Cody Ickes, Ethan Letourneau

WannaCry Ransomware Malware Analysis

Introduction

The WannaCry (also known as WannaDecrypt, WannaDecrypt0r, WannaCrypt, etc) ransomware worm made headlines globally in 2017 when it first appeared. We took interest in this malware because of its notoriety and its novel combination of both ransomware and worm functionalities. In this section, we will first discuss configuration, and then the division of work.

Because WannaCry takes advantage of a backdoor in Windows systems (first discovered by the National Security Agency, EternalBlue), we need to analyze it on a Windows VM, particularly an unpatched Windows 7 VM, as this was the main system the malware targeted originally. So, we are using VirtualBox 6.0.4 to run a VM with the following configuration:

- Windows 7 Home Premium, SP1 64-bit (No further updates)
- 32 GB Dynamically-allocated Disk
- 2 GB RAM
- Java SE Runtime Environment 8u241
- Java Development Kit 13.0.2
- SHA256: 24d004a104d4d54034dbcffc2a4b19a11f39008a575aa614ea04703480b1022c

For division of duties, Mr. Letourneau is to handle the static analysis of the sample, while Mr. Ickes will oversee dynamic analysis. Tools being used in either phase will be detailed in the respective sections.

Static Analysis

Part I: Wannacry.exe

To verify the sample obtained was in fact the malware it was supposed to be, we uploaded it to VirusTotal, which gave the verifying information seen in Figure 1. Now sure that this was in fact the WannaCry malware, we installed a Windows version of the Unix-based “Strings” to see if we could obtain any useful information. From this we could discern that there was no meaningful obfuscation or packing going on in the malware, as using Strings gave us what seemed to be a list of the processes that the malware would use. Additionally, we found a list of file extensions in the

output from Strings, which we assumed would be used by the program to look for files ending in these strings to encrypt. The final bit found from the Strings output was what seemed to be some sort of XML, possibly to be used in the GUI that explains the attack has happened and asks for the ransom. Some of the output from Strings are included in Figure 2.

The next step taken was to load the malware into the PEiD program – from this we discovered that the malware was written in C++ and compiled using Visual Studio 6.0, utilizes Win32 GUI, and also that the entry point would be at 0x9A16. Using the pre-installed plugin for PEiD, Krypto Analyzer, we found signatures for 10 cryptographic algorithms in the file and the addresses they were found at, which will prove helpful when analyzing the instructions in IDA and GHIDRA. Output from PEiD is shown in Figure 3.

The last tool used before we went into disassembly was Dependency Walker, to verify the exact DLL's being loaded, and which functions they'd be using. As shown in Figure 4, seven DLL's were found, being Kernel32, AdvAPI32, WS2_32, MSVCP60, IPHLPAPI, WinINET, and MSVCRT.

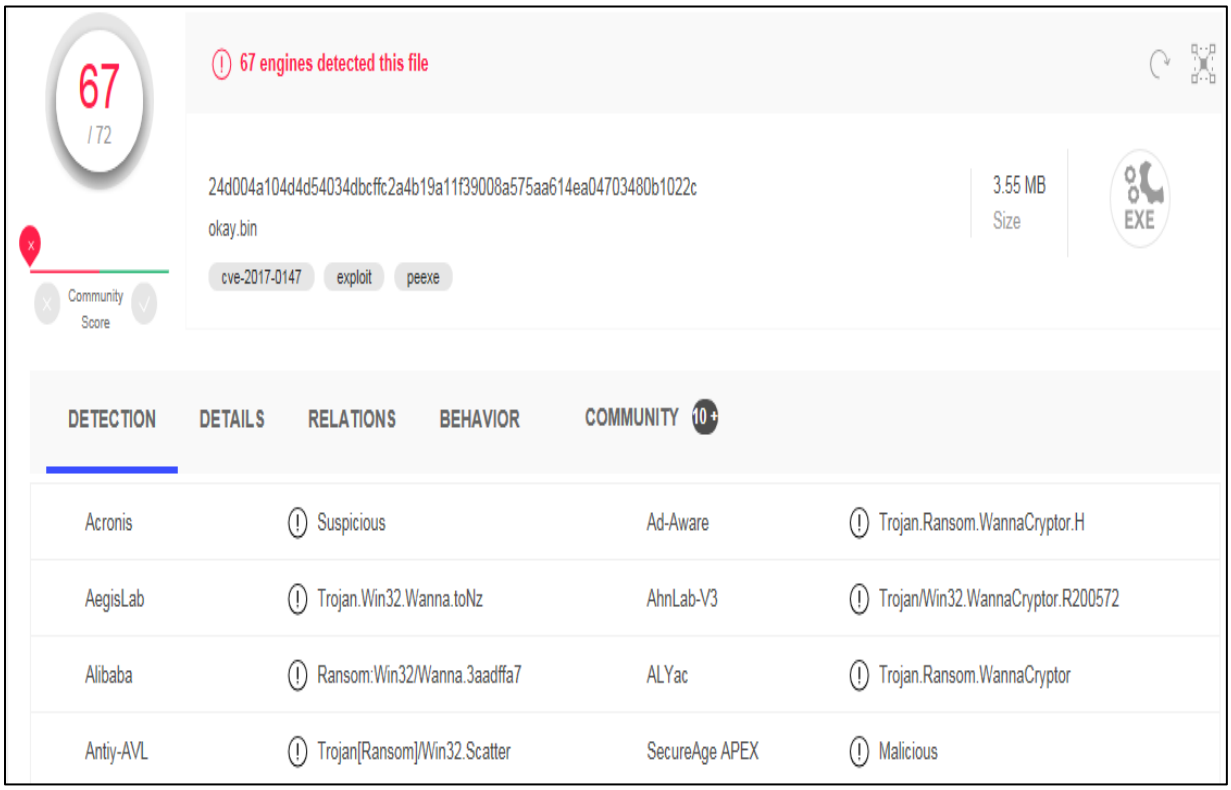


Figure 1: VirusTotal Scan of the file.

