Covert Debugging

Circumventing Software Armoring Techniques

Offensive Computing, LLC

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

Danny Quist

- Offensive Computing, Cofounder
- PhD 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
- Business Services
  - Customized malware analysis
  - Large malware data-mining / access
  - Reverse Engineering

Introduction

- Debugging Malware is a powerful tool
  - Trace Runtime Performance
  - Monitor API Calls
  - Dynamic Analysis == Automation
- Malware is getting good at preventing it
  - Debugger Detection
  - VM Detection
  - Legitimate Software Pioneered these Techniques

Overview of Talk

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

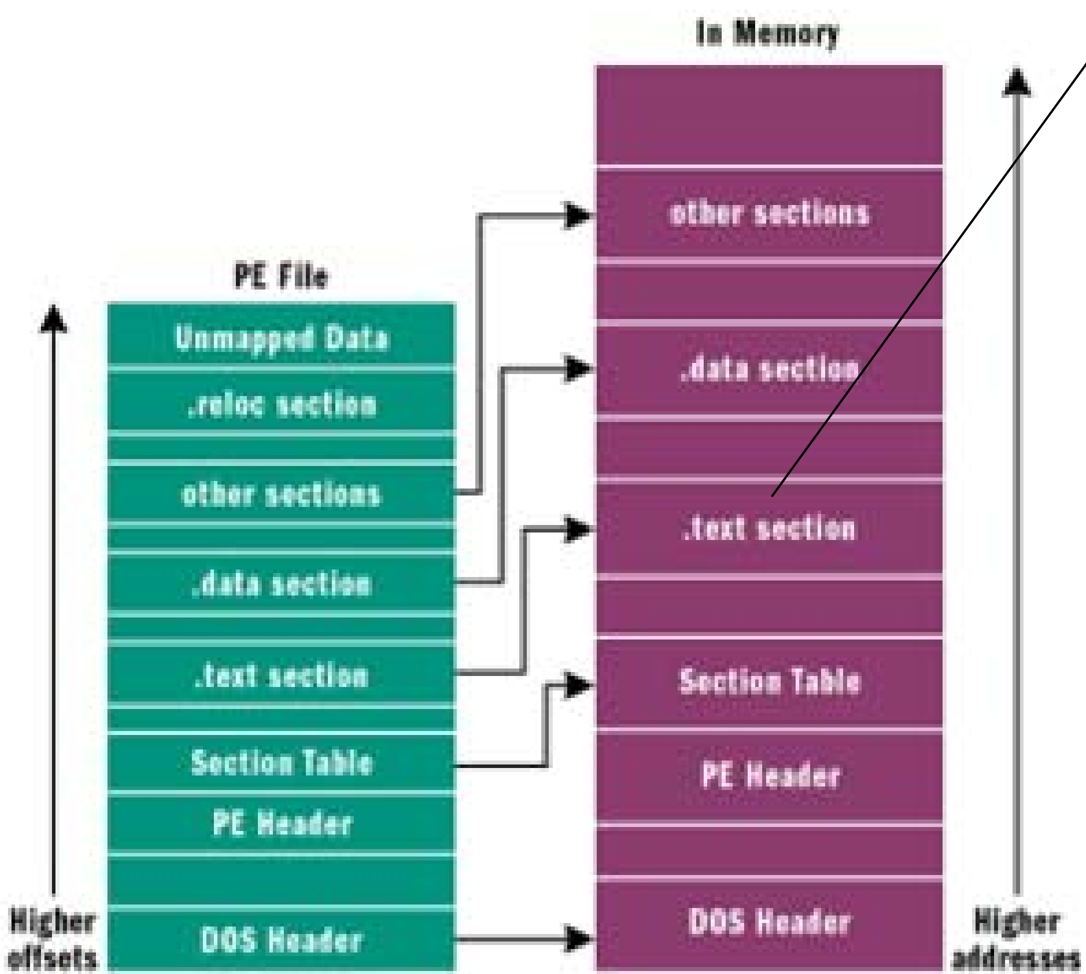
Software Armoring

- Packing/Encryption
- VM Detection
- SEH Tricks
- Debugger Detection
- Shifting Decode Frame
- Example: Microsoft’s Patchguard

Packing/Encryption

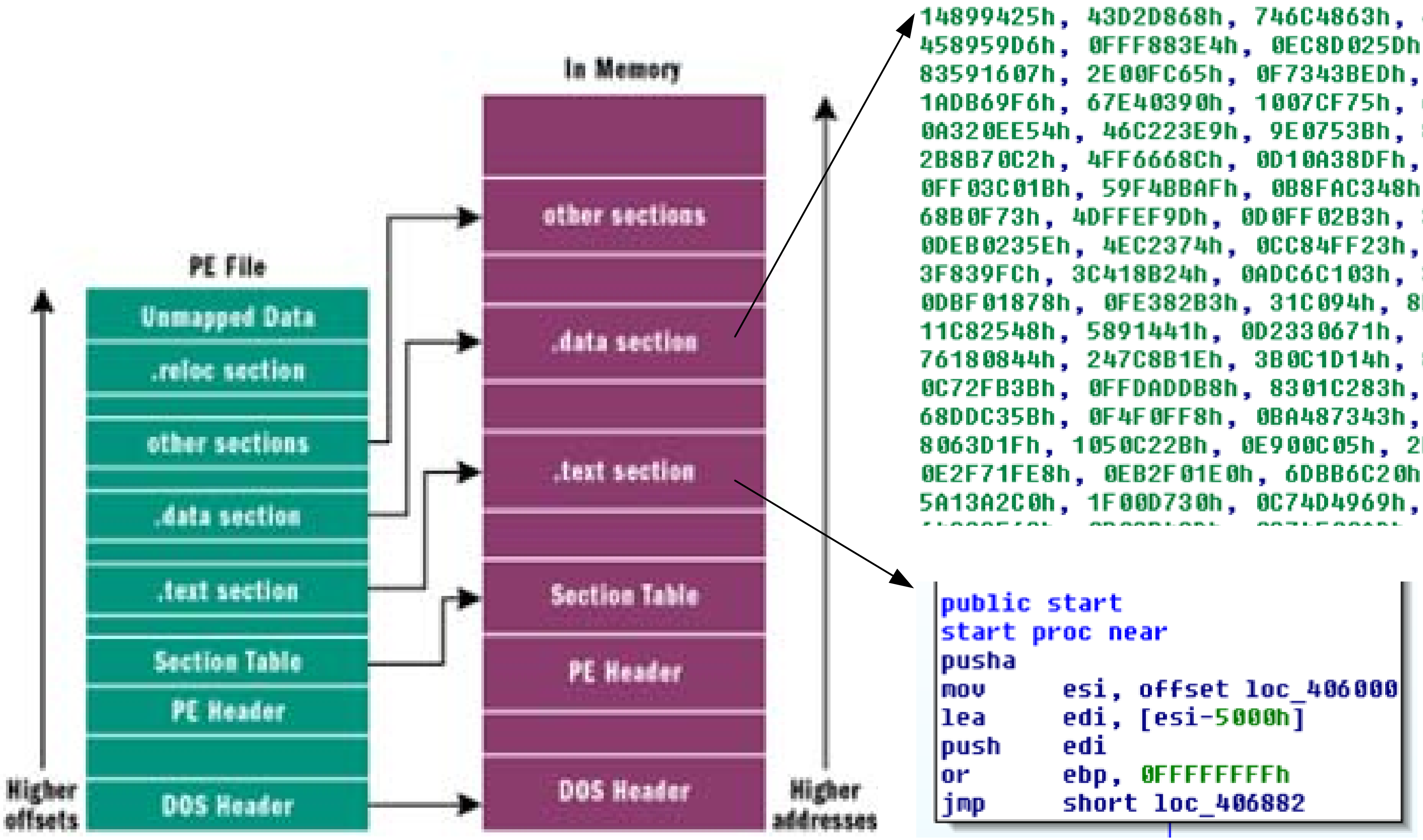
- Self-modifying Code
  - Small Decoder Stub
  - Decompresses the main executable
  - Restores imports
- Play Tricks with Portable Executables
  - Hide the Imports
  - Obscure relocations
  - Encrypt/compress the executable

