



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

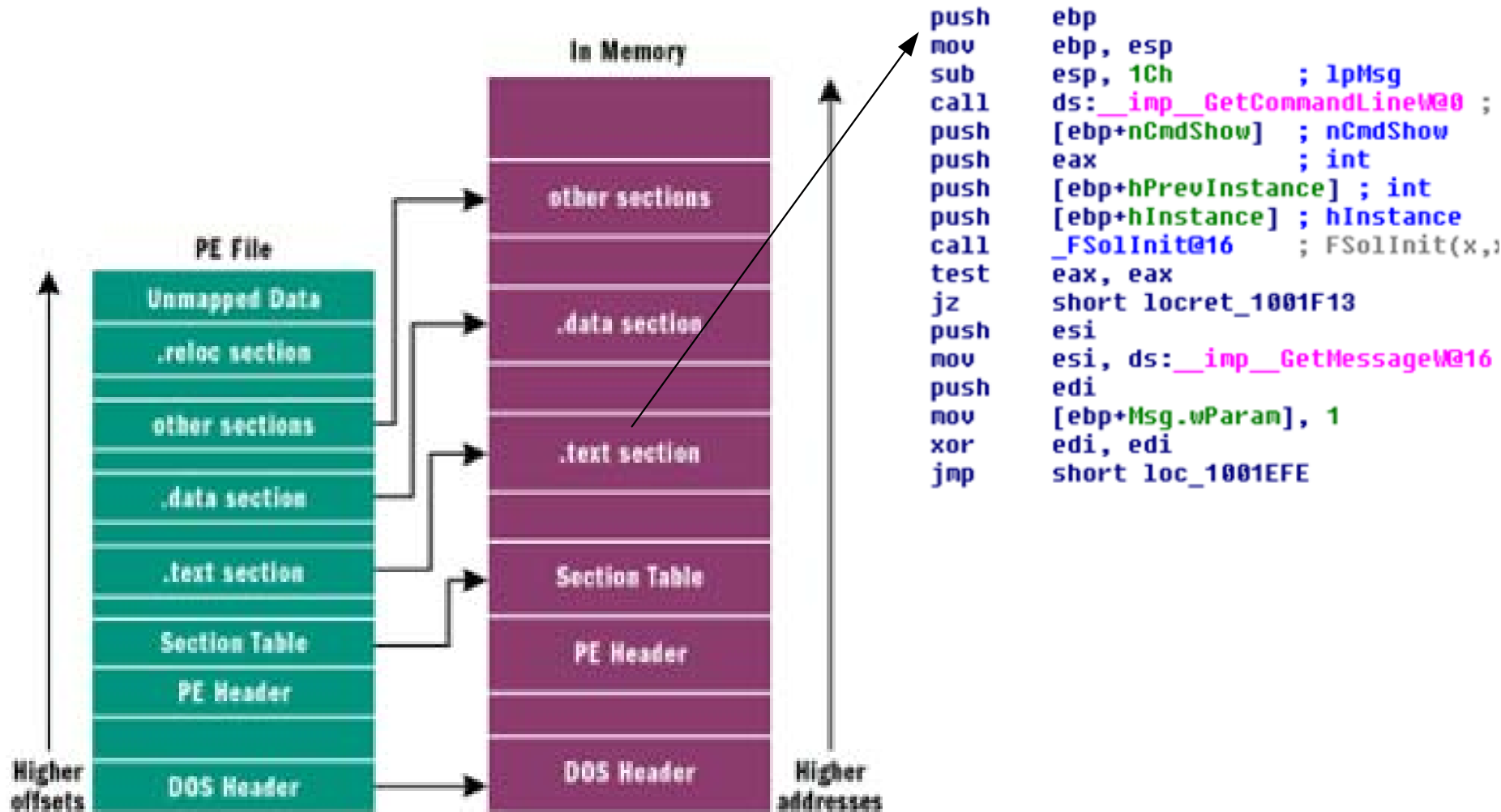


Packing/Encryption

- 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

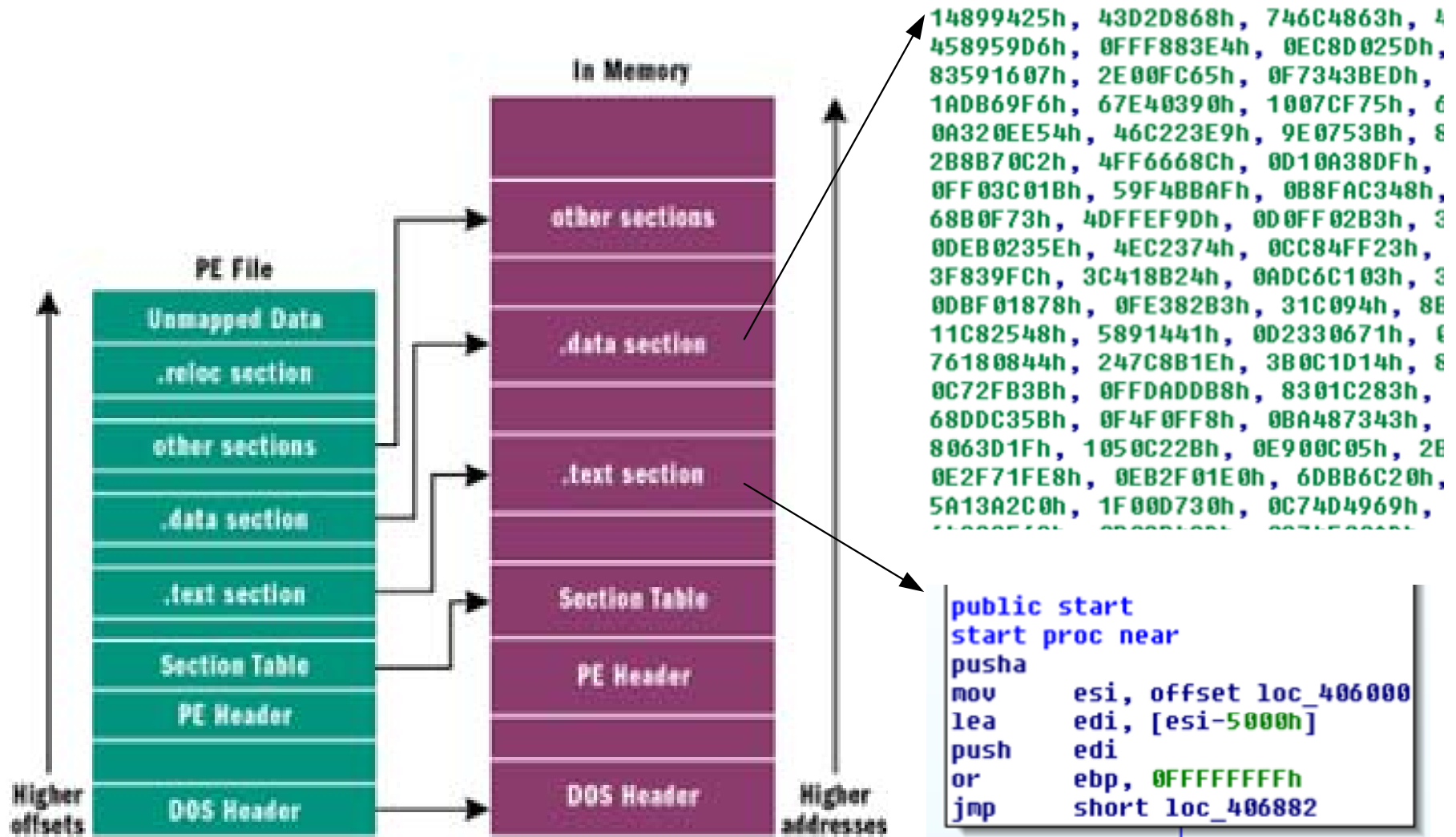


Normal PE File





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.



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

The screenshot displays a Windows desktop environment. In the background, there is a web browser window titled "Offensive Computing" with a "RETRIEVING DATA" status. In the foreground, the PEID v0.94 application is open, showing the following details for the file "C:\packers\upx1.20_calc.exe":

- File: C:\packers\upx1.20_calc.exe
- Entrypoint: 00020310
- EP Section: UPX1
- File Offset: 00007710
- First Bytes: 60,BE,00,90
- Linker Info: 7.0
- Subsystem: Win32 GUI

Below the input fields, the packer is identified as "UPX 0.89.6 - 1.02 / 1.05 - 1.24 -> Markus & Laszlo". The application also includes buttons for "Multi Scan", "Task Viewer", "Options", "About", and "Exit", along with a "Stay on top" checkbox.

In the bottom-left corner, a Metasploit terminal window is open, displaying the following command and output:

```
msf > msfpescan -f upx_scrambler_calc.exe -S
upx_scrambler_calc.exe: UPX-Scrambler RC v1.x [667] (1 matches)
msf >
```



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

The screenshot displays the OllyDbg interface for the file `upx1.20_calc.exe`. The CPU window shows the main thread at address `01012475` with the instruction `PUSH 70`, which is identified as the OEP (Original Entry Point). A comment for this instruction reads: "This is the OEP! Found By : fly".

Several utility windows are open:

- OllyScript**: A message box stating "Just : OEP ! Plz Dump and Fix IAT . Good Luck".
- PEiD v0.94**: A tool window showing the file `C:\packers\upx1.20_calc.exe` with the following details:
 - Entrypoint: `00020310`
 - EP Section: `UPX1`
 - File Offset: `00007710`
 - First Bytes: `60,BE,00,90`
 - Linker Info: `7.0`
 - Subsystem: `Win32 GUI`
- Generic OEP Finder FX [v0.8 Beta]**: A window showing the analysis progress at 100% and the message "OEP Reached".
- GenOEP**: A small window reporting "Found OEP: 01012475".

