Malware Software Armoring Circumvention

Shmoocon 2008

Offensive Computing, LLC

Danny Quist
Valsmith
dquist@offensivecomputing.net
valsmith@offensivecomputing.net

Danny Quist

- Offensive Computing, Cofounder
- Ph.D. Student at New Mexico Tech
- Reverse Engineer
- Exploit Development
- cDc/NSF

Valsmith

- Offensive Computing, Cofounder
- Malware Analyst/Reverse Engineer
- Metasploit Contributor
- Penetration Tester/Exploit developer
- cDc/NSF

Offensive Computing, LLC

- Community Contributions
  - Free access to malware samples
  - Largest open malware site on the Internet
  - ~350k hits per month
- *"It's like an anti-virus company, but without that fake "We're better than you" attitude." - Dave Aitel*
- Business Services

Overview of Talk

- Problem Discussion
- Software Armoring Techniques
- Covert Debugging Requirements
- Dynamic Instrumentation for Debugging
- OS Pagefault Assisted Covert Debugging
- Application – Generic Autounpacking
- Results

What are the problems?

Malware analysis necessary for defense

- Creating signatures
- Understanding attacks (targeted/untargeted)
- Data mining trends and unknown threats
- Determining phylogeny of variants

What are the problems?

Malware wants to stop us from analyzing and understanding it

- Packing hinders our analysis
- Anti-analysis techniques
- Obfuscation hinders automation
- Automation is key to rapid analysis

What is the problem?

- Huge number of malware samples
  - Example: We have almost 300,000
  - More hitting victims every day
- Analyst time is expensive
  - Individual samples can take hours to analyze
- We must automate the process to keep up
- Packers degrade automation
- We need to automatically decrypt malware!

Previous Work

- Shadow Walker
  - Rootkit Memory Hiding
  - Jamie Butler, Sherrie Sparks
- PaX
  - Linux buffer-overflow prevention
- OllyBonE
  - Break on Execute for OllyDbg
  - Joe Stewart
- Memalyze
  - Tracing memory access
  - Skape
- PolyUnpack – Paul Royal et. al @ Georgia Tech
- Halvar's VxClass auto-unpacker

Gaps

- The available solutions are detectable
- Not all are fully automatable
- Smaller percentage of success
- Some rely on signature based techniques
- In some cases slow
- No one solution addresses all these problems

What we will show you

- Techniques that are a crucial step in the process of automating Malware decryption
- Example code that may help you in implementing your own automated decryption tools
- Ideas on what further steps are needed to solve the malware analysis automation problem

What we will not show

- This is research code, not production
- Proof-of-concept
  - a short and/or incomplete realization (or synopsis) of a certain method or idea(s) to demonstrate its feasibility, or a demonstration in principle, whose purpose is to verify that some concept or theory is probably capable of exploitation in a useful manner (wikipedia ftw)

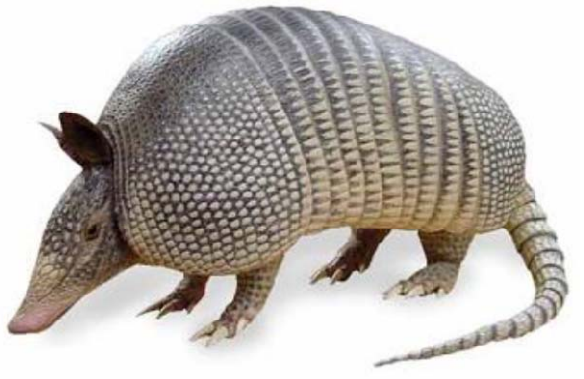
Implications

- Analysis automation now within our reach
- Obfuscation no longer a major obstacle
- Ability to process 1000’s of files rapidly
- Malware authors will have to step it up
  - Raising the bar
- Advanced tools/products can be developed

Software Armoring Overview

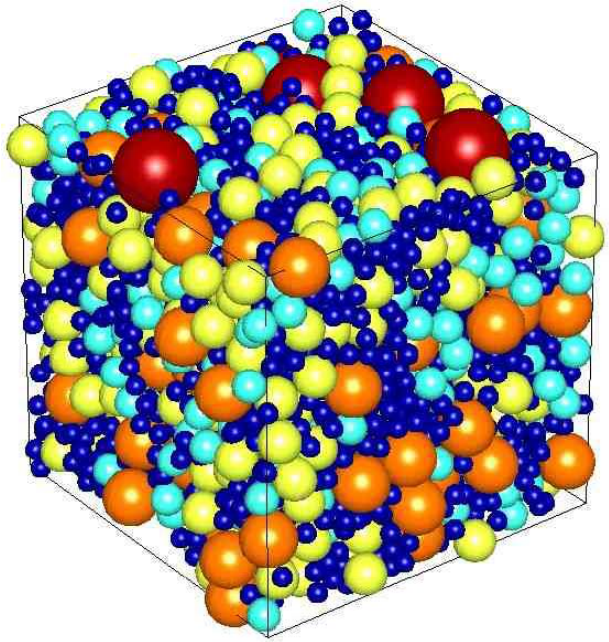
Software Armoring

- Packing/Encryption
- SEH Tricks
- VM Detection
- Debugger Detection
- Shifting Decode Frame

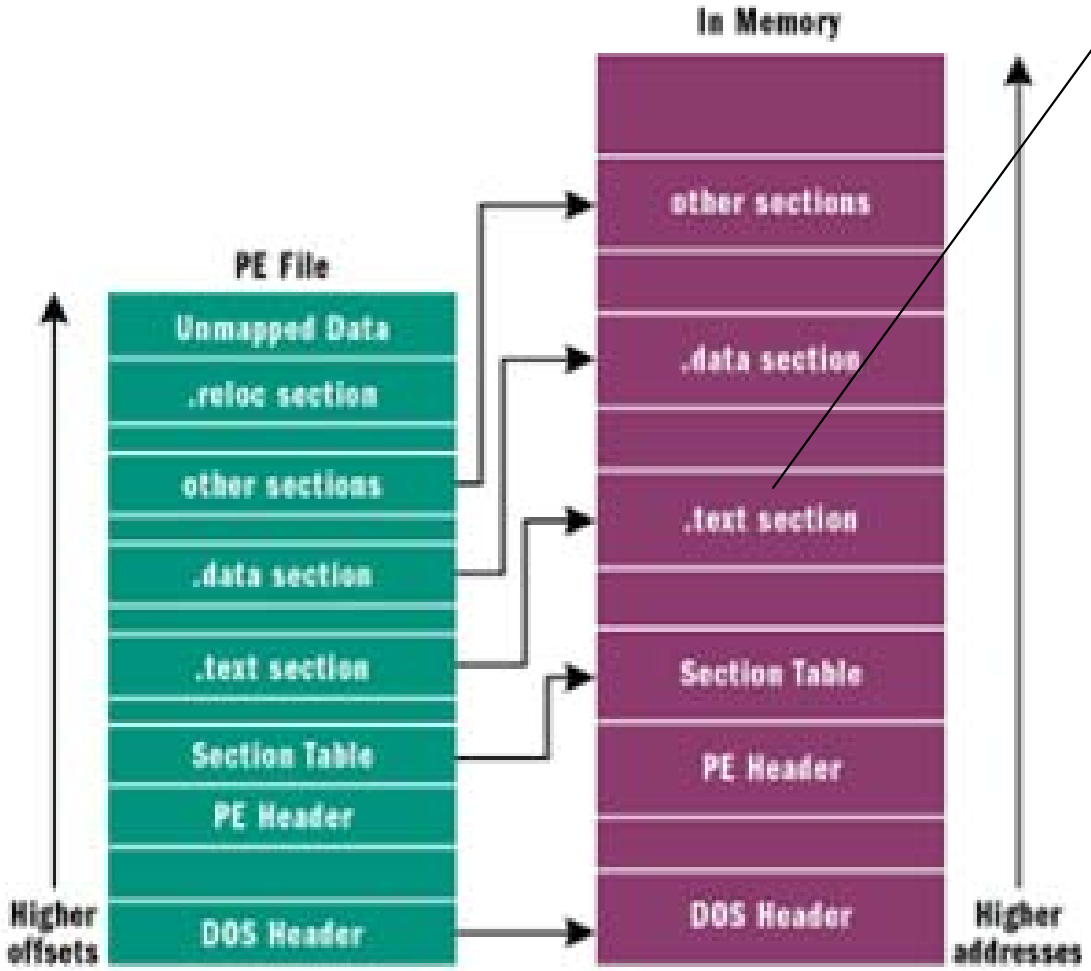


Packing/Encryption

- Self-modifying Runtime Code
  - Small Decoder Stub
  - Decompresses the main executable
  - Restores imports
- Play Tricks with Portable Executables
  - Hide the Imports
  - Obscure relocations
  - Encrypt/compress the executable

