

GRAYSCALE RESEARCH::2008



# The WIN32 Memory Model: **Tying it all together.**

Presented for DC619



# Why Study Memory Mechanics

- To understand process mechanics.
- To learn the subtlety between physical, and virtual space.
- To understand process space manipulation techniques.
  - Process Object Manipulation
  - Table Hooking
  - Driver Chaining
  - DKOM Rootkitting Techniques
  - General Purpose Runtime Alterations
- Because you have to, and you **cant** be an evil hacker without it.





# Prerequisites

**Tenacity. Also probably, C.**



# Tools and Dox

## Tools:

WinDBG: Available in the Windows Debugging Tools package (free)

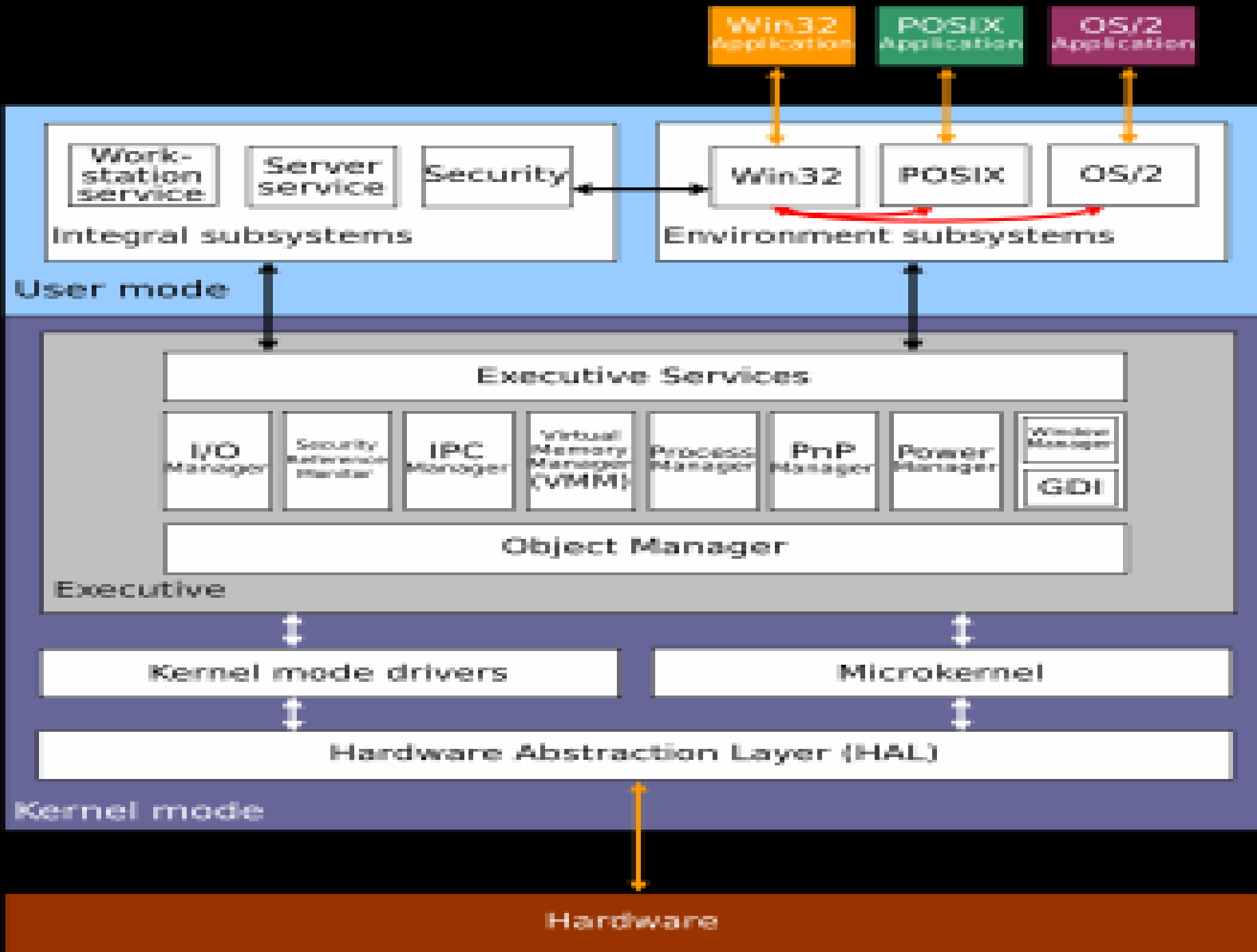
WDK: Newly available driver development kit (free with ms live acct)

Visual C++/Studio: Express Edition (free), Studio (cash).

## Dox:

Windows Userland API: Google for "Windows API Reference" for dox.

Windows Kernel API: Google for "Kernel-Mode driver reference".





# Physical Memory

- Is Literal and existent and can be referenced by hardware testing.
- Called "linear" or "absolute" addressing.
- Has no bearing to context, or scope, and has no real operational functionality in itself besides being referenced by the system processor.

The word memory in reference to windows, is a carte blanche term which is used to reference any and all utilization of physical memory. There is literally a [\\device\physicalmemory](#) device which can be accessed from kernel land to directly access the raw memory on your system.

Note: In windows xp and below, a system process can access this device from userland!!!





# Virtual Memory

- It is ethereal and contextual
- It has philosophical premise in efficiency, and expansion.
- It is quite literally, process space.

Virtual memory is an idea thought up by processor manufactures quite a long time ago, which provides a process its own set of "page tables". These page tables create an environment in which a process can have its own virtual address space. This allows all processes to have essentially, their own parallel operating environment inside a system.





# Virtual Memory and Context

In terms of conversation, a context is the mood, setting, dialogue, and content of a conversation. In the scope of a process, it could be considered to be similar, except all the different elements, are instead of human contextual elements, they are processor register values instead.

Just as ESP and EBP are utilized for context in stack operations, the Intel CR3 register is used as context in virtual memory.





# Virtual Memory and Pages

A page, in essence, is 4096 bytes of memory. Contiguous address space in a process, is simply a side by side layout of different pages.