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 (EFLAGS)
  - 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 bit
- SoftICE Detection
  - Modification of Int3 Scanning

SEH Tricks

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

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

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

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]

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

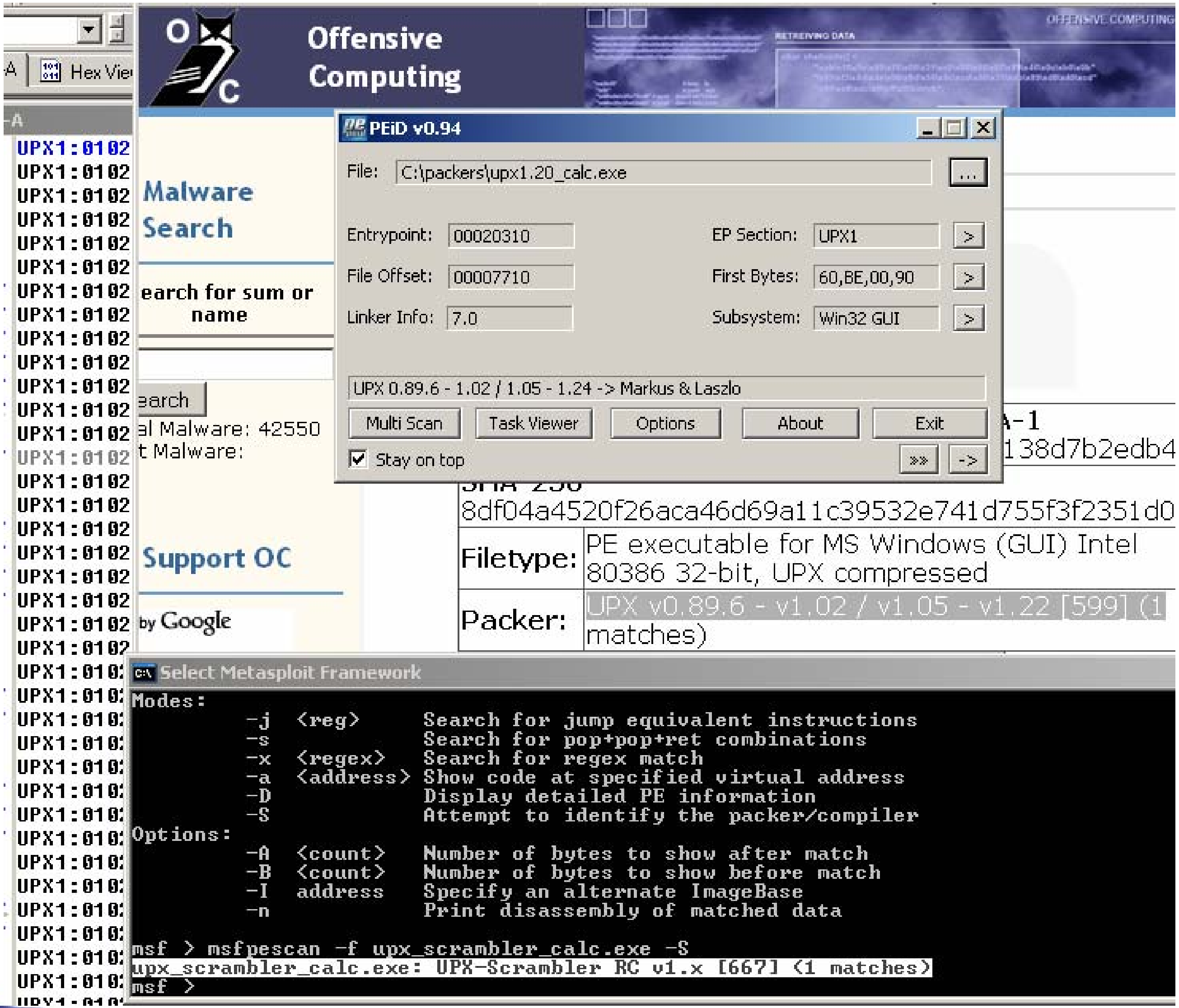
Manual Unpacking Process

- 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
  - Msfpecan / OffensiveComputing.net
  - Manually look at section names
  - Other packer scanners like
    - Protection-id
    - Pe-scan

