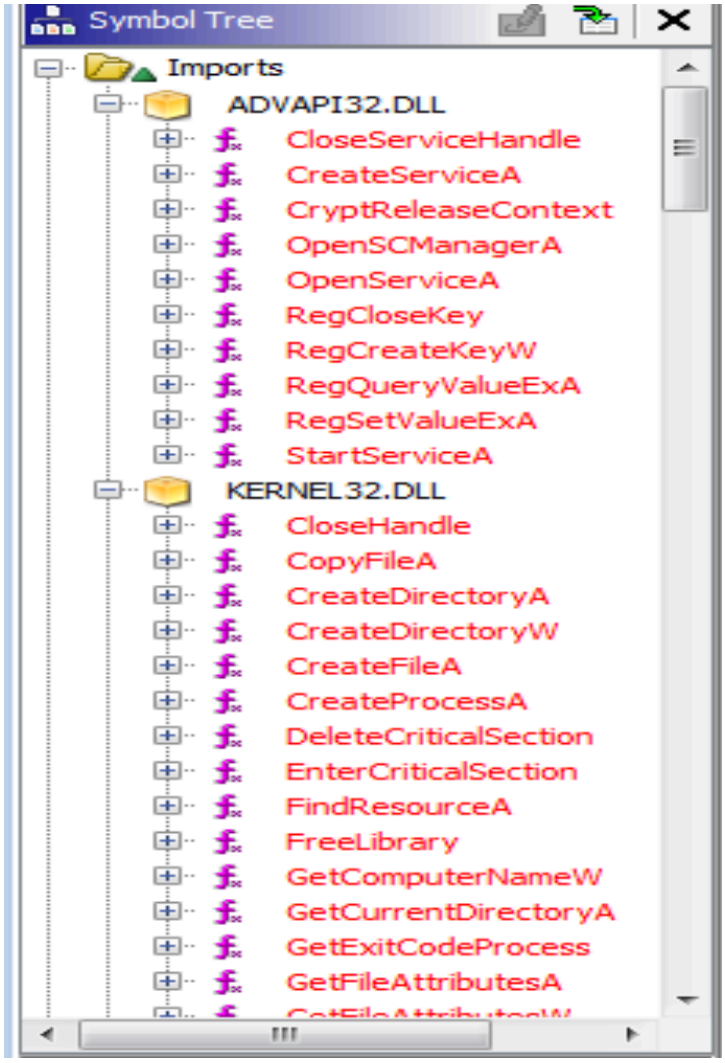


Defined Strings - 378 items

Location	String Value	String Representation	Data Type
0040e928	.sldx	u".sldx"	unicode
0040e420	.slk	u".slk"	unicode
0040e580	.sln	u".sln"	unicode
0040e964	.snt	u".snt"	unicode
0040e4f0	.sql	u".sql"	unicode
0040e4c8	.sqlite3	u".sqlite3"	unicode
0040e4dc	.sqlitedb	u".sqlitedb"	unicode
0040e408	.stc	u".stc"	unicode
0040e45c	.std	u".std"	unicode
0040e934	.sti	u".sti"	unicode
0040e3a8	.stw	u".stw"	unicode
0040e58c	.suo	u".suo"	unicode
0040e7a4	.svg	u".svg"	unicode
0040e6c0	.swf	u".swf"	unicode
0040e3fc	.sxc	u".sxc"	unicode
0040e450	.sxd	u".sxd"	unicode
0040e940	.sxi	u".sxi"	unicode
0040e48c	.sxm	u".sxm"	unicode
0040e39c	.sxw	u".sxw"	unicode
0040e898	.tar	u".tar"	unicode
0040e8b0	.tbk	u".tbk"	unicode
004001f0	.text	".text"	char[8]
0040e88c	.tgz	u".tgz"	unicode
0040e7dc	.tif	u".tif"	unicode
0040e7d0	.tiff	u".tiff"	unicode
0040e9d8	.txt	u".txt"	unicode
0040e468	.uop	u".uop"	unicode
0040e3b4	.uot	u".uot"	unicode
0040e600	.vbs	u".vbs"	unicode
0040e83c	.vcd	u".vcd"	unicode
0040e910	.vdi	u".vdi"	unicode
0040e904	.vmdk	u".vmdk"	unicode
0040e8f8	.vmx	u".vmx"	unicode
0040e6f0	.vob	u".vob"	unicode
0040e9f0	.vsd	u".vsd"	unicode
0040e9e4	.vsdx	u".vsdx"	unicode
0040e6b4	.wav	u".wav"	unicode
0040e42c	.wb2	u".wb2"	unicode
0040eac4	.xlm	u".xlm"	unicode
0040eb1c	.xls	u".xls"	unicode
0040eaf8	.xlsb	u".xlsb"	unicode
0040eb04	.xlsm	u".xlsm"	unicode
0040eb10	.xlsx	u".xlsx"	unicode
0040eae0	.xlt	u".xlt"	unicode
0040eab0	.xltm	u".xltm"	unicode
0040eabc	.xltx	u".xltx"	unicode
0040eaec	.xlw	u".xlw"	unicode
0040e864	.zip	u".zip"	unicode
007597ae	040904B0	u"040904B0"	unicode
0040f7b0	1.1.3	"1.1.3"	ds
0040f440	115p7UMMngoj1pM...	"115p7UMMngoj1pMvkpHijcRdfJNXj6LrLn"	ds
0040f464	12t9YDPgwueZ9Ny...	"12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw"	ds
0040f488	13AM4VW2dhxYgXe...	"13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94"	ds
00759a4c	6.1.7601.17514	u"6.1.7601.17514"	unicode
00759868	6.1.7601.17514 (wi...	u"6.1.7601.17514 (win7sp1_rtm.101119-1850)"	unicode

Given that WannaCry is a ransomware program, the several file extensions noted here could be the file types the malware targets. Interestingly, the strings ".exe" and ".dll" are not seen here, which suggests that WannaCry isn't encrypting executables and system files, as is the norm for ransomware. This could be useful later if we need to execute a decryption tool.