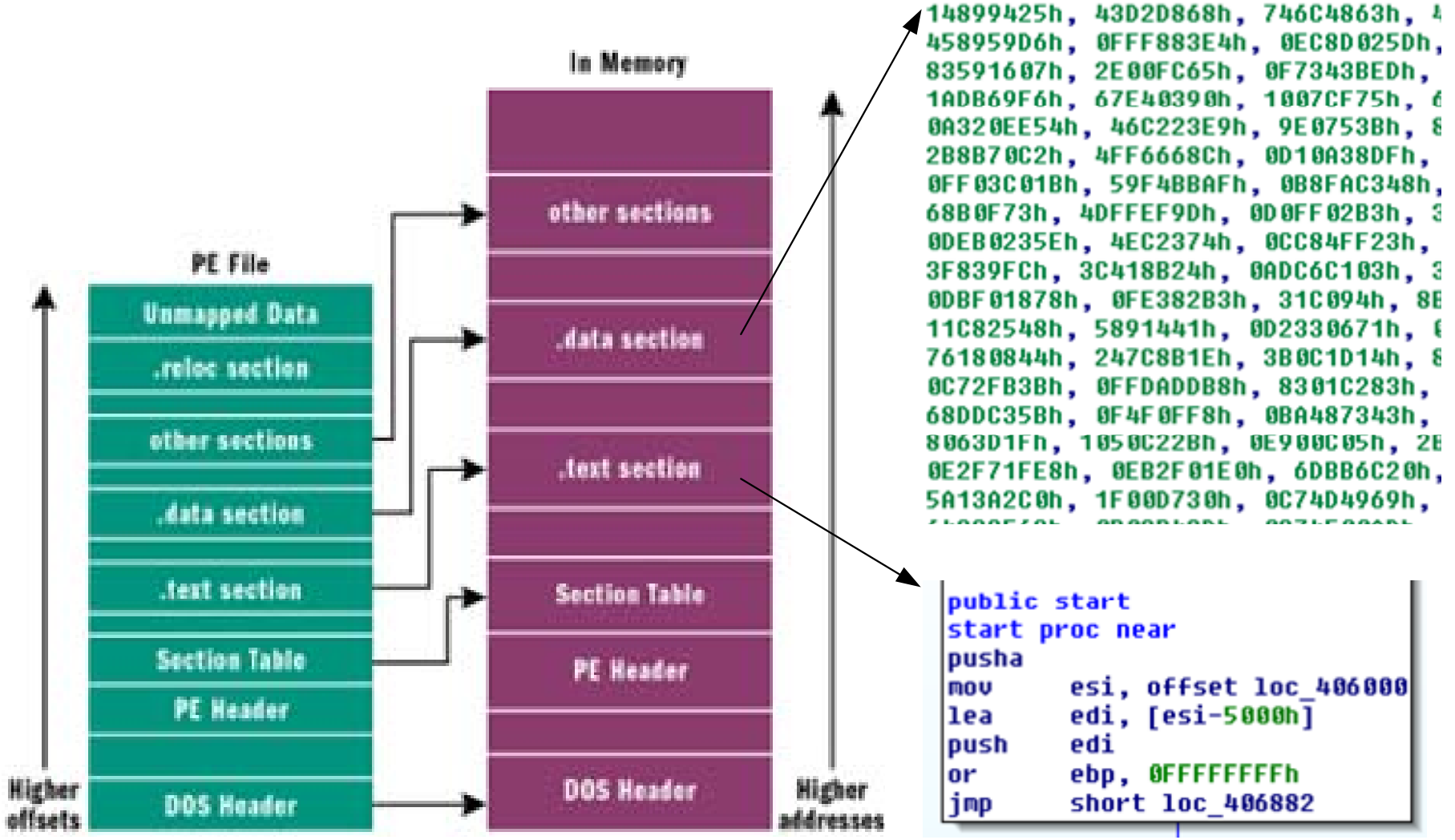


Normal PE File




```
push    ebp
mov     ebp, esp
sub     esp, 1Ch            ; lpMsg
call    ds:__imp__GetCommandLineW@0 ;
push    [ebp+nCmdShow]  ; nCmdShow
push    eax                 ; int
push    [ebp+hPrevInstance] ; int
push    [ebp+hInstance] ; hInstance
call    _FSolInit@16    ; FSolInit(x,
test    eax, eax
jz      short locret_1001F13
push    esi
mov     esi, ds:__imp__GetMessageW@16
push    edi
mov     [ebp+Msg.wParam], 1
xor     edi, edi
jmp     short loc_1001EFE
```

Packed PE File



Virtual Machine Detection

- Single instruction detection
  - SLDT, SGDT, SIDT
  - See: Redpill, Scoopy-Doo, OCVmdetect
- Instructions for Privileged/Unprivileged CPU mode
  - VMs try to be efficient, some instructions insecure
  - Do not fully emulate x86 bug for bug

Debugger Detection

- Windows API
  - IsDebuggerPresent() API call
  - Checks PEB for magic bit
  - Bit toggling works

- Timing Attacks
  - Issue RDTSC instruction, compare to known values
  - Amazingly effective

Debugger Detection (cont.)

- Breakpoint Detection
  - Int3 (0xCC) Instruction Scanning
  - Checksumming of executable
- Hardware Debugging Detection
  - Check CPU Flags for debug signatures
- SoftICE Detection
  - Modification of Int3 Scanning
  - Checksumming
  - BoundsChecker and other signatures

SEH Tricks

- Structured Exception Handler
- Used to handle errors in running code
- Malware will overload this function to unpack code
- Debugger thinks SEH exceptions are for it
- Debugger dies
  - Divide by 0

Shifting Decode Frames

- Execution is split at the basic block level
- Block is decoded, executed, and then encoded again
- Hard to defeat!
- Implemented in Patchguard for Vista 64 and Windows Server 2003 64-bit

Use Hardware for Analysis

- Nearly as capable as VM solutions
- Just as cheap*
- Almost impossible to detect
- Safe solutions available
- Real hardware control possible
  - As will be demonstrated

* Assuming software licensing costs

Cost Comparison

Hardware

- Cheapest Dell $349
  - Brand new
  - Cheaper elsewhere
  - XP License included*
  - Deepfreeze $45

Total cost: $394

Software

- VMWare - $189
  - XP $278.99 *
  - Other solutions cheaper

Total cost: $467.99

* Assuming relevant US piracy laws followed

Replacing Vmware Snapshots

- Faronics Deepfreeze
  - Implements copy on write protection
  - Analogous to VMWare snapshot
  - Kernel driver
  - Not perfect, and hackable (like anything)
  - www.faronics.com
- Disk Image Safe Installation
  - dd your drive in case Deepfreeze fails
  - Last resort restoration

Other Good Tools

- Firewire kernel debugger
  - WinDBG (thanks MSFT)
- Syser Debugger
  - www.sysersoft.com
  - SoftICE replacement
- Debuggers detectable (telock) so be careful

Software Armoring Achilles Heel

If it executes,
it can be unpacked.

[http://www.security-assessment.com/files/presentations/Ruxcon_2006_-_Unpacking_Virus,_Trojans_and_Worms.pdf]

Manual Unpacking

Unpacking

- How an Unpacker Works:
  - Writes to an area of memory (decode)
  - Memory is read from (execute)
  - More writes to memory (optional re-encoding)
- CPU Only Executes Machine Code
- This process can be monitored
- Unpacking is directly related to timing
  - At some point, it ***must*** be unpacked

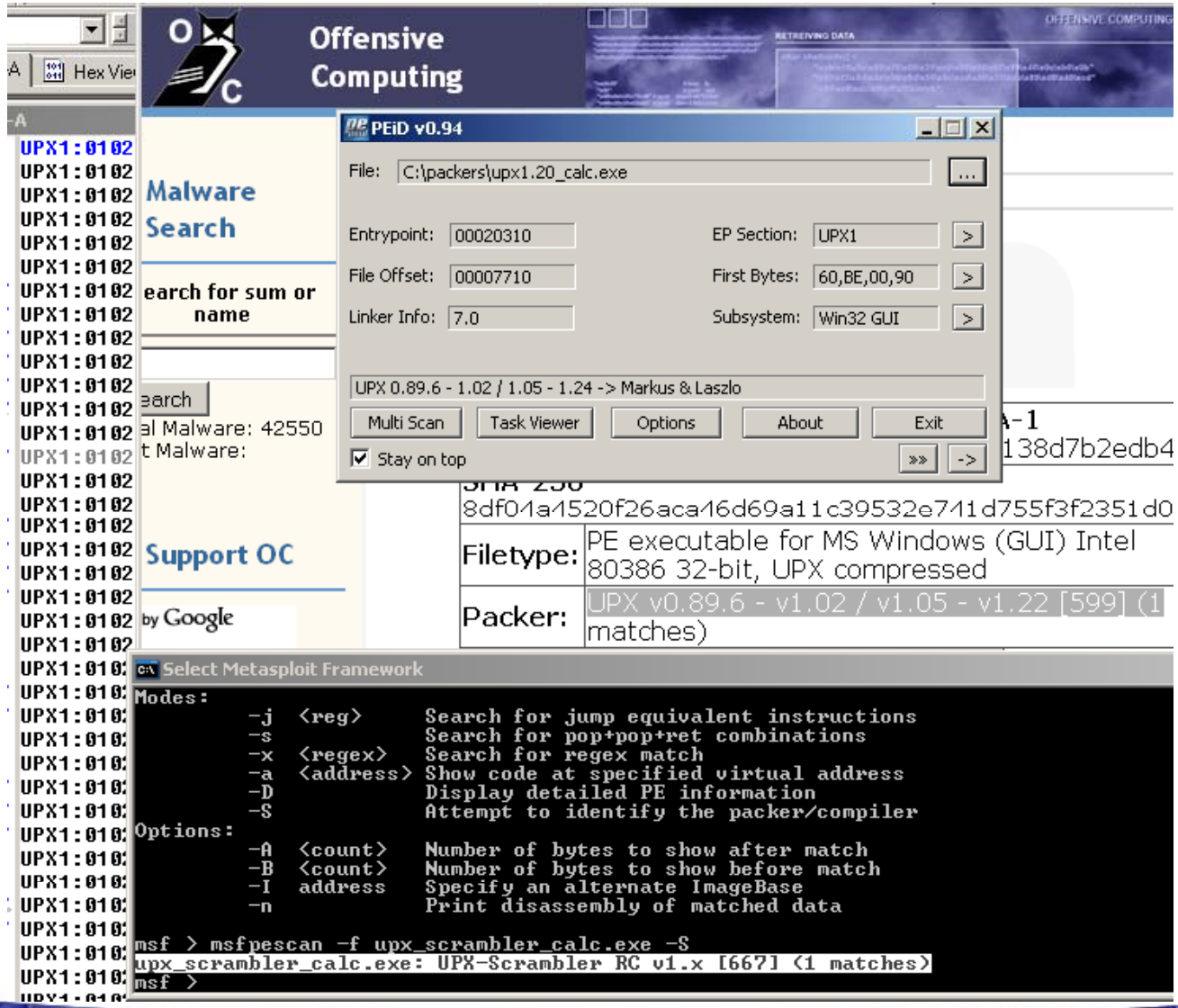
Manual Unpacking Process

- Consists of several stages
  - Identify Packer Type
  - Find OEP or get process to unpacked state in memory
  - Dump process memory to file
  - Fixup file / rebuild Import Address Table (IAT)
  - Ensure file can now be analyzed

