Ryuk Ransomware Analysis

Joshua Main-Smith
joshuamainsmith@ufl.edu
Practical 3 - Malware Reverse Engineering

2021-04-11

Contents

Executive Summary

The Ryuk malware exhibits typical behavior that is consistent with most ransomware. After execution, there are a few UAC prompts followed by mass encryption of the files on the system along with a text file dropped to the desktop asking for a ransom. As is also typical in these situations, the user is asked to drop money into a BTC wallet to avoid detection in the real world. The purpose of the ransomware is apparent from the text file dropped, namely to hold the potentially valuable data that was encrypted in ransom until the user agrees to pay the attackers.

The malware itself doesn't have a lot of obfuscation techniques that are normally present with malware attempting to obscure its actions. There weren't any anti-disassembly techniques, no packing detected, minimal anti-debugging techniques, and minimal ant-vm techniques. Even so, the binary didn't appear to alter its behavior when allowing for debugging or virtual machine use to be detected.

This particular strain appears to be a later variant of the original strain first detected back in 2018. This was determined by the compilation date being shortly after. There didn't appear to be any network activity. There were a few registry keys modified, mostly related to network activity. Several imports related to process token adjustment, and an encrypted file named as a unique ID. The best indicator when encountering this malware would be the presence of a text file dropped to the desktop asking for a ransom and an attempt to encrypt files on the host.

Static Analysis

Hashes and Antivirus Check

The hash values of the binary are included below, as produced by VirusTotal.

MD5	8819d7f8069d35e71902025d801b44dd
SHA-1	5af393e60df1140193ad172a917508e9682918ab
SHA-256	98ece6bcafa296326654db862140520afc19cfa0b4a76a5950deedb2618097ab
Vhash	015076655d155515555088z54hz3lz
Authentihash	26578696205490c933d112e4536d9c08873343b20d2e40b4f54afd3466739592
Imphash	3d84250cdbe08a9921b4fb008881914b
Rich PE header hash	0419af1e713584cc28b658beec622601
SSDEEP	3072:b+hfiA0PJ/lmL4a17VnAy5jtZXDklVT49RQwo:i4AK/lmkaFVz7QQw
TLSH	T13A047D4B72A032F8F173CA3985528556F7BABC7517609B5F03A4833A1F17991AE39F20
File type	Win32 EXE
Magic	PE32+ executable for MS Windows (GUI) Mono/.Net assembly
TrID	Win64 Executable (generic) (48.7%)
TrID	Win16 NE executable (generic) (23.3%)
TrID	OS/2 Executable (generic) (9.3%)
TrID	Generic Win/DOS Executable (9.2%)
TrID	DOS Executable Generic (9.2%)
File size	171.50 KB (175616 bytes)

The hashes for each section are included here:

Hash

451e2e5089bb28b44cc13bbded763dfb

Name	Offset	Size	Hash
PE Header	0000000000000000	0000000000000400	236325e317d7513bd4e482e5b592620d
Section(0)['.text']	0000000000000400	0000000000016200	f529e68056f6c806033ad783007f1122
Section(1)['.rdata']	0000000000016600	000000000000b800	f75617972cb8c00ef7314c823b2ba3e4
Section(2)['.data']	0000000000021e00	0000000000007200	6c114e61f9275720c18145aa218fd9b0
Section(3)['.pdata']	0000000000029000	0000000000001200	9a14e4419da3e3a571fab686a59f0ab0
Section(4)['.gfids']	000000000002a200	0000000000000200	aca6474d521881a1a51a458fec7c1ed4
Section(5)['.rsrc']	000000000002a400	0000000000000200	9874d3aaec7f54ff5cb567e2ed76e60c
Section(6)['.reloc']	000000000002a600	0000000000000800	8d6991cbb6fd6953ccfd16d612ddf850

Submitting the binary to VirusTotal yielded a detection rate of 50/70 engines and three crowdsourced Yara rules indicating this as a variant of Ryuk Ransomware.

Crowdsourced YARA Rules

⚠ Matches rule Ryuk by kevoreilly from ruleset Ryuk at https://github.com/kevoreilly/CAPEv2
↳ *Ryuk Payload*

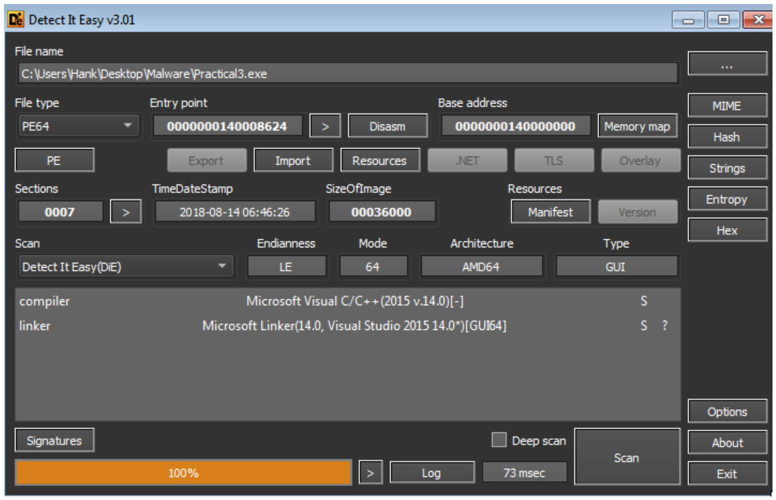
⚠ Matches rule win_ryuk_auto by Felix Bilstein - yara-signator at cocacoding dot com from ruleset win.ryuk_auto at https://malpedia.caad.fkie.fraunhofer.de/
↳ *autogenerated rule brought to you by yara-signator*

⚠ Matches rule MAL_Ryuk_Ransomware by Florian Roth from ruleset crime_ryuk_ransomware at https://github.com/Neo23x0/signature-base
↳ *Detects strings known from Ryuk Ransomware*

These rules are discussed more in the Indicators of Compromise section. Further, the magic number of the binary indicates that it's a portable executable.