Looking at the imports on the left-hand side, we see RegCloseKey, RegCreateKey, RegSetValue, etc, which imply that this malware likely is making edits to the registry keys, as well as CreateFileA to create new files. Additionally, we see CryptReleaseContext, which is a method for using Cryptographic Service Providers (CSP). This is another common feature of ransomware, as the payment is usually demanded in cryptocurrency.

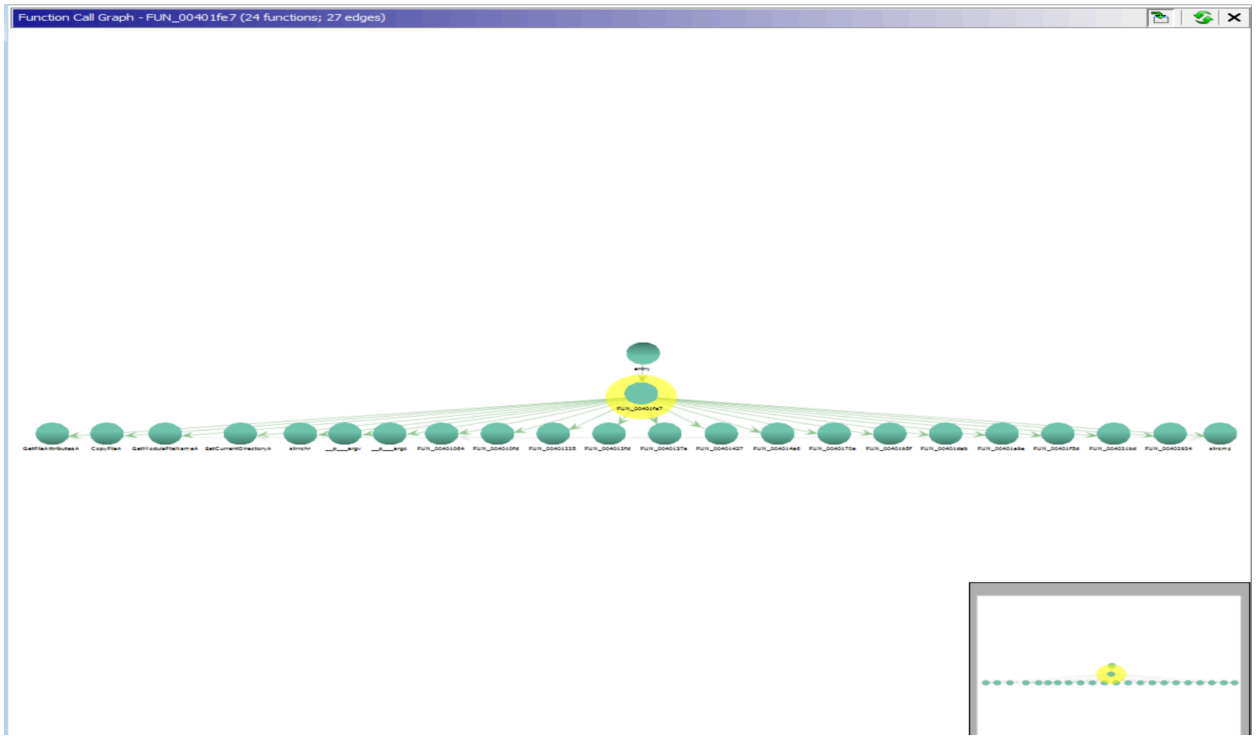


Looking at the functions, we see several unnamed functions with different input arguments:

Functions - 132 items

Label	...	Function Signature	Function Size
FUN_00401000	004...	uint FUN_0040100...	100
FUN_00401064	004...	undefined4 FUN_0...	153
FUN_004010fd	004...	undefined4 FUN_0...	296
FUN_00401225	004...	undefined FUN_00...	216
FUN_004012fd	004...	undefined4 * FUN...	97
FUN_0040135e	004...	void * FUN_00401...	28
FUN_0040137a	004...	undefined FUN_00...	84
FUN_004013ce	004...	undefined4 FUN_0...	105
FUN_00401437	004...	undefined4 FUN_0...	111
FUN_004014a6	004...	byte * FUN_00401...	588
FUN_0040170a	004...	undefined4 FUN_0...	211
FUN_004017dd	004...	undefined4 * FUN...	34
FUN_004017ff	004...	undefined4 * FUN...	28
FUN_0040181b	004...	undefined FUN_00...	17
FUN_0040182c	004...	undefined4 FUN_0...	53
FUN_00401861	004...	undefined4 FUN_0...	88
FUN_004018b9	004...	undefined4 FUN_0...	64
FUN_004018f9	004...	undefined4 FUN_0...	199
FUN_004019e1	004...	undefined4 FUN_0...	100
FUN_00401a45	004...	undefined4 FUN_0...	177
FUN_00401af6	004...	undefined4 FUN_0...	105
FUN_00401b5f	004...	uint FUN_00401b5...	393
FUN_00401ce8	004...	undefined4 FUN_0...	195
FUN_00401dab	004...	undefined4 FUN_0...	243
FUN_00401e9e	004...	undefined FUN_00...	97
FUN_00401eff	004...	undefined4 FUN_0...	94
FUN_00401f5d	004...	undefined4 FUN_0...	138
FUN_00401fe7	004...	undefined4 FUN_0...	391
FUN_004021bd	004...	undefined FUN_00...	44
FUN_004021e9	004...	uint * FUN_00402...	622
FUN_00402457	004...	undefined4 FUN_0...	25
FUN_00402470	004...	undefined4 FUN_0...	219
FUN_0040254b	004...	uint FUN_0040254...	260
FUN_0040264f	004...	int FUN_0040264f...	44
FUN_0040267b	004...	uint FUN_0040267...	162
FUN_0040271d	004...	undefined4 FUN_0...	59
FUN_00402758	004...	uint FUN_0040275...	135
FUN_004027df	004...	int FUN_004027df...	325
FUN_00402924	004...	int FUN_00402924...	168