Manual Unpacking Process

- Several methods to identify packer type
  - PEiD
  - Msfpescan
  - PEFile from Ero Carrera
    - OC patched to harden against ~275k Malware
  - Manually look at section names
  - Other packer scanners like
    - Protection-id
    - Pe-scan

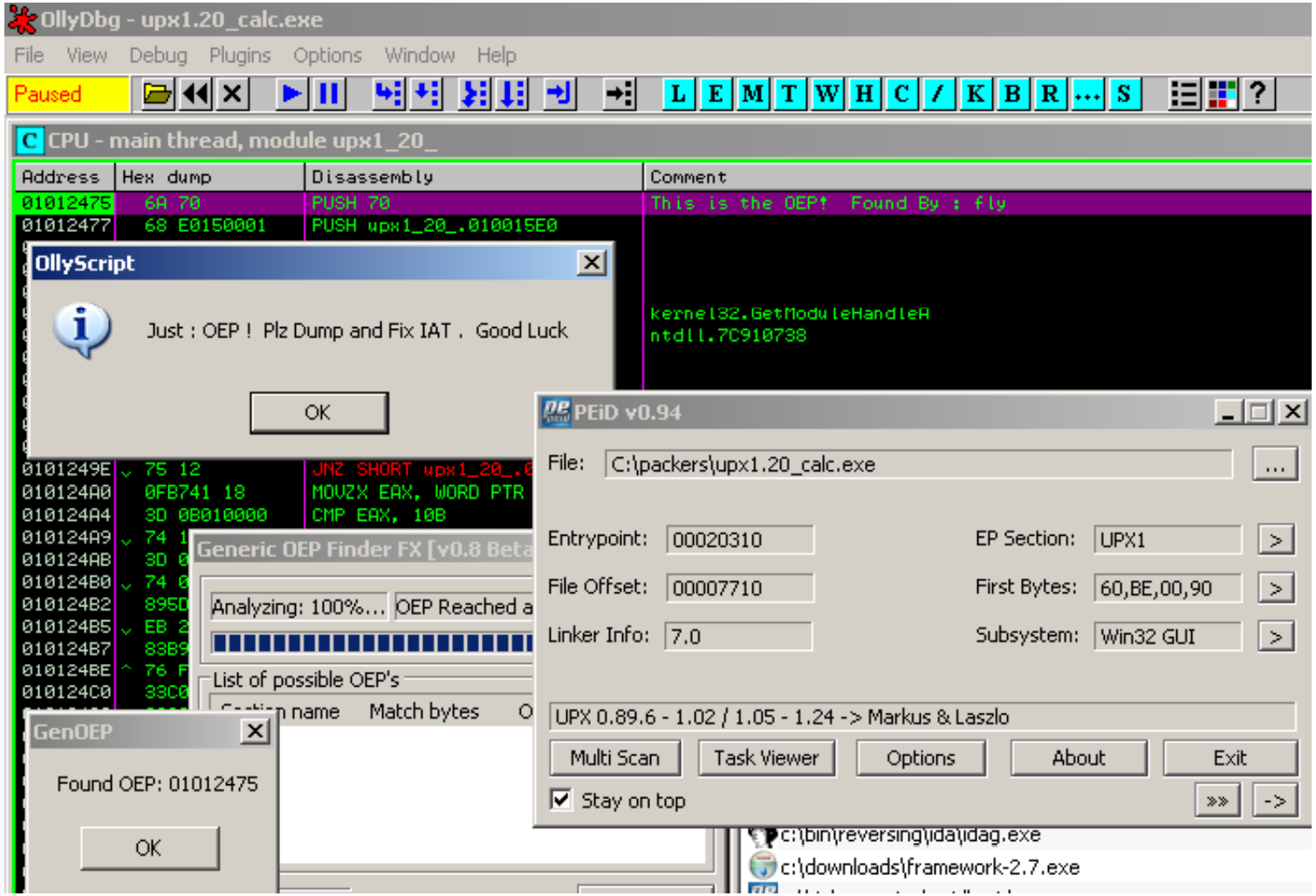
Manual Unpacking Process



Manual Unpacking Process

- Methods to find OEP / unpacked memory
  - OllyScripts
    - http://www.tuts4you.com
    - http://www.openrce.org
  - OEP finder tools
    - OEP finders for specific packers
    - OEP Finder (very limited)
    - PE Tools / LordPe
    - PEiD generic OEP finder

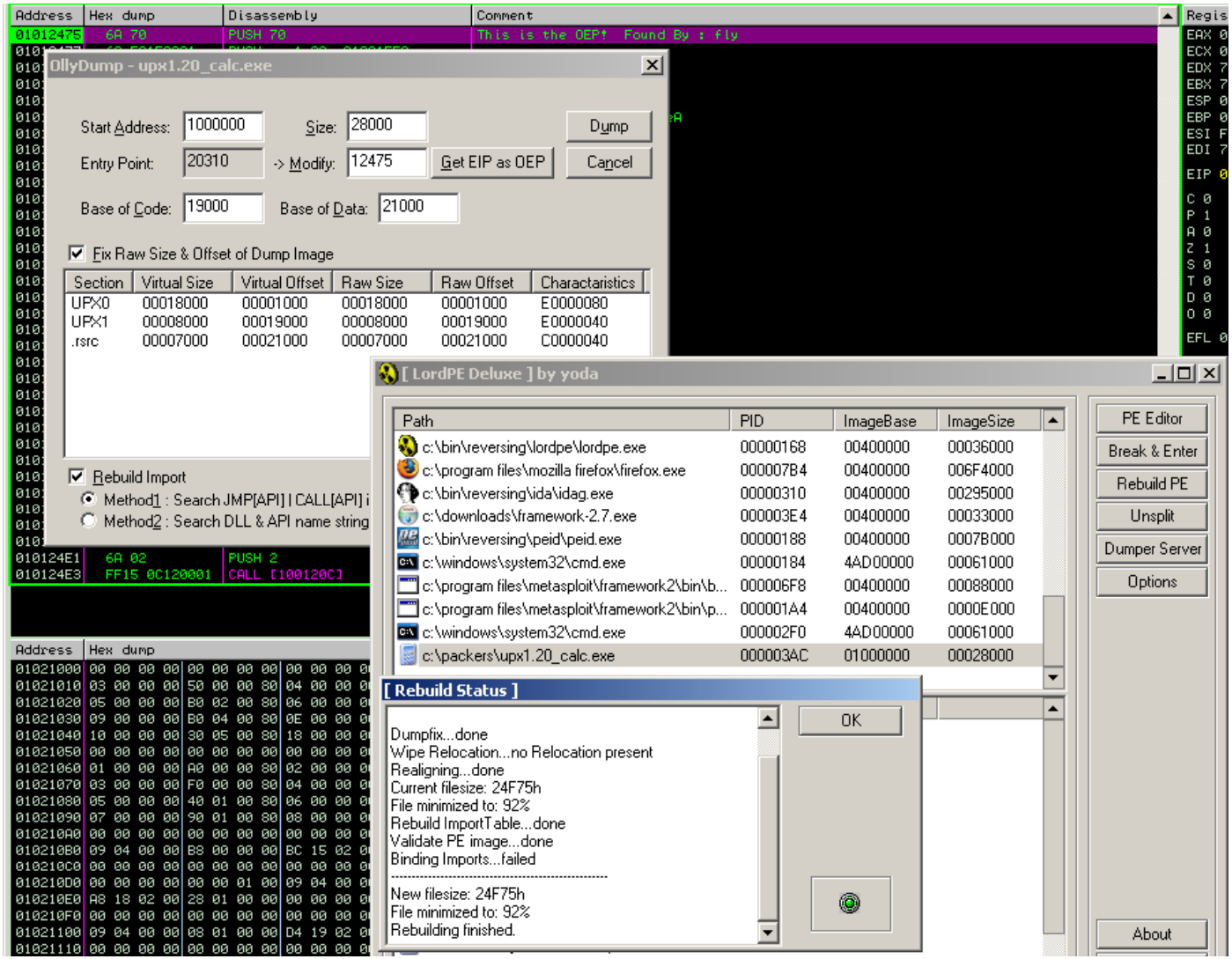
Manual Unpacking Process



Manual Unpacking Process

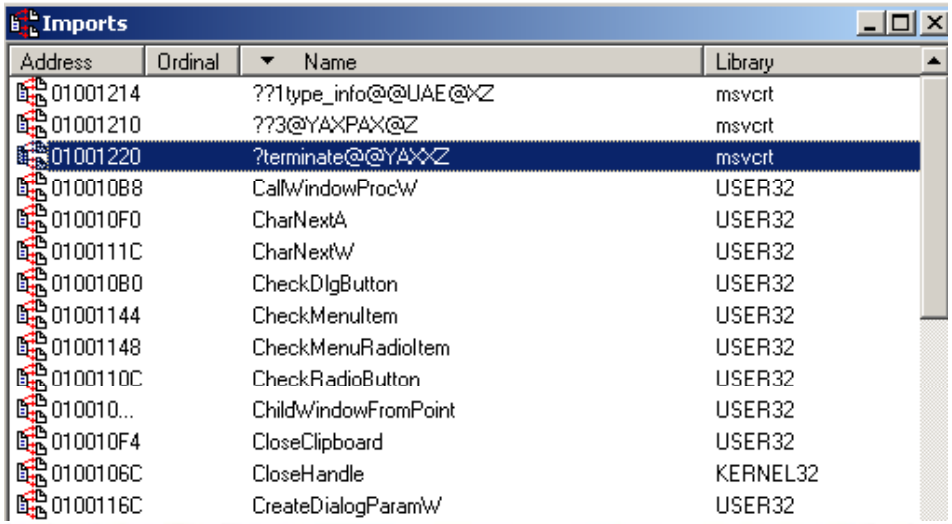
- Dump process memory to file
  - OllyDump
  - LordPE
  - Custom tools
- Example:
  - OpenProcess()
  - ReadProcessMemory()
  - CreateFile()
  - WriteFile()