The disassembly window shows the following instructions:

Address	Hex dump	Disassembly	Comment
01012475	6A 70	PUSH 70	This is the OEP! Found By : fly
01012477	68 E0150001	PUSH upx1_20_.010015E0	
0101249E	75 12	JNZ SHORT upx1_20_.010124A0	
010124A0	0FB741 18	MOVZX EAX, WORD PTR [EAX]	
010124A4	3D 0B010000	CMP EAX, 10B	
010124A9	74 10	JZ SHORT upx1_20_.010124B0	
010124AB	3D 00000000	CMP EAX, 0	
010124B0	74 04	JZ SHORT upx1_20_.010124B8	
010124B2	895D	MOV EDI, EDI	
010124B5	EB 20	MOV EBX, EBX	
010124B7	83B9	MOV ECX, ECX	
010124BE	76 F0	JLE SHORT upx1_20_.010124C0	
010124C0	33C0	XOR EAX, EAX	



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

The screenshot displays the OllyDbg interface with several windows open during the manual unpacking process of a malware executable.

OllyDbg Main Window: Shows the disassembly of the malware. The instruction at address 01012475 is highlighted: `6A 79 PUSH 79`. A comment indicates: `This is the OEP. Found By : Fly`. The registers pane on the right shows the state of various registers.

OllyDump - upx1.20_calc.exe: A dialog box for dumping the executable. It shows the Start Address (1000000), Size (28000), Entry Point (20310), and Base of Code (19000). A table of sections is visible:

Section	Virtual Size	Virtual Offset	Raw Size	Raw Offset	Characteristics
UPX0	00018000	00001000	00018000	00001000	E0000080
UPX1	00008000	00019000	00008000	00019000	E0000040
.rsrc	00007000	00021000	00007000	00021000	C0000040

LordPE Deluxe: A utility window showing a list of running processes. The process `c:\packers\upx1.20_calc.exe` is selected, showing its PID (000003AC) and ImageBase (01000000).

Rebuild Status: A dialog box showing the progress of the unpacking process. The status is as follows:

- Dumpfix...done
- Wipe Relocation...no Relocation present
- Realigning...done
- Current filesize: 24F75h
- File minimized to: 92%
- Rebuild ImportTable...done
- Validate PE image...done
- Binding Imports...failed
- New filesize: 24F75h
- File minimized to: 92%
- Rebuilding finished.



Manual Unpacking Process

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

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

The screenshot shows the manual unpacking process using LordPE Deluxe and Import REConstructor. The LordPE Deluxe window displays the path of the dumped file: `c:\bin\reversing\lordpe\lordpe.exe` with PID `00000168`. The Import REConstructor window shows the process `c:\packers\upx1.20_calc.exe (000003AC)` attached. The imported functions list includes `advapi32.dll`, `gdi32.dll`, `kernel32.dll`, `shell32.dll`, `user32.dll`, and `msvcrt.dll`. The log shows the process of fixing a dumped file and adding a new section. The IAT Infos needed section shows OEP `00020310`, RVA `00001000`, and Size `00000228`. The New Import Infos (IID+ASCII+LOADER) section shows RVA `00000000` and Size `00000830`. The IAT Critical Values section shows OEP `01020310`, RVA `00001000`, and Length `00000228`. The IAT Resolver section shows the 'Fetch IAT' button. The IT Values + generator section shows the 'generate!' button. The bottom status bar shows `upx1.20_calc.exe` in Imports View with Import Edit disabled.

LordPE Deluxe [by yoda]

Path	PID	Ir
c:\bin\reversing\lordpe\lordpe.exe	00000168	0

Import REConstructor v1.6 FINAL (C) 2001-2003 MackT/uCF

Attach to an Active Process

c:\packers\upx1.20_calc.exe (000003AC) Pick DLL

Imported Functions Found

- advapi32.dll FT:hunk:00001000 NbFunc:3 (decimal:3) valid:YES
- gdi32.dll FT:hunk:00001010 NbFunc:3 (decimal:3) valid:YES
- kernel32.dll FT:hunk:00001020 NbFunc:1E (decimal:30) valid:YES
- shell32.dll FT:hunk:0000109C NbFunc:1 (decimal:1) valid:YES
- user32.dll FT:hunk:000010A4 NbFunc:45 (decimal:69) valid:YES
- msvcrt.dll FT:hunk:000011BC NbFunc:1A (decimal:26) valid:YES

Log

Fixing a dumped file...
6 (decimal:6) module(s)
84 (decimal:132) imported function(s).
*** New section added successfully. RVA:00028000 SIZE:00001000
Image Import Descriptor size: 78; Total length: B30
C:\packers\unpacked\upx1.20_calc_lordPE_dumped_exe saved successfully.

