

“SHADOW WALKER”

Raising The Bar For Rootkit Detection

By

Sherri Sparks

ssparks@longwood.cs.ucf.edu

Jamie Butler

james.butler@hbgary.com



Black Hat Japan 2005

What Is A Rootkit?

- Defining characteristic is *stealth*.
 - Viruses reproduce, but rootkits hide!
- Greg Hoglund, author of NT Rootkit defines a rootkit as “a set of programs which patch and trojan existing execution paths within the system”.



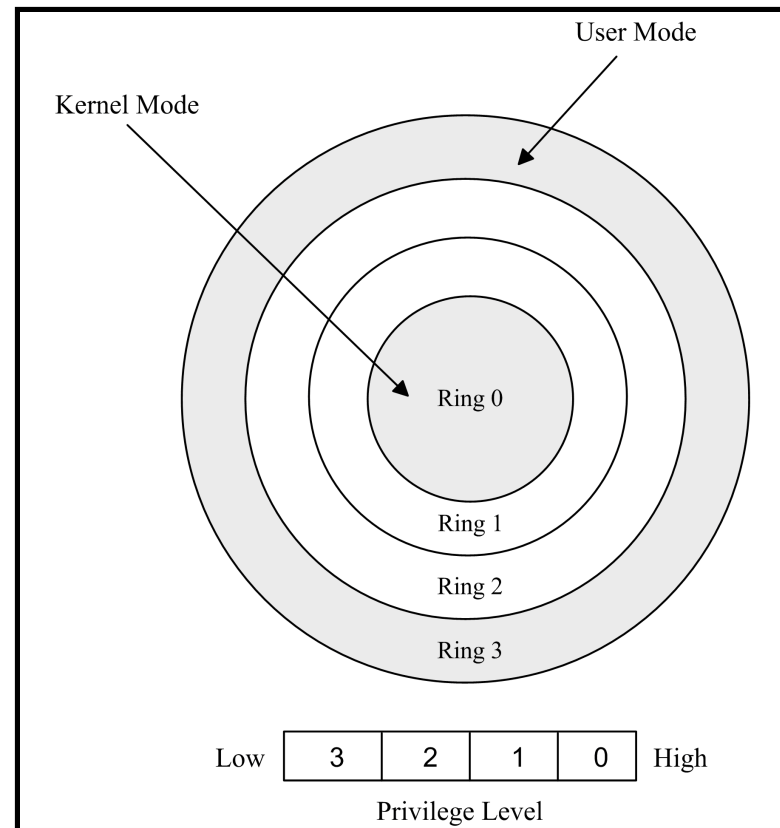
What is a rootkit used for?

- It is usually used by a hacker to conceal his / her presence on a compromised system and make it possible to return undetected at some later date.
- Indirect overlap with parental control software and spyware.



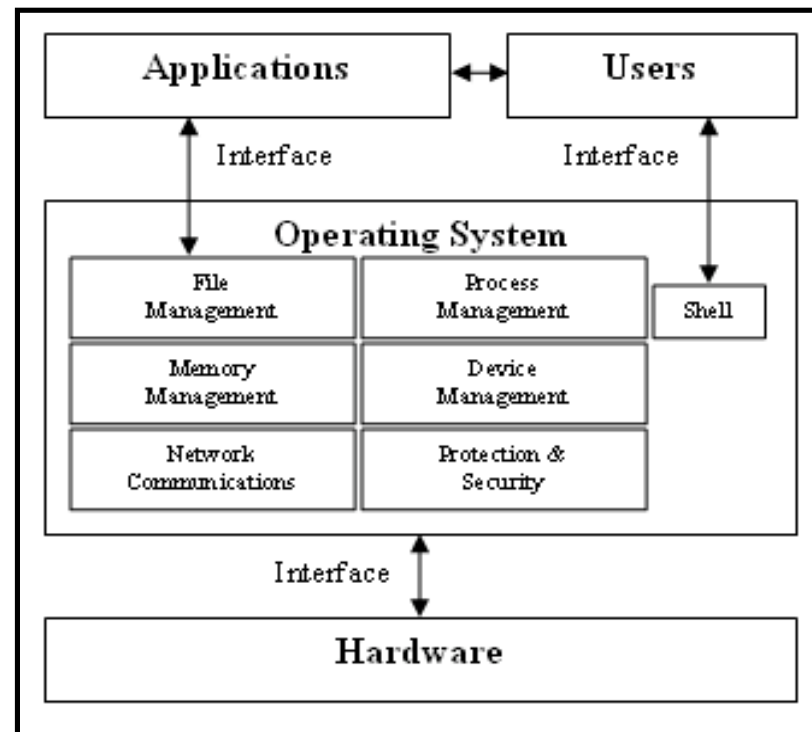
Rootkits & x86 Hardware Architecture: Pentium Protection Rings

- Ring 0 – full access to all memory and the entire instruction set.
 - Kernel Rootkits
- Ring 3 – restricted memory access and instruction set availability.
 - User Rootkits



Rootkits & The Operating System

- The user / application view of the system is defined by what the OS provides to it via the API interface.
- A rootkit hides by intercepting and altering communications at the interfaces between various OS components.
- Rootkits are a form of “man in the middle attack”.



OS Components Attacked By Rootkits

- I/O Manager
 - – Logging keystrokes or network activity
- Device & File System Drivers
 - – Hiding files
- Object Manager
 - – Hiding object (process / thread) handles.
- Security Reference Monitor
 - – Disable security policies governing runtime access checks on objects.
- Process & Thread Manager
 - – Hiding processes & threads
- Configuration Manager
 - – Hiding registry entries



First Generation Rootkits

- Replaced / modified system files on the victim's hard disk
- Example: UNIX login program



Second Generation Rootkits

- Modify static OS components / structures loaded in memory.
 - Table based hooking approaches (IAT, EAT, SSDT, IDT)
 - Inline function hooking
 - Kernel and user mode rootkits