Figure 2: Samples of output from "strings"

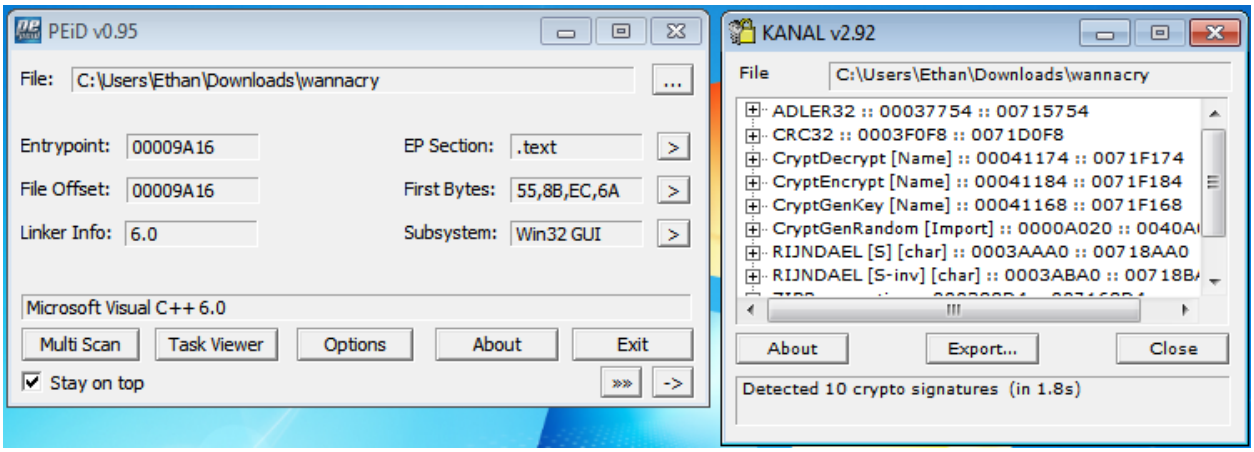


Figure 3: PEiD Findings

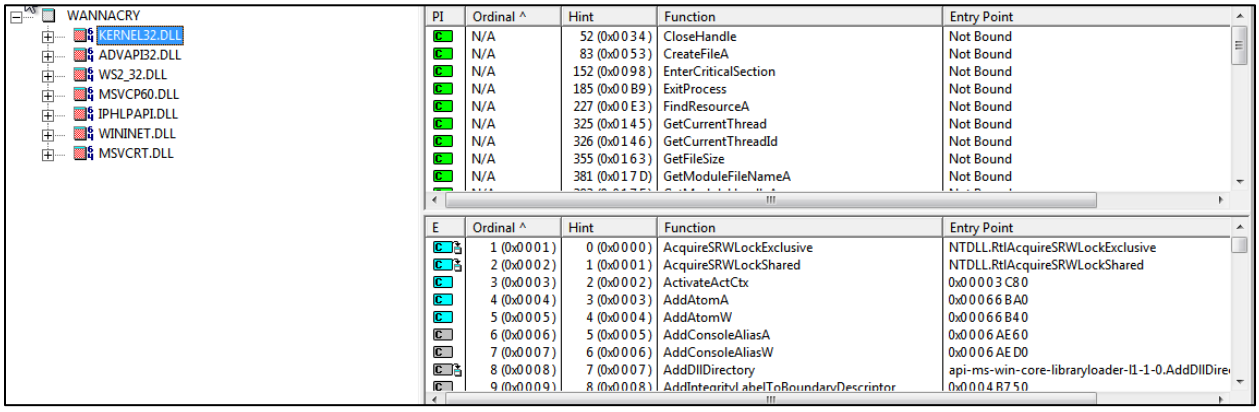
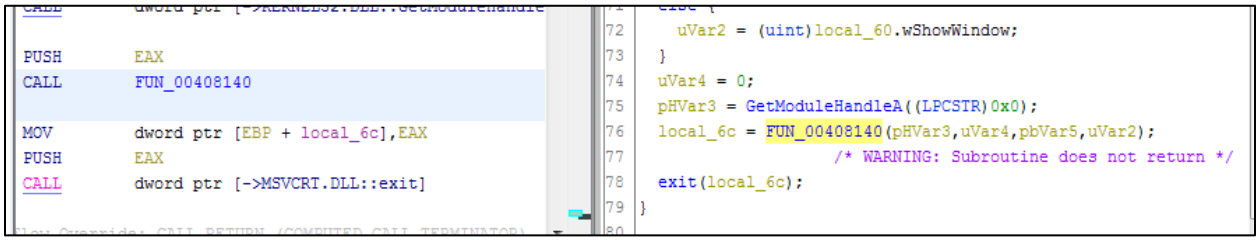
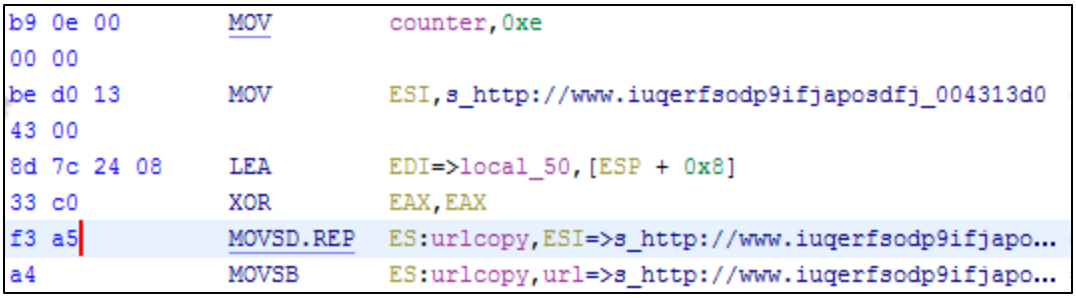


Figure 4: Output from Dependency Walker

Now with all preliminary information logged, we started on analyzing the actual assembly. We used GHIDRA for the analysis as it was the tool we're more familiar with. After loading the binary into GHIDRA and allowing it to analyze the sample, we were delighted to see that it found the entry function, confirming our suspicions that there was no packing done to the malware. The entry function seems to be standard and not malicious itself – the majority of it seems to be setup, such as obtaining start-up information (evidenced by variables of type _startupinfo, the call of GetStartupInfoA, and so on), getting arguments of some kind (call of __getmainargs), and initializing functions (several calls of _initterm, which according to Microsoft documentation walks a function pointer table). At the end of the entry function there is a function call that doesn't return and is the only unnamed function (FUN_xyz) in the entry function that does something significant and has any parameters:



Inside the function, renamed "post_init", there is an odd URL string referenced several times, as seen in the picture on the next page. In the decompiled code, it is mistyped, so we correct it to be a char* and rename it "url". There is a strange x86 instruction being used on it pictured as well, MOVSD.REP, which according to documentation "moves the byte, word, or doubleword specified with the second operand to the location specified with the first operand". The REP part simply means to do it over again for each part of the string. This is awkwardly reflected in the decompiled code as a while loop that essentially copies the string at "url" to another variable, renamed "urlcopy", four bytes at a time (word at a time).



Further down the function, an attempt to open an internet connection to the URL is made (via InternetOpenA and InternetOpenURLA). There is then a check on the return value of

InternetOpenURLA, which documentation from Microsoft tells us will be NULL, 0x0, if the connection fails, and some other number otherwise. If the URL is not reached, then another function seems to be called, otherwise 0 is returned. Doing some reading on WannaCry, this seems to be the "kill switch" URL that was discovered shortly after the malware was introduced in 2017 – if the URL is active and reachable, the attack won't be performed. So, we renamed the function that will be called "kill_failed".

