# Reverse Engineering on Windows

## A Focus on Malware

Pedram Amini [1] Ero Carrera [2]

[1]TippingPoint DVLabs

[2]zynamics GmbH, VirusTotal

BlackHat US - Las Vegas - 2009

# Outline

1. Outline
   - Background
   - Basic Analysis
   - Advanced Analysis
   - Custom Development

# Outline

2 Introduction

3 VM's and Live Analysis

4 Architecture and OS

5 PE File Format

# Outline

6 Overview of Analysis Tools

7 (Dis)Assembly

8 IDA Pro

9 OllyDbg

# Outline

10 Executable (Un)Packing

11 Anti Reverse Engineering

12 Binary Diffing and Matching

13 Advanced Malware Techniques

# Outline

14 Analysis

15 IDA Python

16 PEFile and PyDasm

17 PaiMei

# Part I

# Background Information

# Outline

2 Introduction
- Introduction
- Malware Analysis
- Questions to Consider

3 VM's and Live Analysis

4 Architecture and OS

5 PE File Format

# Instructors

## Pedram Amini

- Manages the TippingPoint DVLabs Security Research Team
  - http://dvlabs.tippingpoint.com
- Wrote PaiMei RE framework and Sulley fuzzing framework
- Founded OpenRCE.org
- Authored "Fuzzing: Brute Force Vulnerability Discovery"

## Ero Carrera

- Employed by Zynamics GmbH and Chief Research Officer at VirusTotal
- Reverse engineering, malware analysis and automation research
- Developer of *pefile*, *pydot*, *pydasm*, *ida2sql* and *Pythonika*

# What is Reverse Engineering?

## Definition

As it applies to software, reverse engineering is the process of unraveling the inner workings of a given system in the absence of source code.

- Also known as RE or RCE
- As simple as monitoring behavior from a high level
- As complex as reviewing individual instructions within an application
- Example questions reverse engineers seek to answer:
  - What exactly does this software do?
  - Which lines of code are responsible for handling user input?
  - How long can this parameter be before I corrupt the targets memory?

# Who Hires Reverse Engineers?

- Computer security firms, for discovering new software vulnerabilities
- AV companies, for analyzing new viruses and worms
- IPS vendors, for patch analysis
- Tuning shops, for dissecting ECUs
- Competitors, for unveiling your secret sauce
- Data recovery shops, for rebuilding specifications when source code is lost

# Questions for the Class

- Who has past experience with IDA / disassembling?
- Who has past experience with OllyDbg / debugging?
- Who has past experience with Python?
- Who is a member of OpenRCE?
- Who has taken BH training before?
- Any professional reversers in the class?

## Introductions

Please introduce yourselves the first time you interact with one of us or the class. Name, country, company, favorite color, etc...

# Course Objectives

- Learn the key concepts of Portable Executable files
- Create a functional and customized RCE environment
- Familiarize with industry standard tools and practices
- Gain real-world, hands on experience
- Wield the power of RCE to deal with common trickery
- Understand and subvert executable protections
- Apply cutting edge technologies to save analysis time
- Learn how to do all of the above quickly
  - You will be forced to put your mouse to rest ;-)
- The class focus is on malware analysis, which is a vast field
  - We've selected subjects that we feel have real-world application

# What is Malware Analysis?

**Definition**

*Malware Analysis* is the study of unknown code to discover it's purpose and functionality.

- Main goal
  - Glean relevant information in a timely manner
  - Can not stress this point enough
- Unfolding of a story
  - Each analysis technique may add another chapter to the story

# Malware Classifications

- Virus
  - Software which infects other applications and uses them as a spreading medium
- Trojan
  - A malicious application which presents itself as something else
- Worm
  - Code with the ability to spread from computer to computer by means of different network protocols
- Spyware
  - Applications aiming to harvest personal information
- Rootkit
  - Hidden tool/s providing stealth services to its writer

# Malware Components

- Infection
  - Exploit
  - Social engineering
- Payload
  - Destruction
  - Theft
  - Stealth
  - Agent
- Propagation

# A Look at Propagation

- Low level attack vectors
  - Stack overflows
  - Heap overflows
  - Format string vulnerabilities
- Higher level attack vectors
  - Browser exploitation
- Highest level attack vectors
  - Social engineering
  - Mass e-mails

# How is it Spreading?

- If over IP, how fast?
- Is the prand() biased?
- Does it contain a mass mailer?
- Does it spread over network shares?
- Does it spread over P2P or other file transfer mediums?
- Does it spread over an exploit?

# Does it Contain a Backdoor?

- Does it connect to an IRC server?
- Does it bind to any ports?
- Does it retrieve data from the web?
- Does it retrieve data from news groups?

# What Modifications Are Made?

- Does it create/modify/edit/delete any registry keys?
- Does it create/modify/edit/delete any files?
- Does it modify any running processes?
- Does it modify itself?

# Why Was it Written?

- Depending on your goal this question could be the most important of all
- Targeted attack?
- Information theft?
- DDoS network?

# Who Wrote it?

- What compiler was used?
- What date was it created on?
- What stylistic characteristics stand out?

# The Logical Three

- We classify code into one of three logical categories:
- Constant code
  - Code that is continuously running during execution
  - Ex: IP Scanning loops, e-mail harvesting loops
- Reactive code
  - Code that is executed in reaction to an event
  - Ex: An exploitable system was discovered
- Dormant code
  - Code that is designed to execute at a certain date
  - Ex: coordinated DDoS attack

# Outline

2 Introduction

3 VM's and Live Analysis
- Virtual Machines
- Live Analysis

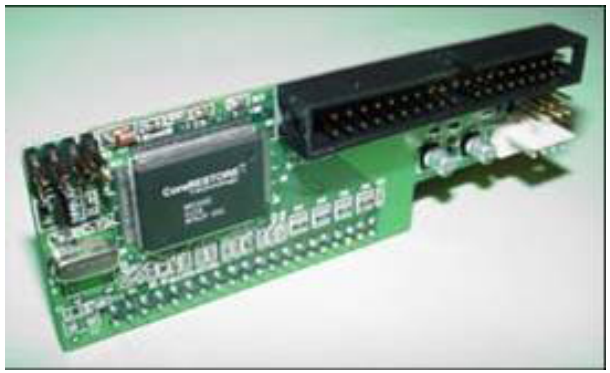
4 Architecture and OS

5 PE File Format

# Virtualization vs. Physical

- Benefits of virtual machine (VM) technology
  - Fast and economical solution for lab environment deployment
  - Network segregation can be virtualized
  - Pre-infection system restore is painless
- Pitfalls of virtual machine (VM) technology
  - Malware can detect the VM and alter behaviour
  - Virtualization *may* have bugs in replication of physical system
- Should have at least one physical system in a malware lab

- CoreRESTORE (www.coreprotect.com)
- Provides hardware level reboot-to-restore functionality



# x86 Virtual Machine Capabilities

- x86 architecture does not meet Popek and Goldberg virtualization requirements
- Virtualization is accomplished via dynamic recompilation (like Java)
- This isn't all that important to know, just know that dynamic recompilation is slower than true self-virtualization
- AMD "Pacifica" and Intel "Vanderpool" have architecture level support for virtualization

[Communications of the ACM, 1974]

# Virtualization vs. Emulation

- VM technologies such as VMware trap all hardware access and simulate the entire motherboard minus the processor
- Emulation technologies such as Bochs completely emulate the processor, hardware devices and memory
- Emulation is slower but provides greater flexibility of control and "safety" in terms of malware "breaking out" of the controlled environment
- Partial emulation can be very helpful in static analysis

# Virtual Machine Technologies

- VMware
  - VMWare server is now free
- Microsoft Virtual PC (Connectix)
  - This is also free
- Microsoft Virtual Server
- Plex86
- Xen
- Parallels

# Emulation Technologies

- Bochs
  - The Bochs instrumentation library is badass, as Ero will show later
- Qemu
- Chris Eagle's IDA x86-emu plug-in (partial emulation)

# VMWare Tips

**Tip**

Use a USB/Firewire disk for increased performance

**Tip**

Save snapshot disk space by installing new software from the network or CD

# What and Why?

**Definition**

Simply put, live analysis constitutes of running suspect code in a "sacrificial" environment while monitoring its activities at a high level

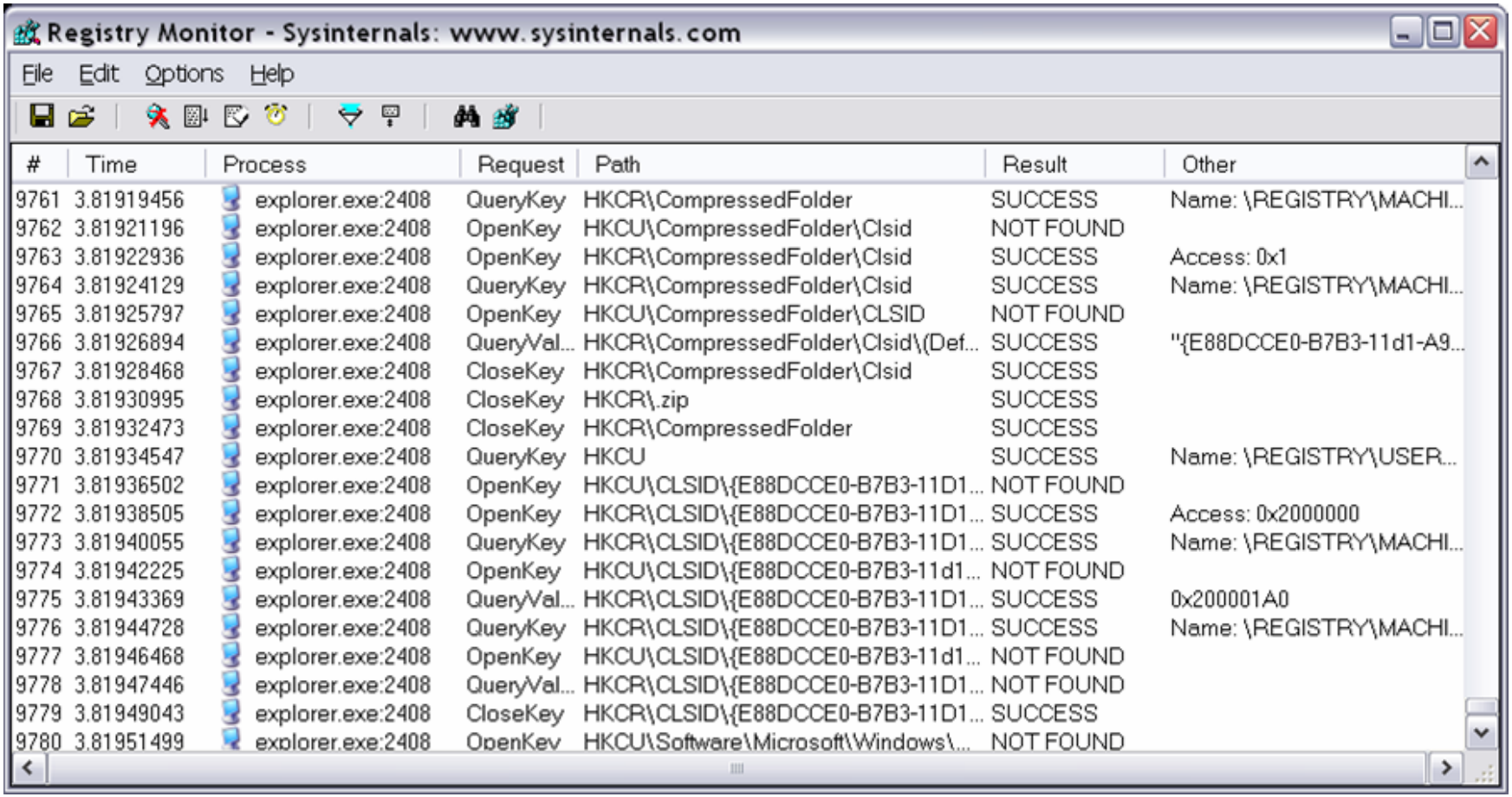
- Live analysis falls well short of reverse engineering
- However, it is still the most frequently used analysis technique
- A good starting point for "unfolding" the story

# SysInternals

- SysInternals is home of Marc Russinovich
- Offers a number of excellent freeware and commercial utilities
- You know Marc from the Sony rootkit debacle
- More recently you may have heard of them due to the Best Buy Geek Squad scandal
- Most recently you may have heard of them because they sold to Microsoft

# SysInternals RegMon

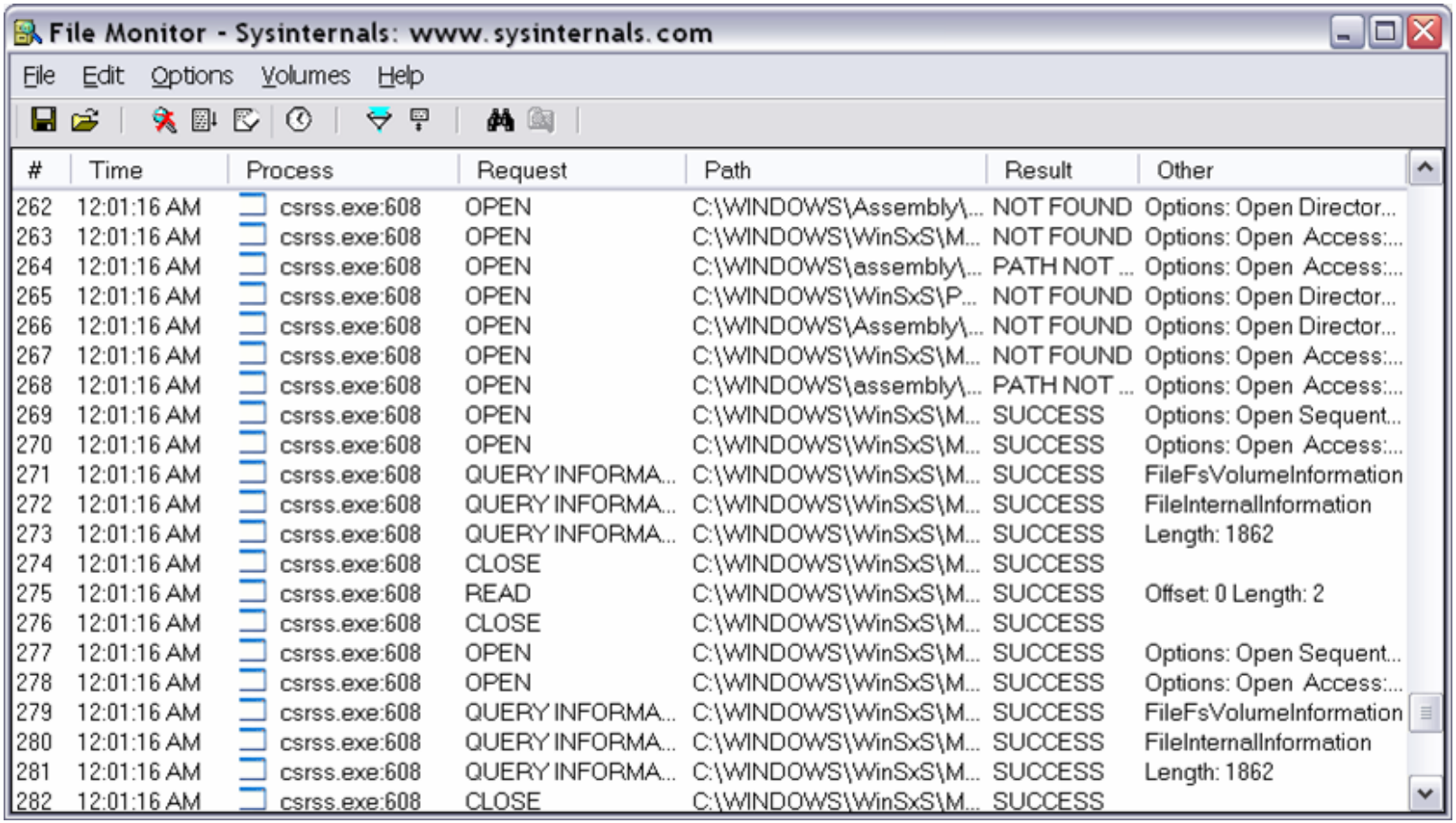
- Monitor all registry access in real time
- Supports filtering




| # | Time | Process | Request | Path | Result | Other |
|---|---|---|---|---|---|---|
| 9761 | 3.81919456 | explorer.exe:2408 | QueryKey | HKCR\CompressedFolder | SUCCESS | Name: \REGISTRY\MACHI... |
| 9762 | 3.81921196 | explorer.exe:2408 | OpenKey | HKCU\CompressedFolder\Clsid | NOT FOUND | |
| 9763 | 3.81922936 | explorer.exe:2408 | OpenKey | HKCR\CompressedFolder\Clsid | SUCCESS | Access: 0x1 |
| 9764 | 3.81924129 | explorer.exe:2408 | QueryKey | HKCR\CompressedFolder\Clsid | SUCCESS | Name: \REGISTRY\MACHI... |
| 9765 | 3.81925797 | explorer.exe:2408 | OpenKey | HKCU\CompressedFolder\CLSID | NOT FOUND | |
| 9766 | 3.81926894 | explorer.exe:2408 | QueryVal... | HKCR\CompressedFolder\Clsid\(Def... | SUCCESS | "{E88DCCE0-B7B3-11d1-A9... |
| 9767 | 3.81928468 | explorer.exe:2408 | CloseKey | HKCR\CompressedFolder\Clsid | SUCCESS | |
| 9768 | 3.81930995 | explorer.exe:2408 | CloseKey | HKCR\.zip | SUCCESS | |
| 9769 | 3.81932473 | explorer.exe:2408 | CloseKey | HKCR\CompressedFolder | SUCCESS | |
| 9770 | 3.81934547 | explorer.exe:2408 | QueryKey | HKCU | SUCCESS | Name: \REGISTRY\USER... |
| 9771 | 3.81936502 | explorer.exe:2408 | OpenKey | HKCU\CLSID\{E88DCCE0-B7B3-11D1... | NOT FOUND | |
| 9772 | 3.81938505 | explorer.exe:2408 | OpenKey | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | SUCCESS | Access: 0x2000000 |
| 9773 | 3.81940055 | explorer.exe:2408 | QueryKey | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | SUCCESS | Name: \REGISTRY\MACHI... |
| 9774 | 3.81942225 | explorer.exe:2408 | OpenKey | HKCU\CLSID\{E88DCCE0-B7B3-11d1... | NOT FOUND | |
| 9775 | 3.81943369 | explorer.exe:2408 | QueryVal... | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | SUCCESS | 0x200001A0 |
| 9776 | 3.81944728 | explorer.exe:2408 | QueryKey | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | SUCCESS | Name: \REGISTRY\MACHI... |
| 9777 | 3.81946468 | explorer.exe:2408 | OpenKey | HKCU\CLSID\{E88DCCE0-B7B3-11d1... | NOT FOUND | |
| 9778 | 3.81947446 | explorer.exe:2408 | QueryVal... | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | NOT FOUND | |
| 9779 | 3.81949043 | explorer.exe:2408 | CloseKey | HKCR\CLSID\{E88DCCE0-B7B3-11D1... | SUCCESS | |
| 9780 | 3.81951499 | explorer.exe:2408 | OpenKey | HKCU\Software\Microsoft\Windows\... | NOT FOUND | |

# SysInternals FileMon

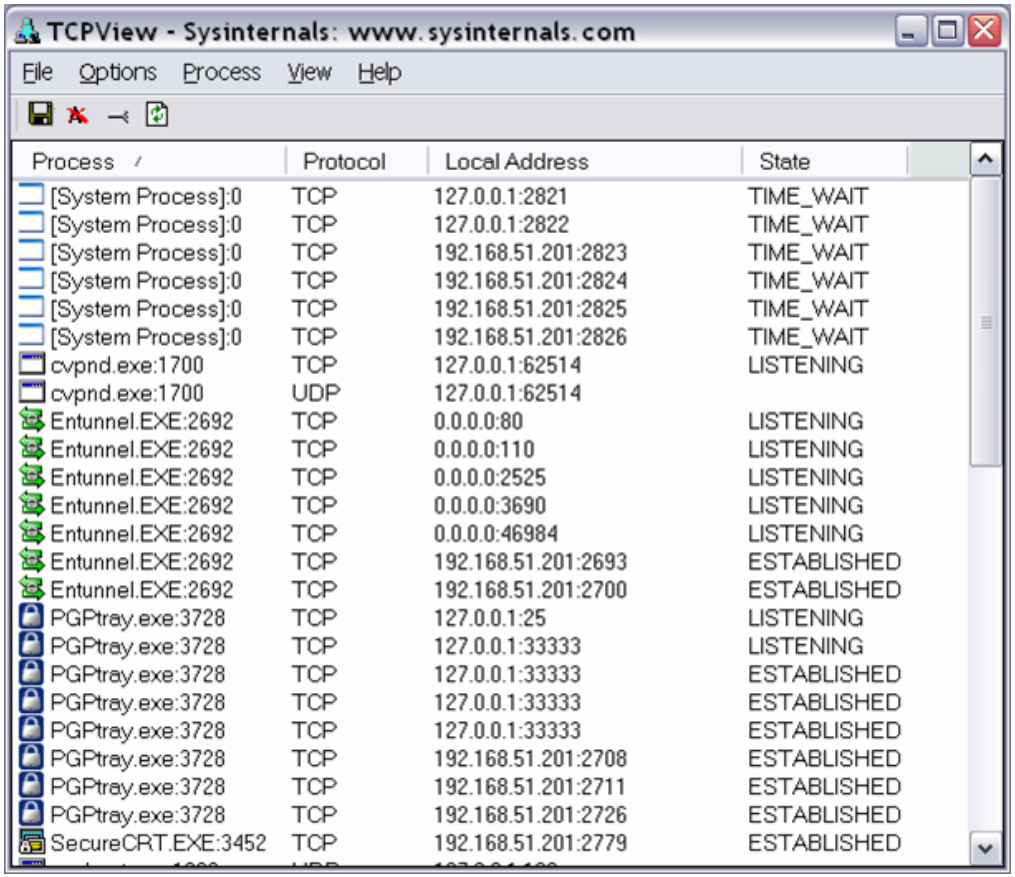
- Monitor all file access in real time
- Supports filtering




| # | Time | Process | Request | Path | Result | Other |
|---|---|---|---|---|---|---|
| 262 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\Assembly\... | NOT FOUND | Options: Open Director... |
| 263 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | NOT FOUND | Options: Open Access:... |
| 264 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\assembly\... | PATH NOT ... | Options: Open Access:... |
| 265 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\P... | NOT FOUND | Options: Open Director... |
| 266 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\Assembly\... | NOT FOUND | Options: Open Director... |
| 267 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | NOT FOUND | Options: Open Access:... |
| 268 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\assembly\... | PATH NOT ... | Options: Open Access:... |
| 269 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | SUCCESS | Options: Open Sequent... |
| 270 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | SUCCESS | Options: Open Access:... |
| 271 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | FileFsVolumeInformation |
| 272 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | FileInternalInformation |
| 273 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | Length: 1862 |
| 274 | 12:01:16 AM | csrss.exe:608 | CLOSE | C:\WINDOWS\WinSxS\M... | SUCCESS | |
| 275 | 12:01:16 AM | csrss.exe:608 | READ | C:\WINDOWS\WinSxS\M... | SUCCESS | Offset: 0 Length: 2 |
| 276 | 12:01:16 AM | csrss.exe:608 | CLOSE | C:\WINDOWS\WinSxS\M... | SUCCESS | |
| 277 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | SUCCESS | Options: Open Sequent... |
| 278 | 12:01:16 AM | csrss.exe:608 | OPEN | C:\WINDOWS\WinSxS\M... | SUCCESS | Options: Open Access:... |
| 279 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | FileFsVolumeInformation |
| 280 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | FileInternalInformation |
| 281 | 12:01:16 AM | csrss.exe:608 | QUERY INFORMA... | C:\WINDOWS\WinSxS\M... | SUCCESS | Length: 1862 |
| 282 | 12:01:16 AM | csrss.exe:608 | CLOSE | C:\WINDOWS\WinSxS\M... | SUCCESS | |

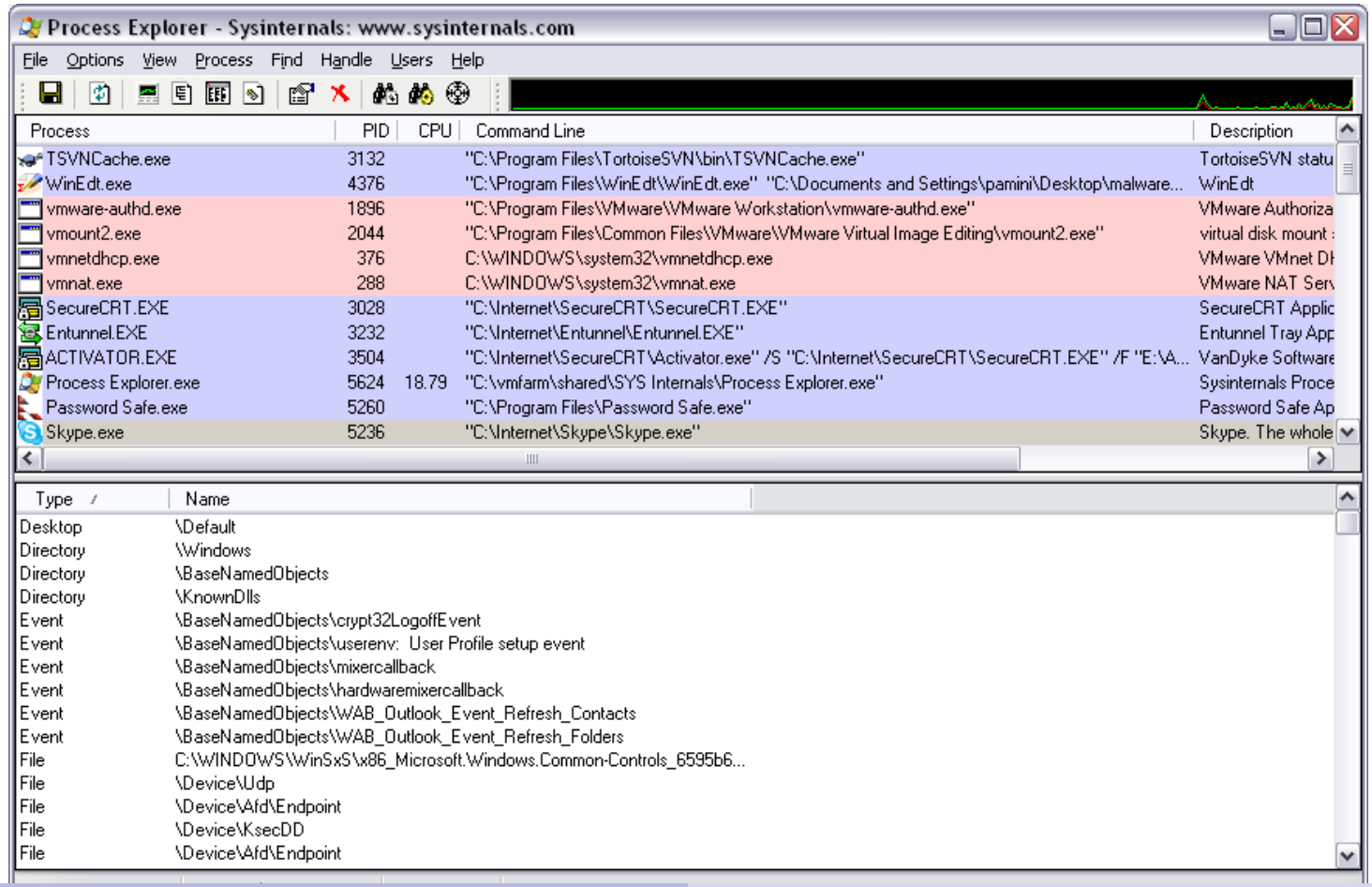
# SysInternals TCPView

- Monitor all per-process socket endpoints in real time
  - ie: Determine what ports a specific process is bound to
- Supports filtering



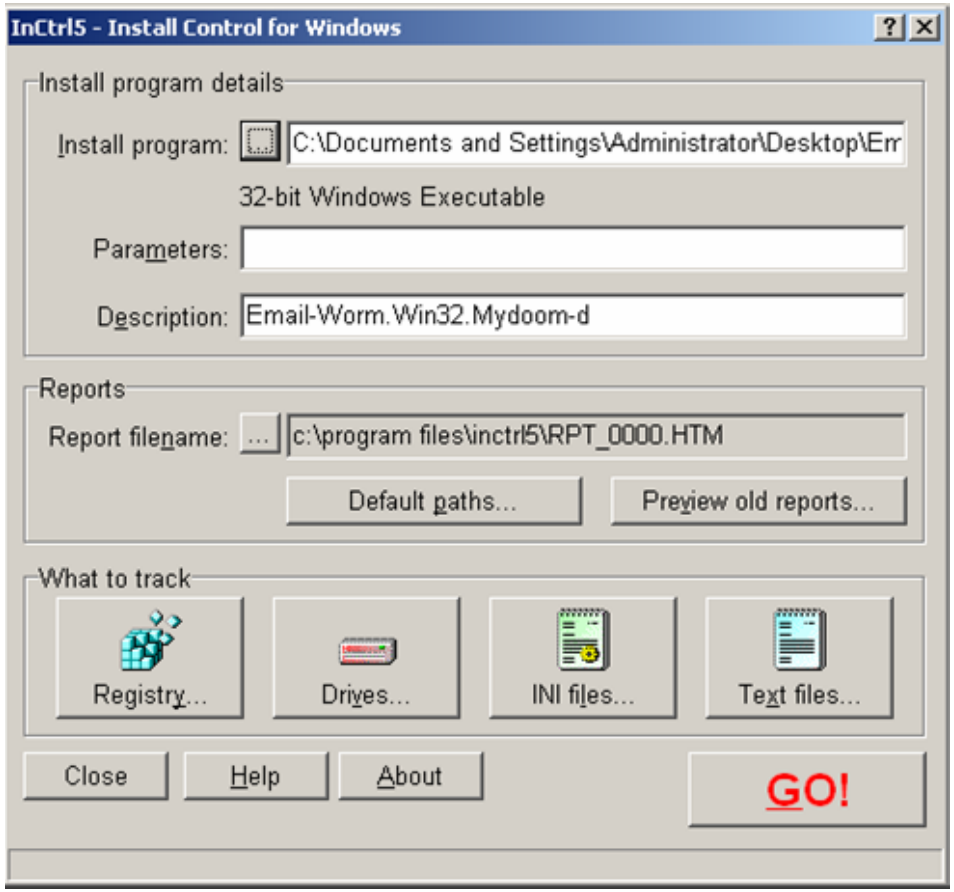
# SysInternals Process Explorer

- Task manager on steroids
- Exposes a plethora of useful information



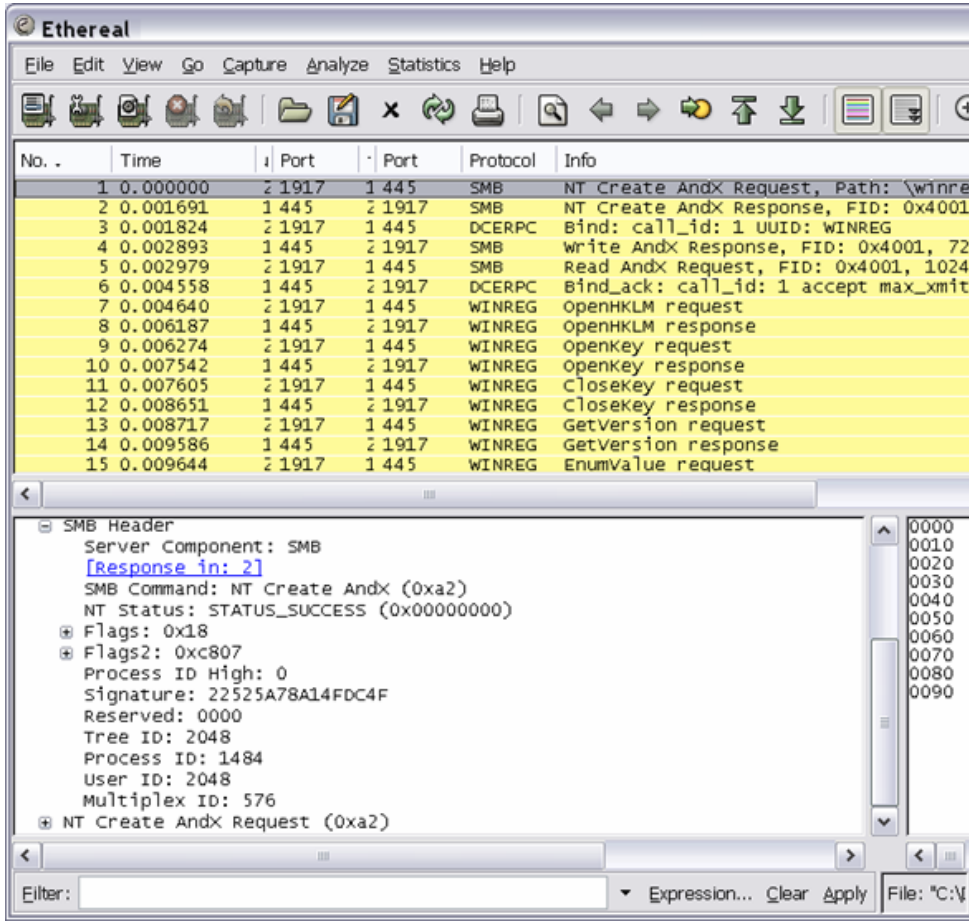
# InCtrl

- Originally designed to monitor changes made by installers
- Applies well to malware
- Takes system "snapshots" pre/post execution
- Generates HTML report of monitors changes



# Wire Shark / Ethereal

- Formerly known as Ethereal
- Wonderful and free network sniffer
- Can decipher a number of protocols
- Also vulnerable, very vulnerable



# Dave Zimmer's Tools

- Who is Dave Zimmer?
  - A VB coding machine
  - His tools are currently available from http://labs.idefense.com
- Malcode Analyst Pack
  - FakeDNS, IDCDumpFix, Mailpot, SCLog, ShellExt, Sniffhit, SocketTool
- Multipot
  - Emulation based honeypot
- SysAnalyzer
  - Like InCtrl, but specifically designed for malware

# Outline

2 Introduction

3 VM's and Live Analysis

4 Architecture and OS
- x86 Architecture
- Microsoft Windows OS

5 PE File Format

# A 50,000ft View of the x86 Architecture

The main components of the x86 include the CPU, memory, disk and registers. The CPU operates on a fetch, decode and execute cycle.

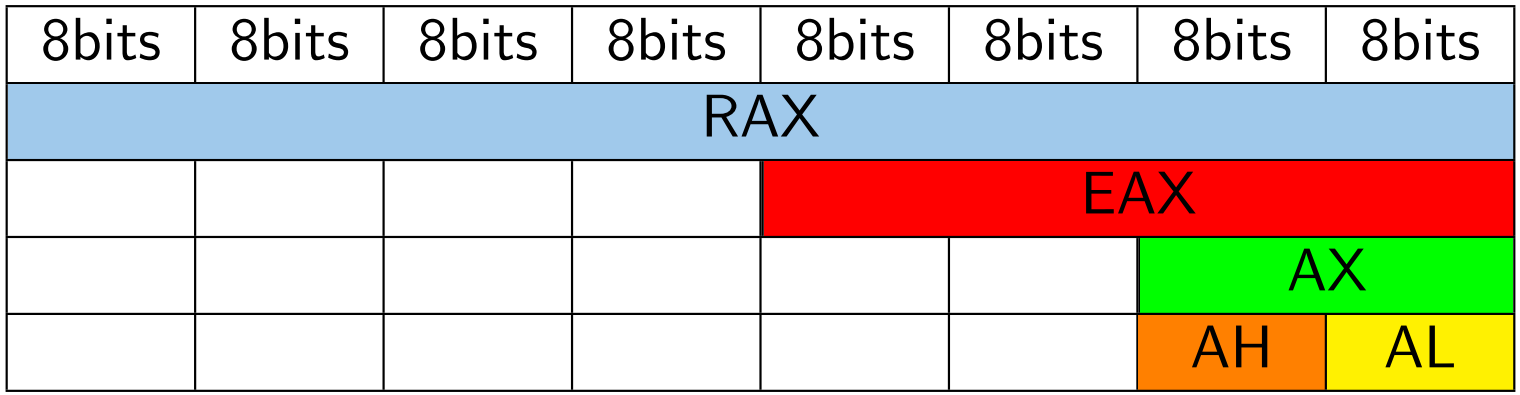
- Applications are implemented with individual assembly instructions
- Individual instructions can:
  - Reference / manipulate memory
  - Perform calculations
  - Determine the next instruction to execute
- A *single* line of high level code usually translates to *multiple* instructions
- The x86 platform is considered CISC vs RISC

# Introduction to Registers

- There are 8 general purpose registers on 32-bit x86 platforms
- Each register is 32 bits long
- Register access is lowest latency (vs. memory or disk)
- As registers are valuable, compilers try to be intelligent with their usage
  - This factor comes into play during reversing

# x86 General Purpose Registers

- EAX
  - Volatile, accumulator, return value of functions
- EBX
  - Non-volatile, base (indirect addressing)
- ECX
  - Volatile, Counter, loop instructions
- EDX
  - Volatile



# General Purpose Registers Continued

- ESI
  - Non-volatile, string source
- EDI
  - Non-volatile, string destination
- ESP / EBP
  - Stack pointer (volatile) / frame pointer (non-volatile)
- EIP
  - Instruction pointer

# The Stack

**Definition**

The stack is an abstract data structure supported by a combination of hardware and software features.