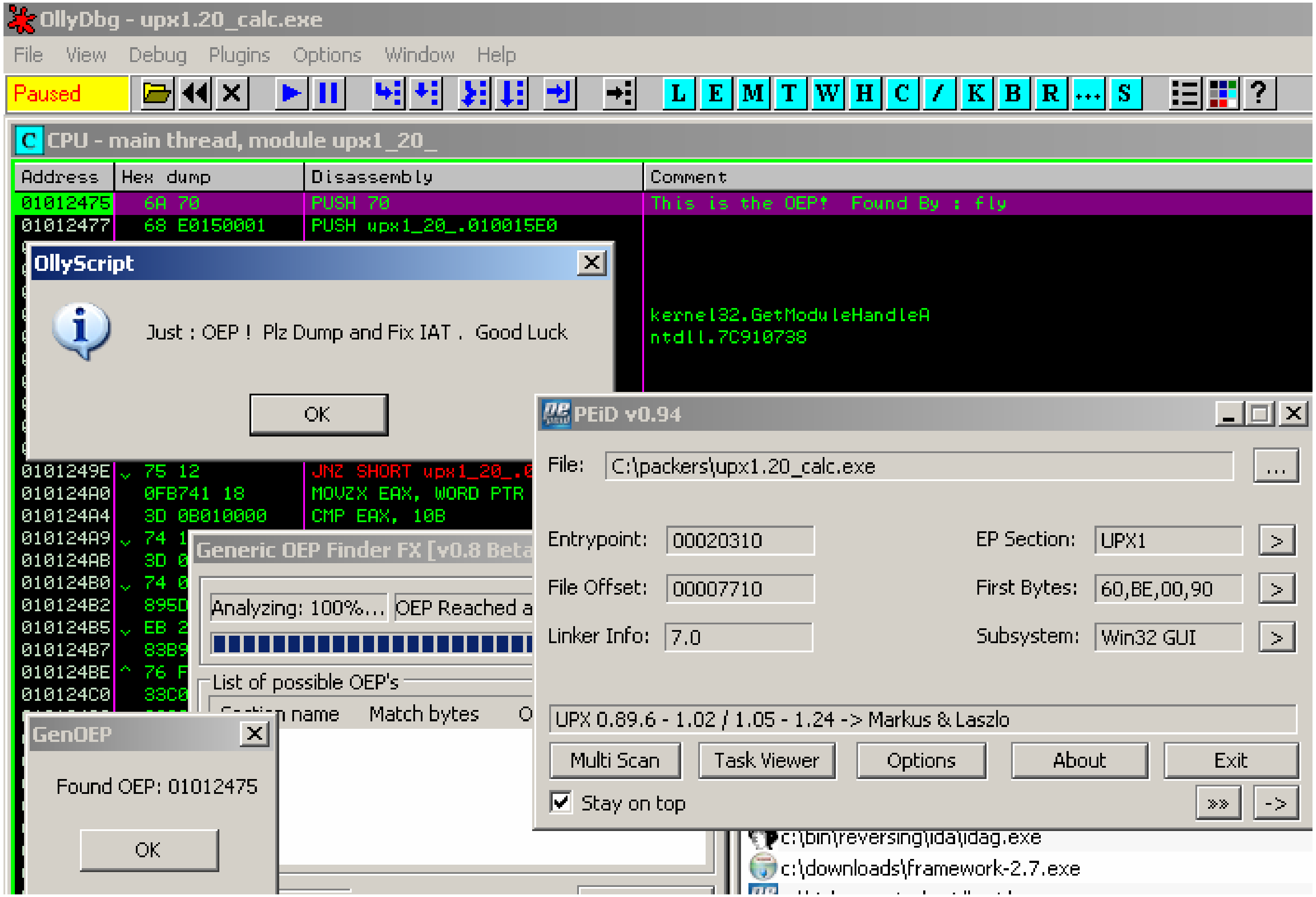
Manual Unpacking Process



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



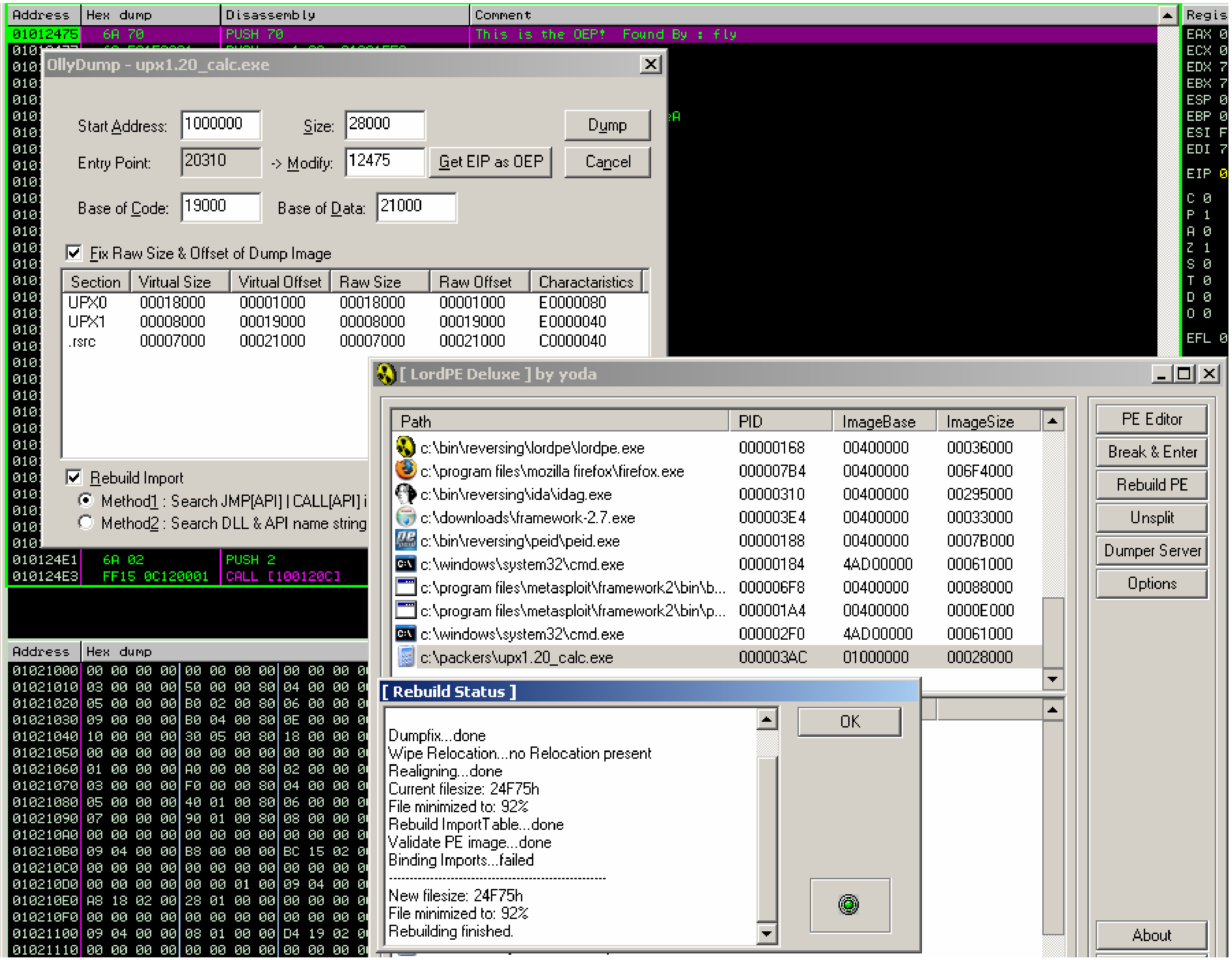
Manual Unpacking Process

– Dump process memory to file
  - OllyDump
  - LordPE
  - Custom tools

– Example:

```
void DumpProcMem(unsigned int ImageBase, unsigned int ImageSize,LPSTR filename,
   LPSTR pid) {
   SIZE_T ReadBytes = 0;  SIZE_T WriteBytes = 0;
   unsigned char * buffer = (unsigned char *) calloc(ImageSize, 1);
   HANDLE hProcess = OpenProcess(PROCESS_VM_READ, FALSE, (DWORD)atoi(pid));
   ReadProcessMemory(hProcess, (LPCVOID) ImageBase, buffer, ImageSize,
   &ReadBytes);
   HANDLE hFile = CreateFile(TEXT("oc_dumped_image.exe"),
        GENERIC_READ|GENERIC_WRITE,
        0,
        NULL,
        OPEN_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        NULL);
   WriteFile(hFile, buffer, ImageSize, &WriteBytes, NULL);
```

