

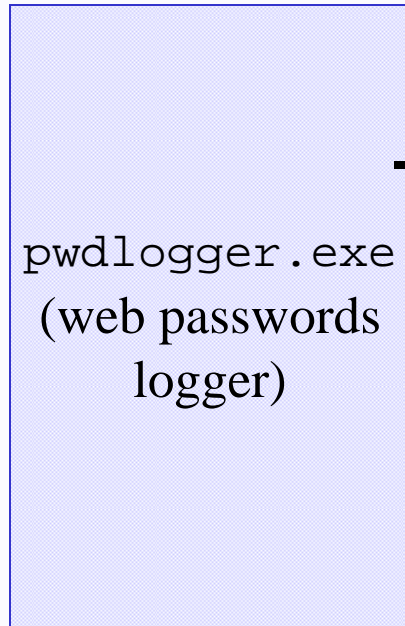
# *Rootkits Detection on Windows Systems*

Joanna Rutkowska

joanna@invisiblethings.org

ITUnderground Conference,  
October 12<sup>th</sup> -13<sup>th</sup> 2004, Warsaw

# *Sample scenario...*



compromised system

email with sniffed passwords  
encoded using some text based  
steganography techniques

# *Function hooking...*

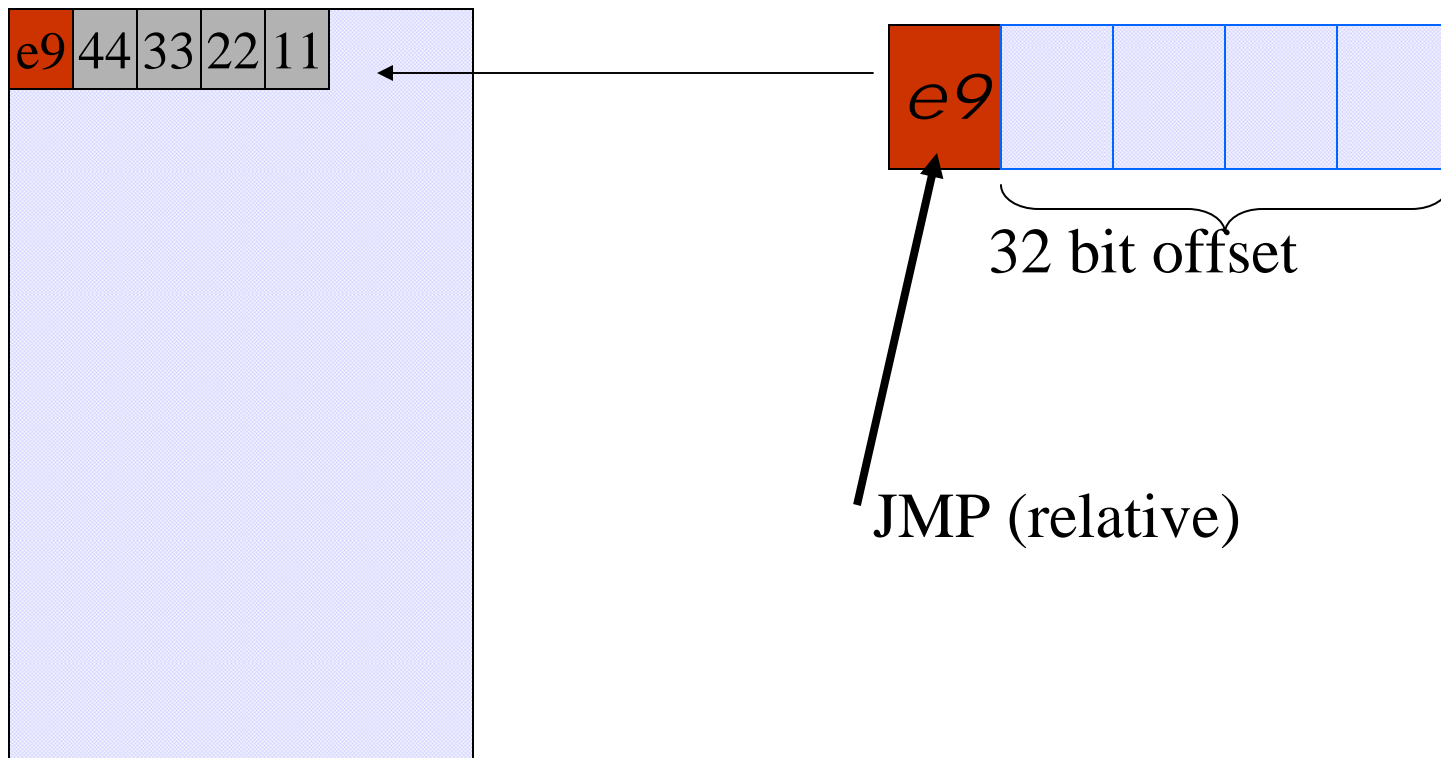
- Logon passwords sniffing

- Redirection:

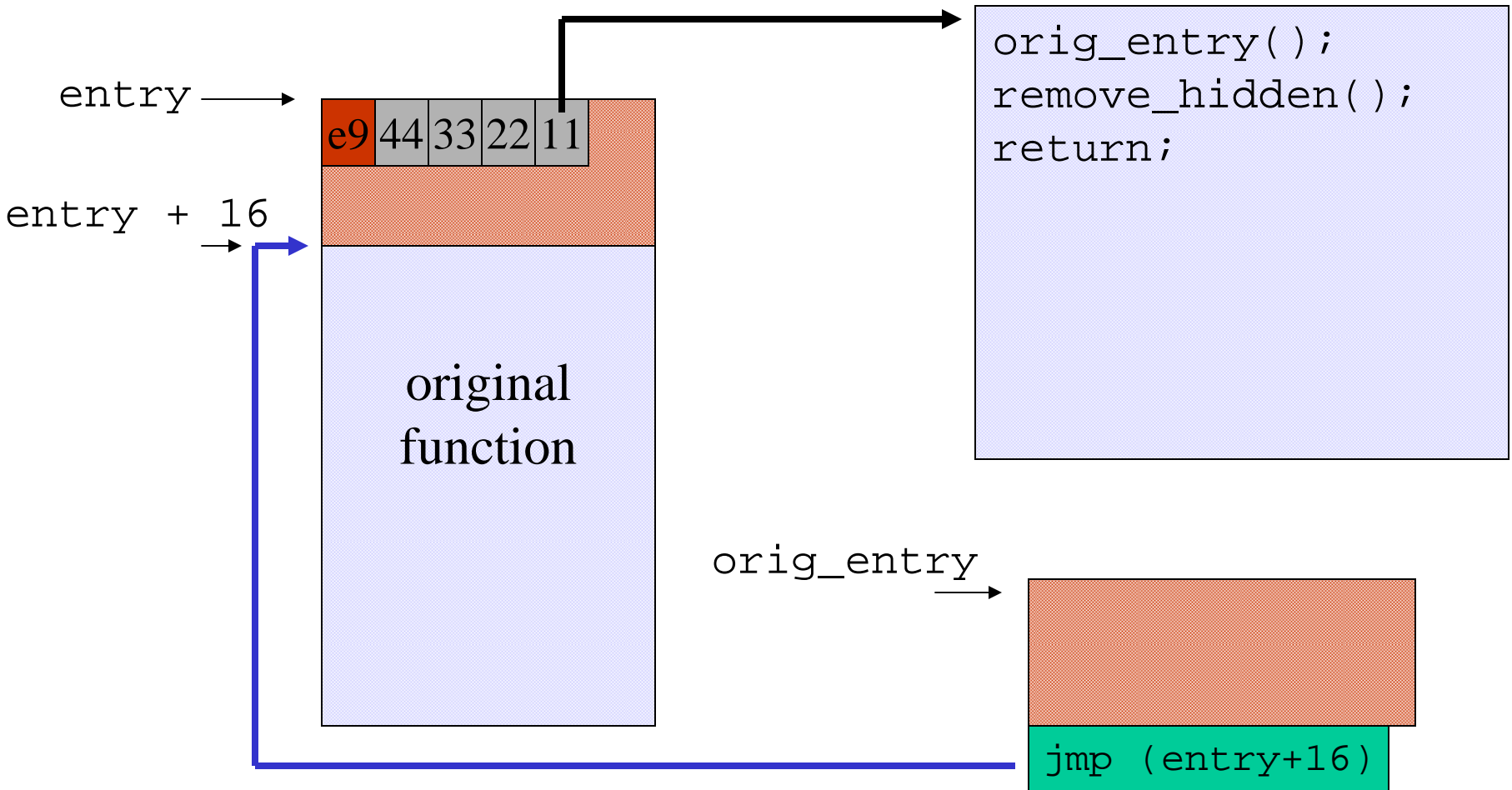
`msgina.dll! WlxLoggedOutSAS( )` to rootkit function, which logs the passwords

- ...and sends them to the attacker (using CC)

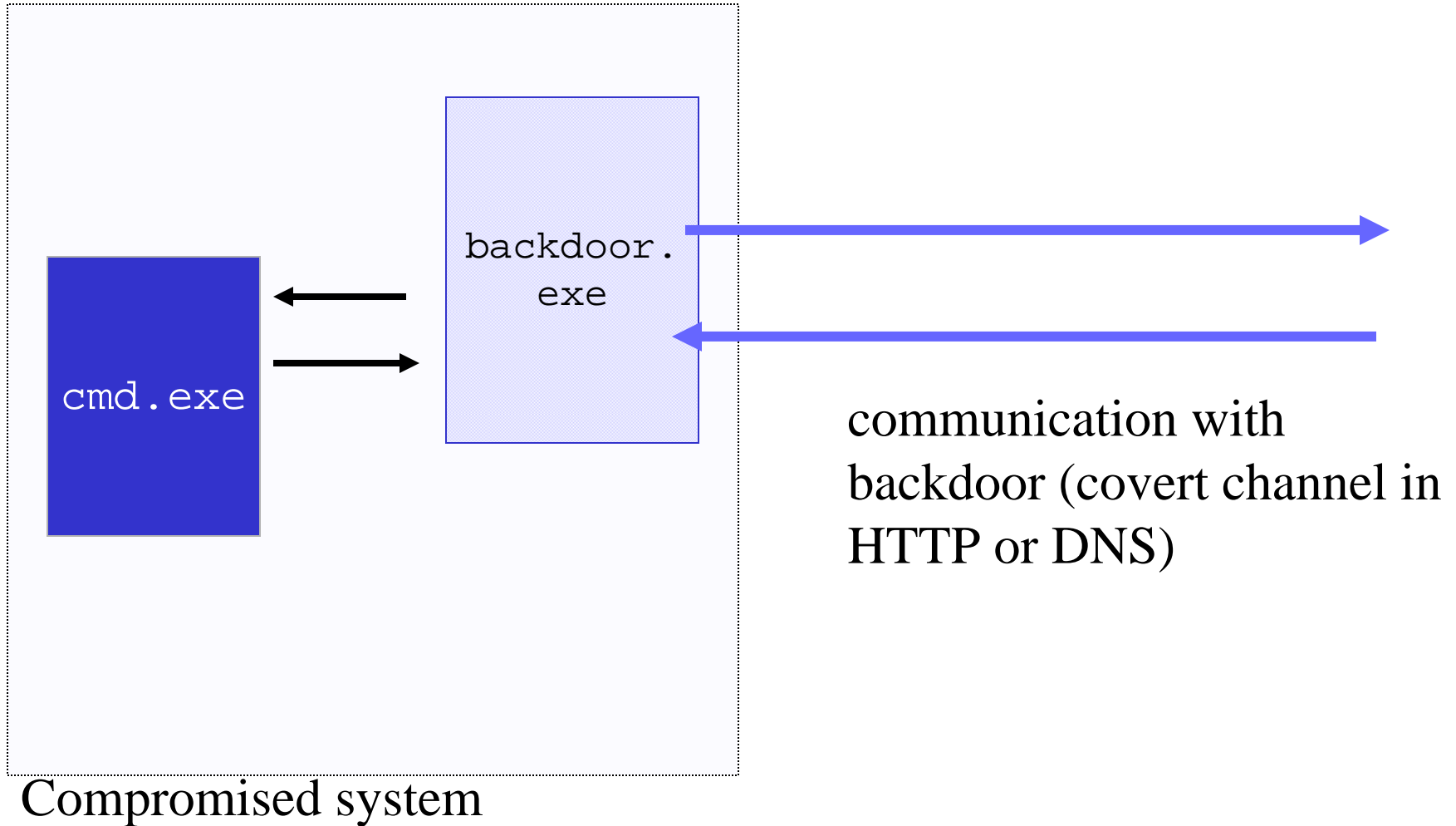
# Function hooking



# Hooking with original code saved



# *Another example*



# *Rootkit goals*

- Hide intruder's processes (`pwdlogger.exe`, `backdoor.exe`, etc...)
- Hide registry keys responsible for starting intruder's tools after system reboot
- Sometimes to hide some files (intruder's tools).