Packing

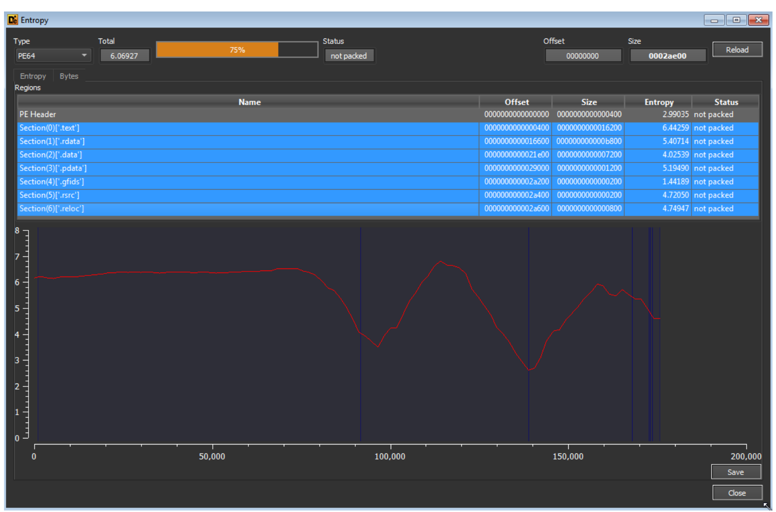
Some of the observations discussed below seem to indicate that the binary is not packed. Analyzing the ransomware in Detect It Easy doesn't show a presence of a packer, but rather the compiler and linker used in crafting the malware.



Both PEStudio and Detect It Easy show normal levels of entropy for each section of the binary.

property	value	value	value	value	value	value	value
name	.text	.rdata	.data	.pdata	.gfids	.rsrc	.reloc
md5	1D450E0B9A3BD6EC4DD274...	E6D779193C8692FB7A36658...	C8E644378A54CAD515E9A2...	E1DF62B9019A7718D9783DF...	0D8AE4CC74966073A7ABAC...	1B99276507C6356B24A31F6...	531E0A90A6501EB254FC829...
entropy	6.443	5.407	4.025	5.195	1.439	4.718	4.749
file-ratio (99.42%)	51.60 %	26.82 %	16.62 %	2.62 %	0.29 %	0.29 %	1.17 %
raw-address	0x00000400	0x00016600	0x00021E00	0x00029000	0x0002A200	0x0002A400	0x0002A600
raw-size (174592 bytes)	0x00016200 (90624 bytes)	0x0000B800 (47104 bytes)	0x00007200 (29184 bytes)	0x00001200 (4608 bytes)	0x00000200 (512 bytes)	0x00000200 (512 bytes)	0x00000800 (2048 bytes)
virtual-address	0x0000000040001000	0x0000000040018000	0x0000000040024000	0x0000000040031000	0x0000000040033000	0x0000000040034000	0x0000000040035000
virtual-size (194164 bytes)	0x000161F0 (90608 bytes)	0x0000B700 (46848 bytes)	0x0000C2F8 (49912 bytes)	0x000011F4 (4596 bytes)	0x000000A8 (168 bytes)	0x000001E0 (480 bytes)	0x00000610 (1552 bytes)
entry-point	0x00008624	-	-	-	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040	0x40000040	0x40000040	0x42000040

The entropy for the entire binary is also graphed with Detect It Easy, showing how the level of entropy changes linearly for each section (separated by the dark blue vertical bar). The entropy for each section is below 7.0, the total entropy of the binary is around 6.0, and the status indicates this as not being packed.



We can further compare the virtual size of each section with its raw size. One indication of a packed binary would be a section having a much larger virtual size than its raw size counterpart. In CFF Explorer, we can compare each section with its virtual and raw size counterparts.

Name	Virtual Size	Virtual Address	Raw Size
Byte[8]	Dword	Dword	Dword
.text	000161F0	00001000	00016200
.rdata	0000B700	00018000	0000B800
.data	0000C2F8	00024000	00007200
.pdata	000011F4	00031000	00001200
.gfids	000000A8	00033000	00000200
.rsrc	000001E0	00034000	00000200
.reloc	00000610	00035000	00000800

As can be seen above, the virtual size and raw size of each section appear to be close in value, a further indication that the binary is not packed.

Finally, the number of strings included with this sample numbered almost to three thousand, which can be seen in the upper right-hand corner of the PEStudio screenshot below.

type (2)	size (bytes)	file-offset	blacklist (30)	hint (248)	group (12)	value (2742)
ascii	120	0x0001FDB4	x	file	-	C:\Users\Admin\Documents\Visual Studio 2015\Projects From Ryuk\ConsoleApplication54\x64\Release\ConsoleApplication54.pdb
ascii	16	0x00021813	x	-	security	LookupAccountSid
ascii	15	0x00021826	x	-	security	OpenThreadToken
ascii	16	0x00021838	x	-	security	OpenProcessToken
ascii	21	0x0002185E	x	-	security	AdjustTokenPrivileges
ascii	20	0x00021877	x	-	security	LookupPrivilegeValue
ascii	18	0x000215F2	x	-	memory	WriteProcessMemory
ascii	10	0x0001E741	x	-	file	MoveFileEx
ascii	10	0x00021719	x	-	file	DeleteFile
ascii	15	0x00021BC5	x	-	file	FindFirstFileEx
ascii	12	0x00021BD9	x	-	file	FindNextFile
ascii	11	0x00021692	x	-	execution	OpenProcess
ascii	24	0x000216A0	x	-	execution	CreateToolhelp32Snapshot
ascii	13	0x000216D5	x	-	execution	Process32Next
ascii	16	0x000216E6	x	-	execution	GetCurrentThread
ascii	14	0x00021727	x	-	execution	Process32First
ascii	18	0x000217C8	x	-	execution	CreateRemoteThread
ascii	12	0x0002189D	x	-	execution	ShellExecute
ascii	12	0x000218C3	x	-	execution	ShellExecute
ascii	16	0x0002195A	x	-	execution	TerminateProcess
ascii	19	0x000219A4	x	-	execution	GetCurrentProcessId
ascii	18	0x000219BA	x	-	execution	GetCurrentThreadId
ascii	21	0x00021C25	x	-	execution	GetEnvironmentStrings
ascii	17	0x000218DE	x	-	exception-handling	RtlCaptureContext
ascii	14	0x00021A48	x	-	exception-handling	RaiseException
ascii	17	0x0002164B	x	-	dynamic-library	GetModuleFileName
ascii	17	0x00021B1D	x	-	dynamic-library	GetModuleHandleEx
ascii	17	0x00021B4F	x	-	dynamic-library	GetModuleFileName
ascii	22	0x000218F2	x	-	diagnostic	RtlLookupFunctionEntry
ascii	17	0x00021CEE	x	-	cryptography	SystemFunction036

Packed binaries usually contain very few strings and imports as a way to obscure its behavior during static analysis. The larger number of strings included here indicates that the malware is not packed.