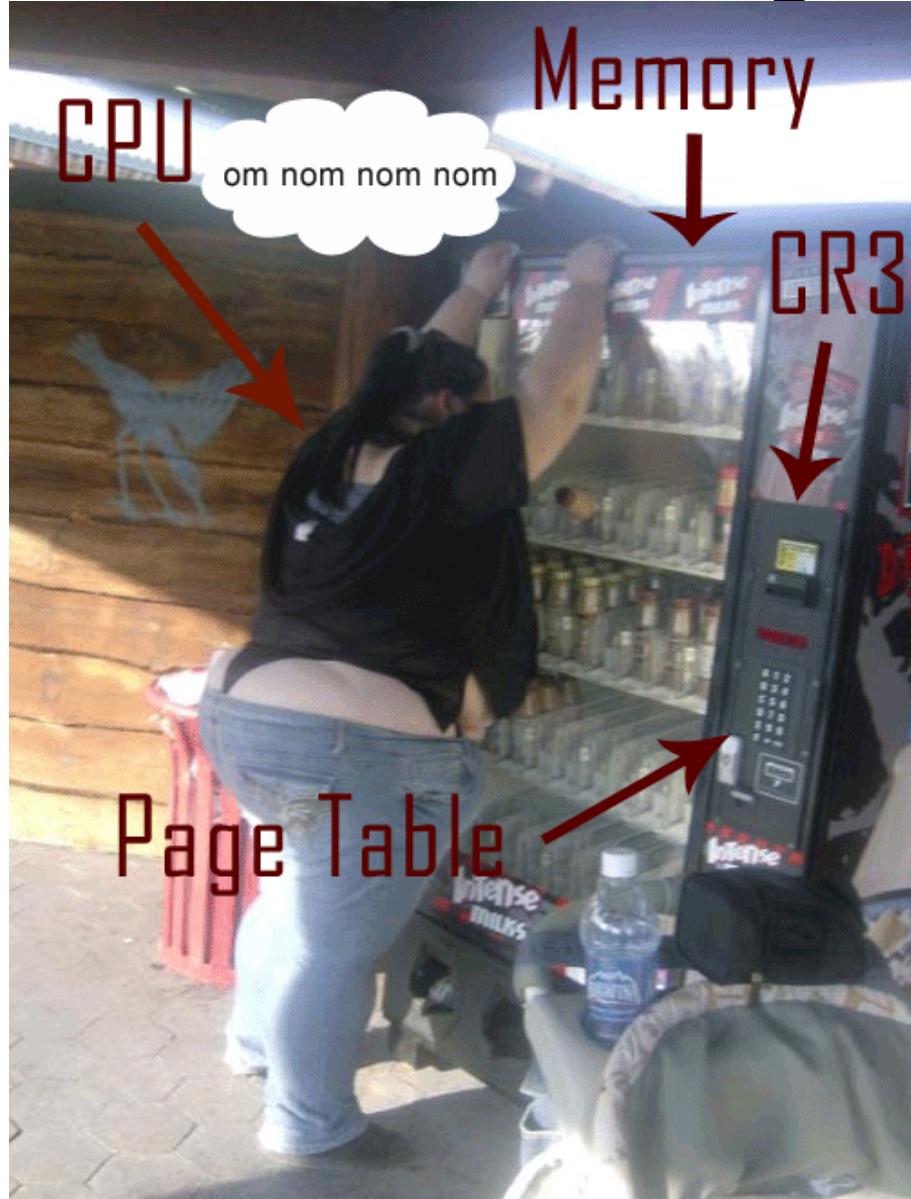
Pages additionally allow a processor to “swap” data onto the hard disk by using a tripwire mechanism called “page faults”.

In order to look up pages, the CPU must use what is known as a page table. The CR3 register contains a pointer to a “page table” table, which can be used to then enumerate available pages.





# The Monster and the Vending Machine



CPU om nom nom nom

Memory

CR3

Page Table



# Virtual Memory and Processes

As everyone most likely knows, a process executes along the notion of timeslices. These timeslices are in essence, shared CPU time. Well, in order for timeslicing to be effective, the processor is essentially duplicated in each process space so that processes can be in control of their own timesliced, virtual processor.

## **Each process thinks:**

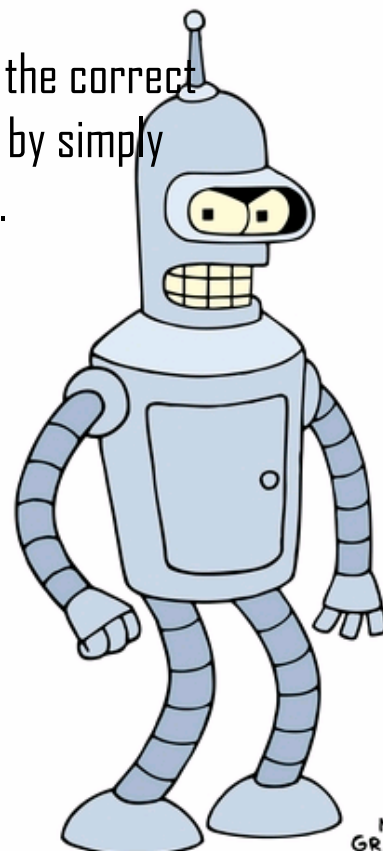
- It has its own physical memory
- Its own processor
- Its own complete system