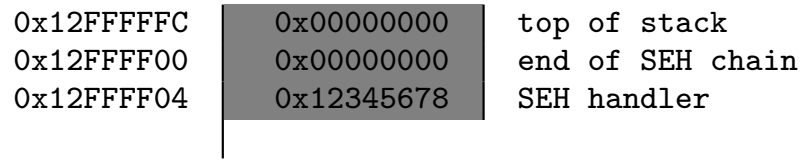
- Stack operations are Last In First Out (LIFO)
- Think of it as a stack of dishes
- The `PUSH` instruction places a 32-bit value on the stack
- The `POP` instruction removes a 32-bit value from the stack
- The stack is used to pass parameters to functions
- The stack is used to maintain call chain state
- In Windows, the stack is used to store the Structured Exception Handling (SEH) chain

# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

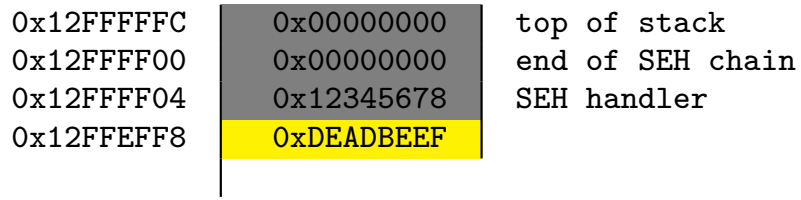


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

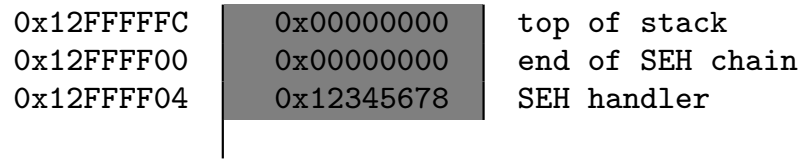


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

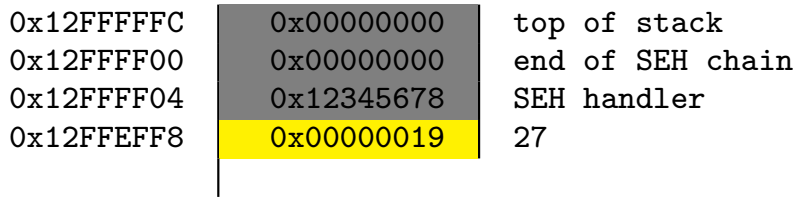


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

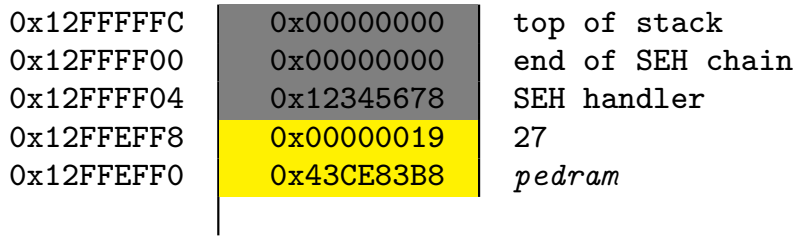


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

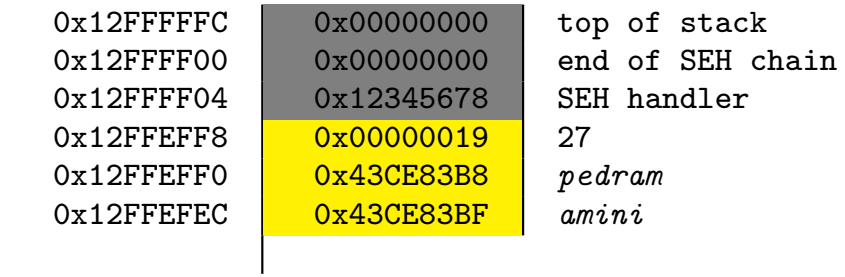


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

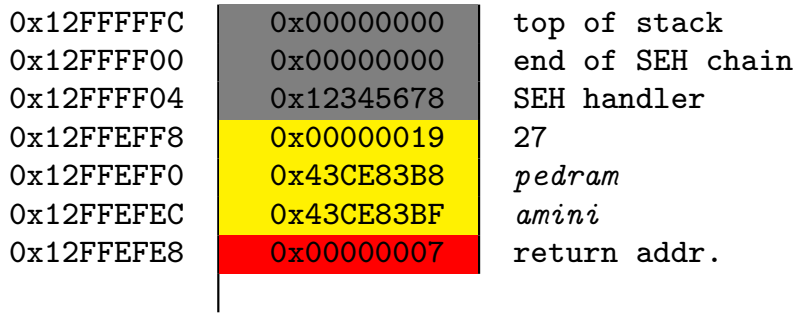


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

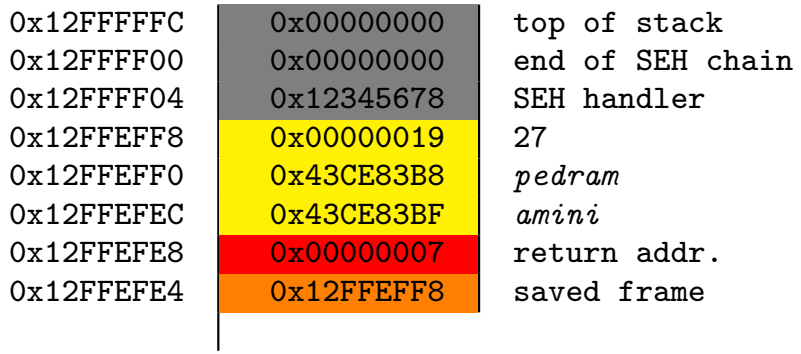


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

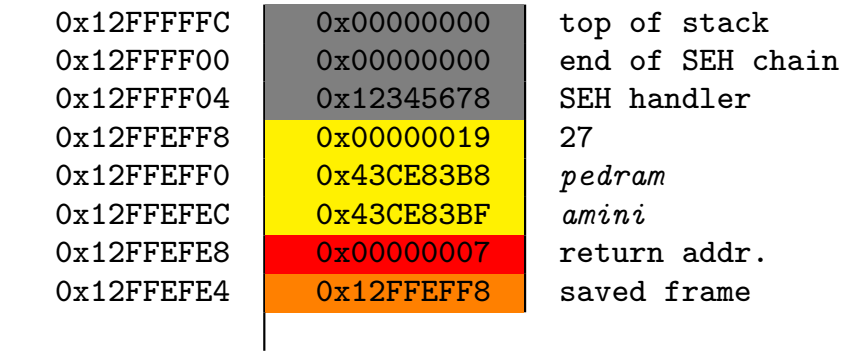


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

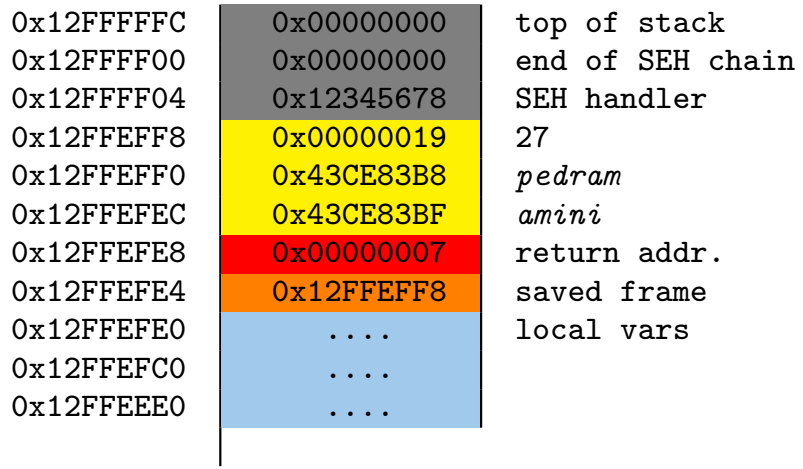


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

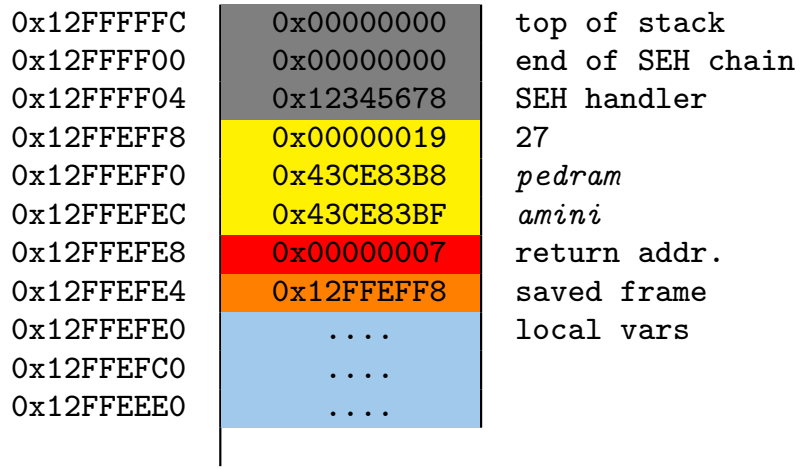


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

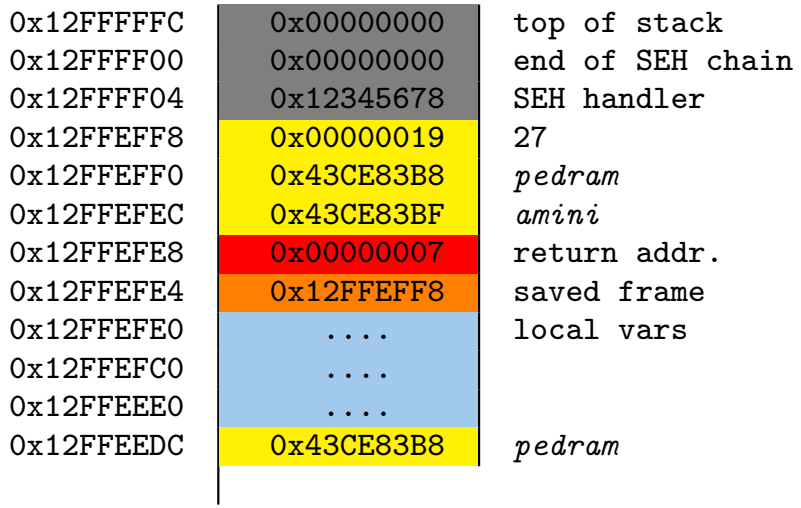


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

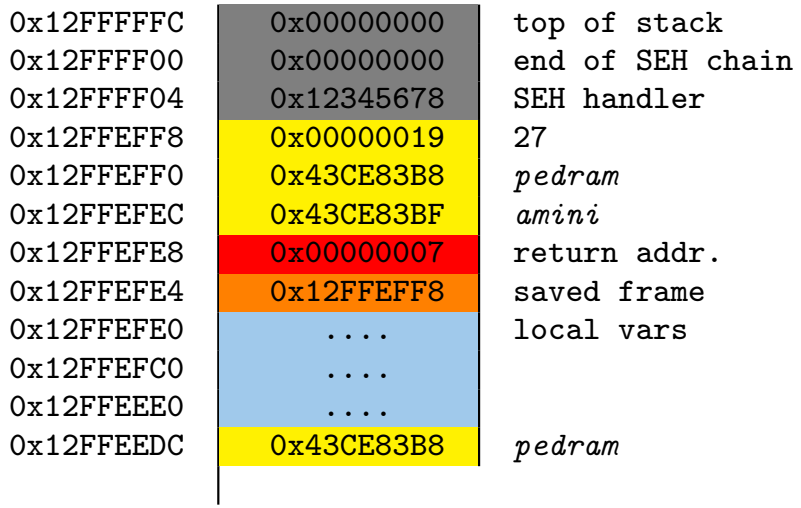


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

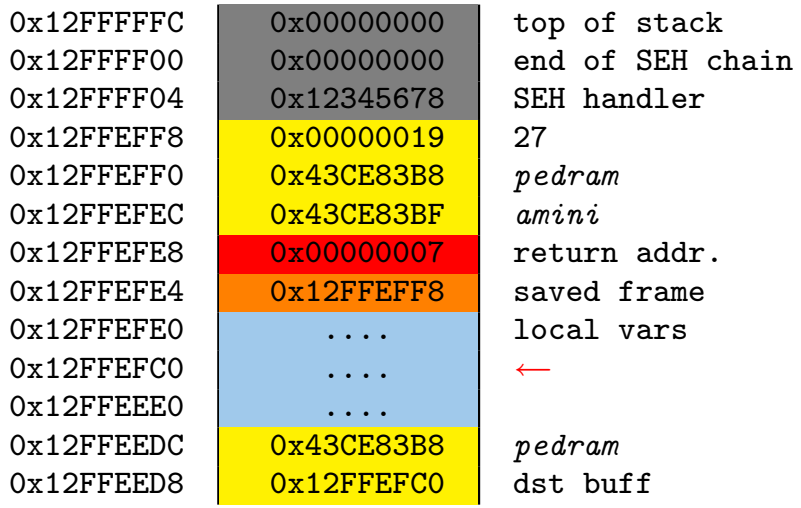


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

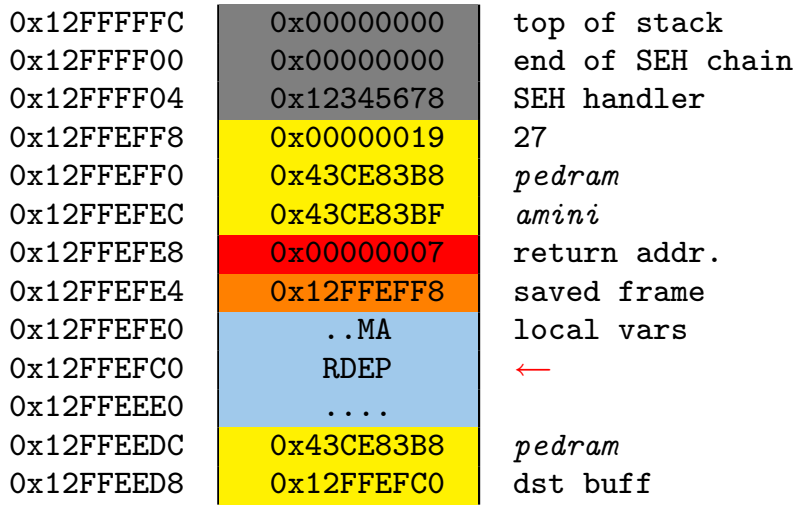


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

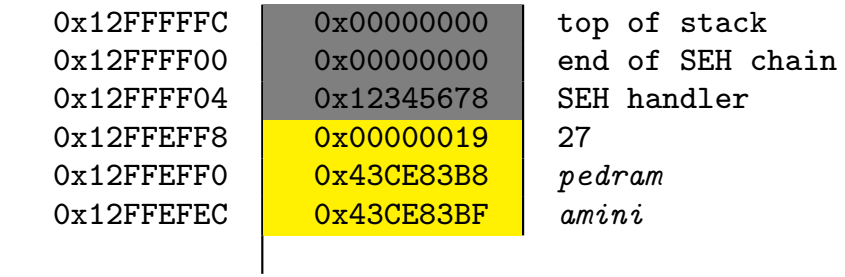


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```

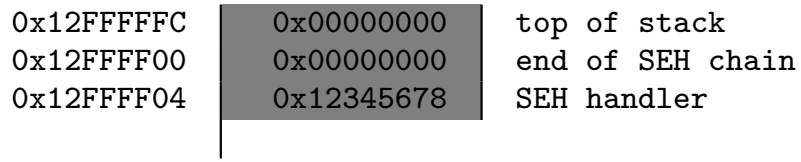


# The Stack: Illustrated

```
0x43ce83b8: "pedram"
0x43ce83bf: "amini"

main:
    0x01 push 0xdeadbeef
    0x02 pop eax
    0x03 push 27
    0x04 push 0x43ce83b8
    0x05 push 0x43ce83bf
    0x06 call copy_name
    0x07 add esp, 0xc
    ...

copy_name:
    0x20 push ebp
    0x21 mov ebp, esp
    0x22 sub esp, 0x100
    0x23 lea eax, [ebp+0xc]
    0x24 push eax
    0x25 lea eax, [ebp-0x20]
    0x26 push eax
    0x27 call strcpy
    ...
    ret
```



# Microsoft Windows Memory Layout

## Memory Space

# Microsoft Windows Memory Layout

## Memory Space

$2^{32}$

# Microsoft Windows Memory Layout

## Memory Space

$2^{32} = 4,294,967,296$ bytes

# Microsoft Windows Memory Layout

## Memory Space

$2^{32} = 4,294,967,296$ bytes
$/1024^3$

# Microsoft Windows Memory Layout

## Memory Space

$2^{32} = 4,294,967,296$ bytes
$/1024^3 = 4$ gigabytes

# Microsoft Windows Memory Layout

## Memory Space

$2^{32} = 4,294,967,296$ bytes
$/1024^3 =$ 4 gigabytes

## Memory Separation

The NT based platforms typically split the available 4 gigabytes of addressable memory into two halves; kernel and user.

# Microsoft Windows Memory Layout

## Memory Space

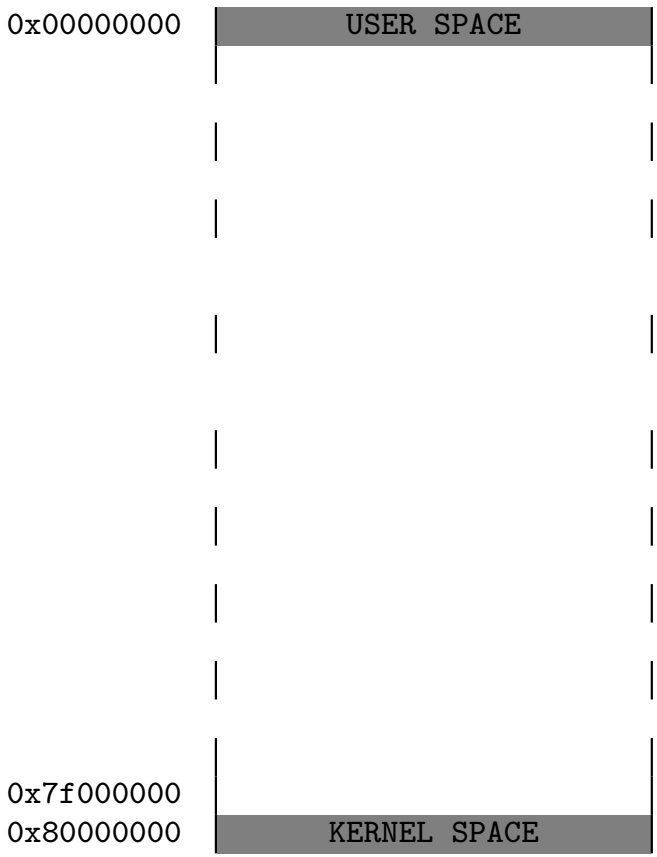
$2^{32} = 4,294,967,296$ bytes
$/1024^3 = 4$ gigabytes

## Memory Separation

The NT based platforms typically split the available 4 gigabytes of addressable memory into two halves; kernel and user.

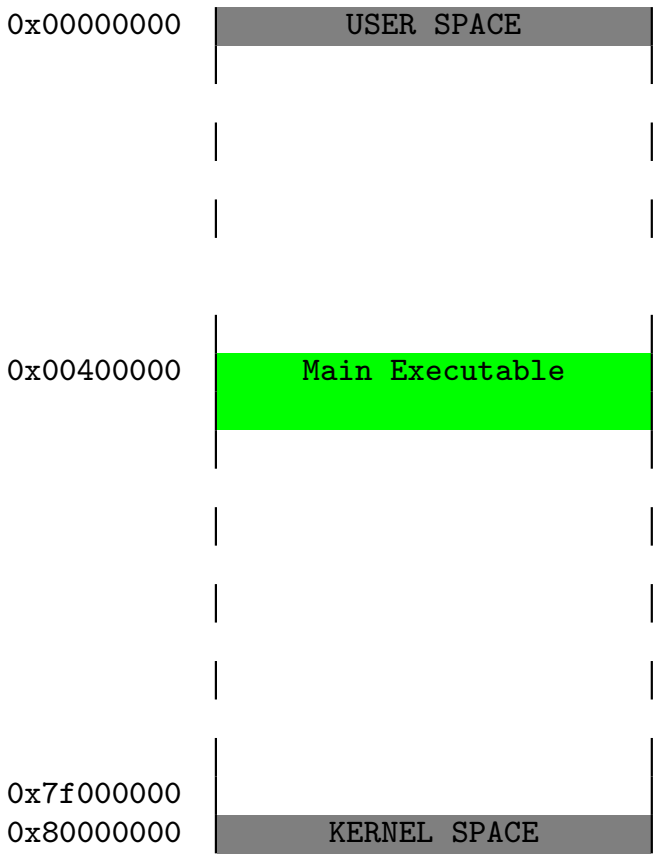
- Each process "sees" it's own 2 gigabytes of virtual memory
  - This is possible thanks to memory paging
  - Processes can not "break out" of their memory space
- The virtual address space is broken in pages
  - Pages are typically 4,096 bytes in size
  - Memory permissions are applied at the page level

# Typical Memory Layout Diagram




## On Process Launch

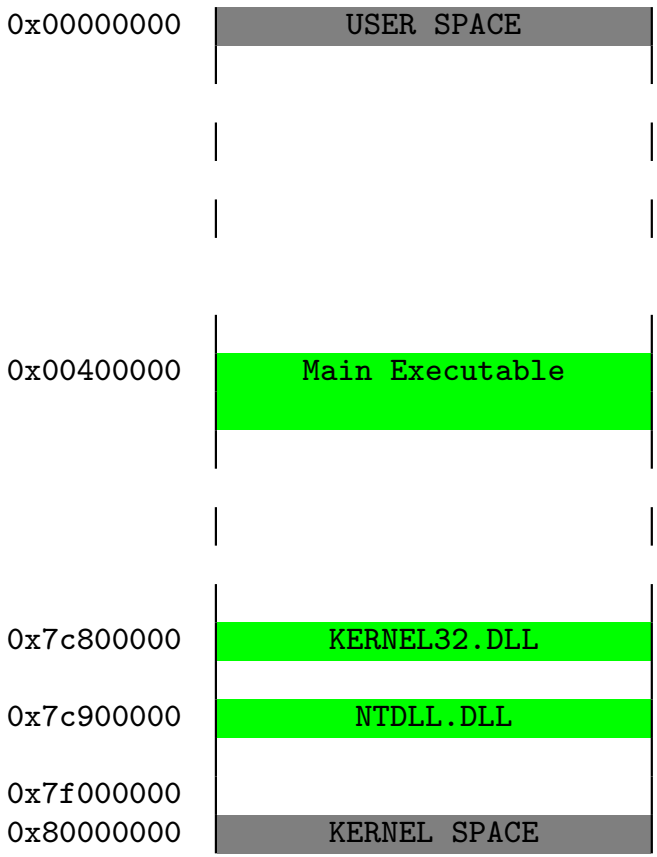
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory

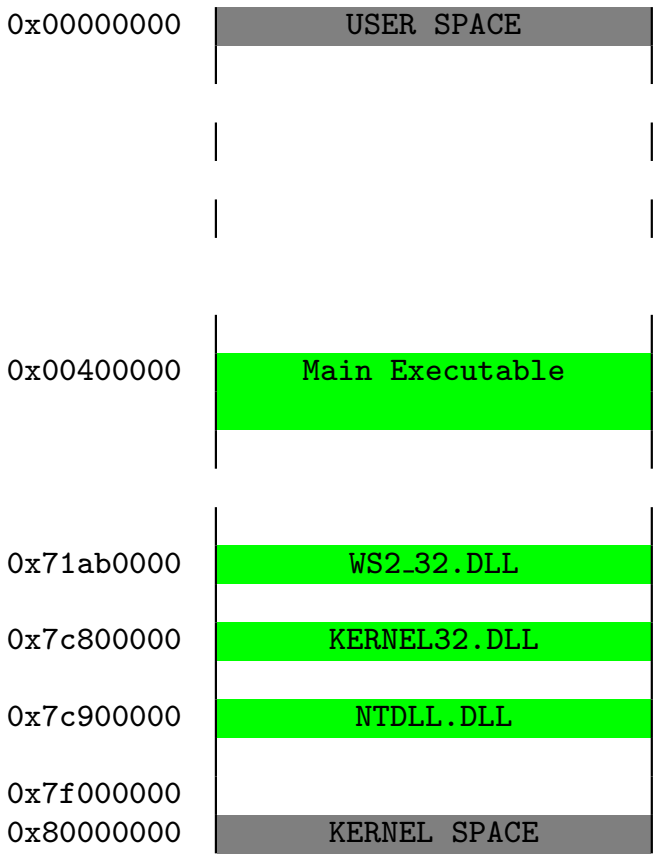
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory

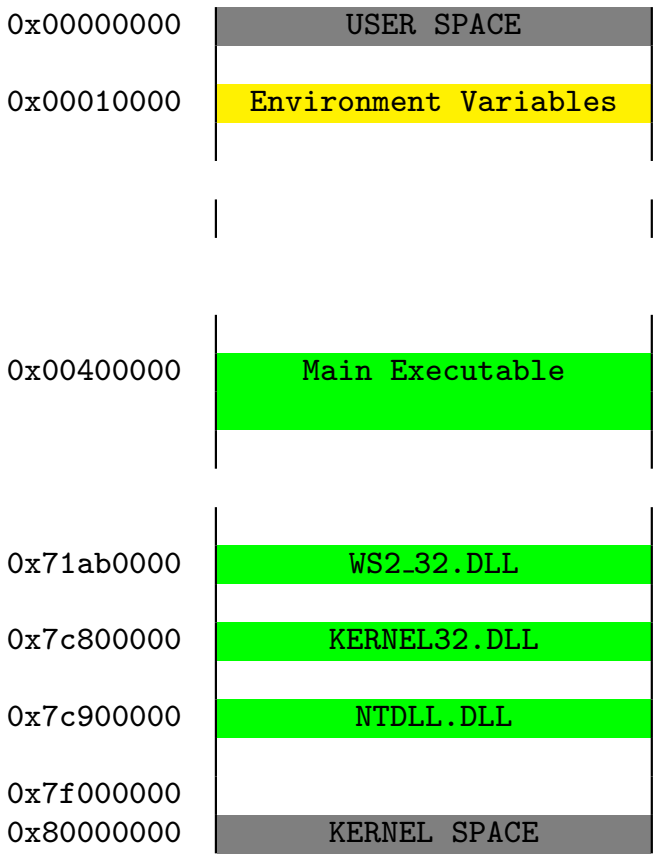
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory

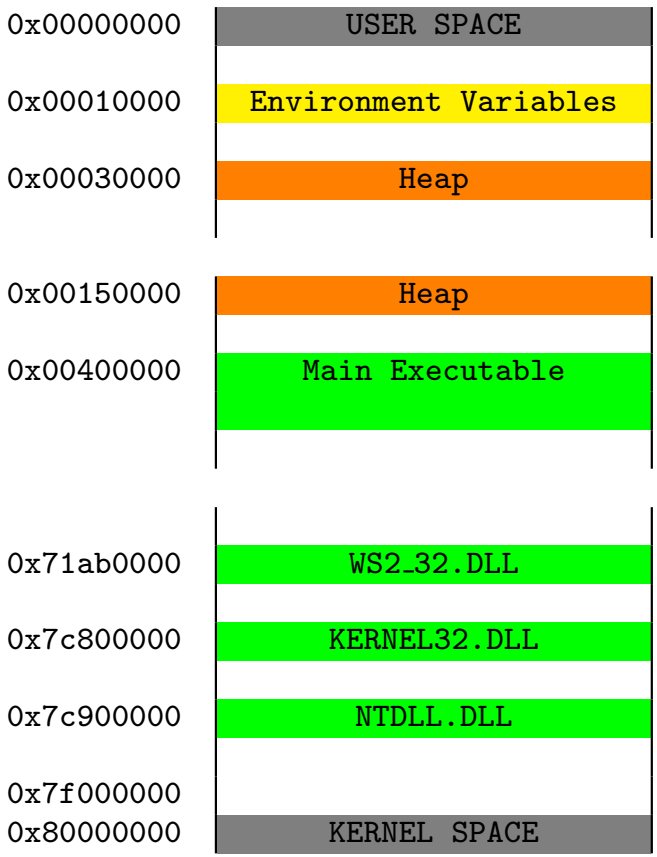
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory
- Environment variables mapped into memory

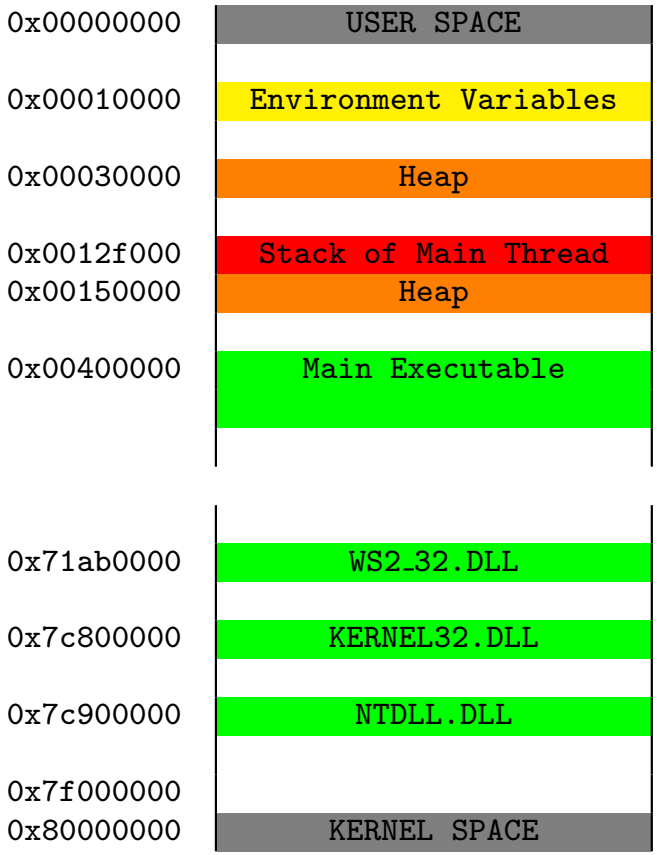
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory
- Environment variables mapped into memory
- Process heaps initialized

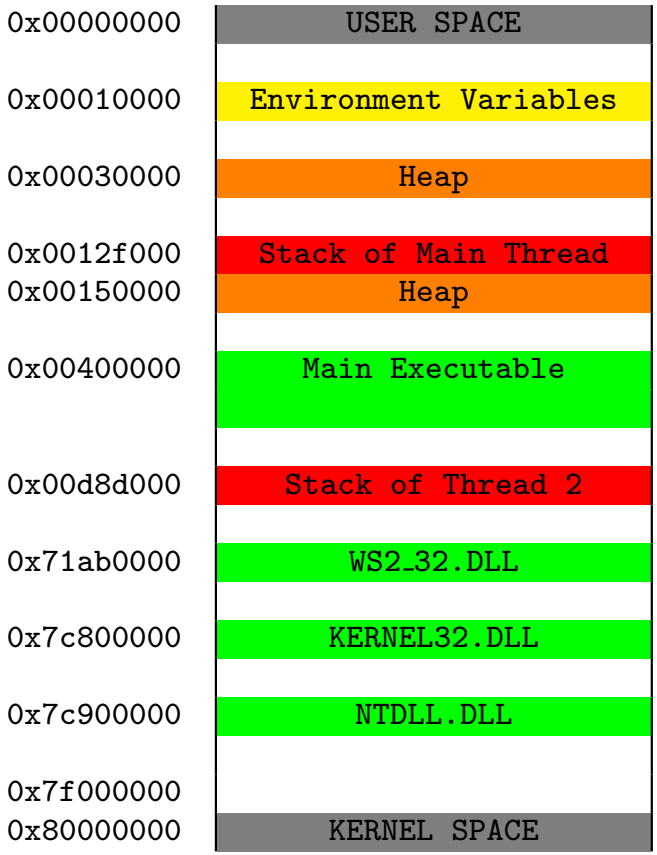
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory
- Environment variables mapped into memory
- Process heaps initialized
- Process stacks initialized

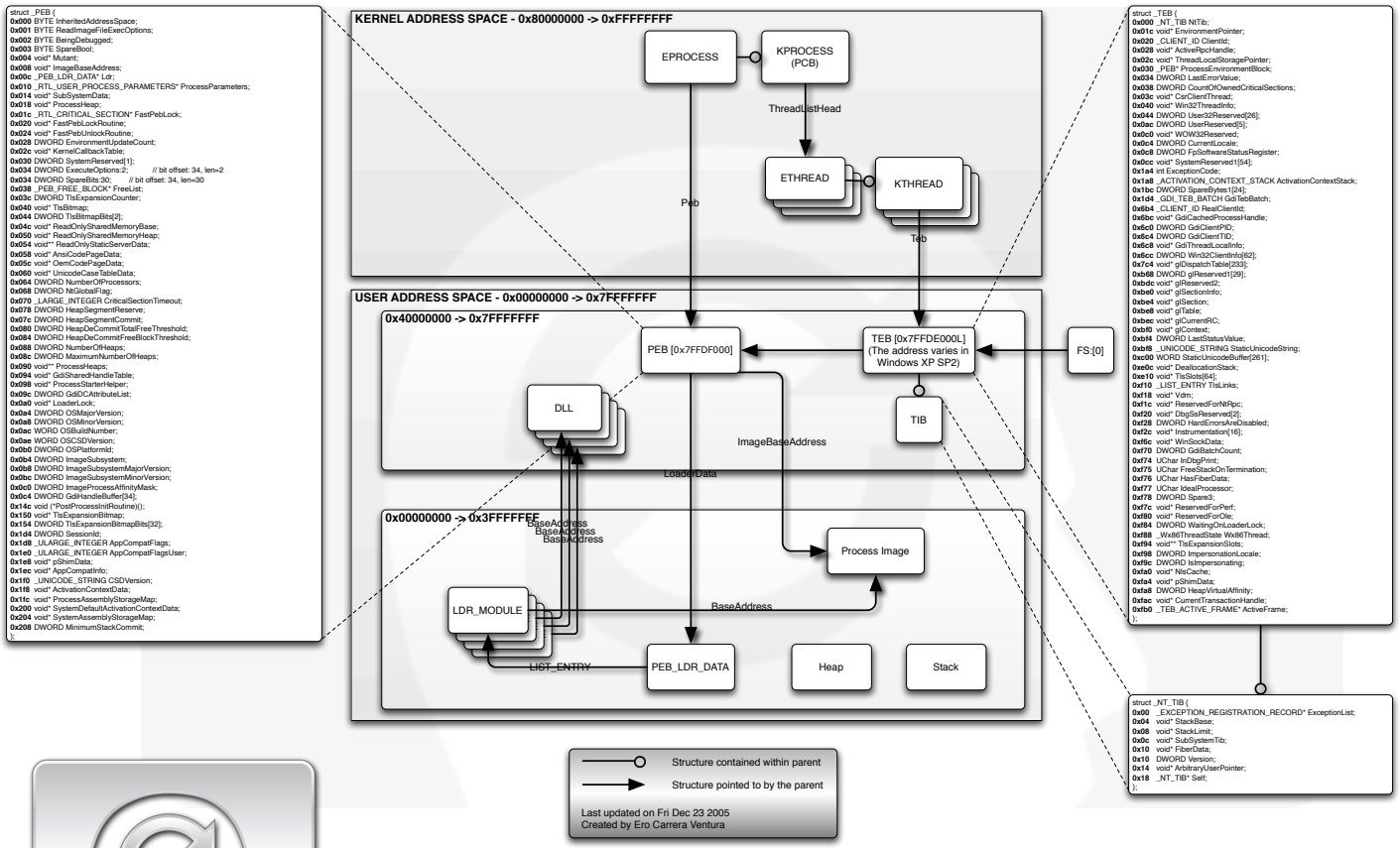
# Typical Memory Layout Diagram




## On Process Launch

- Main image loaded into memory
- Required DLLs loaded into memory
- Environment variables mapped into memory
- Process heaps initialized
- Process stacks initialized

# The Memory Layout Diagram







**Memory Layout for Windows XP**

References:

NTIllusion: A portable Win32 userland rootkit, Kdm; Phrack 62, Volume 0x0b, Issue 0x3e, Phile #0x0c of 0x10

Inside Microsoft® Windows® 2000, Third Edition [Chapter 6: Processes, Threads, and Jobs]
http://www.microsoft.com/mspress/books/sampchap/4354.asp

# The Heap

- Static allocations mostly originate from the stack
- The heap is the source of dynamic memory allocation
- There are multiple heap implementations
- Memory for the heap is allocated from user space
- The heap is essentially a number of doubly linked lists, organized by size
- Allocated blocks are removed from the free lists
- De-allocated blocks are placed back into the free lists

# SEH: Structured Exception Handling

- Exception handlers are simply a registered function to refer to when "something bad happens"
- You can for example register an exception handler to handle attempts to divide by zero
- You can register more then one exception handler
- The chain of exception handlers are stored on the stack
- When an exception occurs, the chain is walked to find an appropriate handler
- There is a catch-all handler, that's what generates the "Windows has detected a general protection fault" dialog

# Exercise

- OllyDbg
  - Attach to or load calc.exe or notepad.exe
  - Hit 'M' and verify memory layout
- LordPE
  - Load calc.exe or notepad.exe
  - Explore the various PE fields and directories
  - We'll walk through the PE file format in depth in a minute

# Outline

2 Introduction

3 VM's and Live Analysis

4 Architecture and OS

5 PE File Format
- Overview and Headers
- Interactive Walkthrough
- Import/Export Address Tables
- Updated PE32+ and Usage Examples

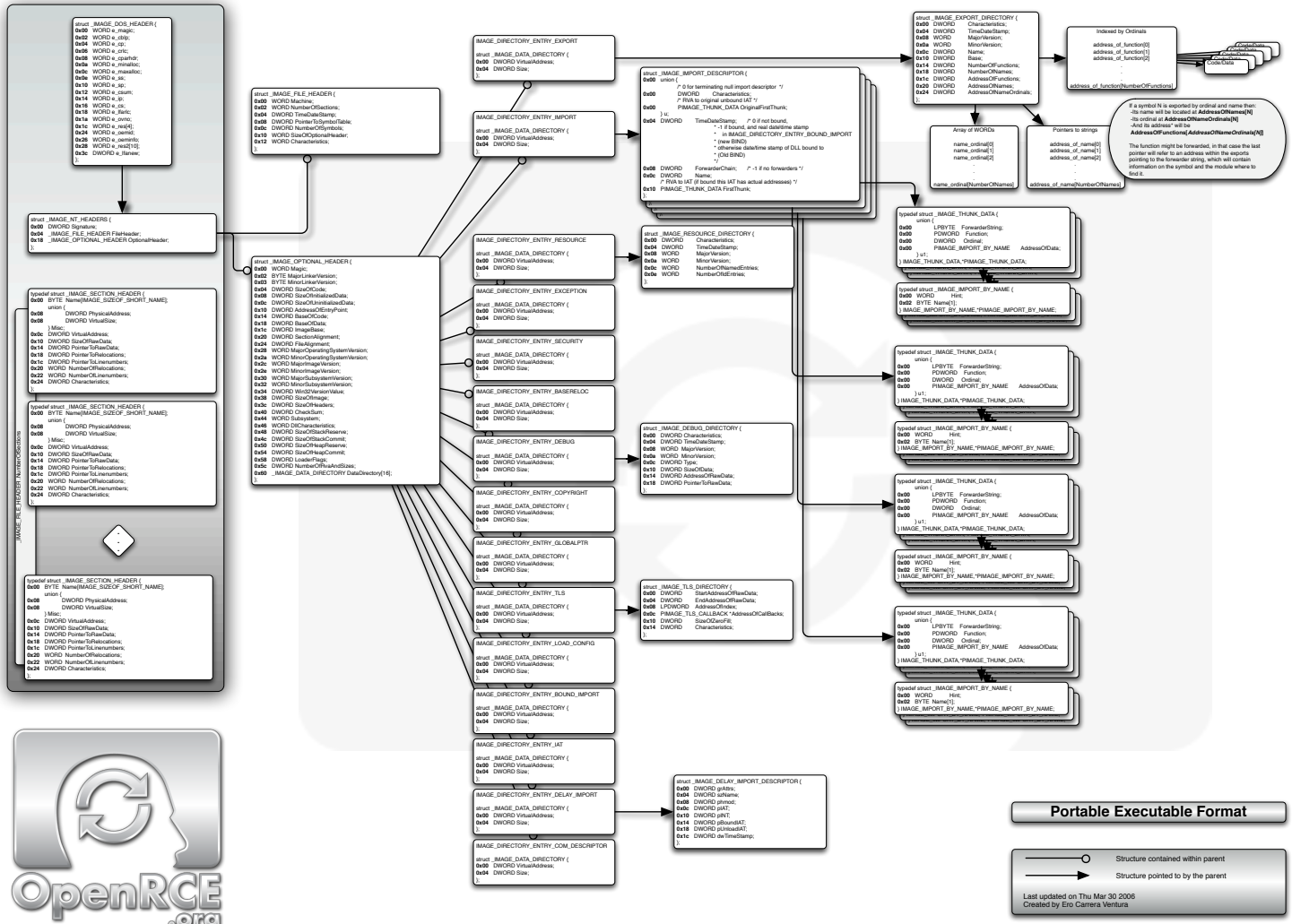
# Portable Executable File Format

## History

Microsoft based the PE file format on the Unix COFF file format. As such it is sometimes referred to as PE/COFF.

- *Portable* in PE means
  - Supports both 32-bit and 64-bit
  - Supports MIPS, DEC Alpha, PowerPC and ARM
- Files with .exe extensions are PE
- Dynamic Link Libraries (DLLs) are PE

# PE Format Layout

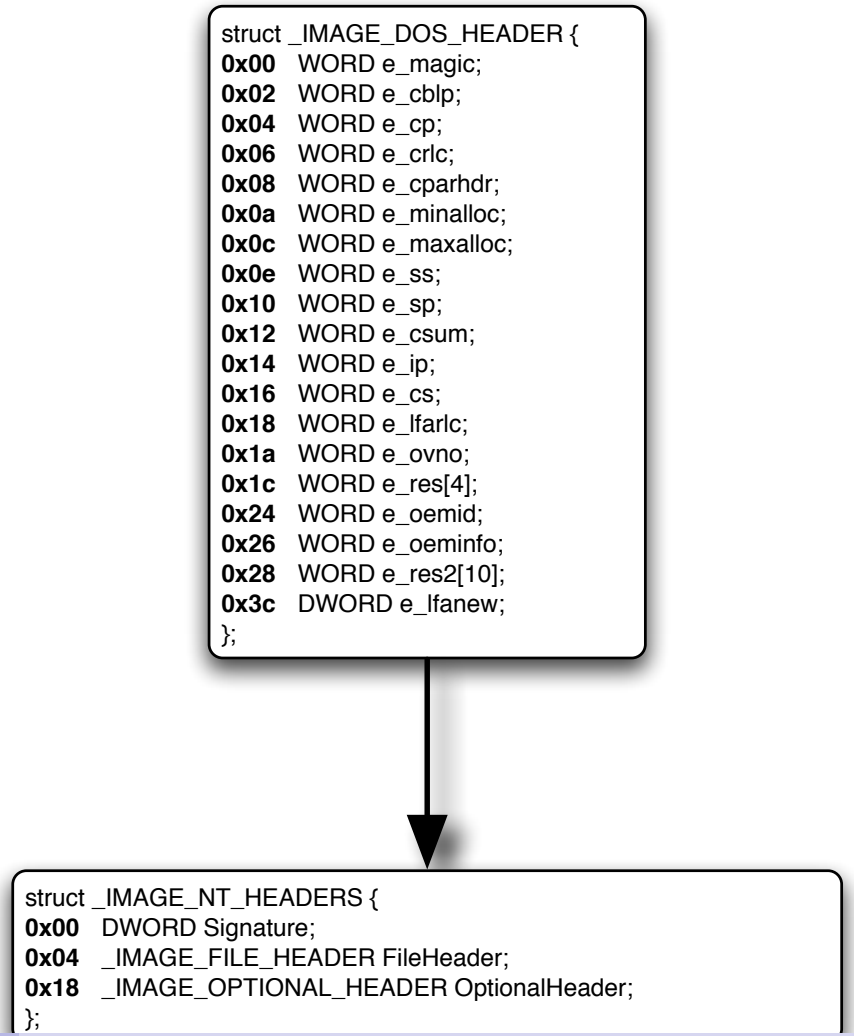


# DOS and NT Headers Overview

- These headers contain the very basic information to process *PE* files
- A *PE* file begins with the *DOS* stub, usually responsible for the *This program cannot be run in DOS mode* message as well as location the *PE* headers
- The PE headers contain the bulk of the information about the PE file
- Different sets of headers will be present depending on the type of data the PE file represents (an executable, a DLL, an object file)

# DOS and NT Headers View

- PE files start with the DOS Header.
  - e_magic = *4D5Ah* MZ
- NT headers comprise the FILE and the OPTIONAL headers.
  - Signature = *5045h* PE



# NT Headers: File Header View

```
struct _IMAGE_FILE_HEADER {
0x00   WORD Machine;
0x02   WORD NumberOfSections;
0x04   DWORD TimeDateStamp;
0x08   DWORD PointerToSymbolTable;
0x0c   DWORD NumberOfSymbols;
0x10   WORD SizeOfOptionalHeader;
0x12   WORD Characteristics;
};
```

# NT Headers: File Header

- It's the first of the *NT Headers* and *File Header* follows immediately after the *PE Signature*
- Contains some interesting fields
  - *Machine* indicates the target architecure for this file
  - *NumberOfSections*, the number of sections in the *PE file*. This value is needed when exploring the section headers
  - *TimeDateStamp* is not of a critical importance, but some malware actually seems not to zero it so it might give some insight on the approximate release time... can be easily faked too
  - *SizeOfOptionalHeader* is an important element. Provides the exact size of the *Optional Header* which is needed in order to properly parse the *PE* file

# NT Headers: Optional Header View

```
struct _IMAGE_OPTIONAL_HEADER {
0x00  WORD Magic;
0x02  BYTE MajorLinkerVersion;
0x03  BYTE MinorLinkerVersion;
0x04  DWORD SizeOfCode;
0x08  DWORD SizeOfInitializedData;
0x0c  DWORD SizeOfUninitializedData;
0x10  DWORD AddressOfEntryPoint;
0x14  DWORD BaseOfCode;
0x18  DWORD BaseOfData;
0x1c  DWORD ImageBase;
0x20  DWORD SectionAlignment;
0x24  DWORD FileAlignment;
0x28  WORD MajorOperatingSystemVersion;
0x2a  WORD MinorOperatingSystemVersion;
0x2c  WORD MajorImageVersion;
0x2e  WORD MinorImageVersion;
0x30  WORD MajorSubsystemVersion;
0x32  WORD MinorSubsystemVersion;
0x34  DWORD Win32VersionValue;
0x38  DWORD SizeOfImage;
0x3c  DWORD SizeOfHeaders;
0x40  DWORD CheckSum;
0x44  WORD Subsystem;
0x46  WORD DllCharacteristics;
0x48  DWORD SizeOfStackReserve;
0x4c  DWORD SizeOfStackCommit;
0x50  DWORD SizeOfHeapReserve;
0x54  DWORD SizeOfHeapCommit;
0x58  DWORD LoaderFlags;
0x5c  DWORD NumberOfRvaAndSizes;
0x60  _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

# NT Headers: Optional Header (1)

- *Magic* = *10Bh* (PE32+ *0x20b*)
- *AddressOfEntryPoint* is where execution of the executable code will begin (it's possible for other code within the executable to gain control before the entry point)
- *ImageBase*. All relative address are based on this one. It's also usually possible to find the *PE* header of the executable at this address in memory (unless it has been intentionally deleted)
- *SectionAlignment* is the alignment of the sections in memory
- *FileAlignment* is the alignment on disk

# NT Headers: Optional Header (2)

- *Operating system related fields* containing version specific information
- *NumberOfRvaAndSizes* is the number of directory entries in the following array. Depending on how many there are the size of the *Optional Header* will vary, something that some tools sometimes forget (assuming a constant default size)
- *DataDirectory* is an array of structures pointing to additional information such as the *Imports* and *Exports* tables.

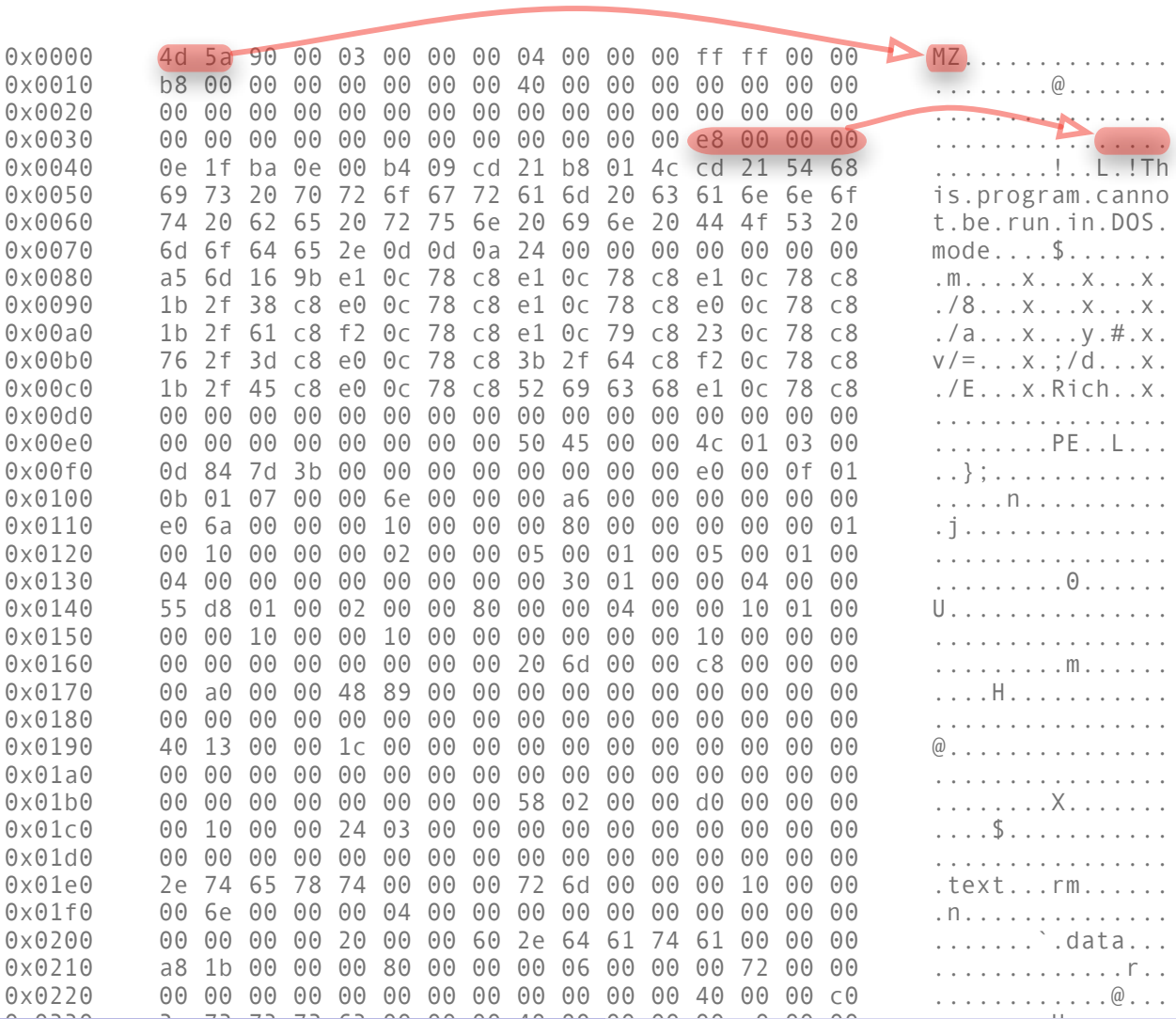
# Section Header View