Compilation Date and Subsystem

The apparent compilation date of the malware is shown below.

file-type	**executable**
cpu	**64-bit**
subsystem	GUI
compiler-stamp	0x5B72C112 (Tue Aug 14 06:46:26 2018)
debugger-stamp	0x5B72C112 (Tue Aug 14 06:46:26 2018)

According to PEStudio, the malware was compiled in August of 2018, which is consistent with with its initial appearance in May of 2018. With the appearance of this variant showing a compilation date a few months later than its first appearance, we can make the assumption that this binary is a later variant of the same strain.

Program Imports

There were three libraries included with this malware, totalling in 95 imports between all the libraries. Most imports originate from *kernel32.dll*. The other two libraries are *advapi32.dll* and *shell32.dll*. Each library, along with its corresponding imports, are shown below.

```
b'KERNEL32.dll'
        b'OpenProcess'
        b'CreateToolhelp32Snapshot'
        b'Sleep'
        b'GetLastError'
        b'Process32NextW'
        b'GetCurrentThread'
        b'LoadLibraryA'
        b'GlobalAlloc'
        b'DeleteFileW'
        b'Process32FirstW'
        b'GetModuleHandleA'
        b'CloseHandle'
        b'HeapAlloc'
        b'GetWindowsDirectoryW'
        b'GetProcAddress'
        b'VirtualAllocEx'
        b'LocalFree'
        b'GetProcessHeap'
        b'FreeLibrary'
        b'CreateRemoteThread'
        b'VirtualFreeEx'
        b'GetVersionExW'
        b'CreateFileW'
        b'GetModuleFileNameW'
        b'GetCurrentProcess'
        b'GetCommandLineW'
        b'SetLastError'
        b'HeapFree'
b'GlobalFree'
b'WriteConsoleW'
b'SetFilePointerEx'
b'HeapReAlloc'
b'HeapSize'
b'RtlCaptureContext'
b'RtlLookupFunctionEntry'
b'RtlVirtualUnwind'
b'UnhandledExceptionFilter'
b'SetUnhandledExceptionFilter'
b'TerminateProcess'
b'IsProcessorFeaturePresent'
b'QueryPerformanceCounter'
b'GetCurrentProcessId'
b'GetCurrentThreadId'
b'GetSystemTimeAsFileTime'
b'InitializeSListHead'
b'IsDebuggerPresent'
b'GetStartupInfoW'
b'RaiseException'
b'InitializeCriticalSectionAndSpinCoun
b'TlsAlloc'
b'TlsGetValue'
b'TlsSetValue'
b'TlsFree'
b'LoadLibraryExW'
b'EnterCriticalSection'
b'LeaveCriticalSection'
b'DeleteCriticalSection'
b'ExitProcess'
b'GetModuleHandleExW'
b'GetStdHandle'
b'WriteFile'
b'GetModuleFileNameA'
b'MultiByteToWideChar'
b'WideCharToMultiByte'
b'GetACP'
b'LCMapStringW'
b'GetFileType'
b'FindClose'
b'FindFirstFileExA'
b'FindNextFileA'
b'IsValidCodePage'
b'GetOEMCP'
b'GetCPInfo'
b'GetCommandLineA'
b'GetEnvironmentStringsW'
b'FreeEnvironmentStringsW'
b'SetStdHandle'
b'GetStringTypeW'
b'FlushFileBuffers'
b'GetConsoleCP'
b'GetConsoleMode'
b'WriteProcessMemory'
```

```
b'ADVAPI32.dll'
        b'SystemFunction036'
        b'LookupPrivilegeValueW'
        b'AdjustTokenPrivileges'
        b'ImpersonateSelf'
        b'OpenProcessToken'
        b'OpenThreadToken'
        b'LookupAccountSidW'
        b'GetTokenInformation'
b'SHELL32.dll'
        b'CommandLineToArgvW'
        b'ShellExecuteW'
        b'ShellExecuteA'
```

There are a few API calls that stand out as being suspicious. The combination of *CreateToolhelp32Snapshot*, *Process32First*, and *Process32Next* are typical in malware that attempt to bypass debugging attempts under a virtual machine by searching through a list of processes. The use of *LookupAccountSid* is likely used alongside the previous three in an attempt to locate a particular process or service typically associated with virtual machine artifacts.

Additionally, the use of *GetCurrentThread*, *OpenProcess*, *OpenProcessToken*, *OpenThreadToken*, *AdjustTokenPrivileges* and *ImpersonateSelf* are typically used in combination in an attempt of privilege escalation by elevating its access token.

Finally, *ShellExecute* is used to perform operations on a given file, such as editing , finding, running (as administrator), etc.

Suspicious Strings

There didn't appear to be any URL links, but there were several strings referencing relative file locations and some that appeared to be command-line commands. Particularly included in the image below, there is a string that appears to stop Sophos System Protection Service. Sophos is security software included in Windows-based enterprise systems. This may be targeted at a particular company that is known to run Sophos, or it could be generically used if it's popular in the industry.

There were several other strings that seemed to stop several different services, such as McAfee, Sophos File Scanner, Enterprise, Client Service, SQLsafe Backup Service, and much more. The relative file directories appear to be centered around teh user directory.

t\Documents and Settings\Default User\finish
stop "Veeam Backup Catalog Data Service" /y
`eh vector vbase copy constructor iterator'
stop "Sophos System Protection Service" /y
`managed vector copy constructor iterator'
api-ms-win-security-systemfunctions-l1-1-0
ext-ms-win-kernel32-package-current-l1-1-0
!This program cannot be run in DOS mode.
`vector vbase copy constructor iterator'
\Documents and Settings\Default User\sys
stop "Sophos Device Control Service" /y
stop MSSQLFDLauncher$PROFXENGAGEMENT /y
stop McAfeeFrameworkMcAfeeFramework /y
`eh vector vbase constructor iterator'

There was also a large string, almost three thousand characters in length, that appeared to be encrypted. This seems like it could be used as the unique identifier (discussed later) that is used to identify which decryption algorithm the attackers need to use to decrypt a user's system in the event the ransom is paid.