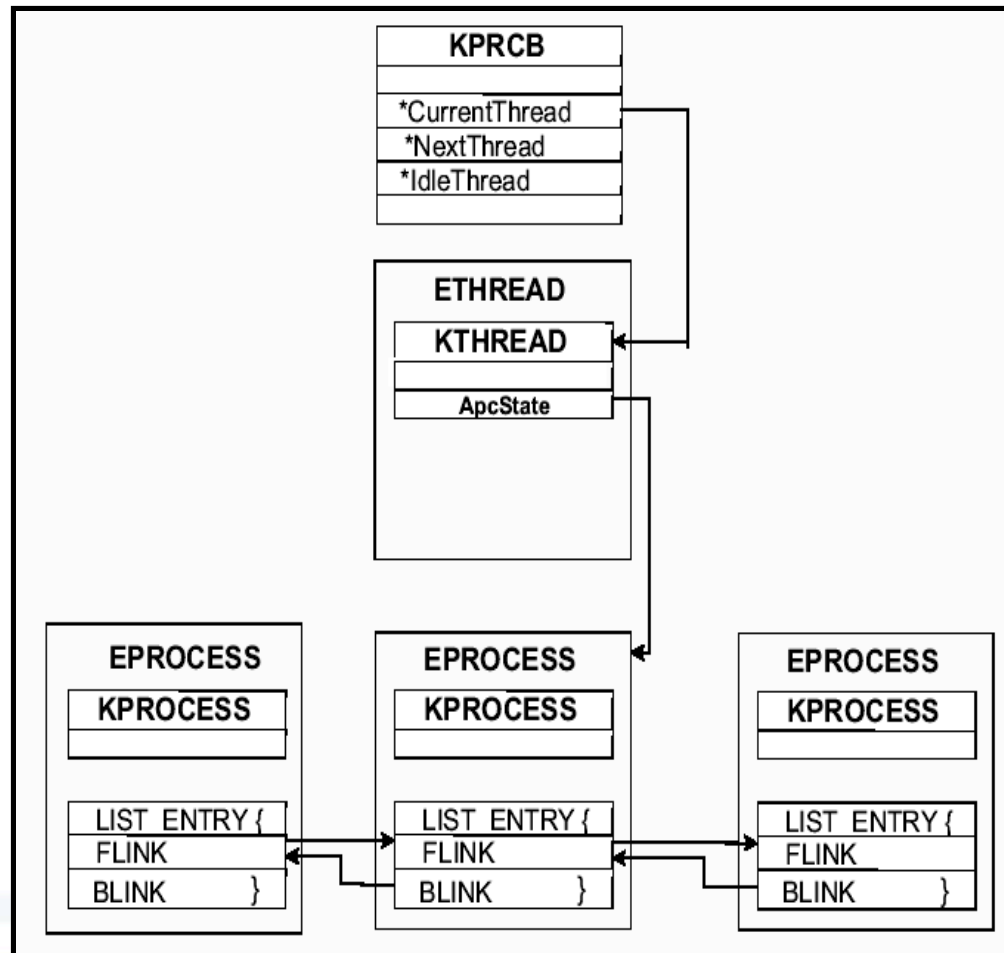


Third Generation Rootkits

- Modify dynamic OS objects loaded in memory.
 - Direct Kernel Object Manipulation (DKOM)
 - Example: FU Rootkit
 - Unlinks process objects from the Windows dynamically linked list of active process objects.
 - Kernel objects represent just about everything in the system (processes, threads, drivers, security tokens, ect.) so the possibilities are virtually unlimited.
 - Exclusively kernel mode rootkits.



Process Hiding w/ DKOM



Current Rootkit Detection Methods

- Behavioral Detection
- Integrity Detection
- Signature Based Detection
- Diff Based Detection



Rootkit File System Detection

- Signature Scanners – AV Products
- Integrity Checkers – Tripwire
- Diff Based Approach
 - Microsoft Strider GhostBuster
 - System Internals Rootkit Revealer
 - F-Secure Blacklight



Behavioral Detection

- Attempts to detect the effects of a rootkit on the victim system which means it may detect previously unknown rootkits.
 - Detecting diverted execution paths.
 - Deviations in executed instructions – PatchFinder by Joanna Rutkowska
 - Detecting Hooks – VICE by Jamie Butler
 - Detecting alterations in the number, order, and frequency of system calls.
- May suffer from a high false positive rate.
 - Most end users don't have the skill to screen out false positives.



Integrity Checking

- Detects unauthorized changes to system files or to loaded OS components in memory.
- Creates an initial baseline database containing their CRC values.
- Periodically calculates and compares the CRC's of these files against the initial trusted baseline.
 - Example: Tripwire
 - Files system integrity checks are ineffective against most modern rootkits which make their changes to memory rather than system files on the disk.



Signature Based Detection

- “Fingerprint Identification”
 - Searches memory or the file system for unique byte patterns (signatures) found in the rootkit’s code.
 - Tried N’ True Approach - Has been used by AV scanners for many years.
 - Highly accurate, but ineffective against unknown rootkit / malware variants (for which a signature does not exist) or deliberately obfuscated code.



Motivations

- Shortcomings Of Current Rootkit Technology
 - The most advanced public kernel rootkits are sitting ducks for primitive signature scans and integrity checking techniques.
 - Large parts of rootkit drivers themselves sit in non paged memory leaving them vulnerable to simple signature scans of system memory.
 - Rootkit modifications to operating system components in memory give them away to memory integrity checkers heuristic checkers like VICE.
 - Need a method to hide the rootkit driver code and its modifications to kernel memory.



Early Viruses Faced A Similar Problem

- Viruses sought to hide their code from file system signature scanners.
 - Their solution: Polymorphism / Metamorphism
 - Attempts to vary the appearance of the viral code from one variant to another.
 - Functionally equivalent, but semantically different copies of the code.
 - Few rootkits have integrated viral polymorphic techniques.



Introducing Shadow Walker

Prototype For A 4th Generation Rootkit?

- An alternative to viral polymorphism – *Virtual Memory Subversion!*
- Proof of concept demonstration that a rootkit is capable of transparently controlling the contents of memory viewed by other applications and kernel drivers.
- **Minimal performance impact !**



Implications Of Virtual Memory Subversion

- In-memory security scanners rely upon the integrity of their view of memory even if they don't rely upon Operating System API's (which may potentially be hooked).
- If we can control a scanner's memory reads we can fool signature scanners and potentially make a *known* rootkit, virus, or worm's code immune to in-memory signature scans!
- We can also fool integrity checkers and other heuristic scanners which rely upon their ability to detect modifications to code (i.e. VICE).

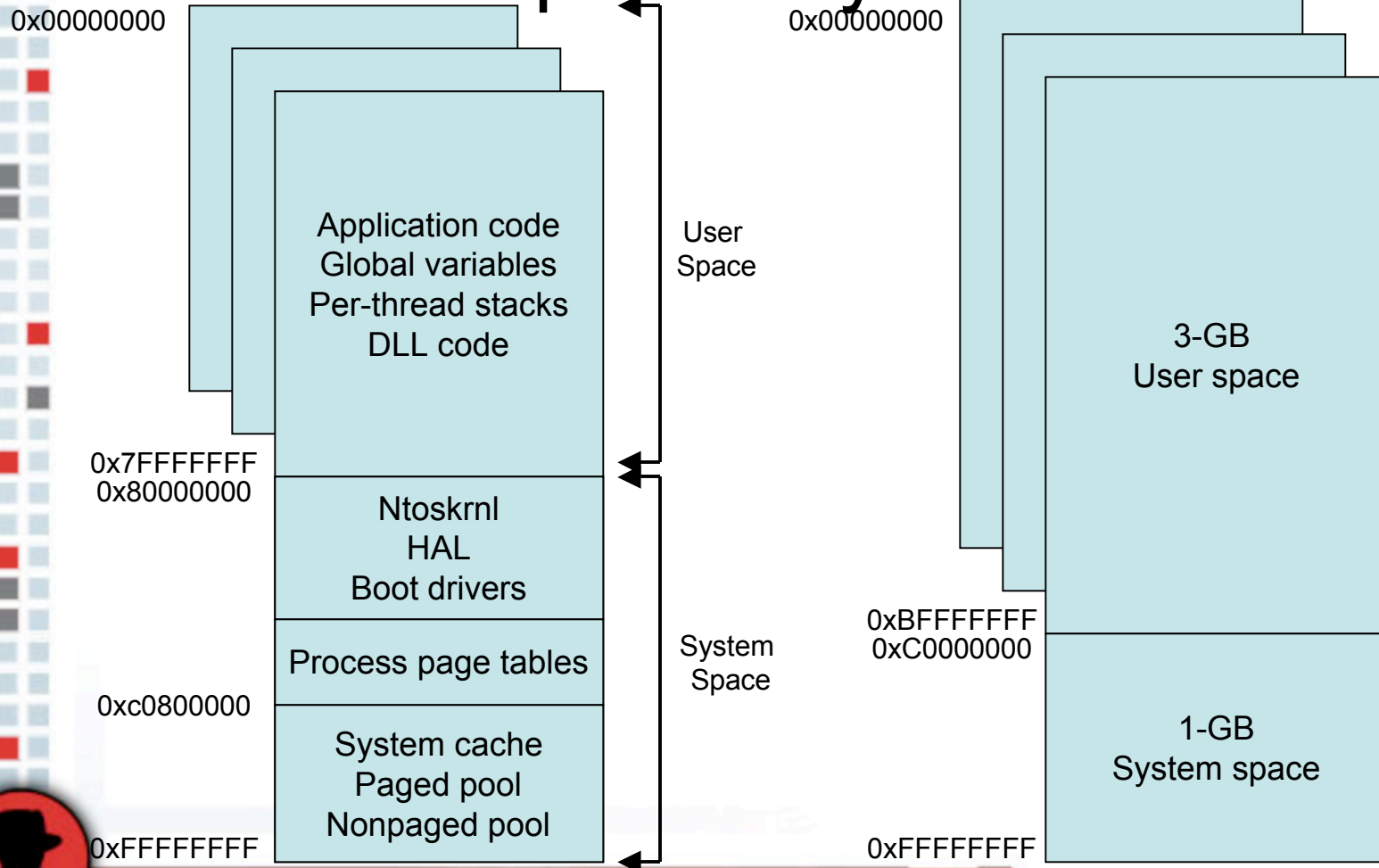


Review

- Windows virtual address space layout
- Virtual Memory
 - Paging vs. Segmentation
 - Page Tables & PTE's
 - Virtual To Physical Address Translation
 - The Role Of The Page Fault Handler
 - The Paging Performance Problem & the Translation Lookaside Buffer
 - Memory Access Types