```
typedef struct _IMAGE_SECTION_HEADER {
0x00    BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
0x08            DWORD PhysicalAddress;
0x08            DWORD VirtualSize;
        } Misc;
0x0c    DWORD VirtualAddress;
0x10    DWORD SizeOfRawData;
0x14    DWORD PointerToRawData;
0x18    DWORD PointerToRelocations;
0x1c    DWORD PointerToLinenumbers;
0x20    WORD  NumberOfRelocations;
0x22    WORD  NumberOfLinenumbers;
0x24    DWORD Characteristics;
};
```

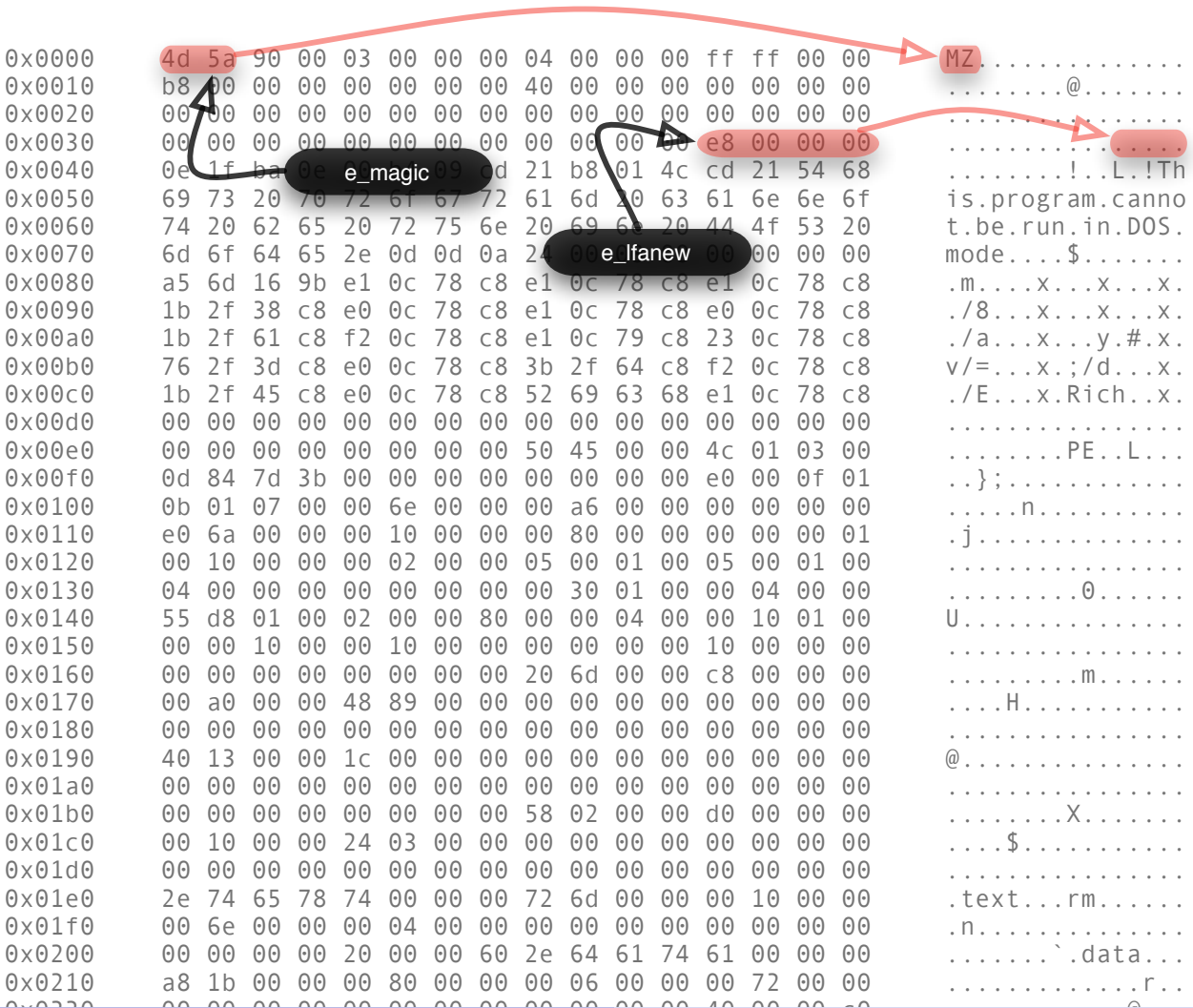
# Section Header

- *VirtualSize* is the size of the section once loaded in memory (can be bigger than *SizeOfRawData*, in that case it's zero padded)
- *VirtualAddress* is the address of the section in memory, relative to the *ImageBase*
- *SizeOfRawData* is the size of the section on disk (can be bigger than *VirtualSize* due that its size is rounded at a *FileAlignment* multiple)
- *PointerToRawData* is the offset within the file to the contents to be loaded in memory (*should* be a multiple of *VirtualSize*)
- *Characteristics* contains flags with information such as whether the section can be executed, read, written into, etc.

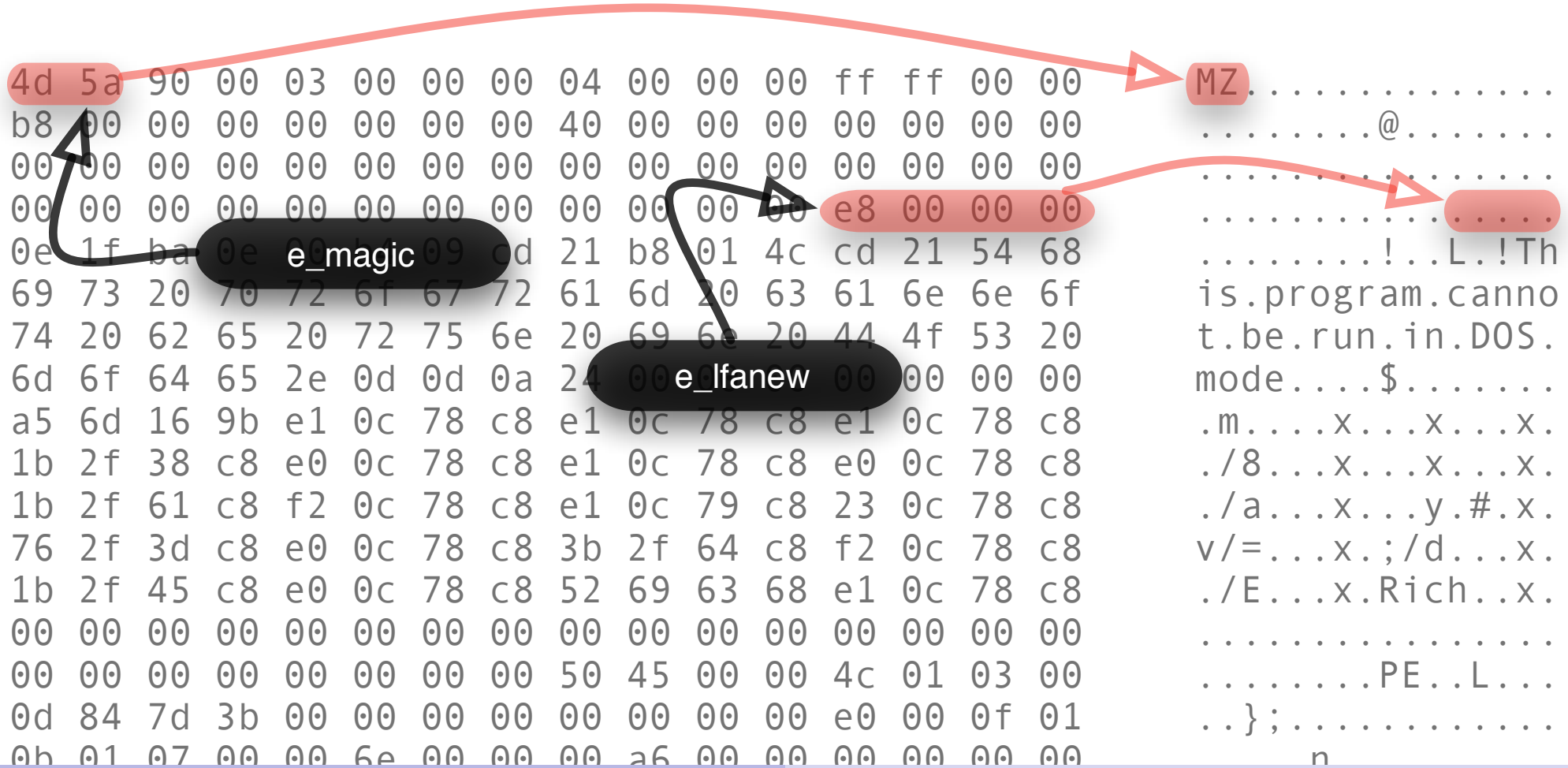
# DOS Header (1)


0x0000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x0010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0030 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 ................
0x0040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0x0050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is.program.canno
0x0060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t.be.run.in.DOS.
0x0070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
0x0080 a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8 .m....x...x...x.
0x0090 1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8 ./8...x...x...x.
0x00a0 1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8 ./a...x...y.#.x.
0x00b0 76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8 v/=...x.;/d...x.
0x00c0 1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8 ./E...x.Rich..x.
0x00d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x00e0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00 ........PE..L...
0x00f0 0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01 ..};............
0x0100 0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00 .....n..........
0x0110 e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01 .j..............
0x0120 00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00 ................
0x0130 04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00 .........0......
0x0140 55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00 U...............
0x0150 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ................
0x0160 00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00 .........m......
0x0170 00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00 ....H...........
0x0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0190 40 13 00 00 1c 00 00 00 00 00 00 00 00 00 00 00 @...............
0x01a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01b0 00 00 00 00 00 00 00 00 58 02 00 00 d0 00 00 00 ........X.......
0x01c0 00 10 00 00 24 03 00 00 00 00 00 00 00 00 00 00 ....$...........
0x01d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01e0 2e 74 65 78 74 00 00 00 72 6d 00 00 00 10 00 00 .text...rm......
0x01f0 00 6e 00 00 00 04 00 00 00 00 00 00 00 00 00 00 .n..............
0x0200 00 00 00 00 20 00 00 60 2e 64 61 74 61 00 00 00 .......`.data...
0x0210 a8 1b 00 00 00 80 00 00 00 06 00 00 00 72 00 00 .............r..
0x0220 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 c0 ............@...

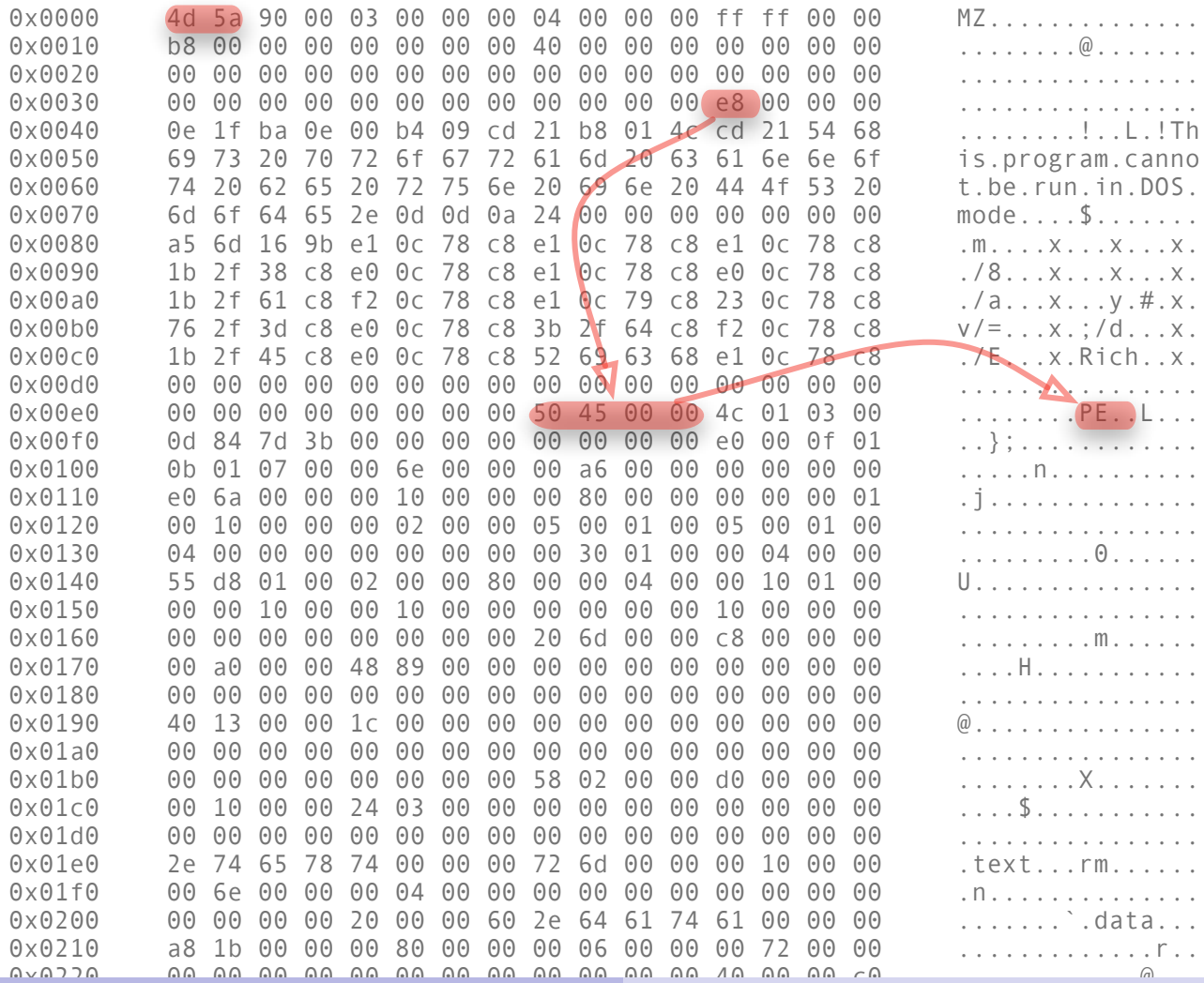
# DOS Header (2)



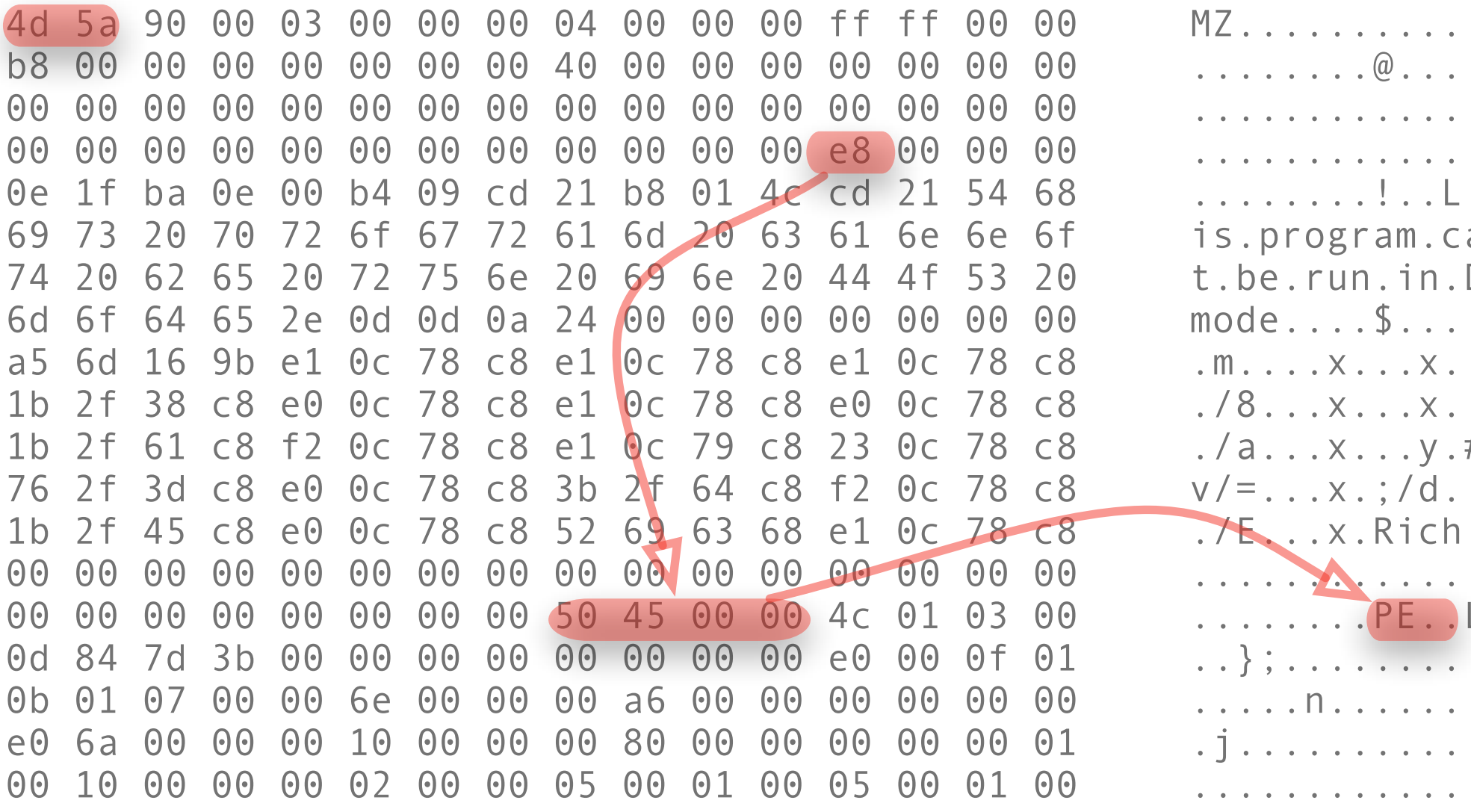
# DOS Header (2)


4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
MZ.....
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
........@.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00
............
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
........!..L.!Th
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
is.program.canno
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
t.be.run.in.DOS.
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
mode....$.......
a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8
.m....x...x...x.
1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8
./8...x...x...x.
1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8
./a...x...y.#.x.
76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8
v/=...x.;/d...x.
1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8
./E...x.Rich..x.
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00
........PE..L...
0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01
..};............
0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00
n
e_magic
e_lfanew

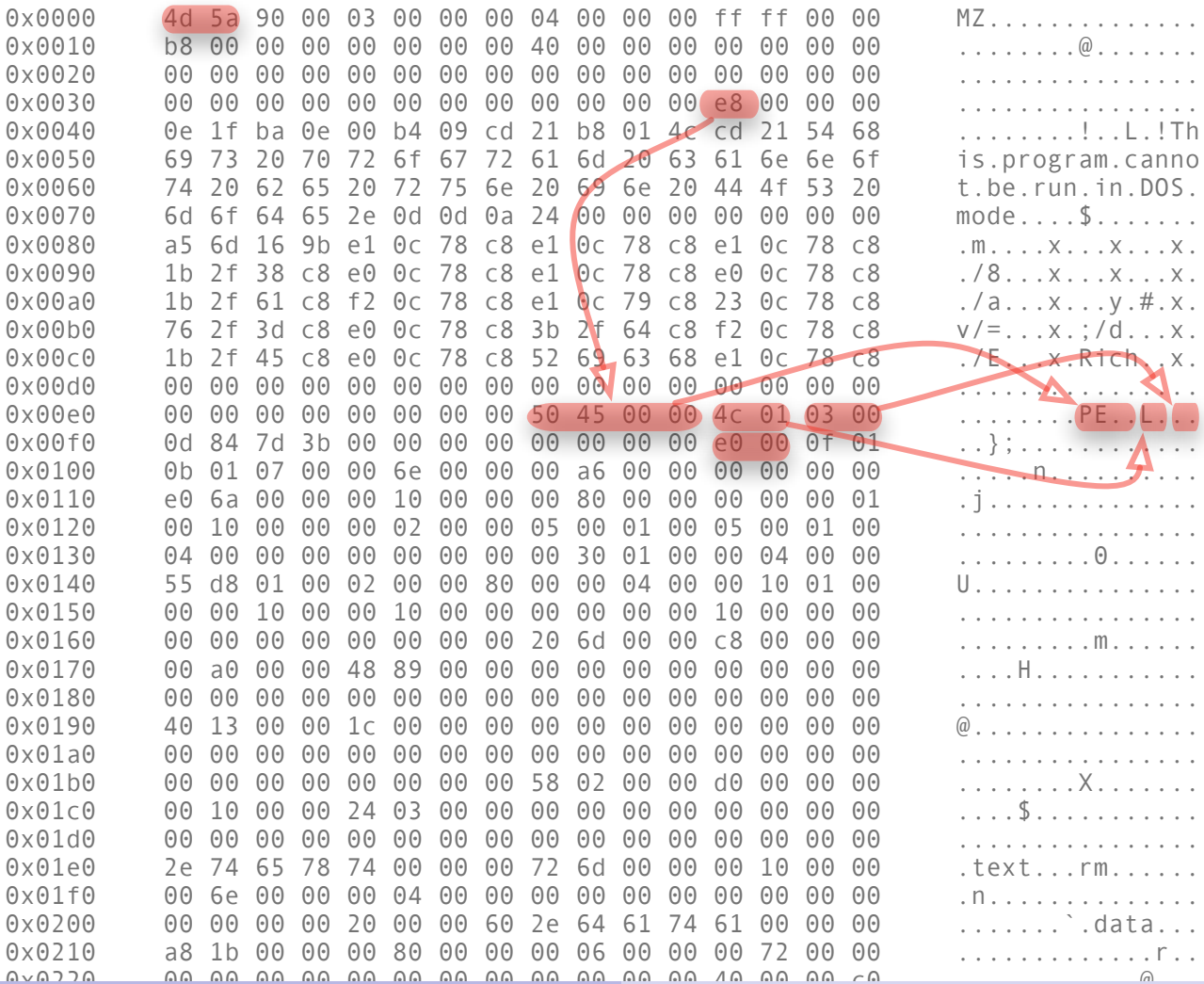
# NT Headers (1)


0x0000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x0010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0030 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 ................
0x0040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0x0050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is.program.canno
0x0060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t.be.run.in.DOS.
0x0070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
0x0080 a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8 .m....x...x...x.
0x0090 1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8 ./8...x...x...x.
0x00a0 1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8 ./a...x...y.#.x.
0x00b0 76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8 v/=...x.;/d...x.
0x00c0 1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8 ./E...x.Rich..x.
0x00d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x00e0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00 ........PE..L...
0x00f0 0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01 ..};............
0x0100 0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00 .....n..........
0x0110 e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01 .j..............
0x0120 00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00 ................
0x0130 04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00 .........0......
0x0140 55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00 U...............
0x0150 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ................
0x0160 00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00 .........m......
0x0170 00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00 ....H...........
0x0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0190 40 13 00 00 1c 00 00 00 00 00 00 00 00 00 00 00 @...............
0x01a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01b0 00 00 00 00 00 00 00 00 58 02 00 00 d0 00 00 00 ........X.......
0x01c0 00 10 00 00 24 03 00 00 00 00 00 00 00 00 00 00 ....$...........
0x01d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01e0 2e 74 65 78 74 00 00 00 72 6d 00 00 00 10 00 00 .text...rm......
0x01f0 00 6e 00 00 00 04 00 00 00 00 00 00 00 00 00 00 .n..............
0x0200 00 00 00 00 20 00 00 60 2e 64 61 74 61 00 00 00 .......`.data...
0x0210 a8 1b 00 00 00 80 00 00 00 06 00 00 00 72 00 00 .............r..

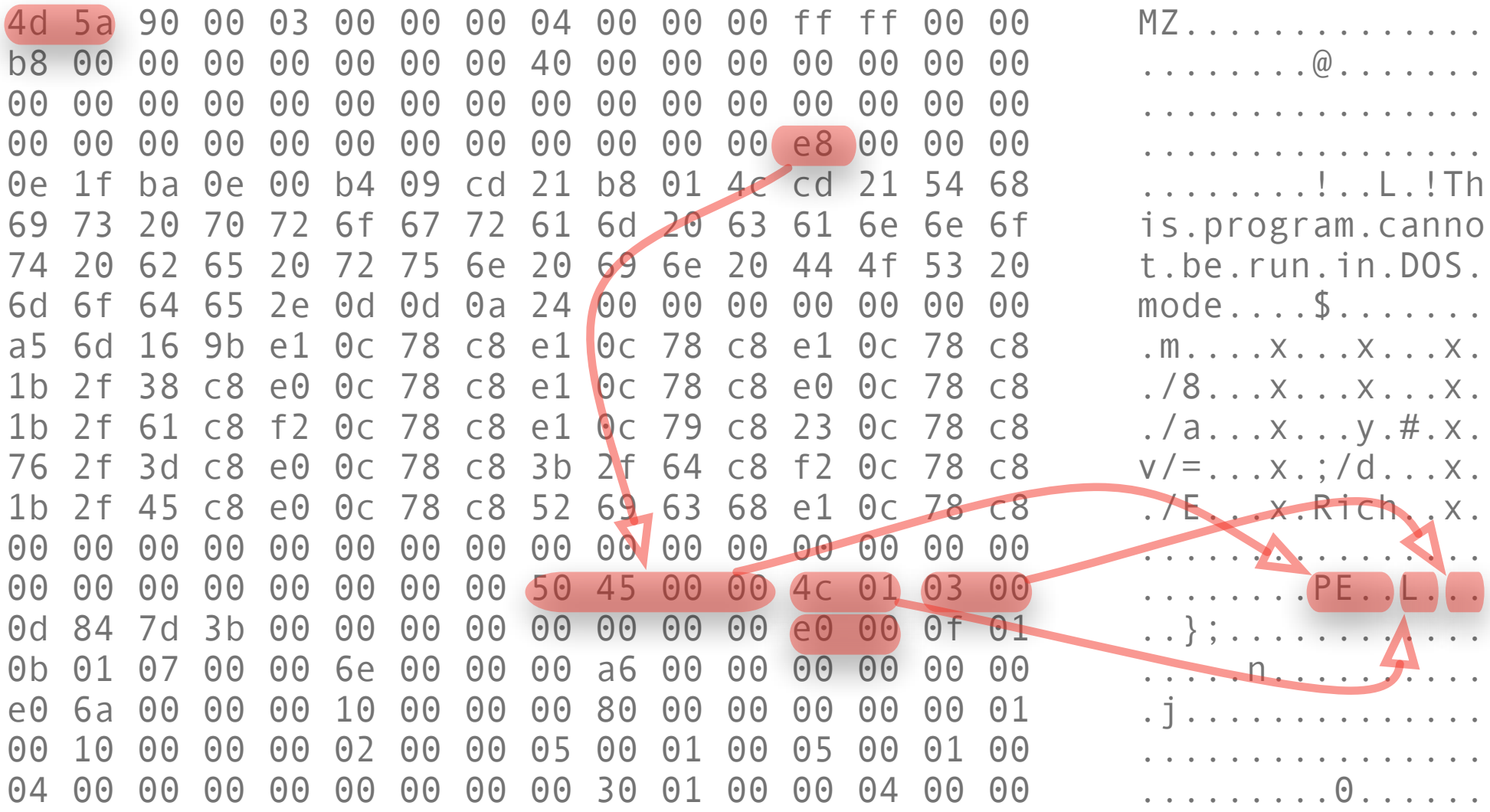
# NT Headers (1)


4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8
1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8
1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8
76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8
1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00
0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01
0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00
e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01
00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00
MZ.....
.....@...
.....
.....
.....!..L
is.program.ca
t.be.run.in.D
mode....$...
.m....x...x.
./8...x...x.
./a...x...y.#
v/=...x.;/d.
./E...x.Rich
.....
.....PE..L
..};.....
.....n.....
.j.....
.....

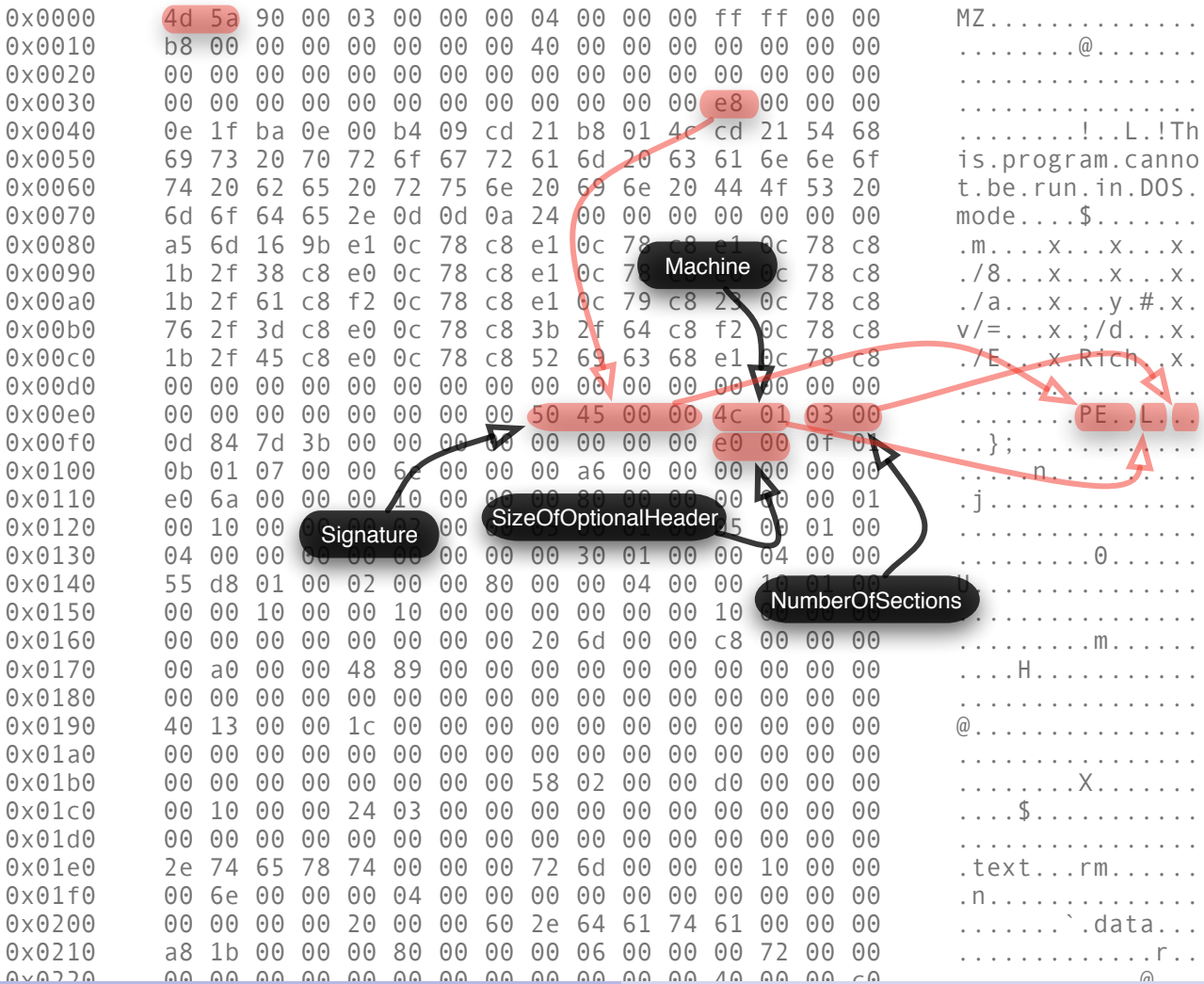
# NT Headers (2)


0x0000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x0010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0030 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 ................
0x0040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0x0050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is.program.canno
0x0060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t.be.run.in.DOS.
0x0070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
0x0080 a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8 .m....x...x...x.
0x0090 1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8 ./8...x...x...x.
0x00a0 1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8 ./a...x...y.#.x.
0x00b0 76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8 v/=...x.;/d...x.
0x00c0 1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8 ./E...x.Rich..x.
0x00d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x00e0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00 ........PE..L...
0x00f0 0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01 ..};............
0x0100 0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00 .....n..........
0x0110 e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01 .j..............
0x0120 00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00 ................
0x0130 04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00 .........0......
0x0140 55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00 U...............
0x0150 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ................
0x0160 00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00 .........m......
0x0170 00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00 ....H...........
0x0180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0190 40 13 00 00 1c 00 00 00 00 00 00 00 00 00 00 00 @...............
0x01a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01b0 00 00 00 00 00 00 00 00 58 02 00 00 d0 00 00 00 ........X.......
0x01c0 00 10 00 00 24 03 00 00 00 00 00 00 00 00 00 00 ....$...........
0x01d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01e0 2e 74 65 78 74 00 00 00 72 6d 00 00 00 10 00 00 .text...rm......
0x01f0 00 6e 00 00 00 04 00 00 00 00 00 00 00 00 00 00 .n..............
0x0200 00 00 00 00 20 00 00 60 2e 64 61 74 61 00 00 00 .......`.data...
0x0210 a8 1b 00 00 00 80 00 00 00 06 00 00 00 72 00 00 .............r..

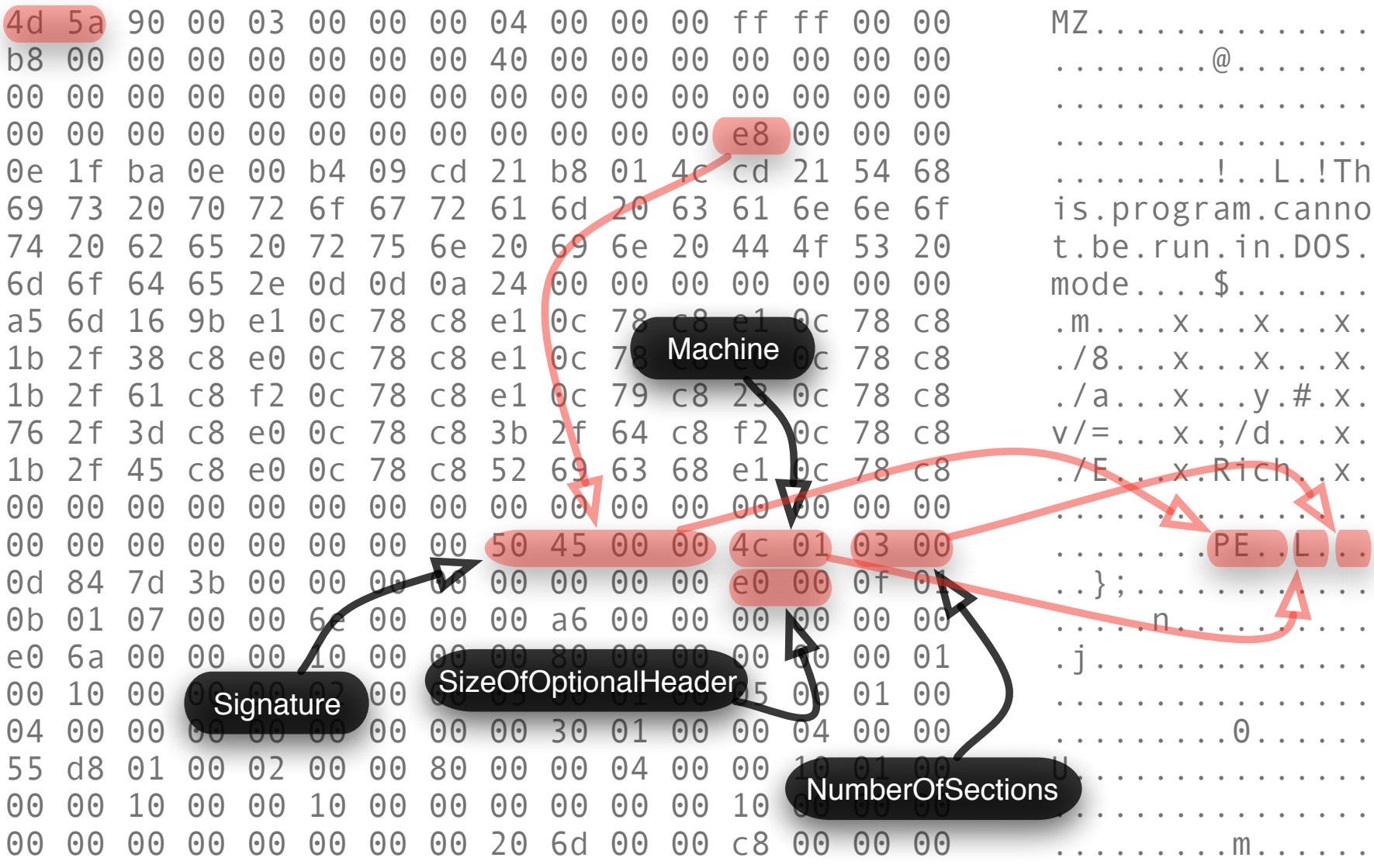
# NT Headers (2)


4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8
1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8
1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8
76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8
1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00
0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01
0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00
e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01
00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00
04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00
MZ..............
........@.......
................
................
........!..L.!Th
is.program.canno
t.be.run.in.DOS.
mode....$.......
.m....x...x...x.
./8...x...x...x.
./a...x...y.#.x.
v/=...x.;/d...x.
./E...x.Rich..x.
................
........PE..L...
..};............
.....n..........
.j..............
................
.........0......

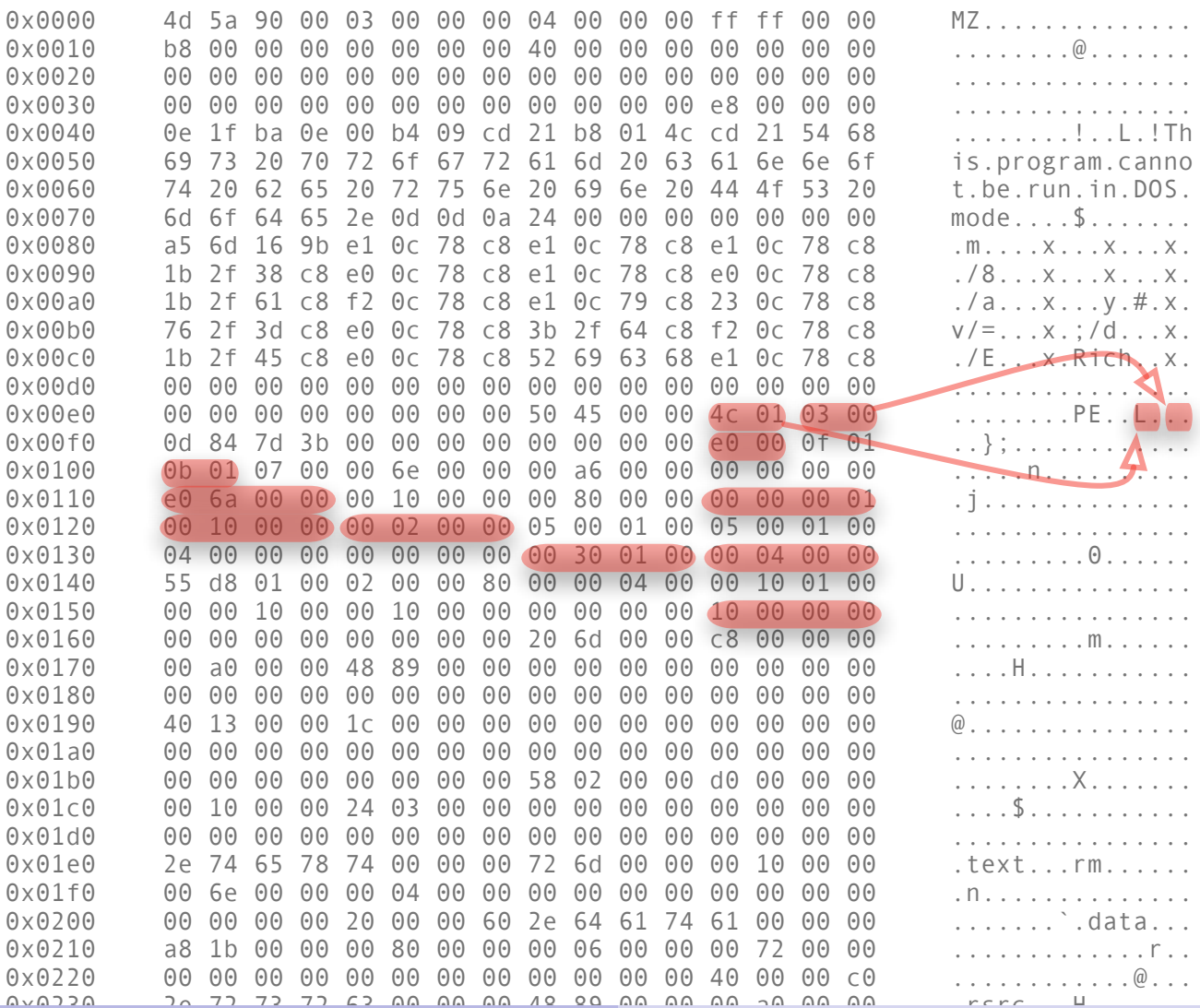
# NT Headers (3)


Machine
Signature
SizeOfOptionalHeader
NumberOfSections

# NT Headers (3)


4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00
0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8
1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8
1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8
76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8
1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00
0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01
0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00
e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01
00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00
04 00 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00
55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00
00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00
00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00
MZ..............
........@.......
................
................
........!..L.!Th
is.program.canno
t.be.run.in.DOS.
mode....$.......
.m....x...x...x.
./8...x...x...x.
./a...x...y.#.x.
v/=...x.;/d...x.
./E...x.Rich...x.
................
........PE..L...
..};............
....n...........
.j..............
................
........0.......
U...............
................
........m.......
Machine
Signature
SizeOfOptionalHeader
NumberOfSections

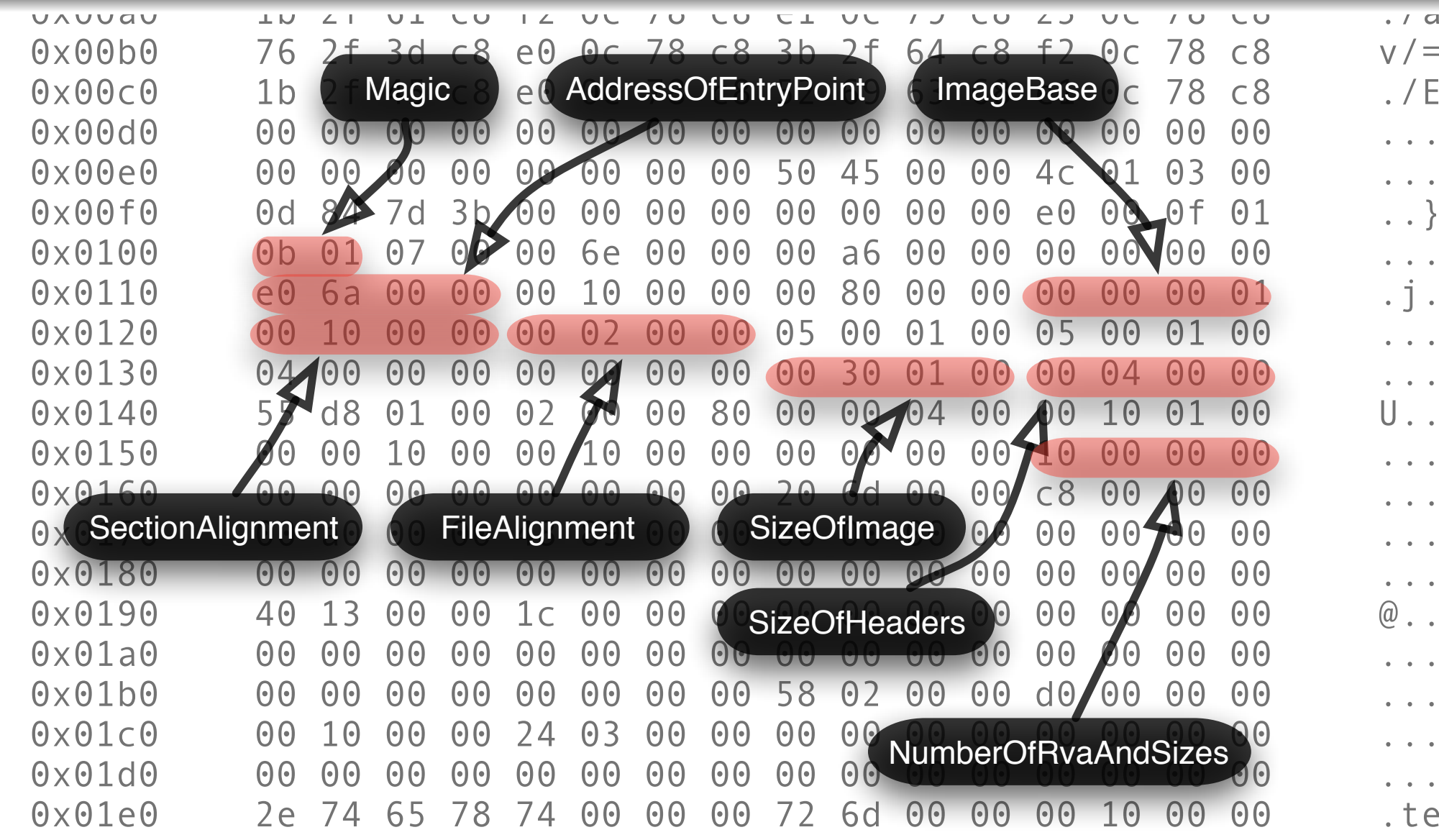
# Optional Header (1)



# Optional Header (2)



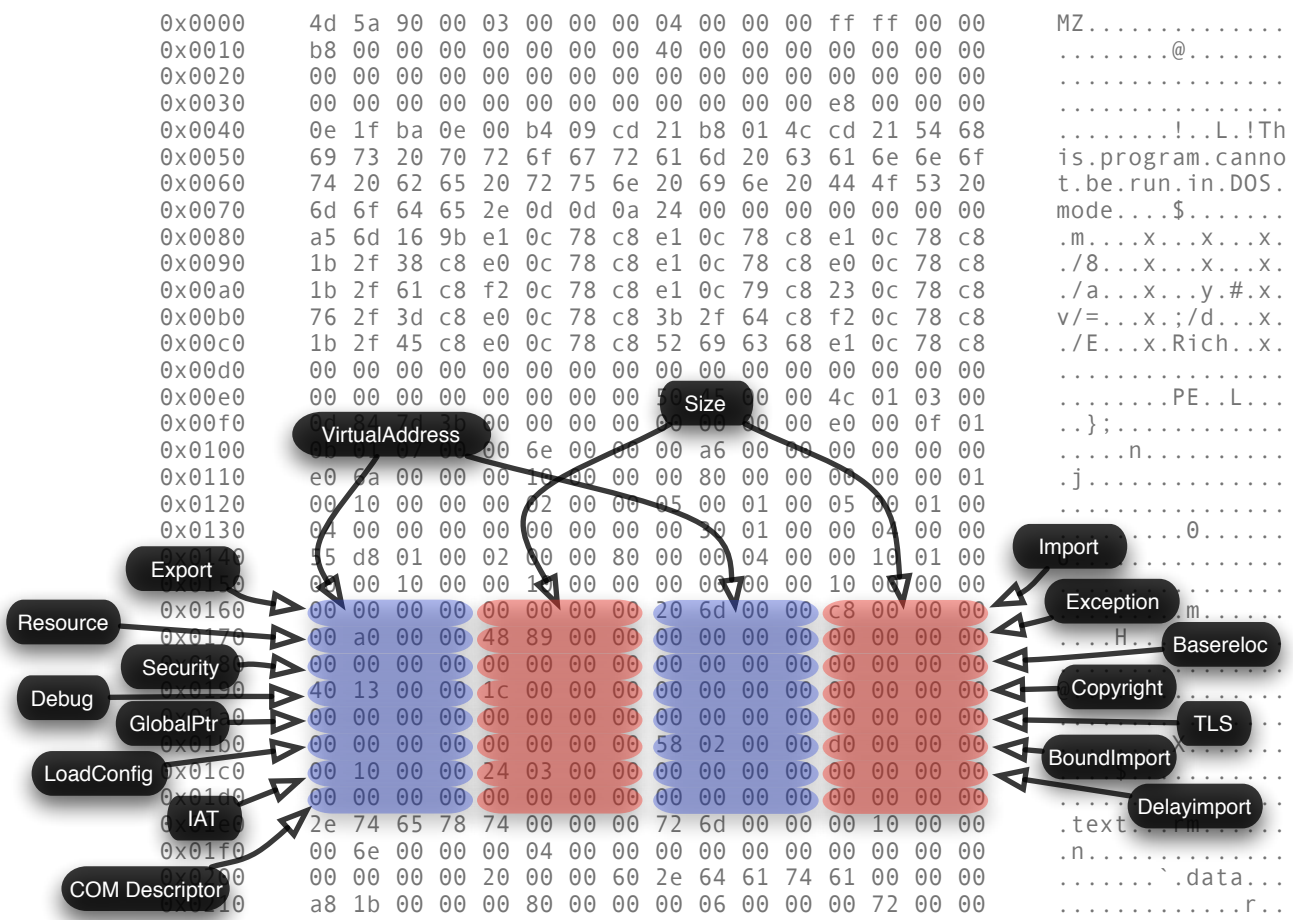
# Optional Header (2)



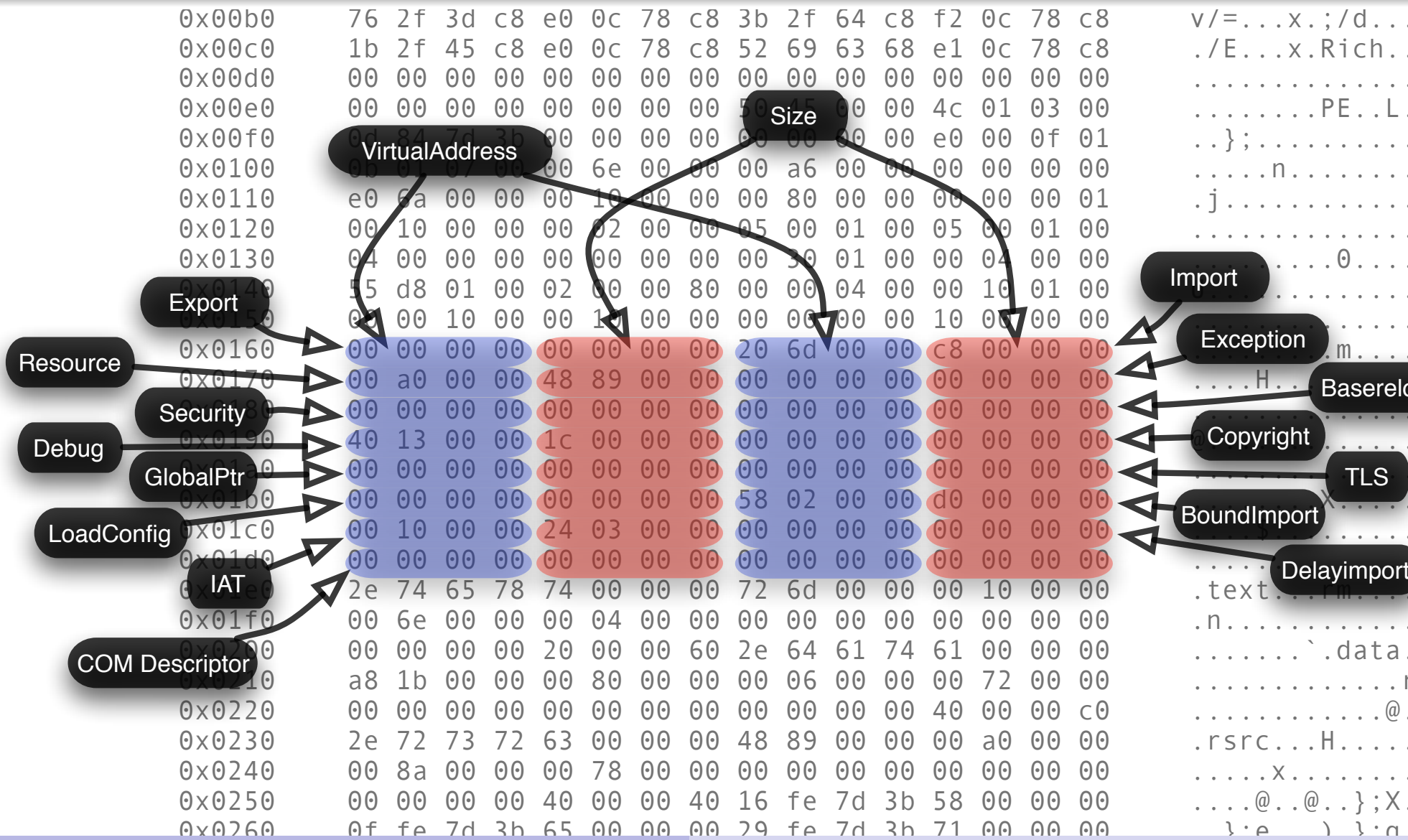
# Directories (1)