type (2)	size...	file-offset	hint (2...	value (2742)
ascii	2944	0x0001F040	size	QGDWqAqQzuTgzpJePKdAcDauoXPOTKYZQSFKBsJYKUYLhAQOiQFduITFKqvyiMwOIzzTjhuNbJVzaxSOtnNzbqNDUWUKVuSejHjMyOVTzsKKFiZHcXmvDFBPLbyknfMKQAfymCrJmtgKnlujwDwdohjBsumNHZSueZRNntZOoyNJwTVmevyjoGiYIbSJzfZIKHqvuUs
ascii	1560	0x0001E758	size	vssadmin Delete Shadows /all /quiet\r\nvssadmin resize shadowstorage /for=c: /on=c: /maxsize=401MB\r\nvssadmin resize shadowstorage /for=c: /on=c: /maxsize=unbounded\r\nvssadmin resize shadowstorage /for=d: /on=d: /maxsize=401MB\r\nvssad
ascii	401	0x0002A44C	-	<?xml version='1.0' encoding='UTF-8' standalone='yes'?>\r\n<assembly xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>\r\n <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">\r\n <security>\r\n <requestedPrivileges>\r\
ascii	120	0x0001FDB4	file	C:\Users\Admin\Documents\Visual Studio 2015\Projects From Ryuk\ConsoleApplication54\x64\Release\ConsoleApplication54.pdb
ascii	110	0x0001EFD0	-	AamRlebaQnPcrdUQxfhhMSyJVzJEQzLgXzEwnmOhTfTNJqEwddvoCIwCjcgmSoqCKoSdcCKfRuzzsCLwEgXFJVCqmHxSBLWemHjvKtyLPNZFsy
unicode	103	0x0001EE08	utility	/C REG ADD "HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /v "svchos" /t REG_SZ /d "
ascii	95	0x00018970	-	!"#$%&'()*+,-./0123456789:;<=>?@abcdefghijklmnopqrstuvwxyz[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
ascii	95	0x00018AF0	-	!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`ABCDEFGHIJKLMNOPQRSTUVWXYZ{|}~

PE Header Sections

The sections included with the binary are *.text*, *.rdata*, *.data*, *.pdata*, *.gfids*, *.rsrc*, and *.reloc*.

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	000161F0	00001000	00016200	00000400	00000000	00000000	0000	0000	60000020
.rdata	0000B700	00018000	0000B800	00016600	00000000	00000000	0000	0000	40000040
.data	0000C2F8	00024000	00007200	00021E00	00000000	00000000	0000	0000	C0000040
.pdata	000011F4	00031000	00001200	00029000	00000000	00000000	0000	0000	40000040
.gfids	000000A8	00033000	00000200	0002A200	00000000	00000000	0000	0000	40000040
.rsrc	000001E0	00034000	00000200	0002A400	00000000	00000000	0000	0000	40000040
.reloc	00000610	00035000	00000800	0002A600	00000000	00000000	0000	0000	42000040

The typical sections that are included with most binaries are *text*, which mostly contains executable code, *rdata*, containing read only data, *data*, containing globally accessible data, *rsrc*, containing resources, *reloc*, containing relocation information, and *pdata*, containing exception handling information (typically present in 64-bit executables). The *gfids* section is sometimes included with Visual Studio applications and is normal (its purpose is apparently unknown).

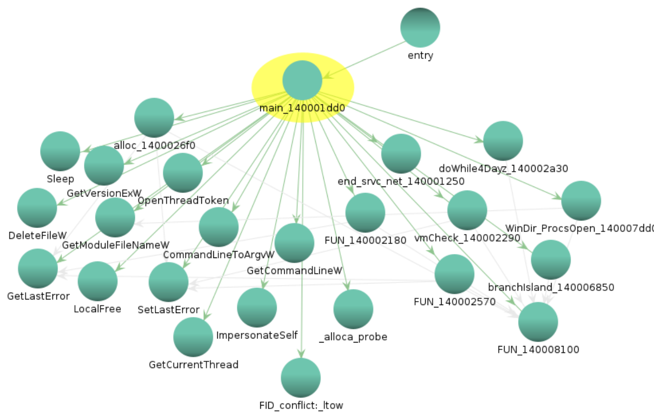
Anti-Disassembly and Reverse Engineering

There appeared to be no anti-disassembly techniques used during analysis. However, there did appear to be evidence of anti-virtial machine usage, which will be discussed shortly.

The main function was determined to be located in the location shown in the screenshot below.

```
                      LAB_14000859e                                   XRE
14000859e 33 c9          XOR      ECX, ECX
1400085a0 e8 ef 08 ...   CALL     FUN_140008e94
1400085a5 e8 92 05 ...   CALL     __scrt_get_show_window_mode
1400085aa 0f b7 d8       MOVZX    EBX, AX
1400085ad e8 5e 39 ...   CALL     _get_narrow_winmain_command_line
1400085b2 4c 8b c0       MOV      R8, RAX
1400085b5 44 8b cb       MOV      R9D, EBX
1400085b8 33 d2          XOR      EDX, EDX
1400085ba 48 8d 0d ...   LEA      RCX, [IMAGE_DOS_HEADER_140000000]
1400085c1 e8 0a 98 ...   CALL     main_140001dd0
1400085c6 8b d8          MOV      EBX, EAX
1400085c8 33 c9          XOR      ECX, ECX
1400085ca e8 dd 09 ...   CALL     FUN_140008fac
1400085cf e8 a8 05 ...   CALL     __scrt_is_managed_app
1400085d4 84 c0          TEST     AL, AL
1400085d6 75 07          JNZ      LAB_1400085df
1400085d8 8b cb          MOV      ECX, EBX
1400085da e8 89 17 ...   CALL     FUN_140009d68
```

A graphical view of the imports called along with custom functions is also shown below. Most of the custom functionality that is called from main is located to the right whereas the various library import functionality is located to the left.



The main program begins by retrieving a pointer to the string of the current process then proceeds to retrieve an array of command line pointers along with the count of arguments (similar to *argv* and *argc*), deleting all files that are retrieved from the command-line of the current process followed by freeing its memory.

```
47  local_30 = 0x140001df2;
48  local_40 = DAT_140024000 ^ (ulonglong)auStack792752;
49  Sleep(5000);
50  lpCmdLine = GetCommandLineW();
51  hMem = CommandLineToArgvW(lpCmdLine,aiStack792712);
52  if (hMem != (LPWSTR *)0x0) {
53    DeleteFileW(hMem[1]);
54  }
55  LocalFree(hMem);
56  end_srvc_net_140001250();
57  uVar22 = 0;
58  lVar11 = 4;
59  p_Var3 = &_Stack792704;
```

The following function called on line 56 appears to kill and stop two potential services, which are *Zoolz.exe* and *Acronis VSS Provider*. This is accomplished by using *ShellExecuteA()*. The API searches for *taskkill* and *net* to execute on each task or service, respectively. The program iterates through each service until the ones in question are found.

```
void end_srvc_net_140001250(void)

{
  char *service;
  longlong iterator;

  service = s_/IM_zoolz.exe_/F_14002a070;
  iterator = 44;
  do {
    ShellExecuteA((HWND)0x0,(LPCSTR)0x0,"taskkill",service,(LPCSTR)0x0,0);
    service = service + 0x32;
    iterator = iterator + -1;
  } while (iterator != 0);
  service = s_stop_"Acronis_VSS_Provider"_/y_140024ab0;
  iterator = 184;
  do {
    ShellExecuteA((HWND)0x0,(LPCSTR)0x0,"net",service,(LPCSTR)0x0,0);
    service = service + 0x4b;
    iterator = iterator + -1;
  } while (iterator != 0);
  return;
}
```

Following this the program acquires various operating system information using the *OSVersionInfoW()* API call. Such information includes the major/minor version, build number, platform ID, and the latest Service Pack installed.