Windows Virtual Address Space Layouts



Virtual Memory

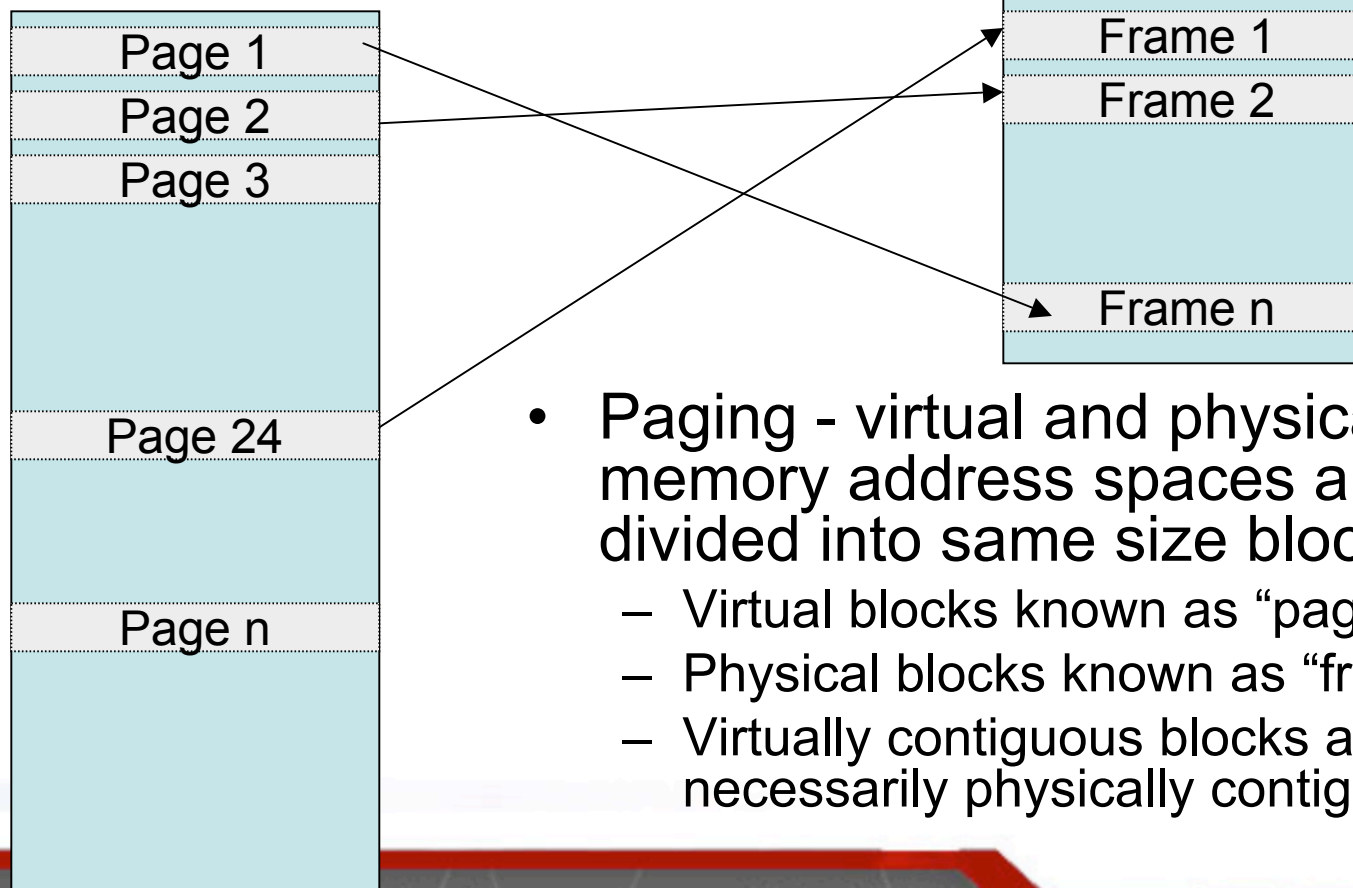
- Separate virtual and physical address spaces.
- Virtual & physical address spaces are managed by dividing them into fixed size blocks.
 - Paging: All blocks are the same size.
 - Segmentation: Blocks may be different sizes.
- The OS handles virtual → physical block mappings.
- Virtual address space may be larger than physical address space.
- Virtually contiguous memory blocks do not have to be physically contiguous.



Virtual To Physical Memory Mapping (Paging)

Virtual Address Space

Physical Address Space

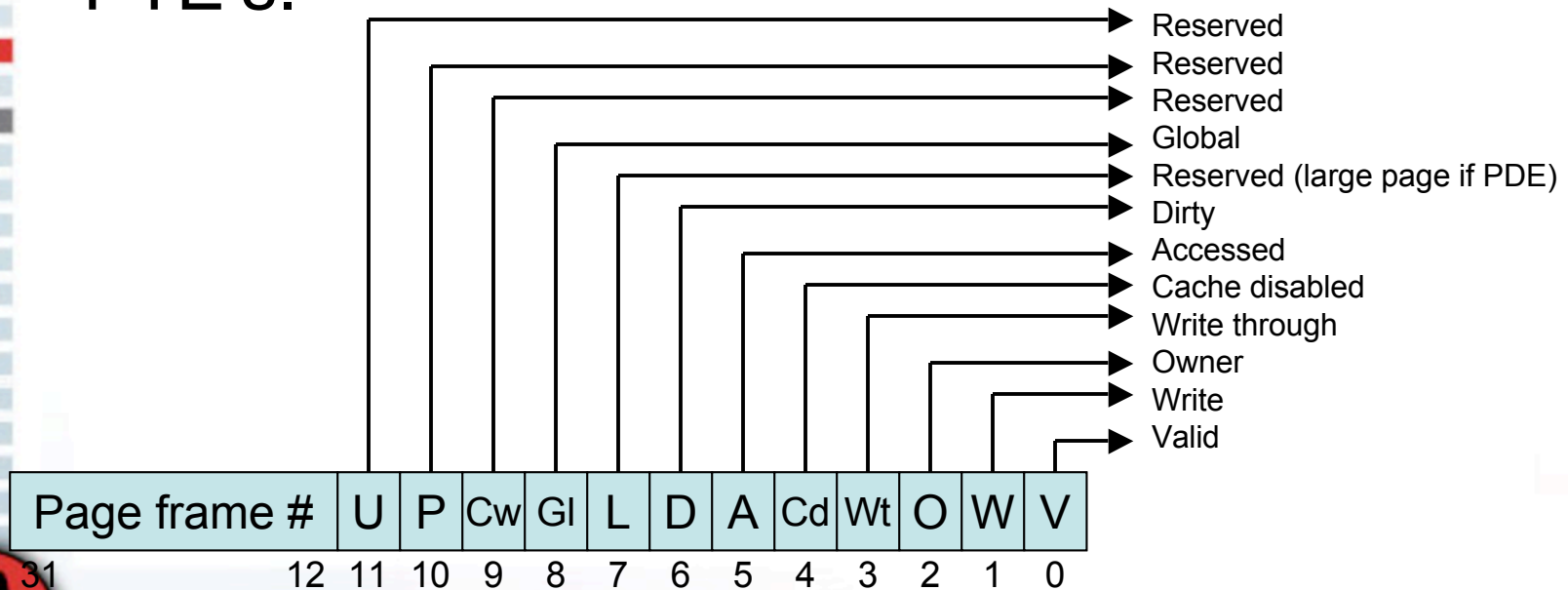


- Paging - virtual and physical memory address spaces are divided into same size blocks.
 - Virtual blocks known as “pages”.
 - Physical blocks known as “frames”.
 - Virtually contiguous blocks are not necessarily physically contiguous!