```
0x0000    4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00    MZ..............
0x0010    b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00    ........@.......
0x0020    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x0030    00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00    ................
0x0040    0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68    ........!..L.!Th
0x0050    69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f    is.program.canno
0x0060    74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20    t.be.run.in.DOS.
0x0070    6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00    mode....$.......
0x0080    a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8    .m....x...x...x.
0x0090    1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8    ./8...x...x...x.
0x00a0    1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8    ./a...x...y.#.x.
0x00b0    76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8    v/=...x.;/d...x.
0x00c0    1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8    ./E...x.Rich..x.
0x00d0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x00e0    00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00    ........PE..L...
0x00f0    0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01    ..};............
0x0100    0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00    .....n..........
0x0110    e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01    .j..............
0x0120    00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 00    ................
0x0130    04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00    .........0......
0x0140    55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00    U...............
0x0150    00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00    ................
0x0160    00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00    ........m.......
0x0170    00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00    ....H...........
0x0180    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x0190    40 13 00 00 1c 00 00 00 00 00 00 00 00 00 00 00    @...............
0x01a0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x01b0    00 00 00 00 00 00 00 00 58 02 00 00 d0 00 00 00    ........X.......
0x01c0    00 10 00 00 24 03 00 00 00 00 00 00 00 00 00 00    ....$...........
0x01d0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
0x01e0    2e 74 65 78 74 00 00 00 72 6d 00 00 00 10 00 00    .text...rm......
0x01f0    00 6e 00 00 00 04 00 00 00 00 00 00 00 00 00 00    .n..............
0x0200    00 00 00 00 20 00 00 60 2e 64 61 74 61 00 00 00    .......`.data...
0x0210    a8 1b 00 00 00 80 00 00 00 06 00 00 00 72 00 00    .............r..
0x0220    00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 c0    ............@...
0x0230    2e 72 73 72 63 00 00 00 48 89 00 00 00 a0 00 00    .rsrc...H.......
```

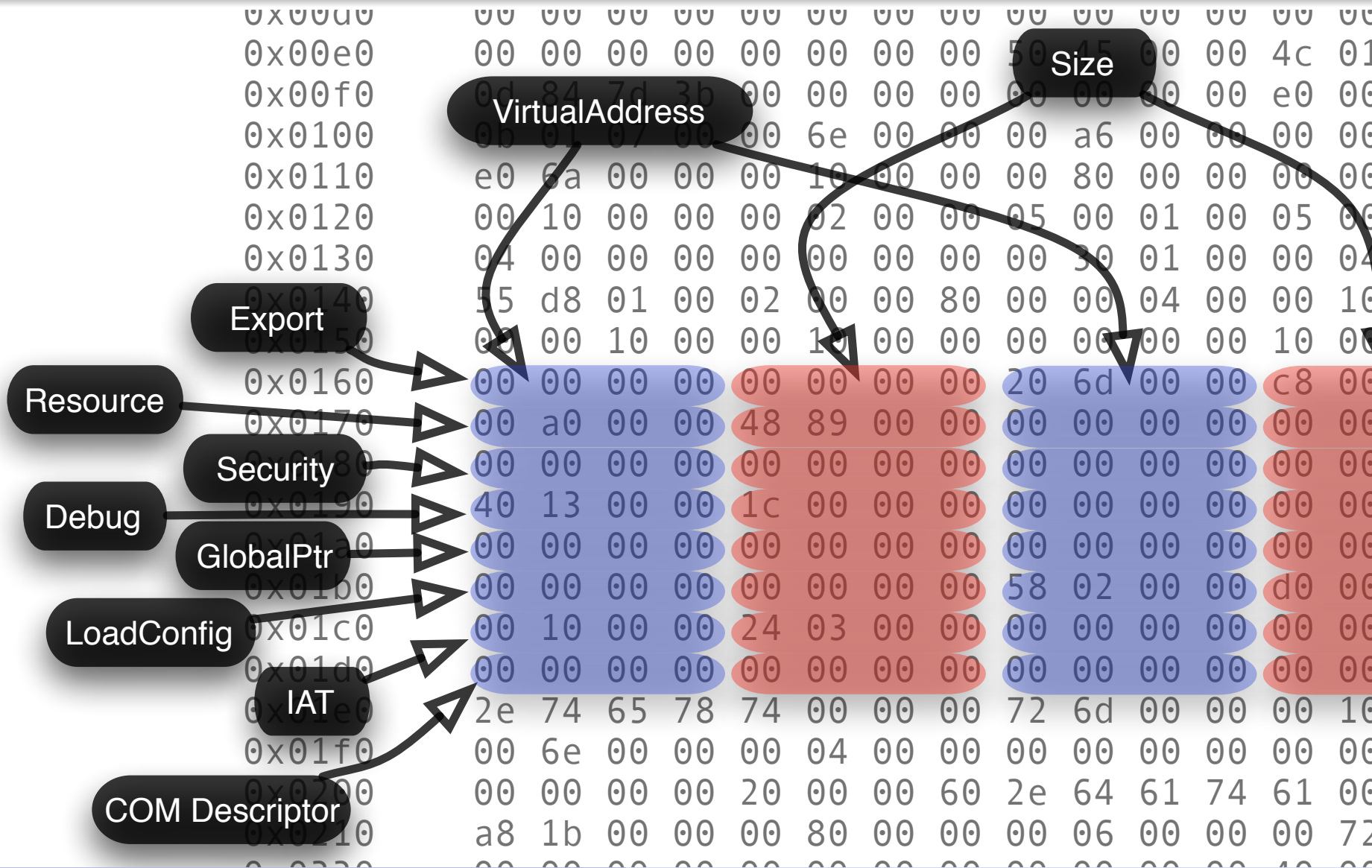
# Directories (2)



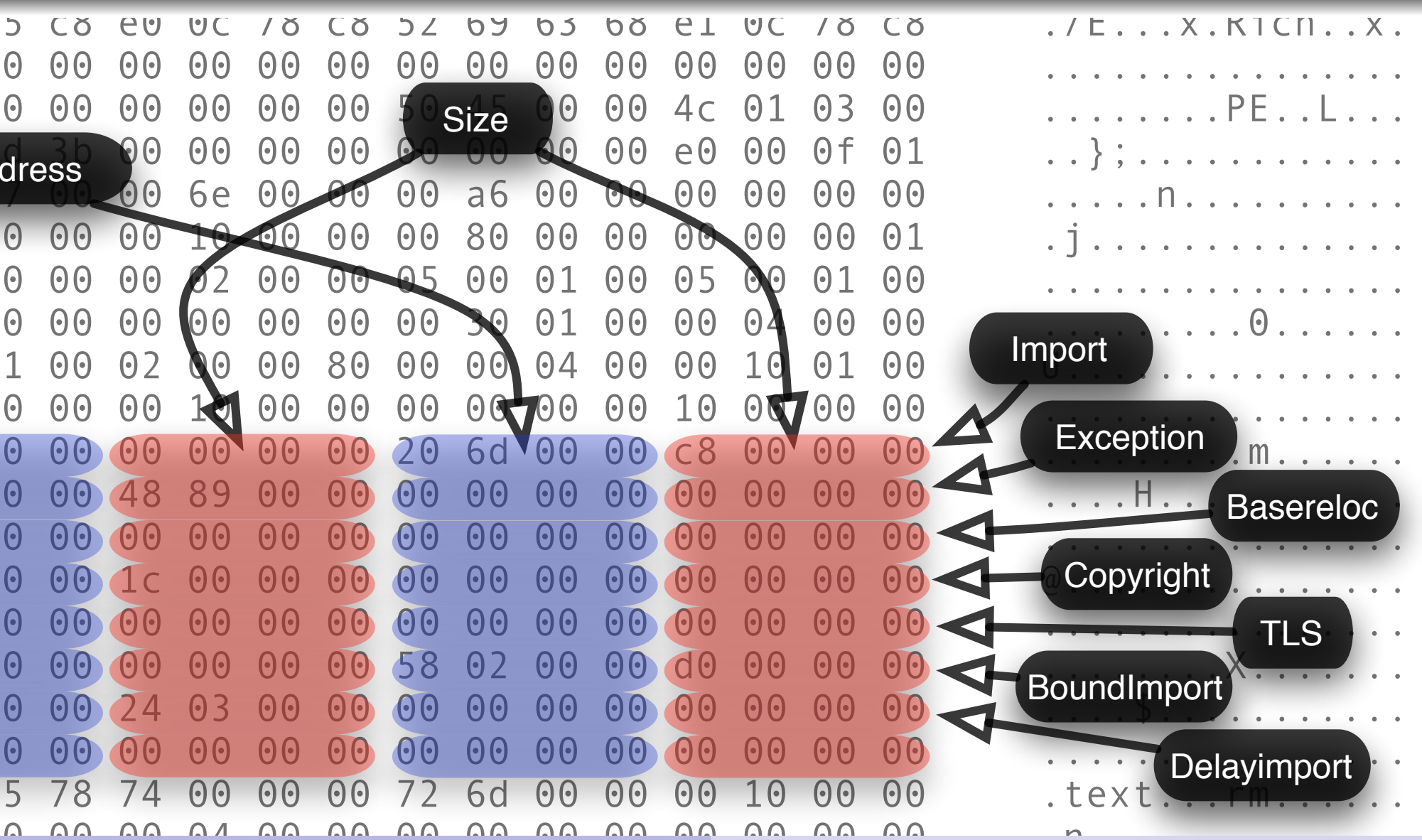
# Directories (2)



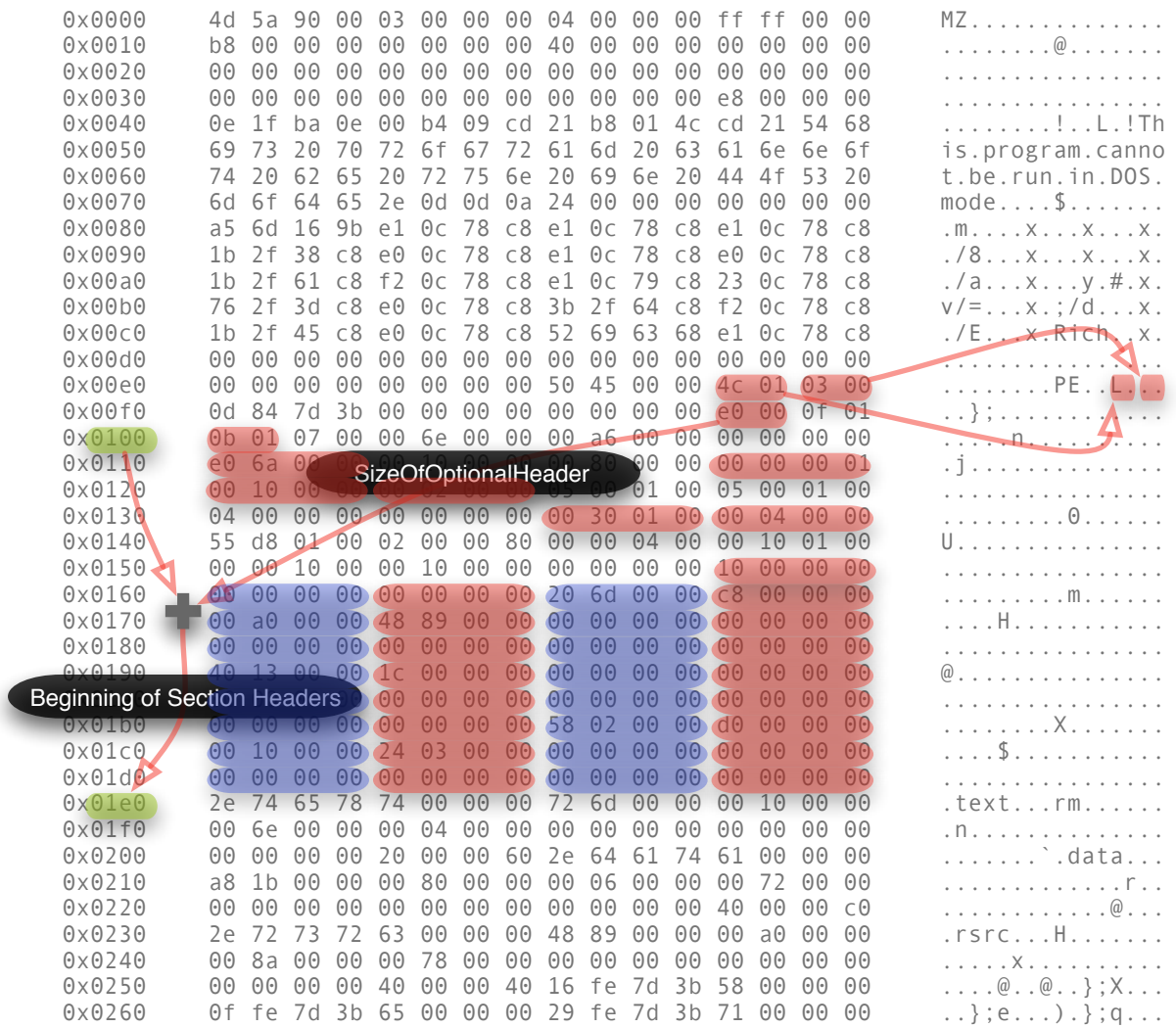
# Directories (2)



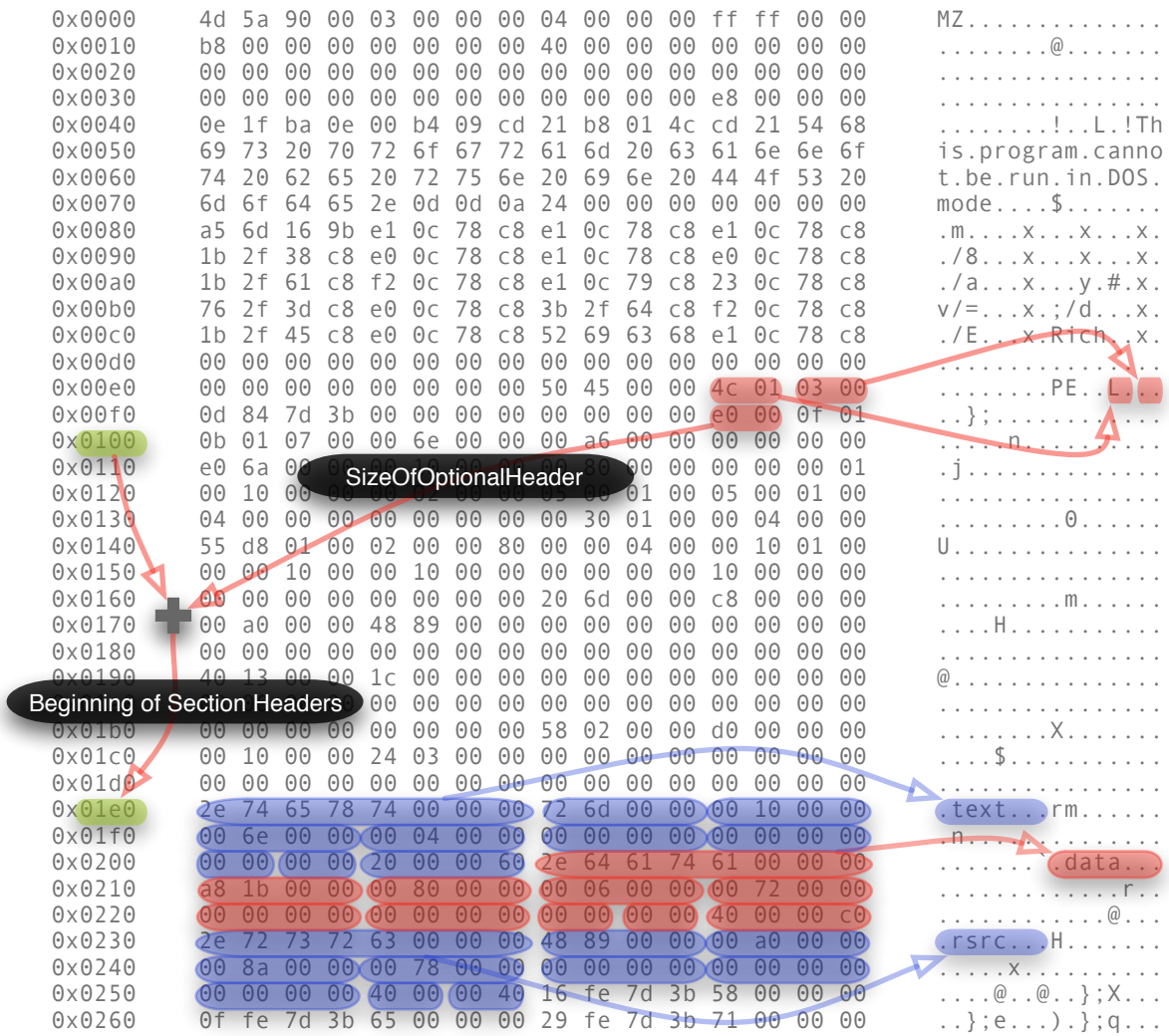
# Directories (2)



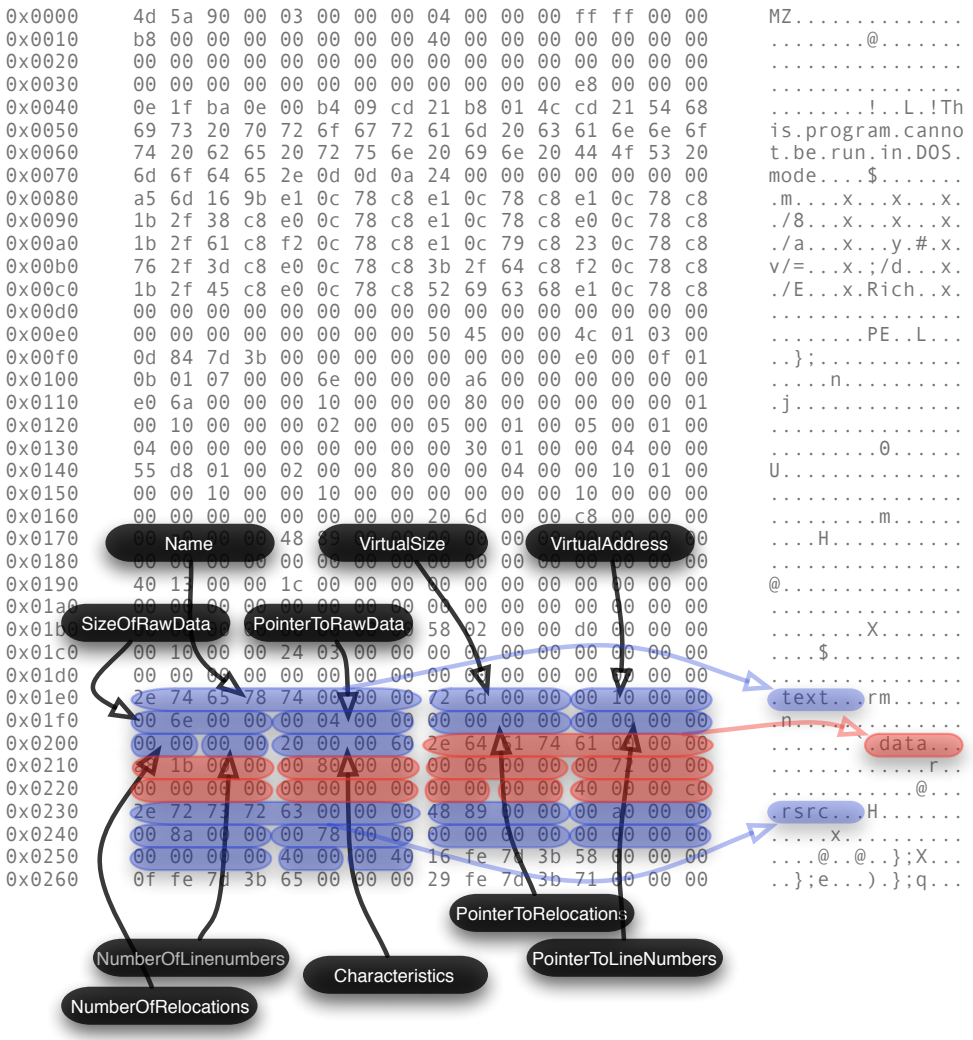
# Section Headers (1)



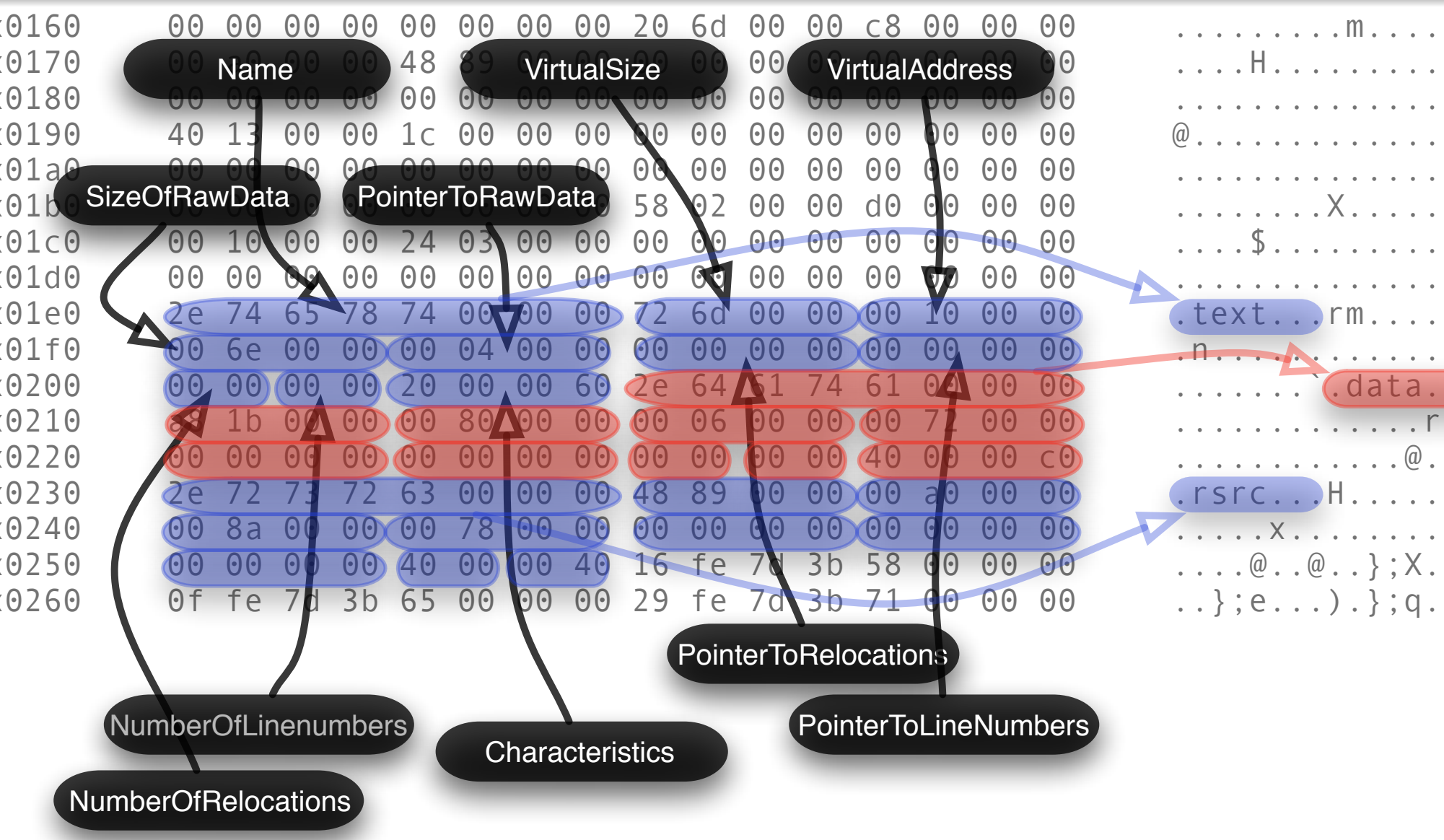
# Section Headers (2)



# Section Headers (3)



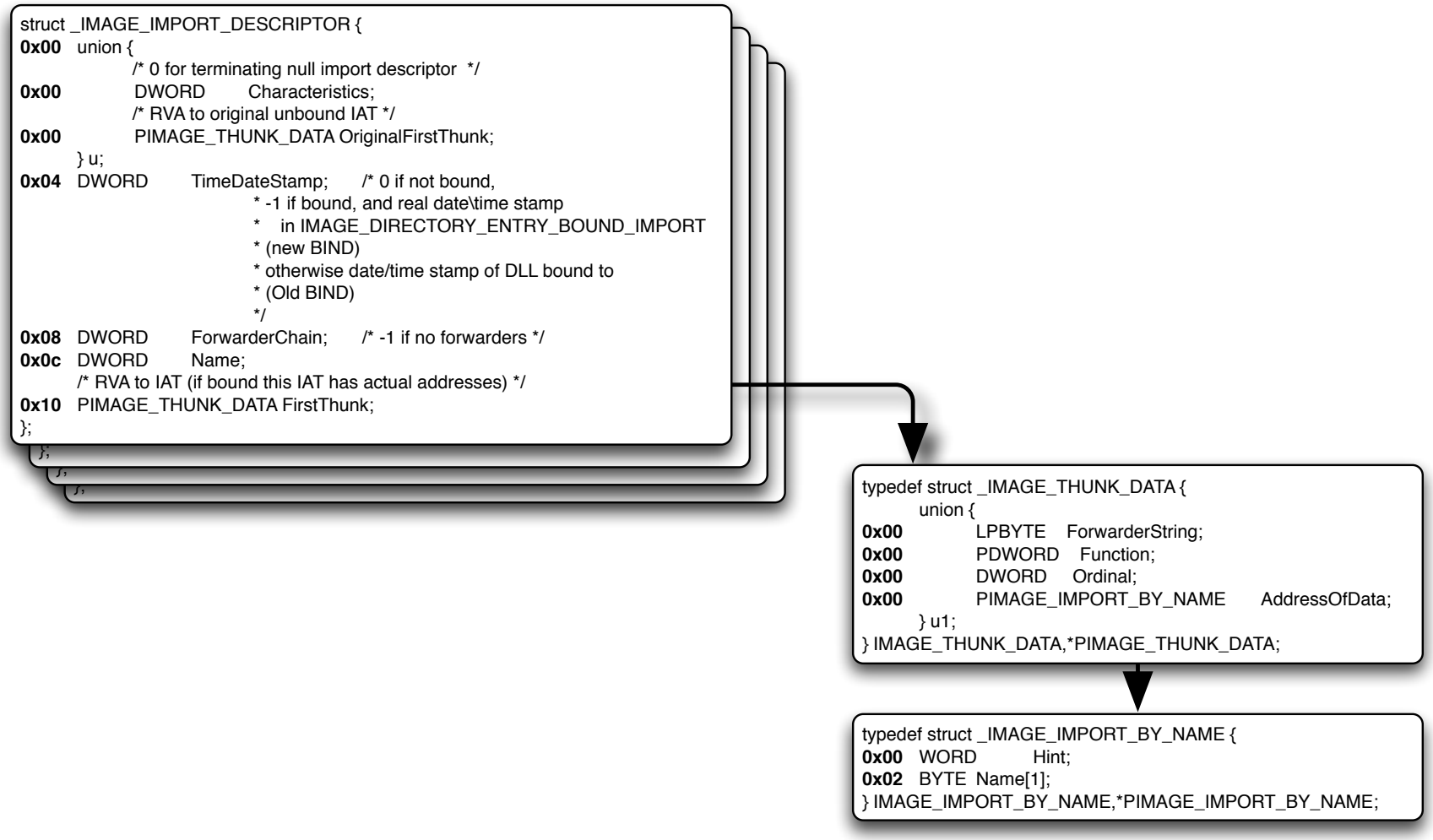
# Section Headers (3)



# Overview of the Import Address Table

- The primary function of the *Import Table* is to provide enough information to the loader to locate the API functions and other symbols needed by the executable
- It also provides us with a summary of the range of actions used by the executable
- Therefore hiding/obfuscating the *Import Address Table* (*IAT*) is a common technique in order to deprive analysts of a quick outlook
- The *IAT* can be rebuilt by different packers/obfucators with varying degrees of complexity

# The Import Address Table Structures View



# The Import Address Table Structures Commented

The *Import Address Table* information is distributed among three different structures. Repeated as necessary to describe the composing elements

- The *IMAGE_IMPORT_DESCRIPTOR* contains information about the *DLL* containing the symbols to import
- The *IMAGE_THUNK_DATA* contains information about the specific symbol imported *A*nd finally, *IMAGE_IMPORT_BY_NAME* contains the name of the imported symbol if it's imported by name and not by ordinal alone

# The IMAGE_IMPORT_DESCRIPTOR Structure

```
struct _IMAGE_IMPORT_DESCRIPTOR {
0x00   union {
               /* 0 for terminating null import descriptor  */
0x00           DWORD       Characteristics;
               /* RVA to original unbound IAT */
0x00           PIMAGE_THUNK_DATA OriginalFirstThunk;
       } u;
0x04   DWORD       TimeDateStamp;      /* 0 if not bound,
                                        * -1 if bound, and real date\time stamp
                                        *    in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
                                        * (new BIND)
                                        * otherwise date/time stamp of DLL bound to
                                        * (Old BIND)
                                        */
0x08   DWORD       ForwarderChain;     /* -1 if no forwarders */
0x0c   DWORD       Name;
       /* RVA to IAT (if bound this IAT has actual addresses) */
0x10   PIMAGE_THUNK_DATA FirstThunk;
};
```

# The IMAGE_IMPORT_DESCRIPTOR Structure Commented

The *IMAGE_IMPORT_DESCRIPTOR* contains information about the *DLL* containing the symbols to import

- *OriginalFirstThunk* contains the relative address of the import table (a *NULL* terminated array of *IMAGE_THUNK_DATA* structures) containing the symbols to be imported
- *Name* is the relative address of the name of the *DLL* from which to import the symbols
- *FirstThunk* is normally identical to *OriginalFirstThunk* except after the imports have been resolved, when it will contain the addressses of the symbols
- If the imports are bound the *TimeDateStamp* field will contain the timestamp of the referred DLL

# The IMAGE_THUNK_DATA Structure

```
typedef struct _IMAGE_THUNK_DATA {
        union {
0x00            LPBYTE    ForwarderString;
0x00            PDWORD    Function;
0x00            DWORD     Ordinal;
0x00            PIMAGE_IMPORT_BY_NAME      AddressOfData;
        } u1;
} IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;
```



```
typedef struct _IMAGE_IMPORT_BY_NAME {
0x00   WORD         Hint;
0x02   BYTE  Name[1];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

# The IMAGE_THUNK_DATA Structure Commented

The *IMAGE_THUNK_DATA* contains information about the specific symbol imported

- *ForwarderString* is not really used, *Microsoft*'s docs no longer even mention this field
- *Function* points to the data for the imported symbol when the image is bound or it has been resolved
- *Ordinal* of the symbol to import
- *AddressOfData* points to a *IMAGE_IMPORT_BY_NAME* structure with the name of the symbol to import
- The symbol is either imported by ordinal or name, this is indicated by the most significant bit, if set indicates the entry should be imported by ordinal and by name otherwise

# The IMAGE_IMPORT_BY_NAME Structure Commented

*IMAGE_IMPORT_BY_NAME* contains the name of the symbol to import

- *Hint* is an index into the exported symbols table of the imported *DLL*. Its purpose is to speed up load, if the symbol at that index matches the name then a sequential lookup can be skipped

# The Import Address Table

Executables wanting to hide imported symbols can resort to a large number of tricks. Usually they will resolve the imported symbols themselves and thus the *IAT* will appear nearly empty. Some of the most popular ways of building the *IAT* are

- Manually going through the *LoadLibrary*, *GetProcAddress* sequence for all symbols
- Looking them up through hashes of their names
- Looking them up through signatures of their code