# Operating Systems, Processes, and Contexts

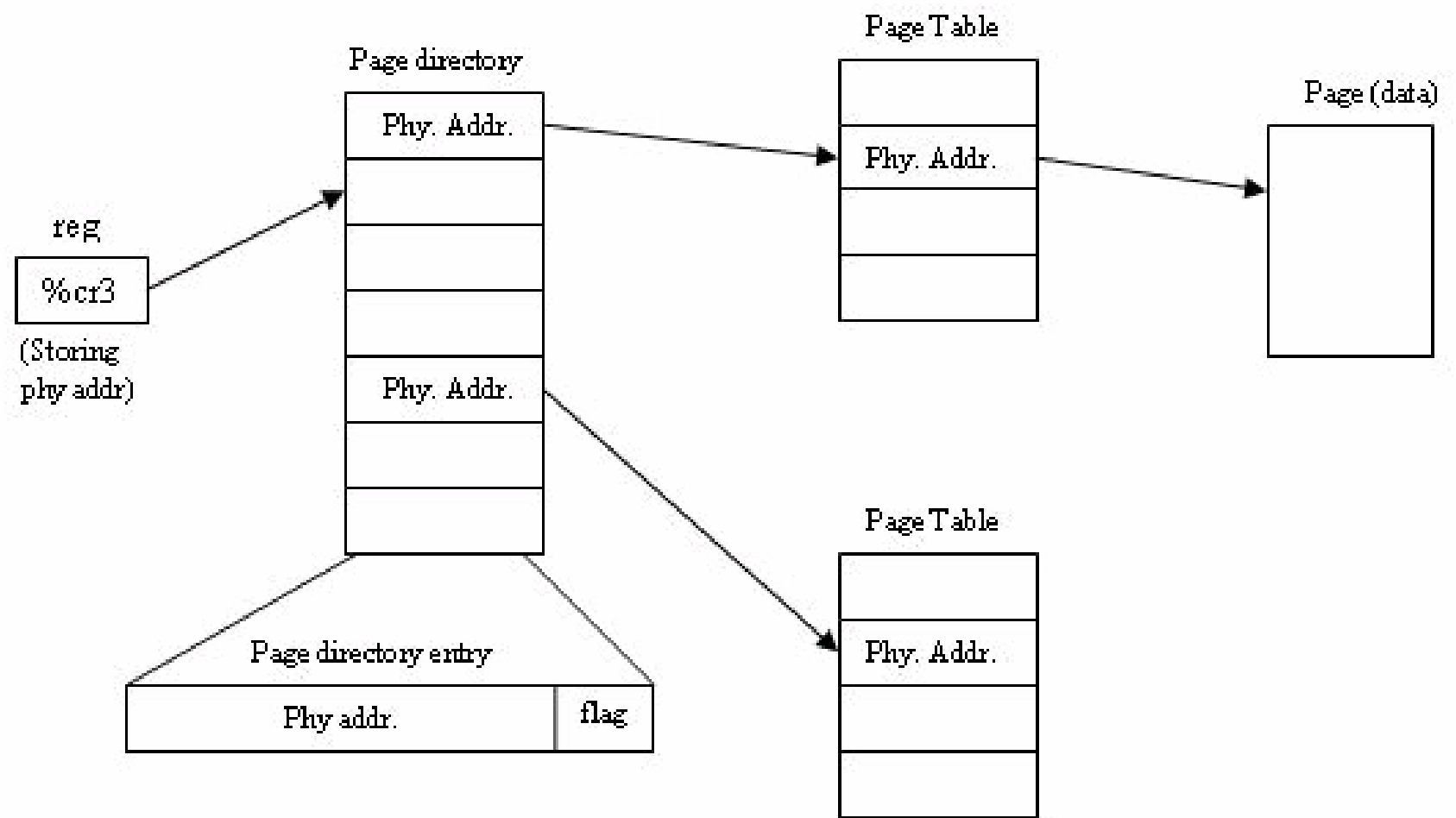
In order to timeslice effectively, the operating system switches between processes by utilizing contexts. As was said in an earlier slide, a context is just a set of register values that can be swapped in and out of a processor or pseudo processor easily.

In terms of processing, context can be switched very quickly simply by plugging in the correct values of a processes registers. For virtual memory, a context switch is provided by simply swapping out the processes CR3 register, with that processes page table address.