# *Live DEMO: Simple rootkit*

## ■ VANQUISH

# MS Kernel Debugger

- *MS Debugging Tools* can be freely downloaded from:
  - [www.microsoft.com/whdc/devtools/debugging/](http://www.microsoft.com/whdc/devtools/debugging/)
- Powerful debugger not only for catching rootkits;)
- Allows
  - ⊕ usermode processes debugging
  - ⊕ kernel debugging
- To debug kernel we need to:
  - start system with /debug switch (I.e. requires reboot on a production system)
  - use *Livekd* tool from [sysinternals.com](http://sysinternals.com) (does not require reboot)

# *Live DEMO: Detection of simple rootkit*

```
C:\>pslist winlogon
```

Name	Pid	Pri	Thd	Hnd	Mem	User Time	(...)
winlogon	456	13	18	478	4412	0:00:00.359	(...)

```
C:\>cdb.exe -p 456
```

```
Microsoft (R) Windows Debugger Version 6.3.0011.2  
Copyright (c) Microsoft Corporation. All rights reserved.  
(...)
```

```
0:018> .exepath c:\Windows\system32
```

```
Executable image search path is: c:\Windows\system32
```

```
0:018> !chkimg -d msgina
```

```
75844b98-75844b9c 5 bytes - MSGINA!WlxLoggedOutSAS
```

```
[ 55 8b ec 83 ec:e9 c1 11 2a 8c ]
```

```
5 errors : msgina (75844b98-75844b9c)
```

```
0:018>
```

# Useful command: !chkimg

```
kd> !chkimg -d ntdll
77f42467-77f4246b 5 bytes - ntdll!ZwCreateFile
[ b8 27 00 00 00 : e9 21 24 06 08 ]
[ ... ]
```

original bytes

new bytes  
(inserted by  
rootkit)

starting address of  
the region which  
was altered (by  
rootkit)

```
kd> u 77f42467
ntdll!ZwCreateFile:
77f42467 e921240608
77f4246c ba0003fe7f
[...]
```

**imp**

**7ffa488d**

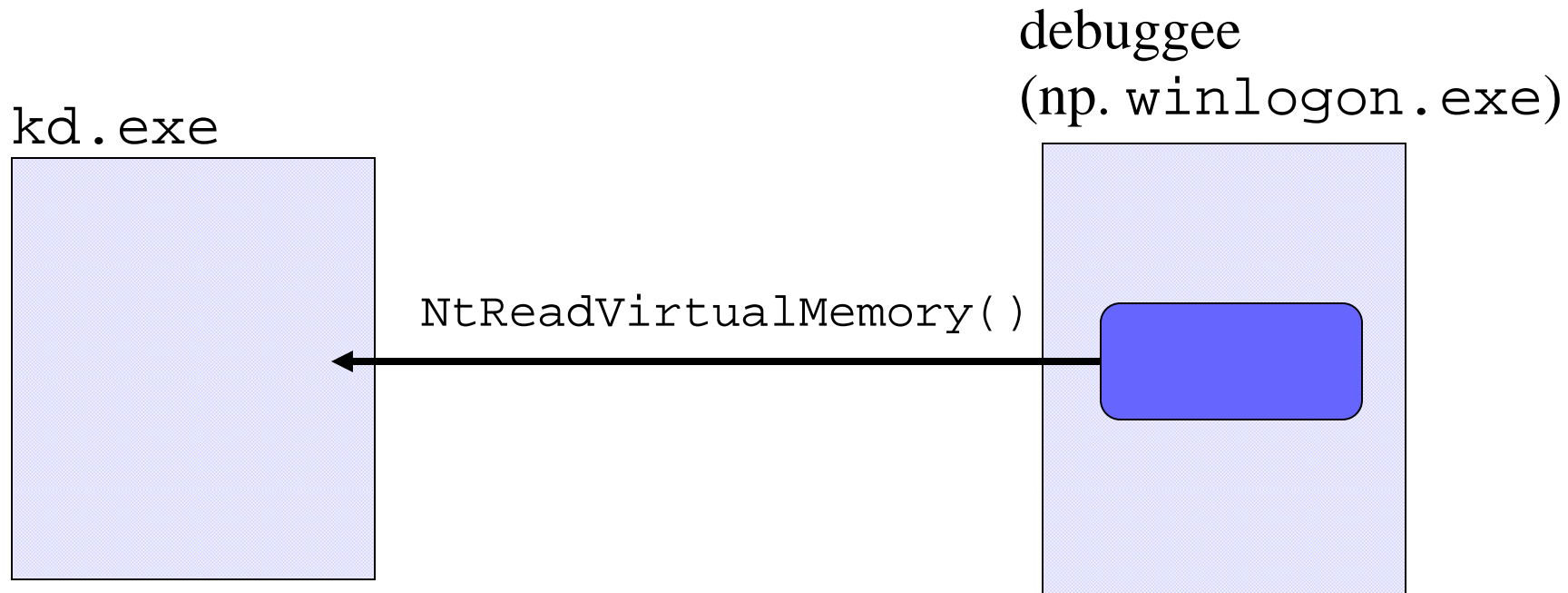
mov

edx,0x7ffe0300

Classic API hooking!

At address **7ffa488d** we should find a rootkit code.

# Debugger cheating



- Some rootkits are hooking `NtReadVirtualMemory()` to cheat debuggers, when they are reading processes memory.
- The same apply to dumping tools, like `pmdump.exe`

# *Live DEMO: rootkit which cheats debugger on the fly*

- Rootkit Hackder Defender

# *Livekd tool*

- Available at (free): `www.sysinternals.com`
- Recommended in MS documentation as an alternative to using `/debug switch`
- Allows for live kernel debugging on the same machine
- Creates virtual memory crash dump and spawns `kd.exe` to work on it
- Kernel memory is read by special kernel driver: `LiveKdD.sys`
- Everything works in READ-ONLY mode – seems to be a safe choice on a production system
- Obviously circumvents `NtReadVirtualMemory()` trick!

# Live DEMO: detection of yyt

```
kd> .exepath c:\windows\system32
Executable image search path is: c:\windows\system32
kd> !chkimg ntdll
25 errors : ntdll (77f426bf-77f42f7f)
kd> !chkimg -d ntdll
(...)
77f42737-77f4273b 5 bytes - ntdll!NtEnumerateKey (+0x78)
[ b8 4b 00 00 00:e9 65 f6 0b 98 ]
(...)
25 errors : ntdll (77f426bf-77f42f7f)
kd> u 77f42737
ntdll!NtEnumerateKey:
77f42737 jmp globalc!HooksCanUnloadNow+0xc2e(10001da1)
77f4273c mov edx,0x7ffe0300
(...)
kd>
```

# Detection of yyt rootkit (2)

ntdll!NtEnumerateKey() seems to be redirected to 0x10001da1 address (as seen on the previous slide). Lets take a look to which module it belongs...

```
kd> lm f a 10001da1
start      end          module name
10000000 10029000    globalc ↓
           C:\WINDOWS\system32\com\sserver\globalc.dll
kd>
```

We see that the suspicious code is within `globalc.dll` module, in the following directory:

`C:\WINDOWS\system32\com\sserver\`

Note: if the rootkit was not using DLL injection technique (as in this case), but rather a RAW memory change, we would not see any module for its code. But still we would be able to detect its presence, although finding the exactable would not be so easy...