Once mapped, they can be integrated into the binary through

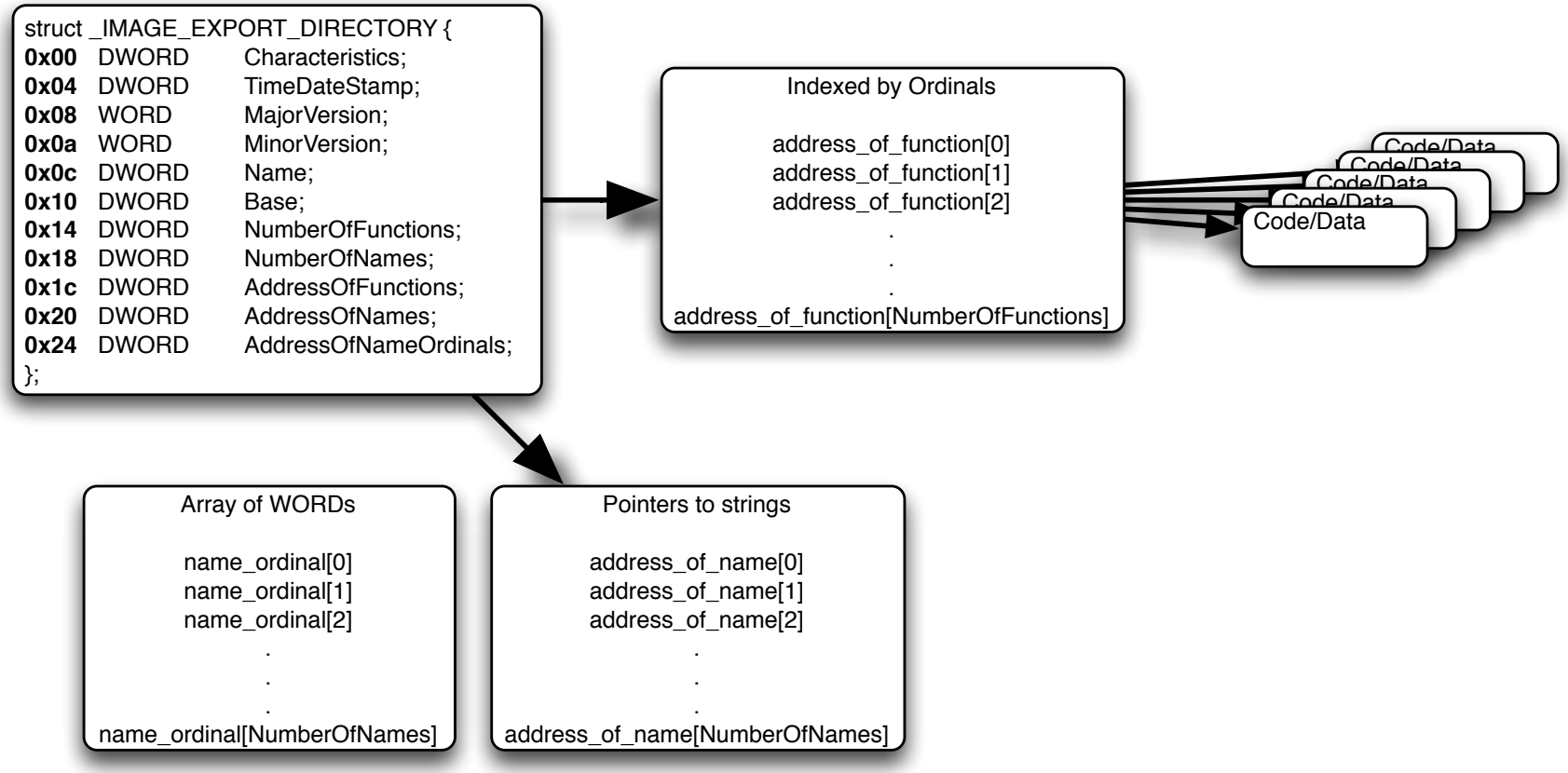
- Peculiar jump tables
- Skipping the *DLL* function's entry point. It confuses import rebuilding techniques *searching for* or *hooking at* known *DLL* function entry points

# The Export Table

Executable files such as applications and *DLL*s can export symbols for other components to import

- Both EXEs and DLLs can export symbols although EXEs rarely do so
- DLLs need to export symbols in order for other executables to import them

# The Export Table Structures



# The IMAGE_EXPORT_DIRECTORY Structure Commented

The *IMAGE_EXPORT_DIRECTORY* contains the information and pointers to code, the name and the ordinal for all exported symbols in the executable

- *Characteristics*, this field should always be 0 according to the specification
- *TimeDateStamp*, *MajorVersion* and *MinorVersion* are self explanatory
- *Name* points to a relative address containing the name of the *DLL*
- *NumberOfFunctions* is the number of pointers in the *AddressOfFunctions* table
- *NumberOfNames* is the number of entries in the *AddresOfNames* table

# The IMAGE_EXPORT_DIRECTORY Structure Commented

*AddressOfNames* and *AddressOfNameOrdinals* run parallel, having an entry for each exported symbol, the name can be *NULL*, the ordinal in *AddressOfNameOrdinals* is used to find the symbol's data by indexing with it *AddressOfFunctions*

- *AddressOfFunctions* points to an array of pointers containing the exported entries, it's indexed by an ordinal
- *AddressOfNames* points to an array of pointers containing the names of the exported entries
- *AddressOfNameOrdinals* points to an array of ordinals used to find the address of the exported data

# The Updated PE32+

- The PE format has been expanded by Microsoft to accomodate for 64-bit architectures
- While on some other aspects the executables have changed, most of the PE headers remain largely untouched
- As a rule of thumb fields involving absolute addresses have been expanded to 8 bytes to accomodate for the 64-bit wide address space
- Fields containing RVAs remain as 4 bytes as the maximum image size is limited to 2GB and only 31-bits are necessary to address it all with relative addresses

# Updated Fields in the Optional Header

- Optional header's *Magic* number is in PE32+ *0x20b*
- *BaseOfData* has been removed from the Optional Header
- The following are now 64-bit wide *ImageBase*, *SizeOfStackReserve*, *SizeOfStackCommit*, *SizeOfHeapReserve*, *SizeOfHeapCommit*

# Other Updated Fields

- The following items have been updated in the IMAGE_TLS_DIRECTORY structure: *StartAddressOfRawData*, *EndAddressOfRawData*, *AddressOfIndex*, *AddressOfCallBacks*
- And the following in the IMAGE_LOAD_CONFIG_DIRECTORY: *SecurityCookie*, *SEHandlerTable*, *SEHandlerCount*

# Curiosities of PE32+

- Due to the new way of handling exceptions, the Exception Directory is supposed to contain most of the functions of the binary, more specifically, the "non-leaf" ones
- This enables tools to readily know most of the functions within a binary without having to rely on discovery through disassembly

# The Tiny PE challenge

Solareclipse took on the *Tiny PE* challenge set by Dil Dabah about creating the smallest valid PE file. The result was:

- The smallest possible PE file: 97 bytes
- The smallest possible PE file on Windows 2000: 133 bytes
- The smallest possible PE file that downloads and executes a file over WebDAV: 133 bytes

[Tiny PE]

# The Tiny PE challenge

In order to achieve such small sizes the following steps were taken:

- Decreasing file alignment
- Removing the DOS stub
- Removing data directories
- Merging the section header within the Optional Header
- Merging the import table within the Optional Header
- Merging the IAT and DLL name
- Reusing the storage provided by non-used fields, only two matter in the DOS header
- Reducing the Optional Header size by truncating it at the minimum necessary length

# Additional Resources

**Microsoft Portable Executable and Common Object File Format Specification**

[Microsoft PE and COFF Specification]

**Portable Executable File Format  A Reverse Engineer View**

[PE File Format  A Reverse Engineer View]

# Part II

# Basic Analysis

# Outline

6 Overview of Analysis Tools
- Debuggers
- Disassemblers / Decompilers
- Other
- Python

7 (Dis)Assembly

8 IDA Pro

9 OllyDbg

# What is a Debugger?

**Definition**

A debugger is a run-time analysis tool that allows you to instrument software at the assembly level.

- Common features include:
  - CPU state information
  - Single stepping
  - Breakpoints
  - Memory exploration and modification
  - Thread enumeration
  - PE file parsing

# Unix Debuggers

- GDB: The GNU Debugger
  - DDD: GUI front end
- ADB
- Fenris, http://lcamtuf.coredump.cx/fenris
- RR0D, http://rr0d.droids-corp.org
  - This is actually an OS independent debugger, but there is no space on the next slide ;-)

# Microsoft Windows Debuggers

- Microsoft WinDbg
  - Powerful, freeware GUI debugging tool. Excellent for kernel development / abuse and security research
- SoftICE
  - Powerful, expensive kernel debugger
  - ICE = In Circuit Emulator
- OllyDbg
  - Powerful, freeware GUI debugging tool. Excellent for malware analysis and security research
- IDA Pro
  - Clunky interface and difficult to use
- PyDbg
  - Scripted debugger, sub-component of PaiMei

# What is a Disassembler?

**Definition**

A disassembler is a static-analysis tool that translates raw bytes into assembly language, essentially the inverse of an assembler.

- There are many x86 disassembler libraries
- All debuggers have disassembling capabilities
- The hardest aspect of disassembly is differentiating between code vs. data
- There are less options for a solid pure disassembler ...

# DataRescue IDA Pro

- The defacto standard in static analysis technology
- Supports multiple architectures
- Cross-platform, GUI and console
- Scriptable and pluggable
  - Lots of custom software written on this platform

# What is a Decompiler?

## Definition

A decompiler attempts to translate raw binary data into a higher level language than assembly. They generally fail miserably.

- *"You can't get the toothpaste back in the tube"*
  - ie: True decompilation is impossible
- Some tools exist, for the most part they provide you with a more readable disassembly

```
if(*(ebp + 12) == -1)
    eax = *( *(ebp + 8) * 4 + 0x422cc4);
else
    if((*(ebp + 12) & -8) != 0)
        eax = eax | -1;
    else
        *(ebp - 4) = *(*(ebp + 8) * 4 + 0x422cc4);
        ecx = *(ebp + 8);
```

# Tools

- REC
  - Useful (but unstable) command line tool
  - We will give you a private IDA plug-in that can import REC output
- REC Studio
  - GUI verson of REC
  - Even more unstable
- Desquirr
  - IDA plug-in
  - Generates decent results but only does so in the messages window
- Boomerang
  - Open source, worth keeping an eye on
- Hex-Rays
  - Most recent of the tools
  - IDA extension written by Ilfak of DataRescue

# Web Services

- Virus Total (virustotal.com)
  - Online AV multi-scanner
- Offensive Computing (offensivecomputing.net)
  - Malware zoo
  - Over 600,000 samples
- CWSandbox (cwsandbox.org)
  - COM, File, Mutex, Registry, Process information
  - Network activity overview
  - Human readable and machine parseable outputs (XML)
- Anubis (anubis.iseclab.org)
  - Same as CW, adds PCAPs

# Syntax and Data Types

Whitespace matters. Use *tabs* or *spaces*, don't mix. No block delimiters.

- This is an **int** (signed, 32bits) : 100
- This is a **long** (signed, infinite): 100L
- This is a **str**: "meow\x00\n" or 'meow\x00\n' (" = ')
- This is a **tuple** (immutable): (10, 20, "25")
- This is a **list** (mutable): [10, 20, "25"]
- This is a **dict** (mutable): {"one":1 , "two":2}
- This is a **set** (mutable): set([1, 2, 3, 4])

# Keywords

29 keywords to rule them all

| | | | |
|---|---|---|---|
| and | assert | break | class |
| continue | def | del | elif |
| else | except | exec | finally |
| for | from | global | if |
| import | in | is | lambda |
| not | or | pass | print |
| raise | return | try | while |
| yield | | | |

# Conditionals, Loops and Exceptions

```
if conditional:
    instruction
    instruction
elif conditional:
    instruction
else:
    instruction
    instruction
```

```
for x in set:
    instruction

for x in xrange(100):
    instruction

while conditional:
    instruction
```

```
try:
    instruction
except:
    instruction
else:
    instruction
```

- Python essentially reads like pseudo code
- Pencil + napkin + 10 minutes = Python code

# Functions and Classes

## Functions

```
def do_something (arg):
    instruction
    return something

def do_something (arg1, arg2=10, arg3=20):
    instruction

do_something(1, 5)
do_something(1, arg3=50)
```

## Classes

```
class my_class:
    def __init__ (self, arg):
        instruction

    def member_routine(self, arg):
        instruction
        return something

class inherit (my_class):
```

- Functions can take a variable number of arguments
- You can specify some or all of the optional arguments
- Use classes to define structures

# Outline

6 Overview of Analysis Tools

7 (Dis)Assembly
- Crash Course
- Assembly Patterns

8 IDA Pro

9 OllyDbg

# The Very Basics

- We look at Intel (vs. AT&T) style assembler
  - ex: MOV destination, source
- Brackets are like a pointer dereference
  - ex: *EAX = [EAX] = dword ptr [EAX]
- LEA is Load Effective Address
  - Transfers offset address of source to destination register
  - It's actually faster to use the MOV instruction
    - ex: LEA EAX, [EBX] vs. MOV EAX, EBX
  - You'll see LEA frequently used for basic math operations
    - ex: LEA EAX, [EAX*EAX] is equivalent to $EAX^2$

# The Very Basics Continued

- XOR-ing a register with itself zeroes it
  - ex: XOR EAX, EAX $\rightarrow$ EAX $= 0$
- Numbers, signed vs. unsigned
  - Unsigned positives range from 0 through 0xFFFFFFFF
  - Signed representation splits the range in half
    - Positives range from 0x00000000 through 0x7FFFFFFF
    - Negatives range from 0x80000000 through 0xFFFFFFFF (-1)

# Even More Basics

- PUSHA(D) / POPA(D)
  - Save / restore all registers
- PUSHF(D) / POPF(D)
  - Save / restore all CPU flags
- The 'D' instructions are 32bit, otherwise 16bit

# Memory Addressing Modes

- Immediate addressing mode
  - `int big_num = 0xDEADBEEF;`
  - `MOV EAX, 0xDEADBEEF`
- Direct addressing mode
  - `int big_num = 0xDEADBEEF; b;`
  - `b = *( &big_num );`
  - `MOV EAX, [0x40508c]`
- Indirect addressing mode
  - `char first_initial = *name;`
  - `MOV AL, [EBX]`
- Indexed addressing mode
  - `int chosen = char_array[0xC01a];`
  - `MOV EAX, [EBX+0xC01A]`
- Scaled indexed addressing mode
  - `some_struct *mystruct = struct_array[20];`

# Instructions You Need to Know About

- INC, DEC, ADD, SUB, MUL, DIV
  - Basic math instructions
- MOV, LEA
  - Manipulate memory and registers
- CALL / RET, ENTER / LEAVE
  - Transfer control to / back from another function
- CMP, TEST
  - Compare values
- JMP, JZ, JNZ, JG, JL, JGE, etc...
  - Transfer control flow to another instruction
- PUSH, POP
  - Put items on / take items off of the stack
- AND, OR, XOR, SHL, SHR
  - Bitwise operations

# CPU Flags You Need to Know About

- CPU flags stored in a 32-bit EFLAGS register
- ZF, Zero Flag
  - Updated when the result of an operation is zero
  - Used for JZ, JNZ etc..
- SF, Sign Flag
  - 1 denotes negative number
  - 0 denotes positive number
- OF, Overflow Flag
  - Used in combination with ZF/SF for JG, JGE etc ...
- CF, Carry Flag (also an overflow indicator)
  - Used in combination with ZF for JA, JAE etc ...

# Calling Conventions

- cdecl
  - Caller cleans up the stack
- stdcall
  - Callee cleans up the stack
- fastcall
  - Arguments can be passed in registers
  - Microsoft: ECX and EDX (stdcall stack handling)
  - Borland: EAX, ECX and EDX (also stdcall)
- thiscall
  - C++ code, object pointer is passed in a register
  - Microsoft: ECX
  - Borland / Watcom: EAX
- naked

# Function Calls

## stdcall

```
some_function(arg1, arg2);

push arg2
push arg1
call some_function
mov [ebp-0xc], eax
```

## cdecl

```
some_function(arg1, arg2);

push arg2
push arg1
call some_function
add esp, 8
mov [ebp-0xc], eax
```

- Function arguments are pushed in inverse order
- The above-left snippet demonstrates stdcall calling convention.
  - The *callee* cleans up the stack
  - Recall that with cdecl the *caller* cleans up the stack
- Can anyone guess why there is a need for stdcall vs cdecl?

# EBP-Based Framing

## Traditional

```
push ebp
mov ebp, esp
sub esp, 0x100
```

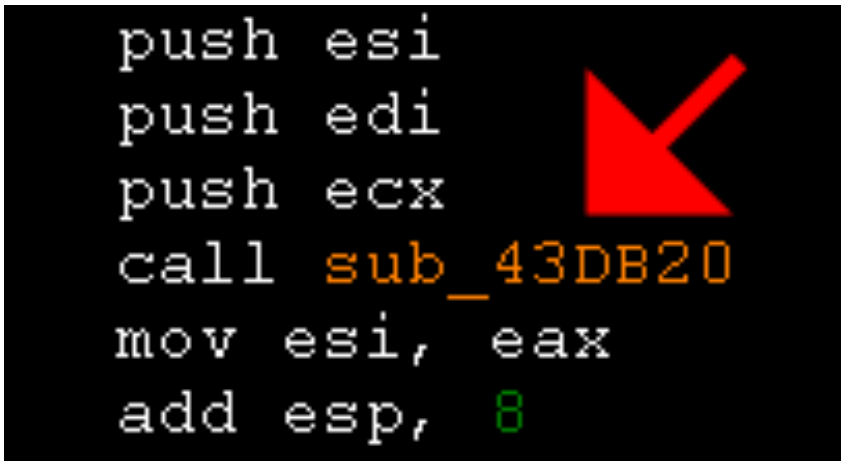
## Recent OS DLLs

```
mov edi, edi
push ebp
mov ebp, esp
```

- Optimized compiles may omit the frame pointer
  - In which case, local variables are referenced from ESP
- `mov edi, edi`
  - Effectively a 2-byte NOP
  - Why didn't they just use NOP, NOP?

# Return Values

- Saving the return value from a call



# 'P' - Parameters are Positive

- Parameters passed to a function are referenced Positive from EBP
  - `[EBP + values]` are typically arguments on the stack
  - `[EBP - values]` are typically local variables

# Branching

When reading a CMP statement, remember that the left side operand is the one being compared against. The trick is to think "is...":

```
ebx       = 0
[eax+14h] = 1

cmp     ebx, [eax+14h]  // think: is ebx...
ja      T1              // above? , 0 > 1 no jump
jb      T2              // below? , 0 < 1 jumps
je      T1              // equal? , 0 = 0 no jump
cmp    [eax+14h], ebx   // think: is [eax+14h]...
ja      T2              // above? , 1 > 0 jumps
jb      T1              // below? , 1 < 0 no jump
je      T2              // equal? , 1 = 0 no jump
```

# Conditional Blocks

```
if (length > 128)
    length = 128;

    mov eax, [ebp+8]
    cmp eax, 0x80
    jbe skip
    mov eax, 0x80
skip:
    ...
```

- JBE (Jump Below or Equal) is an unsigned comparison
- vs. JLE (Jump if Less than or Equal) is a signed comparison
- While we're on the subject of signedness
  - MOVSX vs. MOVZX

# Loops

```
for (i = 0; i < 100; i++)
    do_something();

    xor ecx, ecx
start:
    cmp ecx, 0x64
    jge exit
    call do_something
    inc ecx
    jmp start
exit:
```

- Loops are easier to spot in graphs
  - We'll talk more about this later
  - IDA Plug-ins exist for automating loop detection

# Switch Statements

```
switch (var) {
    case 0: var = 100; break;
    case 1: var = 200; break;
    ...

    jmp switch_table[eax*4]
case_0:
    mov [ebp-4], 100
    jmp end
case_1:
    mov [ebp-4], 200
    jmp end
    ...
end:
```

- Both IDA and OllyDbg have excellent switch/case detection

# Inline memcpy() / strcpy()

```
mov esi, source
mov edi, [ebp-64]
mov ebx, ecx
shr ecx, 2
rep movsd
mov ecx, ebx
and ecx, 3
rep movsb
```

- The rep movsd copies ECX dwords from ESI to EDI
- The rep movsb copies the remainder of the bytes

# Inline strlen()

```
mov edi, string
or ecx, 0xFFFFFFFF
xor eax, eax
repne scasb
not ecx
dec ecx
```

- The repne scasb The repne scasb instruction scans the string in EDI for the character stored in AL (NULL in this case)
- For each scanned character ECX is decremented and EDI is incremented
- At the end of the REPNE sequence
  - EDI = found char + 1
  - ECX = negative string length minus two
    - Logical not, minus one = string length

# Structure Access

```
push ebp
mov ebp, esp
mov eax, off_deadbeef
push ebx
mov ebx, [ebp+arg_0]
push esi
cmp ebx, [eax+14h]
push edi
ja short loc_12345678
cmp [eax+8], ebx
sbb esi, esi
```

- EAX is loaded from a global variable
- Also, [] is used with EAX, which means this global variable is a pointer

# Division

- The 'div' instruction is **incredibly** slow. Compilers don't use it, unless they need the remainder
- Instead, divides are typically seen as a combination of shift instructions
- The following instruction divides EAX by 4:
  - `SHR EAX, 2`

# Pseudo Random Numbers

## Slammer MSSQL Worm PRND Engine

```
MOV    0XFFFFFFB4(%EBP),%EAX ; eax = GetTickCount()
LEA    (%EAX,%EAX,2),%ECX    ; ecx = eax + eax * 2
LEA    (%EAX,%ECX,4),%EDX    ; edx = eax + ecx * 4
SHL    $0X4,%EDX             ; edx = edx << 4 (edx *= 2^4 (16))
ADD    %EAX,%EDX             ; edx += eax
SHL    $0X8,%EDX             ; edx = edx << 8 (edx *= 2^8 (256))
SUB    %EAX,%EDX             ; edx -= eax
LEA    (%EAX,%EDX,4),%EAX    ; eax = eax + edx * 4
ADD    %EBX,%EAX             ; eax += ebx
MOV    %EAX,0XFFFFFFB4(%EBP) ; store target address
```

- Sorry, this is a different flavor of assembly (old excerpt)
- "Random" numbers are frequently generated via large multiplications that rely on integer wrapping
- The above excerpt is equivalent to multiplying by 214,013

# Type Recovery

- There is no type representation at the assembly level
- We need to differentiate...
  - Strings from raw data
  - UNICODE strings from ASCII strings
  - Integers from pointers
  - Integers from booleans
- Sometimes type recovery requires examination across function boundaries
- Type recovery is simple for documented API calls

# Type Recovery: Integers vs. Pointers

- Integers are never dereferenced
- Integers are frequently compared, pointers are not
- Pointers are compared against zero or other pointers
- Arithmetic on pointers is simple, potentially complex for integers

# Type Recovery: Strings vs. Raw Data

- String copying is generally prefixed with a strlen() (or inline equivalent)
  - Raw data copies will not have a prefixed strlen()
- Strings are often compared against readable characters and other strings
  - Raw data may not be
- Trace string parameters down to API calls!

# C++ (Object Oriented) Code

- As mentioned before, the this pointer is typically passed to function through ECX

```
lea ecx, [esp+16]
call member_routine
```

- Virtual Tables or VTables, are commonly seen in C++. Example:

```
mov eax, esi
push 0x100
call dword ptr [eax+50]
```

- When reversing arbitrary objects or structures locating initialization routines such as constructors / destructors can prove helpful

# Branchless Logic

- Compilers will avoid branching if possible
- Many simple compare / branch pairs can be converted into a sequence of arithmetical operations
- The usage of the `sbb` instruction is a typical indicator
- The resulting code runs faster, but is not as readable for human analysis

# Branchless Logic: Example

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0

# Branchless Logic: Example

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)

# Branchless Logic: Example

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1

# Branchless Logic: Example

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1
- Otherwise if eax is greater than 0 we have eax = eax - eax+0 = 0

# Branchless Logic: Example

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1
- Otherwise if eax is greater than 0 we have eax = eax - eax+0 = 0
- The increment will set the possible eax values to 1 or 0

# Mastering These Concepts

- Reverse engineering isn't as much a matter of difficulty as it is a matter of familiarization and practice
- The best way to learn is hands on experience
  - Keep a quick reference handy (see opcodes.hlp)
  - Single step with a debugger
  - Use an emulator
  - Disassemble familiar code
    - Compile and disassemble small snippets of code

# Outline

6 Overview of Analysis Tools

7 (Dis)Assembly

8 IDA Pro
- Overview
- Overview of Views
- Driving IDA
- Customizations

9 OllyDbg

# Introduction to IDA Pro

- IDA is the tool for reverse engineering
- IDA is commercial, costs roughly $450 US
- Scriptable through IDC / IDAPython
- Pluggable through a multitude of languages
- Interactive Disassembler Pro
  - It's named interactive for a reason. IDA makes mistakes and won't recognize everything correctly
  - However, you can interact with the database and correct the errors
- FLIRT
  - Fast Library Identification and Recognition Technology
  - Allows IDA to recognize standard library calls

# Executable Files

idag.exe ..... Microsoft Windows GUI
idaw.exe ..... Microsoft Windows text interface
idau.exe ..... Microsoft Windows / MSDOS generic text interface
win32_remote.exe ..... Remote Windows debugger client
linux_server / linux_server64 ..... Remote Linux debugger client

# File Types

.CFG ..... Configuration file
.IDC ..... IDA Script
.IDB ..... IDA Database

- Processor modules
  - Windows: .W64, .W32, .D32, .DLL
  - Linux: .IL64, .ILX
- Loader modules
  - Windows: .L64, .LDW, .LDX, .LDO
  - Linux: .LLX64, .LLX
- Plug-in modules
  - Windows: .P64, .PLW, .PLD, .PL2
  - Linux: .PLX64, .PLX

# IDA Autoanalysis Algorithm

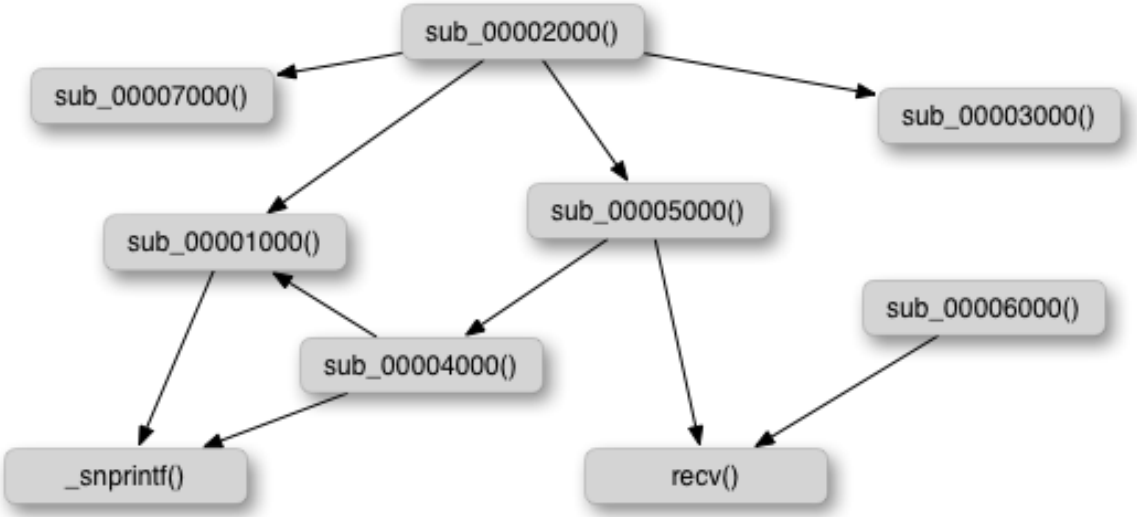
The autoanalysis algorithm is not documented, but it can be roughly described as follows (this was verified by Ilfak):

1. Load the file in the database and create segments
2. Add the entry point and all DLL exports to the analysis queue
3. Find all typical code sequences and mark them as code. Add their addresses to the analysis queue
4. Get an address from the queue and disassemble the code at that address, adding all code references to the queue
5. While the queue is not empty, go to 4
6. Make a final analysis pass, converting all unexplored bytes in the text section to code or data

[Sotirov, 2006]

# Call Graphs

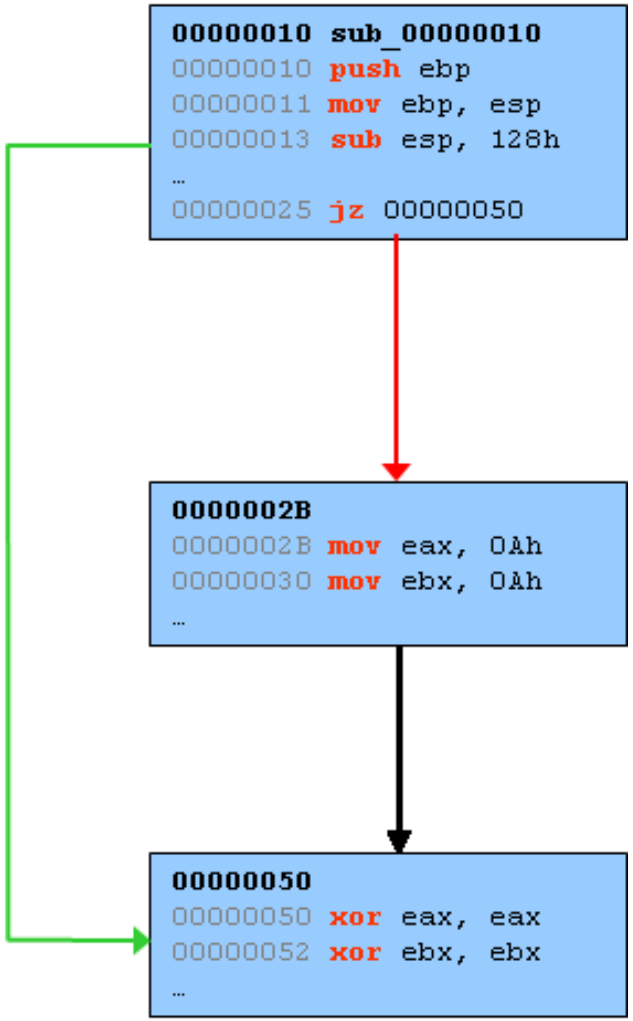
- Disassembled binaries can be visualized as graphs
  - Functions represented as nodes
  - Calls represented as edges
- Useful for viewing the relationships between functions



# Control Flow Graphs

- Functions can be visualized as graphs
  - Basic blocks represented as nodes
  - Branches represented as edges

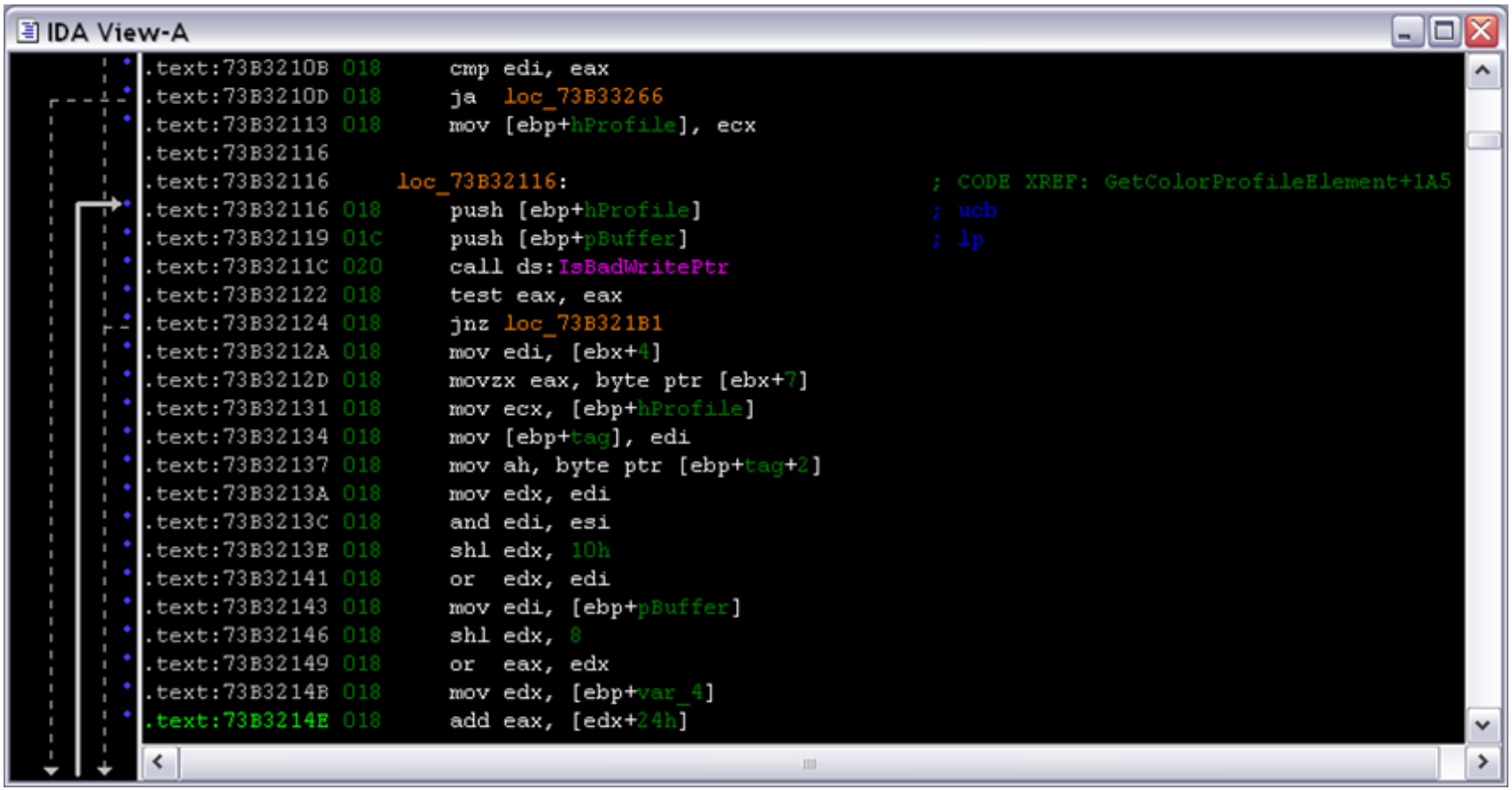
```
00000010 sub_00000010
00000010 push ebp
00000011 mov ebp, esp
00000013 sub esp, 128h
…
00000025 jz 00000050
0000002B mov eax, 0Ah
00000030 mov ebx, 0Ah
…
00000050 xor eax, eax
00000052 xor ebx, ebx
…
```




- Useful for viewing of execution paths

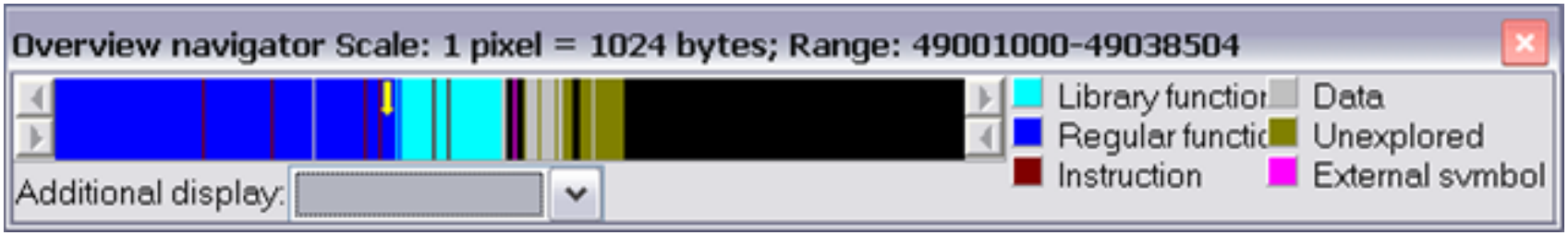
# Disassembly View

- View disassembly, stack offsets, cross refs, strings refs...
- Set breakpoints, change names, apply structures edit functions...



# Navigator Band

- Colorized view of the loaded binary

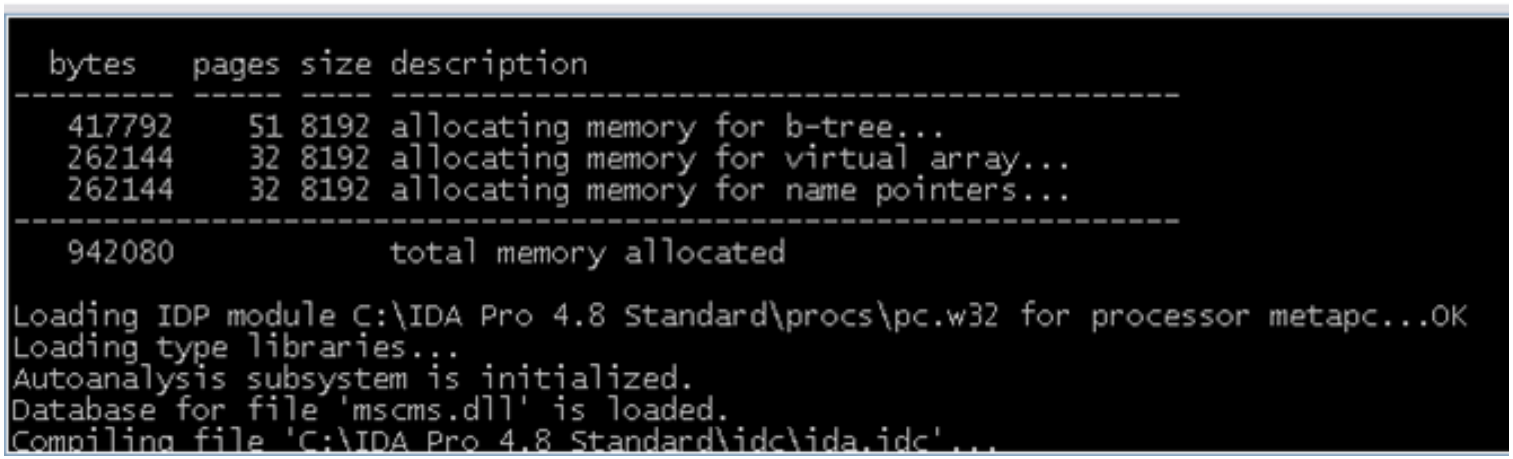



**Tip**

Bands of code that lie interlaced within library functions are probably unrecognized library routines.