Viewing the function tree to get a clearer view of how these functions interact with each other:



The entry function calls FUN_104fe7, which then calls all the other functions in the malware. This can be seen near the end of “entry”:

```
LAB_004078ad:
  } while ((*local_78 != 0) && (*local_78 < 0x21));
  local_60.dwFlags = 0;
  GetStartupInfoA((LPSTARTUPINFOA)&local_60);
  GetModuleHandleA((LPCSTR)0x0);
  local_6c = FUN_00401fe7();
                    /* WARNING: Subroutine does not return */
  exit(local_6c);
}
```

When we navigate to the FUN_00401fe7 function, we see several more function calls within this method, as expected from all the different functions called by FUN_00401fe7.

```
SetCurrentDirectoryA(&local_210);
FUN_004010fd(1);

*(undefined2 *)puVar4 = 0;
*(undefined *)((int)puVar4 + 2) = 0;
wcscat((wchar_t *)local_d8,(wchar_t *)&_Source_0040e034);
local_c = 0;
do {
  if (local_c == 0) {
    hKey = (HKEY)0x80000002;
  }
  else {
    hKey = (HKEY)0x80000001;
  }
  RegCreateKeyW(hKey,(LPCWSTR)local_d8,(PHKEY)&local_8);
  if (local_8 != (HKEY)0x0) {
    if (param_1 == 0) {
      local_10 = 0x207;
      LVar2 = RegQueryValueExA(local_8,&DAT_0040e030,(LPDWORD)0x0,(LPDW
                               &local_10);
      bVar6 = LVar2 == 0;
      if (bVar6) {
        SetCurrentDirectoryA((LPCSTR)&local_2e0);
      }
    }
    else {
      GetCurrentDirectoryA(0x207,(LPSTR)&local_2e0);
      sVar1 = strlen((char *)&local_2e0);
      LVar2 = RegSetValueExA(local_8,&DAT_0040e030,0,1,&local_2e0,sVar1
      bVar6 = LVar2 == 0;
    }
```

Looking at the first function call, it takes in an argument 1, and handles several registry edits, including creating keys and setting them to new values. However, the actual values of the registry keys and values created are not directly visible, so I used dynamic analysis with x32dbg to find that data, as will be shown later.

Looking further down in the function, we also find another method FUN_00401e9e to assign one of 3 randomly generated cryptocurrency tokens (as shown earlier alongside the rest of the extracted strings). The assembly instructions for the method are shown below first, followed by a reconstruction of the source code with variable names and comments.

```
                    FUN_00401e9e                                    XREF[1]:     FUN
00401e9e 55              PUSH        EBP
00401e9f 8b ec           MOV         EBP,ESP
00401ea1 81 ec 18        SUB         ESP,0x318
         03 00 00
00401ea7 8d 85 e8        LEA         EAX=>local_31c,[0xfffffce8 + EBP]
         fc ff ff
00401ead 6a 01           PUSH        0x1
00401eaf 50              PUSH        EAX
00401eb0 c7 45 f4        MOV         dword ptr [EBP + token_array[0]],s_13AM4VW2dhx...
         88 f4 40 00
00401eb7 c7 45 f8        MOV         dword ptr [EBP + token_array[1]],s_12t9YDPgwue...
         64 f4 40 00
00401ebe c7 45 fc        MOV         dword ptr [EBP + token_array[2]],s_115p7UMMngo...
         40 f4 40 00
00401ec5 e8 36 f1        CALL        File_ReadWrite
         ff ff
00401eca 59              POP         ECX
00401ecb 85 c0           TEST        EAX,EAX
00401ecd 59              POP         ECX
00401ece 74 2d           JZ          LAB_00401efd
00401ed0 ff 15 20        CALL        dword ptr [->MSVCRT.DLL::rand]
         81 40 00
00401ed6 6a 03           PUSH        0x3
00401ed8 99              CDQ
00401ed9 59              POP         ECX
00401eda f7 f9           IDIV        ECX
00401edc 8d 85 9a        LEA         randomNum=>local_26a,[0xfffffd9a + EBP]
         fd ff ff
00401ee2 ff 74 95 f4     PUSH        dword ptr [EBP + EDX*0x4 + -0xc]
00401ee6 50              PUSH        randomNum
00401ee7 e8 bc 57        CALL        strcpy
         00 00
00401eec 8d 85 e8        LEA         randomNum=>local_31c,[0xfffffce8 + EBP]
         fc ff ff
00401ef2 6a 00           PUSH        0x0
00401ef4 50              PUSH        randomNum
00401ef5 e8 06 f1        CALL        File_ReadWrite
         ff ff
00401efa 83 c4 10        ADD         ESP,0x10

                    LAB_00401efd
00401efd c9              LEAVE
00401efe c3              RET
```