Manual Unpacking Process

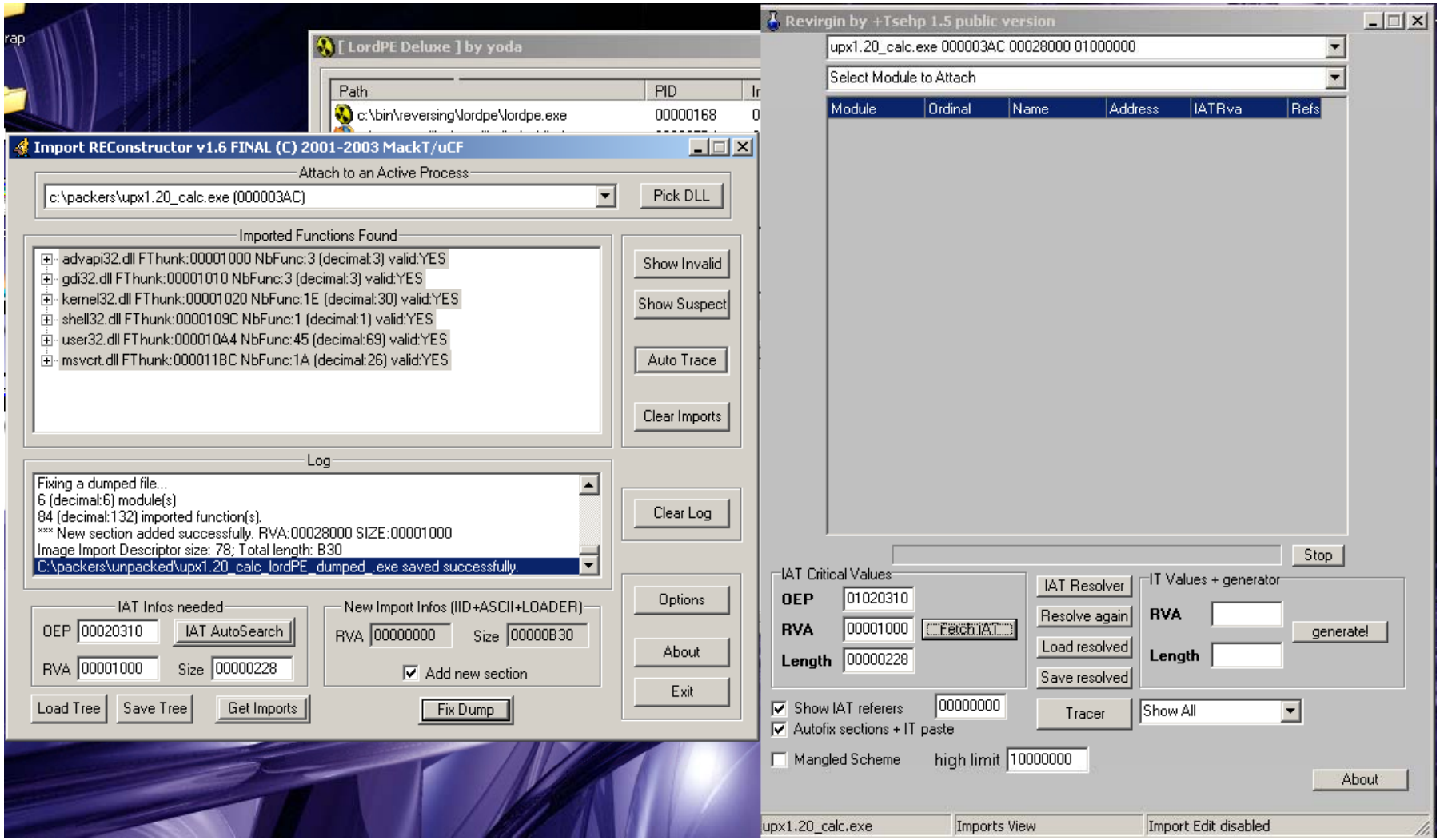


Manual Unpacking Process

- Fixup file / rebuild Import Address Table (IAT)
  - ImportRec probably best tool
  - Revirgin by +Tsehp
  - Manually with a hex editor (tedious)
- IAT contains list of functions imported
  - Very useful for understanding capabilities



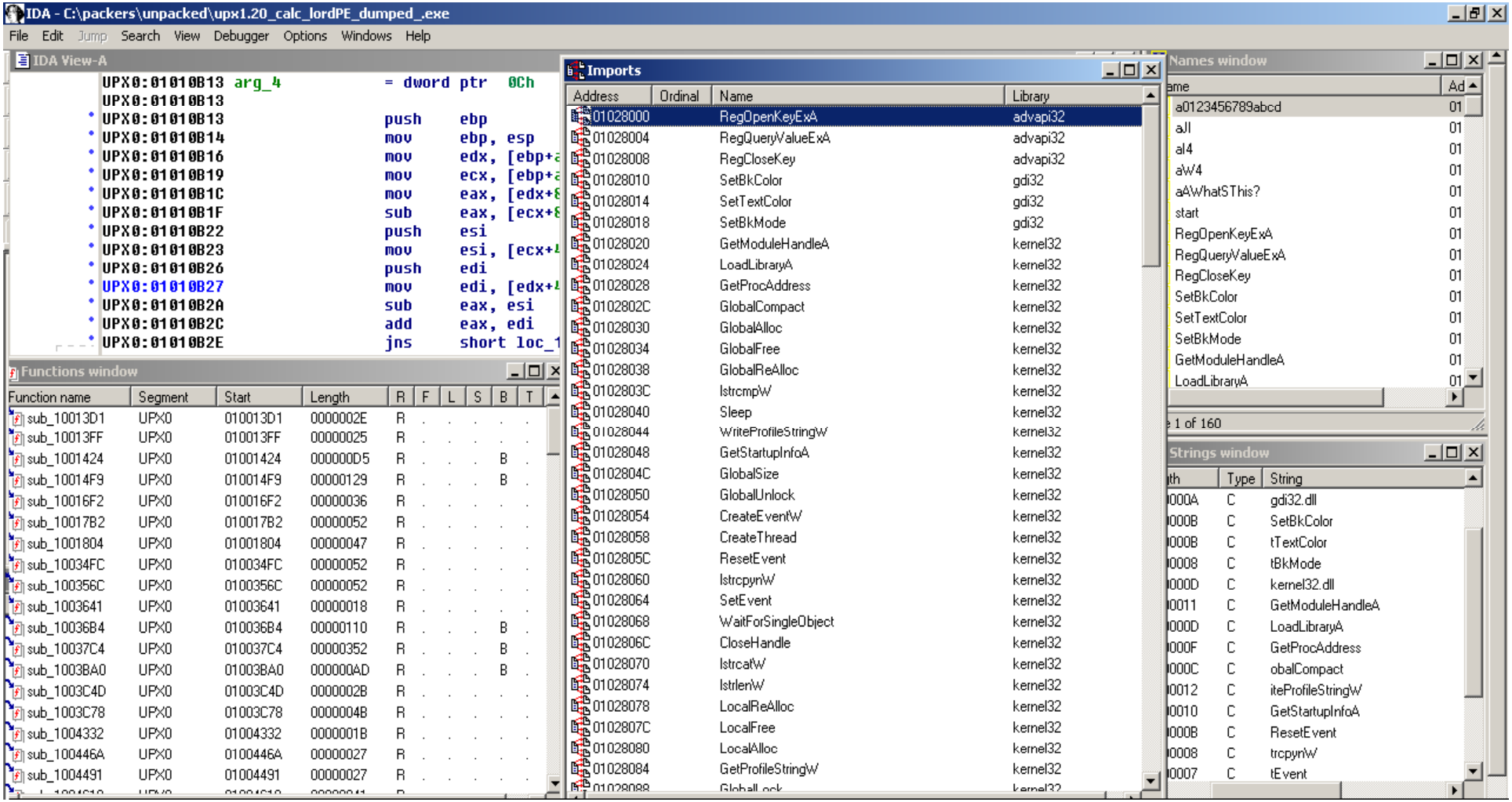
Manual Unpacking Process



Manual Unpacking Process

- Ensure file can now be analyzed
- Clean disassembly should be available
- IAT should be visible
- Functions should be found
- Strings clear and useful
- Manual unpacking process can be tedious
- Hardest part is generally finding the OEP

Manual Unpacking Process



Manual Unpacking Process

- Show Manual Unpacking Movie

So What?