Manual Unpacking Process



Manual Unpacking Process

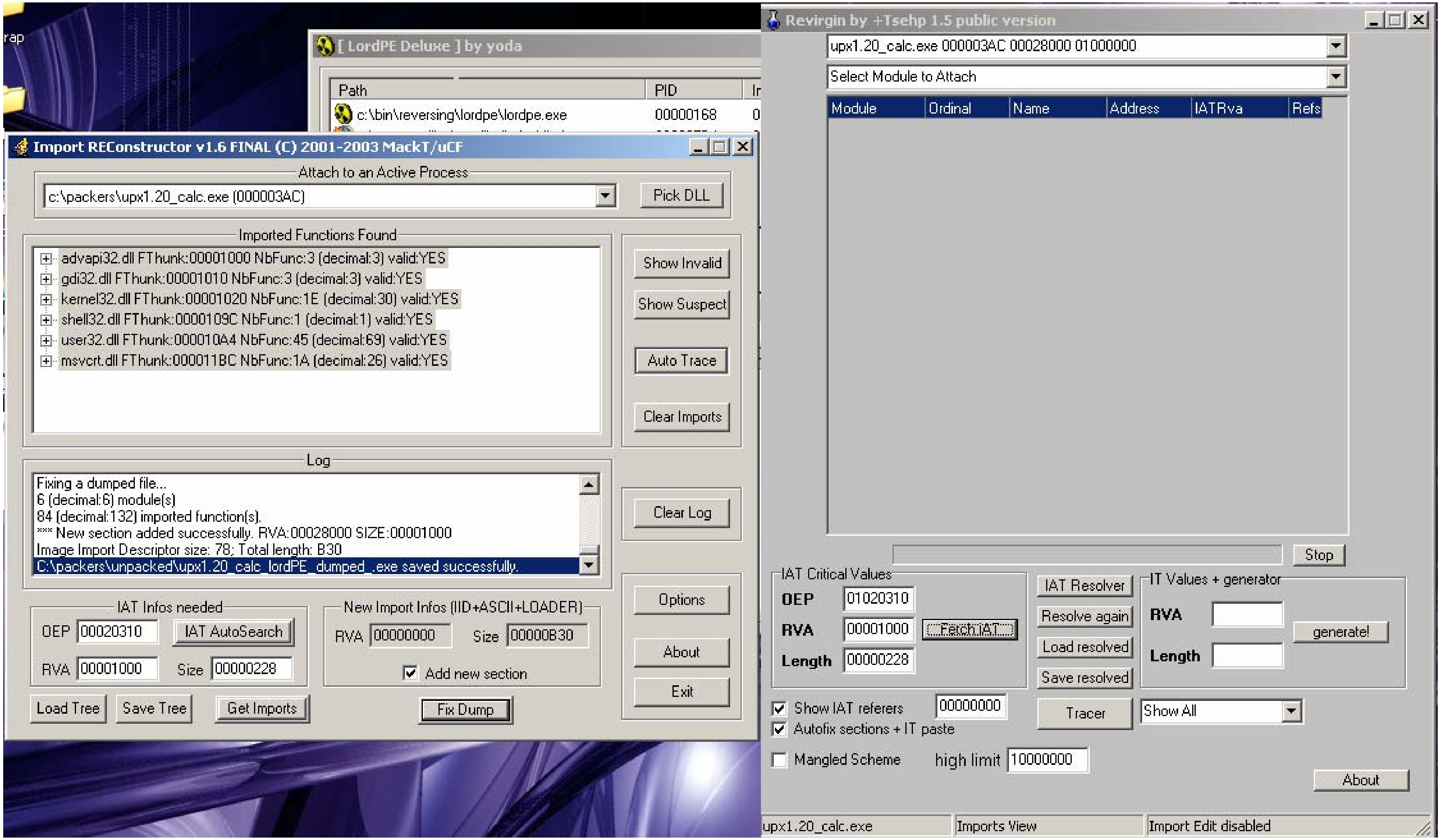
– Fixup file / rebuild Import Address Table (IAT)
  - ImportRec probably best tool
  - Revirgin by +Tsehp
  - Manually with a hex editor (tedious)

– IAT contains list of functions imported
  - Very useful for understanding capabilities

Imports

Address	Ordinal	Name	Library
01001214		??1type_info@@UAE@XZ	msvcrt
01001210		??3@YAXPAX@Z	msvcrt
01001220		?terminate@@YAXXZ	msvcrt
010010B8		CallWindowProcW	USER32
010010F0		CharNextA	USER32
0100111C		CharNextW	USER32
010010B0		CheckDlgButton	USER32
01001144		CheckMenuItem	USER32
01001148		CheckMenuRadioItem	USER32
0100110C		CheckRadioButton	USER32
010010...		ChildWindowFromPoint	USER32
010010F4		CloseClipboard	USER32
0100106C		CloseHandle	KERNEL32
0100116C		CreateDialogParamW	USER32