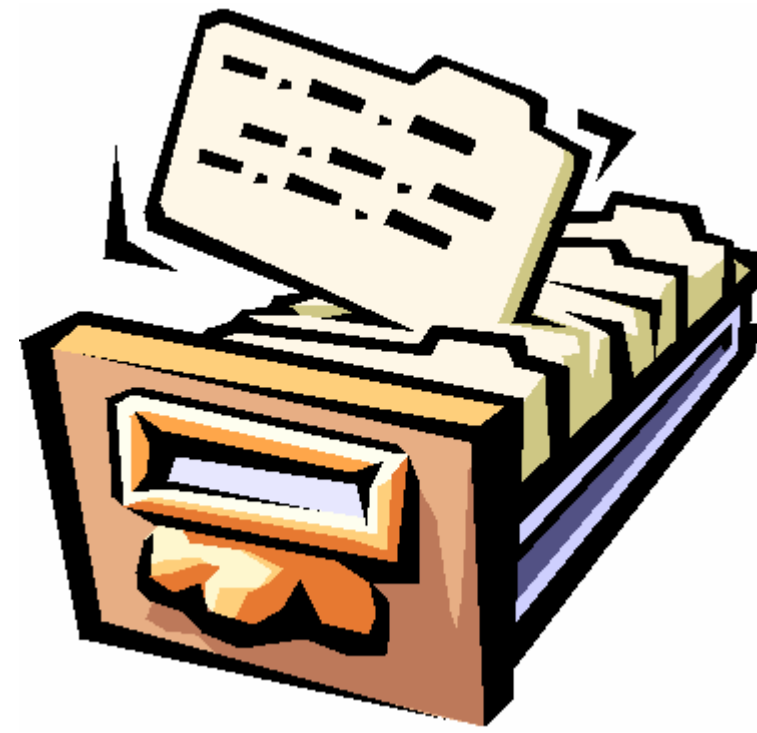
# "Page Table" Table



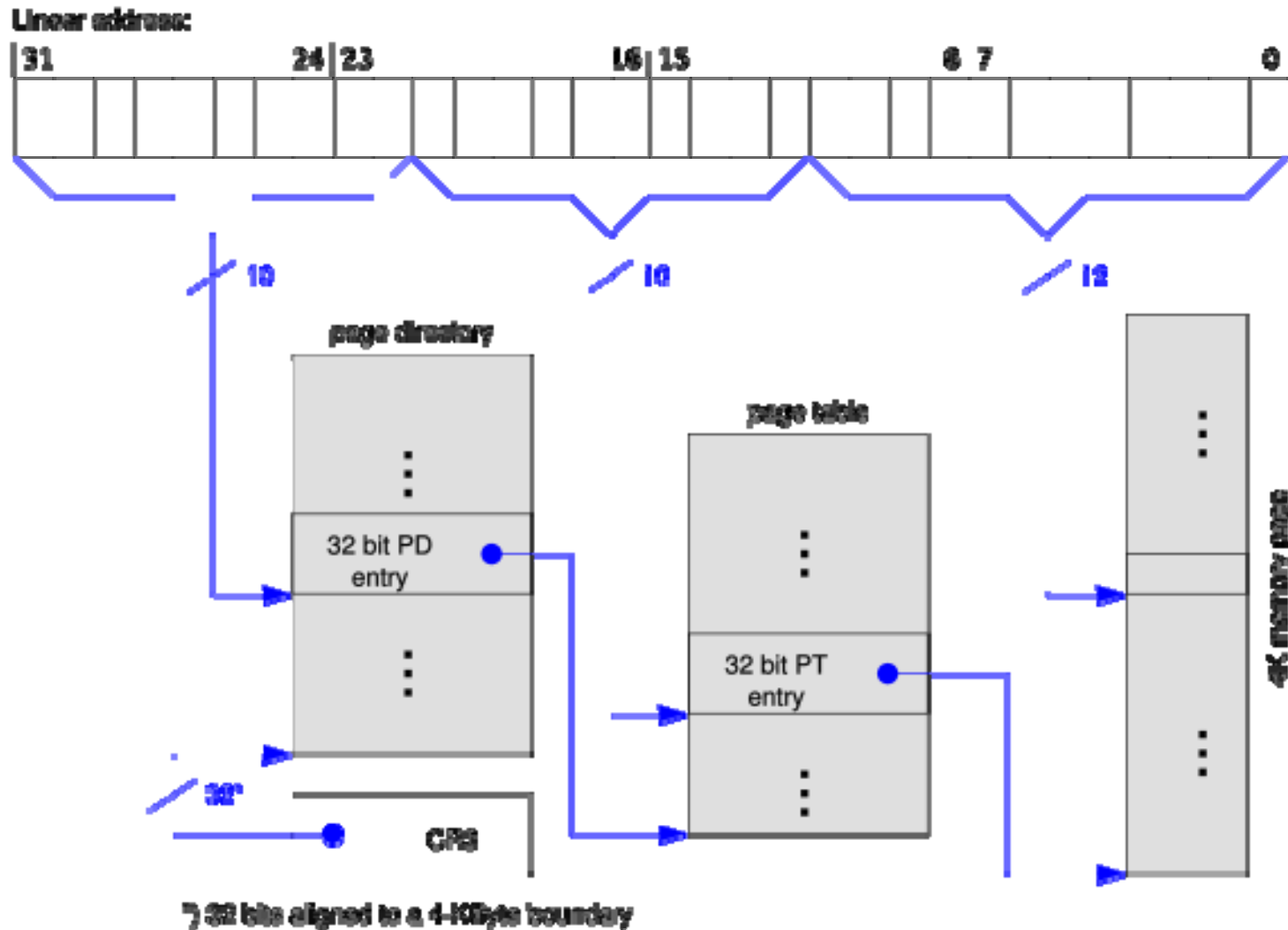


# Page Table Entries (Linear)

- Each entry in any given page table is 32bits.
- The first 20 bits of an address is the offset to the page in physical memory.
- The remaining 12 least significant bits, are used as an offset into the exact page.



# Linear (Physical) Addressing Form





# Translation Lookaside Buffer

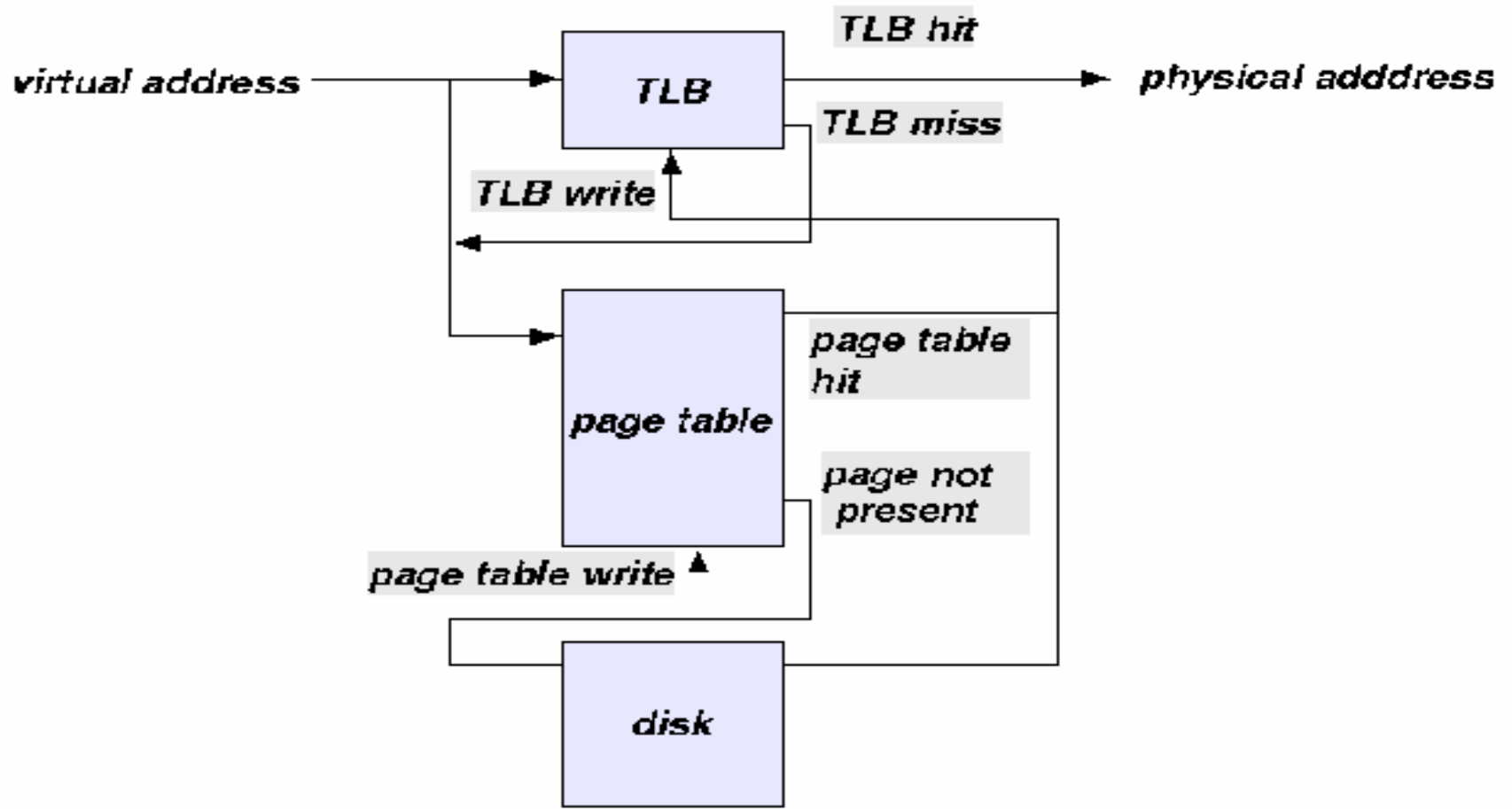
To improve the speed of looking up pages, CPU manufactures developed what is known as the TLB or translation lookaside buffer.