- These are all variations on a theme
- There should be a generic way to debug
- Need to modify at a fundamental level
- Solution should be:
  - Generic – Work across set of executables
  - Efficient – Good performance for non-debug
  - Undetectable (as much as possible)
  - Extensible – Automation is the key

Unpacking: The Algorithm

- Track written memory
- If that memory is executed, it's unpacked
- Must monitor:
  - Memory writes
  - Memory Executions
- Automate the process

Dynamic Instrumentation

Dynamic Instrumentation

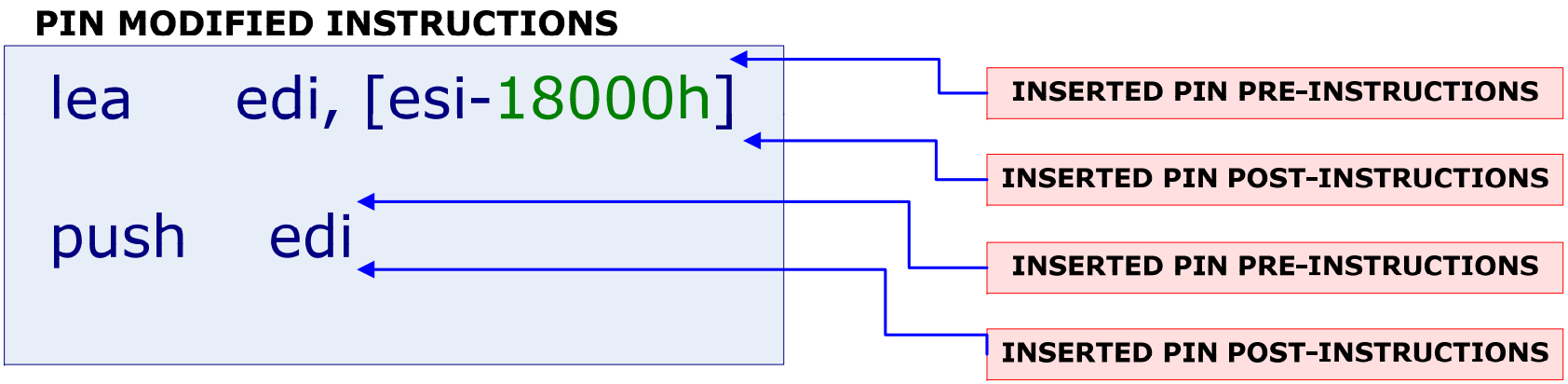
- Allows a running process to be monitored
- Intel PIN
  - Uses Just-In-Time compiler to insert analysis code
  - Retains consistency of executable
  - Pintools – Use API to analyze code
  - Good control of execution
    - Instruction
    - Memory access
    - Basic block
  - Process Attach / Detach

Dynamic Instrumentation

NORMAL INSTRUCTIONS

```
pusha
mov     esi, offset dword_1019000
lea     edi, [esi-18000h]
push    edi
or      ebp, 0FFFFFFFFh
jmp     short loc_1020332
```

Dynamic Instrumentation



Dynamic Instrumentation

PIN MODIFED INSTRUCTIONS

```
pusha
        PIN INSTRUCTIONS
mov     esi, offset dword_1019000
        PIN INSTRUCTIONS
lea     edi, [esi-18000h]
        PIN INSTRUCTIONS
push    edi
        PIN INSTRUCTIONS
or      ebp, 0FFFFFFFFh
        PIN INSTRUCTIONS
jmp     short loc_1020332
```

Implementation

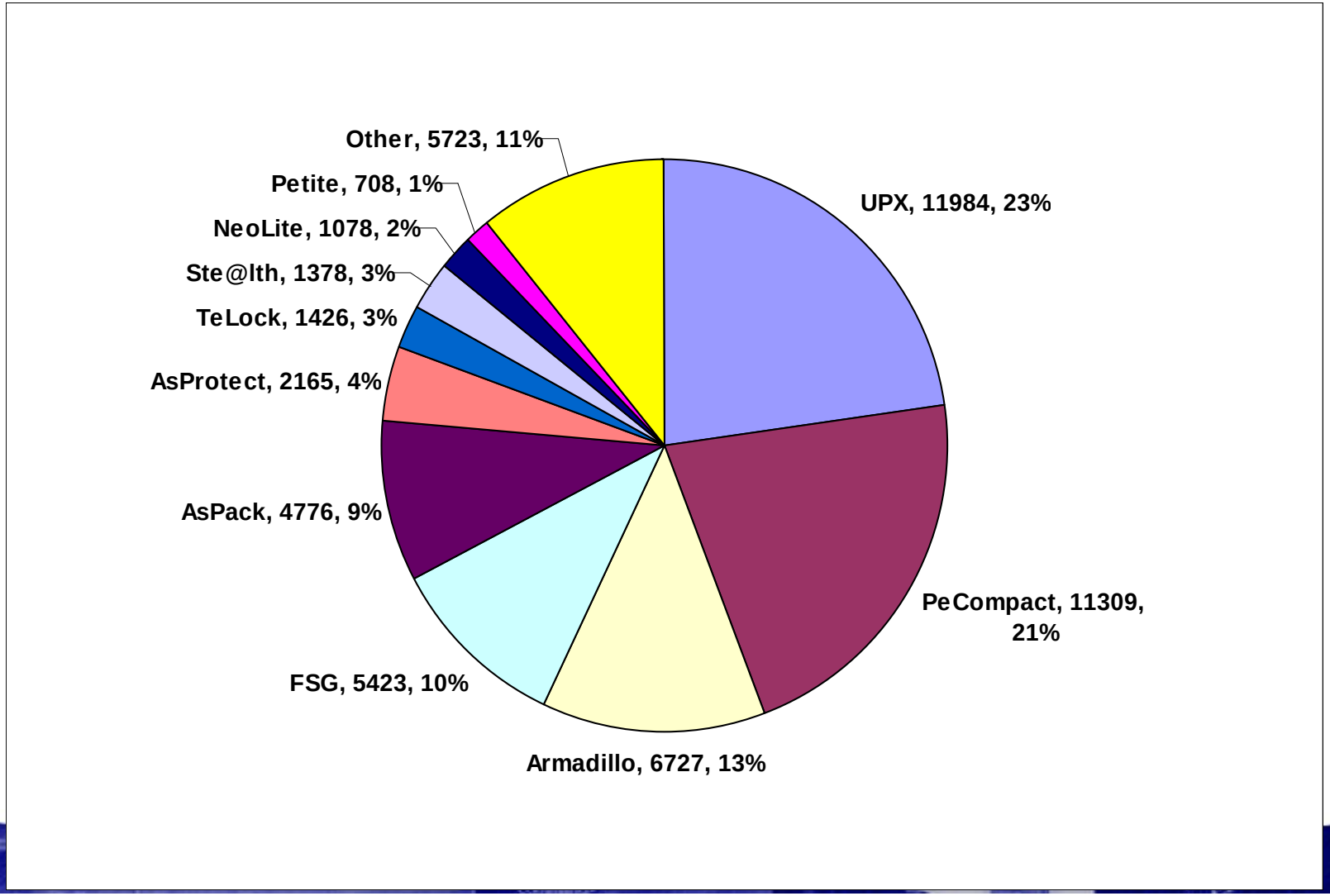
- Use PIN hooks for
  - Memory Writes
  - Executes
- Track writes in hash table
- If execution occurs on written data, dump

Results

- Successful against:
  - Most commonly used packers
  - Packers that don't self verify
  - ~70% of packed malware in OC collection

Dynamic Instrumentation - Packers

- 153701 Samples Scanned / 54123 Detected Packers



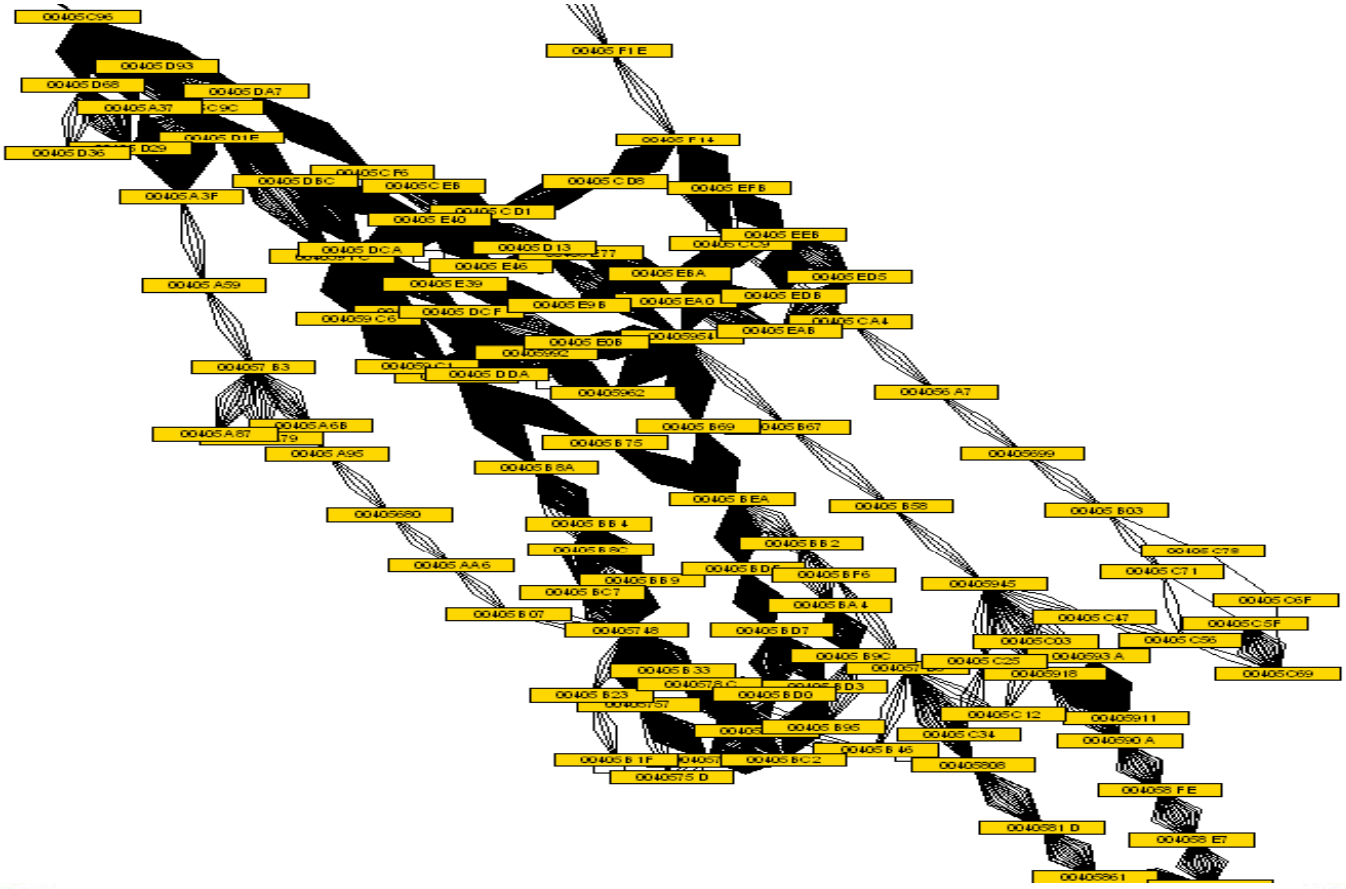
Dynamic Instrumentation

- Instruction tracing for the following packers
  - Aspack
  - FSG
  - PECompact
  - UPX
- Created Simple Hello World Application
- Graphed results with Oreas GDE

