

Detecting Windows Server Compromises

Joanna Rutkowska

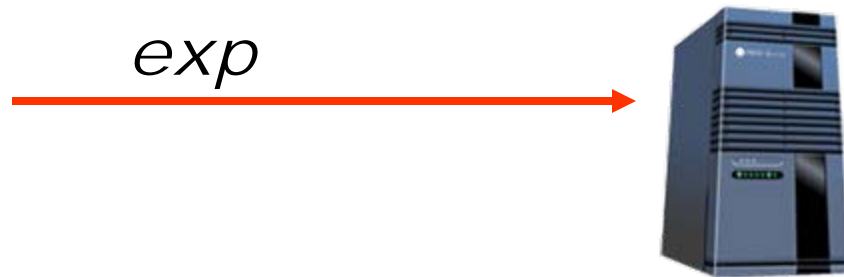
joanna@mailsnare.net

Contents

- ✦ Basic rootkit techniques overview.
- ✦ Simple detection overview (integrity checks).
- ✦ More advanced techniques for hiding objects.
- ✦ Accessing dispatcher database.
- ✦ Execution Path Analysis.
- ✦ Summary of detection.

Exploitation scenario no. 1

- ✦ Exploit the bug.
- ✦ Do some harm (or steal some information).



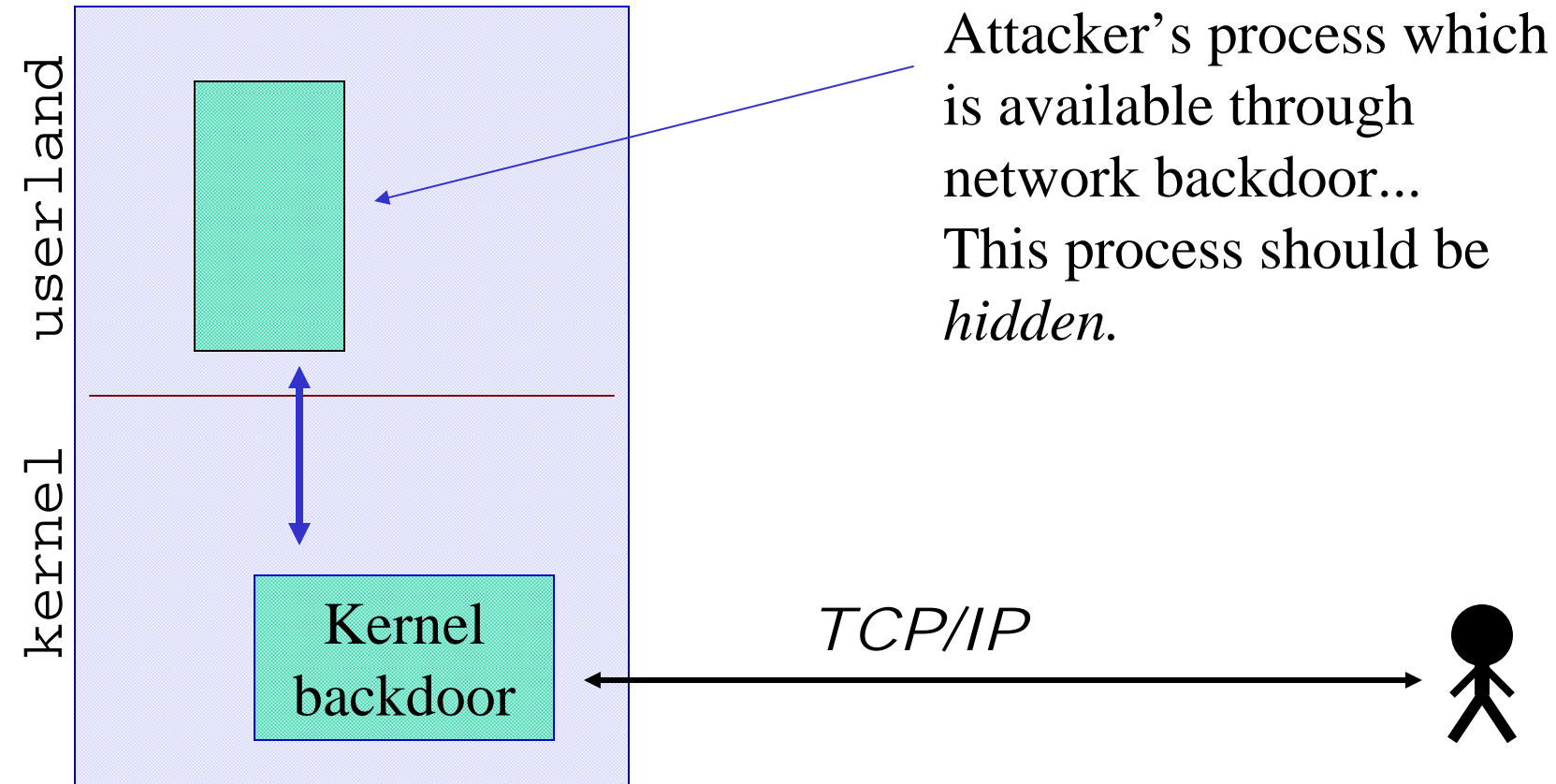
Exploitation scenario no. 2

- ✦ Exploit a bug (get into).
- ✦ Leave a hidden backdoor.
- ✦ Make use of this backdoor.
- ✦ But use it silently.
- ✦ Perform some activity in the hacked system.
- ✦ But try to do it stealthy.

Network Backdoors

- ⊕ Intercepts normal packet processing:
 - ⊕ Kernel NDIS driver (e.g. *NT rootkit*)
 - ⊕ WinSocket functions hooks (e.g. *Hacker Defender*)
- ⊕ Exploits some form of steganography.
- ⊕ Backdoor needs to provide some service.
- ⊕ Almost always it means: **be able to run arbitrary processes.**

Smart backdoor



Process Hiding

- ✦ Essential feature if attacker is interested in staying invisible
- ✦ Almost every *rootkit* provides such functionality

Other objects hiding

⊕ Loaded drivers hiding

This is needed if rootkit has been written as a conventional driver. This could be unnecessary if we use for e.g. `\Device\PhysicalMemory` to insert rootkit into kernel.