This simple construct is a cache of recently used page entries from the page table, which is searched FIRST, before the page table, when translating addresses.





# How the sprockets really turn.



# Win32: Accessing Userland Process Memory

Windows provides a large API which can be utilized to easily access memory for in processes.

|                      |   |
|----------------------|---|
| OpenProcess()        | Retrieve a handle to a process.   |
| ReadProcessMemory()  | Read memory from a process handle (virtual memory).                                 |
| WriteProcessMemory() | Write memory to a process.  |
| GetThreadContext()   | Retrieve a threads context.   |
| SetThreadContext()   | Sets a thread context.  |
| VirtualQuery()       | Inquire and retrieve process memory ranges.   |
| GetSystemInfo()      | Retrieve lowest and highest possible memory ranges (useful for enumerating ranges). |



# Win32: Making Sense of Process Memory

The windows snapshot API can be used to enumerate a significant amount of information about running process that can easily make sense of a running processes memory layout.

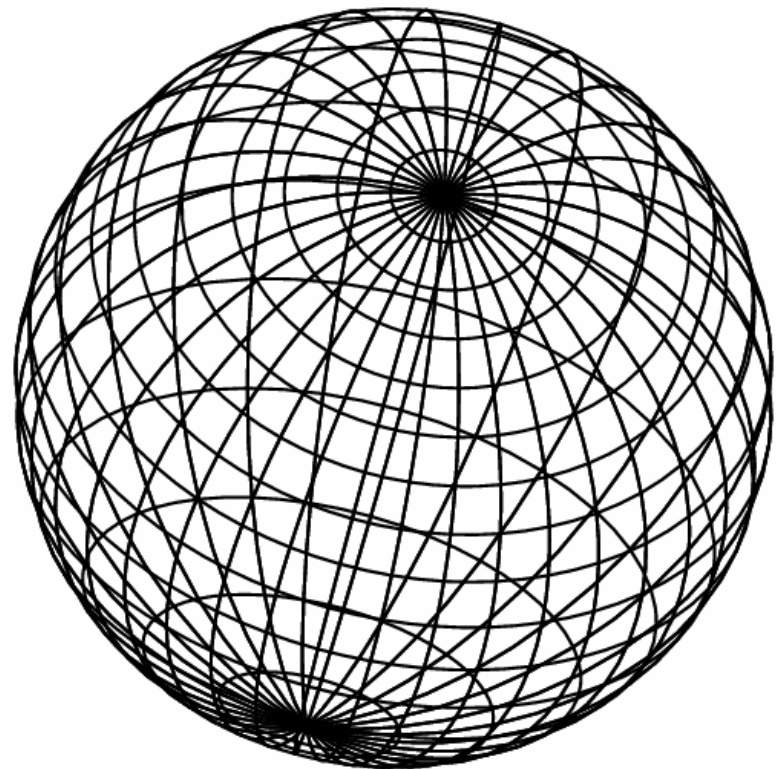
|   |  |
|---|--|
| <code>CreateToolhelp32Snapshot()</code> | Retrieve a handle to a process snapshot.                           |
| <code>Heap32ListFirst/Next()</code>     | Retrieve a full list of process heaps.                             |
| <code>Module32First/Next()</code>       | Retrive a list of process modules, sizes, permissions, and ranges. |
| <code>Process32First/Next()</code>      | Retrieve a process list for all processes on the system.           |
| <code>Thread32First/Next()</code>       | Retrieve a thread list for a given process.                        |



# Process Modules

Modules contain executable code in the PE binary format, which can be extracted directly from memory using read process memory.

This information includes DLLs and EXE files, and can be used to reconstruct binaries from memory alone.

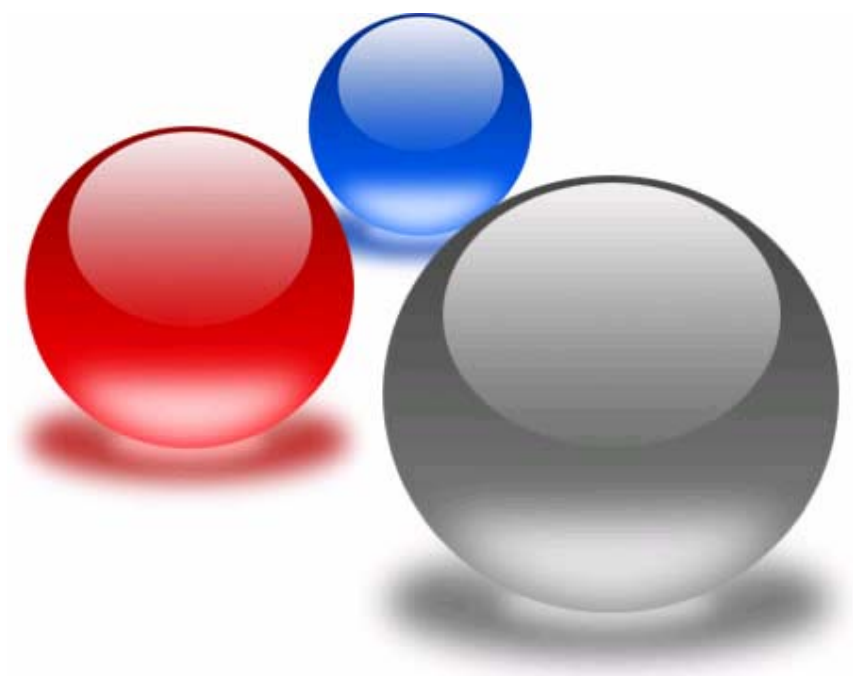


# Process Heaps and Heap Entries (chunks)

Heaps are created by the system run time library (Rtl) and are used to store dynamically created process memory.