# Detection of yyt rootkit (3)

```
C:\>dir C:\WINDOWS\system32\com\  
10/03/2004  03:59 PM      <DIR>          .  
10/03/2004  03:59 PM      <DIR>          ..  
03/25/2003  02:00 PM                189,440 comadmin.dll  
03/25/2003  02:00 PM                61,440 comempty.dat  
03/25/2003  02:00 PM                91,620 comexp.msc  
03/25/2003  02:00 PM                 8,704 comrepl.exe  
03/25/2003  02:00 PM                 5,632 comrereg.exe  
03/25/2003  02:00 PM                19,456 mtsadmin.tlb  
           6 File(s)                376,292 bytes  
           2 Dir(s)   1,518,030,848 bytes free  
  
C:\>
```

...it turns up that sserver\ directory is hidden...

# Detection of yyt rootkit (4)

...but we could enter it, since we know its name

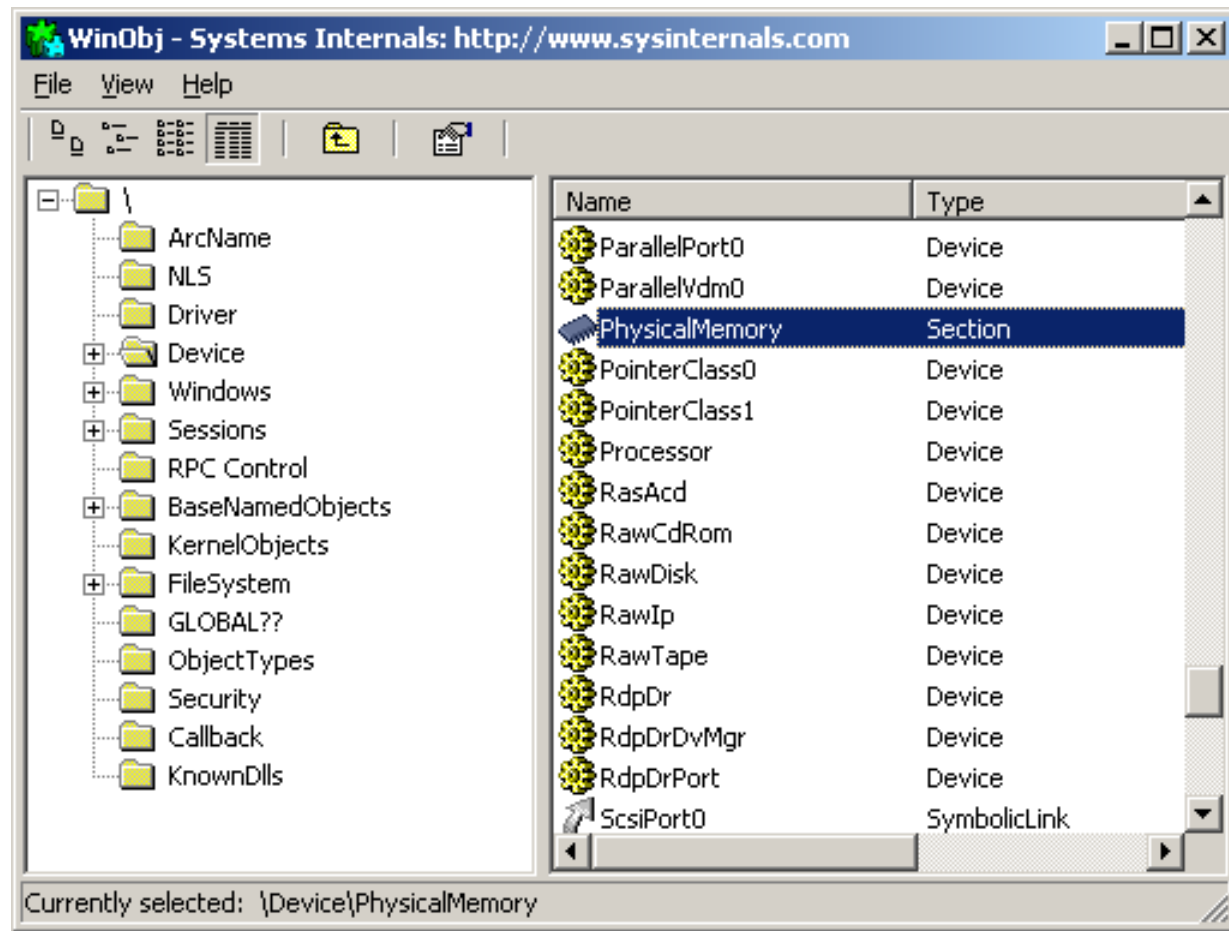
```
C:\>cd C:\WINDOWS\system32\com\sserver\  
  
C:\WINDOWS\system32\Com\sserver>dir  
10/03/2004  03:59 PM      <DIR>          .  
10/03/2004  03:59 PM      <DIR>          ..  
05/31/2004  02:43 PM                108,544 comine.exe  
10/03/2004  03:59 PM                135,168 globalc.dll  
10/03/2004  03:59 PM                 32,064 npf.sys  
10/03/2004  04:14 PM                 1,418 rtkit.log  
          4 File(s)                277,194 bytes  
          2 Dir(s)   1,518,030,848 bytes free  
  
C:\WINDOWS\system32\Com\sserver>
```

So, we can see a rootkit program, DLL which was used to infect other processes, WinPCAP driver and finally the rootkit log.