⊕ File/Directory Hiding

⊕ File Contents cheating

⊕ Registry keys/contents hiding

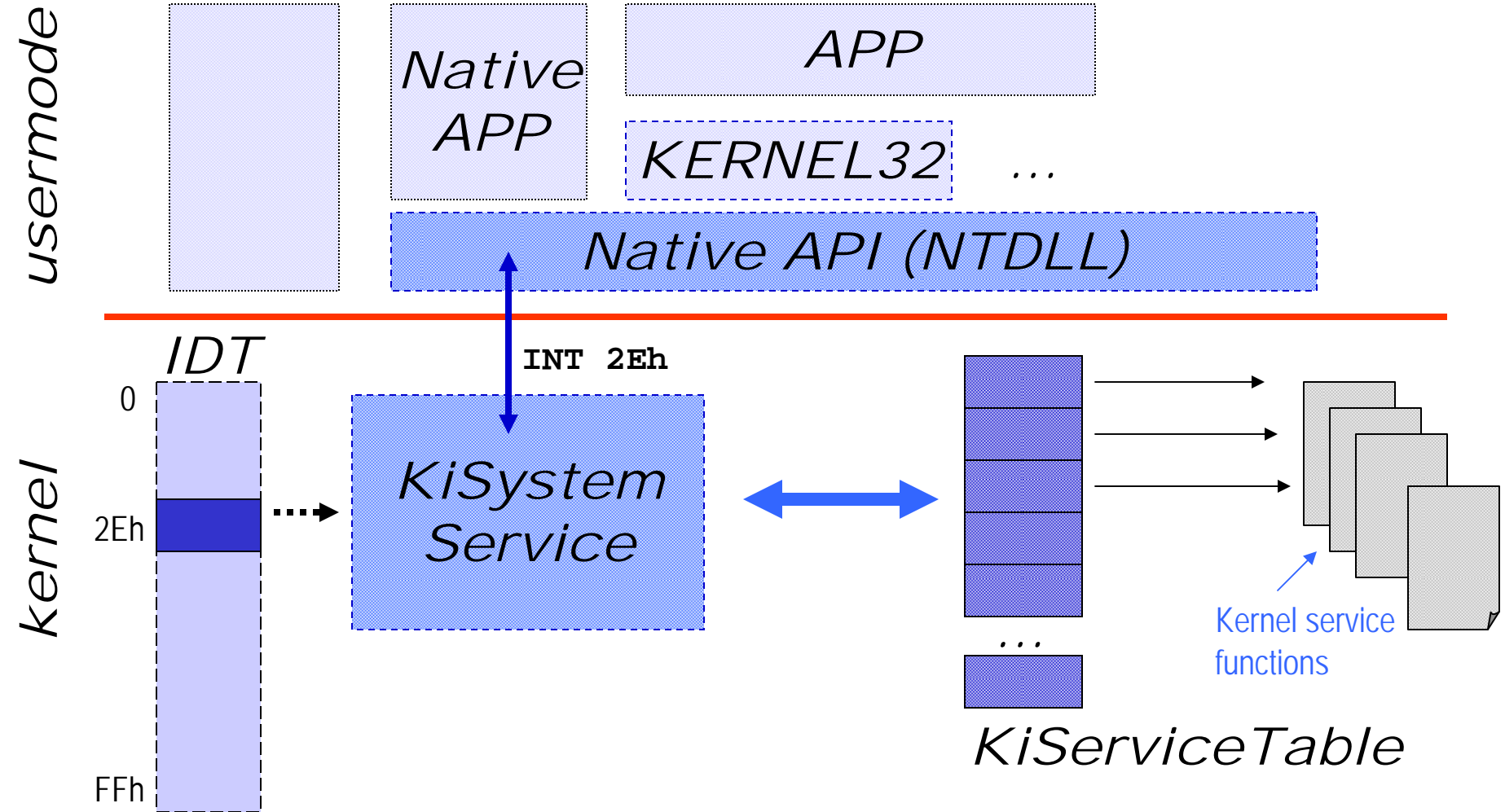
Notes on file creation

- ⊕ Today, after break-in to Windows box, it seems that we need to upload some files to the target system (for e.g. rootkit.sys file, some tools, etc...)
- ⊕ This is not stealthy (although later, we can use rootkit to hide arbitrary files) because of the forensic analysis efforts...
- ⊕ Real stealth → no disk modifications!
- ⊕ Usually servers are rarely restarted, so attacker is not really interested in surviving the reboot.
- ⊕ On the other hand, it would be good to have a detection tool to detect file/directory hiding without the need of removing the disk(s) from the server...

Virtual Playground Idea

- ✦ To support really stealth break-ins (i.e. no disk modifications)
- ✦ Piece of code which creates a new object, which can be used as a virtual drive.
- ✦ This virtual drive should be available for attacker's processes only (stealth reason)
- ✦ This should be integrated with some shellcode, so that virtual driver will be started by shellcode execution (so, no disk modification needed)
- ✦ This is not yet implemented :(

Rootkit's playground



Traditional ways for hiding various objects

- ⊕ Replacing files (e.g. DLLs)
- ⊕ Hooking DLL's functions (API/IAT hooking)
- ⊕ Modifying DLL's functions (Raw Code Change)
- ⊕ Hooking entries in SDT/SST/KiServiceTable (very popular)
- ⊕ Hooking IDT 2Eh entry
- ⊕ Modifying Kernel Code (Raw Code Change)

Integrity checking

⊕ Filesystem/Registry

⊕ Process memory

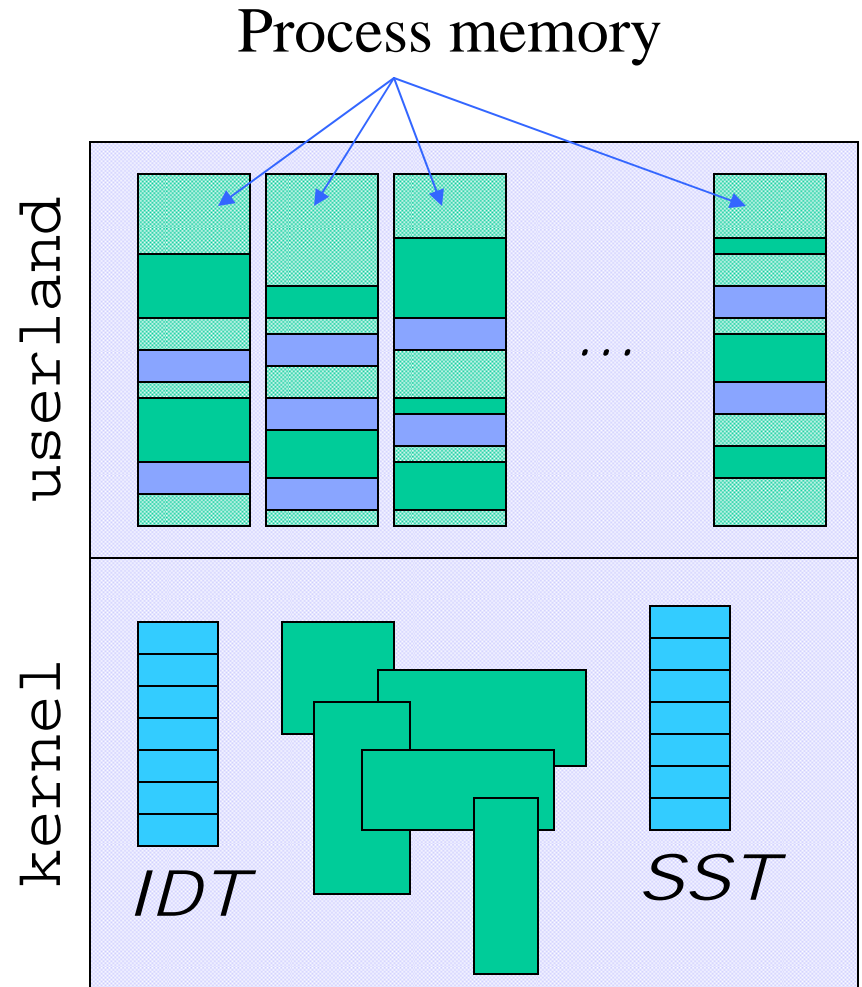
⊕ Code sections 

⊕ IAT tables 

⊕ Kernel SST/IDT 

⊕ Kernel Code 

We need a baseline of the clear system to compare with!