Aspack 2.12

Results

- Unpacking loop is easy to find



Dynamic Instrumentation Results

- Generic Algorithm Described Previously works well
- All addresses verified by manual unpacking
- Addresses display clustering, which must be taken into account
- Attach / Detach is effective for taking memory snapshots of an executable

Dynamic Instrumentation Caveats

- Detectable
  - Memory checksums
  - Signature scanning
- Difficult to use (sorry)
- Extend this to work generically, non-detectably
- Slow – ~1,000 times slower than native
  - Other methods/tools can be even slower
- Need faster implementation

Towards a Solution

- Core operating system component that:
  - Monitors all memory
  - Intercepts memory accesses
  - Fast Interception and Logging
  - Fundamental part of OS

Overloading the Memory Management Unit

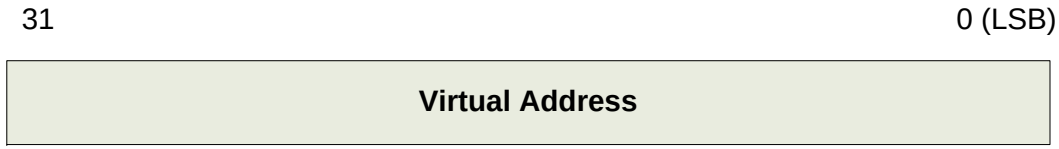
or

OS 101
How Virtual memory Works

Intel Memory Management

- Each process has its own memory
- Memory must be translate from Virtual to Physical Address
- Non-PAE Mode 32bit Processors use 2 page indexes and a byte index
- Each process has its own Page Directory

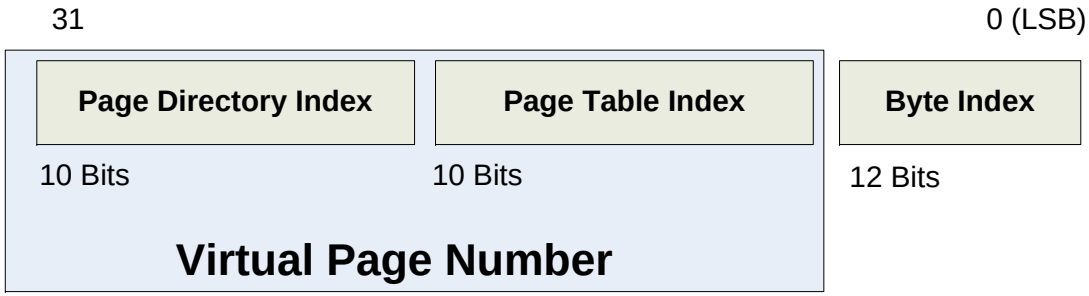
Example Memory Translation



CPU References Virtual Memory Address

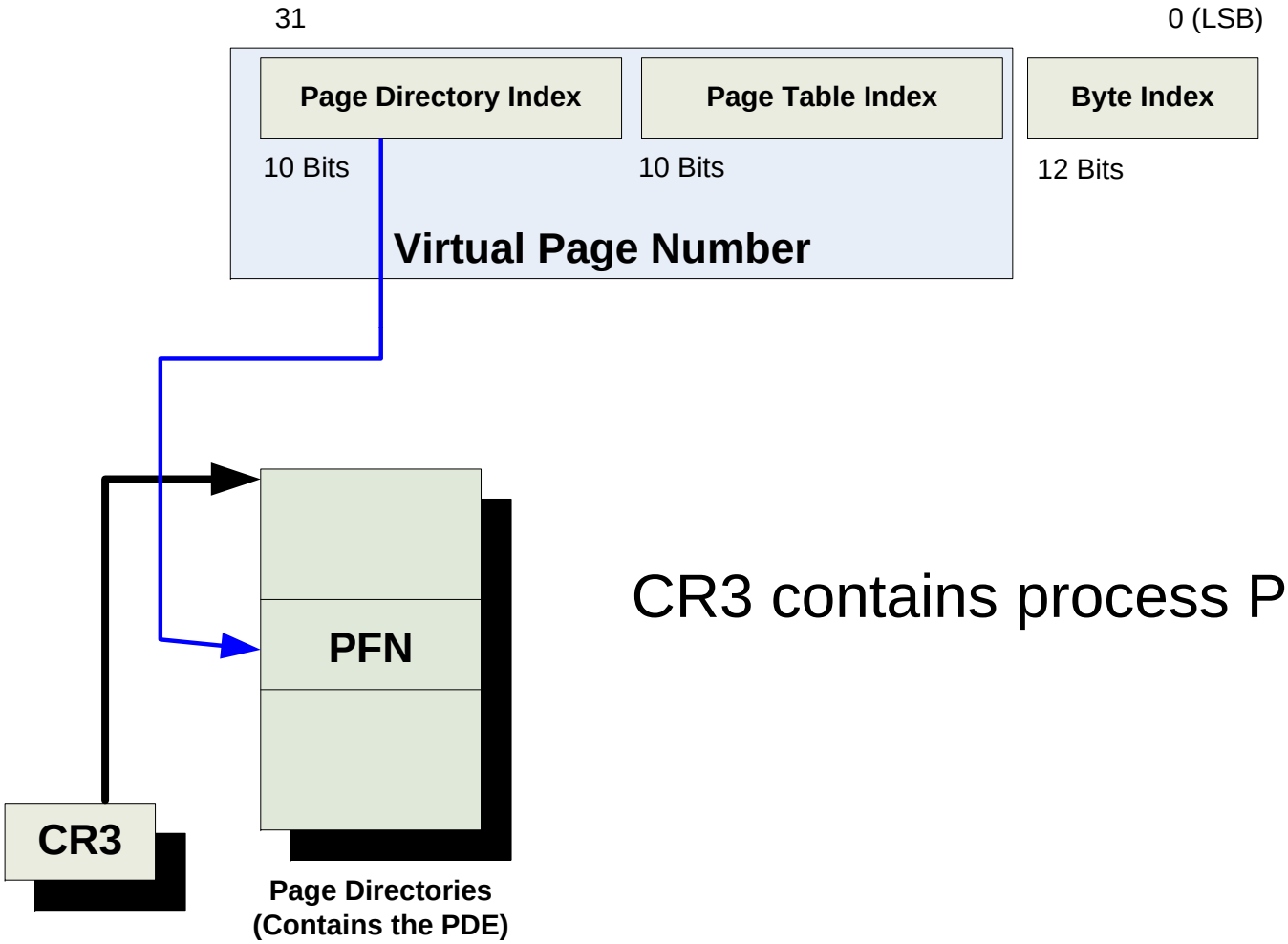
[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