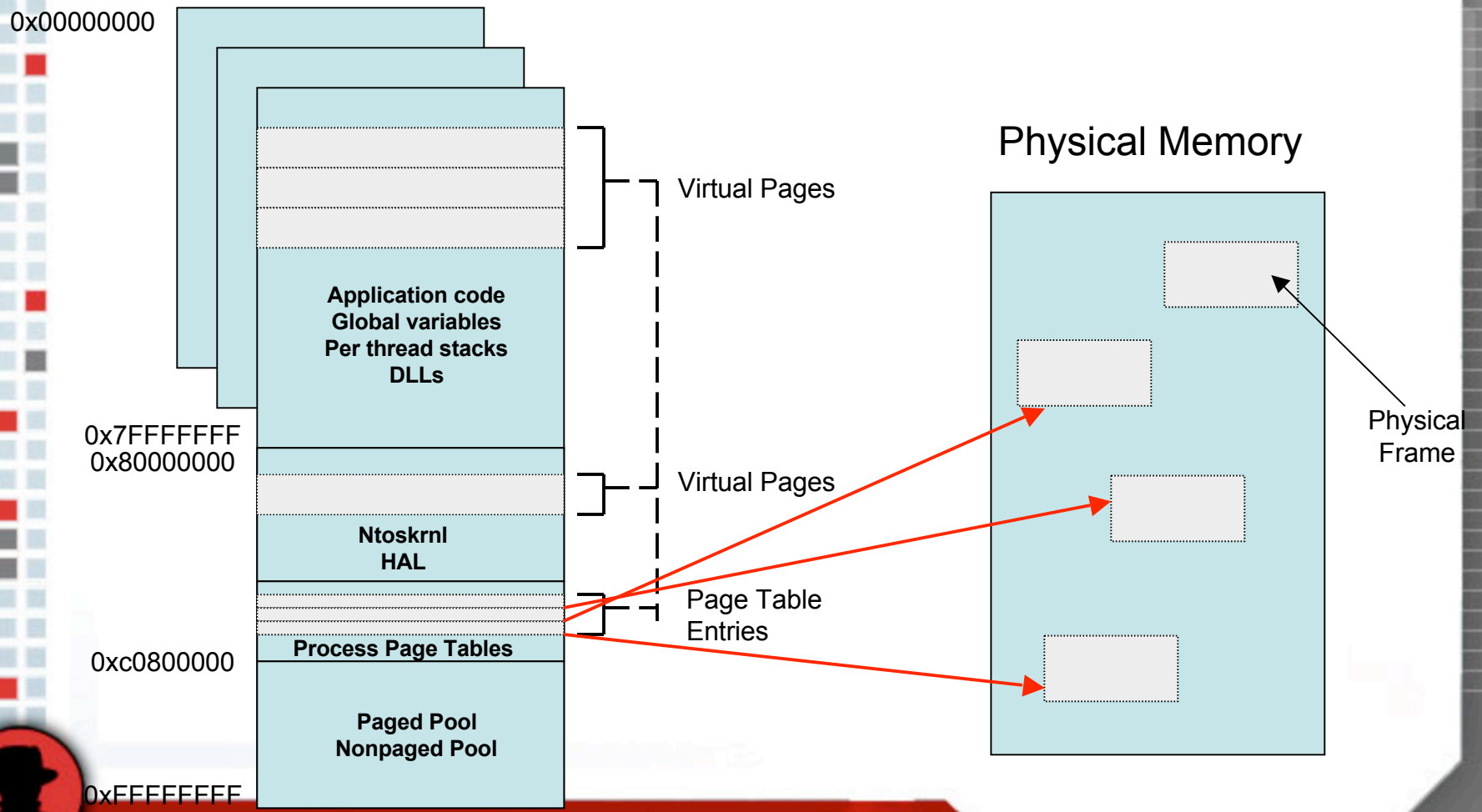


X86 PTE Format

- Virtual to physical mapping information is kept in page tables in structures called PTE's.