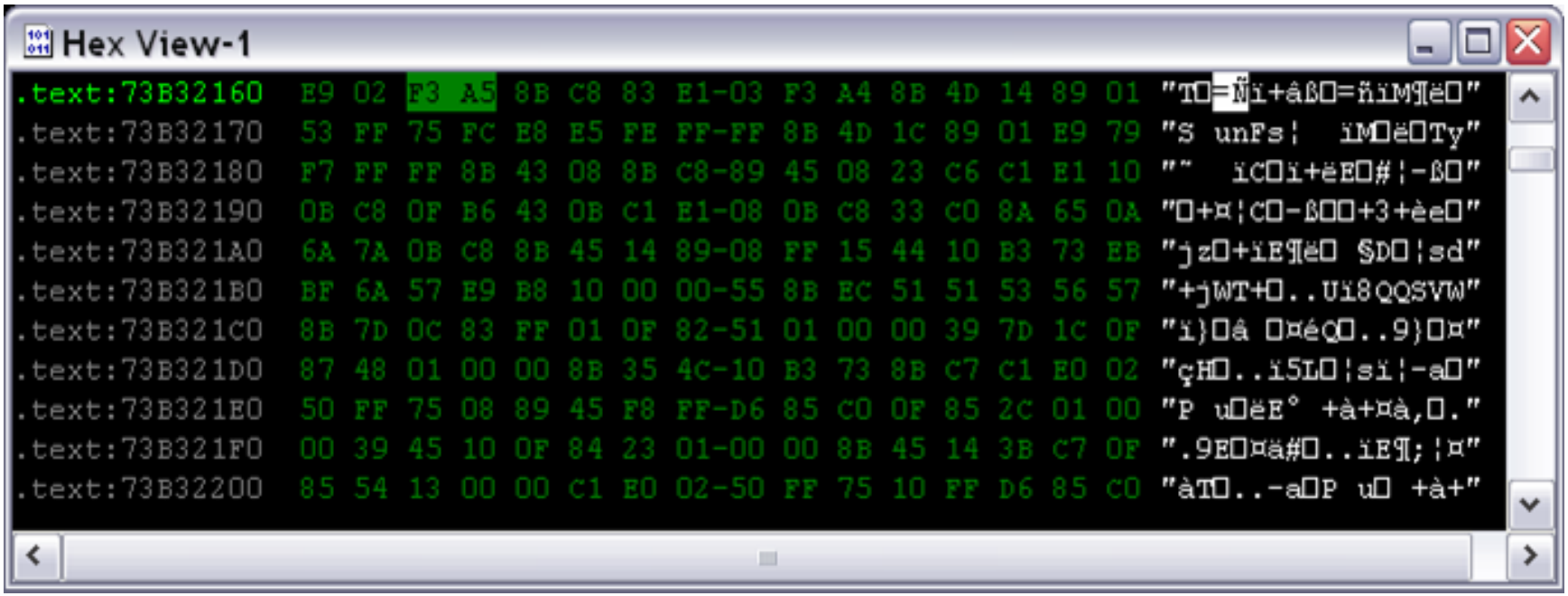
# Messages Window

- Contains informational messages from IDA
- Output from plug-ins
- Output from command bar



# Hex View

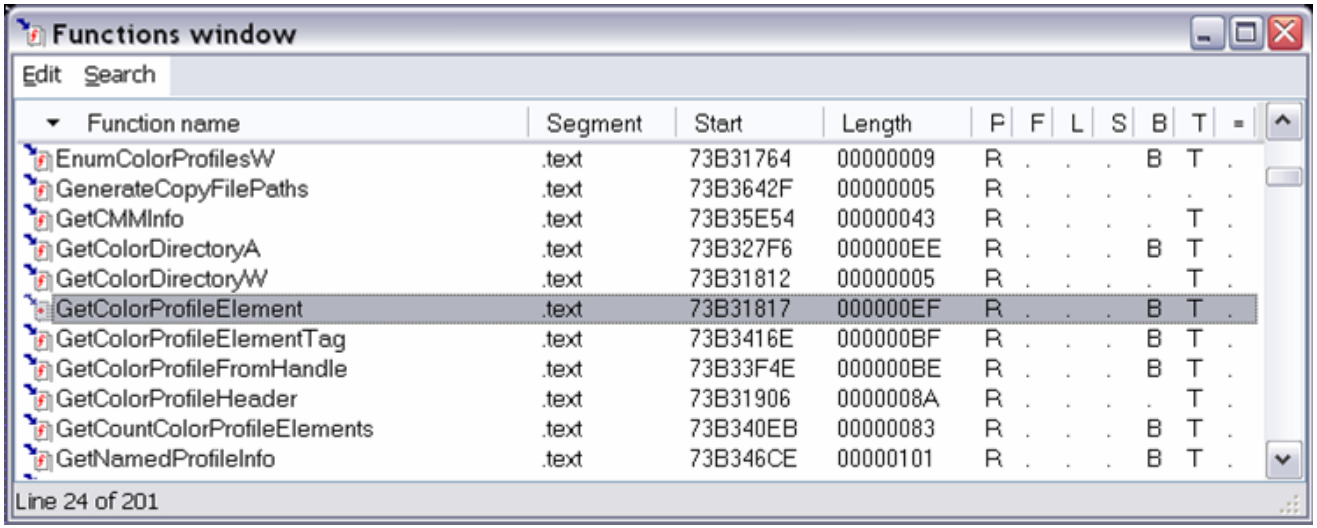
- Hex dump of binary
- Can be synchronized with disassembly view



# Function List

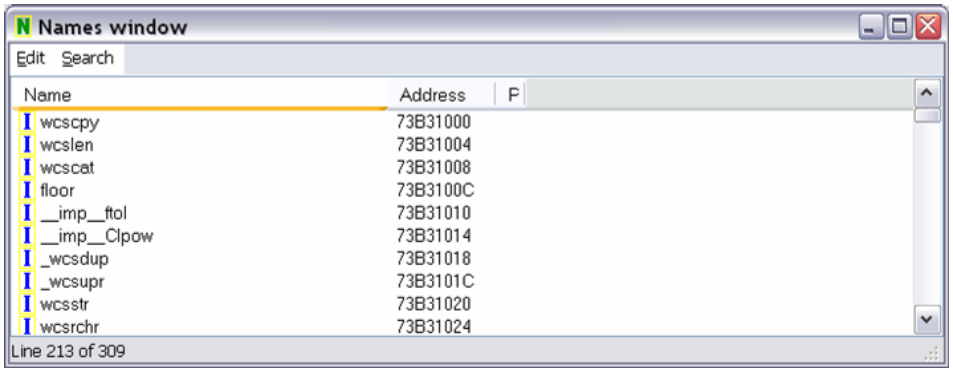
List of all functions defined in the current binary

| | |
|---|---|
| R | Function returns to caller |
| F | Far function |
| L | Library function |
| S | Static Function |
| B | EBP based frame |
| T | Function has type information |
| = | Frame pointer is equal to the initial stack pointer |



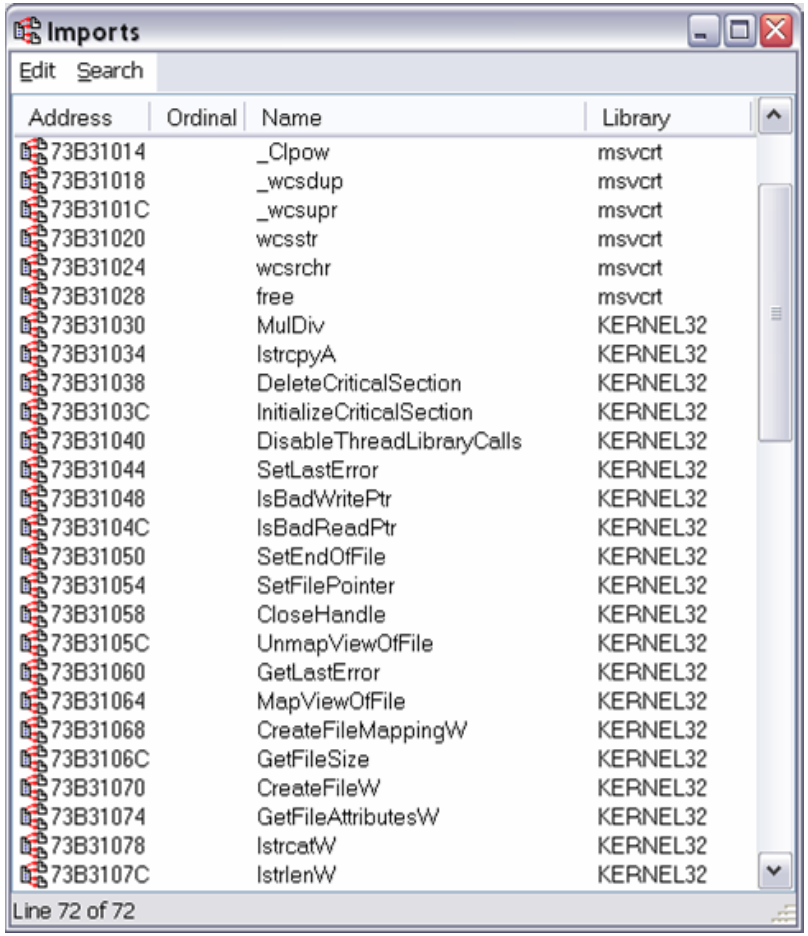
# Names / Symbols List

| | |
|---|---|
| L | Library function |
| F | Regular function |
| C | Instruction |
| A | ASCII string |
| D | Data |
| E | Imported name |



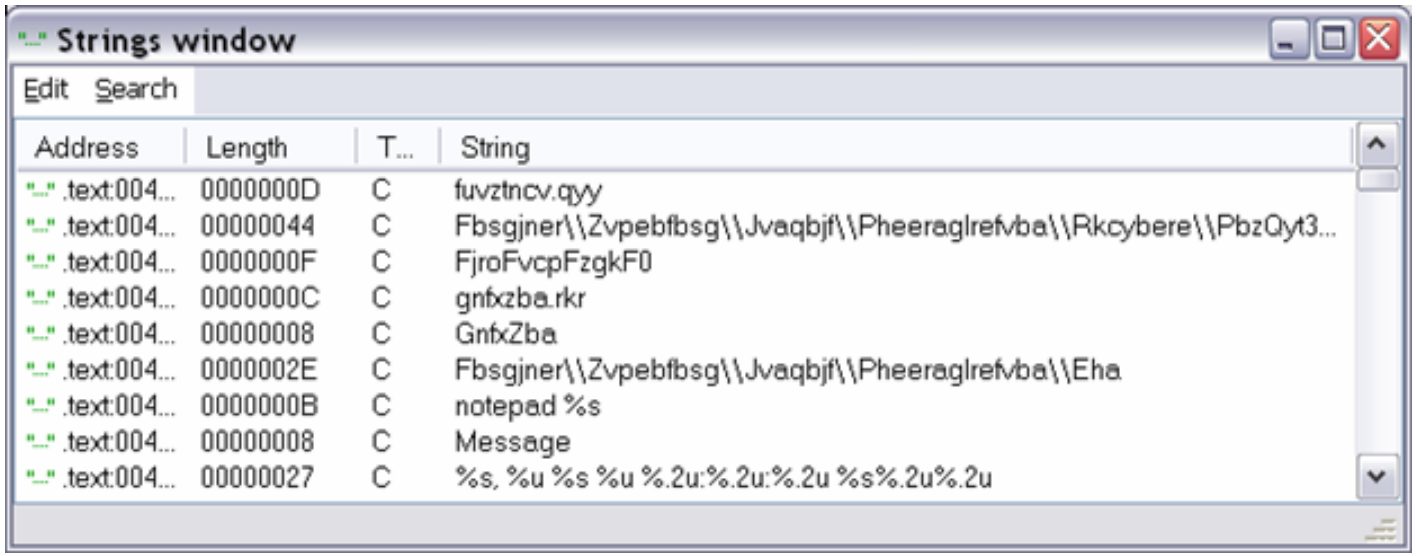
# Imports List

- View imported libraries
- View imported API
- Search
- You can make an educated guess of the capabilities and functionality simply by analyzing the import table



# Strings List

- View and search the complete list of discovered strings
- Take a look at the string list from below. Is it obvious to anyone that some strings are encoded?



# Notepad

- Useful for jotting down notes / comments
- Saved in your IDB



# Debugger

- Worth mentioning
- Personally, I think it's hideous
- Key benefits
  - Can utilize the power of IDC on breakpoints
  - Powerful plug-in API
  - Excellent disassembly base of course
- On another note, it is good for ARM (Pocket PC) debugging
  - Pedram's new hobby

# Renaming Variables

Use the N key to rename the argument

```
0042CA70      TRACE_COMMAND_DISPATCH proc near          ; DATA XREF:
0042CA70
0042CA70      var_40= dword ptr -40h
0042CA70      sscanf_buffer= dword ptr  8
0042CA70
0042CA70 000      sub esp, 40h
0042CA73 040      mov byte_4B17C8, 1
0042CA7A 040      push esi
0042CA7B 044      mov esi, [esp+44h+sscanf_buffer]
0042CA7F 044      push offset str_Off_0                   ; "&OFF&"
0042CA84 048      push esi                                ; char *
0042CA85 04C      call ds:strstr
0042CA8B 04C      add esp, 8
0042CA8E 044      test eax, eax
0042CA90 044      jz  short loc_42CAA0
```

# Navigating the Dead Listing

- CTRL+UP/DOWN to scroll without losing highlight
- CTRL+LEFT/RIGHT to jump between items
- SHIFT+ENTER to highlight the current item
- ALT+UP/DOWN to find the last/next occurrence of the current item

# Marking Positions

- ALT-M mark position
- CTRL-M jump to marked position
- To clear marks:
  - Jump → Clear mark, then select the mark to erase

# Cross References

- Using the Names Window, double click on an API call that you want to find in the target program. This will highlight the call in the jump table within the disassembly window
- Now click on the cross-references button. This will bring up a cross-references window
- By double clicking on the XREF, you will bring focus to the line of code that is making the call
- Shortcuts
  - X show all XREFs to operand
  - CTRL+X show all XREFs to current EA

# Forward and Backward Navigation

- By double clicking or highlighting and pressing return, you can follow a XREF directly.
- Sometimes multiple XREFs will be shown in the deadlisting
- If you have followed multiple XREFs you can navigate back through your XREF history by pressing ESC.
- You can navigate forward through your XREF history by pressing CTRL+Enter
- Using the arrow keys, the ENTER key, and the ESC key, you can navigate the disassembly view quickly.

# Overview

- Toolbars
- Custom desktops
- Color palette
  - IDA Pro\Customizations\IDA Color Palette.reg
- Others (next couple of slides)
  - IDA.CFG
  - IDAGUI.CFG
  - IDC scripts and hotkeys (IDA.IDC)

# IDA.CFG

```
ASCII_PREFIX        = "str."
MAX_NAMES_LENGTH    = 128
NameChars           = "$?@>"
SHOW_XREFS          = 200
SHOW_BASIC_BLOCKS   = YES
SHOW_SP             = YES
MangleChars         = "$:?([.)] "
```

# IDAGUI.CFG

```
HELPFILE                = "c:\OPS.HLP"
DISPLAY_PATCH_SUBMENU = YES
DISPLAY_COMMAND_LINE  = YES
"ChartXrefsTo"          = "Ctrl-Shift-T"
"ChartXrefsFrom"        = "Ctrl-Shift-F"
```

# IDA.IDC

```
#include <pedram_function_tagger.idc>
#include <pedram_jump_to_func_top.idc>
#include <pedram_export_disassembly.idc>
#include <pedram_assign_color.idc>

AddHotkey("Ctrl-Shift-X",     "export_disassembly");
AddHotkey("Ctrl-Shift-J",     "jump_to_func_top");
AddHotkey("Ctrl-Shift-Enter","track_follow");
AddHotkey("Ctrl-Shift-N",     "track_name");
AddHotkey("Ctrl-Shift-A",     "hotkey_assign_color");
AddHotkey("Ctrl-Alt-A",       "hotkey_deassign_color");
AddHotkey("Ctrl-Shift-B",     "hotkey_assign_block_color");
AddHotkey("Ctrl-Shift-D",     "hotkey_deassign_block_color");
```

# Plugins

- There are a number of plug-ins available
- For starters, install pGRAPH.plw
  - Copy it to %IDA%\plugins
- Use ALT+3 to launch the plug-in
- For those of you who are interested, see the next slides for instructions on installing IDA Sync

# Exercise

- Customize your IDA environment
- Install IDA Python and other plug-ins of choice
- Use shortcuts!
  - See CD\IDA Pro\IDA Pro Shortcut Keys Quick Reference.pdf

# Outline

6 Overview of Analysis Tools

7 (Dis)Assembly

8 IDA Pro

9 OllyDbg
- Overview
- Overview of Views
- Driving OllyDbg

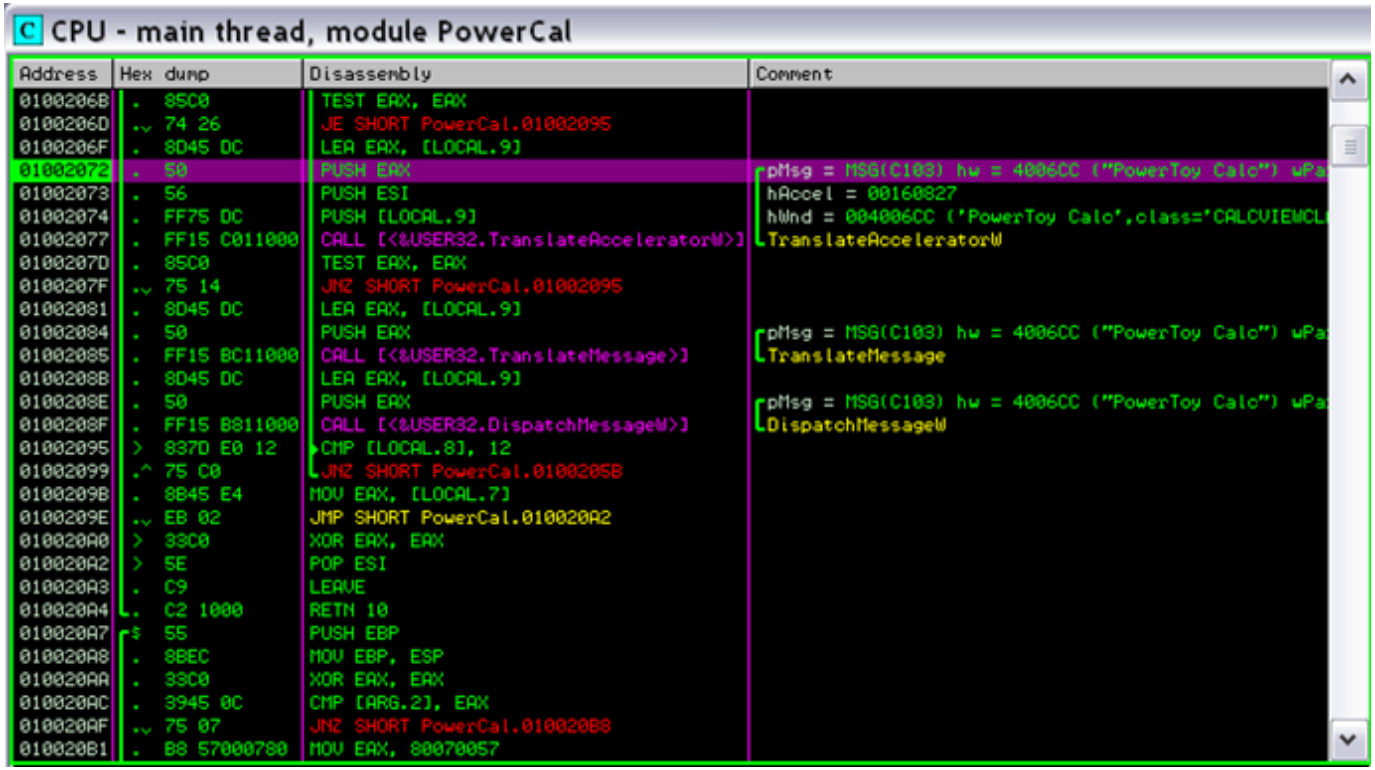
# Introduction to OllyDbg

- Freeware
- Contains a lot of *cracker friendly* features
- Pluggable through a multitude of languages
- Scriptable through OllyScript
- Contains a powerful disassembler
  - Also available as a library

# Learning Curve

- Very daunting on first launch
- There are tons of *hidden* features
  - I am still finding new ones
- Fortunately there is excellent documentation
- The best way to learn however ...
  - Open something in OllyDbg
  - Play around with the various features
  - Explore the various windows
  - Customize your environment with the features you will be using

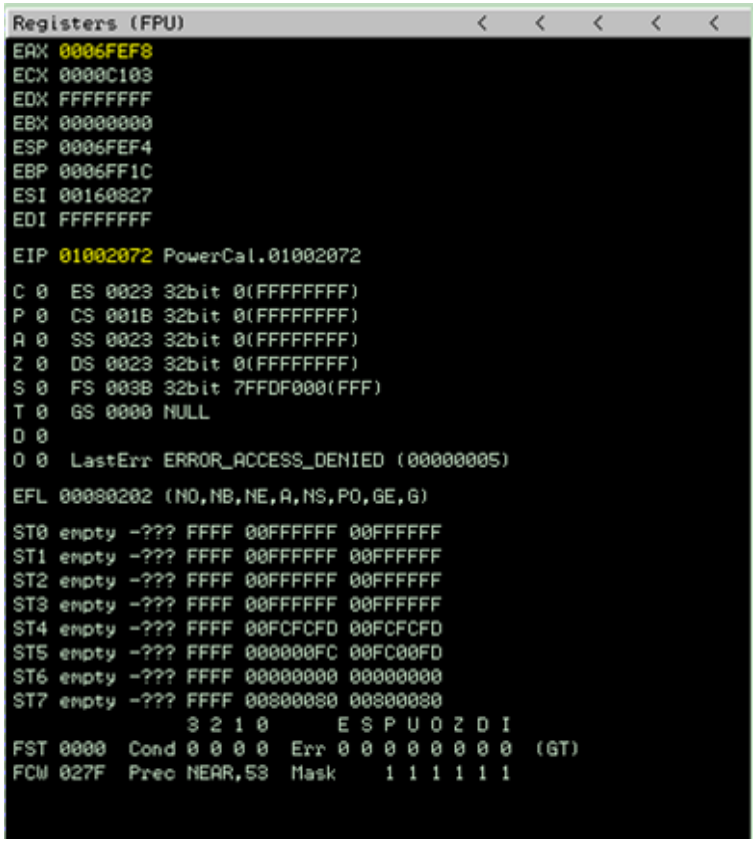
# CPU Main Window




- Comment-able disassembly
- Argument enumeration
- Smart de-referencing

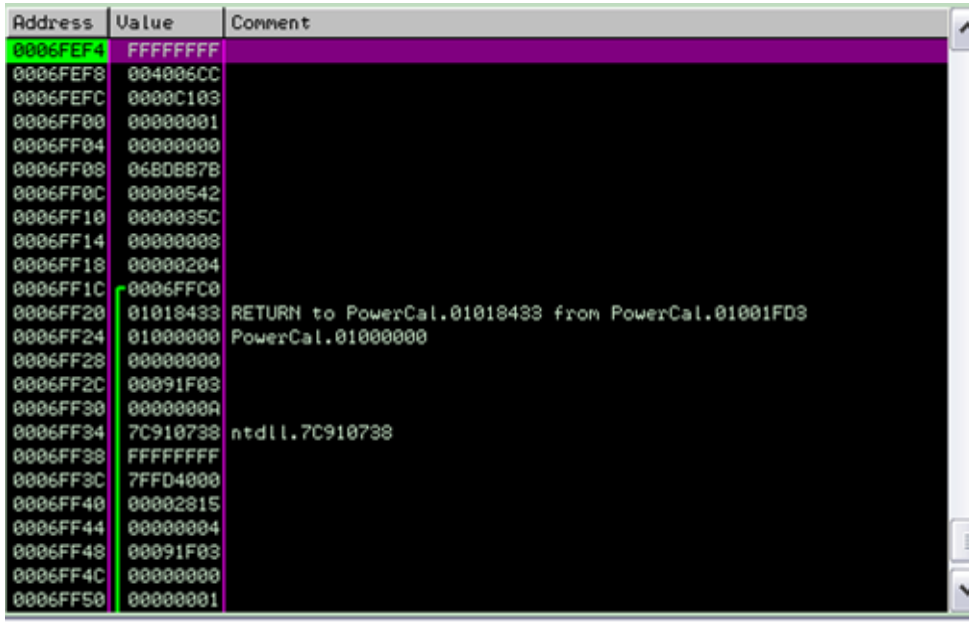
# CPU Registers

- Editable register values
- CPU flags
- Smart de-referencing
- Floating point, MMX, 3DNow! Support
- Follow in stack/dump shortcuts



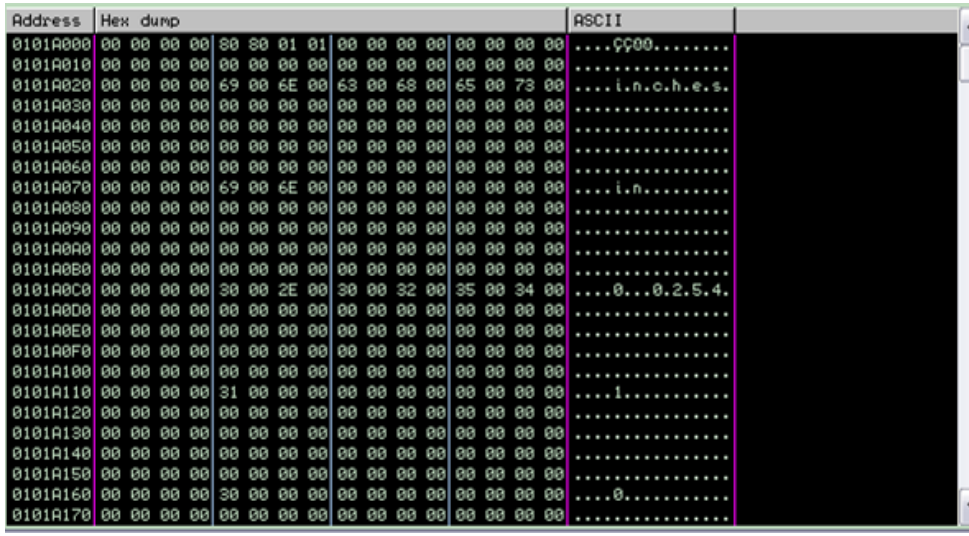
# CPU Stack

- Editable
- Lockable
- View relative to ESP/EBP
- Smart de-referencing
- SEH chain
- Saved frames
- ASCII/Unicode dump

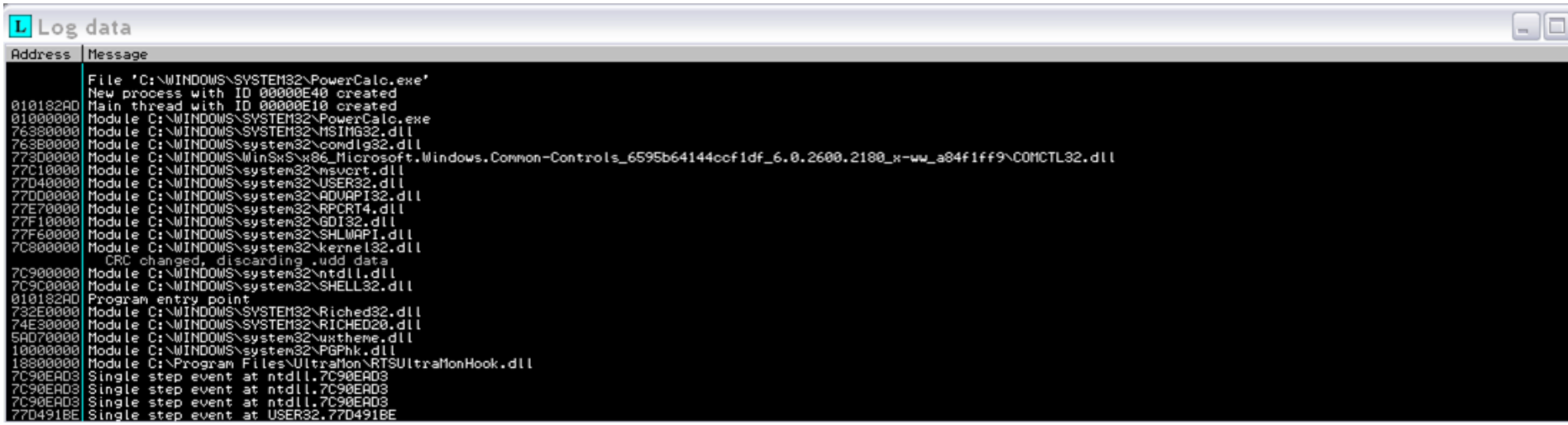


# CPU Dump

- Unique and handy OllyDbg feature
- Multiple view types: hex, text, disassembly, long pointers, PE header and it's extensible with plug-ins (see SPECFUNC)

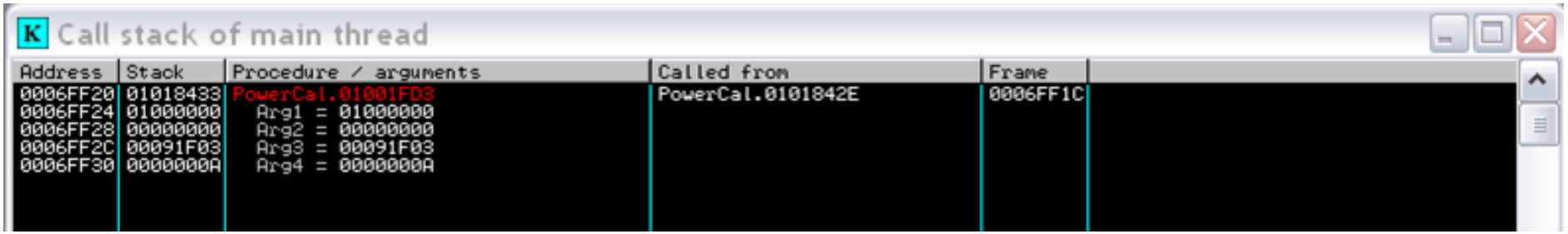


# Log Window



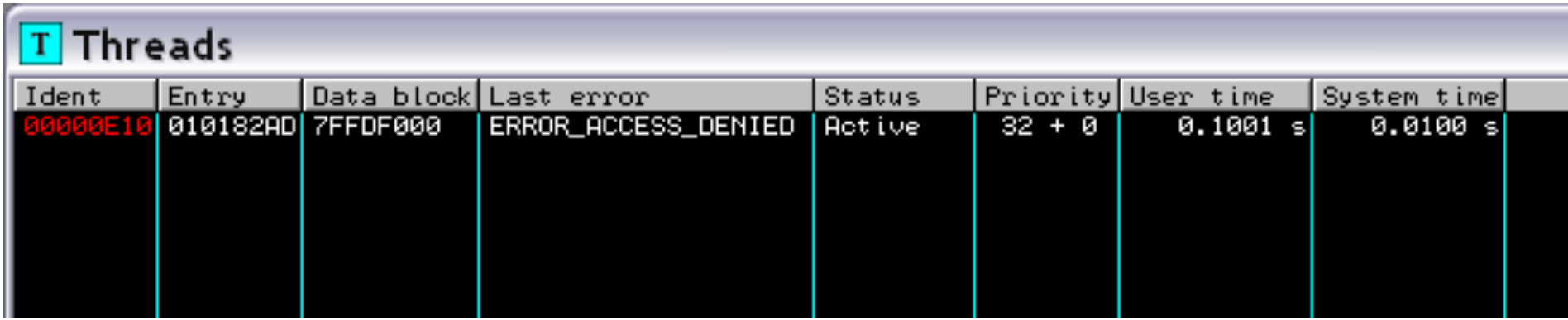

- Debugger / plug-in output messages
  - Breakpoint / log breakpoint output
  - Access violations
  - Etc...
- Debuggee messages through OutputDebugString() API

# Call Stack




- Collapsible with arguments
- Stack walking is not an exact science, especially when standard EBP based frames are not used.
- Olly is pretty smart about it

# Threads

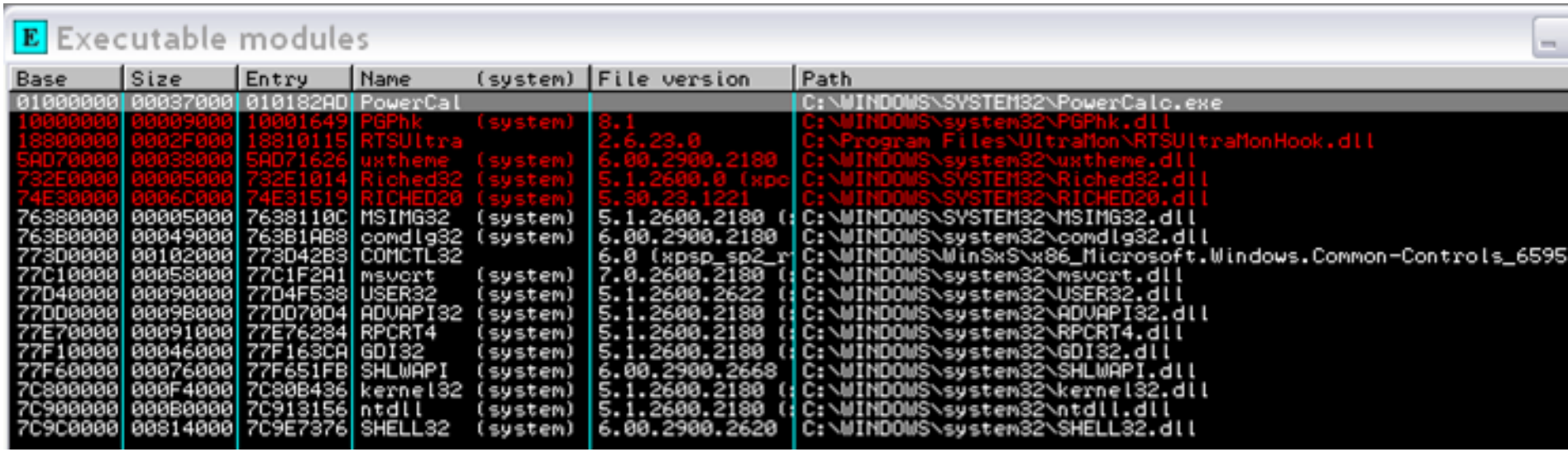



- List of threads in current process

**Tip**

When attempting to determine which thread processes an input, say for example network packets, fire off some packets forcing the process to work and watch the User/System time columns.

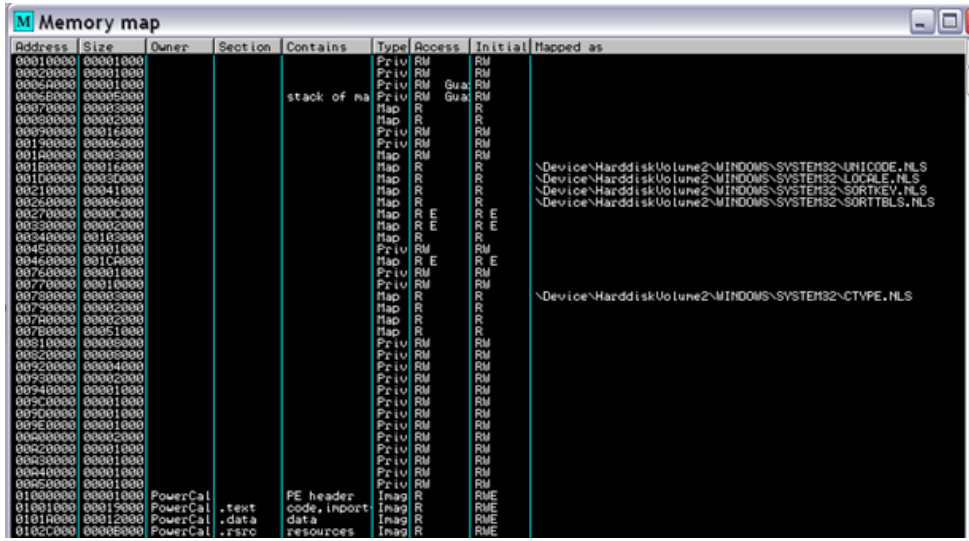
# Executable Modules



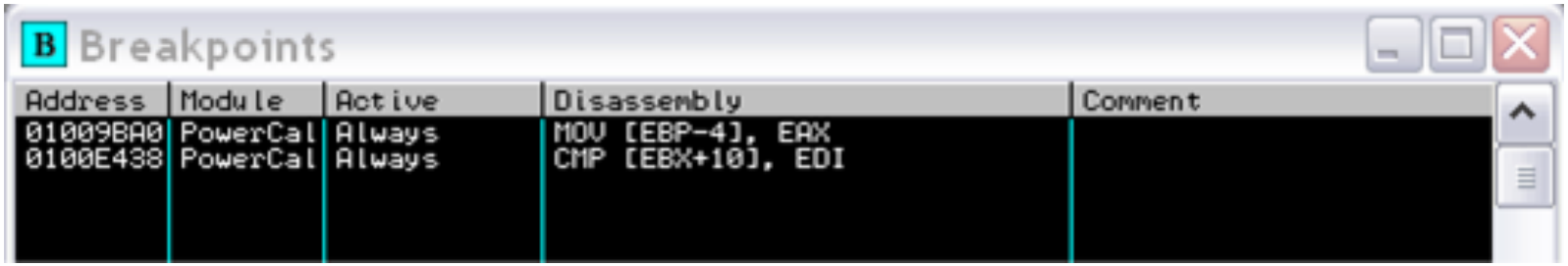

- List of modules currently loaded in the debuggee's process space
- The Name column contains the hidden system tag (run traces)
- Ctrl+N to view exported names
- Breakpoints / log breakpoints can be set
- Olly knows the argument types for a lot of standard API calls

# Memory Map

- Searchable full-range memory map
- Stack is tagged, heaps are not
- OllyDbg Heap Vis plug-in allows you to list, search and visualize the heap

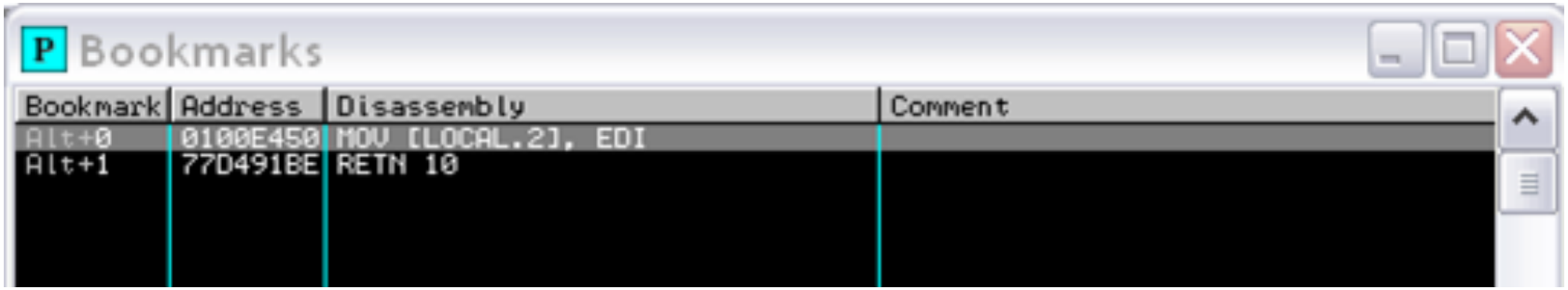


# Breakpoints




- OllyDbg supports regular breakpoints, conditional breakpoints and conditional log breakpoints
- Breakpoints set in the main module are persistent across debugger sessions.
- OllyDbg BP Manager plug-in allows you to import/export breakpoints lists
- You can use the BP Manager to automatically load breakpoints on module load

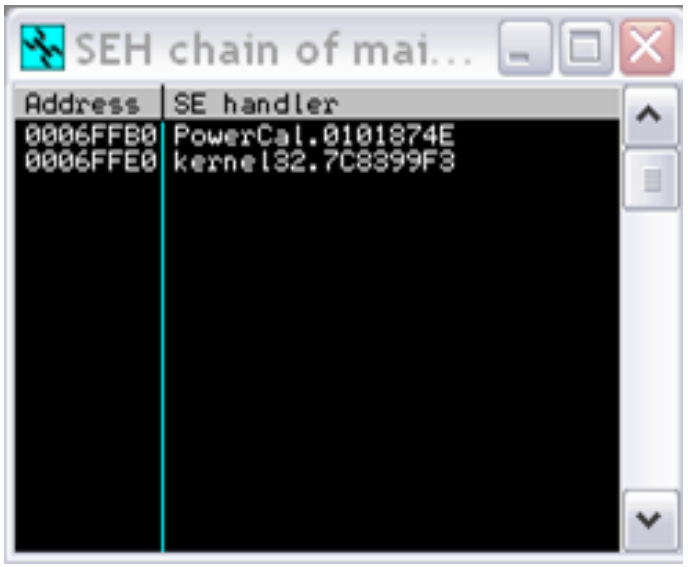
# Bookmarks




- Bookmarking is provided by a plug-in bundled with OllyDbg
- Simply set and quick jump to various locations
- Identical to "marks" in IDA

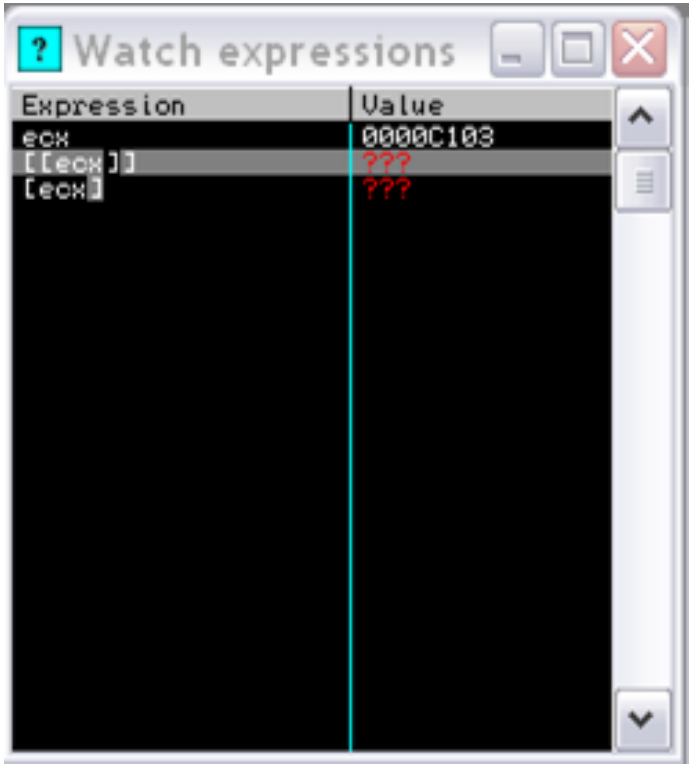
# SEH Chain

- Unique OllyDbg feature
- Does not show the vector based exception handling chain

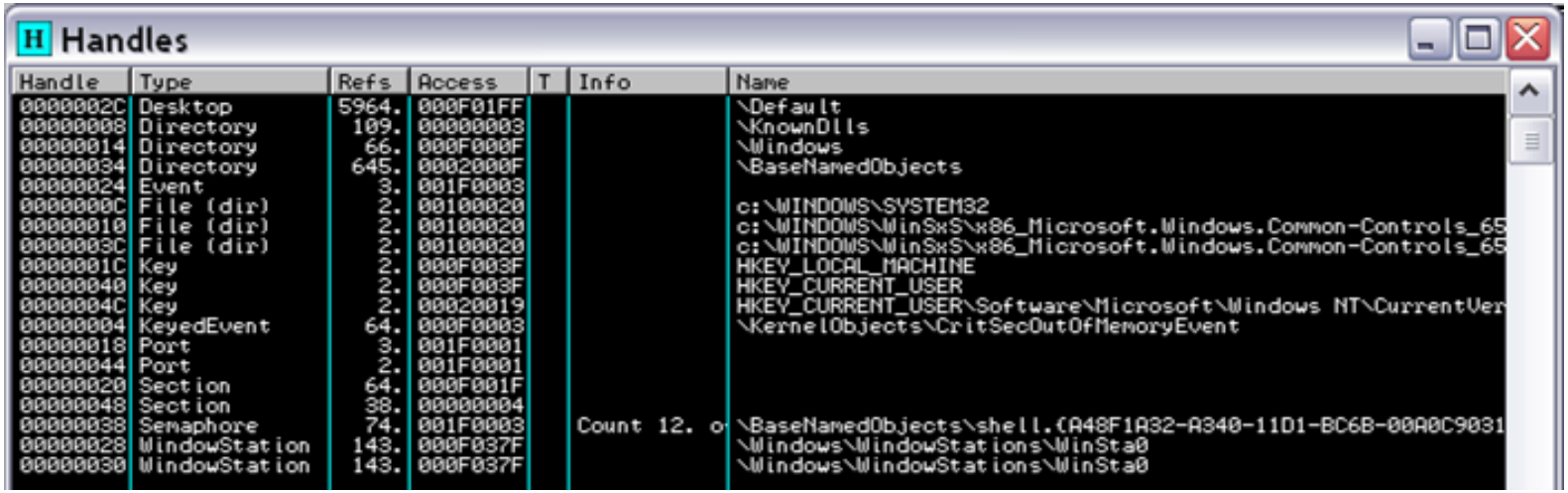


# Watch Expressions

- OllyDbg supports an expression language allowing the user to create real-time watches
- Useful when tracking changes to complex data types



# Handles




- List of open handles
- Tag / untag
  - Not sure entirely what this is for.

# Plug-ins

**Fact**

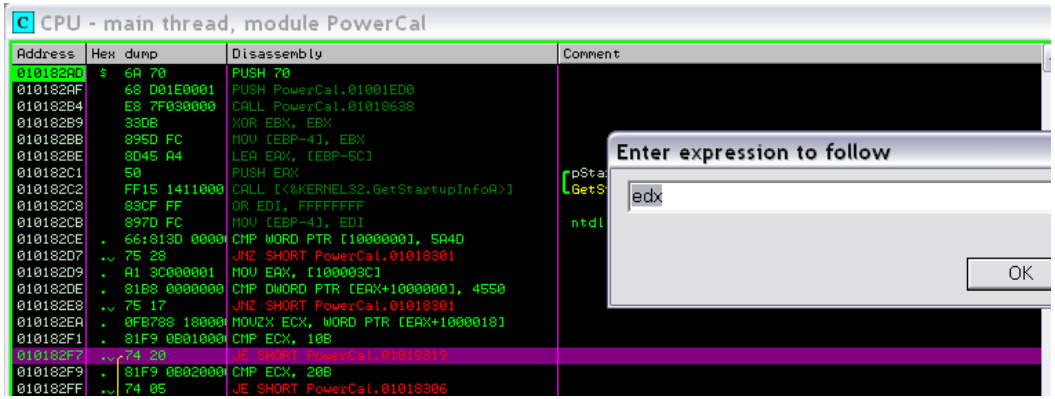
OllyDbg has excellent plug-in support and great API documentation. Much better than IDA.