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

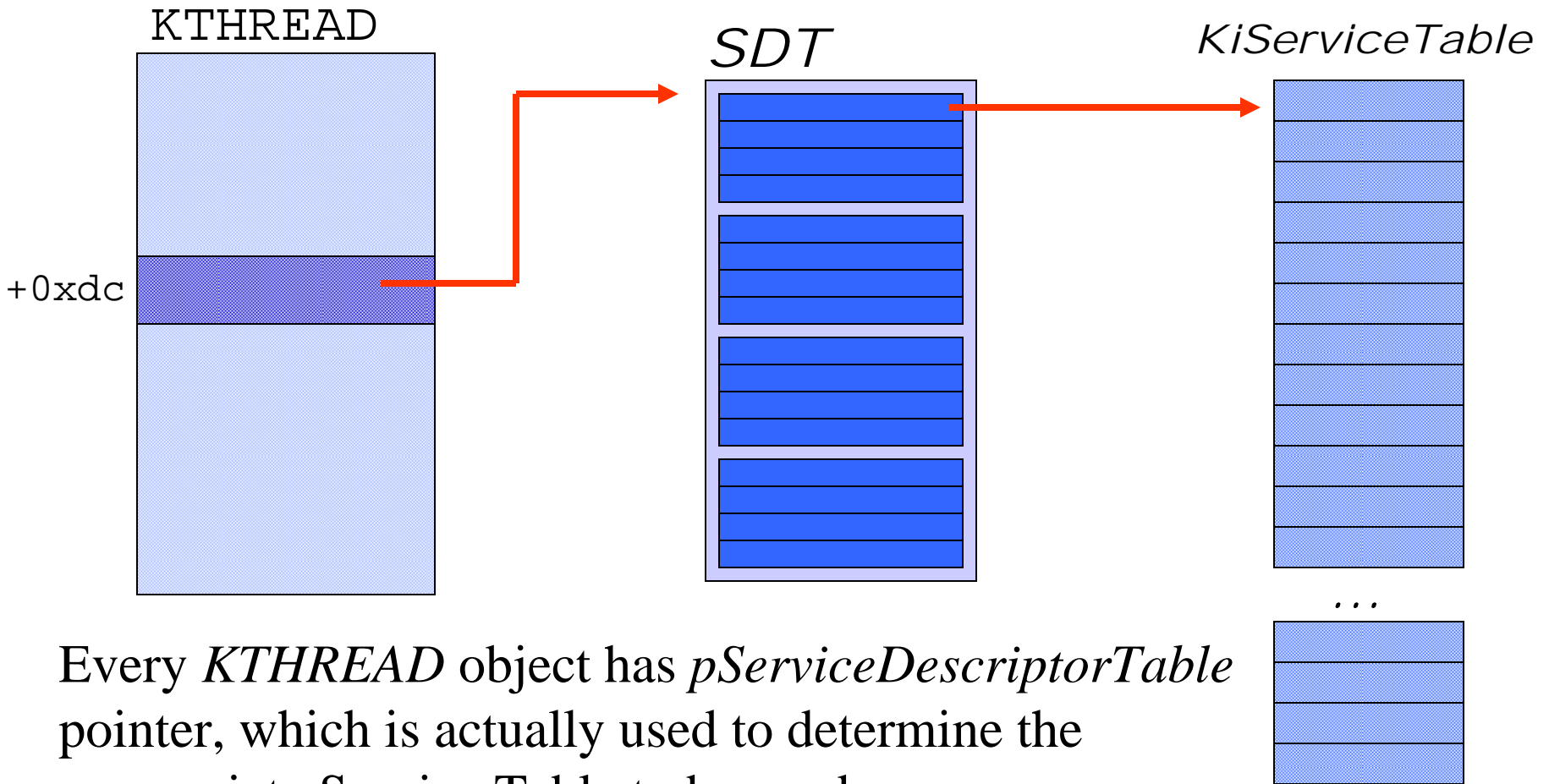
Two examples

- ✦ Strange pointer hooking
- ✦ Kernel data manipulation

Example 1

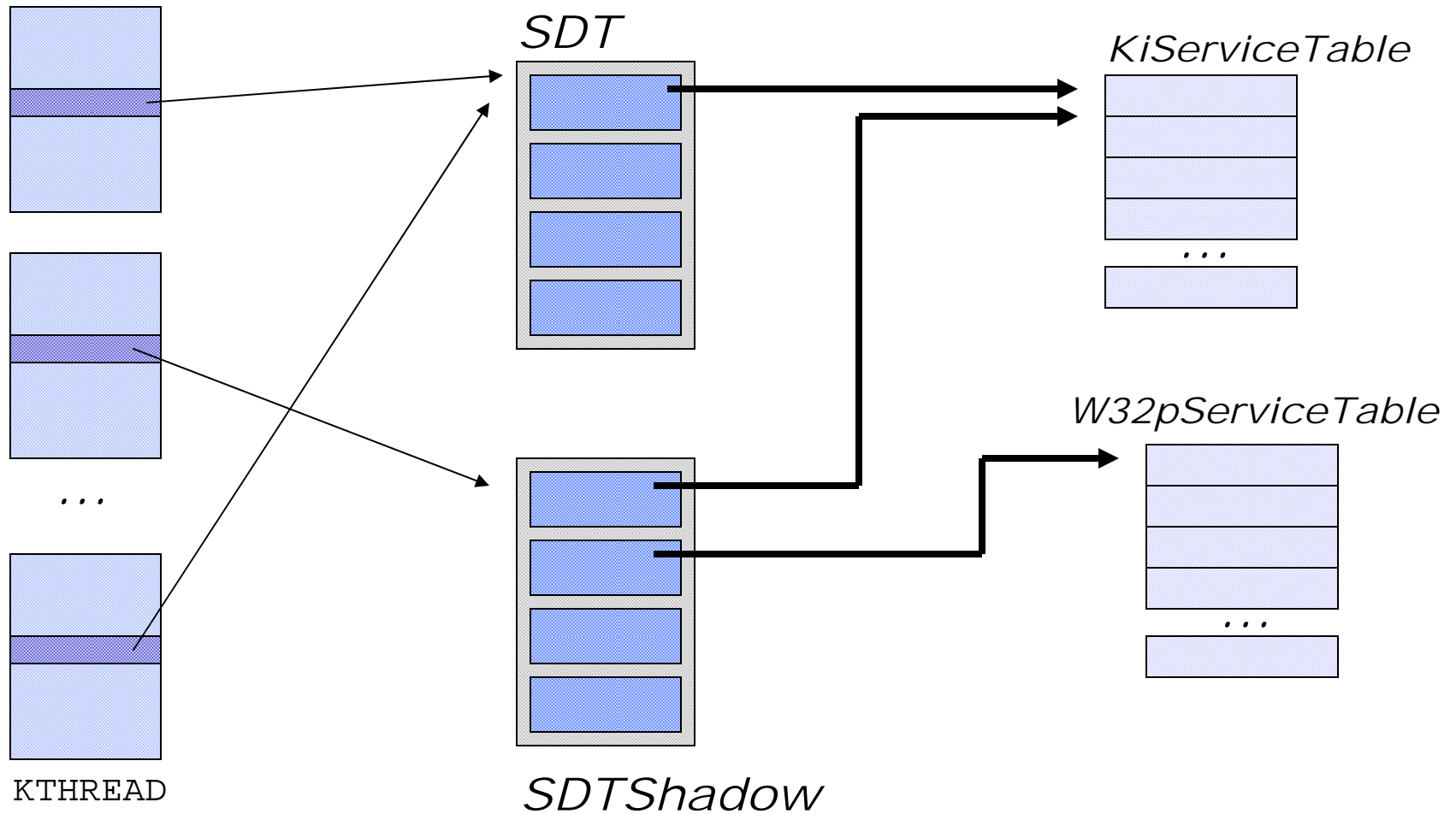
Not only *IDT* or *KiServiceTable* can be hooked...

SDT

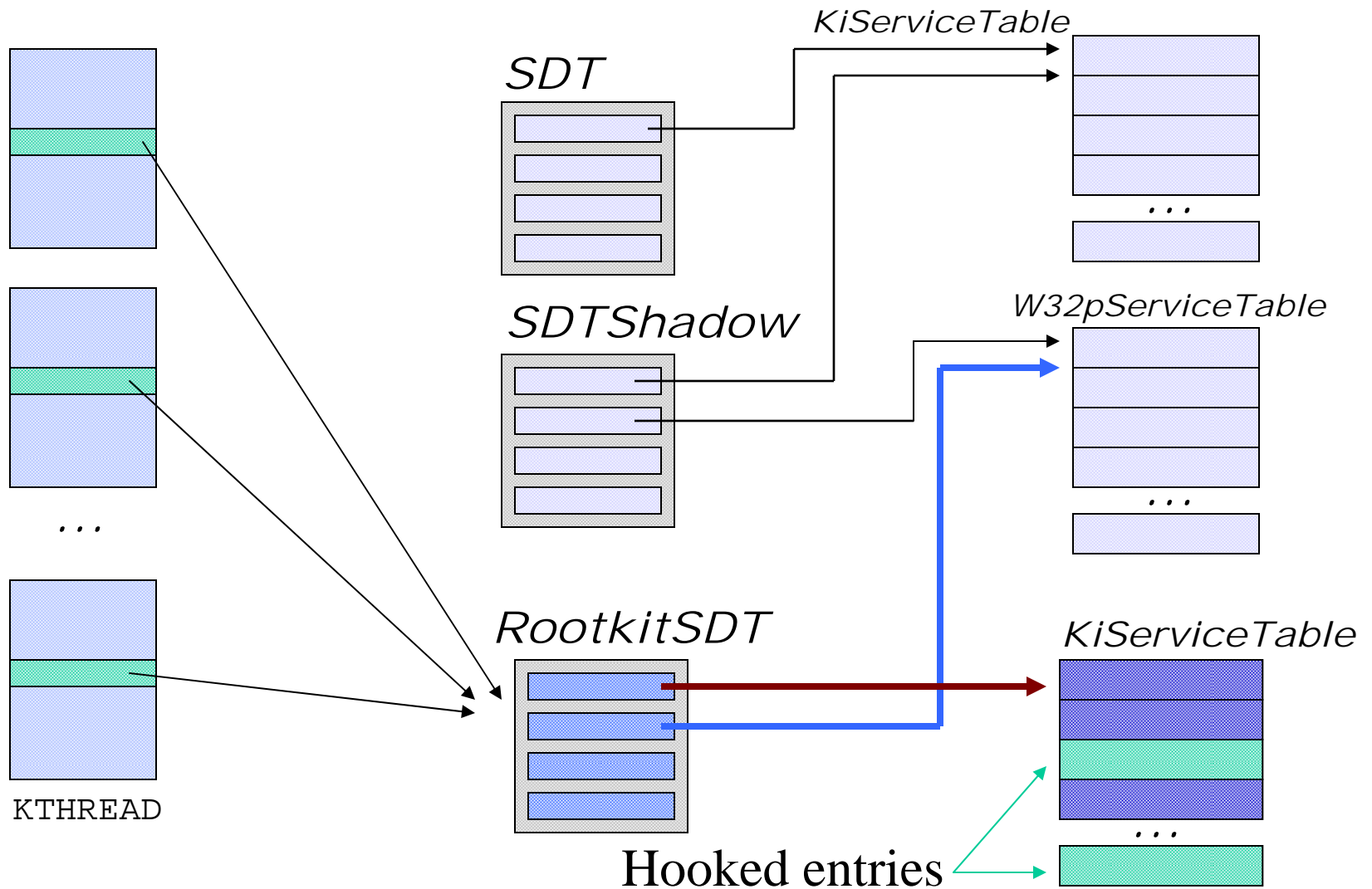


Every *KTHREAD* object has *pServiceDescriptorTable* pointer, which is actually used to determine the appropriate Service Table to be used.