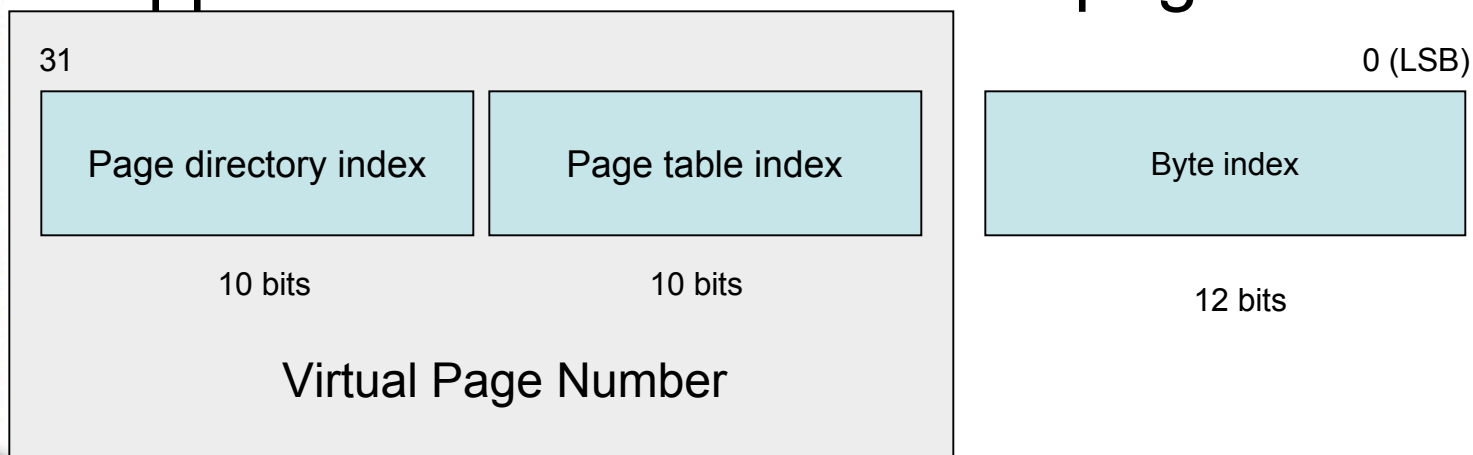


The Big Picture

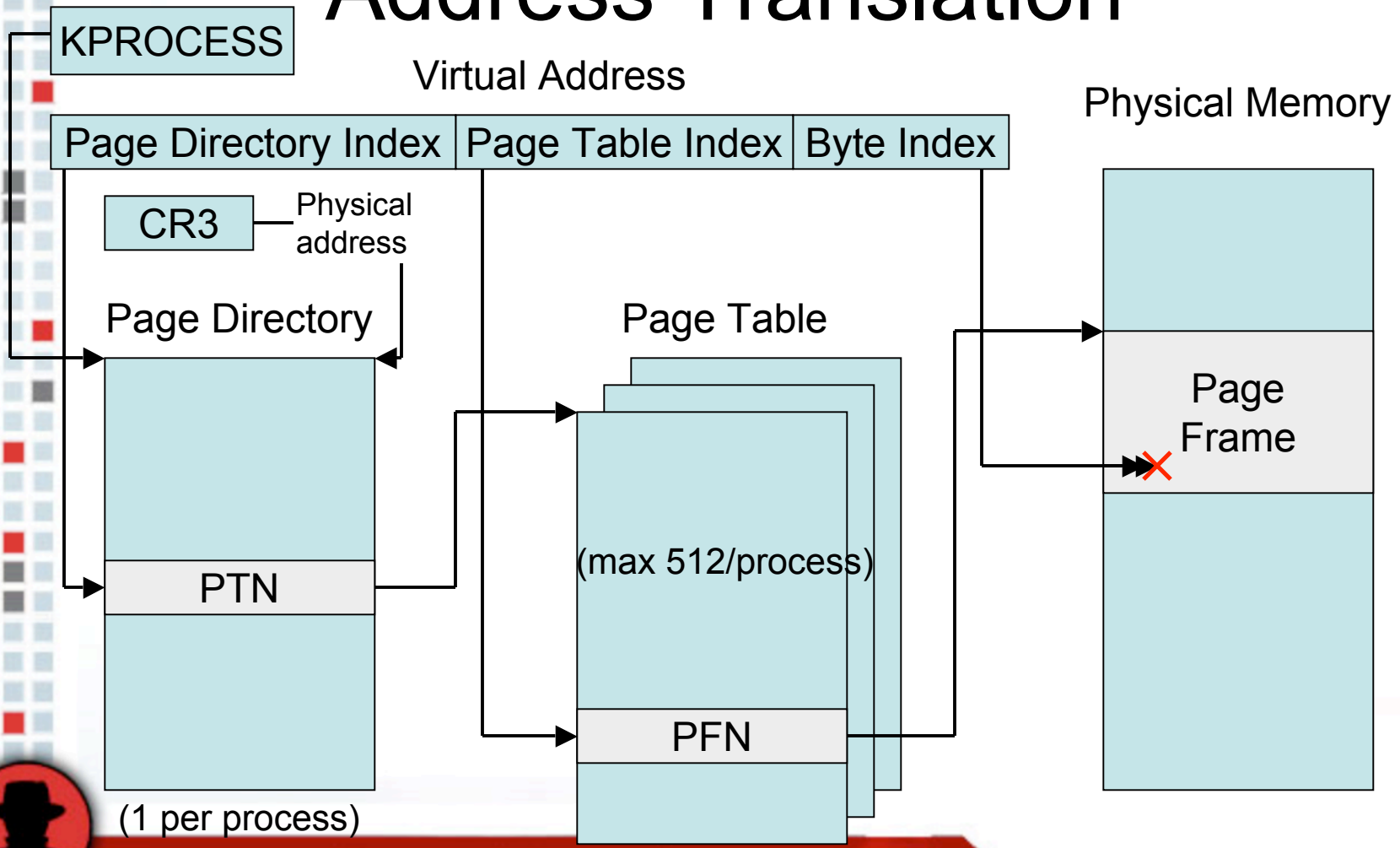


X86 Virtual Address

- Virtual addresses form indexes into page tables.
- Page tables may be single or multi-level.
- X86 uses a 2 level page table structure w/ support for 4K and 4MB sized pages.



X86 Virtual To Physical Address Translation



Page Faults

- Because physical memory may be smaller than the virtual address space, the OS may move less recently used pages to disk (the pagefile) to satisfy current memory demands.
- A page fault occurs on:
 - An attempted access to a virtual address whose PTE is marked not present and whose translation is not cached in the TLB.
 - Memory protection violations.
 - User mode code attempting to write to a kernel mode memory.
 - An attempt to write to memory marked as read-only.



Page Fault Path

Memory Access

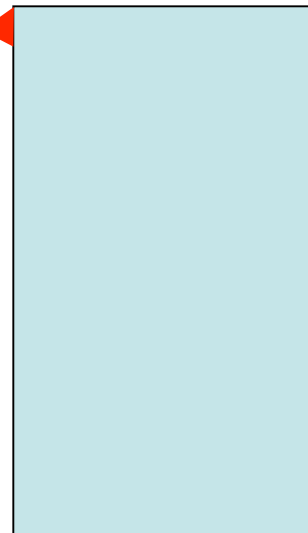
Page Directory

VPN=12, Frame =132 PRESENT

Page Table

VPN =12, Frame=27 NOT PRESENT

Page Fault Handler



Physical Memory



Disk (Page File)

The Paging Performance

- Virtual memory incurs a steep performance hit!
- 2 level page table scheme like x86:
 - Best Case: 3 memory accesses per reference!
(page dir + page table + offset)
 - Worst Case: 3 memory accesses + 2 disk I/O requests per memory reference!
(page dir + I/O + page table + I/O + offset)
- Solution: Translation Lookaside Buffer (TLB)
 - The TLB is a high speed hardware cache