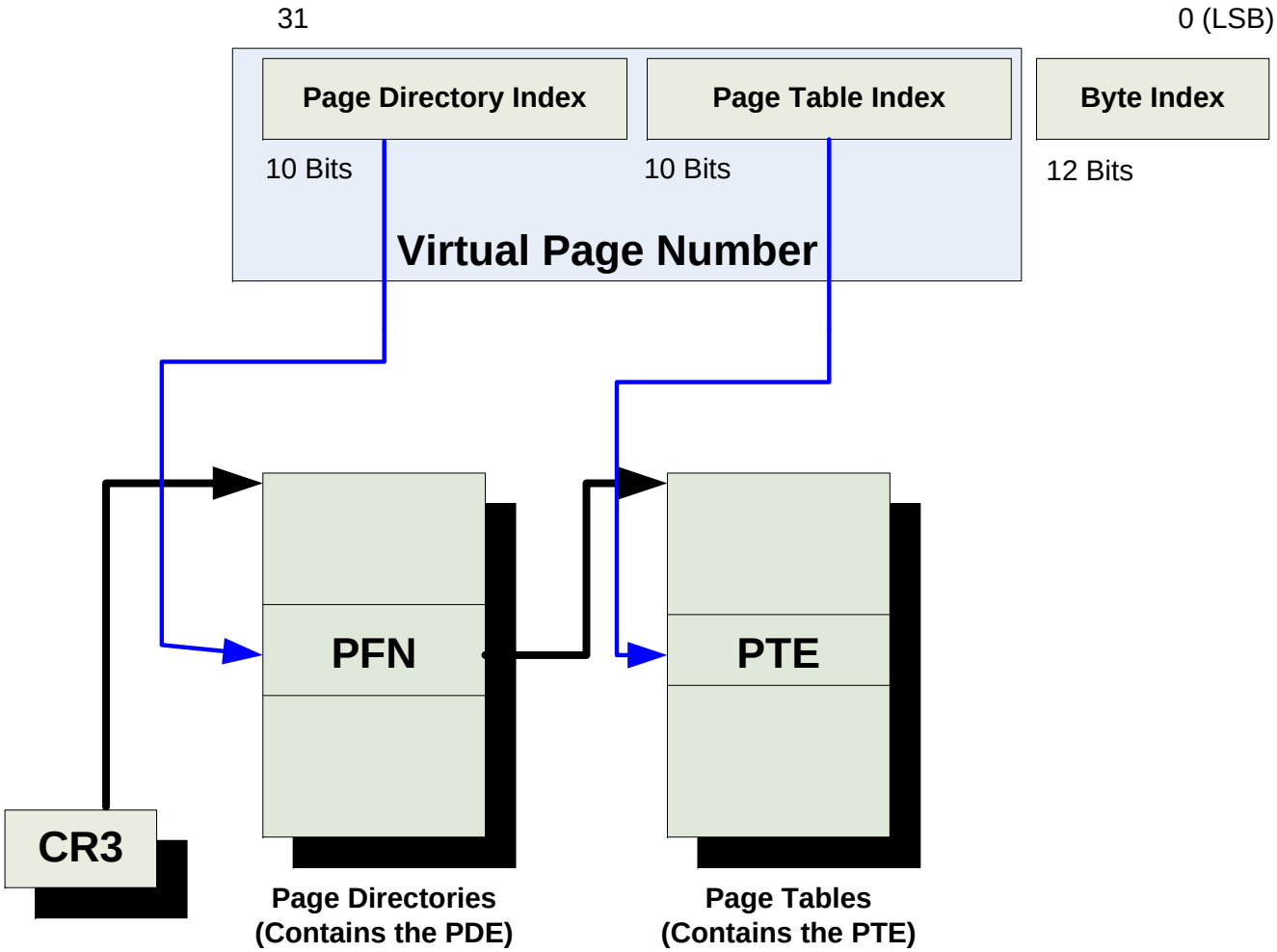
Example Memory Translation



CR3 contains process Page Directories

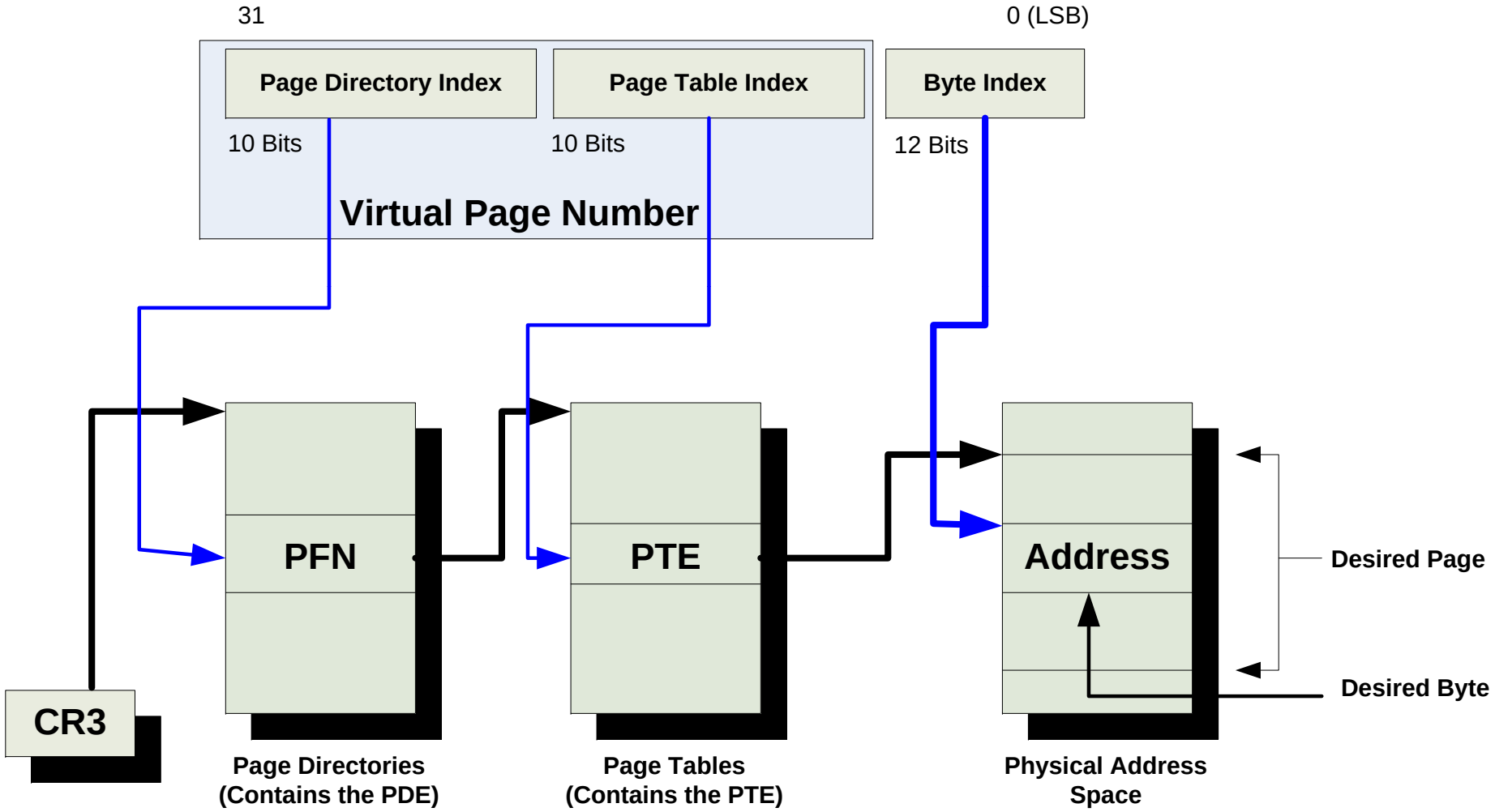
[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

MMU Data Structures

- Page Directory Entry is hardware defined
  - Contains permissions, present bit, etc.
- Page Table Entry also hardware defined
  - Permissions (Ring0 vs. all others)
  - Present bit (paged to disk or not)
  - “User” defined bits (for OS)

Virtual Address Translation

- Translation Lookaside Buffer (TLB) is major source of optimization
- Hardware resolves as much as possible
- Invokes page fault handler when
  - Page is not loaded in RAM
  - Incorrect privileges
  - Loaded, but mapped with demand paging
  - Address is not legal (out-of-range)
- All indicated by special fields

Intel TLB Implementation

- Two TLBs maintained
  - Data **DTLB**
  - Instructions **ITLB**
- ITLB more optimized than DTLB
  - Less lookups for instructions == faster code
  - DTLB accessed less

Intel TLB Population

- Data TLB
  - Address is cached upon lookup

```
mov eax, dword ptr [eax]
```

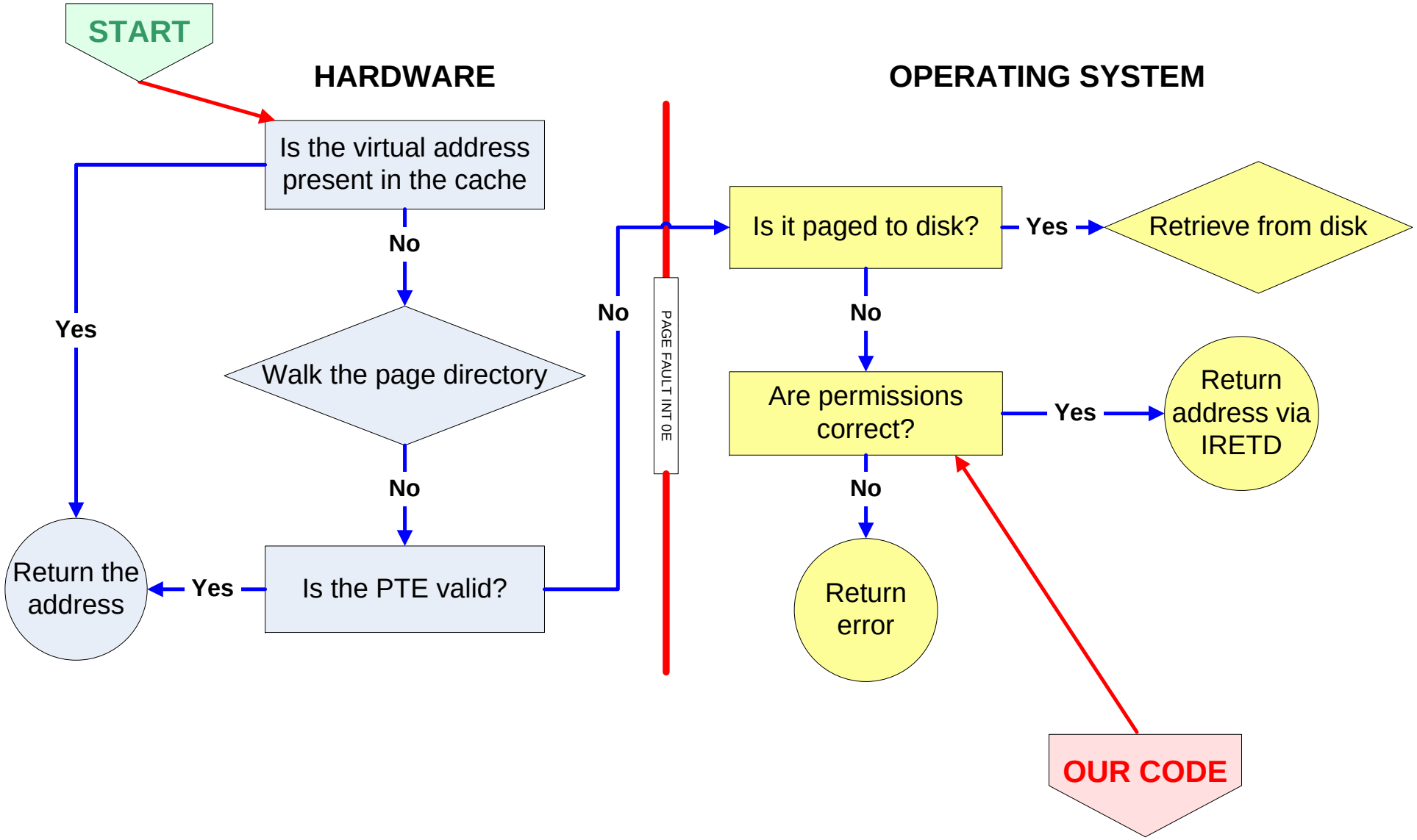
- Instruction TLB
  - Address is cached upon execution

```
mov ecx, dword ptr [eax]
mov [eax], 0xC3 // 0xC3 is a near ret
call eax
mov [eax], ecx
```