The function on line 79 locates the Windows directory location for the system and views the various processes open.

The program then gets the current thread, opens the token for the current thread, then impersonates the security context of the calling process to obtain the access token. This is acquired using a combination of *GetCurrentThread*, *OpenThreadToken*, and *ImpersonateSelf*. This is likely an attempt at privilege escalation by elevating its current security context.

```
GetVersionExW((LPOSVERSIONINFOW)&_Stack792704);
DVar20 = 1;
WinDir_ProcsOpen_140007dd0();
SetLastError(0);
hCurThread = GetCurrentThread();
TokenHandle = &pvStack792720;
ProcessHandle = 0;
DesiredAccess = 0x28;
BVar4 = OpenThreadToken(hCurThread,0x28,0,TokenHandle);
if ((BVar4 == 0) && (DVar5 = GetLastError(), DVar5 == 0x3f0)) {
  ImpersonateSelf(SecurityImpersonation);
  hCurThread = GetCurrentThread();
  TokenHandle = &pvStack792720;
  ProcessHandle = 0;
  DesiredAccess = 0x28;
  OpenThreadToken(hCurThread,0x28,0,TokenHandle);
}
FUN_140002180(pvStack792720,DesiredAccess,ProcessHandle,TokenHandle);
lVar10 = 0xc15c0;
pwVar16 = awStack792416;
while (lVar10 != 0) {
  lVar10 = lVar10 + -1;
  *(undefined *)pwVar16 = 0;
  pwVar16 = (wchar_t *)((longlong)pwVar16 + 1);
}
```

Immediately following there is a function that appears to follow a number of steps that is typical in bypassing VMware by searching for particular artifacts. This is evidenced by the use of *CreateToolhelp32Snapshot*, *Process32First*, and *Process32Next*. This combination is used to search through a list of processes running on the system, which may reveal the use of a Virtual Machine if a process unique to such environments are detected. The parameter provided by CreateToolhelp32Snapshot is 2, which includes all the processes on the system (Process32First and Process32Next are used to iterate through each process).

The program then proceeds to check the process handles and tokens of the processes that are iterated through.

```
hSnapshot = (HANDLE)CreateToolhelp32Snapshot(2);
if ((hSnapshot != (HANDLE)0xffffffffffffffff) &&
   (bBufCopy = Process32FirstW(hSnapshot,lppe), bBufCopy != 0)) {
  bBufCopy = Process32NextW(hSnapshot,lppe);
  if (bBufCopy != 0) {
    puVar3 = (undefined4 *)(param_1 + 0x20c);
    TokenSID = ppvVar4;
    do {
      SetLastError(0);
      ProcessHandle = OpenProcess(0x1fffff,0,dwProcessId);
      if (ProcessHandle != (HANDLE)0x0) {
        wcsncpy((wchar_t *)((longlong)(int)ppvVar4 * 0x210 + param_1),local_24c,0x103);
        puVar3[-1] = dwProcessId;
        bProcessOpen = OpenProcessToken(ProcessHandle,0x20008,&TokenHandle);
        if (bProcessOpen == 0) {
          *puVar3 = 0;
        }
```

Further down within the same function, we see that *LookupAccountSidW* used. This takes in as input a security identifier (a unique identifier for accounts and processes on the network). The name of a domain is then received of the account found given the SID. The program then checks if the name of the domain at least starts with *NTA*.

This may possibly be checking for NetFlow Traffic Analyzer provided by SolarWinds for VMware vSphere. This check may be targeted toward a particular company known to run this service, providing the added benefit of remaining obscure if checking for this service is atypical among malware strains.

```
    LookupAccountSidW((LPCWSTR)0x0,*TokenSID,(LPWSTR)0x0,&cchName,(LPWSTR)0x0,
                      &cchReferencedDomainName,&peUse);
    Name = (LPWSTR)GlobalAlloc(0,(ulonglong)(cchName * 2));
    ReferencedDomainName = (LPWSTR)GlobalAlloc(0,(ulonglong)(cchReferencedDomainName * 2))
    ;
    LookupAccountSidW((LPCWSTR)0x0,*TokenSID,Name,&cchName,ReferencedDomainName,
                      &cchReferencedDomainName,&peUse);
    if (((*ReferencedDomainName == L'N') && (ReferencedDomainName[1] == L'T')) &&
       (ReferencedDomainName[3] == L'A')) {
      *puVar3 = 1;
    }
    else {
      *puVar3 = 2;
    }
    GlobalFree(ReferencedDomainName);
    GlobalFree(Name);
  }
```