Looking in to the kill_failed function, the first thing done is a call to GetModuleFileNameA, which "retrieves the fully qualified path for the file that contains the specified module", and stores the path in the second argument, so we rename that "filepath". There is then a call to _p__ argc, which gets simply argc – if it is less than two, meaning no arguments (argv[0] is program itself), another function is called, followed by a return.

```
GetModuleFileNameA((HMODULE)0x0,&filepath,0x104);
arg_count = __p___argc();
if (*(int *)arg_count < 2) {
  FUN_00407f20();
  return;
}
```

This function called just calls two other functions back-to-back, each of which has a lot going on, so we'll come back to them and just rename the function "no_arguments" for now. After this check, if there are two or more arguments, there is a call to OpenSCManagerA, which opens a connection to the Service Control Manager of the specified machine – in this case when the first argument is 0x0, the local computers manager is being reached. There is then a check for the return value of OpenSCManagerA – if it is 0x0, it means the function failed, so the check is if it is not equal to 0x0. If it is not, there is a call to OpenServiceA, which returns a handle to the service specified in the second argument, here being "mssecsvc2.0". Upon searching for this service online I found that though this seems like it could be a Windows program, it is actually a service created by WannaCry that is detected as malware itself. After this call, there is another check for the return value to be not 0x0, and if it is not (meaning the function succeeded), another function is called, passing in the return value of the OpenServiceA call, named appropriately by GHIDRA "hSCObject". After the call to the function, the mssecsvc2.0 and Control Manager are closed by calls to CloseServiceHandle. We rename this function that passes the mssecsvc handle "mssecsvc_handler" and look into it.

In this, there is a lot of calculation involving the second parameter (passed in as 0x3c from the previous function), ending with a call to ChangeServiceConfig2A, which has the mssecsvc2.0

handle as the first argument, the handle whose configuration were changing, 2 as the second parameter, and 0 as the third. These other two parameters indicate the "SERVICE_CONFIG_FAILURE_ACTIONS" are being changed to be unchanged. As far as I can tell, the function effectively does nothing. Upon return, the kill_failed function finally calls StartServiceCtrlDispatcherA, with our mssecsvc2.0 information as parameters – this "connects the main thread of a service process to the service control manager, which causes the thread to be the service control dispatcher thread for the calling process".

Looking back to the functions called in the case of no arguments, the first one is similar to the rest of the kill_failed function, with some differences. The first thing done by this function is sprintf our filepath variable from before with the format string "%s -m security" into a local variable. This sets it to be "C:\...\wannacry -m security", so we rename the local variable "wc_flagged". There is the similar OpenSCManagerA for the local computer, and a CreateServiceA that looks to be significantly longer than the StartServiceA, so we thought to look at each argument to it to see exactly what it meant:

```
hService = CreateServiceA(hSCManager,s_mssecsvc2.0_004312fc,
                          s_Microsoft_Security_Center_(2.0)_S_00431308,0xf01ff,0x10,2,1,
                          wc_flagged,(LPCSTR)0x0,(LPDWORD)0x0,(LPCSTR)0x0,(LPCSTR)0x0,
                          (LPCSTR)0x0);
```

SC_HANDLE CreateServiceA(

SC_HANDLE hSCManager = hSCManager, the Service Control Manager of the local machine,
LPCSTR lpServiceName = "mssecsvc2.0", the name of the new service,
LPCSTR lpDisplayName = "Microsoft Security Center (2.0) Service", the service's display name,
DWORD dwDesiredAccess = 0xf01fff, According to MS docs = "SERVICE_ALL_ACCESS",
DWORD dwServiceType = 0x10, meaning the service will run its own process,
DWORD dwStartType = 2, meaning the service starts on startup of the machine,
DWORD dwErrorControl = 1, meaning if it fails to start, startup continues as normal,
LPCSTR lpBinaryPathName = "C:\...\wannacry -m security", the path to the service's binary,
LPCSTR lpLoadOrderGroup = 0, doesn't belong to a group,
LPDWORD lpdwTagId = 0, doesn't change a tag,
LPCSTR lpDependencies = 0, no dependencies,
LPCSTR lpServiceStartName = 0, uses Local System Account to name the service,
LPCSTR lpPassword = 0, no password.

);

So, in summary, this creates a service that displays as Microsoft Security Center (2.0) Service that is in fact the WannaCry malware on startup and with full administrative permissions. The rest of the

function then starts the service if it was successfully created and returns. So, we rename this function as “create_wc_service”.

Looking at the second function in the no_arguments function, at first glance there is a lot of CreateFileA, WriteFileA, and so on happening, so we believed this to be the real meat of the malware. The function first loads a handle to Kernel32.dll, and gets the process addresses for processes inside this DLL, including the two mentioned above as well as CloseHandle and CreateProcessA, checking with if statements that each step is loaded correctly (not equal to 0x0). Then a call to FindResourceA tries to find a resource with the name parameter being 0x727 – we aren’t sure what this is, so we just rename it “Resource_0x727”, as well as renaming all the variables mentioned before as shown:

```
kernel32dll = GetModuleHandleW(u_kernel32.dll_004313b4);
if (kernel32dll != (HMODULE)0x0) {
  CreateProcess = GetProcAddress(kernel32dll,s_CreateProcessA_004313a4);
  _CreateFile = GetProcAddress(kernel32dll,s_CreateFileA_00431398);
  _WriteFile = GetProcAddress(kernel32dll,s_WriteFile_0043138c);
  _CloseHandle = GetProcAddress(kernel32dll,s_CloseHandle_00431380);
  if ((((CreateProcess != (FARPROC)0x0) && (_CreateFile != (FARPROC)0x0)) &&
      (_WriteFile != (FARPROC)0x0)) && (_CloseHandle != (FARPROC)0x0)) {
    Resource_0x727 = FindResourceA((HMODULE)0x0,(LPCSTR)0x727,&DAT_0043137c);
```

The resource is then loaded to another variable with LoadResource and locked with LockResource, each getting another if statement check that they were successful. After a check to make sure the size of the resource is not zero, and after what seems to be some setting of memory to zero, we comr upon these strange looking sprintf calls:

```
sprintf(&local_208,s_C:\%s\%s_00431358);
sprintf(&local_104,s_C:\%s\qeriuwjhrf_00431344);
```

Looking to the actual assembly, it seems there are a few arguments that have been excluded from the first call because there are some instructions between them:

```
00407dea 68 6c 13 43 00   PUSH     s_tasksche.exe_0043136c
00407def 66 ab            STOSW    ES:EDI
00407df1 aa               STOSB    ES:EDI
00407df2 68 64 13 43 00   PUSH     s_WINDOWS_00431364
00407df7 8d 44 24 70      LEA      resourc_size=>local_208,[ESP + 0x70]
00407dfb 68 58 13 43 00   PUSH     s_C:\%s\%s_00431358
00407e00 50               PUSH     resourc_size
00407e01 ff d6            CALL     ESI=>MSVCRT.DLL::sprintf
```