IAT Infos needed

OEP IAT AutoSearch

RVA Size

New Import Infos (IID+ASCII+LOADER)

RVA Size

Add new section

Options About Exit

IAT Critical Values

OEP RVA Length

Fetch IAT

IAT Resolver

Resolve again Load resolved Save resolved

IT Values + generator

RVA Length generate!

Show IAT referers Autofix sections + IT paste Mangled Scheme high limit

Tracer Show All

Stop

About

upx1.20_calc.exe Imports View Import Edit disabled



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

The screenshot displays the IDA Pro interface for a file named 'C:\packers\unpacked\upx1.20_calcLordPE_dumped_exe'. The main window shows assembly code for a function starting at address 01010B13. The code includes instructions for pushing arguments, setting up the stack, and performing various operations on registers like ebx, esp, edx, ecx, eax, esi, and edi. The 'Imports' window lists various system API calls such as RegOpenKeyExA, RegQueryValueExA, and RegCloseKey, all imported from 'advapi32'. The 'Functions window' lists several subroutines (sub_10013D1 to sub_1004491) with their respective segments, start addresses, lengths, and flags. The 'Names window' shows a list of symbols, including 'a0123456789abcd' and 'a11'. The 'Strings window' displays a list of strings, such as 'gdi32.dll', 'SetBkColor', and 'tTextColor', along with their memory addresses and types.