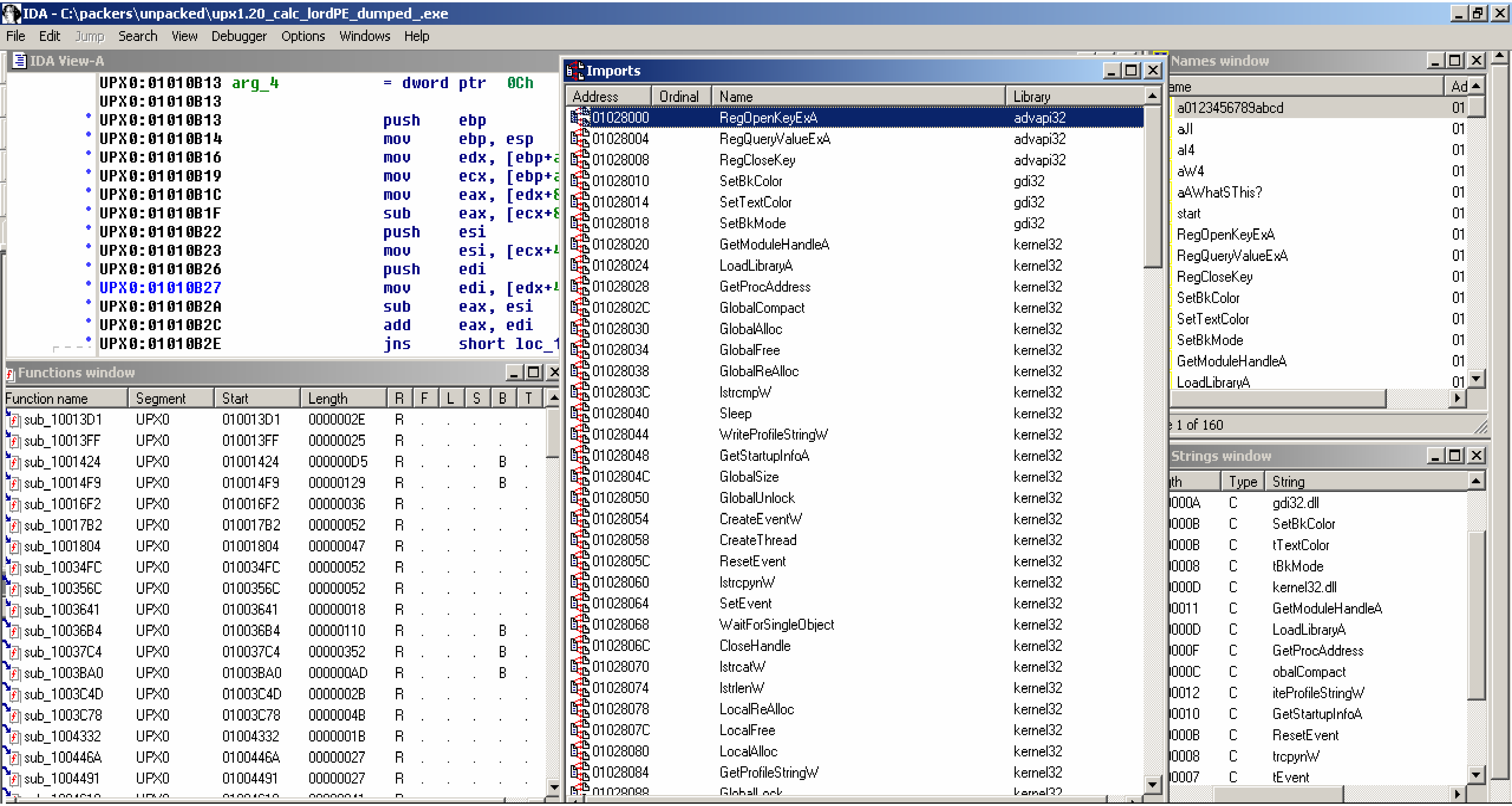
Manual Unpacking Process



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



Unpacking: The Algorithm

- Track written memory
- If that memory is executed, it’s unpacked
- Must monitor:
  - Memory writes
  - Memory Executions
- Break on execute useful here
- Automate the process

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 Attaching / Detaching

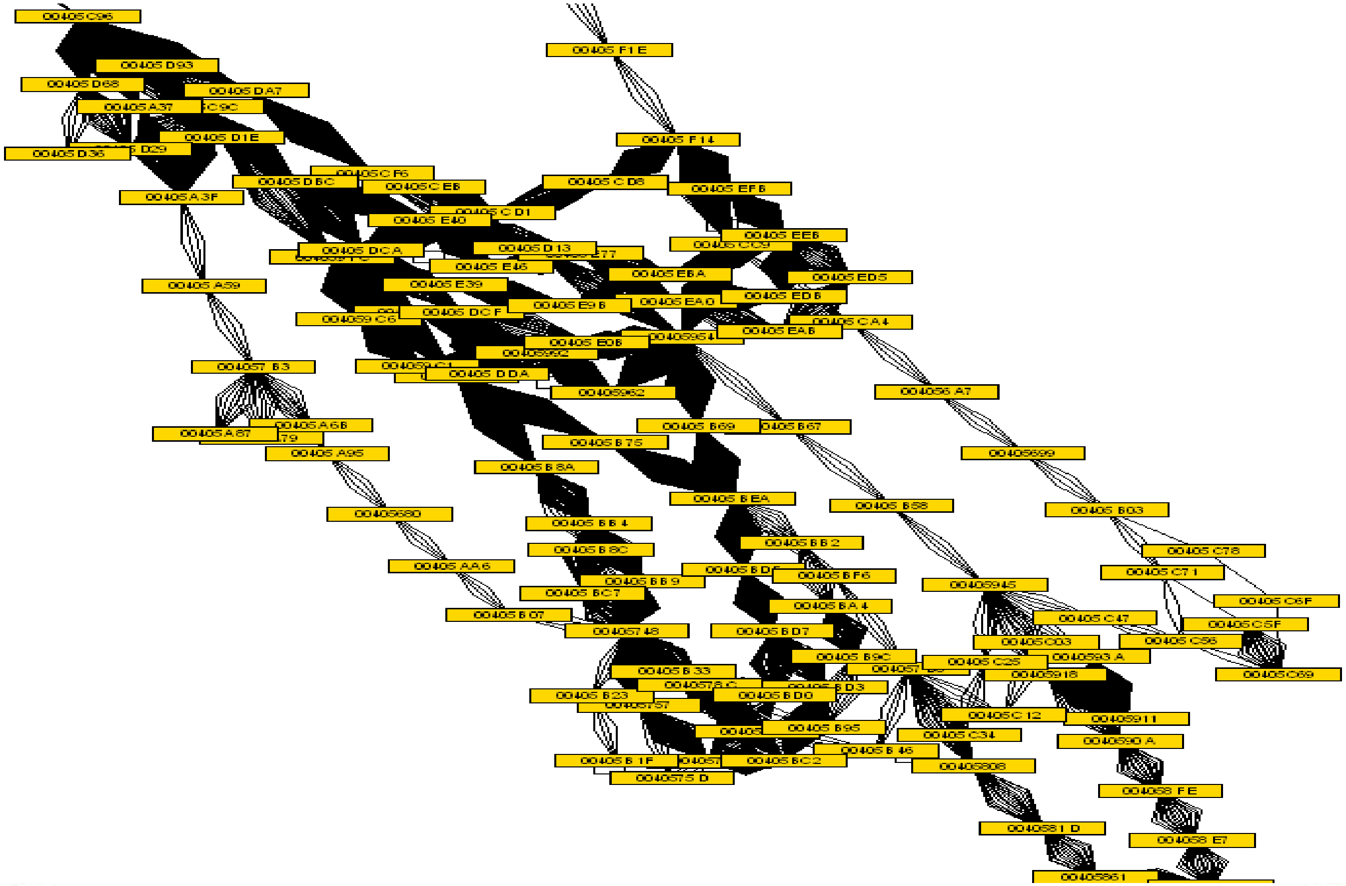
Dynamic Instrumentation

- Instruction tracing for the following packers
  - Armadillo
  - Aspack
  - FSG
  - MEW
  - PECompact
  - Telock
  - 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 address 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 Problems

- Detectable
  - Memory checksums
  - Signature scanning
- Extend this to work generically, non-detectably
- Slow – ~1,000 times slower than native
- Need faster implementation