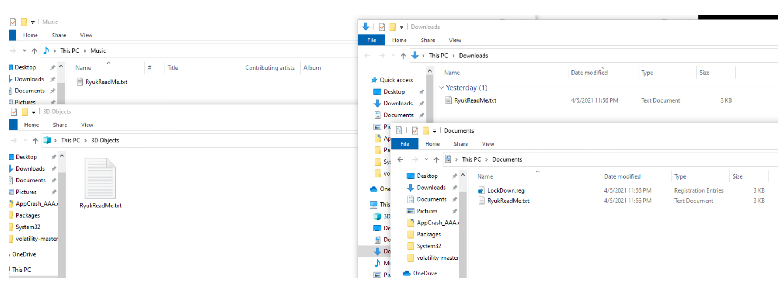
Dynamic Analysis

Notable Post-Execution Behavioral

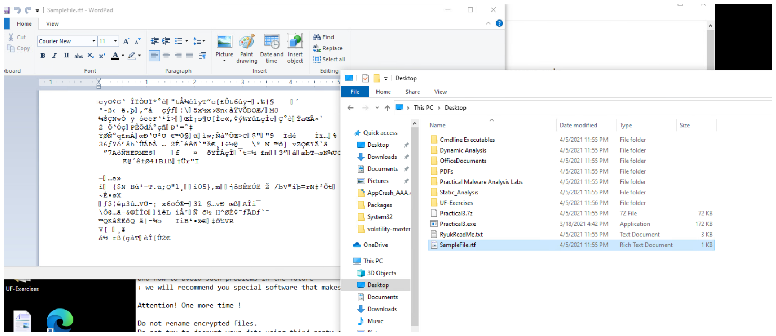
Upon execution, there were a few UAC prompts, followed by a notification that Microsoft Onedrive has stopped, a text file appearing on the desktop, a notification that the Microsoft account needs to be fixed ("most likely due to a password change"), and that the firewall has been turned off. The message contained in the text file was as follows:

```
Gentlemen!

Your business is at serious risk.
There is a significant hole in the security system of your company.
We've easily penetrated your network.
You should thank the Lord for being hacked by serious people not some stupid schoolboys or dangerous punks.
They can damage all your important data just for fun.

Now your files are crypted with the strongest millitary algorithms RSA4096 and AES-256.
No one can help you to restore files without our special decoder.

Photorec, RannohDecryptor etc. repair tools
are useless and can destroy your files irreversibly.

If you want to restore your files write to emails (contacts are at the bottom of the sheet)
and attach 2-3 encrypted files
(Less than 5 Mb each, non-archived and your files should not contain valuable information
(Databases, backups, large excel sheets, etc.)).
You will receive decrypted samples and our conditions how to get the decoder.
Please don't forget to write the name of your company in the subject of your e-mail.

You have to pay for decryption in Bitcoins.
The final price depends on how fast you write to us.
Every day of delay will cost you additional +0.5 BTC
Nothing personal just business

As soon as we get bitcoins you'll get all your decrypted data back.
Moreover you will get instructions how to close the hole in security
and how to avoid such problems in the future
+ we will recommend you special software that makes the most problems to hackers.

Attention! One more time !

Do not rename encrypted files.
Do not try to decrypt your data using third party software.

P.S. Remember, we are not scammers.
We don't need your files and your information.
But after 2 weeks all your files and keys will be deleted automatically.
Just send a request immediately after infection.
All data will be restored absolutely.
Your warranty - decrypted samples.

contact emails
leonardobrody@tutanota.com
or
ZanePayton@protonmail.com

BTC wallet:
17v2cu8RDXhAxufQ1YKiauBq6GGAZzfnFw

Ryuk
```

This same text file was also written to all the directories of the system. The screenshot of File Explorer below shows a few.



When attempting to open a file on the system, it was apparent that various documents had been encrypted.



Network Communication

There didn't appear to be any notable network activity.

Registry Keys Created or Modified

There were only a few registry keys that were modified.

Registry Keys Opened

<HKLM>\Software\Microsoft\WBEM\CIMOM

<HKCU>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Registry Keys Set

- <HKCU>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\UNCAsIntranet

  > 0x00000000

- <HKCU>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\AutoDetect

  > 0x00000001

- <HKCU>\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\svchos

  > <PATH_SAMPLE.EXE>

Registry Keys Deleted

<HKCU>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\ProxyBypass

<HKLM>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\ProxyBypass

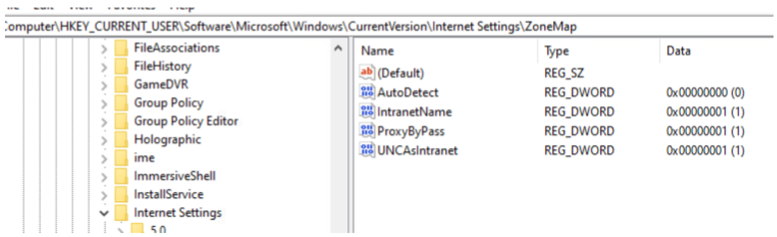
<HKCU>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\IntranetName

<HKLM>\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\\IntranetName

UNCAsIntranet is disabled, disallowing network paths to be mapped to the intranet zone. *AutoDetect* is enabled, allowing automatic detection of the intranet as long as a domain is connected. And *Run - svchos* seems to be a way to establish persistence.

The registry keys that are deleted are various rules regarding use of the intranet.

This wasn't able to be replicated when running the binary dynamically, though.



Both UNCAsIntranet and AutoDetect remained at their default value during dynamic analysis.

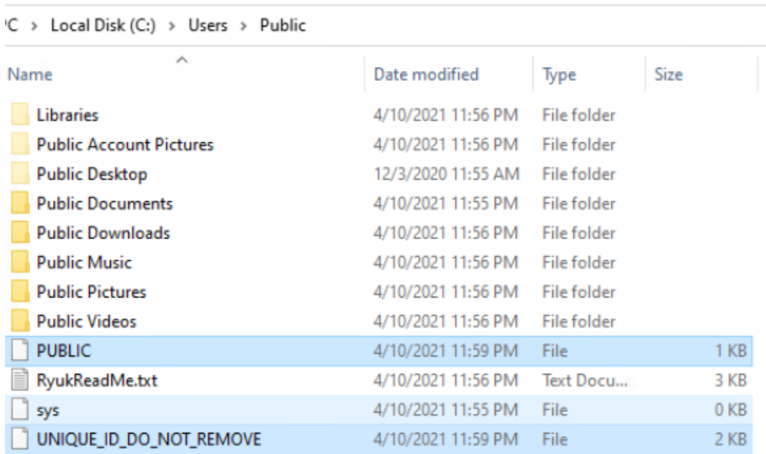
Files Created or Modified

According to VirusTotal, there were several files written to the drive.