Overriding the signature for each will fix this, and the two additional strings on the first will be “WINDOWS” and “tasksche.exe”, making the string “C:\WINDOWS\tasksche.exe”. There is one

additional argument for the second sprintf, which is just “WINDOWS” again, making the second string “C:\WINDOWS\qeriuwjhrf”. We rename the buffers these are fed into to reflect these discoveries. MoveFileA is used next to move the tasksche.exe to the “qeriuwjhrf” folder. The next line in the decompiled code looks to also be muddled or misinterpreted by GHIDRA. Some variable is set to the return of (*_CreateFile)() – the variable is the handle set earlier. This doesn’t make much sense, And looking into the assembly we see that again there are a couple missed parameters. There wasn’t a way to edit this “function” signature, and I did some reading of forum posts about pointers to service calls and found that they have their own type that they can be set to. So, I set the data type of the global to CreateFileA, and the stack parameters populated the function – the return type is a handle to the created file, so we renamed it aptly:

```
tasksche_handle =
     (*CreateFile)(&c_win_tasksche,0x40000000,0,(LPSECURITY_ATTRIBUTES)0x0,2,4,
                   (HANDLE)0x0);
```

There are similar calls using the WriteFile and CloseHandle variables from earlier, so we set those to be their correct type to make sure their parameters are correct as well. The WriteFile’s parameters were corrected, while the CloseHandle’s parameters remained unchanged. WriteFile has some odd parameters, including the return of the LockResource call we saw earlier that was used to make sure the resource was locked:

```
(*WriteFile)(tasksche_handle,0x727_Locked,nNumberOfBytesToWrite,
             (LPDWORD)&0x727_Locked,(LPOVERLAPPED)0x0);
```

Looking into the documentation reveals that if LockResource is successful, then the return value is actually a pointer to the first byte of the resource. This now makes more sense, and we know the 0x727 resource is being written to tasksche.exe. 0x727_Locked is renamed to reflect its actual value.

The rest of this function, save for the final line, looks like more loops dealing with memory again. The final line before closing of the handles is a call to the CreateProcess variable we saw earlier. After fixing the type, the arguments mostly make sense, save for the second. This is supposed to be the command line command to create the process. We renamed the variable and looked back into the loops above to see it was filled with some odd data that is untranslated by GHIDRA, seeming just numbers:

```
00431340 20 2f 69 00     addr        DAT_00692f20                    = ??
```

Knowing that these must be characters, we looked to an ASCII chart to see that 20 2F 69 00 translated to " /i/0", or just /i with a space before it and null-terminated. So, the CreateProcess call creates a process with the command line "…/tasksche.exe /i", which is a program which is data from the stil-unknown 0x727 resource. So, we rename the function properly to reflect what it does.

This is the end of what the sample seems to do, so the malicious actions are probably actually being done by this newly made executable. After doing some digging around, we found that this can be done by GHIDRA – so we go to the spot in memory where the 0x727 resource is stored and export it as an executable, we name 727.tmp.

Part II: 0x727 Resource

After analyzing the new executable, we see that GHIDRA again can identify the entry point, so we start here. The entry function for this is very similar to that of the original sample, where it is doing mainly preliminary work and the final function called is essentially the real start of the program.

Looking into this function we see a call to GetModuleFileNameA, which is putting the name of the executable into a local variable we'll rename. Immediately after, another function is called – looking into this we see a call to GetComputerNameW, which stores the name of the machine and the size of the name into two locals we'll rename as well. There is then this:

```
local_8 = 0;
_Seed = 1;
sVar1 = wcslen((wchar_t *)&com_name);
if (sVar1 != 0) {
  puVar6 = &com_name;
  do {
    _Seed = _Seed * *puVar6;
    local_8 = local_8 + 1;
    puVar6 = puVar6 + 1;
    sVar1 = wcslen((wchar_t *)&com_name);
  } while (local_8 < sVar1);
}
srand(_Seed);
```

Wcslen is just a string length function for a wide string, and the variable name _Seed gives us a clue as to what is going on here - _Seed is an unsigned integer that is generated in the do-while loop

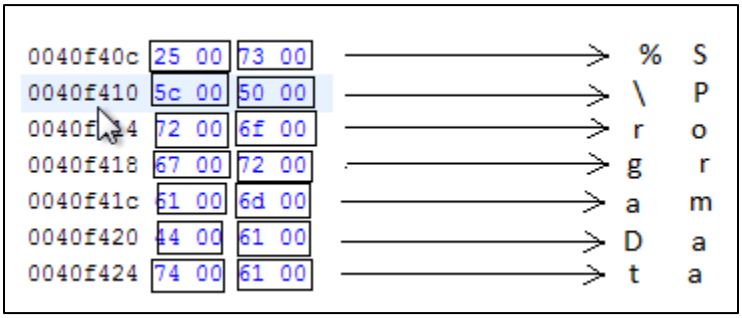
based on the string of the computer name which is fed into srand, which will initialize the rand() function to generate a pseudo-random number using the argument as a seed. Rand() is called just after this block and several times after in the rest of the function to store random numbers into the parameter passed to the function. The storing of these numbers have the parameter cast to a char *, and changing the type of the parameter to a char * reveals that the random numbers are being used to generate a random string that is bookended by a null terminator just before the return. So, we rename param_1 to be "random_string", and the function to be "generate_random_string". Back in the previous function, we see a global is being used as the parameter and rename it as well.

Next, there is a check on the number of arguments – if there is one, it is compared with an odd-looking global, which upon further inspection is a mis-typed char * that is 69 2F, i.e., or "/i" argument from before. If this is the argument, there is a second check on a return value of another function we jump into.

In this function, we see the global random_string from before being copied to a local variable using MultiByteToWideChar, and then a copying of the Windows directory with GetWindowsDirectoryW into another local variable. Then, there is a swprintf call that looks odd:

```
swprintf(&local_2d4,(size_t)&_Count_0040f40c,&Win_Dir);
```

There is no format string and only a size_t that is being gotten globally. Perhaps this was a sprintf mistakenly translated to swprintf, because the global variable does not look like an integer, and when we look at it as a character string it makes perfect sense:



So, this swprintf is most likely putting the Windows directory, likely C:\, in front of another directory "\ProgramData". So, we rename the local_2d4 accordingly. GetFileAttributeW is called, and there is a check to see if it returns 0xFFFFFFFF when passing in this new directory. Additionally, there is a check on the return type of another function that we will look into now.

This function is small, taking in three parameters. It calls CreateDirectoryW to make a directory from the first parameter with LPSECURITY_ATTRIBUTES = 0x0, indicating it to be inheriting the parent directory attributes. Then, there is a call to SetCurrentDirectoryW to this newly created directory, and if this is successful, the same creation and setting is done for the second parameter, inside this newly made directory. Once in the second directory, there is a swfprintf of the first two parameters to be put int format string "%s\%s" and set in the third parameter, then return 1. If the first SetCurrentDirectoryW doesn't work, this function returns 0. So, we rename the function to reflect its workings.