# *Creating memory dumps*

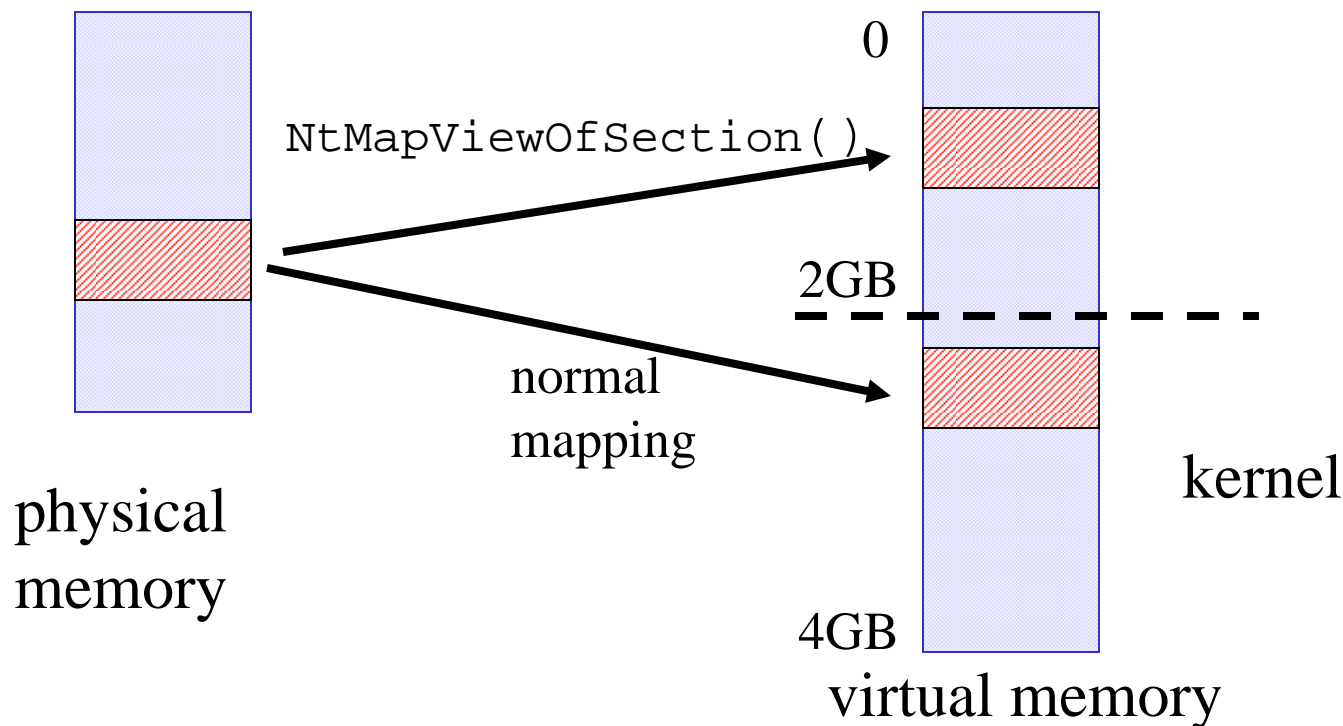
- Livekd + KD works fine as far as online analysis is concerned
- Sometimes we would like to save a crash dump and perform the whole analysis in the lab
- Standard mechanism for creating crash dumps requires system reboot (or at least booting with /debug switch, which in fact is unacceptable on a production system)

# \Device\PhysicalMemory



# Access to physical memory

- `\Device\PhysicalMemory` object of type *Section*
- `NtOpenSection()` ← to get the handle
- `NtMapViewOfSection()` ← to map piece of physical memory to process virtual memory



# *Tools for creating dumps*

- `dd.exe`, together with other forensic tools can be downloaded from:

`http://users.erols.com/gmgarner/forensics/`

- ⊕ `dd if=\\.\PhysicalMemory of=c:\physmem.img`

- ⊕ unfortunately such *dump* is not compatible with *crashdump* which could be used under MS KD

- ⊕ `dd` however, has many other interesting applications.

- `livekd + kd`:

- ⊕ To create full crash dump:

  - `.dump /f c:\kernel.dmp`

- ⊕ reading dump into debugger:

  - `kd -z c:\kernel.dmp`

# *Usermode rootkits short summary*

- We focused on usermode rootkits so far.
- Can hide quite well in the system.
- Usually they work together intruder's process(es) making them invisible. Those processes may include backdoor processes (like in yyt), command prompt for the backdoor (hxdef) or some kind of password sniffer.
- So far we are able to detect them all using LiveKd and MS KD.

# *Other types of rootkits*

- Kernel rootkits which modify System Service Table (famous NTRootkit by Greg Hoggan)
- ✦ detection: SST analysis using KD – see script on the next slide. It simply checks, for every SST entry to which module it belongs.
- Kernel rootkits which hook function code (similar to usermode rootkits)
- ✦ detection: almost the same as with usermode rootkits – use `!chkimg` command.
- The above two types are however not very popular in the wild

# *Useful script for checking SST*

```
$$ Get the service table length
r $t0 = poi(KeServiceDescriptorTable+8);
.for ( r $t1 = 0; @$t1 < @$t0; r $t1 = @$t1 + 1)
{
    r $t2 = (KiServiceTable+@$t1*4);

    $$ get the syscall handler address into $addr
    as /x $addr (poi (@$t2) & 0`FFFFFFFF);

    $$ check to which module $addr belongs
    .block {
        as /c $module lm lm a $addr;
    }

    as /x $sysno @$t1;

    .block {
        .echo Service ${$sysno} : handler at ${$addr} --> ${$module};
    }
    ad ${/v:$module};
    ad ${/v:$addr};
    ad ${/v:$sysno};
}
}
```