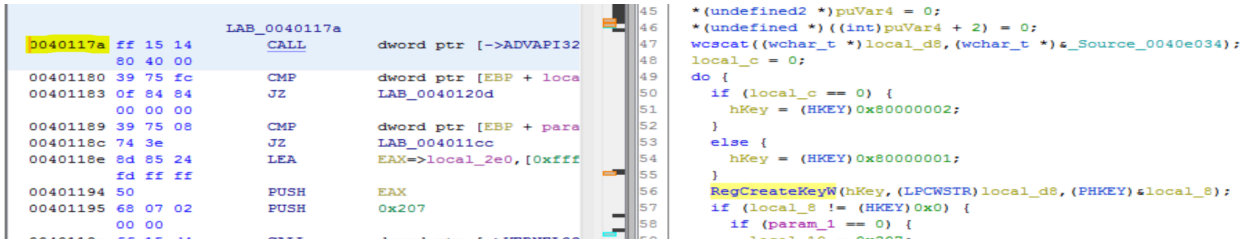
Using the decompiled output in Ghidra as a template, I reconstructed the source code in C syntax:

```
1
2  void FUN_00401e9e(void)
3
4  {
5    uint fileInput;
6    int randomNum;
7    undefined fileOutput [178];
8    char new_token [602];
9    char *token_array [3];
10
11   token_array[0] = s_13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb_0040f488;
12   token_array[1] = s_12t9YDPgwueZ9NyMgw519p7AA8isjr6S_0040f464;
13   token_array[2] = s_115p7UMMngoj1pMvkpHijcRdfJNXj6Lr_0040f440;
14   fileInput = File_ReadWrite(fileOutput,1);
15   if (fileInput != 0) {
16     randomNum = rand();
17     strcpy(new_token,token_array[randomNum % 3]);
18     File_ReadWrite(fileOutput,0);
19   }
20   return;
21 }
22
```

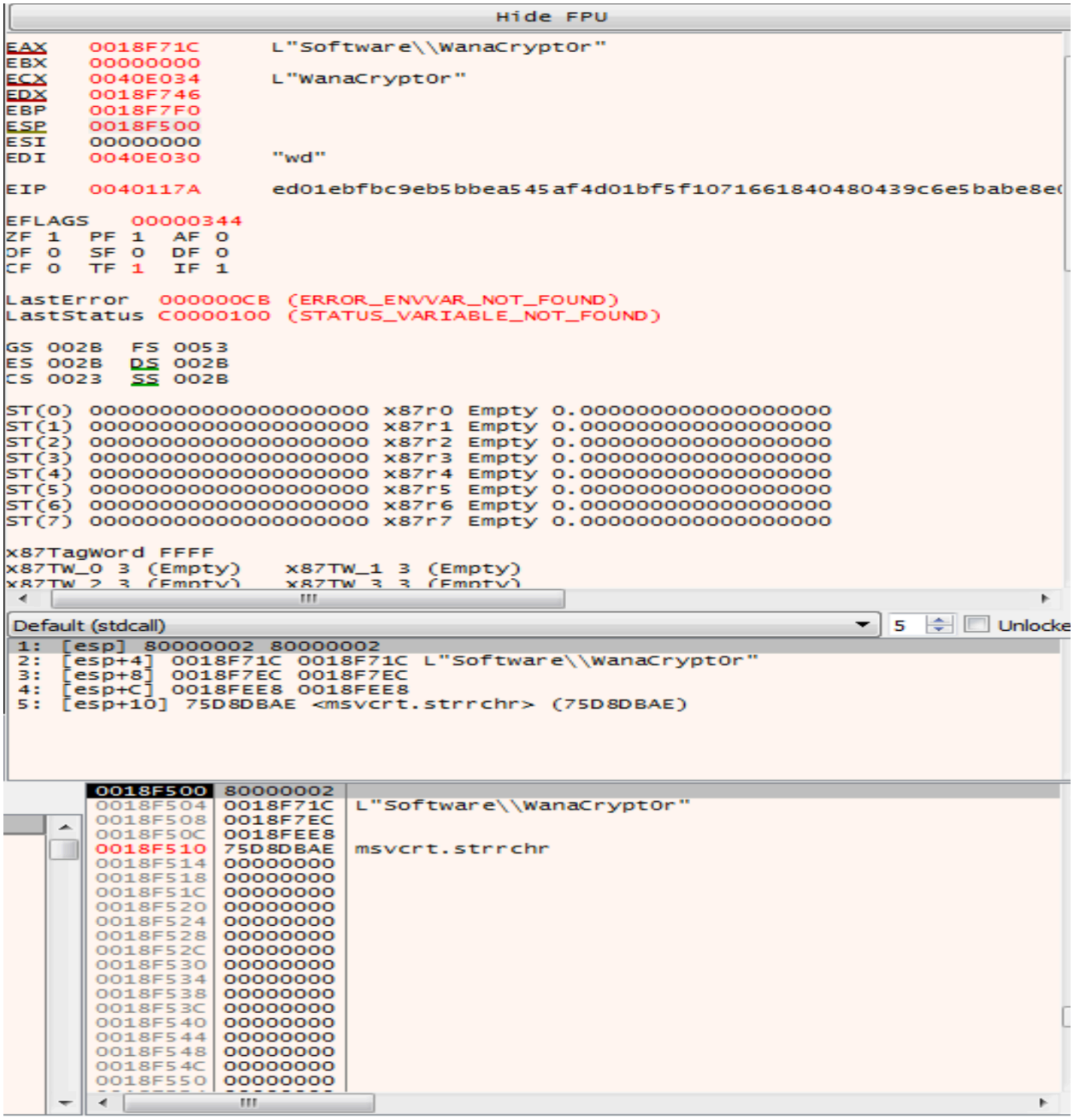
```
void FUN_00401e9e(void)
{
    uint fileInput;
    int randomNum;
    undefined fileOutput [178];
    char new_token [602]; // the token that will be chosen
    char *token_array [3]; // array that will take in 3 cryptographic tokens

    token_array[0] = s_13AM4WW2dhxYgXeQepoHkHSQuy6NgaEb_0040f488;
    token_array[1] = s_12t9YDPgwueZ9NyMgw519p7ZA8isj6r6S_0040f464;
    token_array[2] = s_11p5p7UMMngo1jPMvkrpHijcRdEJUNXj6Lr_0040f440;
    fileInput = File_ReadWrite(fileOutput, 1); // fileOutput read from c.wnry
    //(as defined in File_ReadWrite, when arg2 is 1, read into arg1),
    //then passed to fileInput
    if (fileInput != 0) { // checks to make sure c.wnry was read properly
        // random number is generated, then reduced mod 3 to generate some number between 0 and 2
        // then that number is passed as an index for which token is picked
        randomNum = rand();
        strcpy(new_token, token_array[randomNum % 3]);
        File_ReadWrite(fileOutput, 0); // file written back to c.wnry (argument 0 = write)
    }
    return;
}
```