So, this function is called with the C:\ProgramData directory as parameter 1, and the random_string global as parameter 2, creating the directory C:\ProgramData\randomstring, and putting this string in the first parameter of this outer function – if this function returns zero, i.e. fails, or the previous GetFileAttributeW call returns 0xFFFFFFFF, then there is another swfprintf call to put the Windows directory ahead of another global string that is misrepresented, this one reads "%s\Intel". Then there is another attempt to use the above-mentioned function to put the random_string directory into the Intel directory. If this fails, and if calling the above-mentioned function using just the Windows directory to put the random_string directory in also fails, there is a call of GetTempPathW to try and put the random_string directory into C:\TEMP\. So, this function basically tries to find a place to put the random_string directory into, so we'll rename the whole thing "put_random_string_dir_into_param" .

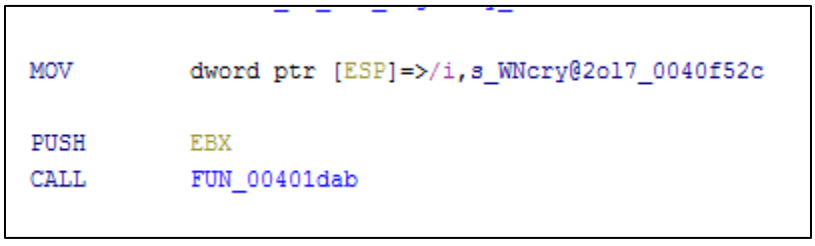
Coming back out to the "real" entry function, if the put_random_string_dir_into_param function succeeds and there is a "/i" argument to the program, then the tasksche.exe is copied into the file path made using CopyFileA. If the return value for GetFileAttributesA is not 0xFFFFFFFF, i.e., the file exists, then another function is called. This a short function, but it calls several other ones. It first gets the full path name of the newly inserted tasksche.exe and inputs it into a local buffer we rename "full_path_name", then inputs it into another function and checks it did not return zero. This other function tries to open a service with the same name of the random_string if it exists, or create the service if it doesn't, much like in the original sample using OpenServiceA and CreateServiceA. Once it's sure that it exists, it starts the service – if it can, it returns 1, otherwise 0. We rename this "create_and_start_service". There is then a call to another function which basically uses OpenMutexA with the string "Global\\MswinZonesCacheCounterMutexA". This mutex would guarantee no more than one instance of the service runs at the same time – we rename this "get_mutex". So, if create_and_start_service and get_mutex both work, this function returns 1. Otherwise, there is another function call to a function that tries to run the command at the full path

name, which would be the tasksche.exe. If this is successful and get_mutex for this started program is successful, the function will also return 1 here. Otherwise, it returns 0. So, this function called in the real entry will be called “create_or_start_tasksche”.

Back in the real entry, if create_or_start_tasksche works, zero is returned and we suppose the program is over. Recall that this was all done, being the creating the random string based on the machine name, putting it as a directory in one of several possible locations, and creating and/or starting the service, was all done if the 0x727 program had an argument “/i”.

If the program was run without /i, SetCurrentDirectoryA will be called with the file_path variable, being the path to the tasksche.exe. Then a series of functions are called in succession, the first of which concatenates a constant character string reading “Software\” with another one of the global data pieces that GHIDRA did not recognize as a string; looking at it and translating it from ASCII gives us the string “WanaCrypt0r”. Then this first function has a loop that seems to run only twice, one where a variable called hKey = 0x80000002 and where it equals 0x80000001. This is then used in a call to RegCreateKeyW as the first argument, along with the string mentioned before and a local variable. The value being either 0x80000002 or 0x80000001 determines if the key is for the local machine or the user. If the parameter passed to the function is zero, then the registry is created for each using RegQueryValueExA, and the value of the registry is stored in the fifth parameter, which is a local variable we will rename. Zero is returned by RegQuertValueExA if successful, and if it is, SetCurrentDirectoryA is used to set the current directory to the registry value. If the parameter to our function is not 0, GetCurrentDirectoryA is called with the registry value passed in, meaning we store the registry value to our current directory. We’ll rename this first function as “create_or_set_registry_cwd”.

The second function in this list has an odd-looking signature in the decompiled code, being one static 0x0 value passed as an HMODULE. Looking at the disassembly, we see that GHIDRA misses a string that should also be a parameter:



After fixing the function signature to have this string “WNcry@2ol7” as a parameter, we jump into it. It looks like we have yet another resource being loaded in this function, this time with a value of 0x80A:

```
hResInfo = FindResourceA(param_1,(LPCSTR)0x80a,&DAT_0040f43c);
if (((hResInfo != (HRSRC)0x0) &&
    (hResData = LoadResource(param_1,hResInfo), hResData != (HGLOBAL)0x0)) &&
   (pvVar1 = LockResource(hResData), pvVar1 != (LPVOID)0x0)) {
  DVar2 = SizeofResource(param_1,hResInfo);
  piVar3 = (int *)FUN_004075ad(pvVar1,DVar2,param_2);
```

This function looks similar to the function that loaded the 0x727 resource in the original sample, so rather than going through the whole thing again we see if we can extract this resource from 727.tmp. We once again go to the “Labels” and can see there is one called “Rsrc_XIA_80A_409” – going here and trying to export reveals that it has multiple files in it. Exporting as an x86 binary yielded little, it was mostly gibberish, and exporting as a file system also did not work. After digging around for a tool that could extract resources that was not a debugger (Did not want to accidentally run the malware), I came across a Linux tool called “wrestool”. We installed a Ubuntu VM (Version 18.04, 64-bit version), zipped up the 727.tmp and shared a folder between the two VMs briefly to transport it. Using this tool to list the resources in 727.tmp, we see this:

```
ethan@ethan-VirtualBox:~/Desktop/1233$ wrestool -l .rsrc_\[007100a4\,00a6a0a3\]_8909597174225962853.tmp
--type='XIA' --name=2058 --language=1033 [offset=0x100f0 size=3446325]
--type=16 --name=1 --language=1033 [type=version offset=0x359728 size=904]
--type=24 --name=1 --language=1033 [offset=0x359ab0 size=1263]
```

The first resource, 2058, is our 0x80A resource (Decimal 2058 = Hex 0x80a), so we try to extract it using this tool. Doing so, we want to know file information about this resource, namely its type. Thankfully, ubuntu has the “file” tool which can give us this information, and we find out that this is actually a ZIP archive. When we try to unzip it, we find it is password-protected:

```
ethan@ethan-VirtualBox:~/Desktop/1233$ wrestool -x -R --name=2058 .rsrc_\[007100a4\,00a6a0a3\]_8909597174225962853.tmp > 80a.tmp
ethan@ethan-VirtualBox:~/Desktop/1233$ file 80a.tmp
80a.tmp: Zip archive data, at least v2.0 to extract
ethan@ethan-VirtualBox:~/Desktop/1233$ unzip 80a.tmp
Archive:  80a.tmp
[80a.tmp] b.wnry password:
```