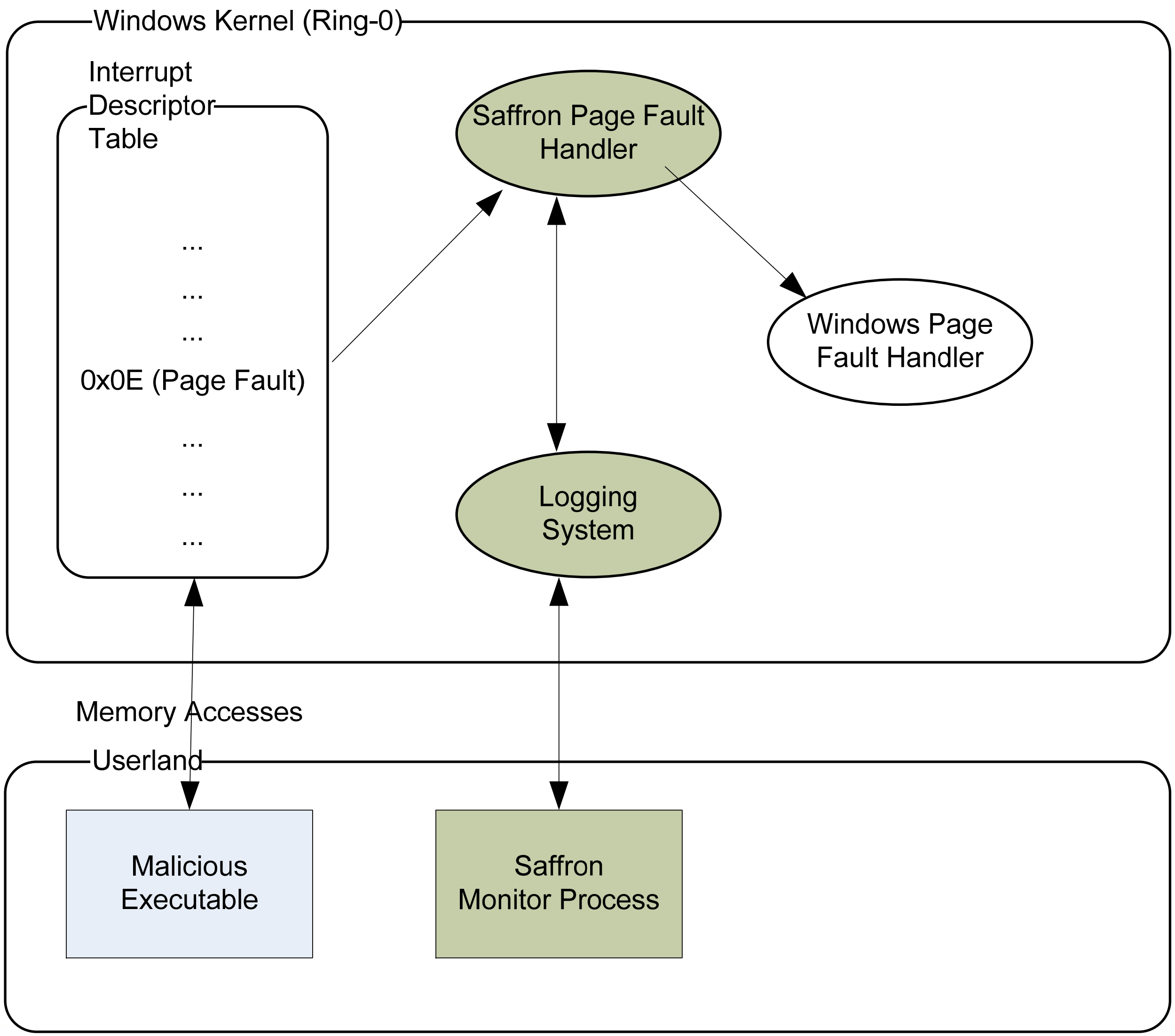
Towards a Solution

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

Introducing Saffron

- Intel PIN and Hybrid Page Fault Handler
- Extension of OllyBonE Kernel Code
- Designed for 32-bit Intel x86 CPUs
- Replaces Windows 0x0E Trap Handler
- Logs memory accesses

Saffron System Implementation



Virtual Memory Translation

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

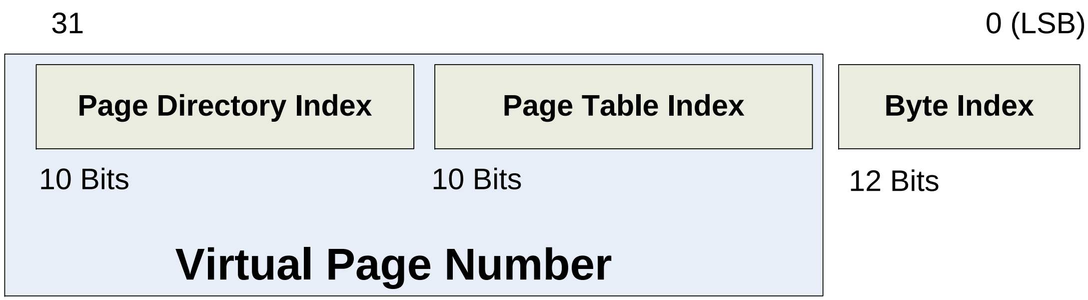
Example Memory Translation

31	0 (LSB)
Virtual Address	

CPU References Virtual Memory Address

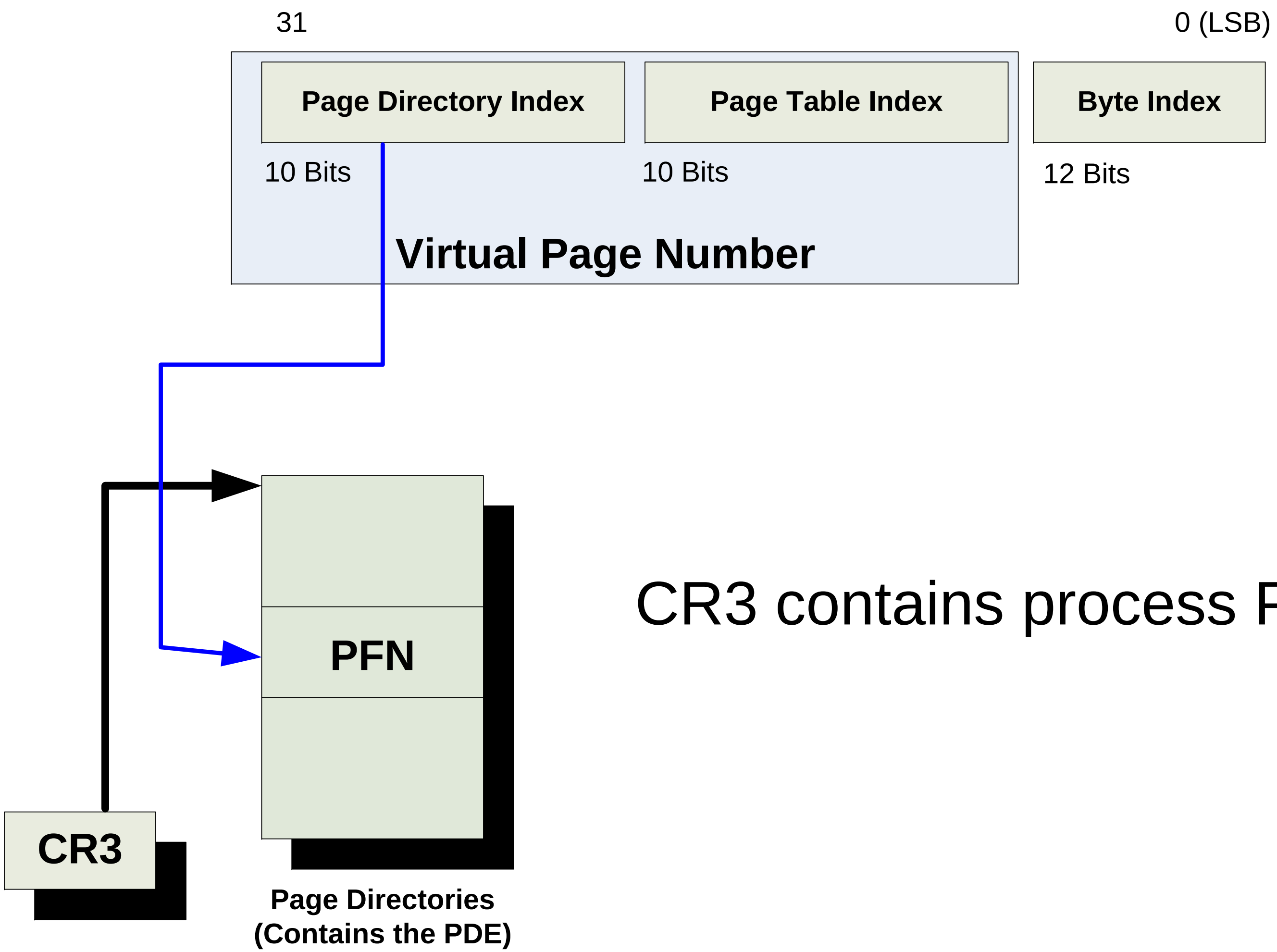
[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