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

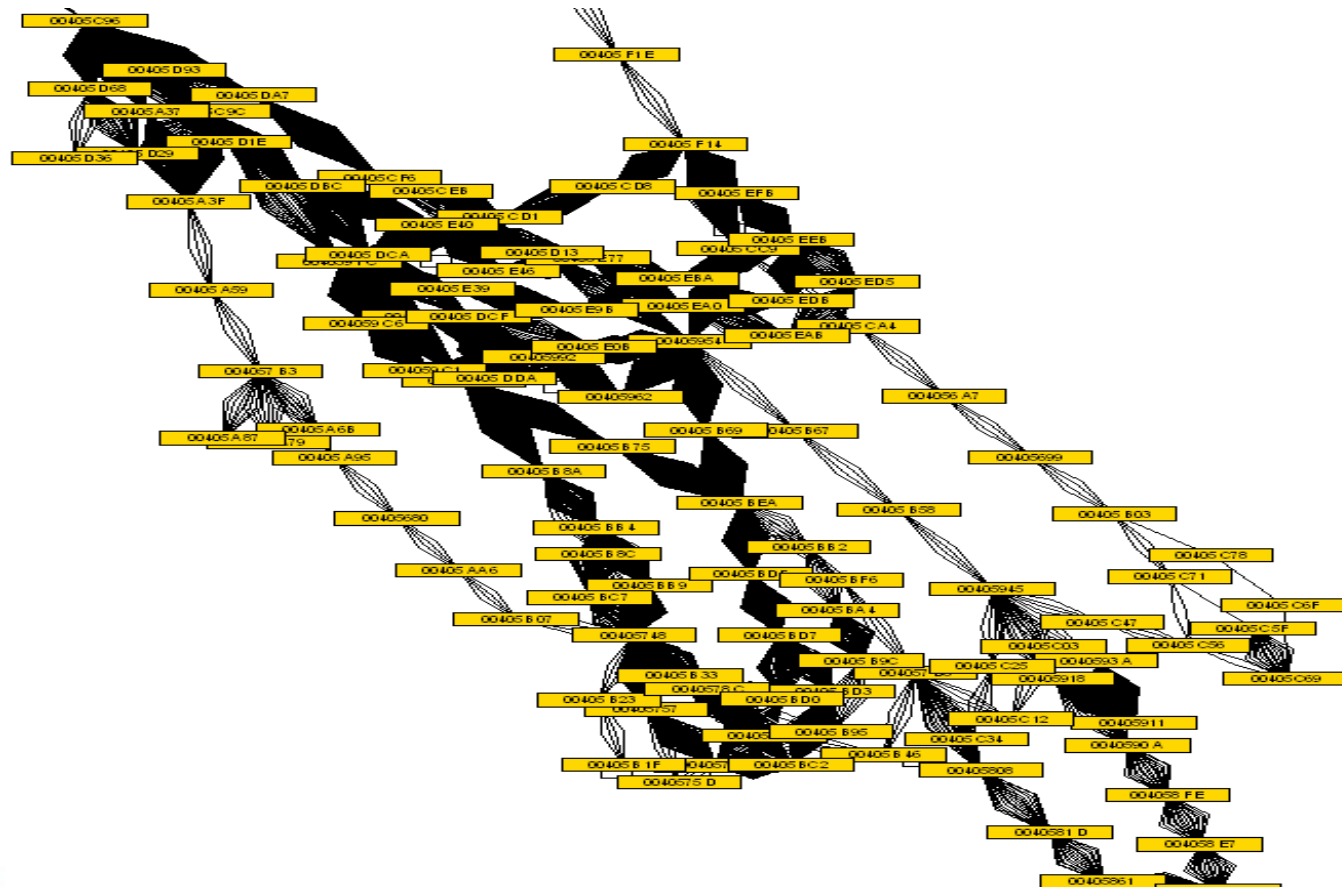


Aspack 2.12



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



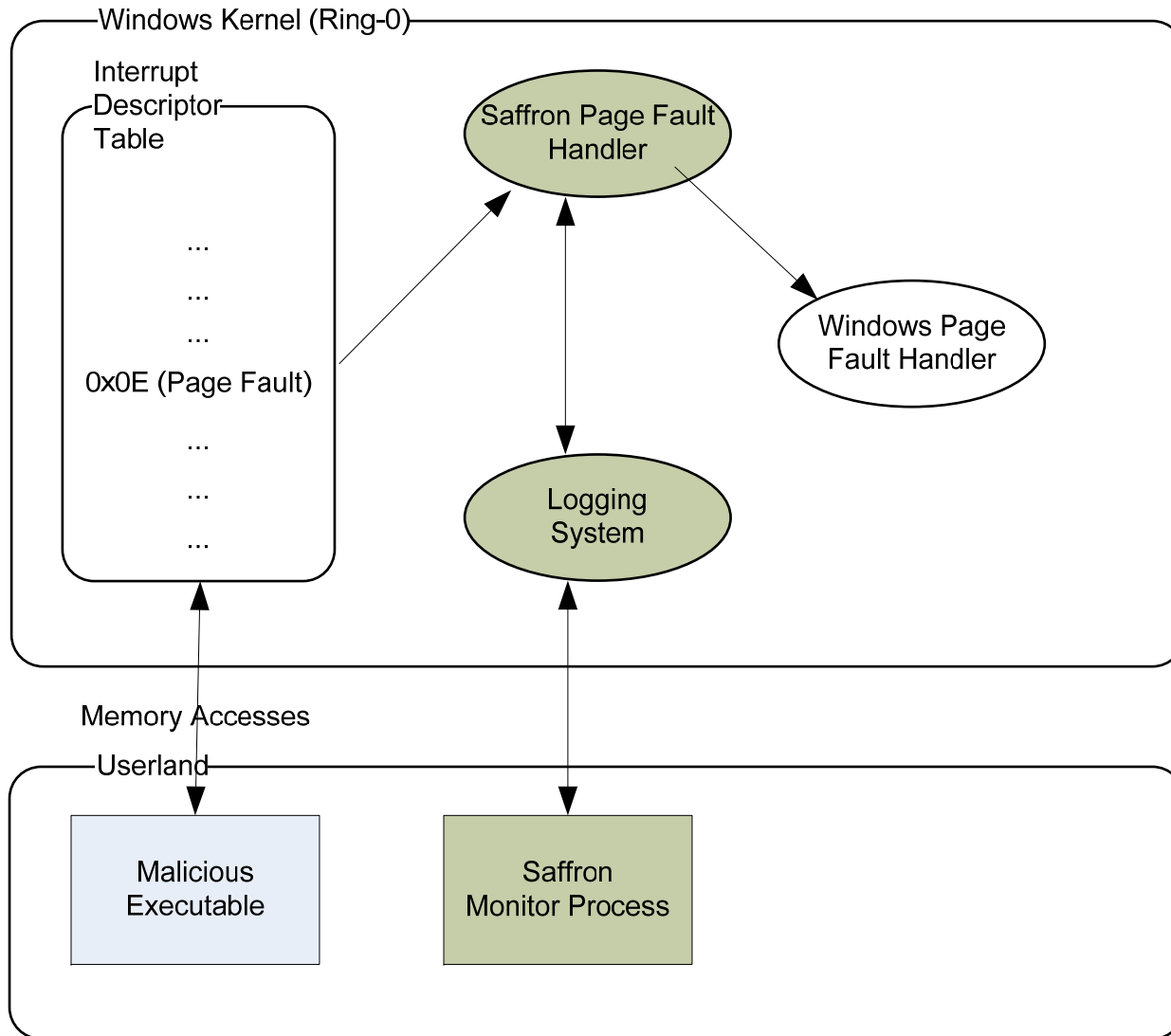
Introducing Saffron

- 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

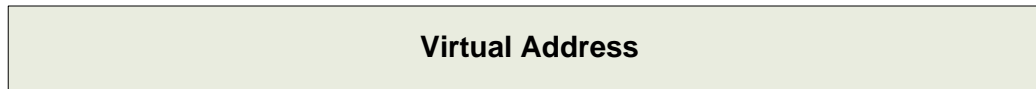
- Each process has its own memory
- Memory must be translate from Virtual to Physical Address
- Non-PAE 32bit Processors use 2 page indexes and a byte index
- Each process has its own Page Directory