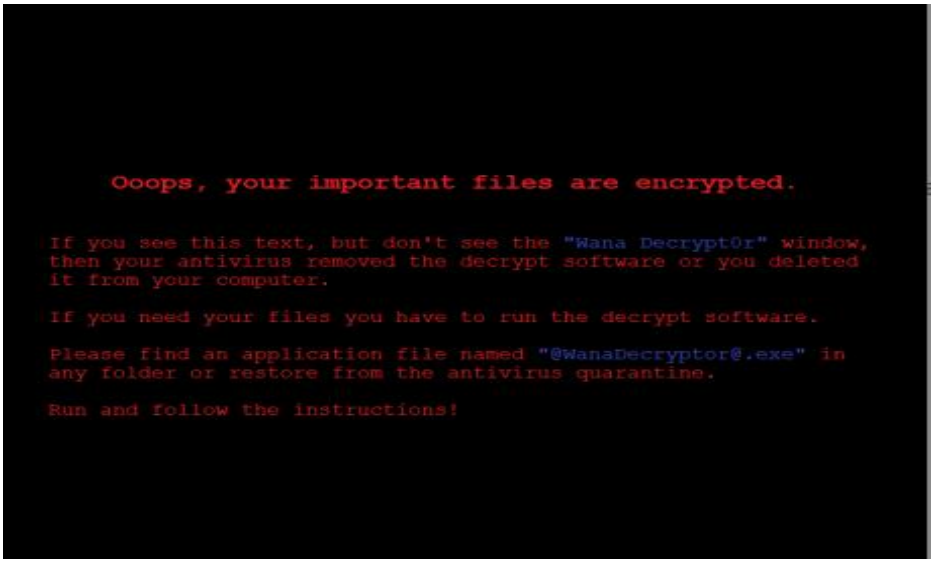
If we recall back to the function that was creating and running the resource in the 727.tmp, we remember that odd string that was passed with it, “WNcry@2ol7”. If we put this in as the password, it works. We end up with the following files:

```
ethan@ethan-VirtualBox:~/Desktop/1233$ dir
727.zip  b.wnry  msg     s.wnry      taskse.exe  u.wnry
80a.tmp  c.wnry  r.wnry  taskdl.exe  t.wnry
```

Running “file” on each reveals b.wncry is a bitmap image (displayed below) which is a message about your files being encrypted. “msg” is a directory with files named for many languages, and the English one has what appears to be the ransom note, formatted in XML perhaps. “r.wnry” has a message in Q&A form about the ransom and delivery of bitcoin with “%s” modifiers for what would be the delivery addresses. “taskdl”, “taskse” and “u.wnry” are all executables, and “s.wnry” is another ZIP file.

- Taskdl.exe is a task deletion tool used by the resource.
- Taskse.exe is the worm component, spreading the malware via Remote Desktop Protocol.
- U.wrny is the decryption tool used by the “Decrypt some of my files” button on the GUI.

“c.wnry” and “t.wnry” are both data files. Looking at the text of each, for c.wnry we see several “.onion” addresses, which appear to be bitcoin addresses, but the other data file is gibberish with no text. We try to open the s.wrny archive, but it does not work – a quick search tells us the ZIP will contain a download to the Tor browser, which is provided in the infamous GUI.



Before seeing if we need to reverse any of these .exe files, we go back to the 727.tmp. The second function in this succession, as said before, extracts this 80A resource. Going into the third function, the decompiled code looks clean:

```
uint uVar1;
int iVar2;
undefined local_31c [178];
char local_26a [602];
char *local_10 [3];

local_10[0] = s_13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb_0040f488;
local_10[1] = s_12t9YDPgwueZ9NyMgw519p7AA8isjr6S_0040f464;
local_10[2] = s_115p7UMMngoj1pMvkpHijcRdfJNXj6Lr_0040f440;
uVar1 = FUN_00401000(local_31c,1);
if (uVar1 != 0) {
  iVar2 = rand();
  strcpy(local_26a,local_10[iVar2 % 3]);
  FUN_00401000(local_31c,0);
}
return;
```

The first three lines after the declaration of variables stores global strings into the local_10, which look to be the Bitcoin addresses, so we'll rename the variable accordingly. Then, an undefined string is passed into a function with the number "1" – looking into this we see the first thing done is a check on that second parameter. If it is 0, a local string _Mode is set to a global string, which is "wb", and else it is set to another global reading "rb". Then fopen is called with the c.wnry data file (the one we saw before containing the .onion addresses) and the _Mode variable, which will indicate either reading or writing is to be done. Then there's another check on that second parameter – if it is 0, there is a fwrite call, writing the contents of the first parameter to c.wnry, otherwise fread is called, reading contents from c.wnry into the parameter. So, we rename this "0write_1read". Back into the third function, one of the three bitcoin addresses is randomly chosen via a rand() call and copied into another local array, but doesn't seem to be passed anywhere. Looking into the disassembly, it seems that one large string was mistakenly broken into two because of how they're accessed – fixing this reveals that the address passed into the 0write_1read function is the same that is being written to here. 0write_1read is called again with "0" passed in, making it write the randomly selected address to c.wnry. So, this third function will be called "write_bitcoin_addr_to_cnwry".

To summarize these three functions called in succession, the first sets or creates the registry value for the current working directory, the second extracts the resource 80A containing all the files we saw, and the third writes one of three bitcoin addresses to a data file in that resource. Back in the real entry to 727.tmp, two programs are ran in succession with flags, the first being "attrib +h", which sets the file to hidden. The second is "icacls" with many flags, which is a program that sets access controls for the files in the current working directory. The flags to this basically give all permissions, continuing despite errors, and suppressing any success messages. The next function called immediately calls another function, which calls GetProcAddress for six cryptographic processes, such as CryptImportKey, CryptEncrypt, CryptDecrypt, and so on, returning 1 if they all

succeed and 0 if any do not, and storing the handles in globals we'll rename like we saw before. Back in the function that called this, which we rename "load_crypto_processes", it seems to be the exact same function but with processes that deal with file manipulation, so we rename these global variables and the function itself as "load_processes".