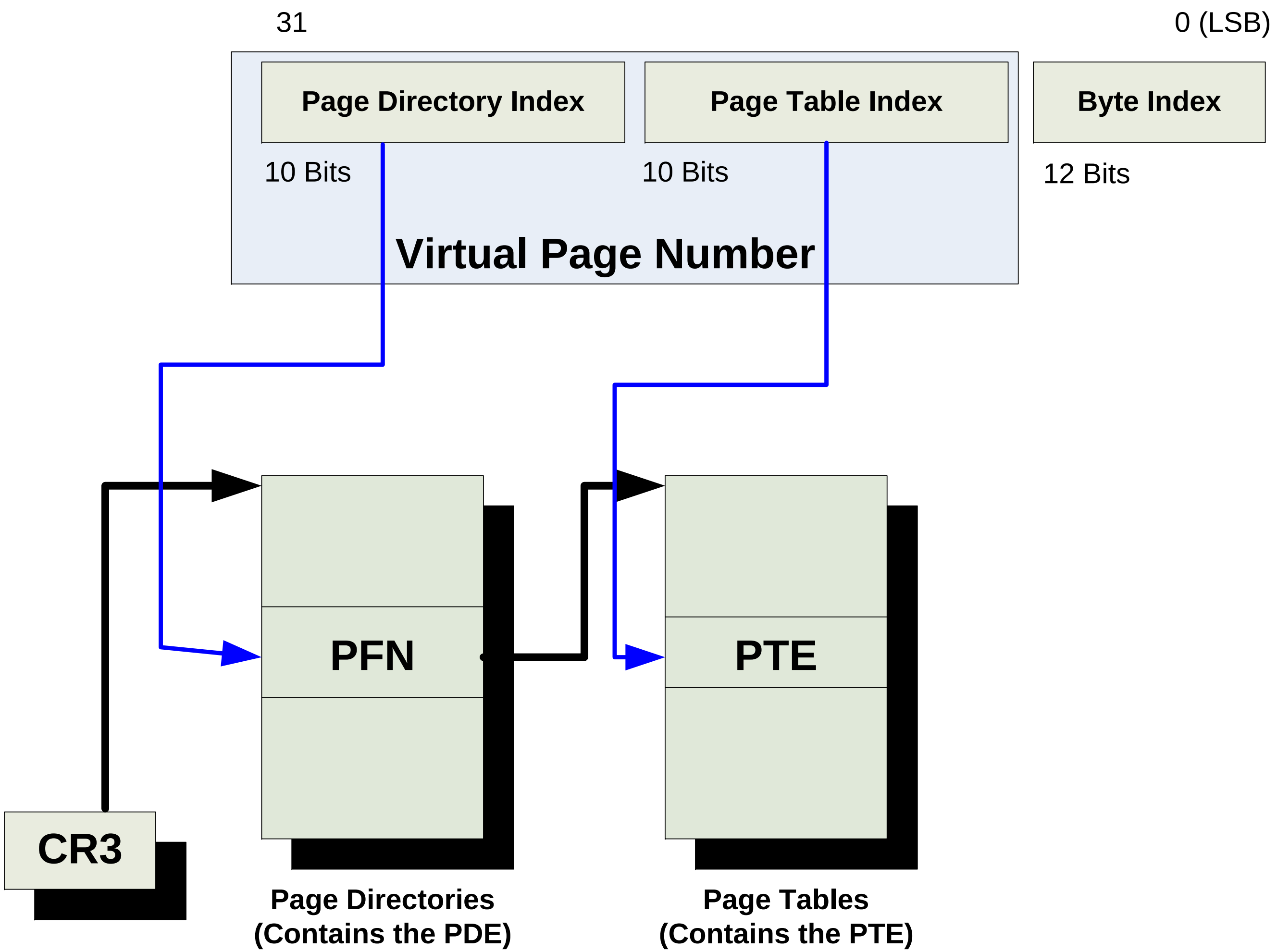
Example Memory Translation



CR3 contains process Page Directories

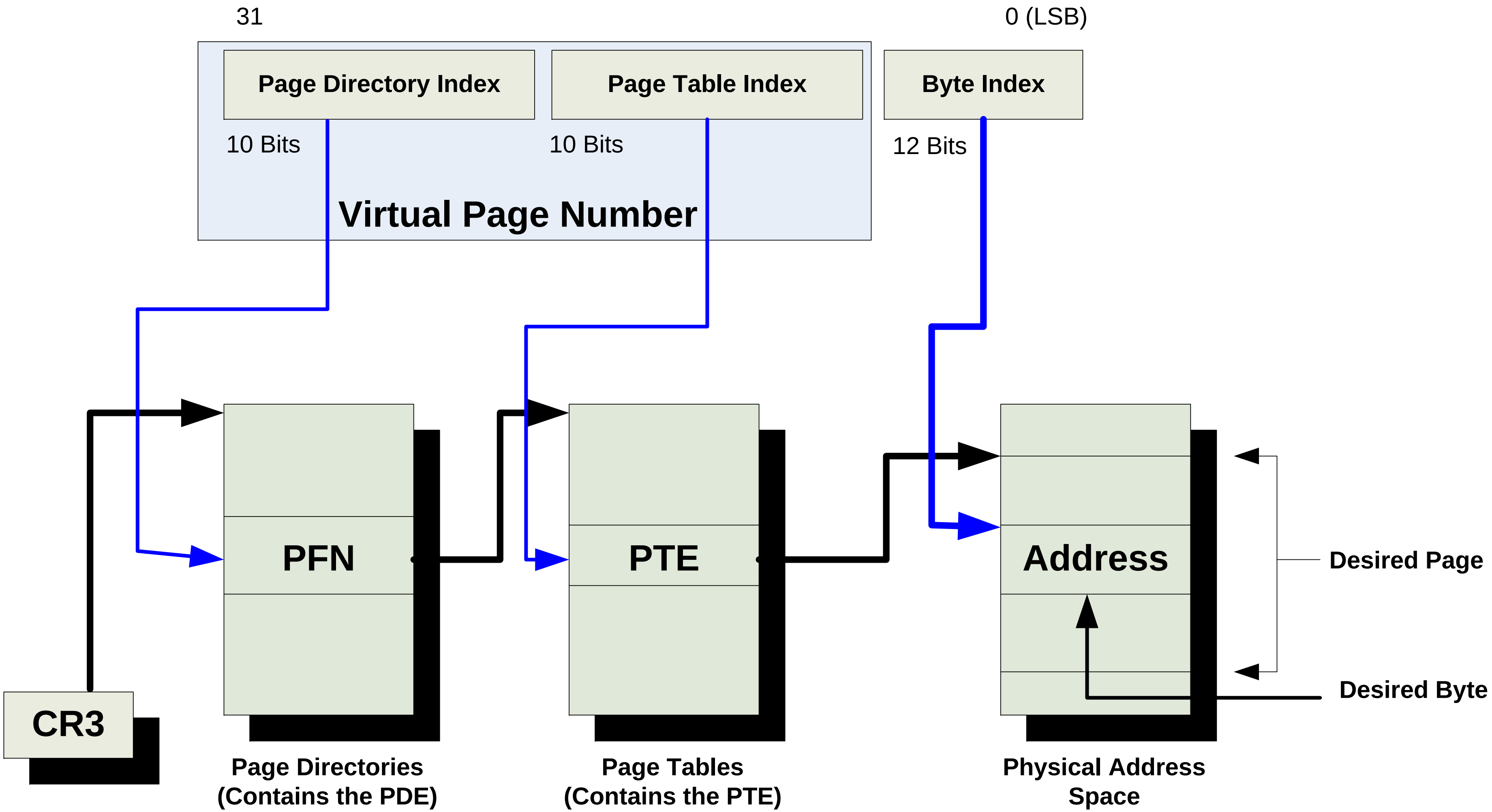
[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