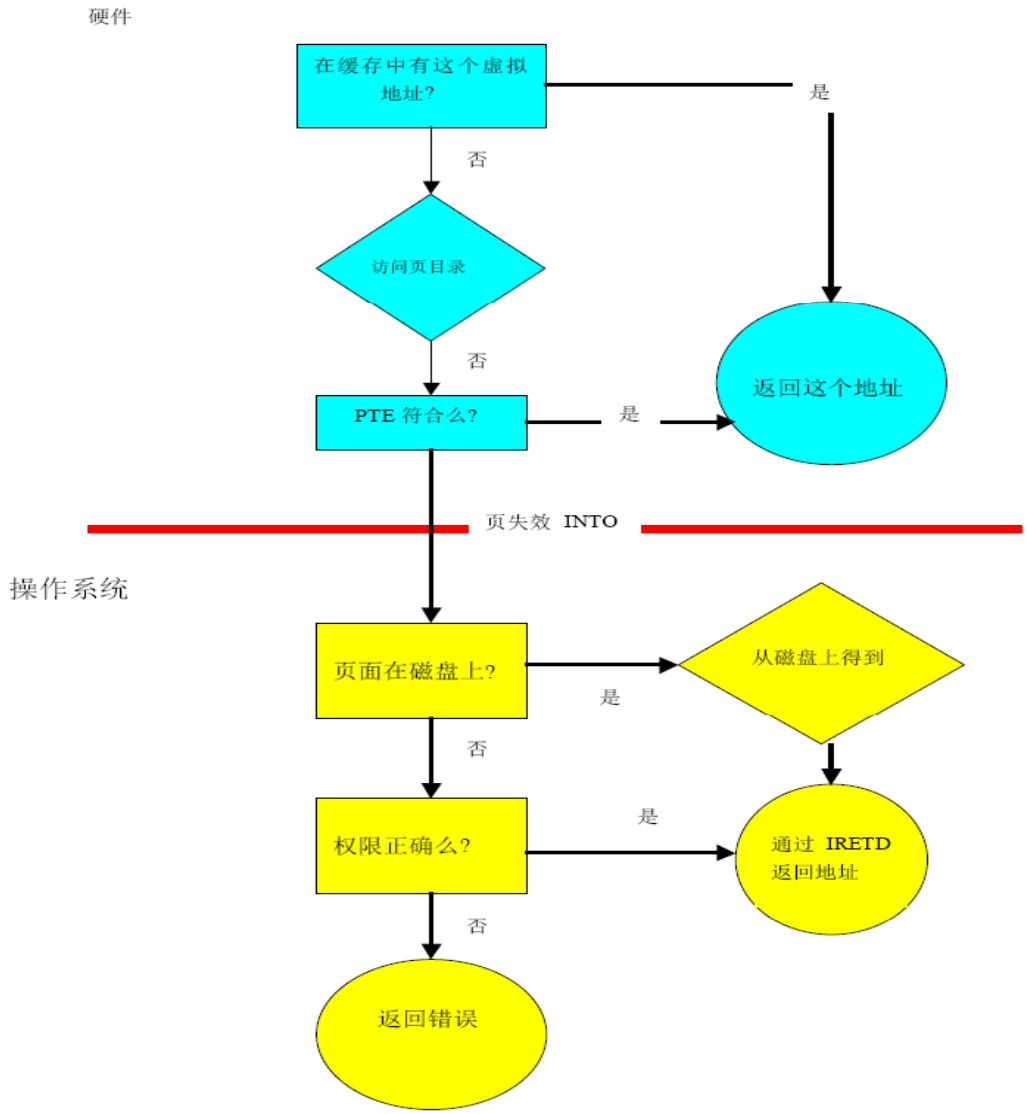
INTRODUCING SAFFRON

Introducing Saffron

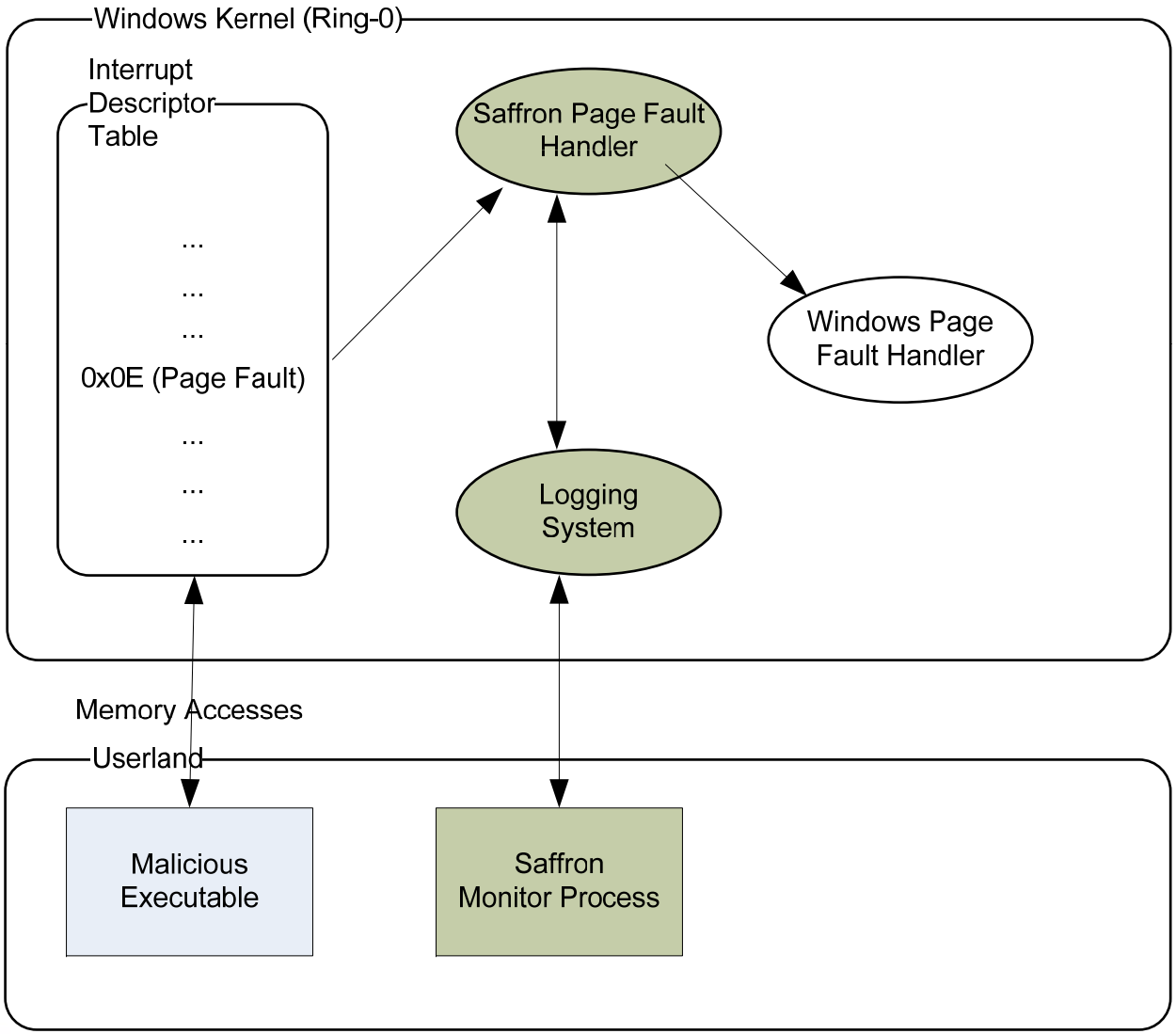
- Intel PIN and Hybrid Page Fault Handler
- Inspired by OllyBonE (Joe Stewart, DC14)
- Designed for 32-bit Intel x86 CPUs
- Replaces Windows 0x0E Trap Handler
- Logs memory accesses


START
HARDWARE
OPERATING SYSTEM
Is the virtual address present in the cache
No
Yes
Walk the page directory
No
Return the address
Yes
Is the PTE valid?
No
PAGE FAULT INT 0E
Is it paged to disk?
Yes
Retrieve from disk
No
Are permissions correct?
Yes
Return address via IRETD
No
Return error
OUR CODE

Translated (stolen) Version



Saffron System Implementation



Process Monitoring

- Mechanism
  - Overloading of supervisor bit in page fault handler
  - Mark supervisor bit on each valid PTE
  - Invalidate the page in the TLB with INVLPG
- Finding Memory
  - All process memory must be found
  - Iterate through all pages for a process
  - Read PE Header and find sections

Trap to Page Fault Handler

- Determine if a watched process
- Unset the supervisor bit
- Loads the memory into the DTLB
- Resets supervisor bit

Modifying the Autounpacker

- Watch for written pages via ITLB
- Monitor for executions into that page
- Mark Address as Original Entry Point
- Dump memory of the process

Results

- Reads, writes, and executes are exposed
- Program execution can be tracked, controlled
- Memory reads, writes are extremely apparent
- Executions only show for each individual page
- Very Fast!

Autounpacker Results

- Effective method for bypassing detection
  - SEH decode problem is easily solved
  - Memory checksum
    - No process memory is modified
    - Good dumps obtained
  - Effective across wide range of packers

Autounpacking Caveats

- System Requirements
  - Windows XP, SP2
  - No Data Execution Prevention (DEP)
  - Single CPU
    - Disable multiple CPUs in BIOS
    - /ONECPU flag in boot.ini
  - 32-bit Only (could be ported to 64bit)

Big Announcement

Technique now works on Vmware 6

Autounpacking Caveats

- Real Hardware / VMWare 6.0 or higher
  - Virtual Machines (Older versions of Vmware)
    - Play their own tricks with the ITLB
    - Extremely detectable
  - Real Hardware Take proper precautions
    - Restoration procedure
    - Isolated network
- Must not have a kernel debugger attached
    - Hilarity will ensue (silly TeLock)

Demo of Unpacking

- Demonstrate Saffron PFH

Future Work

- Initial release of Saffron-DI
  Blackhat USA 2007
- Packaged Version of Saffron-Kernel
  – Drag and drop unpacking
- Offensive Computing Integration
  – Any day now ™

Questions?

- Paper, presentation, code available at

  www.offensivecomputing.net

- Thanks to:
  - Lorie Liebrock, Houdini, Skape, Bugcheck, Skywing, Ty Bodell, Uninformed, #vax