The next function called looks very strange, so to see if anything was left out we look to the disassembly:

```
004020fe 8d 8d 1c        LEA         ECX=>local_6e8,[EBP + 0xfffff91c]
         f9 ff ff
00402104 e8 f4 f1        CALL        FUN_004012fd
         ff ff
00402109 53              PUSH        EBX
0040210a 53              PUSH        EBX
0040210b 53              PUSH        EBX
0040210c 8d 8d 1c        LEA         ECX=>local_6e8,[EBP + 0xfffff91c]
         f9 ff ff
00402112 e8 20 f3        CALL        FUN_00401437
         ff ff
```

The second function called here is defined as a thiscall, while the one we look into is not, even though they're both being passed local_6e8 via ECX. Changing the calling convention of our function to thiscall gives it the local_6e8 parameter, which we'll rename "this", but the code doesn't look much different despite the change. It looks oike GHIDRA can't translate the assembly here well. ECX, the this pointer, is put into ESI, and then ESI is used to initialize a bunch of data like an array or data structure. So this seems to be a constructor of some sort. After this the functions called have several functions inside them, and functions inside those. We found that it looked to be encrypting the files of the machine here with shifting of bits and using cryptographic constants. We decided to save time by trying to find a way to locate cryptographical constants in the file. Giving this a quick google returned a GitHub page for a GHIDRA script called "yara.py" that promised to do just this. Loading it and running it on the program gave us insight on where the encryption was happening in the jungle of nested functions after the constructor, and the constants seemed to correspond to the methods found by the KryptoAnalyzer plugin we used during the preliminary information gathering.

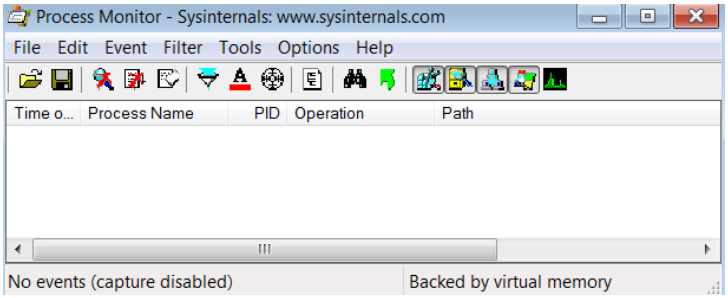
Dynamic Analysis

Provided the intrusiveness of the malware, several precautions have been taken before dynamic analysis can begin. The same Windows 7 virtual machine is in use with all removable devices, including network adapters, disconnected. The program will then be ran without administrative privileges starting from a clean snapshot. In performing analysis, several tools will be used to monitor process creation and execution along with system information. The tools being used are Regshot, RegFsNotify, Procmon, and Procexp, all running with administrative privileges. While Wireshark would be beneficial to use for analysis, the network adapters being disconnected will not allow Wireshark to monitor any network activity.

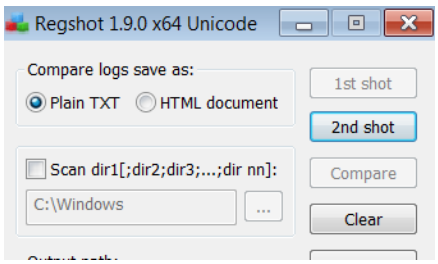
In each analysis the tools are used in the following ways:

- Regshot – first shot taken prior to malware execution, second shot taken immediately after
- RegFsNotify – monitoring execution flow, running before malware is executed, output saved
- Procmon – stopped and cleared up until malware is ran, records until malware process finishes
- Procexp – running in background, used to determine malware progress

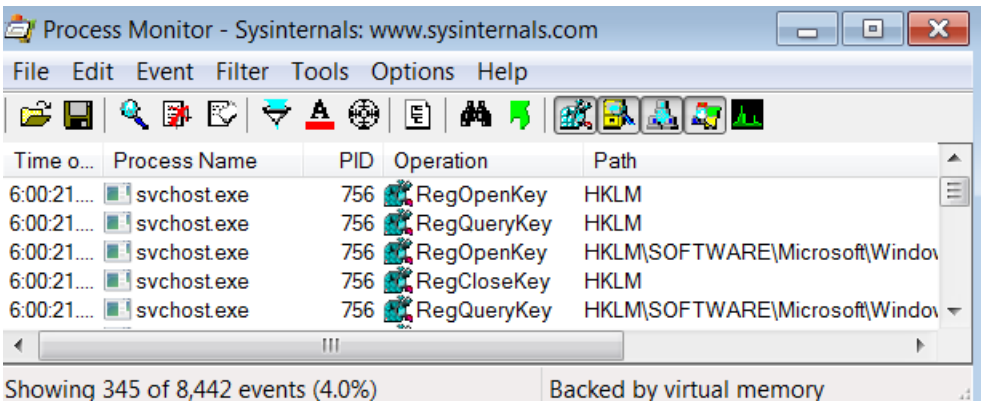
Before the malware can be ran, we first open Procexp to run in the background and open Procmon, turning off Capture and Clearing the log.



Procmon needs to have its Capture disabled and log cleared as it stores the information in memory, if left unmanaged, it can result in system crashes. Next, Regshot is started and our 1st shot is taken, capturing the current registry.

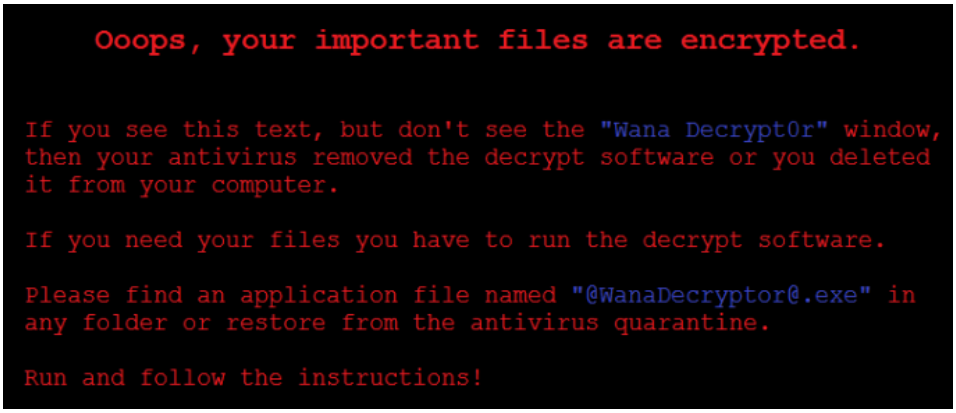


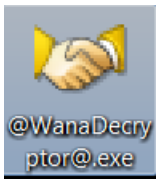
After the shot is taken, we open RegFsNotify and begin Capturing in Procmon, immediately running the malware after.



Upon running the malware many registry changes were in RegFsNotify. Unfortunately, capturing and reading the results proved to be too difficult as the large amount of data progressed too quickly. Furthermore, the results are stored in a log, however, I failed to account for WannaCry's encryption and the data is now unrecoverable. This experience has shed some light on working with these types of malware and we cannot make use of RegFsNotify.

Upon successful execution of the malware, we get many changes made to the machine. The background changes with information on decrypting, all files are encrypted with a READ_ME in each directory, and we get a window with information. One of the less noticeable changes is the malware executable being overwritten with the Wana Decrypt0r 2.0 executable. These modifications can all be seen below:





While the malware encrypted most of our data, it still needs to leave some files untouched to ensure that it does not encrypt itself and other important system files. As a result, most of our

programs can still be ran. With Regshot, we now take a 2nd shot and compare the two. There are 119 total changes, with 20 keys added, 2 values deleted, 46 values added, and 51 values modified.