SDT & SDTShadow



Strange Pointer Hooking Example



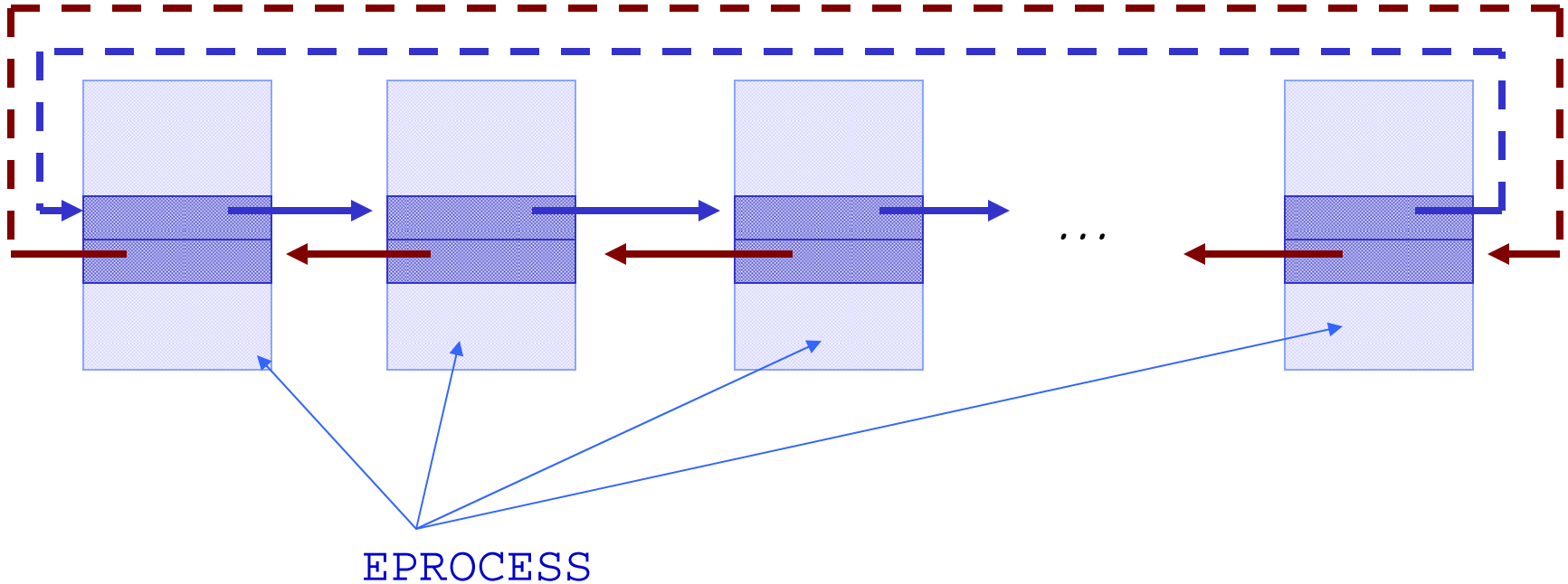
Strange Pointers Hooks

- ✦ *KTHREAD.pServiceDescriptorTable* is just an example
- ✦ Probably many other pointers can be hooked to achieve some rootkits goals (see *He4Hook* rootkit for example).
- ✦ It is extremely difficult to spot all this crucial pointers...
- ✦ On the other hand, we cannot check all the kernel memory (i.e. All data sections) since data are something which naturally changes hundreds times per second...
- ✦ But among all of these data are pointer like *KTHREAD.pServiceDescriptorTable* for example...
- ✦ And we don't know where...

Example 2

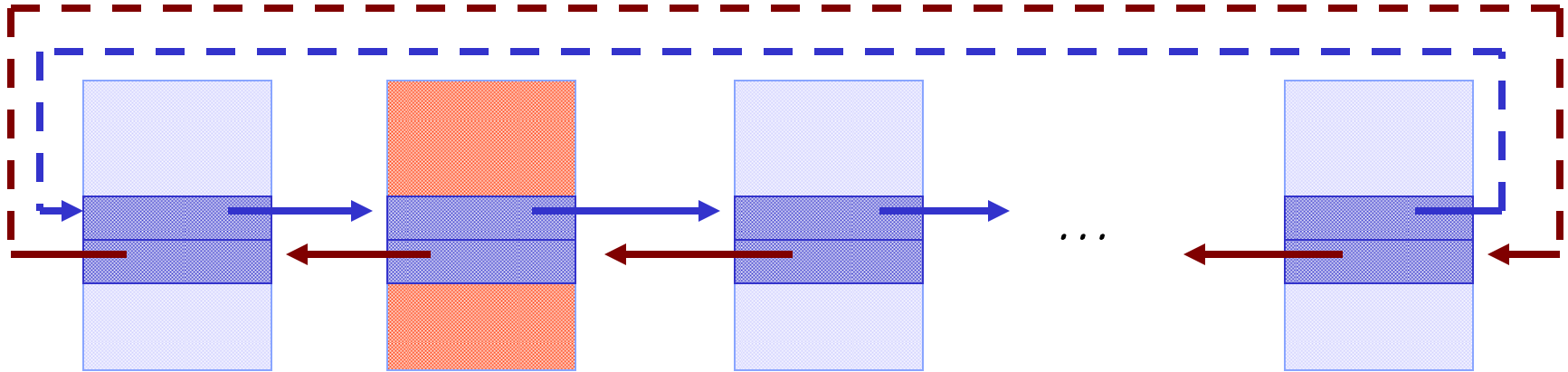
Sometimes we do not have to change execution path...

ActiveProcessLinks

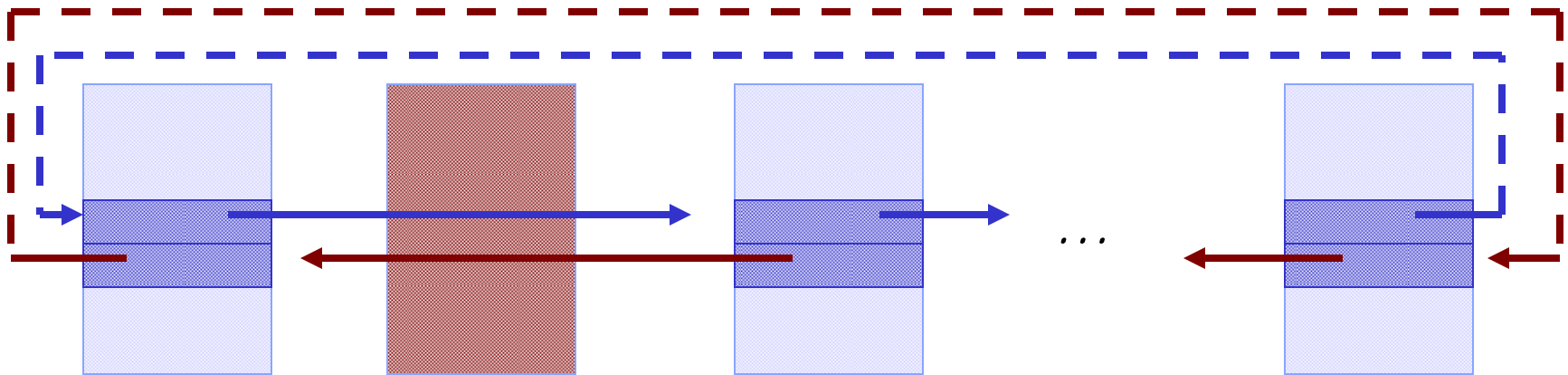


All active processes in the system are kept on the single list.
This list is implemented by pair of pointers in each
EPROCESS block:
EPROCESS.ActiveProcessLinks

Fu rootkit



Attacker's process



Now it is hidden

Unlinked process

- ✦ Surprisingly still gets CPU!
- ✦ Because windows scheduling is thread based.
- ✦ Every thread in the system is held on some list, either for running or sleeping threads

Rootkit Technology Summary

*Execution path
Change*

*Only data structures
Change (e.g. fu)*

-
- ✦ *Simple hooking
(IAT, SDT/SST, IDT)*
 - ✦ *Raw code change*
 - ✦ *Strange pointers change*

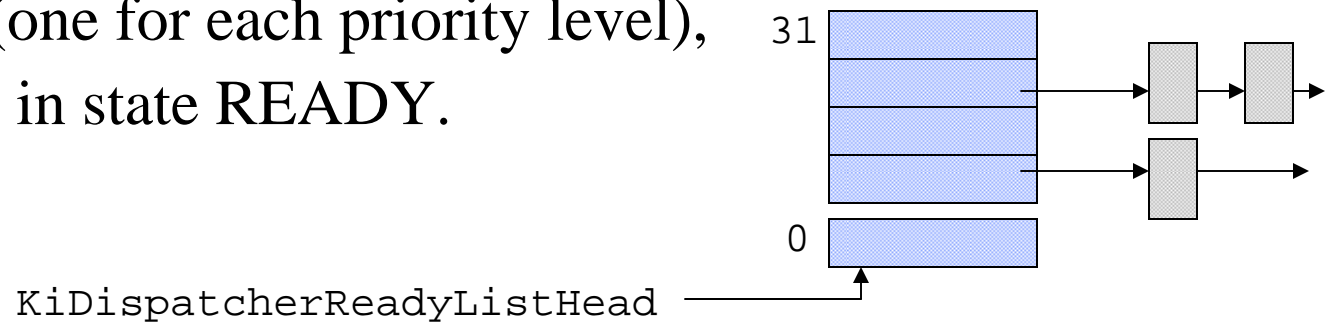
More advanced detection

- ✦ Hidden (unlinked) processes
- ✦ Strange hooks...