Example Memory Translation

31

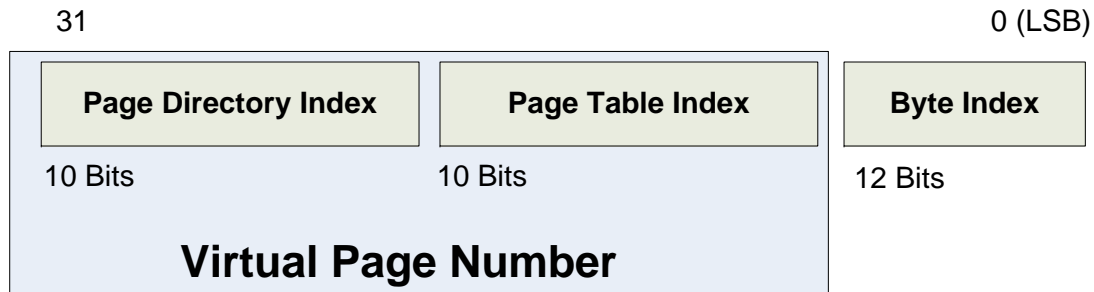
0 (LSB)



CPU References Virtual Memory Address

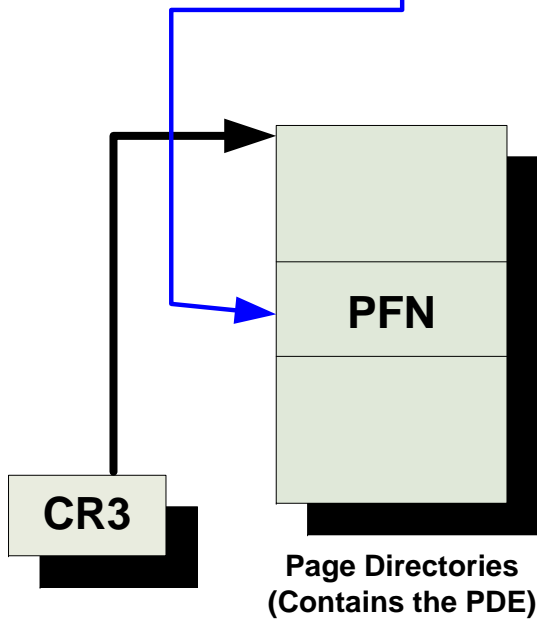
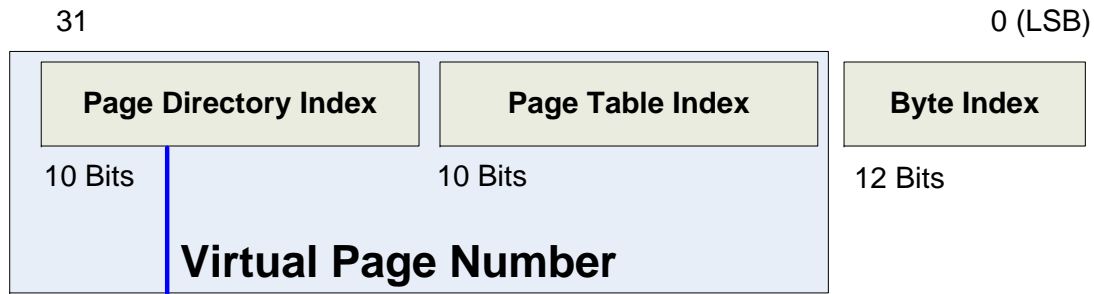