For the relevant keys added, we see:

```
\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.bmp\OpenWithList
\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.WNCRY
\Software\Microsoft\Windows\CurrentVersion\Explorer\FileExts\.WNCRY\OpenWithList
\Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs\.bmp
\Software\Classes\VirtualStore
\Software\Classes\VirtualStore\MACHINE
\Software\Classes\VirtualStore\MACHINE\SOFTWARE
\Software\Classes\VirtualStore\MACHINE\SOFTWARE\Wow6432Node
\Software\Classes\VirtualStore\MACHINE\SOFTWARE\Wow6432Node\WanaCrypt0r
\Software\WanaCrypt0r
_Classes\VirtualStore
_Classes\VirtualStore\MACHINE
_Classes\VirtualStore\MACHINE\SOFTWARE
_Classes\VirtualStore\MACHINE\SOFTWARE\Wow6432Node
_Classes\VirtualStore\MACHINE\SOFTWARE\Wow6432Node\WanaCrypt0r
```

We can see nested value creations, one example is the creation of VirtualStore, VirtualStore\MACHINE, VirtualStore\MACHINE\Wow6432Node, and VirtualStore\MACHINE\Wow6432Node\WanaCrypt0r. These path creations are likely used to ensure WannaCry runs as a system program.

With deleted values, we can see:

```
,Internet Explorer\Main\Start Page Redirect Cache: "http://www.msn.com/?ocid=iehp"
,Internet Explorer\Main\Start Page Redirect Cache AcceptLangs: "en-US"
```

These deletions are likely performed to change Internet Explorer's start page. What was originally http://www.msn.com/?ocid=iehp is probably overwritten with a new start page that links to the WannaCry webpage.

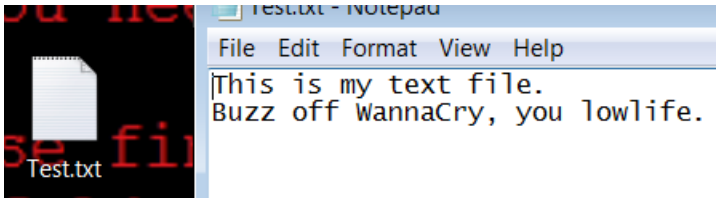
We can see the following values added:

```
,Internet Explorer\GPU\VendorId: 0x00000000
,Internet Explorer\GPU\DeviceId: 0x00000000
,Internet Explorer\GPU\SubSysId: 0x00000000
,Internet Explorer\GPU\Revision: 0x00000000
,Internet Explorer\GPU\VersionHigh: 0x00000000
,Internet Explorer\GPU\VersionLow: 0x00000000
,Internet Explorer\GPU\DXFeatureLevel: 0x00000000
,Internet Explorer\GPU\Wow64-VendorId: 0x00000000
,Internet Explorer\GPU\Wow64-DeviceId: 0x00000000
,Internet Explorer\GPU\Wow64-SubSysId: 0x00000000
,Internet Explorer\GPU\Wow64-Revision: 0x00000000
,Internet Explorer\GPU\Wow64-VersionHigh: 0x00000000
,Internet Explorer\GPU\Wow64-VersionLow: 0x00000000
,Internet Explorer\GPU\Wow64-DXFeatureLevel: 0x00000000
,Windows\CurrentVersion\Run\iunsqcmobalc230: ""C:\Users\Ethan\Desktop\Ransomware.WannaCry\tasksche.exe""
)r\wd: "C:\Users\Ethan\Desktop\Ransomware.WannaCry"
```

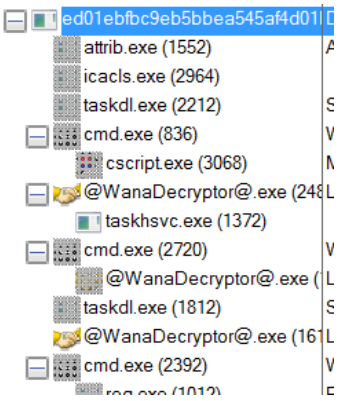
Information is zeroed out for Internet Explorer, confirming our suspicions that WannaCry is modifying it. Additionally, we can see that the malware's original location at C:\Users\Ethan\Desktop\Ransomware.WannaCry\malware.exe is overwritten with tasksche.exe. Upon further investigation, tashsche.exe is the rewritten name for the malware that appears when running. This information may be helpful when looking at Procmon and Procexp.

Looking at Procmon and Procexp, searching for tasksche.exe does not return any results. My suspicion is that the process is running from its initial state, restarting the machine would likely return results. In Procexp, we attempt to kill the original process to see what happens.

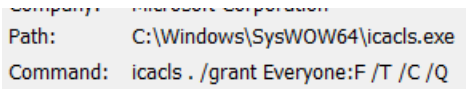
Upon killing the process tree, we create a new text file on the Desktop to check if the malware is still running as a hidden process. It appears that no other process is running as the text file remains unencrypted. As determined in the static analysis, the program will likely run again after a restart, however, it does not keep itself running after termination.



Finally, we will look at Procmon and see the lifecycle of the malware. Using the process tree, we can see the history of the program and what actions it has taken.



At icacls.exe, a command is ran to grant full access (:F) to Everyone.



Next, we can identify a cmd.exe process being spawned and running an unknown .bat file (likely the unzipping of the malware identified in the static analysis). The /c flag serves to run the command and then terminate the process, effectively cleaning itself up.

Path:	C:\Windows\SysWOW64\cmd.exe
Command:	C:\Windows\system32\cmd.exe /c 291651587475838.bat

Aside from opening the malware window, the only unique entry left in the process tree is a registry edit via cmd.exe. The command modifies the value “iunsqcmobalc230” under the SOFTWARE Run key with the full path to tasksche.exe, the newly named malware, all without prompting for confirmation. This addition is likely made to run the program from the unique value.

Path:	C:\Windows\SysWOW64\cmd.exe
Command:	cmd.exe /c reg add HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run /v "iunsqcmobalc230" /t REG_SZ /d "\"C:\Users\Ethan\Desktop\Ransomware.WannaCry\tasksche.exe\"" /f

The encryption process itself was difficult to identify and understand. Within the Procmon log, there were several uses of the Windows Cryptography library, certainly playing a part in the process. However, once the malware began going through all the files, it created the encrypted file, read the data from the original, and wrote back to the encrypted file. There was no other information between the read and the write, but, the read and writes were being performed by the process itself which is where the encryption likely occurred.

Ultimately, we were able to observe some of malware’s behavior, but not enough to figure out how the entire encryption process works. It was clear that changes to the registry were being made and several hidden processes were spawned to run commands. Most of the work was likely being performed from the .bat script ran by the process, separating the commands from the program itself. The malware has shown itself to be complex and highly effective at locking out the user.