Dispatcher Database:

⊕ *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)

KLISTER – a tool for listing (all) processes

- ✦ Reads dispatcher database.
- ✦ By scanning 3 lists described earlier, collects the list of currently running threads in the system.
- ✦ After the list of threads is completed, for each thread checks which process it belongs to.
- ✦ This way we create the *real* list of all active processes in the system.
- ✦ NOTE: Not all dispatcher lists are exported by the kernel – we need debug symbols to find them.

KLISTER DEMO

Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration.

Cheating KLISTER

- ✦ Is it possible to hide process so that KLISTER won't find it?
- ✦ Yes, modify the dispatcher code!
- ✦ This is not good solution, since code area modification can be detected (see earlier discussion).
- ✦ Cheat when accessing memory (Chameleon idea).

Chameleon Concept

- ⊕ Extremely stealth Windows Rootkit.
- ⊕ Only basic functionality:
 - ⊕ Process hiding
 - ⊕ Smart (steganography based) network backdoor
- ⊕ Totally undetectable by memory scanning (IDT, SST, Code, Dispatcher Database)
- ⊕ Undetectable, even if the sources will be made public (somewhat similar to modern crypto algorithms).

Chameleon: processes hiding

- ✦ Unlink process object (like *fu*)
- ✦ Threads of the unlinked process still in the dispatcher database. They have to be there, so that hidden process get some CPU.
- ✦ To hide those threads, we should cheat at every memory access to their objects, if the instructions which reads the memory is not part of the scheduler code...
- ✦ This means Page Fault hook ;) (performance?)
- ✦ This means IDT or code change...
- ✦ So, we also need to use DR_x register to hide hooked entries...

Chameleon

- ⊕ Not yet implemented :(
- ⊕ It is not yet clear if it would be possible to implemented such thing :(
- ⊕ Drop me an email if you are interested in development... ;)

Rootkit Technology Summary

*Execution path
Change*

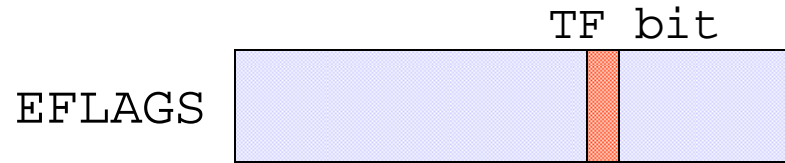
*Only data structures
Change (e.g. fu)*

-
- ✦ *Simple hooking
(IAT, SDT/SST, IDT)*
 - ✦ *Raw code change*
 - ✦ *Strange pointers change*

Different approach to rootkit detection

- ⊕ How to detect execution path changes?
- ⊕ Trace the execution path for some typical system activities (like system services)
- ⊕ Compare the trace with the trace saved after the installation of clear system
- ⊕ So, we need a baseline, but it is mostly acceptable requirement (exactly as in case of most integrity checkers).

Step Mode on IA-32



- ✦ TF bit in EFLAGS register.
- ✦ When in step mode: exception #DB (1) is raised after the execution of every machine instruction.
- ✦ #DB exception handler is stored at IDT[1].
- ✦ TF bit is cleared when `int 2Eh` instruction is used to enter the kernel mode.

EPA - implementation

```
INC counter;  
IRET;
```

```
if (pid==trace_pid) setTFbit();
```

```
KiSystemService:  
...  
CALL [EBX];  
...  
IRET;
```

IDT

0h

1h (#DB)

2Eh

FFh

Implementation in Windows

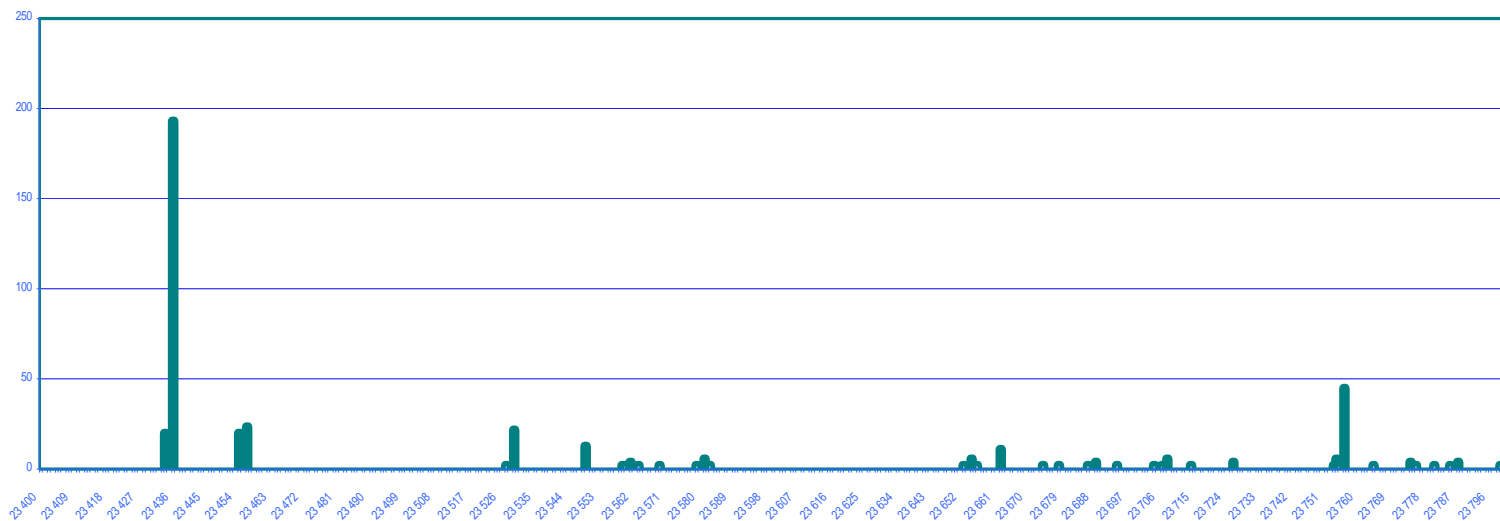
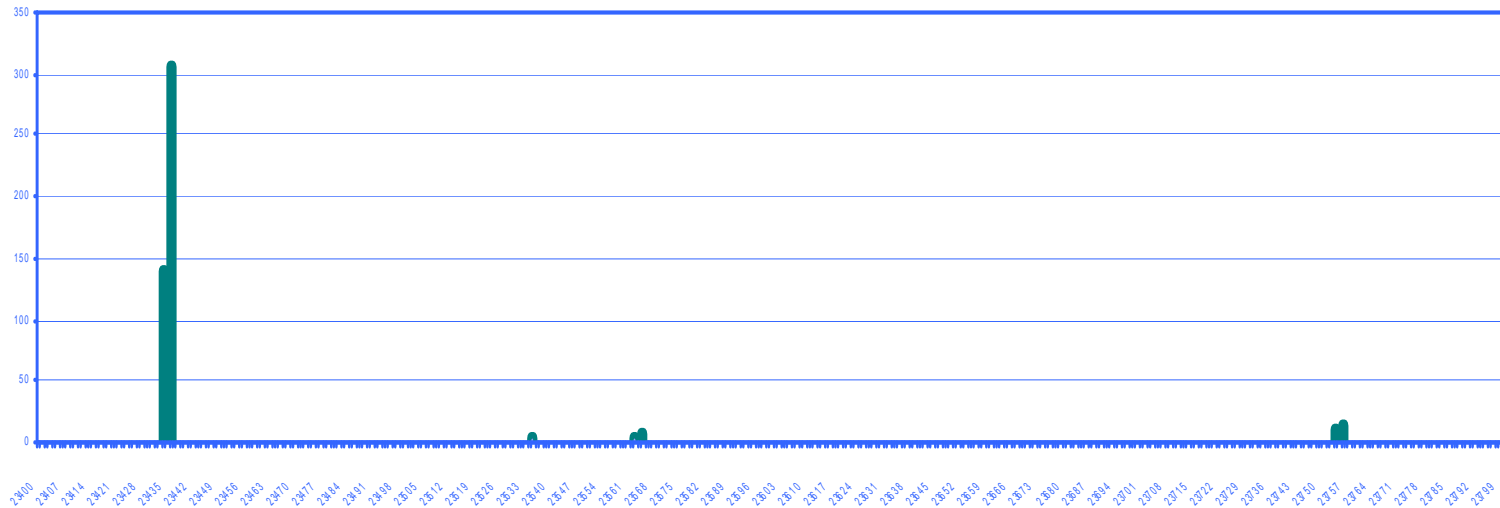
- ⊕ All normal processes are executed in non step mode (so, there is no impact on performance).
- ⊕ Only tester process set TF bit when executing tests.
- ⊕ TF bit affects only process which set this bit (i.e. Tester process) and not any process other nor, for e.g. interrupt handlers.