Debugging:

Looking at the registry edits found in the function earlier, we can now try to find the actual registry keys and values that are called with dynamic analysis. I loaded the file into x32dbg, and found the address at which the value was being set:

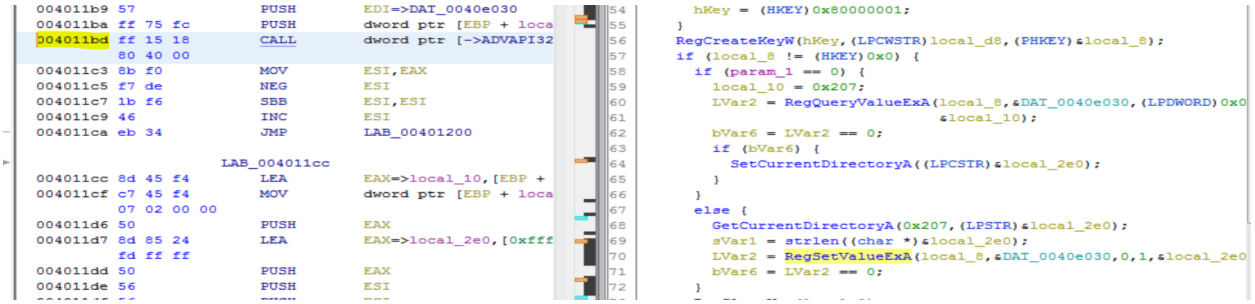


Then, I set a breakpoint at that address and ran the code, and the registry key was then shown in the EAX register:

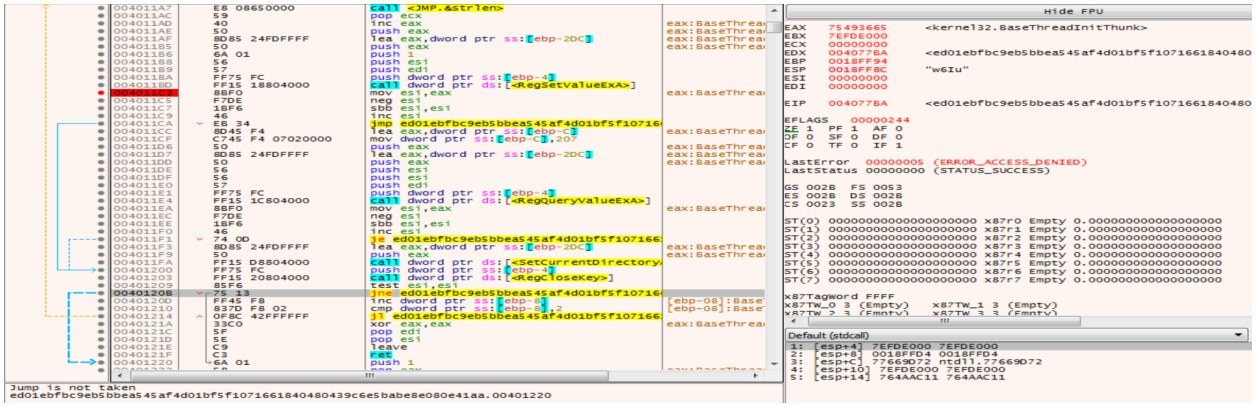


Looking in RegEdit, there is a directory called “Software”, and based on this method, it looks like the malware when run will create a new directory called Software\\WanaCryptor as the registry key.