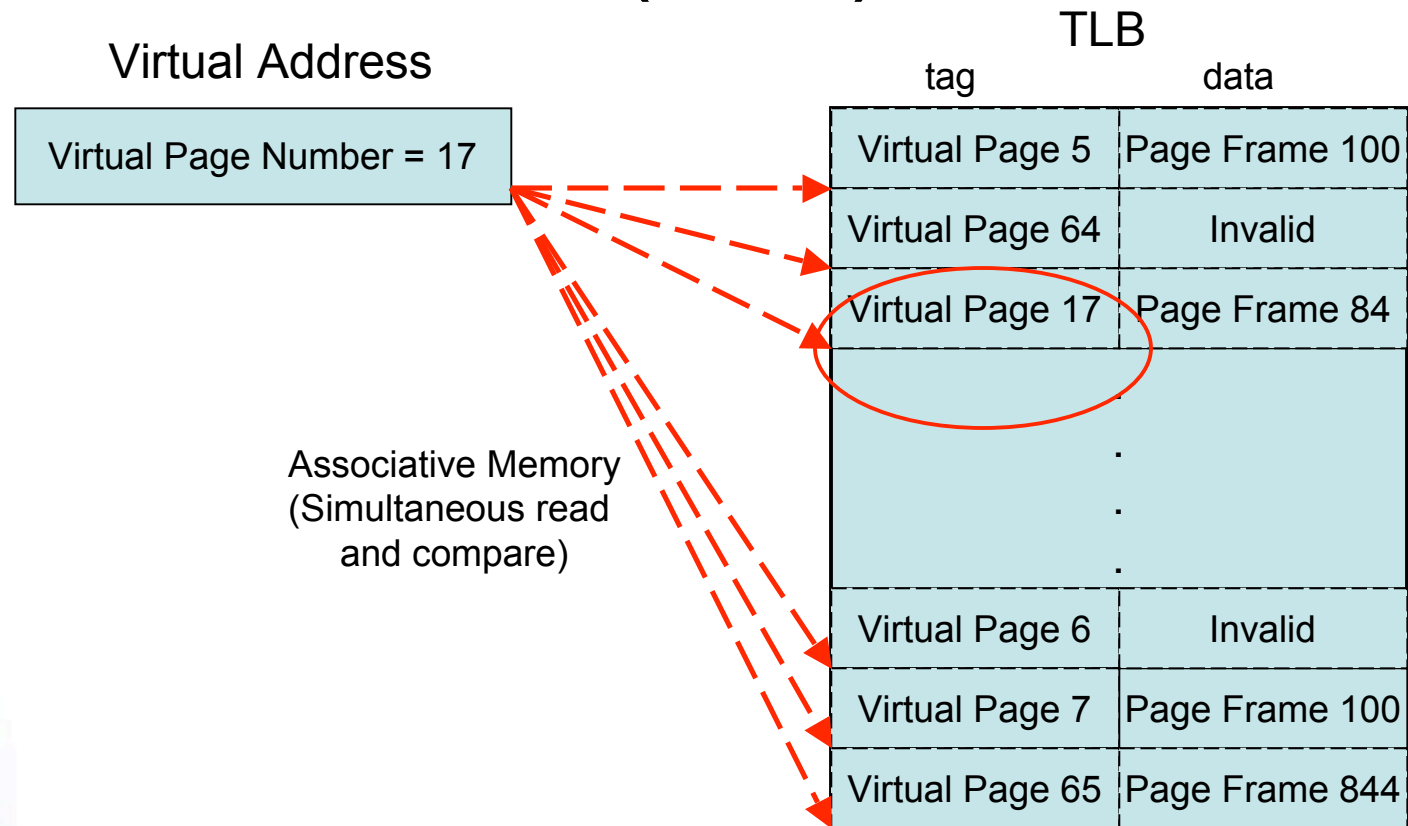


Translation Lookaside Buffer

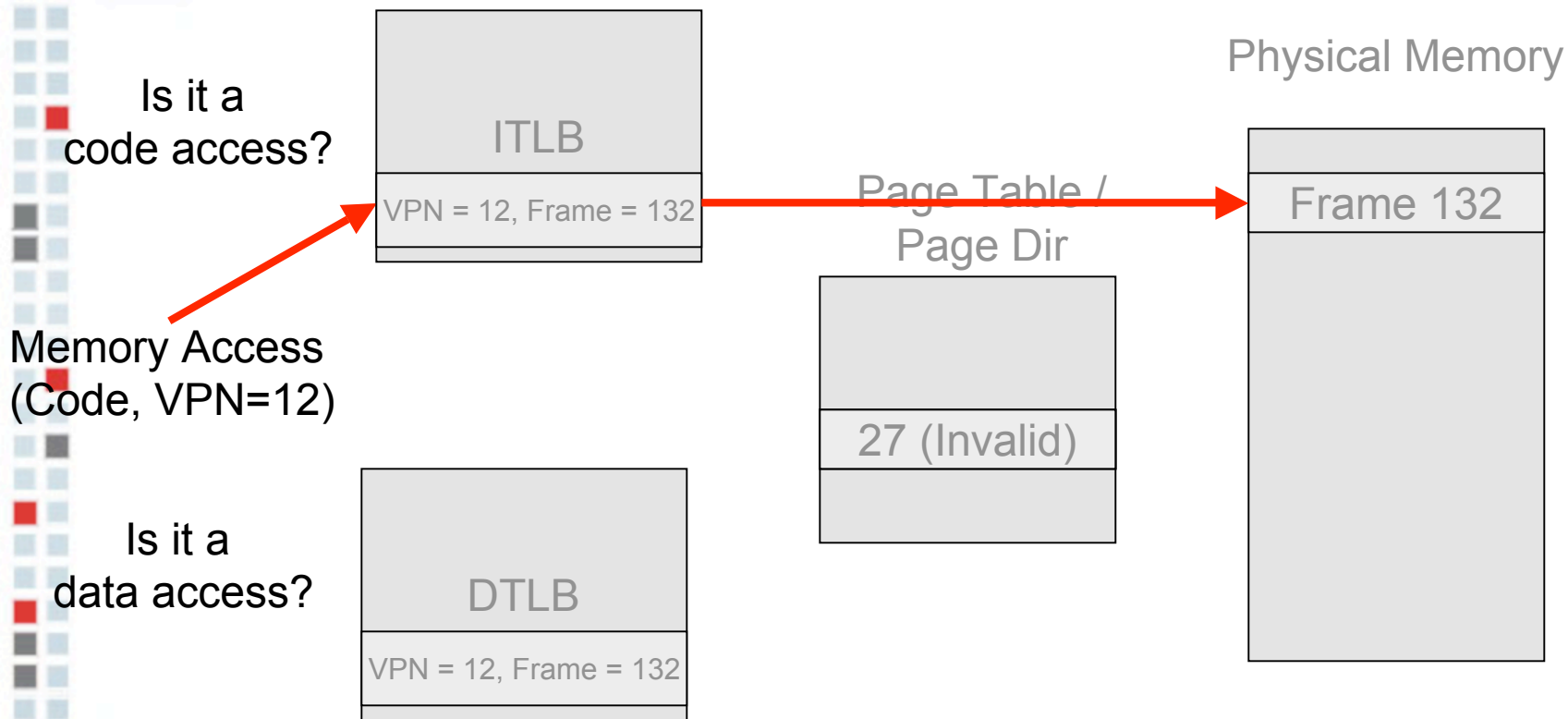
- On memory access, TLB is searched first for the virtual to physical translation!
- High speed associative memory
 - “Hit” → translation was found in the TLB
 - “Miss” → translation was not found in the TLB
- X86 Uses Split TLB architecture
 - ITLB: holds virtual to physical translations for code
 - DTLB: holds virtual to physical translations for data
- Modern TLB’s have extremely high “hit” rates and seldom incur the performance hit of a page table walk.



Translation Lookaside Buffer (TLB)