Test example

```
for (int i = 0; i < N; i++) {  
    pfStart();  
    hFind = FindFirstFile(  
        "C:\\\\WINNT\\system32\\drivers",  
        &FindFileData);  
    pfStop();  
    FindClose(hFind);  
}
```

- ⊕ We will get N samples of this test.
- ⊕ We can calculate average value, but it is not very deterministic :(
- ⊕ Or, we can create a histogram...

FindFirstFile() histogram



Nondeterministic results

- ⊕ Windows kernel is a complex program, same task can be completed by different execution paths
- ⊕ This is not good, we need deterministic behavior to see the difference caused by the rootkit extra code
- ⊕ Solution: create a histogram

False Positives

- ✦ Sometimes peek's position changes a little (typically less than 20 instructions)
- ✦ Is it rootkit or just false positive?
- ✦ In such case we need to see exactly what instructions caused the difference!
- ✦ This requires deep technical knowledge from the user :-/
- ✦ Possibly 'diff -c' can be replaced by some more sophisticated program;)

Comparison of two traces:

```
*** RegEnumKey-clear.trace
--- RegEnumKey-current.trace
*****
*** 273,278 ****
--- 273,281 ----
    0x80416f60
    0x80416f63
    0x80416f65
+ 0x80416f67
+ 0x80416f6a ←
+ 0x80416f6d
    0x80416f74
    0x80416f76
    0x80416f77
```

Extra instructions (actually only addresses), which caused false positive. In current implementation we need to attach debugger to see the actual instructions.

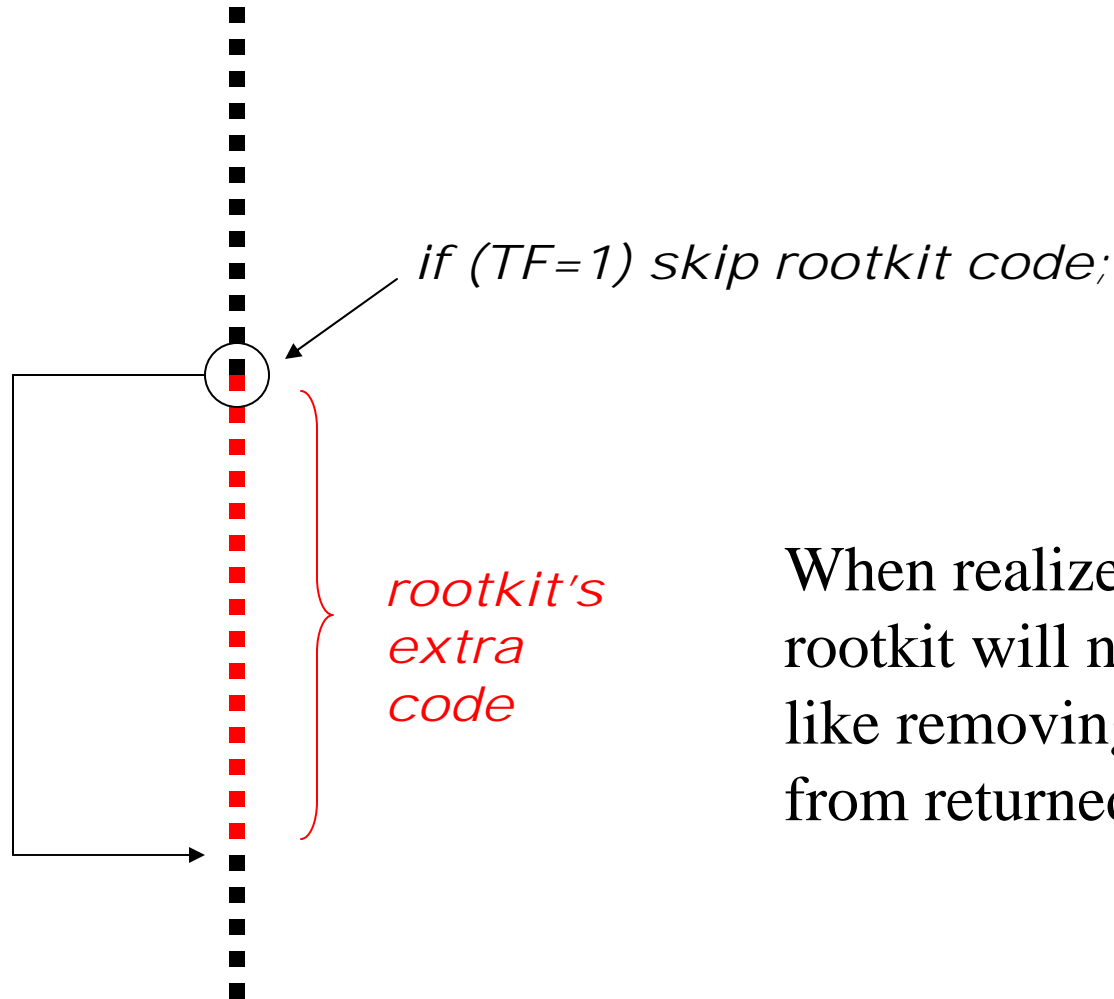
Patchfinder Demo

Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration. Live demonstration.

Cheating EPA

- ⊕ Is it possible to write a rootkit, which will not be detected by EPA (Patchfinder)?
- ⊕ One example, are only data modification rootkits (like *fu*), but they should be detected by KLISTER...
- ⊕ However there are some tricks to cheat EPA by rootkits which do modify execution path.
- ⊕ Lets have a look at the attacks and defenses...

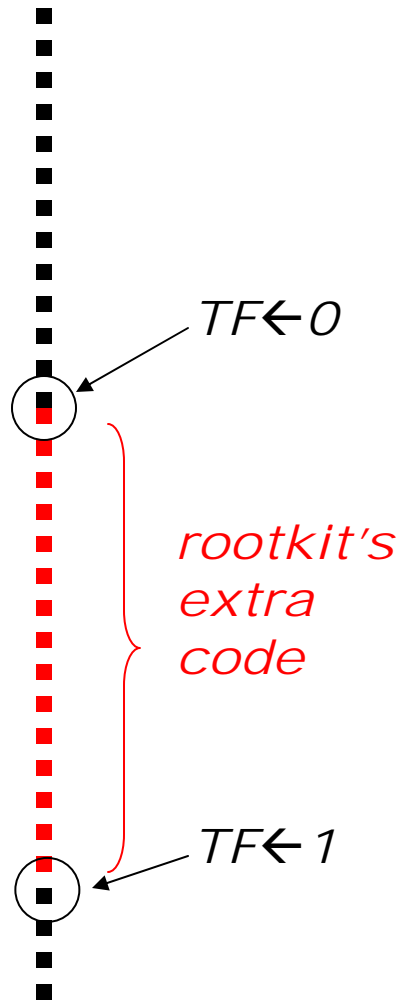
Attack 1: TF check



When realized that it is traced, rootkit will not take any actions like removing files/processes from returned lists, etc...

Attack 2: disable step mode

Before performing any actions, rootkit simply disables step mode.