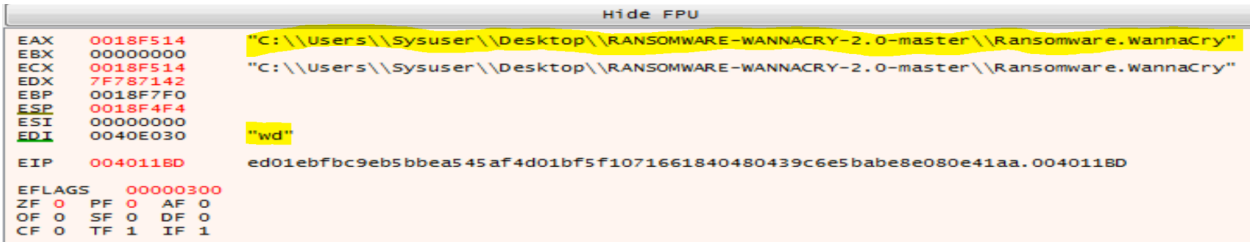
Next, to find the registry value itself, I ran it with a breakpoint at the address at which RegSetValue was called:



First I tried running with a breakpoint set for the instruction after the call, but no registry values were shown in the registers.



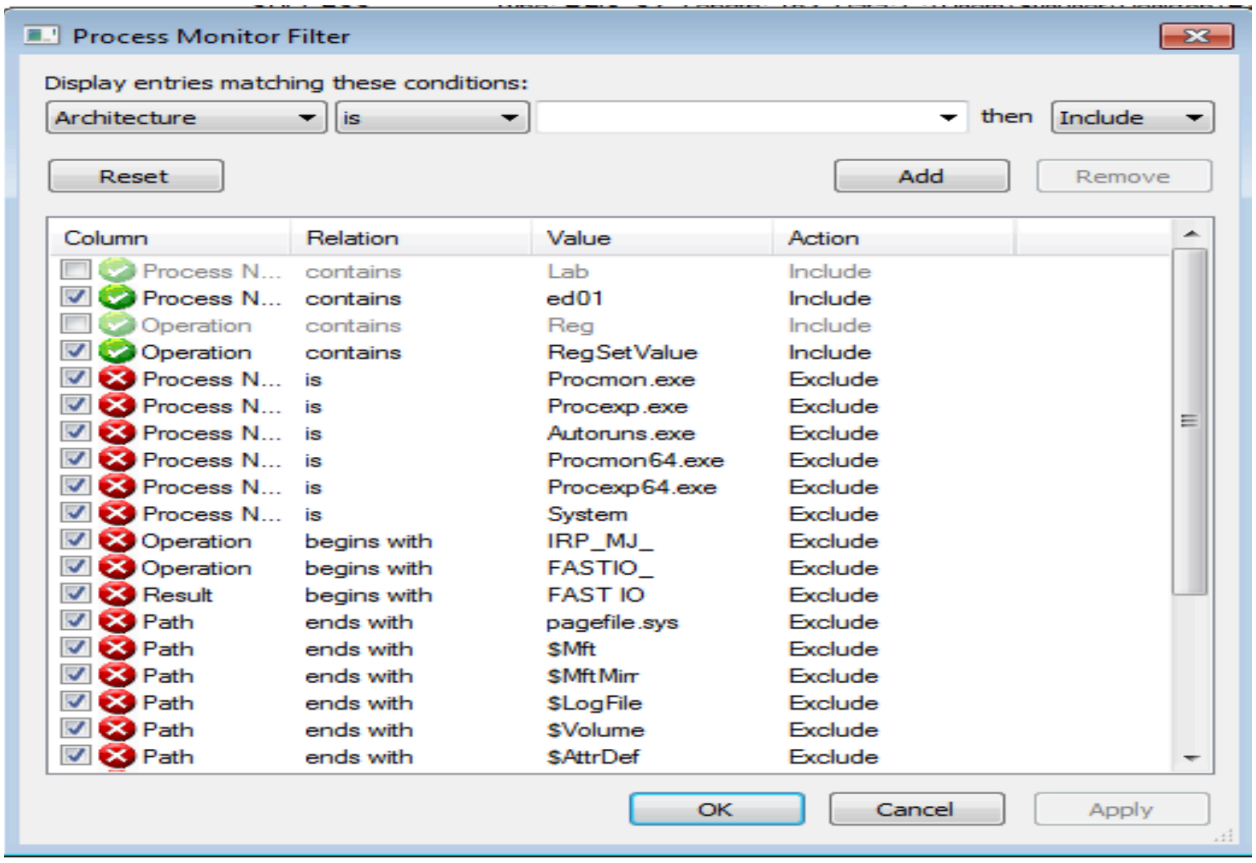
I restarted the debugger and ran it again, setting the code to break at the function call for RegSetValueExA, and this time received more useful output.