# Integrity checking on many levels

⊕ Filesystem/Registry

⊕ Process memory

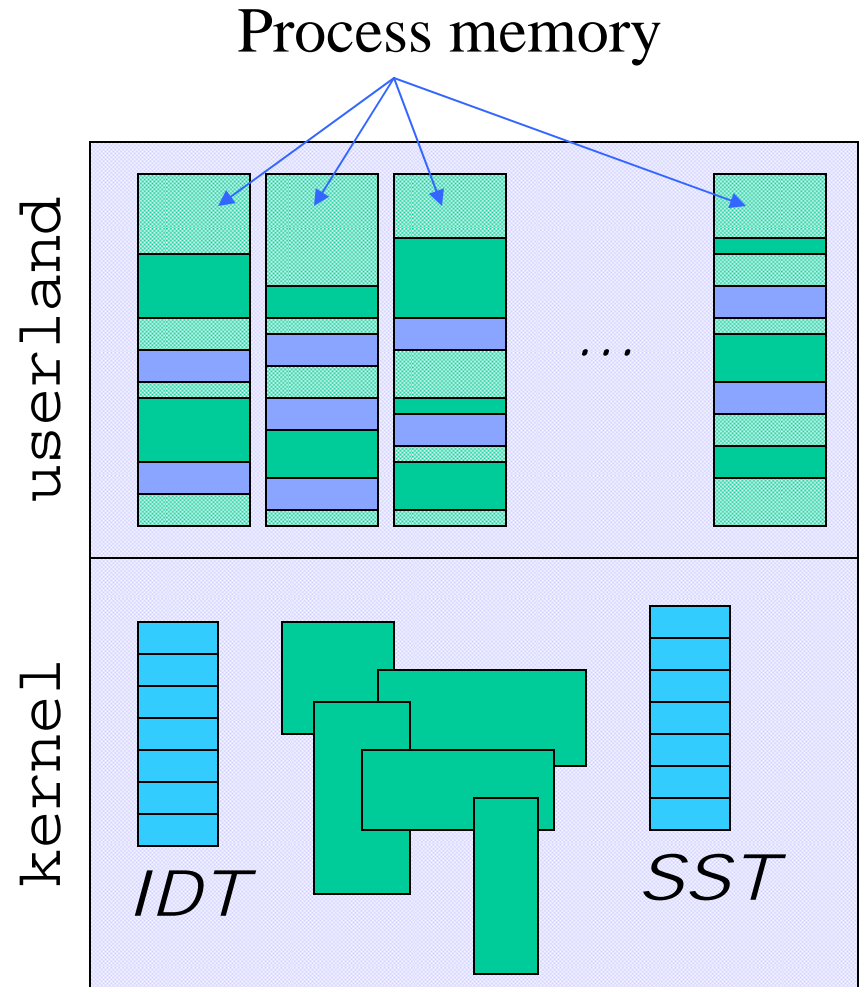
⊕ Code sections 

⊕ IAT tables 

⊕ Kernel SST/IDT 

⊕ Kernel Code 

Is it enough? Of course not;)



# *Is it enough?*

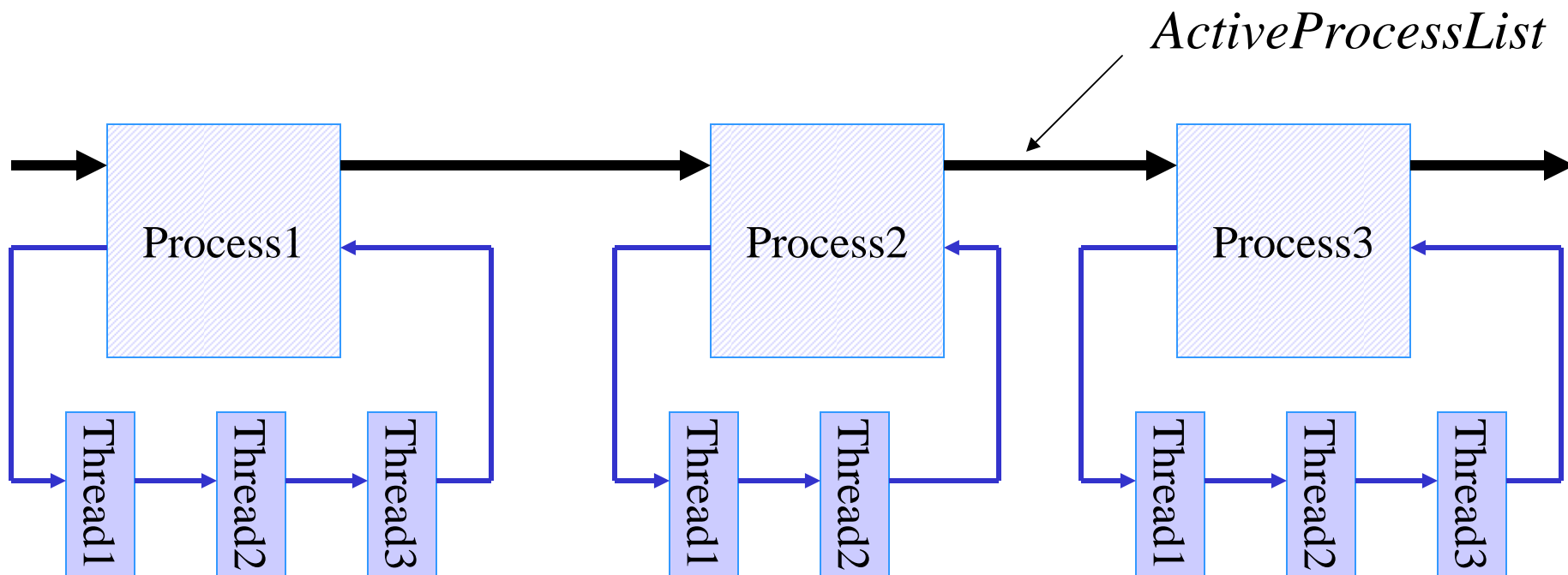
- Is it possible to write a rootkit, which:
- Does not modify any files/registry
- Does not modify process user memory space
- Does not hook *IDT* nor *SDT/SST/ KiServiceTable*
- Does not change kernel code area

# *DKOM (Direct Kernel Object Modification)*

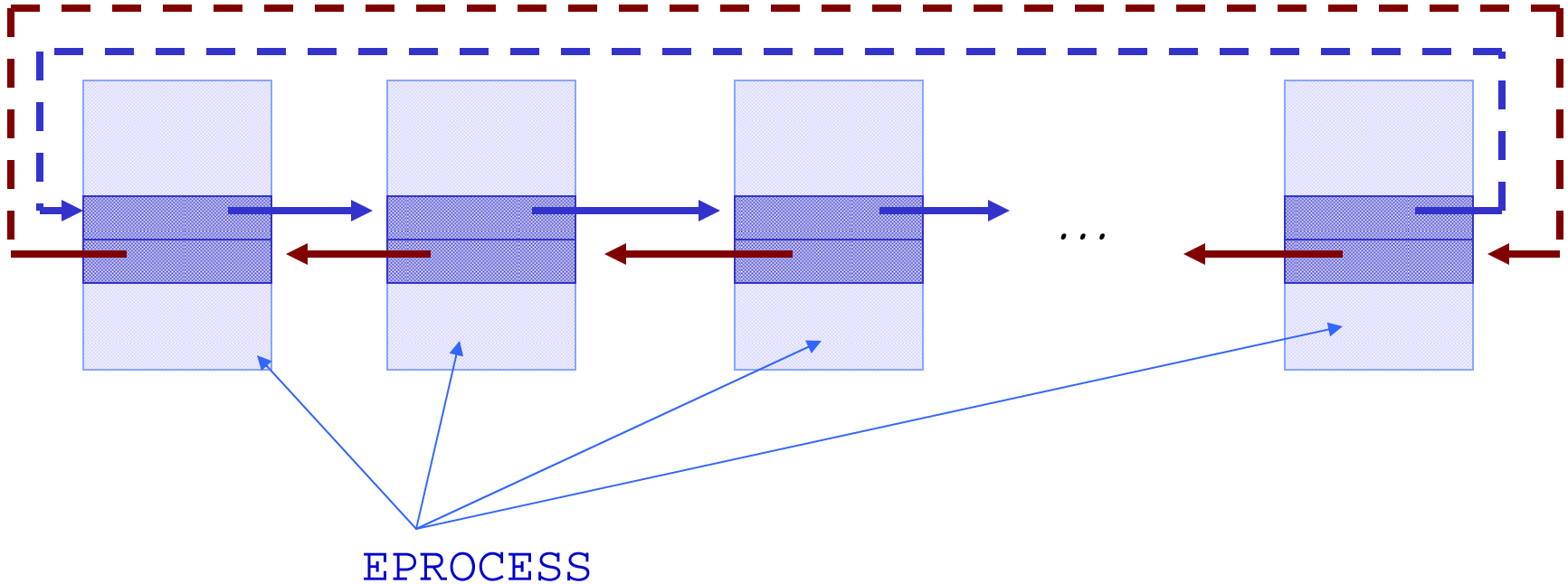
- New approach to hiding in the system
- Pioneered by James Butler
- Implemented in fu rootkit
- Does not require any code changes, only unlinking some objects from the kernel internal lists