Heap chunks or entries, are the actual storage containers for allocated memory created with malloc, alloc, HeapAlloc or other similar calls.

The heap is organized as a linked list, examine the microsoft rtlheap documentation for additional information.

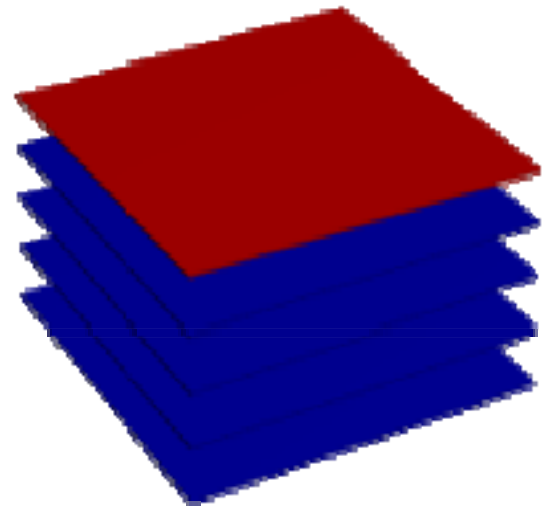




# Process Stacks

Stacks are a one-per-thread data structure, which are operated on by the processor to store local function data.

Stacks are a LIFO (Last In First Out) Data structure which are operated on typically by x86 PUSH and POP instructions. You can also reference stack data directly utilizing MOV or other instructions.



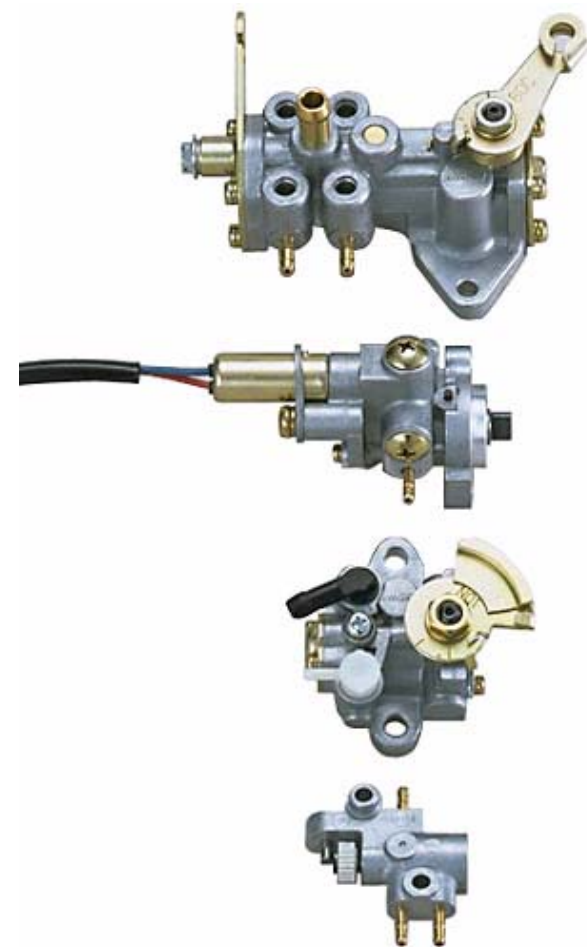


# Processes and Threads

While not technically within the scope of this presentation, the process and thread snapshot APIs provide one crucial thing that can be important while performing memory analysis.

Context.

Processes can be used to derive a thread snapshot list, and threads can be directly opened, and their context and registers, retrieved.





# Userland Example





# Exploring Kernel Memory

The APIs provided earlier are userland APIs, and cannot be used to explore anything in the kernel. To explore kernel memory different techniques are to be used.

- Page Tables
- VADs
- Physical Memory Tables
- Memory Manager Routines





# X86 Ring 0 and Ring 3

The Kernel's system process runs at a Ring 0 which essentially has access to anything on the box.

Userland operations are restricted, and operate at ring 3.

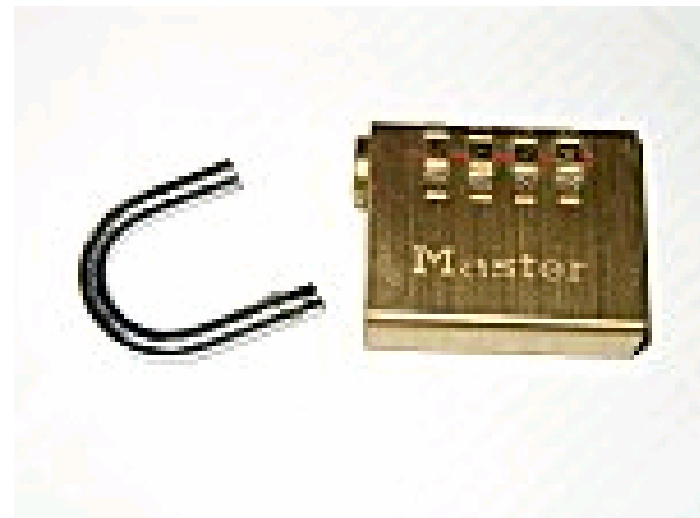




# Accessing the Kernel

Because of a device drivers necessity in terms of accessing physical memory ranges on a host, they are forced to operate in Ring 0. If a user creates a driver and installs it into a system, this driver has full privilege over the system and can perform any x86 instruction on anything anywhere anytime for any reason.

There are no real protections.