- GODUP
  - Load labels and comments from IDA
- OllyDump
  - Dump the running process to disk, IAT rebuilding (malware purposes)
- Heap Vis
  - Enumerate and search process heap
- Breakpoint Manager
  - Import / export breakpoint sets
- De-Attach Helper
  - Adds detach support and WinDbg similar "attach to last" functionality

# Hot Keys

| | |
|---|---|
| F9 | Execute debuggee |
| CTRL + F9 | Execute until return |
| ALT + F9 | Execute until user code |
| F12 | Pause |
| F7 | Step into |
| F8 | Step over |
| F2 | Set / clear breakpoint |
| CTRL + G | Follow expression |
| any key | Add comment |
| : | Add label |

# Following Expressions

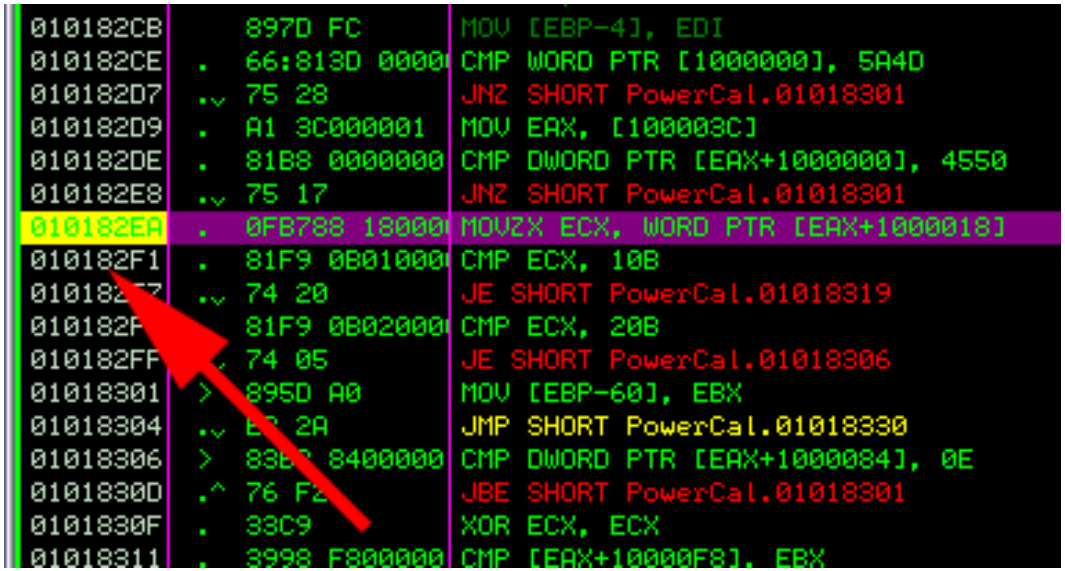



- Use CTRL+G to open the *follow expression* window
- You can specify an address, register, or the label of a function, such as recv or sprintf

**Tip**

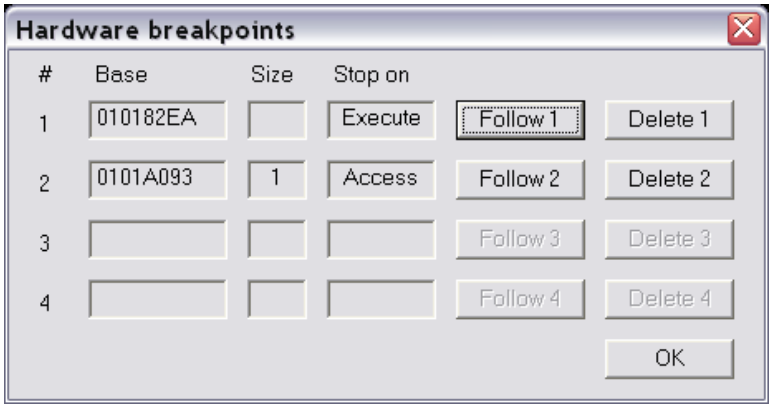
Sometimes you have to CTRL+G twice in order to get it to resolve to the actual location. This appears to be a bug.

# Software Breakpoints




- Set focus on the target instruction
- Hit F2 to set / unset the breakpoint

# Hardware Breakpoints

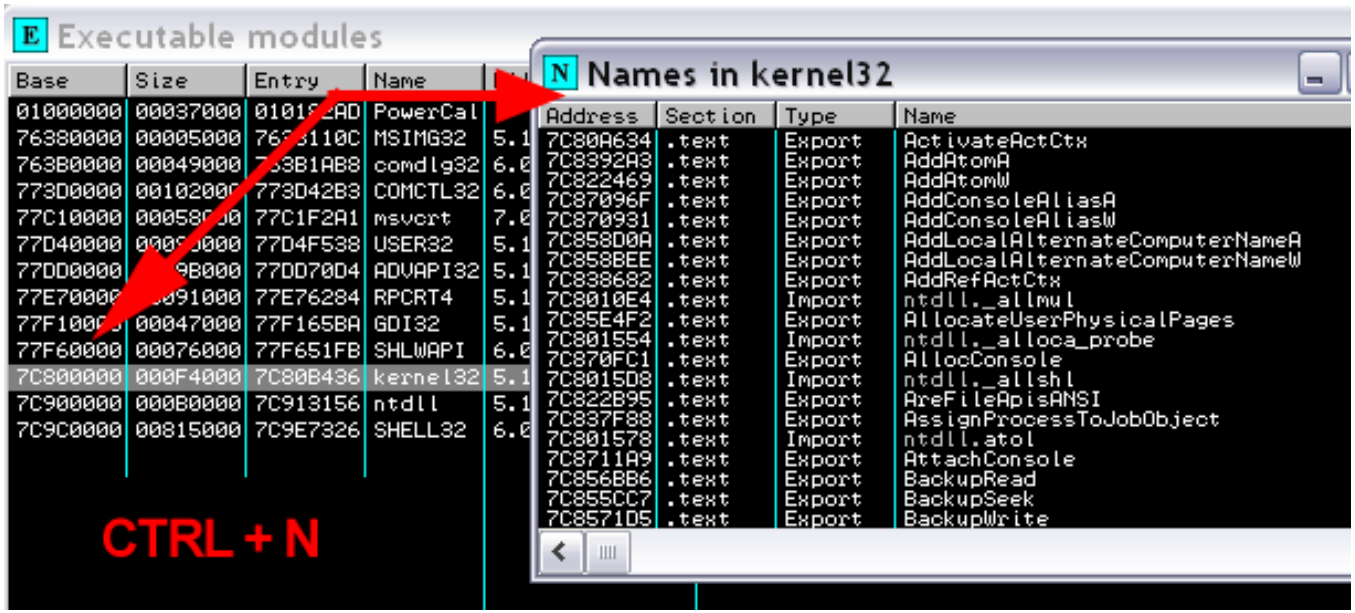



- You get 4 hardware breakpoints
- CPU supported, so transparent to software
- Set focus on the target instruction or memory address
- Right click → breakpoint → hardware
- Three classes: access, write and execute
- Three sizes: byte (1), word (2), dword (4)

# Memory Breakpoints

- Actually it's just memory breakpoint, you get one
- OllyDbg accomplishes this task by changing the permissions of the page that contains your target address
- You should opt to use hardware breakpoints over this feature
- In case you really need a 5th on-access type breakpoint, it's there for you

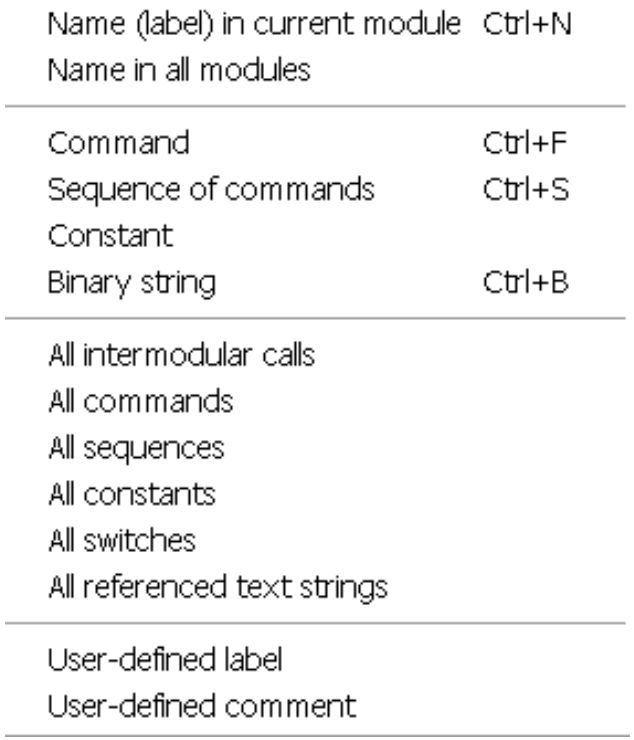
# Enumerating Functions




- Go to the *executable modules* window
- Highlight the DLL you are interested in
- Hit CTRL + N to pull up a list of imports/exports
- You can hit F2 directly on this list to set breakpoints

# Searching

- One of the best OllyDbg features
- You can search modules, memory, stack
- Search for strings or commands
- Search for variable register commands
- Search for all string references



# Run Trace and Animate

- These features are unique to OllyDbg
- Animate is useful for watching loop or recursion activity
- Run trace is useful for all sorts of tasks
  - Unfortunately run trace is extremely slow
  - Speed was a major motivating factor in my development of alternative code coverage techniques
  - Pedram can show you demo's of Process Stalker and PaiMei if we have time

# Part III

# Advanced Analysis

# Outline

10 Executable (Un)Packing
- Executable Packing
- Executable Unpacking
- Packer Usage Statistics
- Unpacking Traces

11 Anti Reverse Engineering

12 Binary Diffing and Matching

13 Advanced Malware Techniques

# Binary Protection

- Malware authors use both packers and crypters to reduce file size and increase analysis time
- Packers confuse disassembly by making code look like data
- A packer may contain anti-debugger/disassembler tricks such as SEH trickery and jumping into the middle of instructions
- A number of packers exist
  - PE: UPX, ASPack, tElock, yodacrypt, FSG, etc...
  - ELF: UPX, BurnEye, etc...
- We will create a basic packer to gain a better understanding of unpacking

# Packer Components

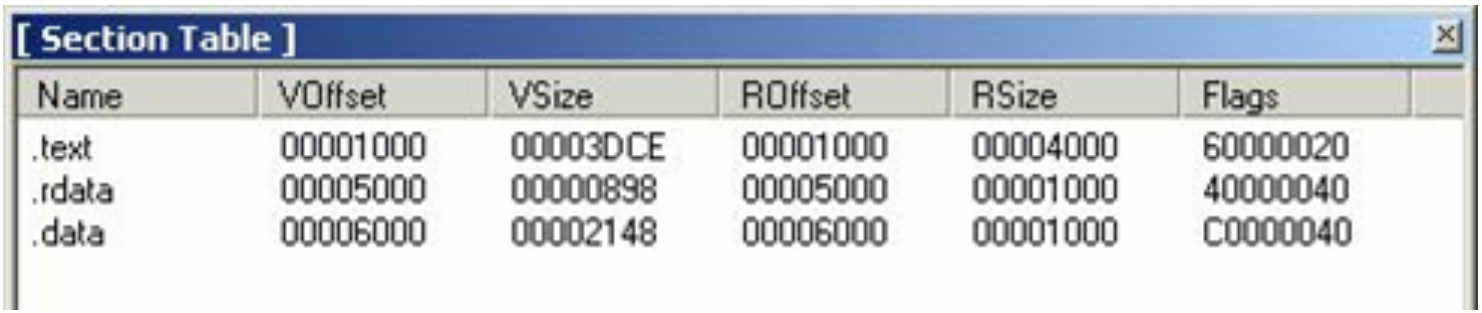
- The original executable code of the target application must be obfuscated or encrypted
- When the application is launched the packer decoder routine must be executed first (entry point modification)
- The decoder routine must restore the original executable code
- The decoder routine must transfer control to the original executable code
- To get a better understanding of this subject we're going to manually construct a packer

# What Packing is Not

- We don't consider run-time instrumented protection "packing"
  - Shiva
- Such protection schemes do exist, but are rarely used in malware
  - Probably because they require compile time modifications

# The Target Application

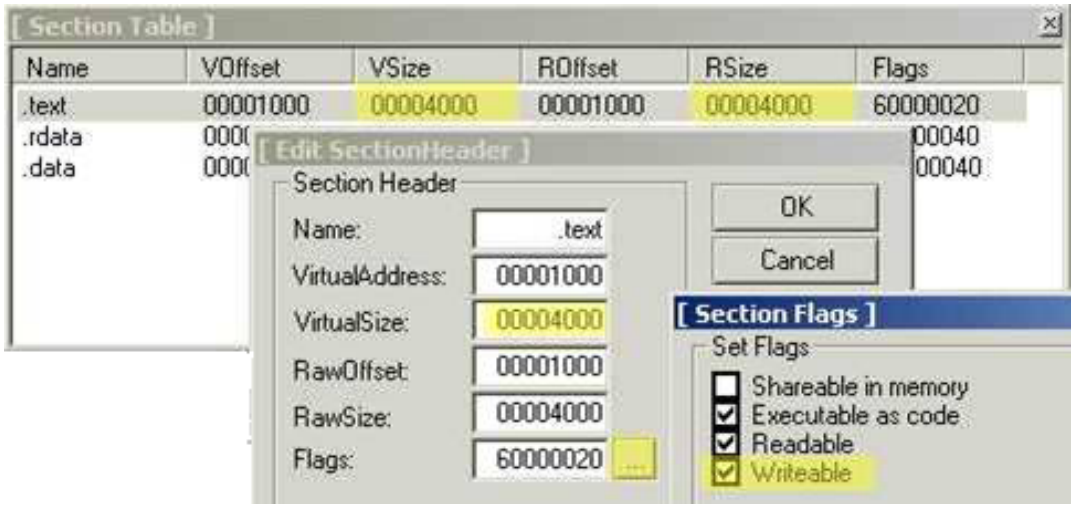
- original_hello.exe
  - Simple application written in C
  - Prints "Hello World" and waits for keypress
- Examine the PE structure information of the original executable:




- Notice the virtual size of 0x3DCE and the raw size of 0x4000 indicating that there should be 562 bytes of slack space at the end of the .TEXT section
- Verify this fact with a hex editor (offset 0x4DCE)

# Abusing Slack Space

- We are going to take advantage of the identified slack space by inserting our decoder stub into it
- PE modifications must be made
  - The .TEXT section must be made writable so our decoder stub can modify it
  - The VSize must be increased so our decoder stub gets mapped into memory



# Abusing Slack Space

- Next, we must modify the Entry Point (EP) to point to our decoder stub. Lets set it to 0x4E00
- Alternative strategies for storing a decoder stub include
  - New sections
  - Instruction gap abuse
  - Entirely new PE file

# Encoding the Executable Code

- For the purposes of this exercise we will manually encode the original executable code
- Simple XOR routine (VB):

```
StartAt = &H1048   'Original EP
length  = &H2D86   '3DCE - 1048 (Original VSize  Original EP)

Open filename For Binary As file

For i = 1 To length
    offset = StartAt + i
    Get file, offset, byte
    byte = byte Xor &HF
    Put file, offset, byte
Next

Close file
```

- Compiled helper app: do_opcode_crypt.exe

# Writing the Decoder Stub

- Our binary is now ready to accept a decoder
- We'll write the decoder in C, compile it and then extract the byte code

```
void main (void)
{
    int i;
    char b;

    char *buffer = 0x400000;   // imagebase
    long length  = 0xBEEF;     // length of code (placeholder)

    buffer += 0xDEAD;          // OEP offset (placeholder)

    for(i = 0; i < length; i++)
    {
        b = buffer[i];
        b = b ^ 0xF;
        buffer[i] = b;
    }

    _asm jmp buffer
}
```

# Converting the Decoder Stub

- Compile the decoder stub in Visual C
- Set a breakpoint at the top of the code
- Enter the debugger with F5 and choose goto disassembly
- Disassembly excerpt:

```
EB 09          jmp   main+43h
8B 4D FC       mov   ecx, dword ptr [ebp-4]
83 C1 01       add   ecx, 1
89 4D FC       mov   dword ptr [ebp-4], ecx
8B 55 FC       mov   edx, dword ptr [ebp-4]
3B 55 F0       cmp   edx, dword ptr [ebp-10h]
7D 22          jge   main+6Dh
8B 45 F4       mov   eax, dword ptr [ebp-0Ch]
03 45 FC       add   eax, dword ptr [ebp-4]
8A 08          mov   cl, byte ptr [eax]
88 4D F8       mov   byte ptr [ebp-8], cl
0F BE 55 F8    movsx edx, byte ptr [ebp-8]
```

# Inserting the Decoder Stub

- Copy out the hex bytes (remove newlines):

```
C745F400004000C745F0EFBE00008B45F405ADDE00008945F4C745FC
00000000EB098B4DFC83C101894DFC8B55FC3B55F07D228B45F40345
FC8A08884DF80FBE55F883F20F8855F88B45F40345FC8A4DF88808EB
CDFF65F4
```

- Open the target application in WinHex
- Highlight offset 0x4E00, hit Ctrl+B (write clipboard) and select ASCII Hex
- The final step is to modify the placeholders with real values. Remember, little endian

```
Offset      0  1  2  3
00004E10   .. .. AD DE    (DEAD)
00004E10   .. .. 48 10    (1048)

Offset      0  1  2  3  4  5  6  7  8  9  A  B
00004E00   .. .. .. .. .. .. .. .. .. .. EF BE  (BEEF)
00004E00   .. .. .. .. .. .. .. .. .. .. 86 2D  (2D86)
```

# Watch it in Action

- Load the modified file in OllyDbg
- Look at the original EP and verify that it's not recognized as executable code
- Set a breakpoint at the end of the decoder stub

```
00404E53  EB CD    JMP SHORT final.00404E22
00404E55  FF65 F4  JMP [EBP-C]              ; final.00401048
```

- Hit F9 to run through the decoder
- Look at the original EP again
  - Right click "Analysis→Analyze code" (Ctrl-A)

# Exercise

- Create your own packer
- Follow these high-level steps:
  - Copy original_hello.exe to pe_modified_hello.exe
  - Modify pe_modified_hello.exe with LordPE
  - Run do_opcode_crypt.exe to create crypted.exe
  - Insert the decoder stub into crypted.exe
  - Verify crypted.exe works
  - Verify crypted.exe is indeed "crypted"
- Some shortcuts were made for you

# Detecting Packed Executables

- Entry Point (EP) lies outside of .TEXT section
- .TEXT section is writable
- Sections with 0 size
- Limited imports
- No strings
- Entropy (compressibility of bytes)
  - Ghetto check: compression

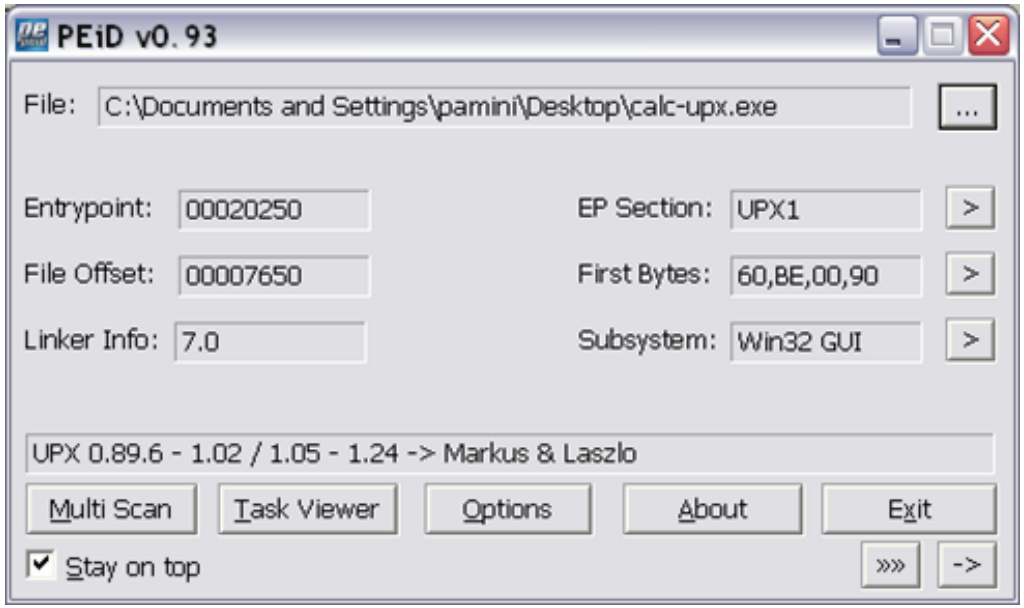
```
$ cat calc.exe | gzip -v > /dev/null
 55.8%

$ cat calc-upx.exe | gzip -v > /dev/null
 24.9%

$ cat calc-aspack.exe | gzip -v > /dev/null
 25.2%
```

# The Easy Way: PEiD

- Has a multitude of signatures (somewhat outdated)
- You can add your own (many publicly exist)
- Entropy detection
- Generic detection
- Generic unpacking with intelligent OEP detection



# Methodologies

- There are three basic methodologies to unpacking
- Post execution analysis
  - An easy and generic approach
  - Most dangerous
- Controlled run-time analysis
  - More difficult to do
  - More control / less dangerous
  - PEiD attempts to automate this, but we don't trust anything automated
- Static analysis / basic emulation
  - Most difficult to do
  - Safe

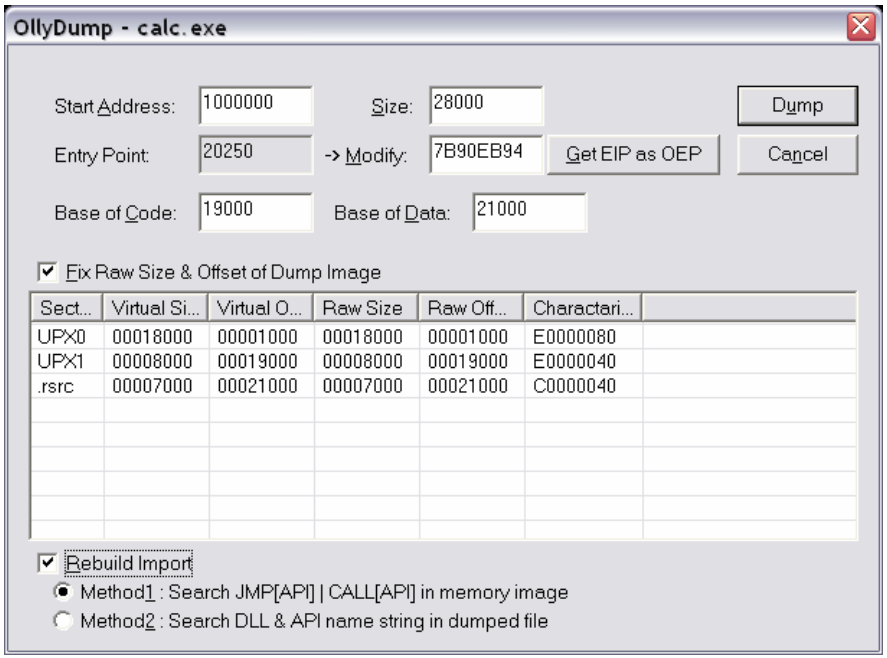
# Post Execution Analysis

- Launch the packed binary in a controlled environment
- The decoder stub will run leaving an unpacked copy of the binary in memory
- Dump the running process to disk
  - OllyDbg +OllyDump
  - ProcDump
  - Etc...
- Fix the Import Address Table (IAT) and analyze (more on this later)

# Controlled Run-Time Analysis

- Load the packed binary in a debugger
- Single step through the decoder stub until a transfer is made to the Original Entry Point (OEP)

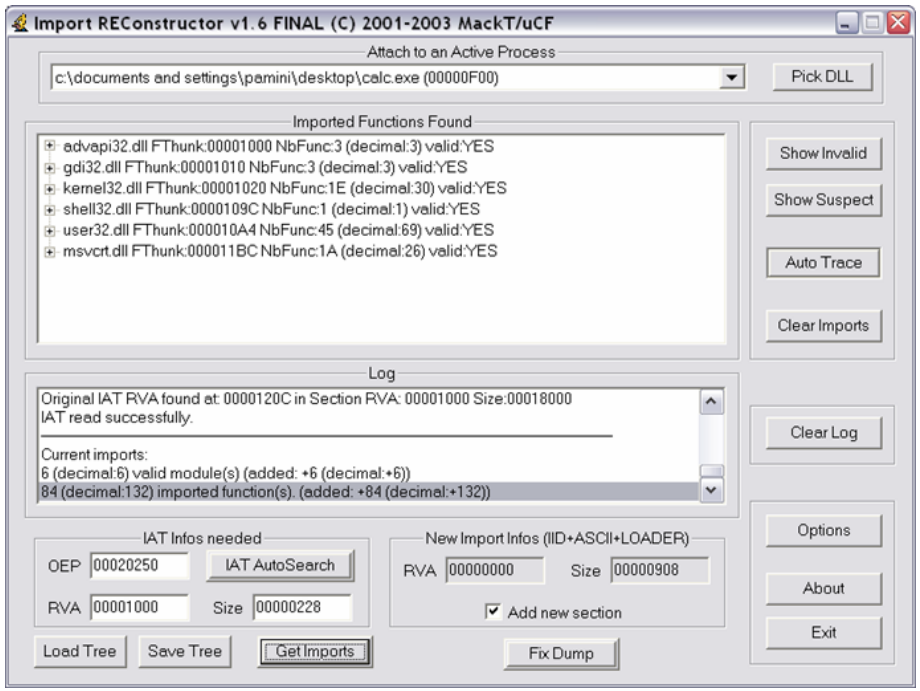
- Dump the running process to disk
  - OllyDbg +OllyDump
  - Etc...
- Fix the Import Address Table (IAT) and analyze (more on this later)



# IAT Reconstruction

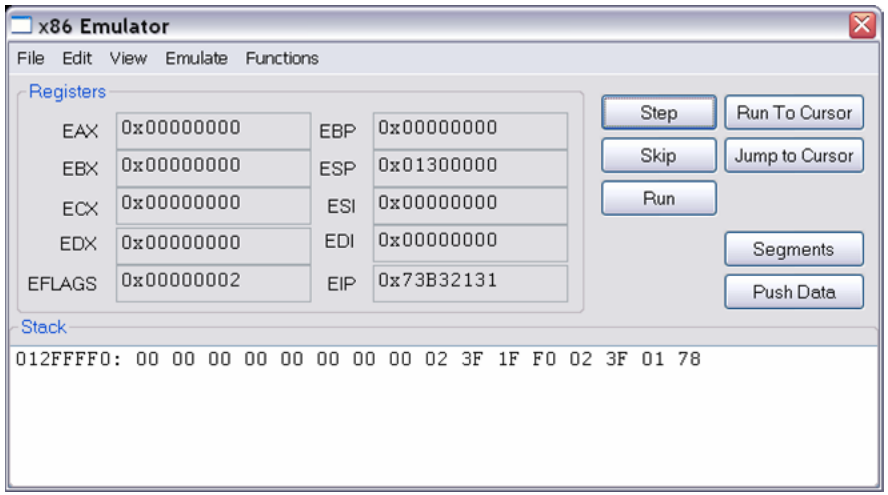
- Why do we want to rebuild the IAT?
  - Analyzing a malicious binary without symbols is burdensome
- There are a number of approaches to IAT reconstruction

- We will cover three approaches
  - OllyDump
  - ImportRec
  - IDCDumpFix
- Depending on the target, one approach may work better than another



# Static Analysis / Basic Emulation

- If possible, such as with UPX, manually unpack the binary with the relevant unpacker
- Process the binary with an emulator such as Chris Eagle's x86-emu plug-in
- Single step through the decoder stub until the IDA database is completely decoded



# Exercise

- Examine the PE structure of calc*.exe
- Unpack calc-aspack.exe using post execution analysis
- Unpack calc-upx.exe using controlled run-time analysis
- Unpack the custom packed file from the previous exercise using x86-emu
- For the daring, unpack calc-aspack.exe using controlled run-time analysis. Watch for the anti-disasm trickery at EP.
- Rebuild IATs using various techniques and compare.
- See the following files for hints ONLY if you get stuck
  - OpenRCE ASPack Notes.txt
  - OpenRCE UPX Notes.txt

# Excercise

- Automate the process of dumping and reconstruction with pydbg and pefile
- Attach to a running process
- Preprocess common system DLLs and harvest the default addresses of the exported symbols
- Scan and dump the address space while also looking for possible pointers to the DLL's exported symbols' addresses
- Generate IDAPython/IDC output that can be loaded directly into IDA to recover the symbols in the loaded dump

# Original Entry Point (OEP) Discovery Techniques

- Graph analysis
- PUSH/POP trick
- Common APIs
- Markov chains on instruction patterns
- Taint analysis

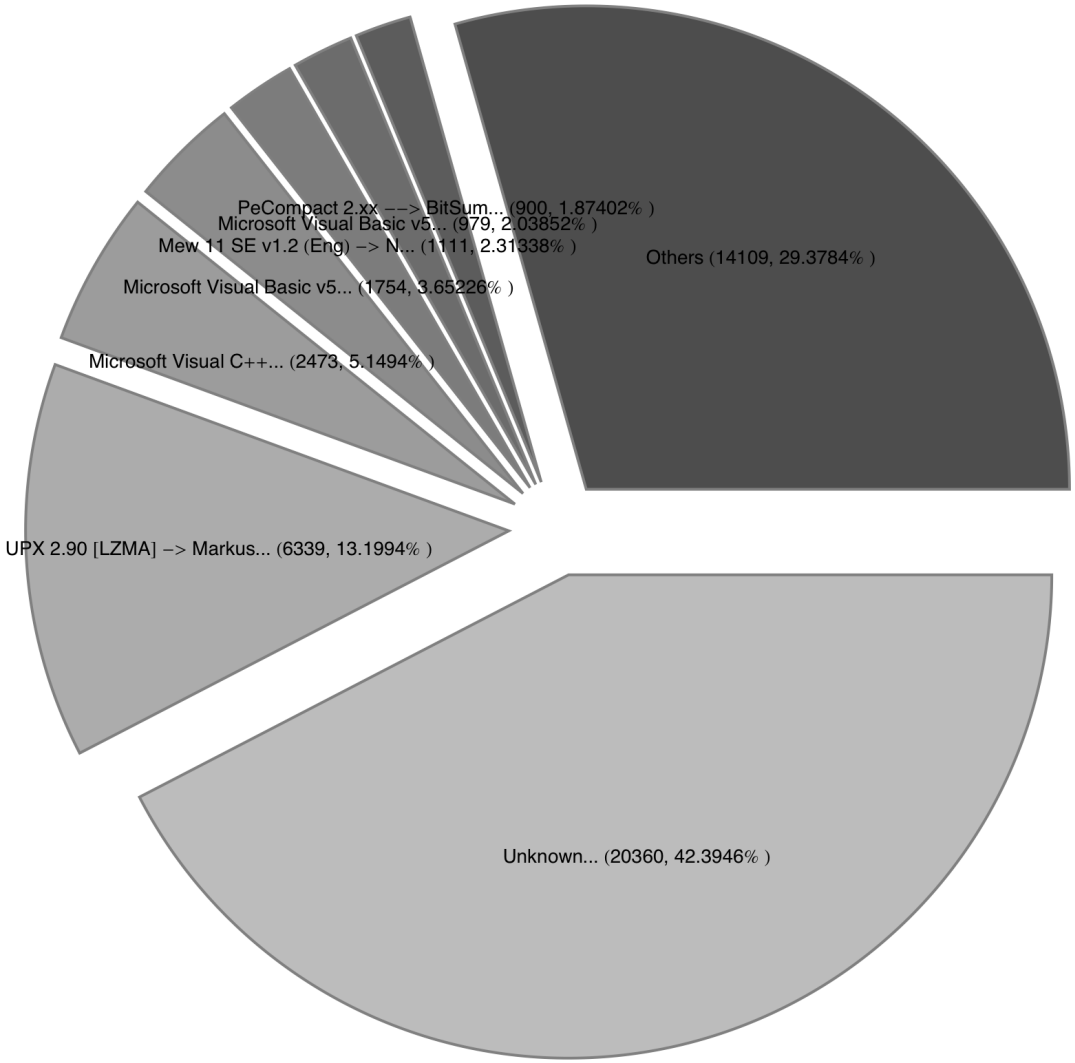
# The Corpus

- Had just under 50.000 malware samples lying around
- Ran them through a beta of *pefile* which employs *PEiD*'s packer signature database
- In around 42% of the files no packer could be found
- In the remaining ( 58%) of the files over 200 packers and compilers were identified

# Most Frequently Used

- The most frequently occurring packer among the samples, *UPX*, would probably not surprise anyone that has been looking at malware for a while.
- It appeared in around 13% of the samples.
- *Mew*, *PeCompact*, *ASPack*, *FSG*, *y0da's Crypter*, *Armadillo* and other versions of UPX took the next places.

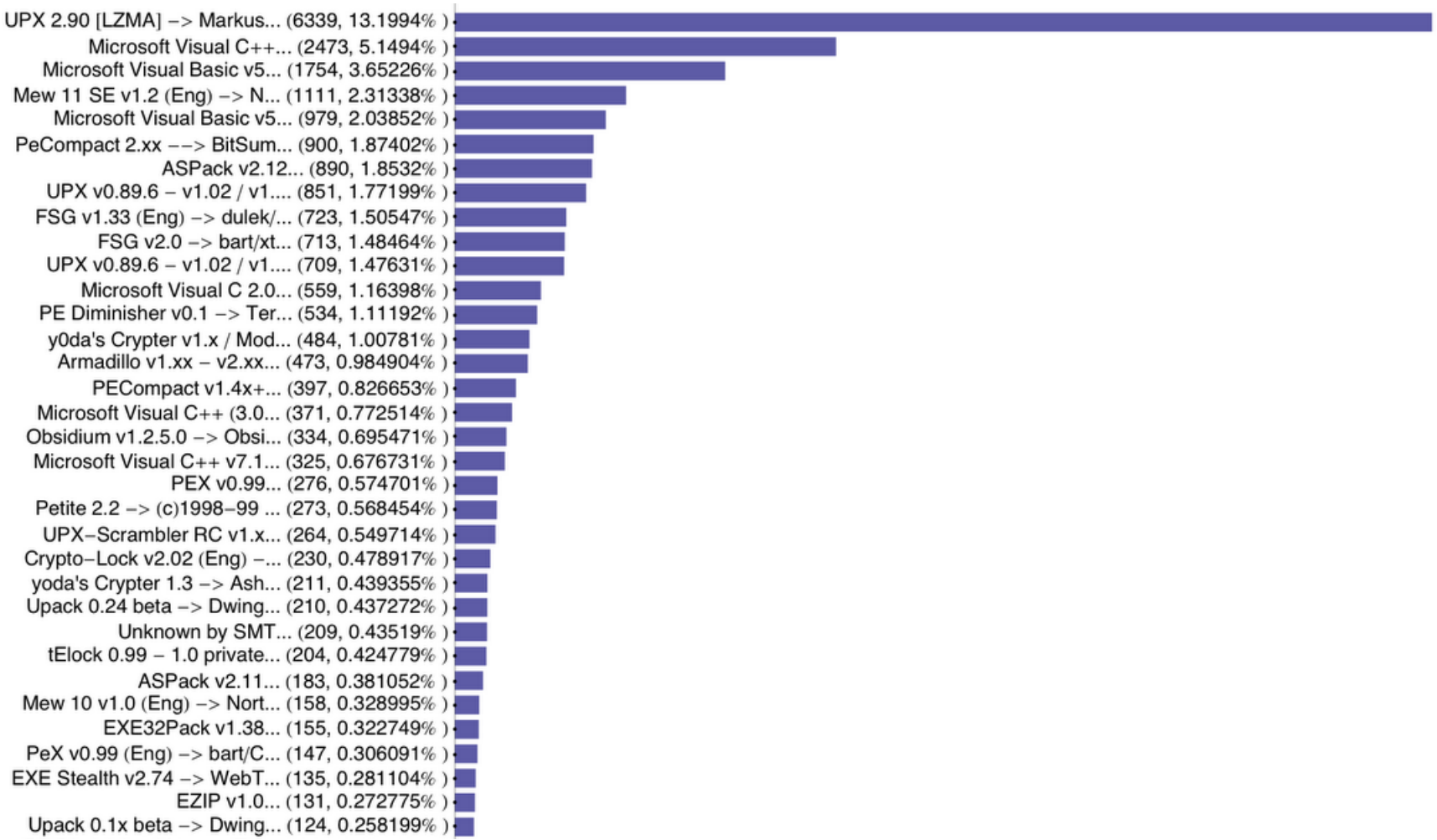
# Most Common Packers



# The Long Tail

- Reverse engineers and analysts have to confront a large variety of packers and their modified versions
- PEiD's largest databases have over 2000 entries
  - http://www.secretashell.com/BobSoft/
- The most obscure packers tend to appear with a relatively low frequency
- Yet, it's usually those more exotic packers, the ones incorporating fancier techniques
- In the future we will see more technologies aimed at generically unpacking samples (like the old SCU, or PEiD's and Procdump's methods)

# The Long Tail



# The Concept

- Thought it would be cool to be able to see how unpacking happens for different packers
- It's as "easy" as tracing all memory writes and EIP location as the unpacker goes on its way
- With help of a special tool that can be achieved without worrying about packer specifics

# Plotting the Data

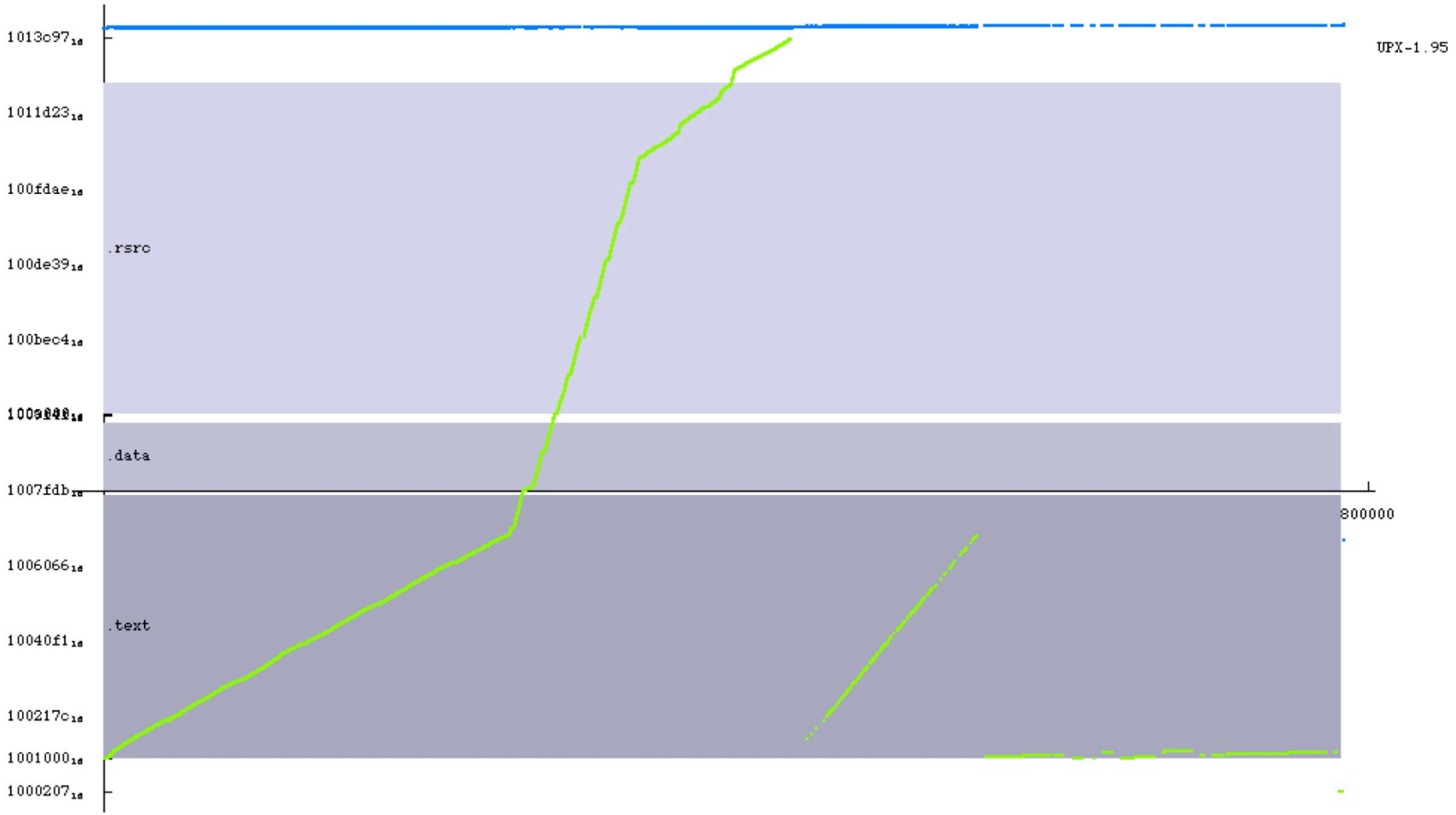
- A subset of all the data extracted was plotted with the help of Mathematica
- EIP = blue
- Memory writes = green
- The address space being displayed is the one of the unpacked process, therefore it'll be the same for all packers (all traces are of *Notepad.exe* being unpacked)
- The horizontal axis represents time (instructions executed)
- The vertical axis represents the address being executed or written to (it has been limited roughly to the address range of the unpacked image)

# UPX-1.95

*UPX-1.95* is one of the most frequently used packers, it does have a good compression ratio but has no features to attempt to prevent dumping or to obfuscate the unpacking process

- The unpacking code runs in a very constrained area
- Just decompresses all the sections at once, the slope variations come from different data being faster to decompress than other
- In the final pass it just fixes the *IAT* to point to the *DLL* functions

# UPX-1.95 Visualization

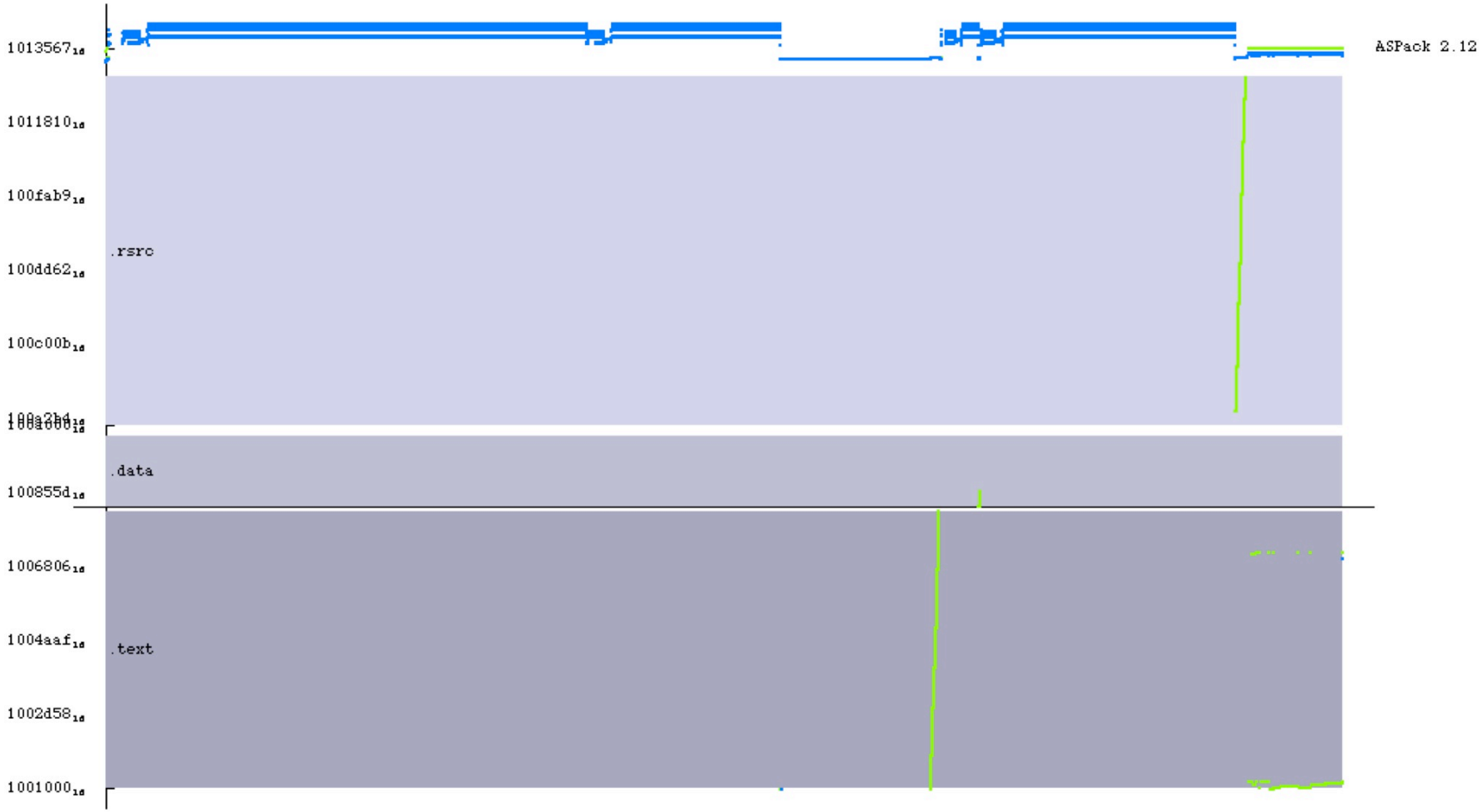


# AsPack 2.12

*AsPack* compresses all sections and goes through a lengthy unpacking algorithm until finally writing the unpacked data to the target sections