# EPROCESS Objects

- Every processes is described by EPROCESS object
- Every thread is described by ETHREAD object
- Structures' internals can be seen under KD:  
eg. `kd> dt _EPROCESS`



# *ActiveProcessLinks*

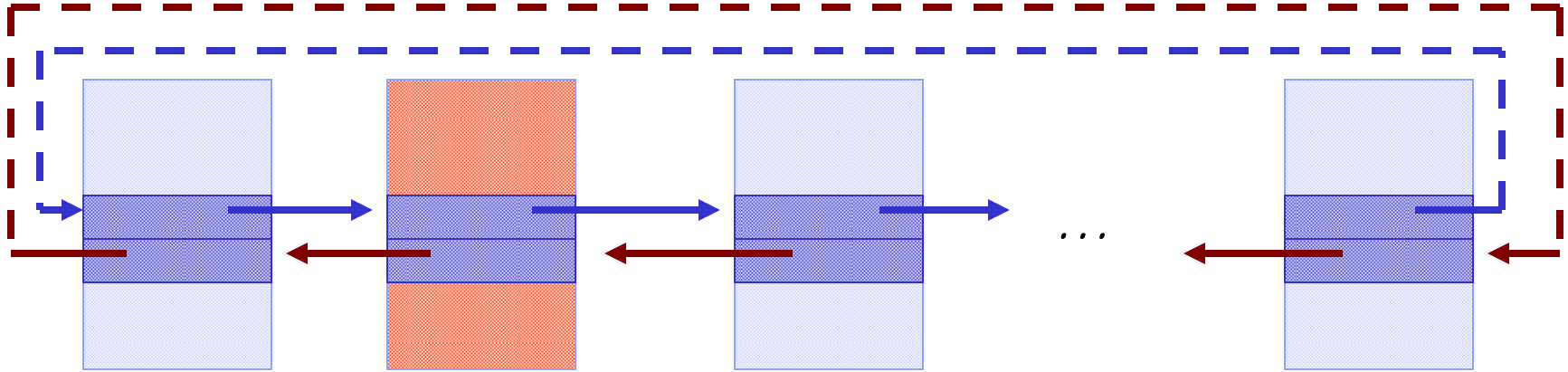


All process objects are kept on this list. It is implemented by `ActiveProcessLinks` fields in the `EPROCESS` structure.

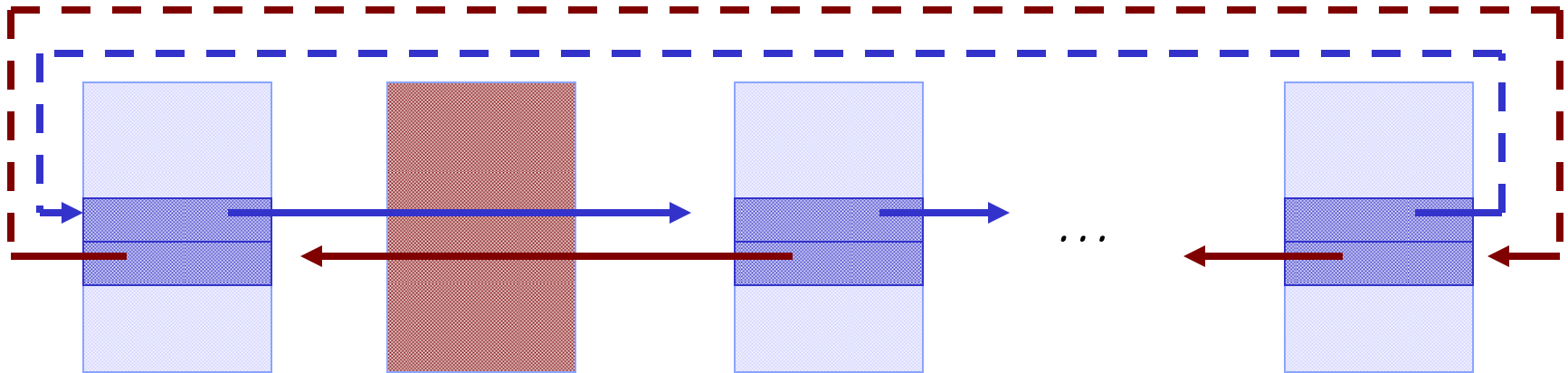
# *EPROCESS Object*

```
kd> dt nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x06c ProcessLock : _EX_PUSH_LOCK
+0x070 CreateTime : _LARGE_INTEGER
+0x078 ExitTime : _LARGE_INTEGER
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : Ptr32 Void
+0x088 ActiveProcessLinks : _LIST_ENTRY
+0x090 QuotaUsage : [3] Uint4B
+0x09c QuotaPeak : [3] Uint4B
+0x0a8 CommitCharge : Uint4B
+0x0ac PeakVirtualSize : Uint4B
+0x0b0 VirtualSize : Uint4B
+0x0b4 SessionProcessLinks : _LIST_ENTRY
+0x0bc DebugPort : Ptr32 Void
(...)
```

# *Fu rootkit*



Intruder's process



Now it is hidden!

# *Live DEMO*

- `klogger.exe` – simple keystroke logger processes
- FU rootkit –DKOM based rootkit, which we use to hide `klogger.exe` process
- OK, so how to detect the hidden processes now?