Memory Access Path w/ TLB Hit



Frame 0

Frame 122

Frame 27

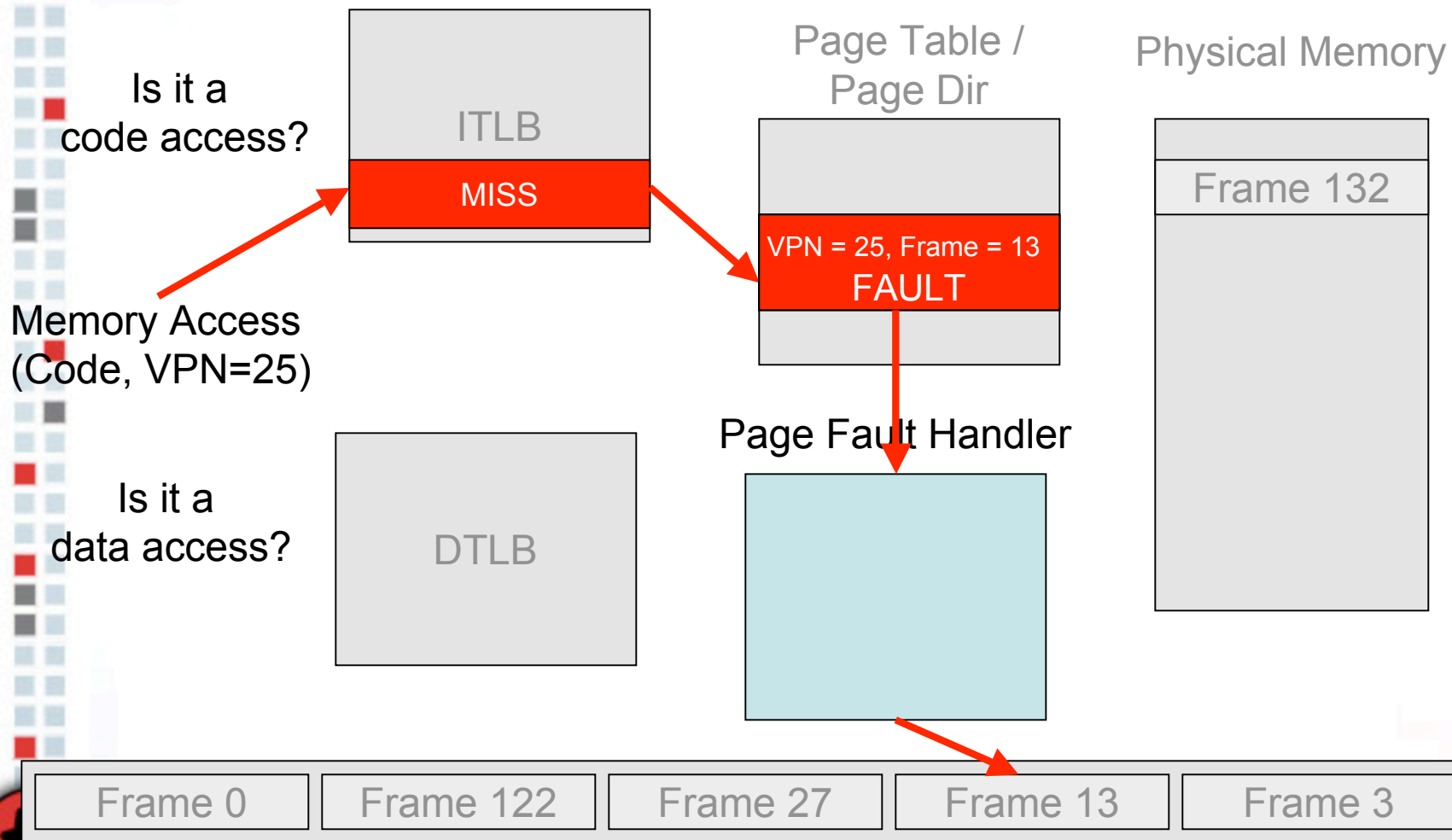
Frame 13

Frame 3

Disk (Page File)

Black Hat Japan 2005

TLB Miss w/ Page Fault



Memory Access Types

- Basic memory access types:
 - Read
 - Write
 - Execute
- Under IA-32, execute access is implied:
 - Read / Execute
 - Read / Write / Execute



NX?

(Non Executable Memory)

- For some applications it is advantageous to be able to differentiate between read / write and execute accesses.
 - Buffer Overflow Protection
- IA-32 does not provide hardware support for non executable memory
 - PaX → Read / Write / No Execute memory semantics on the IA-32 with software support
 - Windows XP SP2 and Windows Server 2003 SP1 added OS software support for NX.
 - Side Note: Hardware support for NX memory has been added to some processors including AMD 64 processors, some AMD sempron processors, IA-64, and Intel Pentium 4.



Hiding Executable Code

- Use PaX technology offensively.
- To hide code, we want to differentiate between read / write and execute accesses to the hidden code.
 - Read accesses of the code section of a rootkit driver may indicate presence of a scanner.
 - Nearly the inverse of PaX



Implementation Issues

- We need a way to filter execute and read / write accesses.
- We need a way to “fake” the read / write memory accesses when we detect them.
- We need to ensure that performance is not adversely affected.



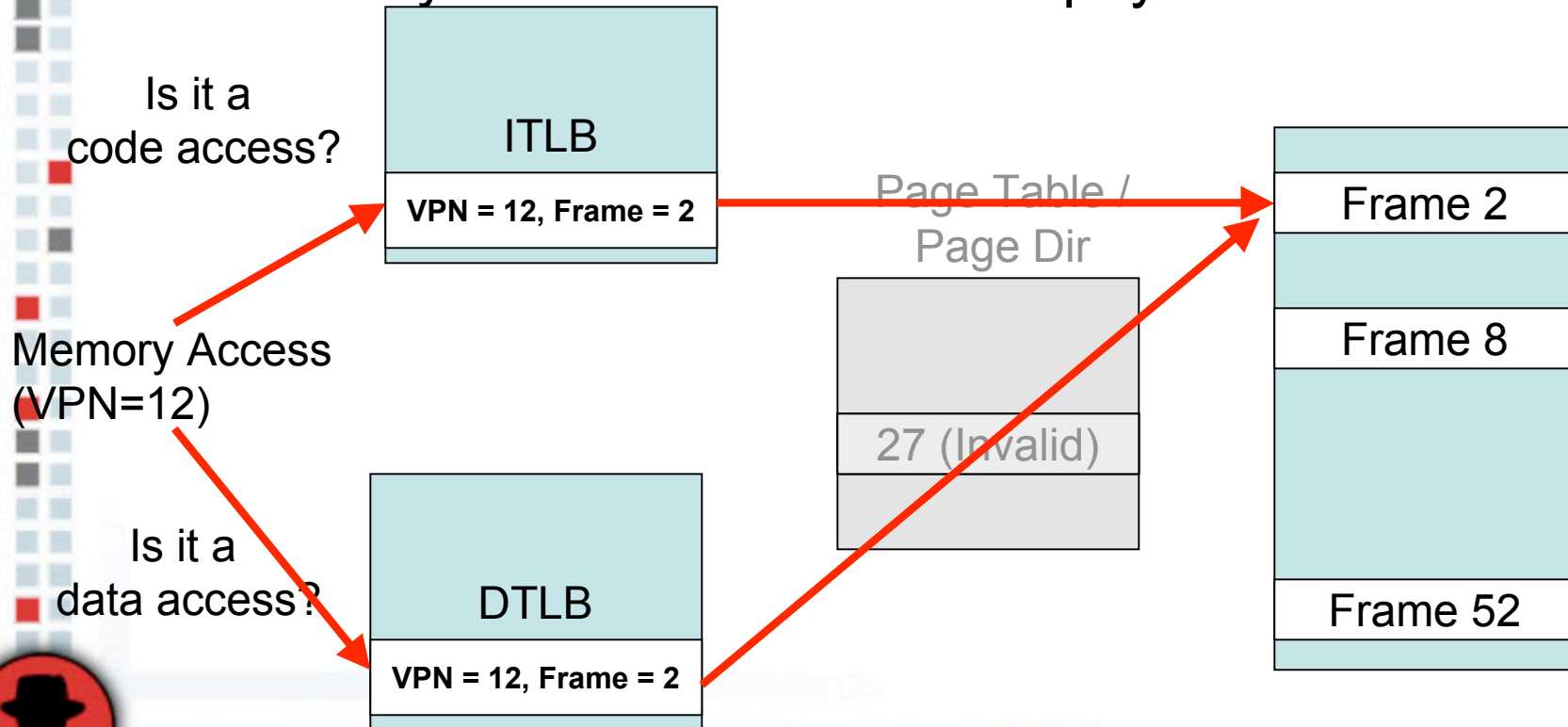
Differentiating Between Execute and Read / Write

- We can trap memory accesses by marking their PTE's "non present" and hooking the page fault handler.
 - If **instruction pointer == faulting address**, then it is an execute access! Otherwise, it is a read/write.
- We also need to differentiate between page faults due to the memory hook and normal page faults.
 - Pages must be nonpaged memory.
 - Pages must be locked down in memory.