Files Written

C:\users\public\public
C:\users\public\unique_id_do_not_remove
%ALLUSERSPROFILE%\microsoft\crypto\rsa\machinekeys\08e575673cce10c72090304839888e02_36d1130a-ac2e-44f7-9dc1-e424fbcbe0ee
D:\ryukreadme.txt
D:\$recycle.bin\ryukreadme.txt
D:\$recycle.bin\s-1-5-21-1960123792-2022915161-3775307078-1001\ryukreadme.txt
<SYSTEM32>\dwm.exe
C:\ryukreadme.txt
C:\documents and settings\ryukreadme.txt
C:\far2\ryukreadme.txt
C:\far2\addons\ryukreadme.txt
C:\far2\addons\colors\ryukreadme.txt
C:\Far2\Addons\Colors\Custom_Highlighting\black_from_Fonarev.reg
C:\Far2\Addons\Colors\Custom_Highlighting\black_from_july.reg
C:\Far2\Addons\Colors\Custom_Highlighting\black_from_Myodov.reg
C:\Far2\Addons\Colors\Custom_Highlighting\Colors_from_admin_essp_ru.reg
C:\Far2\Addons\Colors\Custom_Highlighting\Colors_from_Gernichenko.reg
C:\Far2\Addons\Colors\Custom_Highlighting\Colors_from_Sadovoj.reg
C:\Far2\Addons\Colors\Custom_Highlighting\Descript.ion
C:\Far2\Addons\Colors\Custom_Highlighting\dn_like.reg
C:\Far2\Addons\Colors\Custom_Highlighting\FARColors242.reg
C:\Far2\Addons\Colors\Custom_Highlighting\GreenMile.reg
C:\Far2\Addons\Colors\Custom_Highlighting\hell.reg
C:\Far2\Addons\Colors\Custom_Highlighting\import_colors.bat
C:\Far2\Addons\Colors\Custom_Highlighting\nc5pal2.reg
C:\Far2\Addons\Colors\Custom_Highlighting\Rodion_Doroshkevich.reg
C:\Far2\Addons\Colors\Custom_Highlighting\VaxColors.reg
C:\far2\addons\colors\custom_highlighting\ryukreadme.txt
C:\Far2\Addons\Colors\Default_Highlighting\black_from_Fonarev.reg
C:\Far2\Addons\Colors\Default_Highlighting\black_from_july.reg

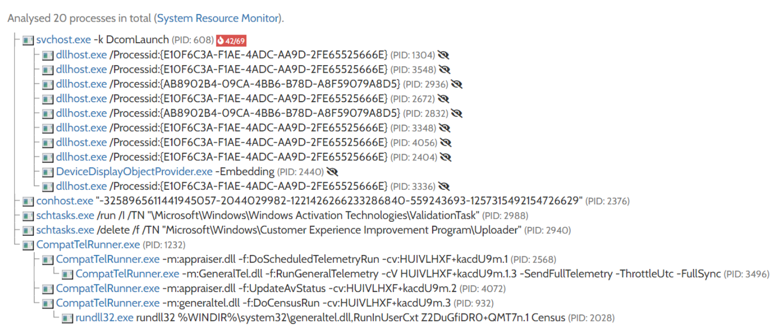
This is also expected as ransomware is notable for encrypting files, hard drives, etc. until a ransom is paid. A few unique files were the PUBLIC and UNIQUE _ID_DO_NOT_REMOVE. Trying to open the first file in Notepad yielded a prompt that the file is already in use. The second was encrypted. It's possible that the second file is used by the attackers in determining which decryption algorithm to use in the event that a ransom is paid.



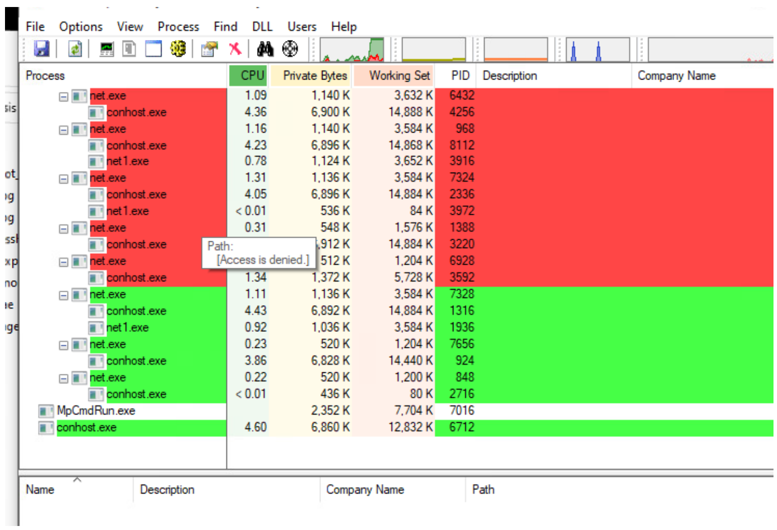
Most of the other files written to were dropping the ransom text file and various register keys.

Processes Started

There were several processes started by the malware, shown below.



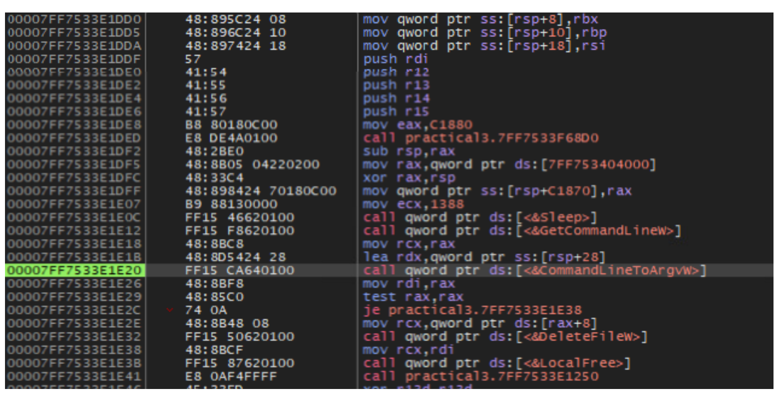
Most of these are likely an attempt to query kernel debugger information, particularly *dllhost.exe*, *CompatTelRunner.exe*, *DeviceDisplayObjectProvider.exe* and *CompatTelRunner.exe*. Also of note is *svchost.exe*, which is typically used in malware as an attempt to avoid detection.



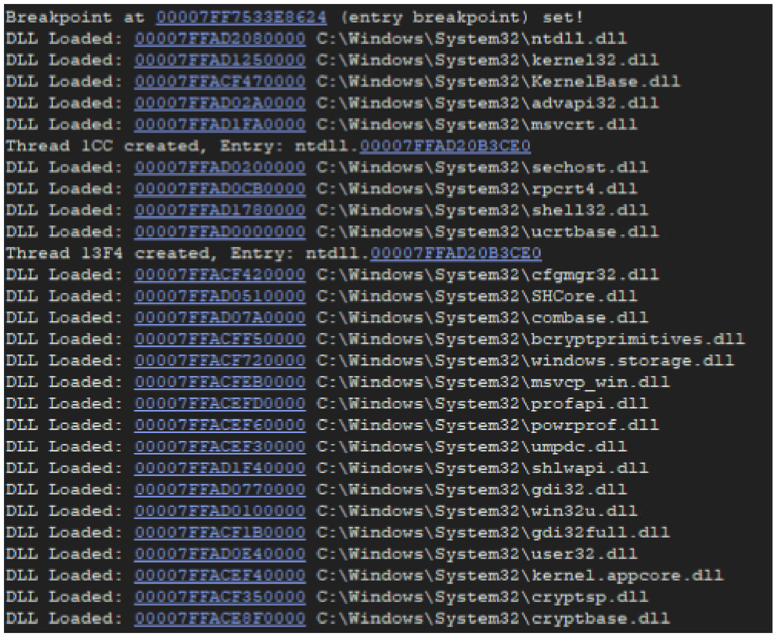
Watching Process Explorer while the malware is running will show several processes starting and quitting very quickly. Above is a screenshot of several new processes starting and some ending.