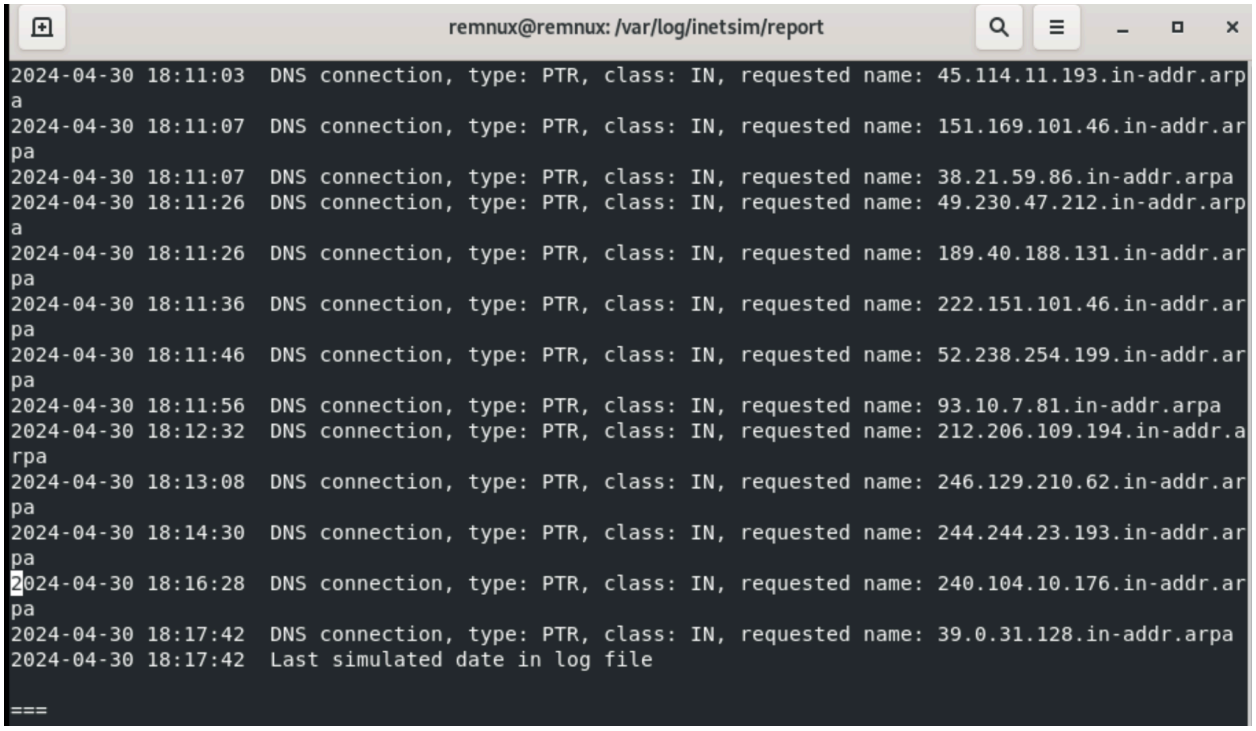
Checking the EDI register, it looks like “wd” is our registry value.

Running the Malware

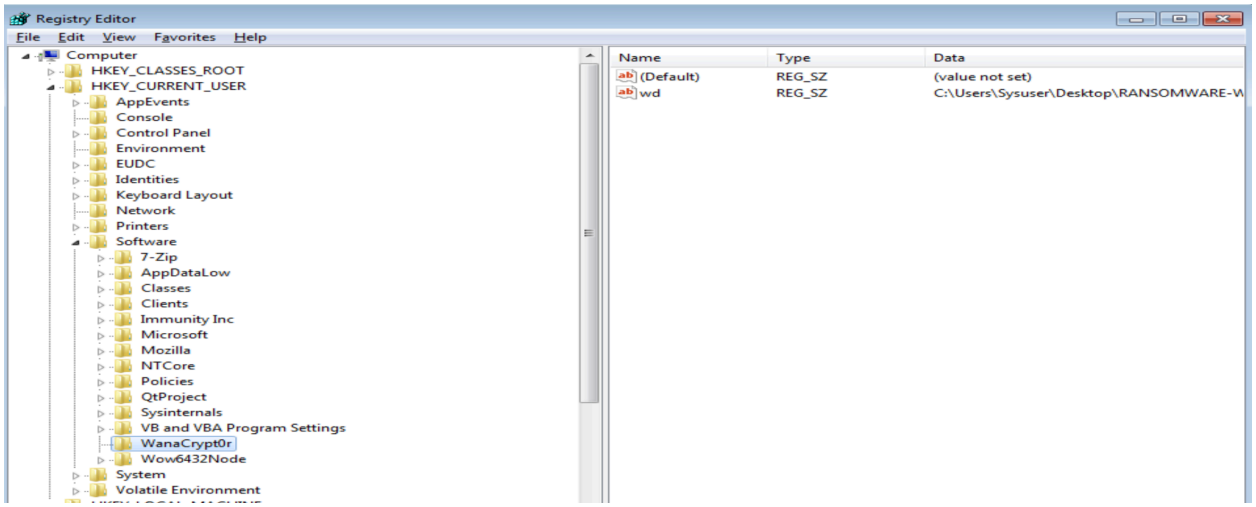
After checking my VM settings to make sure it was not connected to the internet and had no possible attack vectors to the host, I set up inetsim in a remnux machine, opened Wireshark, and opened ProcMon to run the malware, setting the following filters:



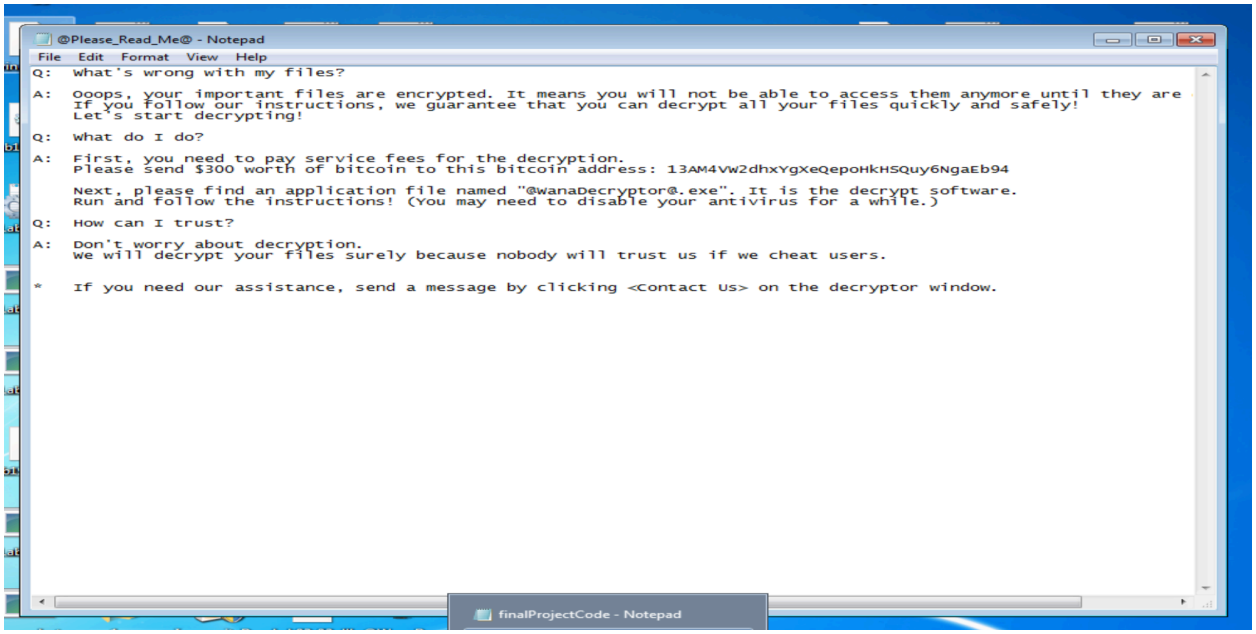
Checking the inetsim log after execution, the same DNS queries are seen as on Wireshark during the process of running the executable. Several reverse DNS lookup requests are made, which could imply attempted connection to a command-and-control server.



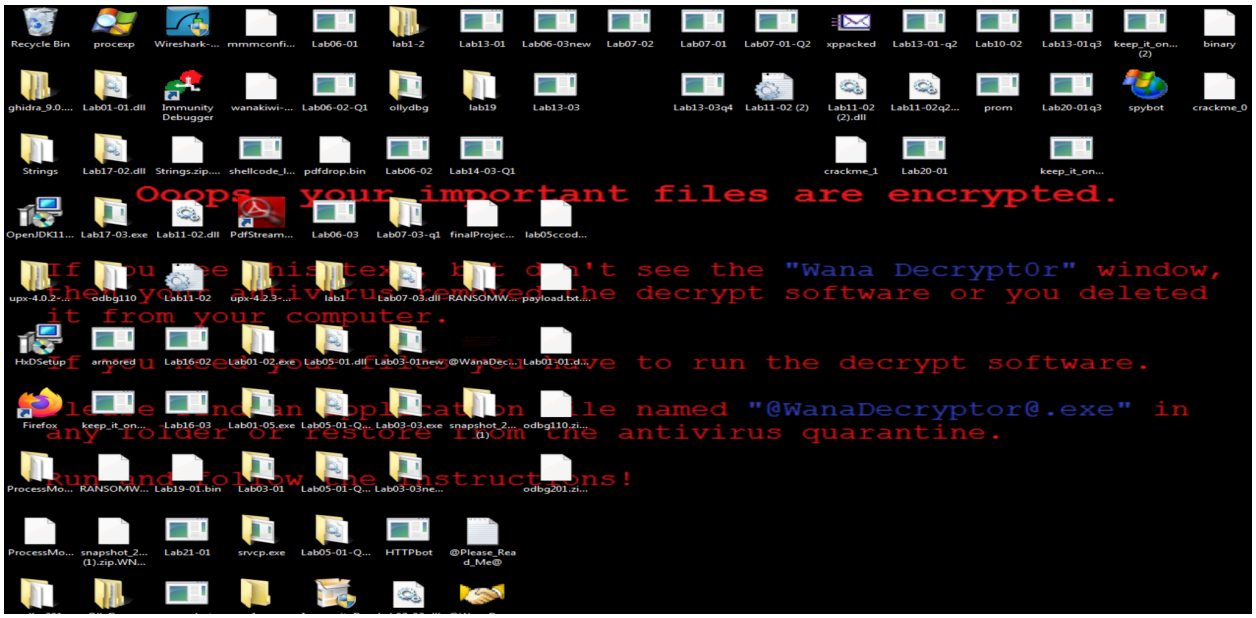
Checking process monitor, we see a new instance of “regsetvalue” generated by the executable - to confirm this, I opened up the registry editor, and saw a new directory in HKCU/Software, with the same key name,WanaCrypt0r, and the same value, wd, as seen in the debugger output.



File Modifications and attempted cleanup:



This file is created, along with several new files according to the process monitor. This file seems to be a ransom note, which specifies a bitcoin address which matches one of the 3 seen in the array from the Ghidra disassembly - randomly chosen out of the 3 as per the decompiled source code.



Additionally, every non-executable file in my system has now been encrypted, and given a ".WNCRY" extension. Fortunately, I had a decryption tool WanaKiwi, which was developed after the initial WannaCry attack, and tries to reconstruct the address of the decryption key from memory. After running the decryptor and waiting for some time, the decryptor was able to recover and start decrypting my files, and then deleted the WannaCry executable itself -

however, instead of cleanly restoring my files, an encrypted copy of each file with the .WNCRY extension remained, which would have to be deleted manually.