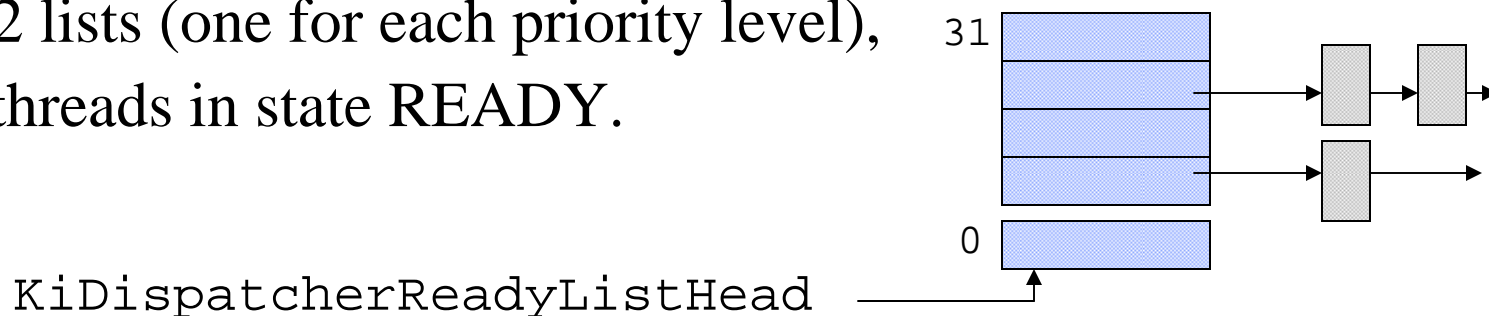
# *Detection of hidden processes*

- Dispatcher thread list analysis
- Heuristic search for blocks of data, which “looks” like KPROCESS structures

# Dispatcher Database (Win2000):

## ■ KiDispatcherReadyListHead

actually 32 lists (one for each priority level),  
grouping threads in state READY.



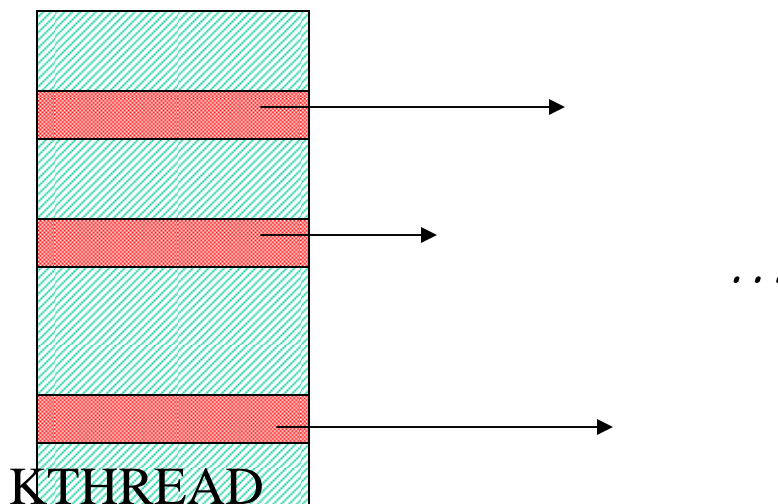
## ■ KiWaitInListHead

## ■ KiWaitOutListHead

Threads which are not ready for execution (for e.g. waiting on some object)

# Dispatcher Database (Win2003)

- Lack of `KiWaitInListHead` and `KiWaitOutListHead` on XP and 2003 systems ☹
- The threads are still kept on some lists however, but nobody(?) knows where those lists are starting...
- Solution: we need to search for the threads starting from known threads and follow pointers in `KTHREAD`...



# *KPROCESS Object*

```
kd> dt nt!_KPROCESS
+0x000 Header          : _DISPATCHER_HEADER
(...)
+0x028 Int21Descriptor : _KIDTENTRY
+0x030 IopmOffset     : Uint2B
+0x032 Iopl           : UChar
+0x033 Unused         : UChar
+0x034 ActiveProcessors : Uint4B
+0x038 KernelTime     : Uint4B
+0x03c UserTime       : Uint4B
+0x040 ReadyListHead  : _LIST_ENTRY
+0x048 SwapListEntry  : _SINGLE_LIST_ENTRY
+0x04c VdmTrapcHandler : Ptr32 Void
+0x050 ThreadListHead : _LIST_ENTRY
+0x058 ProcessLock    : Uint4B
(...)
```

# *KTHREAD Object*

```
kd> dt nt!_KTHREAD
    +0x000 Header                : _DISPATCHER_HEADER
    +0x010 MutantListHead        : _LIST_ENTRY
    (...)
    +0x060 WaitListEntry       : _LIST_ENTRY
    +0x060 SwapListEntry        : _SINGLE_LIST_ENTRY
    (...)
    +0x0a0 WaitBlock            : [4] _KWAIT_BLOCK
    +0x100 QueueListEntry     : _LIST_ENTRY
    +0x108 ApcStateIndex        : UChar
    (...)
    +0x1a8 LegoData             : Ptr32 Void
    +0x1ac ThreadListEntry    : _LIST_ENTRY
    +0x1b4 LargeStack           : UChar
    (...)
```

# *Algorithm for finding all threads in the system (works on 2000, XP and 2003)*

```
for each process p on PsActiveProcessList do {  
    for each thread t which belongs to process p do {  
        findAllThreadsFromList (t, WaitList)  
        findAllThreadsFromList (t, QueueListEntry)  
        findAllThreadsFromList (t, ThreadListEntry)  
    }  
}  
  
findAllThreadsFromList (KTHREAD t, int listOffset) {  
    for each element e on list (t+listOffset) do  
        if (isKTHREAD(e)) then addThreadToFoundList (e);  
}
```

# *Thread → Process*

- Having a list of all threads in the system it is trivial to compile a list of all processes in that system.
- See slide #30

# *Live DEMO: detecting hidden processes on Win 2003*

```
C:\tools>psfinder.exe /t show all
```

PID	Name
-----	
4	System
372	smss.exe
432	csrss.exe
456	winlogon.exe
500	services.exe
(...)	
512	lsass.exe
1180	explorer.exe
476	IEXPLORE.EXE
604	cmd.exe
<b>h 304</b>	<b>klogger.exe</b>
412	cmd.exe
620	psfinder.exe
(...)	

# *Alternative for process hiding rootkit*

- Dedicated rootkit (for e.g. integrated keylogger and covert channel)
  - ⊕ Implemented in kernel
  - ⊕ (Partly) implemented as an additional thread inserted in to some system processes (note the difference with current usermode rootkits)
- Hard to write. Well...;)
- Hard to detect...

# Summary

- MS Kernel Debugger can be very useful tool for system compromise detection
- It should be safe to use it even on a production server (together with LiveKD)
- Presented techniques (KD + DKOM process detection) allows to detect all known rootkits (known to the author)
- As we saw DKOM rootkit require special tools (like klister) and cannot be detected by KD (anybody would like to implement klister as KD script?)