Attacks 1 & 2 defense

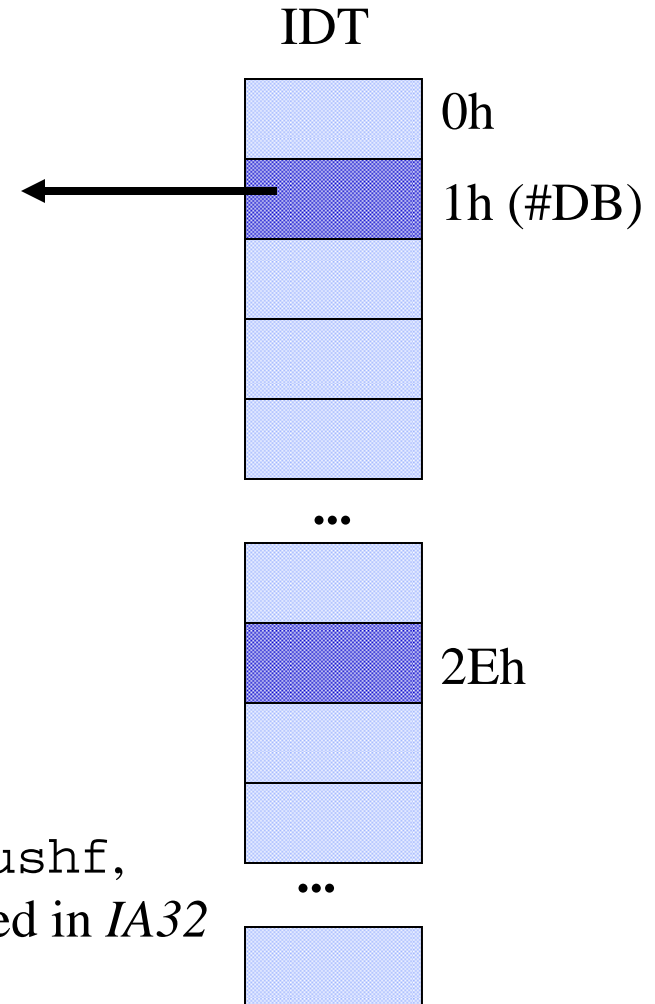
- ⊕ Presented attacks still adds some (minor) number of instructions to execution path.
- ⊕ We can find out this extra instructions by comparing traces with `diff` (see earlier)
- ⊕ Also we can try to protect TF bit in EFLAGS register...

TF bit protection

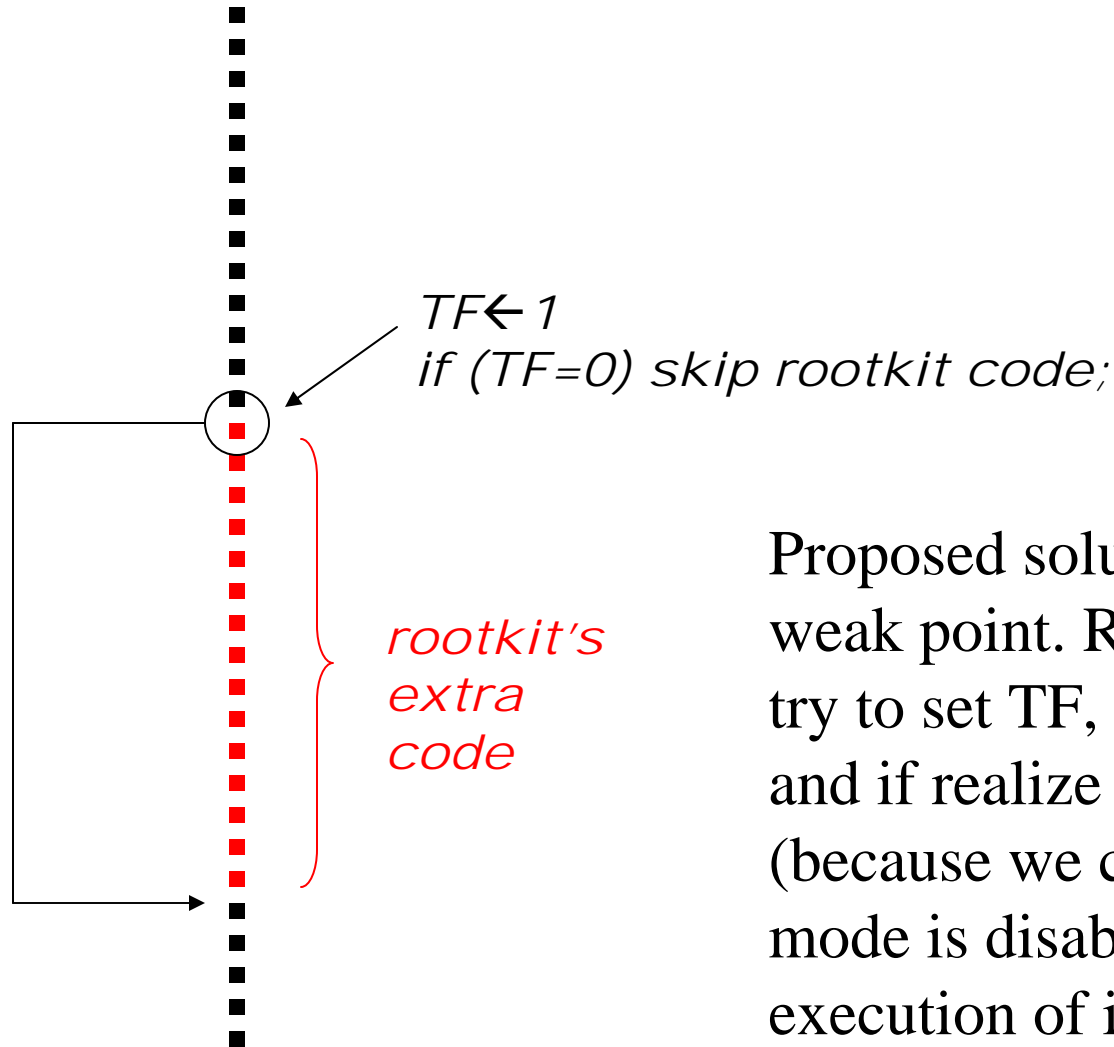
We have to extend our #DB handler:

```
if (nextInstr == 'POPF') {  
    setTFbit in [ESP]  
}  
  
if (prevInstr == 'PUSHF') {  
    clearTFbit in [ESP]  
}  
  
INC counter;  
IRET;
```

Note: there are a few instructions similar to `popf/pushf`, which can access `EFLAGS` register. They are specified in *IA32 Software Developer's Manual*.



Attack 3: cheat TF protection

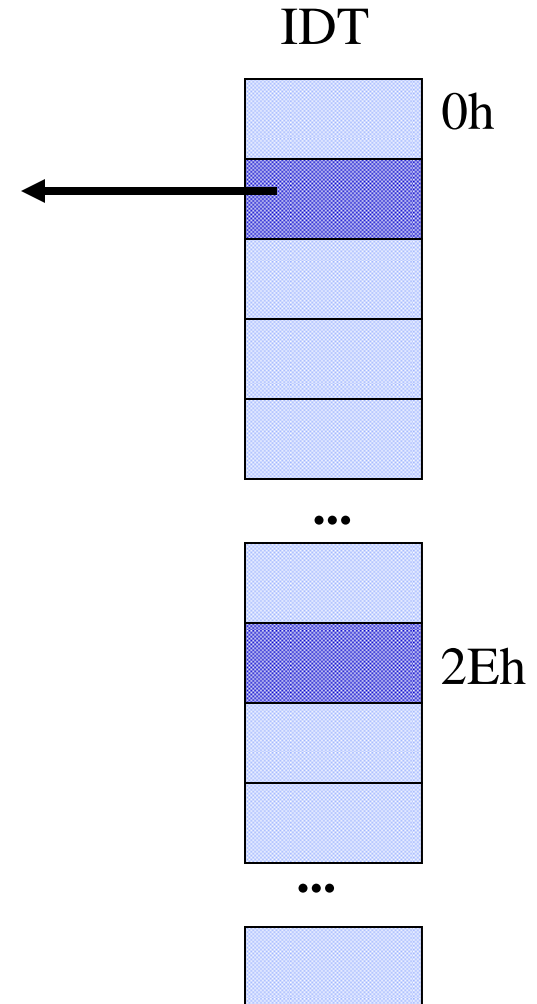


Proposed solution still has a weak point. Rootkit can first try to set TF, then check TF, and if realize that TF is zero (because we cheat that step mode is disabled) skip execution of its own code...