Example Memory Translation





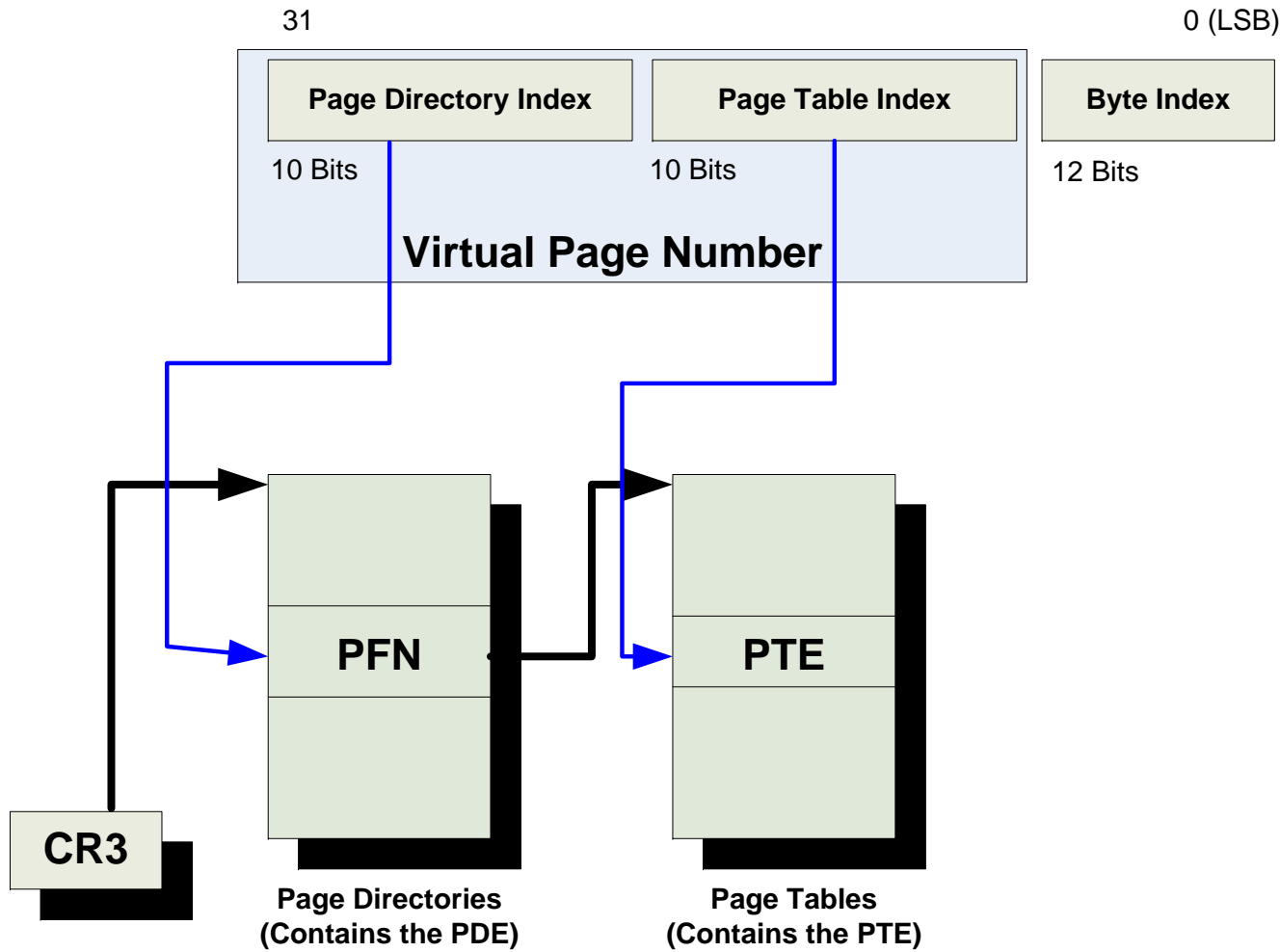
Example Memory Translation



CR3 contains process Page Directories

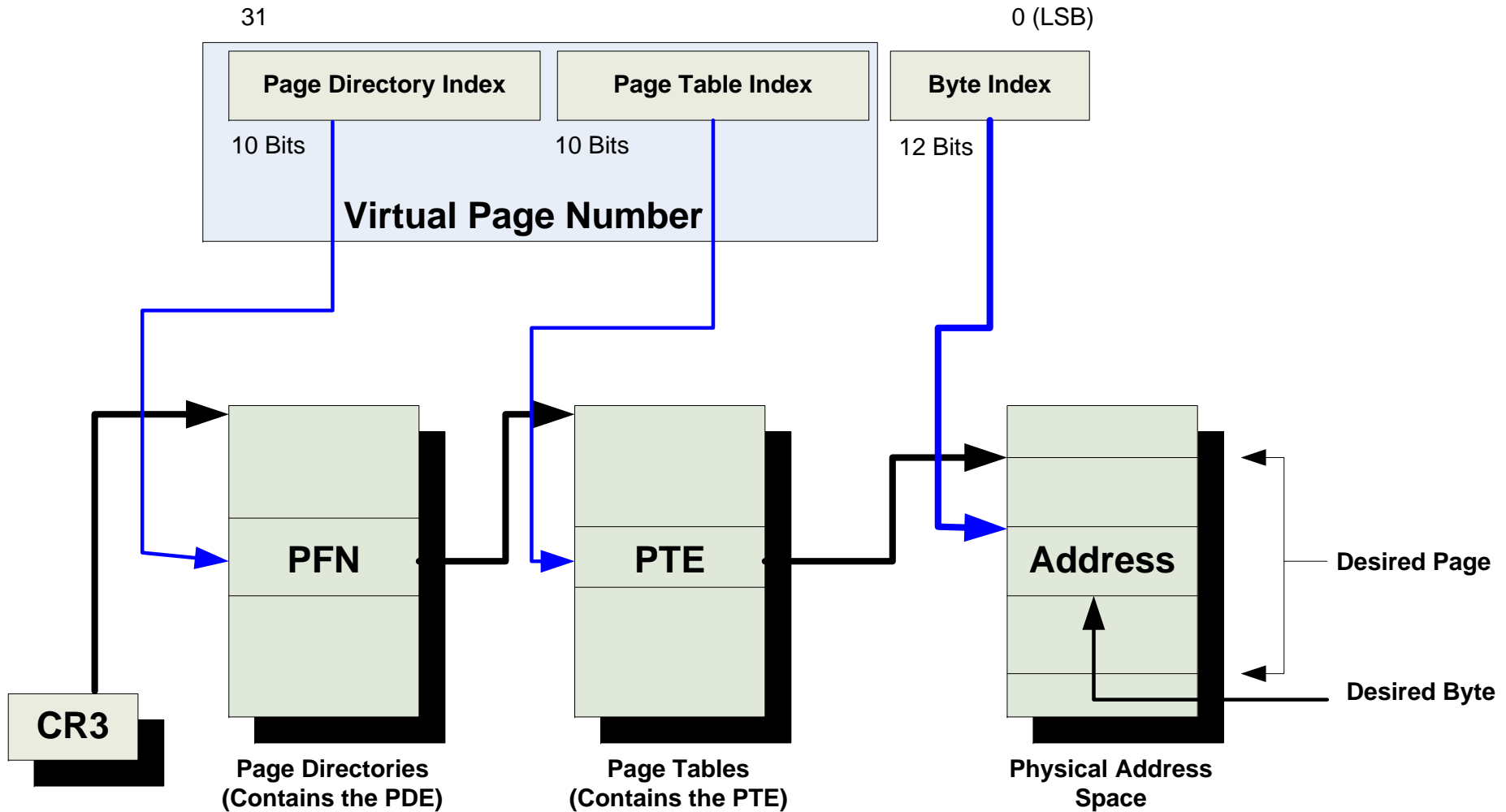


Example Memory Translation





Example Memory Translation