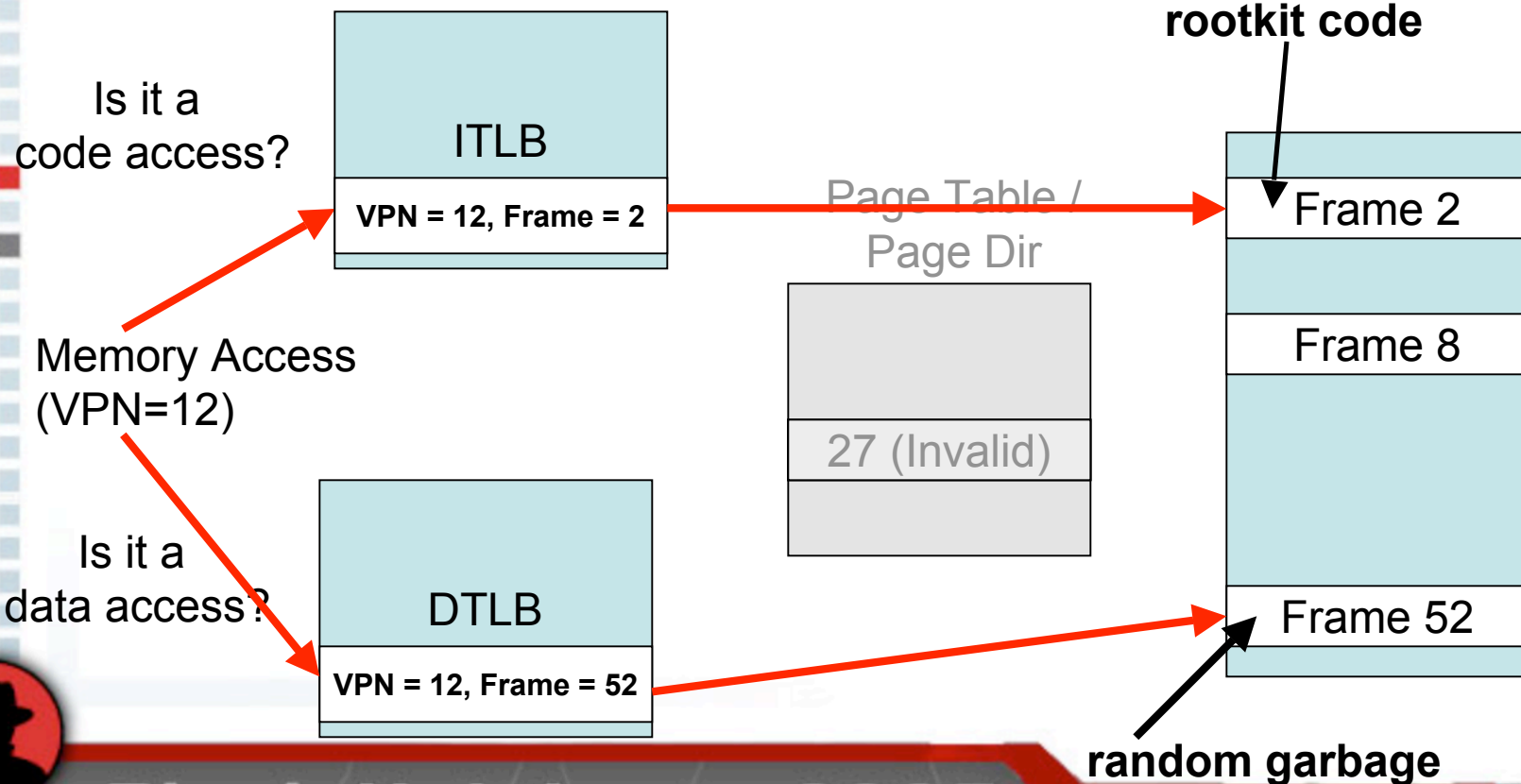
Faking Read / Writes By Exploiting The Split TLB (1)

- Normal Synchronized ITLB and DTLB translate code and data memory accesses to the same physical frame.



Faking Read / Writes By Exploiting The Split TLB (2)

- Desynchronized ITLB and DTLB translate code and data memory accesses to different physical frames.



Software TLB Control

- Reloading cr3 causes all TLB entries except global entries to be flushed. This typically occurs on a context switch.
- The invlpg causes a specific TLB entry to be flushed.
- Executing a data access instruction causes the DTLB to be loaded with the mapping for the data page that was accessed.
- Executing a call causes the ITLB to be loaded with the mapping for the page containing the code executed in response to the call.



Shadow Walker Components

- Memory Hook Engine
 - Hook Installation Module
 - Custom Page Fault Handler
- Modified FU Rootkit



Memory Hook Installation

- Install new PF handler (Int 0E).
- Insert page into global hash table of hooked pages for quick lookup in PF handler.
- Mark page not present.
- Flush the TLB to ensure that we trap all subsequent memory accesses in the PF handler.



Custom Page Fault Handler

- Primary task is to filter read / write and execute accesses for hooked pages.
 - Passes down faults on unhooked pages to the OS page fault handler.
 - Manually loads ITLB on execute access to hooked page.
 - Manually loads DTLB on data access to hooked page.
- Most memory references will be resolved via the TLB path and will not generate page faults.
- Page faults on hooked pages will occur:
 - On the first execute and data accesses to the page.
 - On TLB cache line evictions of a hooked mapping.
 - On explicit TLB flush (i.e. context switch).



PF Handler Pseudocode

- Pseudocode for enforcing execute diverted read / write semantics on kernel pages.

Page Fault Handler:

```
if( ProcessorMode == USER_MODE )
    jmp PassDownToOs

if( FaultingAddress == USER_PAGE )
    jmp PassDownToOs

//faulting address is from a hidden page
if( FaultingAddress == HIDDEN_PAGE)
{
    if(FaultingAddress == EIP)
        jmp LoadItlb //execute access
    else
        jmp LoadDtlb
}
else jmp PassDownToOs
```

Load Itlb:

```
ReplaceFrame(PTE.FaultingAddress)
PTE.FaultingAddress == PRESENT
CallIntoHiddenPage //load ITLB
PTE.FaultingAddress == NOT PRESENT
ReplaceFrame(old_PTE.FaultingAddress)
jmp ReturnWithoutPassdownToOs
```

Load Dtlb

```
PTE.FaultingAddress == PRESENT
ReadFaultingAddress //load DTLB
PTE.FaultingAddress == NOT PRESENT
jmp ReturnWithoutPassdownToOs
```



What About Data Hiding

- Less optimal
 - Hidden page must not be in DTLB.
 - One page fault per data access to hidden page.
- For memory access to go through, data accesses must be filtered in PF handler and the DTLB must be loaded w/ the correct mapping.
- Memory hook must maintain control (i.e. DTLB must be flushed after access).
 - Protocol based approach.
 - Debugging approach (i.e. single step).



Modified FU Rootkit

- Proof of concept rootkit hidden by our memory hook engine.
- Runs as a system thread and regularly scans the active process list looking for processes named `_fu_` and unlinks them.
- No dependence on userland initialization.
 - No symbolic link
 - No functional device
- In memory rootkit
 - Could be installed from a kernel exploit to avoid disk detection.



Impact On System Performance

- Modern TLB's have extremely high "hit" rates.
- Therefore, most translations on our "hidden pages" will go through the TLB path rather than the slower page fault path.
- Using the memory hook engine to hide a rootkit driver's code has no subjectively noticeable impact on overall system performance!



Known Limitations

- No PAE Support
- No Hyperthreading / Multiprocessor Support
- Currently hides only 4K sized kernel pages (i.e. system space / upper 2 GB of the virtual address space).



Detection

- Non present pages in non paged memory range are abnormal!
- The PF handler code itself cannot be concealed using this technique since we cannot mark the page containing the PF handler non present (must use polymorphic solution).
- Difficult to conceal IDT hooks (i.e. PF handler).
- Cannot protect against remapping.
 - Virtual memory scans are inadequate!
 - More advanced scanners should be based upon remapping of the physical address space.
 - Hardware memory scanners.



Yin and Yang

- There are defensive applications
 - IDS, AV, Firewall Drivers
- Rootkits and other malicious software often compromise security software via in memory patching.
- Execute / Diverted Read-Write semantics can be used to provide light weight code integrity.
 - Accesses to a security driver's code section can be deflected to a separate "shadow" page frame where they would have no effect!



DEMO



Black Hat Japan 2005

?



Black Hat Japan 2005

References / Acknowledgements

- The PaX Project
- Halvar Flake
- Joanna Rutkowska
- A generic attack on checksumming-based software tamper resistance by Glenn Wurster P.C. van Oorschot, and Anil Somayaji
 - Concurrent, related work on memory subversion.