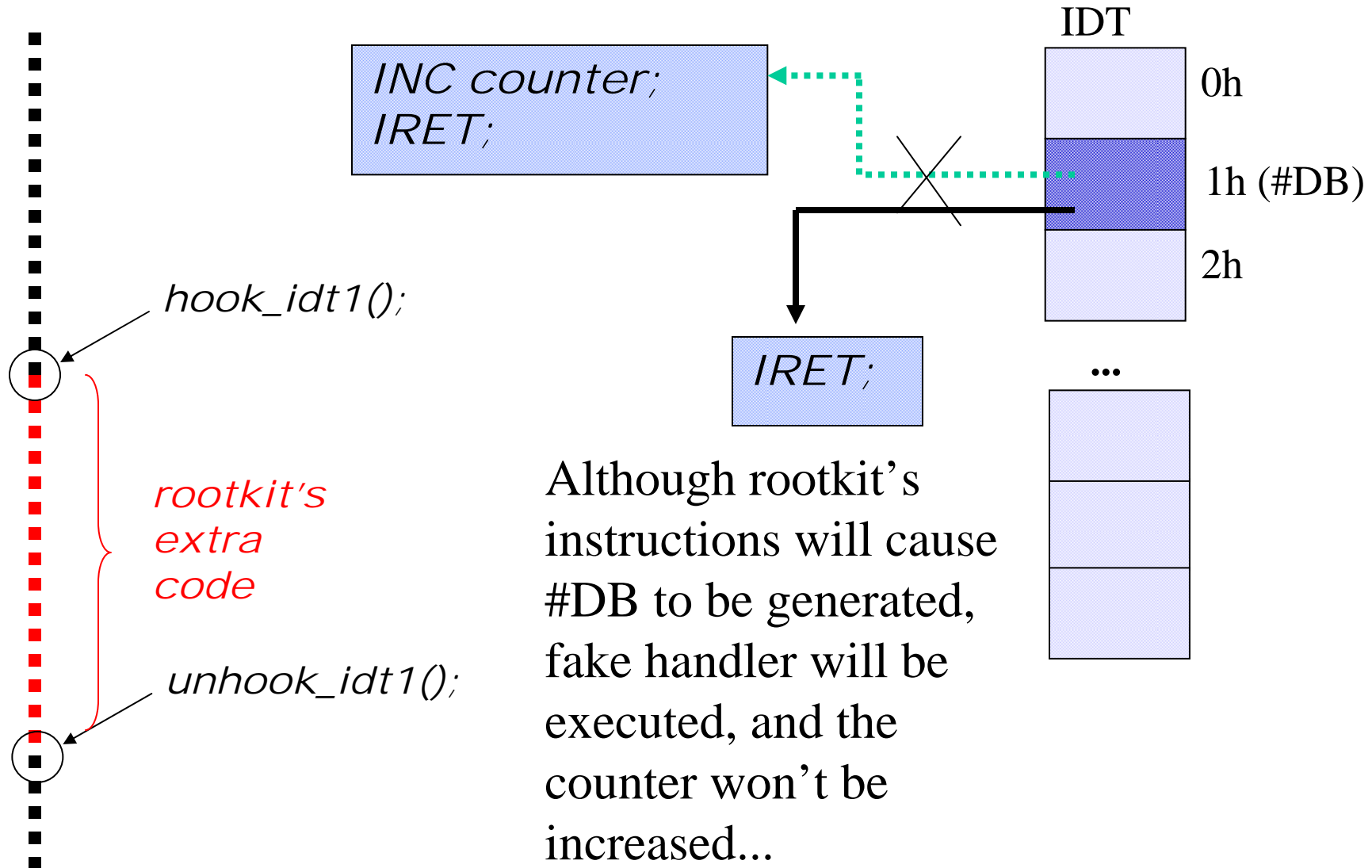
Better TF protection

```
Init: tfbit ← 0;
```

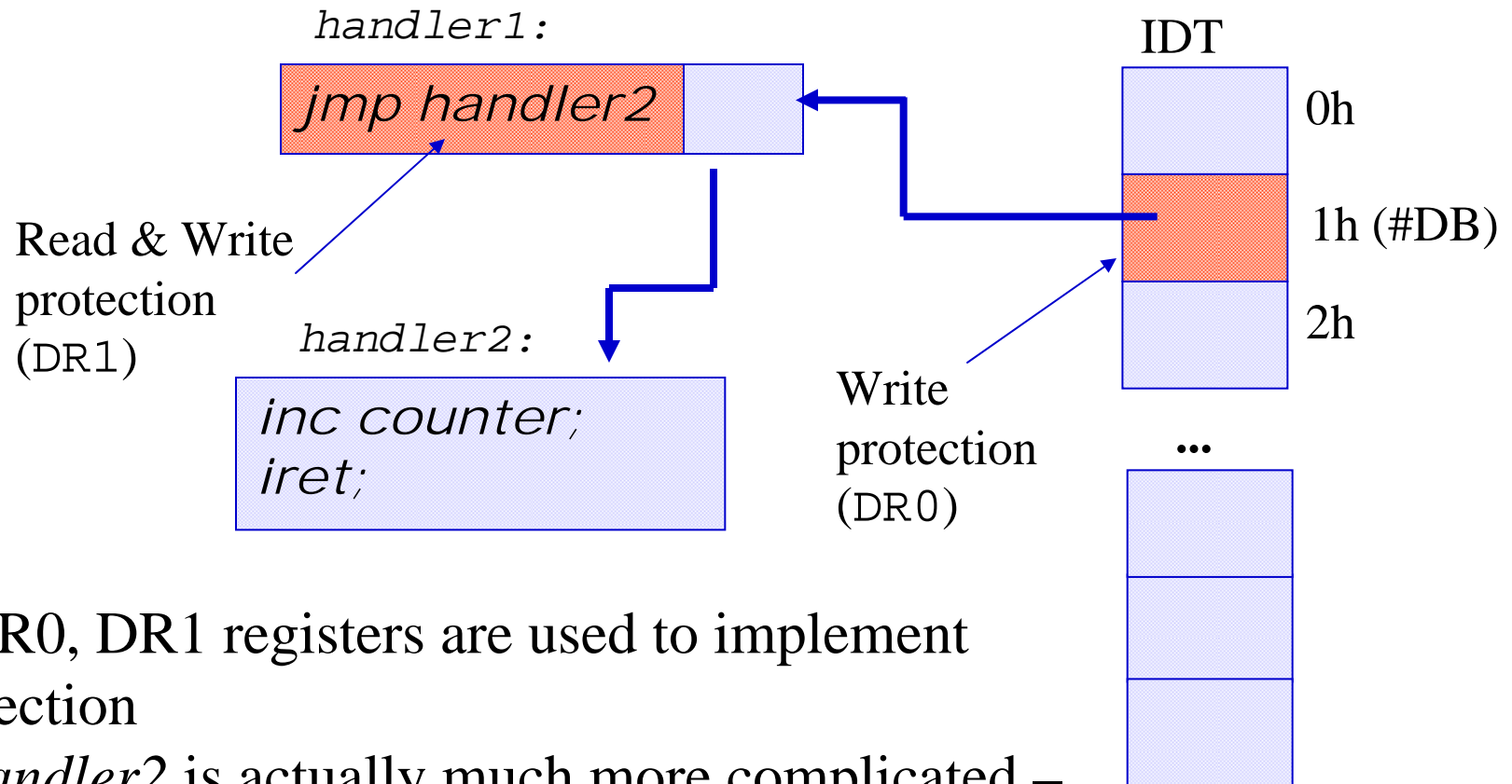
```
if (nextInstr == 'POPF') {  
    tfbit = getTFbit from [ESP];  
    setTFbit in [ESP];  
}  
  
if (prevInstr == 'PUSHF') {  
    setTFbit in [ESP] to tfbit;  
}  
INC counter;  
IRET;
```



Attack4 : IDT[1] hook



IDT[1] protection



✦ DR0, DR1 registers are used to implement protection

✦ *handler2* is actually much more complicated – for e.g. it must recognize situation when DR_x protection is violated.

EPA: future development

- ✦ Many new testes should be added to detect more „patches”
- ✦ Possibility to test not only system services, but also some internal kernel functions
- ✦ Complete path recording (together with instructions)
- ✦ Comparing the trace instead of only the number of instructions...
- ✦ TF bit protection should be implemented

EPA: summary

- ⊕ Current implementation still should be considered as proof-of-concept.
- ⊕ Most (or all?) of the present rootkits can be detected by techniques described earlier: kernel integrity checkers, and KLISTER tool.
- ⊕ So, it is probably a tool for detection of the future rootkits...
- ⊕ But we don't know when these „future” rootkits appear on our servers ;)

Summary of detection methods

- ⊕ File System/Registry Integrity Checker
- ⊕ Kernel memory Integrity Checker
 - ⊕ IDT
 - ⊕ SDT/SST/KiServiceTable
 - ⊕ Code Area (Not yet implemented)
- ⊕ Kernel Dispatcher Database (*KLISTER*)
- ⊕ EPA (*Patchfinder*)

Summary (1)

- ✦ **Process Memory Integrity Checker**: some work has been done by James Butler, but the tool is not public, and also needs some improvements (like IAT tables handling).
- ✦ **Kernel Code Integrity Checker**, tool which will force rootkits developers to use more invention, has not yet been implemented(?)!
- ✦ ***KLISTER*** – tool which detects all processes hidden by all current publicly available rootkits!

Summary (2)

- ✦ *Chameloen* rootkit, which could potentially cheat **KLISTER** has not yet been implemented :(
- ✦ **EPA** has been implemented as a *Patchfinder* tool. With proper tests, it could be potentially used to detect all presented rootkit technologies. However much work still has to be done here...

Summary (3)

- ⊕ All of this, just to make sure, that your crucial server has not been compromised...
- ⊕ If you are interested in further development of presented detection technologies, or you would like to deploy them on your servers, do not hesitate to contact me ;)