Example Memory Translation



[Microsoft Windows Internals, Fourth Edition, Microsoft Press]

MMU Data Structures

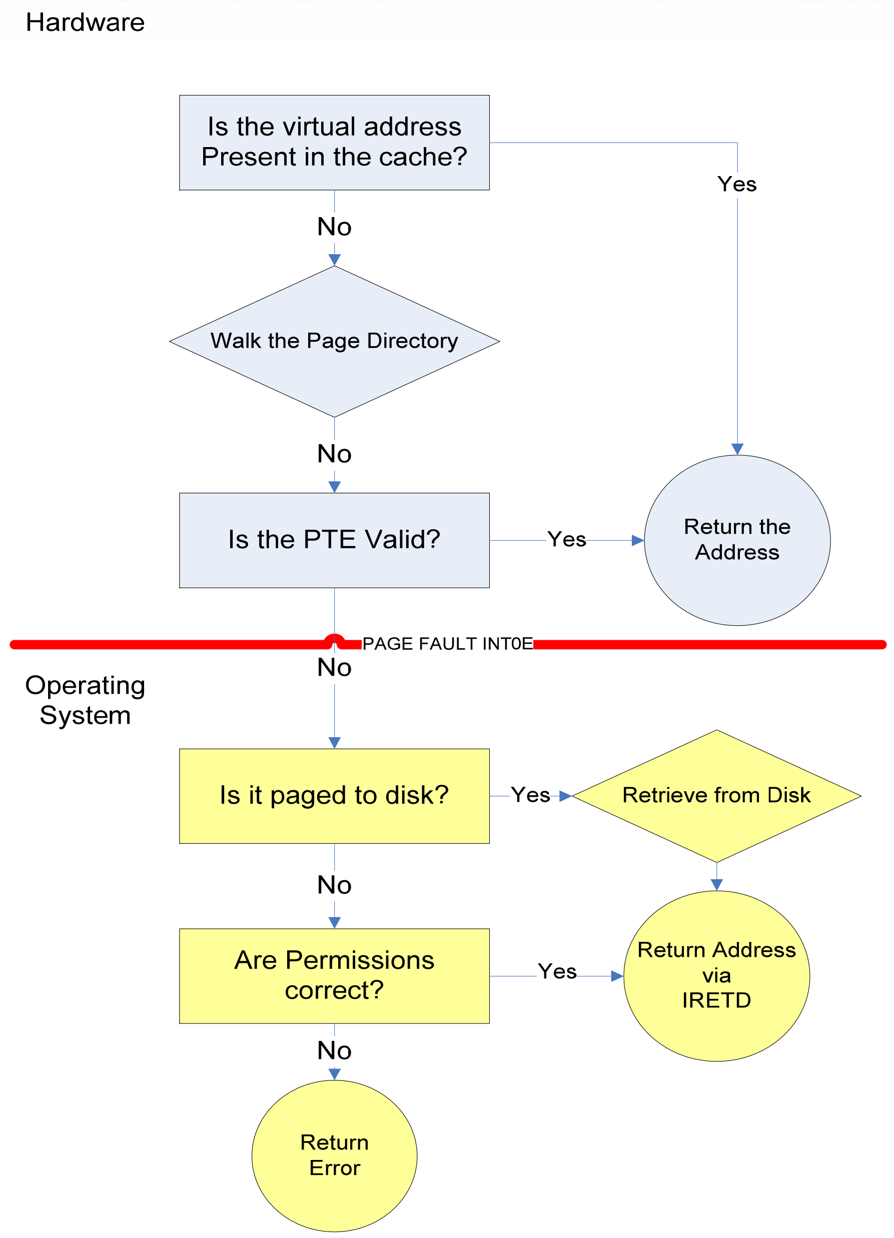
- Page Directory Entry is hardware defined
  - Contains permissions, present bit, etc.
- Page Table Entry also hardware defined
  - Permissions (Ring0 vs. all others)
  - Present bit (paged to disk or not)
  - “User” defined bits (for OS)

Virtual Address Translation

- 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 ITLB == faster code
  - DTLB accessed less


Hardware
Is the virtual address Present in the cache?
Yes
No
Walk the Page Directory
No
Is the PTE Valid?
Yes
Return the Address
PAGE FAULT INT0E
No
Operating System
Is it paged to disk?
Yes
Retrieve from Disk
No
Are Permissions correct?
Yes
Return Address via IRETD
No
Return Error

Process Monitoring

- Overloading of supervisor bit in page fault handler
- All process memory must be found
- Iterate through all pages for a process
  - Windows application memory 0x00000000 – 0x7FFFFFFF
- Mark supervisor bit on each valid PTE
- Invalidate the page in the TLB with INVLPG
- Hook heap allocation so new pages are watched

Trap to Page Fault Handler

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

Results

- Memory accesses are visible
- 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

Modifying the Autounpacker

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

Video Demo of Unpacking

- Demonstrate Saffron

Autounpacker Results

- Effective method for bypassing debugger attacks
  - SEH decode problem is easily solved
  - Memory checksum
    - No process memory is modified
    - p0wn3d!!!
- Shifting decode frame
  - Slight modification under development, but effective

Future Work

- Develop full-fledged API
- Problems
  - Sometimes all page markings are lost
  - Still detectable at some level

Questions?

- Paper, presentation available at

www.offensivecomputing.net