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



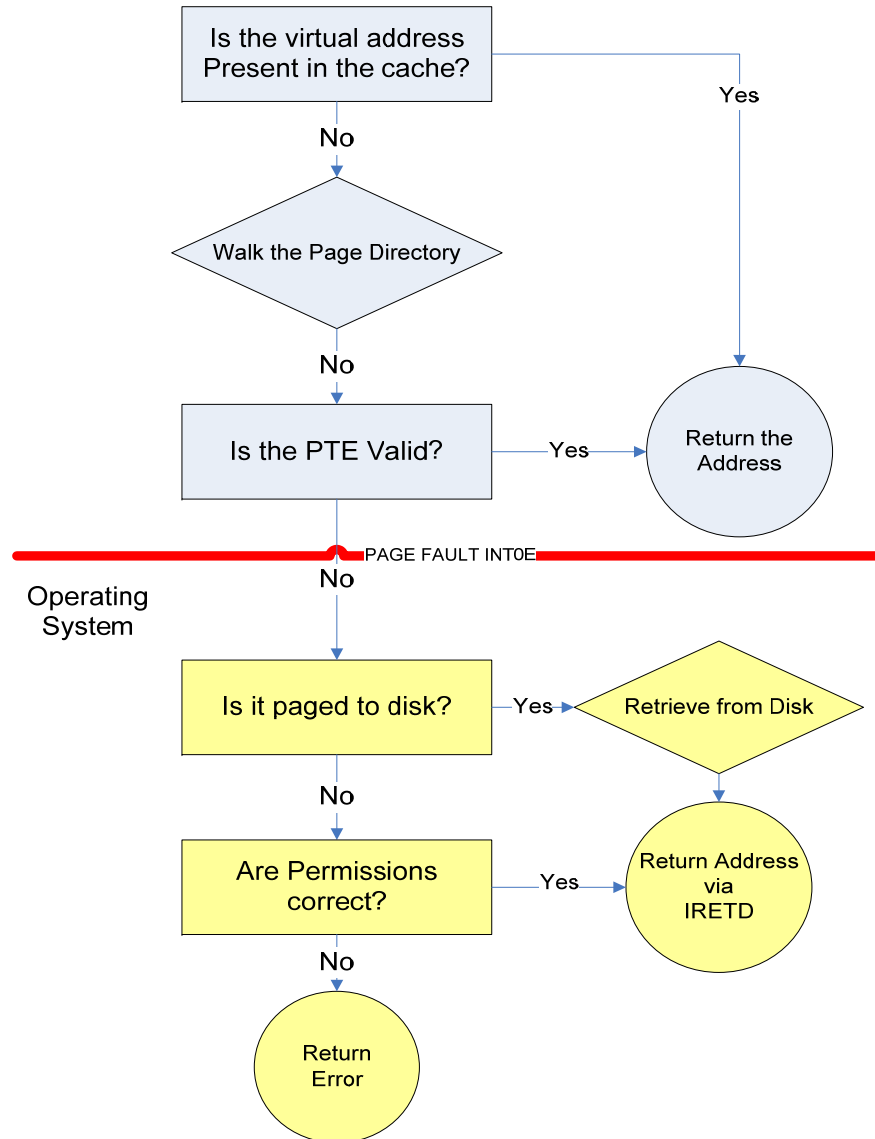
Intel TLB Implementation

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



Offensive Computing - Malware Intelligence

Hardware





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