Debugging and Anti-Debugging

The first thing that was noted while debugging is that the address for what was determined to be main was the same when loaded into a debugger, indicating that the program wasn't rebased and can easily be used in tandem with Ghidra.

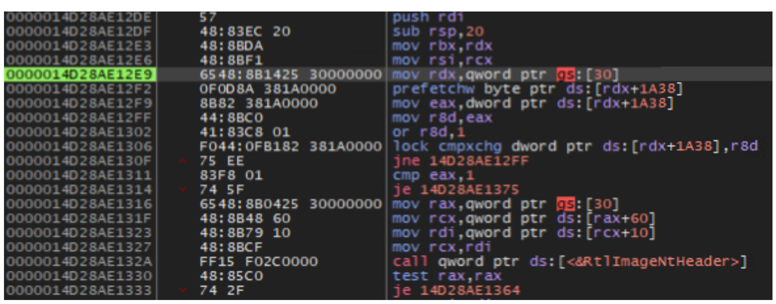


Something also to take note of are the DLLs that were loaded into memory by the time the debugger had reached the entry point of the program.

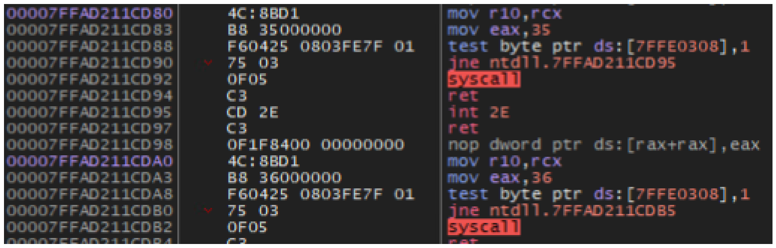


A few to take note of are *bcryptprimitives.dll*. *cryptsp.dll*, and *cryptbase.dll*, which are DLLs that are used in encryption schemes.

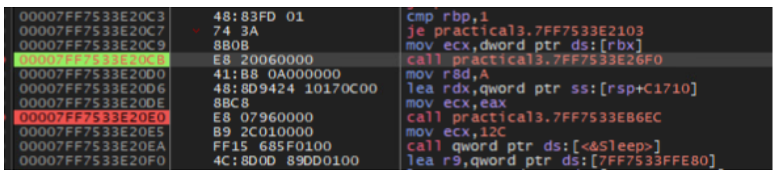
There appeared to be minimal anti-debugger usage in the program, but there were a few discovered. As was expected from the suspicious imports with the inclusion of *isDebuggerPresent*, there was a check if a debugger is present, gs:[30].



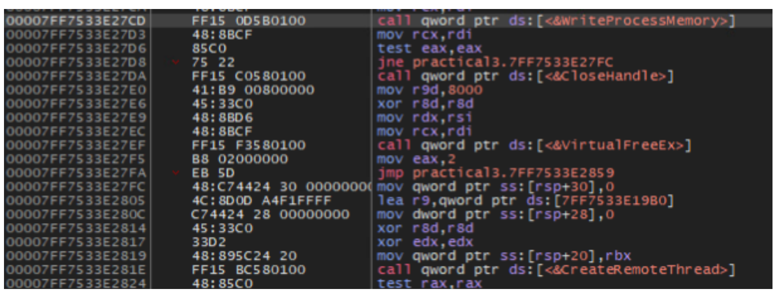
There was also the inclusion of *syscall*, which can be used to avoid detection for the use of certain tracing mechanisms.



While debugging, it was found that after stepping over the function shown below (at address 0x...E200CB) there were several UAC prompts, a text file written to the desktop, and files were encrypted.



Peering inside the function revealed several process calls and a call to *WriteProcessMemory*. There was also an allocation to virtual space, then a remote thread was created that runs in the virtual space just allocated.



What seems to be happening is that a handle of the module is grabbed, space is allocated for a remote running process, then WriteProcessMemory is used to encrypt the disk.

Indicators of Compromise

The most obvious and immediate indicator would be the ransom note written to the desktop (and several other directories) containing the wording shown in the initial behavior section at the beginning of dynamic analysis. Following this would be an observance of several encrypted files, the firewall being turned off, and several UAC prompts.

Below is one of the Yara rules found to match the binary that was submitted.

```
import "pe"

rule MAL_Ryuk_Ransomware {
   meta:
      description = "Detects strings known from Ryuk Ransomware"
      author = "Florian Roth"
      reference = "https://research.checkpoint.com/ryuk-ransomware-targeted-campaign-b
      date = "2018-12-31"
      hash1 = "965884f19026913b2c57b8cd4a86455a61383de01dabb69c557f45bb848f6c26"
      hash2 = "b8fcd4a3902064907fb19e0da3ca7aed72a7e6d1f94d971d1ee7a4d3af6a800d"
   strings:
      $x1 = "/v \"svchos\" /f" fullword wide
      $x2 = "\\Documents and Settings\\Default User\\finish" fullword wide
      $x3 = "\\users\\Public\\finish" fullword wide
      $x4 = "lsaas.exe" fullword wide
      $x5 = "RyukReadMe.txt" fullword wide
   condition:
      uint16(0) == 0x5a4d and filesize < 400KB and (
         pe.imphash() == "4a069c1abe5aca148d5a8fdabc26751e" or
         pe.imphash() == "dc5733c013378fa418d13773f5bfe6f1" or
         1 of them
      )
}
```

It looks for several strings, including relative directory paths and for the ransom text file dropped in several directories.

The Yara rule I created is below.

Yara Rule

```
rule Ryuk_String {
   meta:
   description = "Ryuk Ransomware"
   author = "Josh"
   strings:
   $a = "RyukReadMe.txt"
   $b = "QGDWqAqQzuTgzpJePKdAcDauoXPOTKYZQSFKBsJYKUYLhAQOiQFdulTFKqvyiMwOIzzTjhuNbJVzaxSOtnNzbqNDUWU

   condition:
   $a and $b
}
```

It includes a string of the ransom text along with the three thousand character string found during static analysis. For ease of copy/pasting, here's a link to the unique id (since the whole string doesn't fit in the document).

Sources

Hybrid Analysis
VirusTotal
Intezer Analyze