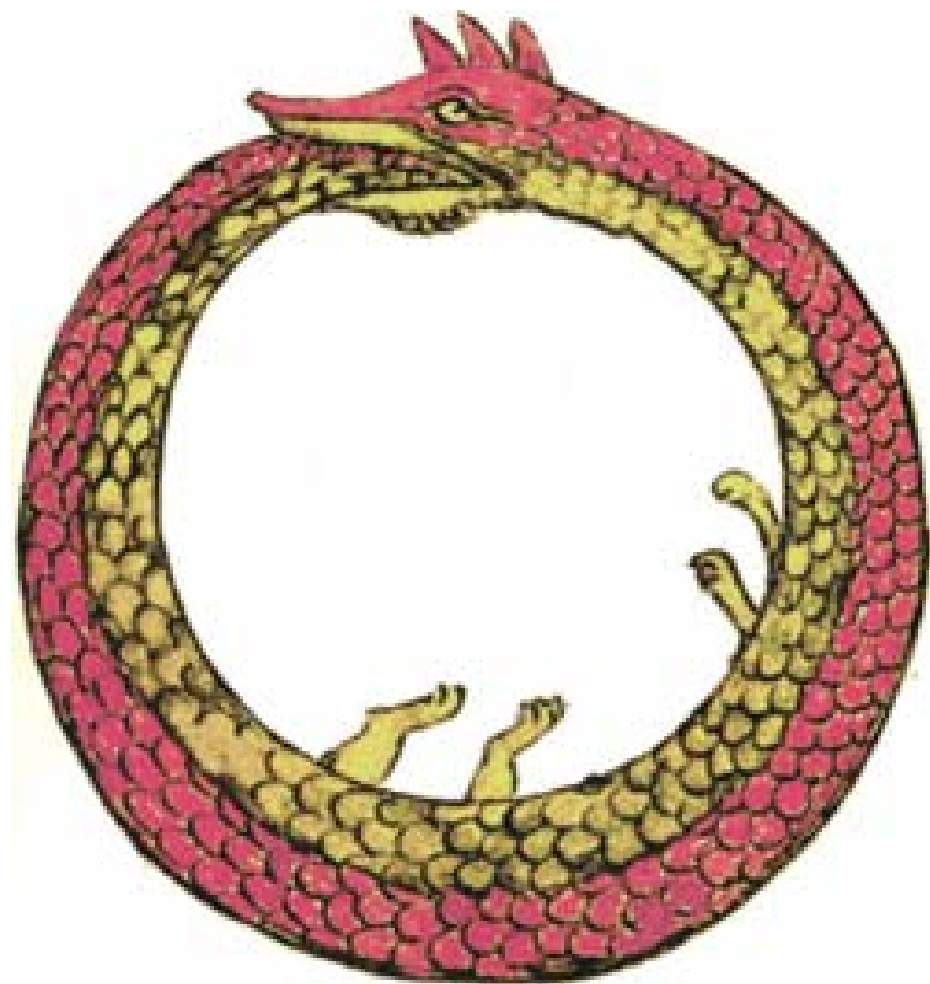




# Self Monitoring Process Code

When a driver is loaded, it literally becomes part of the System process. In order to monitor the kernel process, the driver has to be able to literally watch itself.

In order to find out ranges of addresses in the kernel, the simplest way is to navigate what is known as the Process VAD tree.

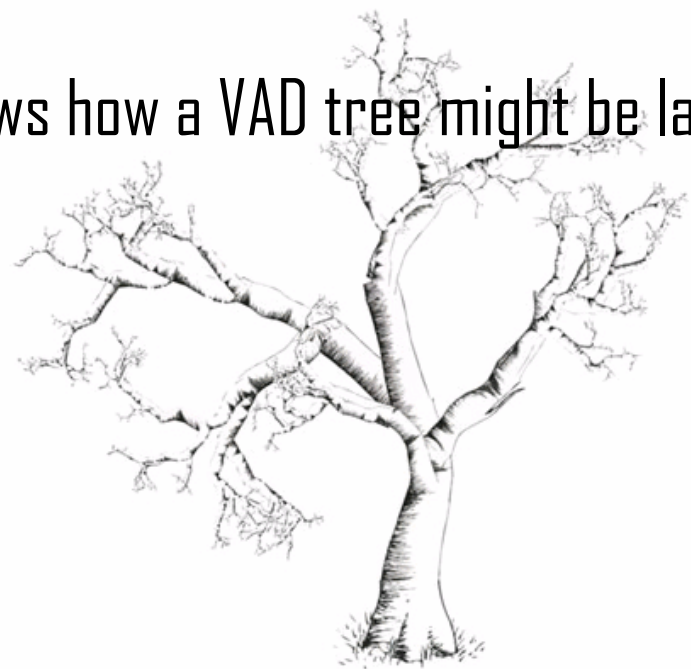




# Virtual Address Descriptors

Virtual Address Descriptors (VADS) are a very efficient, self balancing, binary tree. When utilizing VirtualQuery in userland, essentially the userland routine builds its list of process ranges based around VAD values.

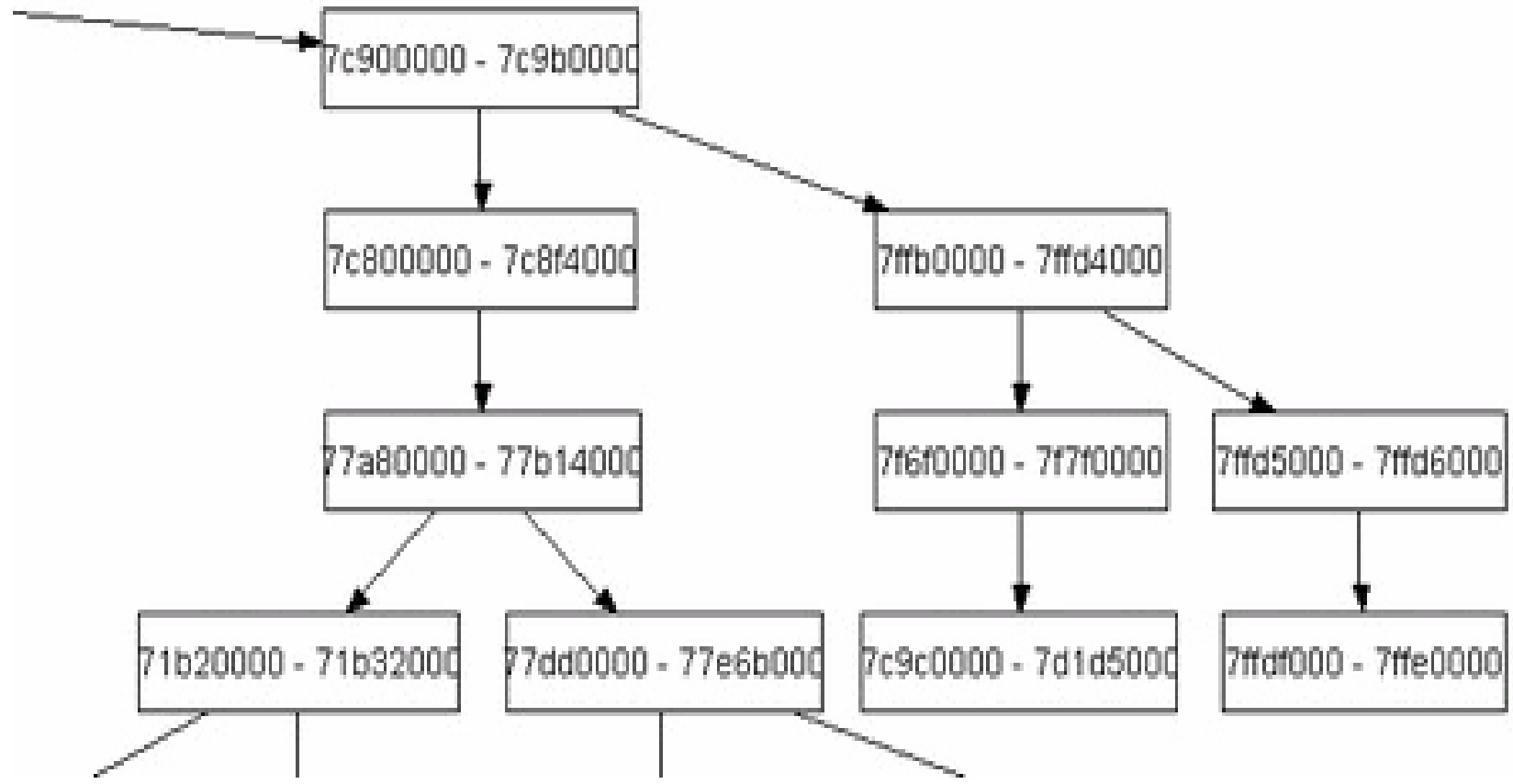
The next slide shows how a VAD tree might be laid out for a given process.







# VAD Tree Detail View





# Accessing VADs

The `_MMVAD` structure is accessible through a processes `_EPROCESS` structure, which can be accessed through the `PsGetCurrentProcess()` `ntddk` function. The function will return a pointer to the structure, which can be used then to navigate around a process.

There is a problem however, `EPROCESS` is opaque.





# Opaque Structures

An Opaque structure, is a datatype that changes from Windows build version, to windows build version, and is never intended to be build portable.

The EPROCESS and MMVAD structures are especially opaque and can only be found per build, by examining the data types exported by build debugging symbols.





# WinDBG and Symbol Servers

A simple way to extract these opaque structures is to use the "dt" command in windbg. DT refers to datatype, and can be used to explore any datatypes available in symbol packages.

Retrieve Eprocess Structure:

```
dt nt!_EPROCESS
```

Retrieve MMVAD Structure:

```
dt nt!_MMVAD
```





# Build Symbol Packages

Its additionally possible to download symbol packages directly from microsoft. Microsoft has symbols for every OS build since windows 2000, available for free.

You could theoretically download the symbol packages and use the windows dbghelp symbol api to dynamically extract these structures without having to use windbg at all.





# \_EPROCESS in Vista

```
typedef struct _EPROCESS
{
    KPROCESS Pcb;
    EX_PUSH_LOCK ProcessLock;
    LARGE_INTEGER CreateTime;
    LARGE_INTEGER ExitTime;
    EX_RUNDOWN_REF RundownProtect;
    .
    . (over 50 other fields)
    .
    MM_AVL_TABLE VadRoot; // over extended MMVAD
    ULONG Cookie;
    ALPC_PROCESS_CONTEXT AlpcContext;
} EPROCESS, *PEPROCESS;
```



# \_MMVAD in Vista

```
typedef struct _MMVAD
{
    ULONG u1;
    PMMVAD LeftChild;
    PMMVAD RightChild;
    ULONG StartingVpn;
    ULONG EndingVpn;
    ULONG u;
    EX_PUSH_LOCK PushLock;
    ULONG u5;
    ULONG u2;
    union
    {
        PSUBSECTION Subsection;
        PMSUBSECTION MappedSubsection;
    };
    PMMPTE FirstPrototypePte;
    PMMPTE LastContiguousPte;
} MMVAD, *PMMVAD;
```



# Putting it all together.

1. `PsGetCurrentProcess()` to retrieve pointer to `EPROCESS`.
2. `EPROCESS->VadRoot` to get the first `MMVAD`.
3. Walk `MMVAD` tree to enumerate viable ranges.





# Caveats

Because all drivers are loaded at ring 0, they all can duke it out if they prefer. This means if an attacker compromises a machine, and hosts a rootkit, the attacker is in essence at a level of control which can alter all sorts of kernel structures (aka. DKOM). The attacker can literally unlink nodes from the VAD tree, in order to make them invisible to the system process range lookup.

This however does not hide them from the page table. If you are doing forensic work, always use the page table if at all possible.





# More about VAD Unlinking

The concept of unlinking certain VAD nodes to hide memory from the userland is not a complicated task. All one has to do is change one 4 byte value to another 4 byte value.

By changing a node pointer to null, all nodes in that branch become hidden by the Windows VAD walking routines. However, because they are mapped into memory by the OS, the page table entries remain the same.

By this logic, the virtual memory can remain accessible to the system process, but not visible in the VAD.



# Accessing Physical Memory

Physical memory in Win32 is a device, it's a device and you can find it in the `\\device\physicalmemory` path. This device was accessible from userland in windows versions prior to server 2k3.

Mark Russonivich concocted some source code that utilized kernel32 exports in order to map sections of the physical memory device, into process virtual memory.



# Open and Map

The kernel function `ZwOpenFile` can be used to literally open the physical memory device as if it were a file.

`ZwMapViewOfSection`, takes a section of an open file, and maps it into a processes virtual memory space.

By mapping known physical address ranges, and copying them from memory,



# Kernel Example





# Conclusion

By studying memory layouts, you only serve to empower yourself as an attacker, defender, and researcher.

By being able to lower it to a no-nonsense view, you can reliably find things that were intended to be hidden, and hide things that are sensitive in pockets of memory that you control.

Seriously, its fun stuff.