- Does not compress padding or empty section areas
- The actual writing of the data happens in rather tight loops

# AsPack 2.12 Visualization

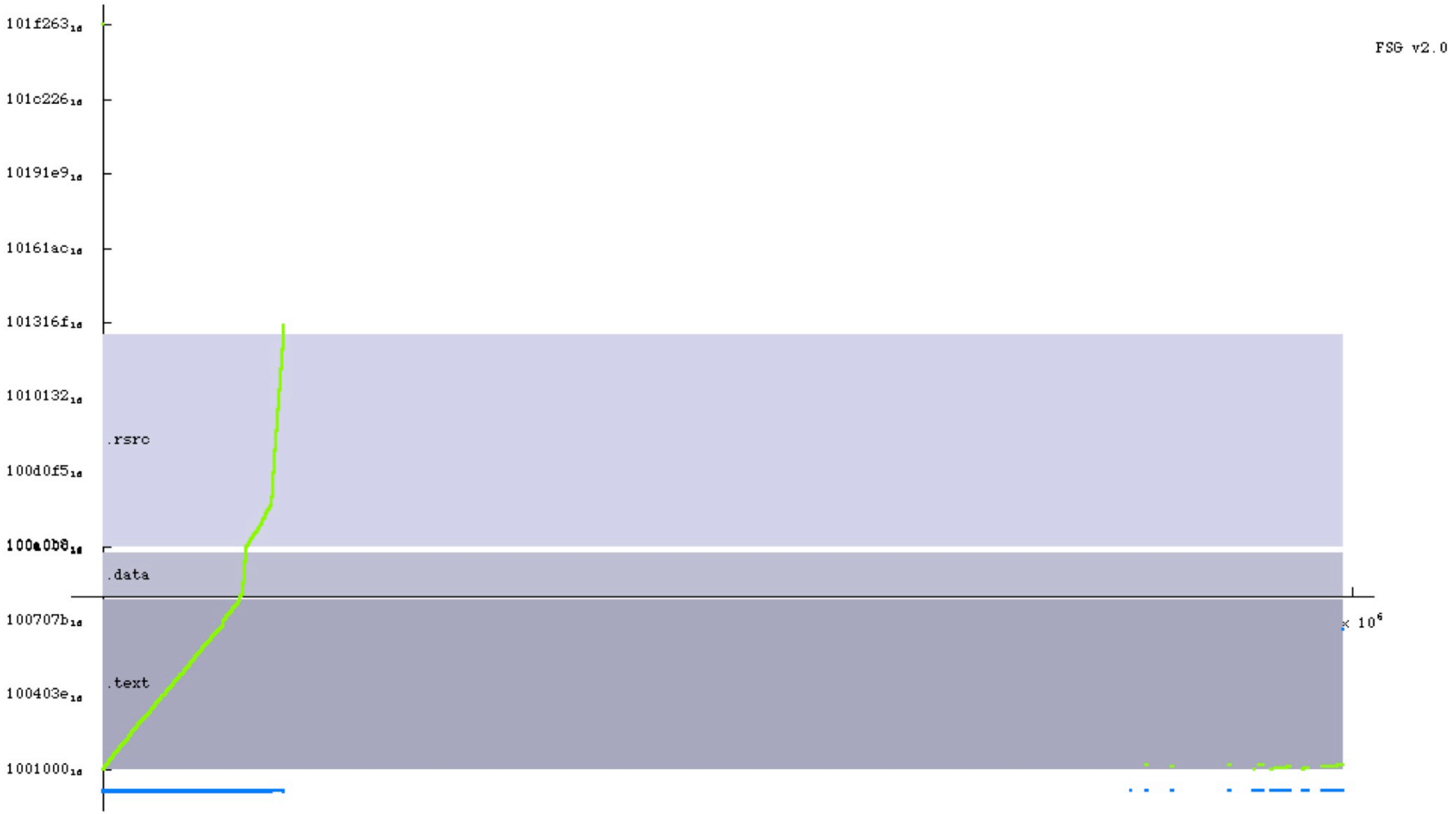


# FSG v2.0

*FSG* is extremely simple, it simply goes though a decompression loop writing the data straight to the final destination

- Differences on the slope of the memory writes data indicate different compression ratios. Redundant data is faster to decompress
- The *EIP* (blue) is lost for a while, the reason is that it's executing outside the visible range, within *DLL*s

# FSG v2.0 Visualization

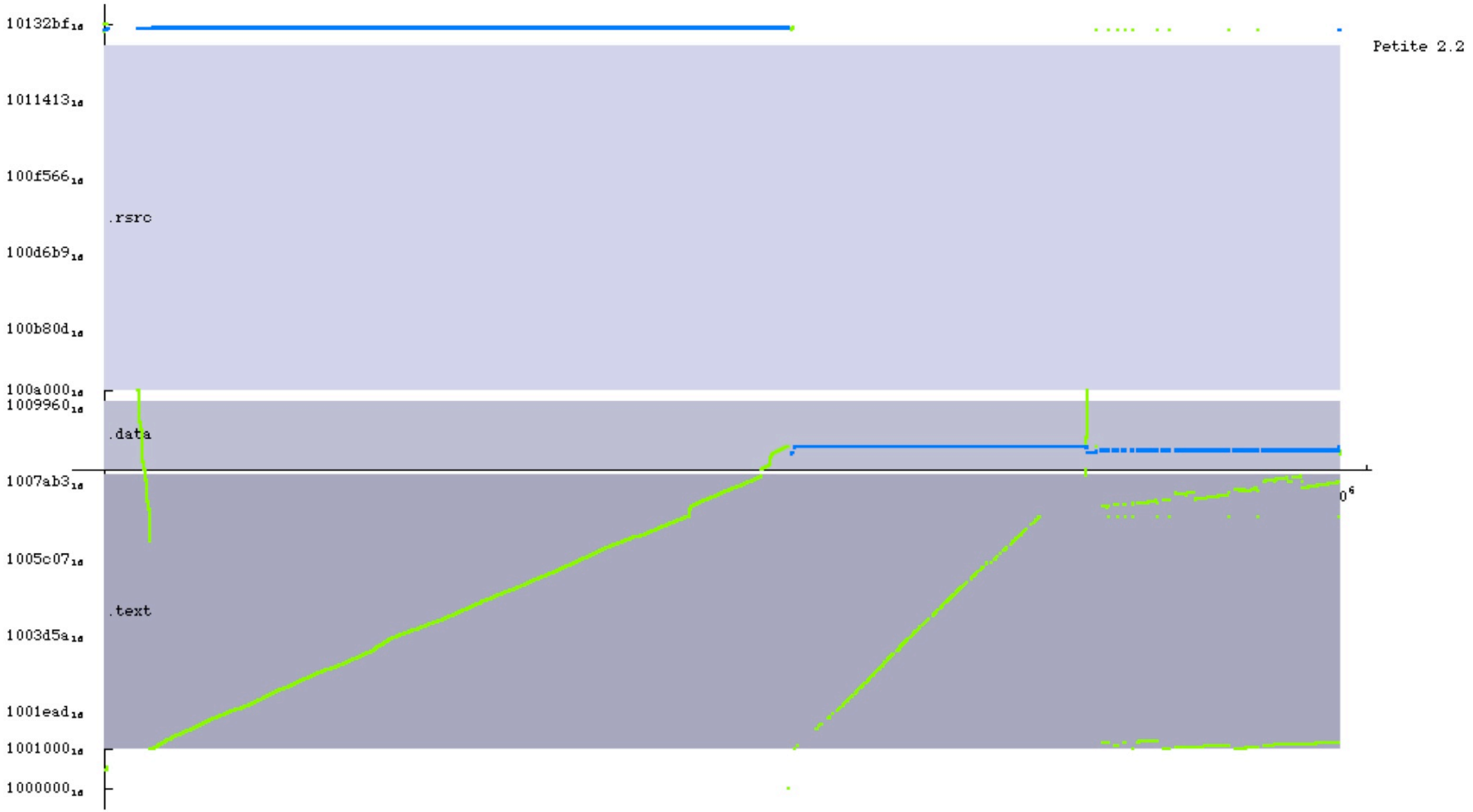


# Petite 2.2

*Petite 2.2* is an interesting case, here we can see an instance of a multi-stage packer

- Data written to the *.data* section is later run
- It does two passes over the code section
- There is, as with other packers, a final pass fixing the *IAT*

# Petite 2.2 Visualization

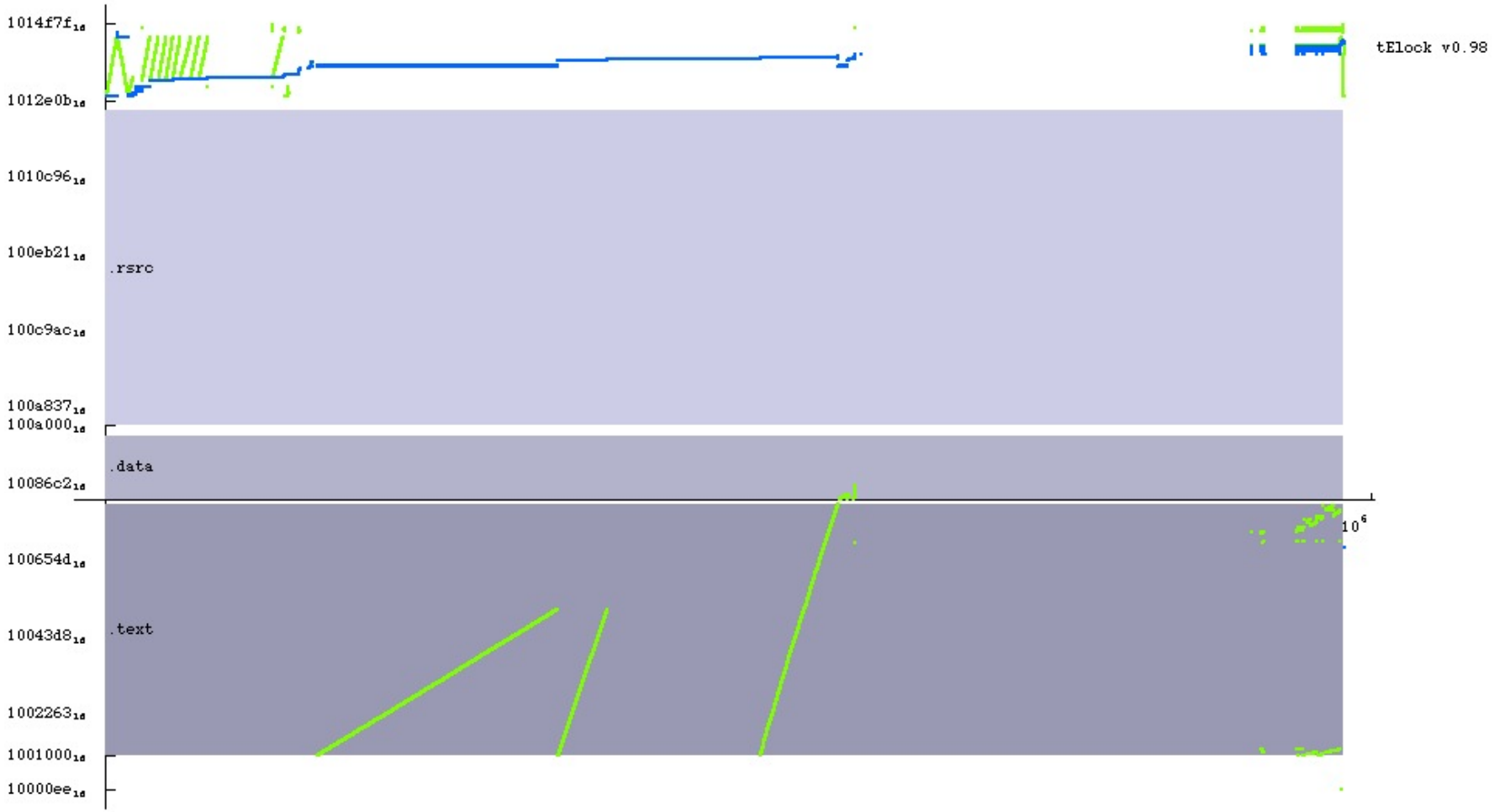


# tElock V0.98

*tElock V0.98* is a rather complex case. Again it's possible to see a multi-stage packer

- The code section is written in three passes
- It does not compress the *.rsrc* section but just the valid data in *.data*
- As with the *FSG* packer, the EIP escapes the shown address range to go execute *DLL* code
- It does a quick final *IAT* fix-up

# tElock V0.98 Visualization

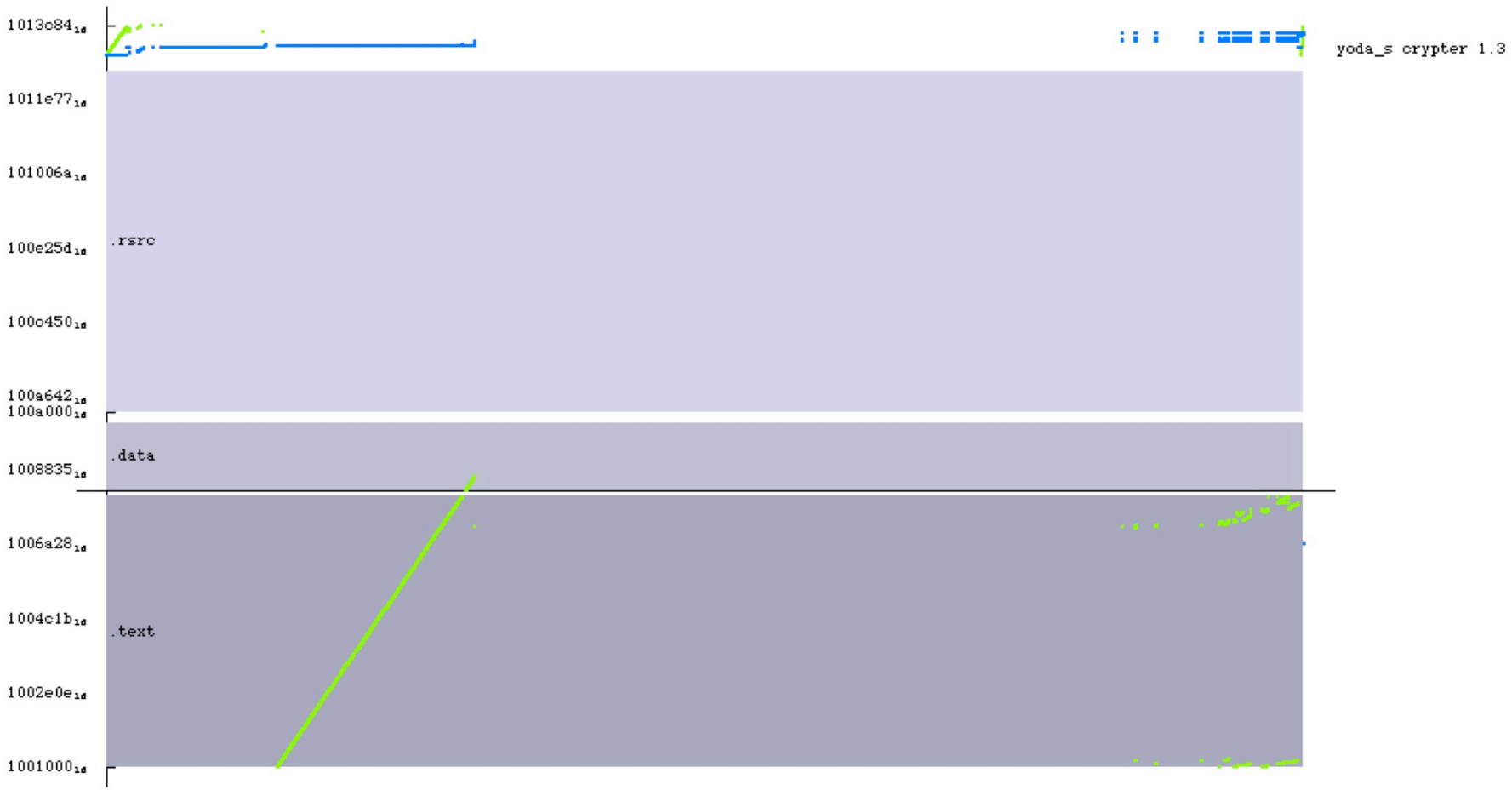


# Yoda's Crypter v1.3

*Yoda's Crypter v1.3* is fast and straightforward

- Only compresses the *.text* and *.data* sections
- Runs within a small area of code after the last section of the original binary
- It's also multi-stage

# Yoda's Crypter v1.3 Visualization

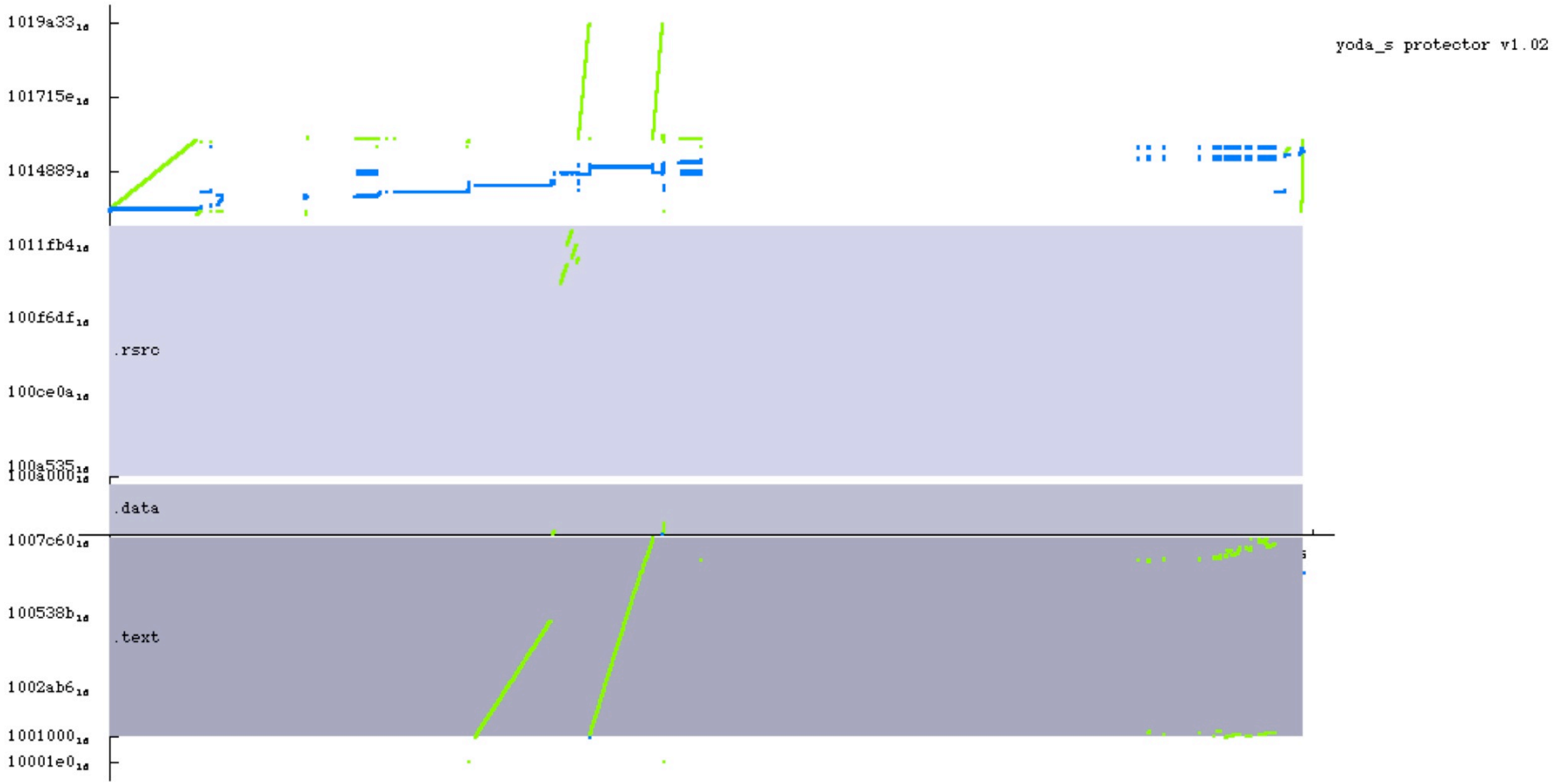


# Yoda's Protector v1.02

*Yoda's Protector v1.02* is a fairly complex one

- We can see a multi-stage packer in action once again
- Only packing the defined data in the *.text* and *.data* sections
- After some time executing in the *DLL*'s range it comes back to the unpacker code to fix the *IAT* and, like all other packers, passes control to the original application

# Yoda's Protector v1.02 Visualization



# Outline

10 Executable (Un)Packing

11 Anti Reverse Engineering
- Anti-Debugging
- Anti-Disassembling
- Anti-PE Analysis
- Anti-VM

12 Binary Diffing and Matching

13 Advanced Malware Techniques

# Debugger Detection 1 of 4

- Kernel32.IsDebuggerPresent()
  - Most easily discovered and defeatable
- INT 3 (0xCC) scans and CRC checks (triggered by means of exceptions)
  - Easy to bypass with hardware breakpoints
- Timers
  - Very basic but powerful
  - Example:

```
start = GetTickCount();
do_some_stuff();
if (GetTickCount()  start > threshold)
debugger_detected();
```

- RDTSC  Read Time-Stamp Counter (EDX:EAX)

# Debugger Detection 2 of 4

- CheckRemoteDebuggerPresent()
  - Queries the debugger port by calling NtQueryInformationProcess()
  - Harder to defeat but doable through hooks
- Detecting hardware breakpoints
  - Install a SEH, trigger an exception and check the DR* registers in the process' context structure
  - Can also set magic values and verify they are kept
  - There are other ways of retrieving the process' context structure
- INT 2Dh
  - Without a debugger running we can trigger an exception and catch it with a SEH
  - With a debugger present, we won't normally get control after the exception
  - Additionally execution will continue a byte ahead from the address immediately after the INT 2Dh

# Debugger Detection 3 of 4

- PEB.BeingDebugged
  - We can get the value directly and compare it to what's returned by IsDebuggerPresent(), if it's different a debugger might be trying to trick us
- NtGlobalFlag
  - PEB -> NtGlobalFlag. If 70h == FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK and FLG_HEAP_VALIDATE_PARAMETERS, implies debugger is active
- Query the debbuger port
  - NtQueryInformationProcess(-1, 7,&dword_var, 4, 0)
  - Will return the debugger port is one is attached
  - Tricky to work around, we can either hook ZwQueryInformationProcess or intercept the subsequent syscall with a driver

# Debugger Detection 4 of 4

- PEB.ProcessHeap->ForceFlags
  - If not equal to zero implies a debugger is active, but only if the process was started by the debugger
- Memory tags
  - If a process is started for debugging, Windows will tag freed and reserved memory by filling it with 0xFEEEFEEE
  - PEB.Ldr points to _PEB_LDR_DATA, which is a good candidate where to scan for the pattern
- Spotting Single-Stepping
  - Set SEH and set the Trap Flag:

```
PUSHFD
XOR DWORD PTR[ESP],154h
POPFD
```

  - or execute INT1 (0xF1) which will generate a Single Step exception
  - If the SEH is called, no debugger is attached

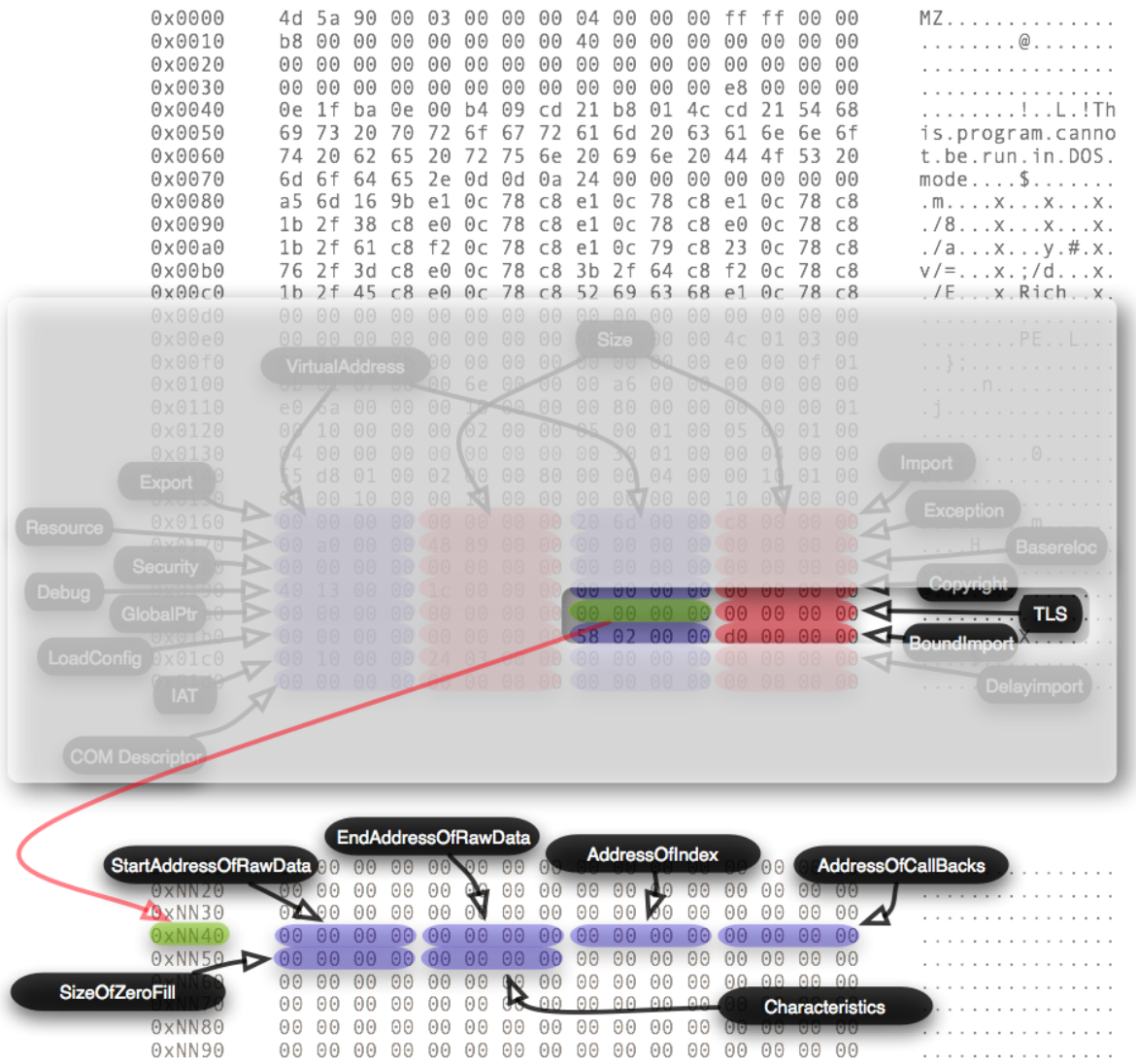
# Debugger Pre-Interaction Execution

- When loading a target malicious binary do not assume that malicious code can not execute prior to the initial break
- DLLMain() execution occurs pre-interaction (unless you hook DLL load events)
  - http://www.security-assessment.com/Whitepapers/PreDebug.pdf
- This technique has been known by the underground for some time
- PE Thread Local Storage (TLS) initialization startup code

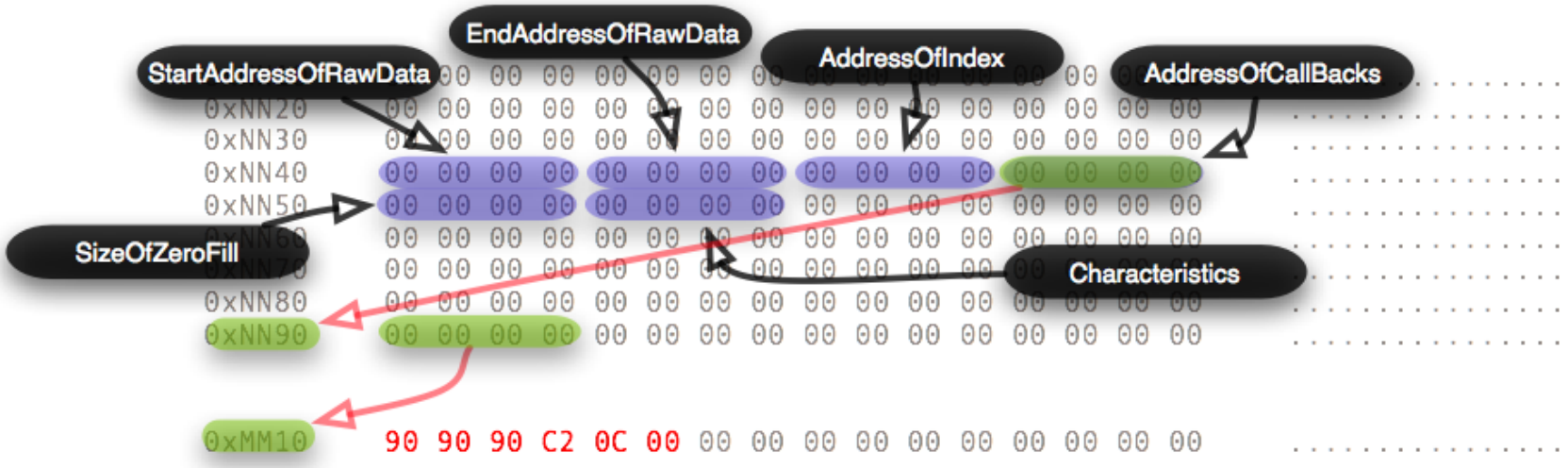
# A Look Into the TLS Trick

- *TLS* stands for *Thread Local Storage* and it's meant to be used to allocate storage for thread-specific data
- The *TLS* structure, *IMAGE_TLS_DIRECTORY*, pointed to by the *TLS* directory entry has a small number of fields
- The one of special interest is *AddressOfCallBacks*, pointing to a list of callbacks
- TLS callbacks have the following form:
  - typedef void (MODENTRY *PIMAGE_TLS_CALLBACK) ( PTR DllHandle, UINT32 Reason, PTR Reserved );

# Finding the TLS Structure



# The TLS Structure



# TLS Trick Conclusions

- Inserting code as a *TLS* callback allows to run our code before the main entry point of the program is reached
- This could be used to decrypt or otherwise modify the image of the file in a way that confuses someone trying to debug it and is unaware of this functionality
- IDA is clever and knows this, it will find the TLS entry point and offer it as the starting point in the disassembly
- For more on this, check out the blog post at
  - http://blog.dkbza.org/2007/03/pe-trick-thread-local-storage.html

# OllyDbg Vulnerability

- Format string vulnerability in OutputDebugString()
  - http://www.securiteam.com/windowsntfocus/5ZP0N00DFE.html
  - To reproduce, attach to an instantiation of Python:

```
import win32api
win32api.OutputDebugString("%s"    * 50)       # crash
win32api.OutputDebugString("%8.8x" * 15)       # stack data
```

- Format string vulnerability in INT 3 processing
  - Exploitable when module name contains format string token
  - http://www.securiteam.com/windowsntfocus/5WP0B1FFPA.html
- Latest version, v1.10d still vulnerable!
  - The former is easier to exploit than the latter

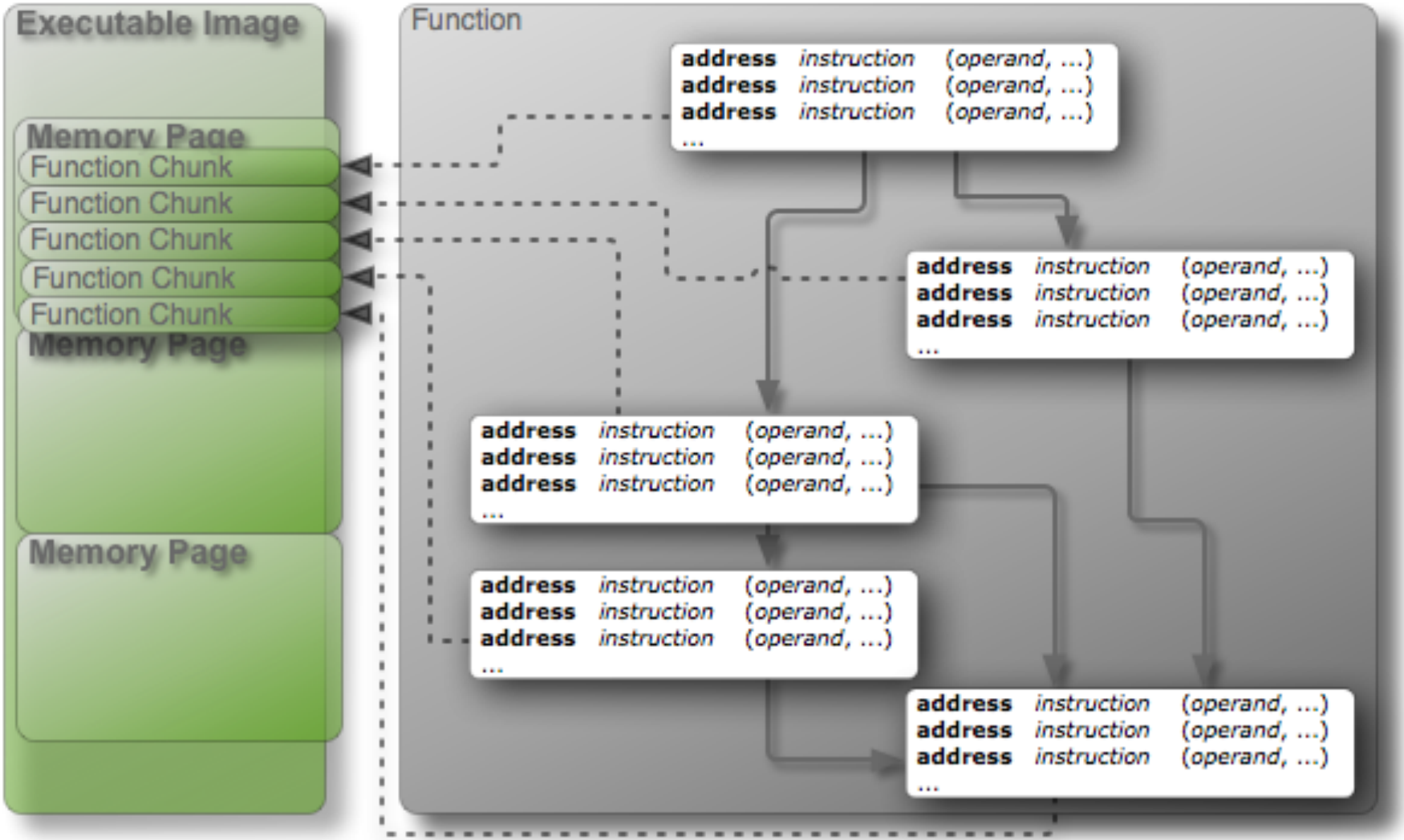
# Disassembler Mucking

- JMP-ing or CALL-ing into an instruction (ASPack)
  - Breaks disassembly
- Executable packing, crypting or otherwise encoding
- PE header modifications
  - As shown before
- Advanced compiler optimizations
- Vulnerabilities ;-)

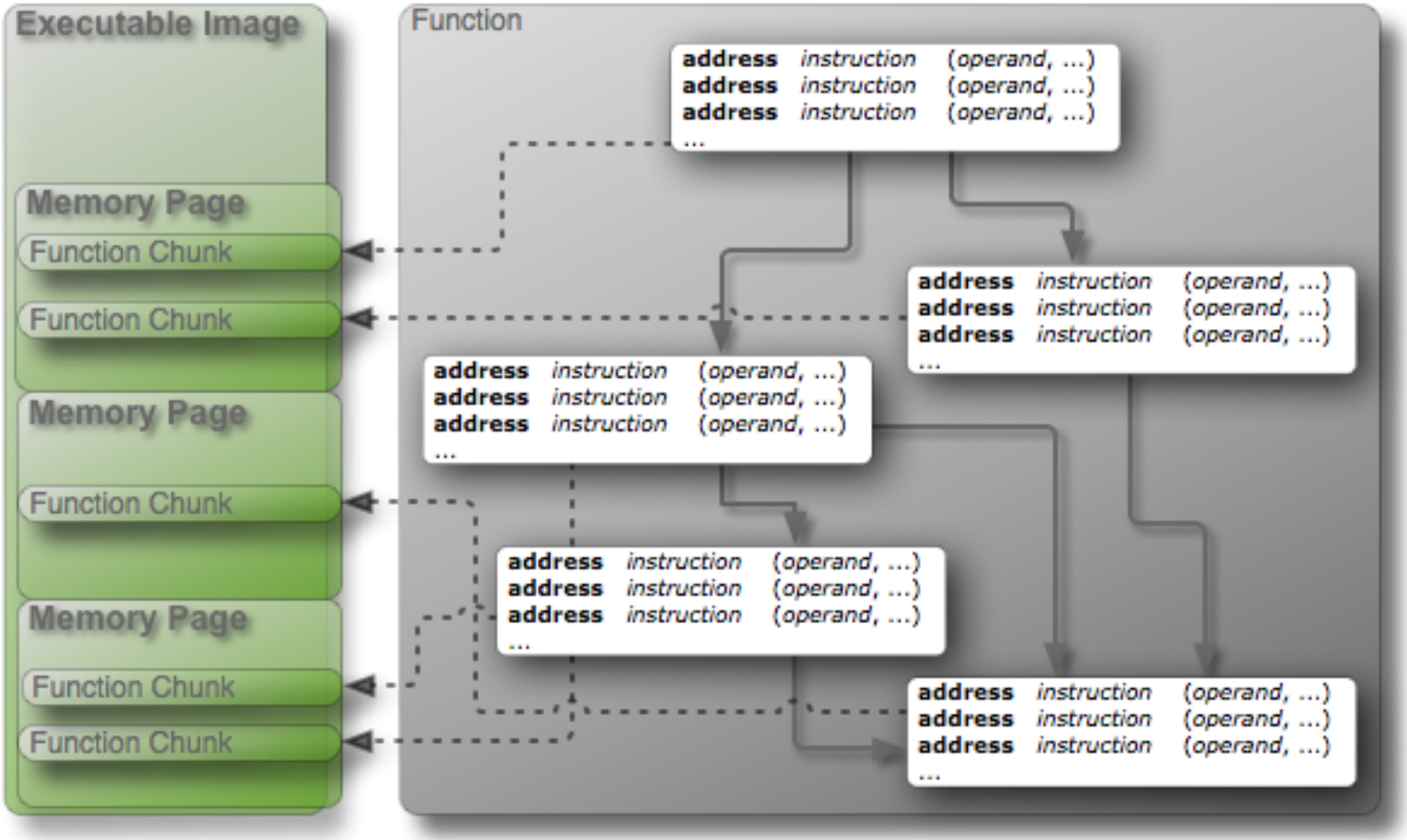
# Advanced Compiler Optimizations

- *PGO* (*Profiling Guided Optimization*) produces multi-chunked functions
- Problems arise when those chunks are shared by different functions (only the tail of the function)
- For more details refer to:
  - http://blog.dkbza.org/2006/12/simply-blocks-basically.html
  - http://blog.dkbza.org/2007/01/binnavis-basic-block-handling.html

# Basic Block Rearranging Illustrated. Unoptimized layout



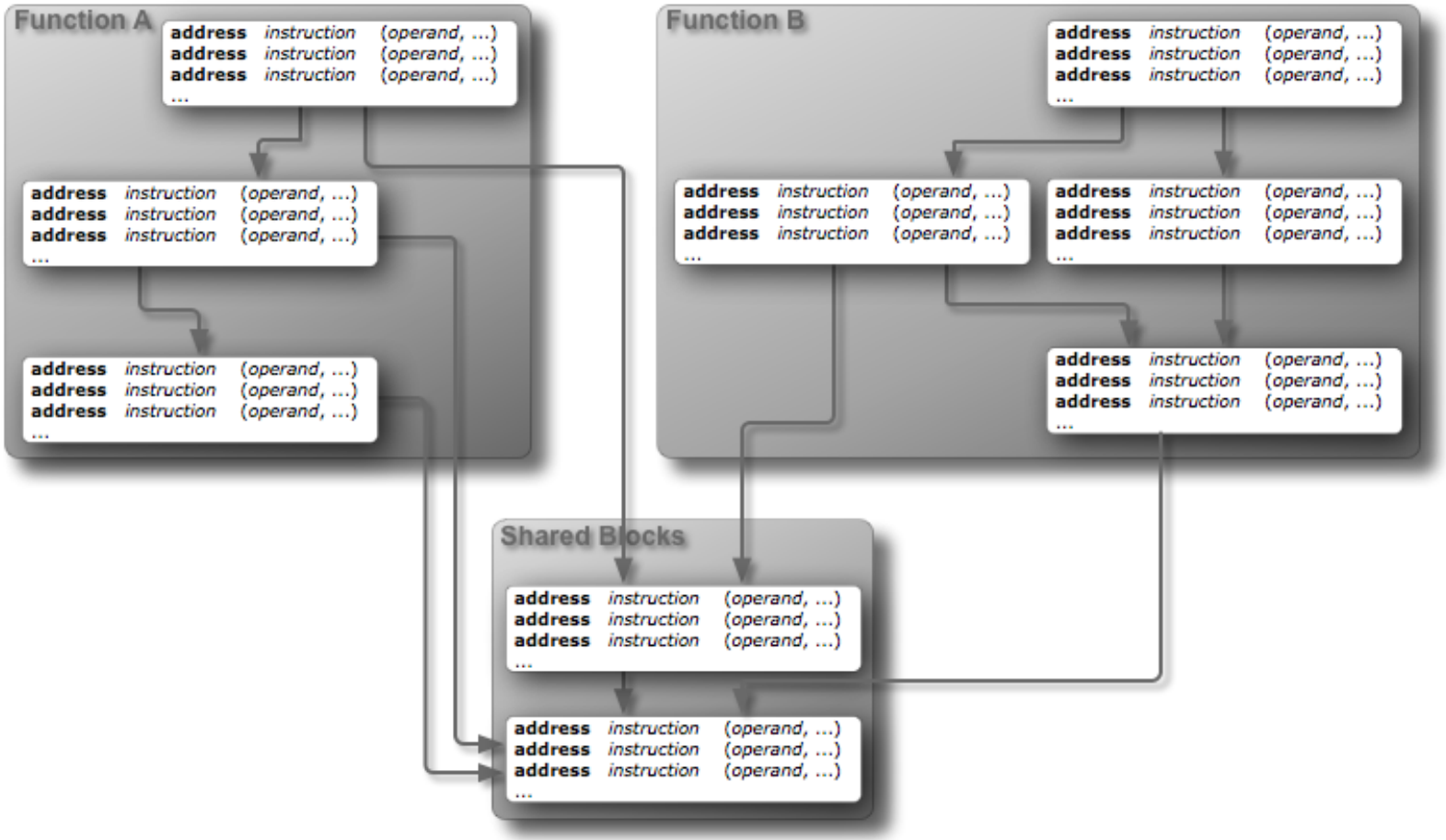
# Basic Block Rearranging Illustrated. Optimized layout



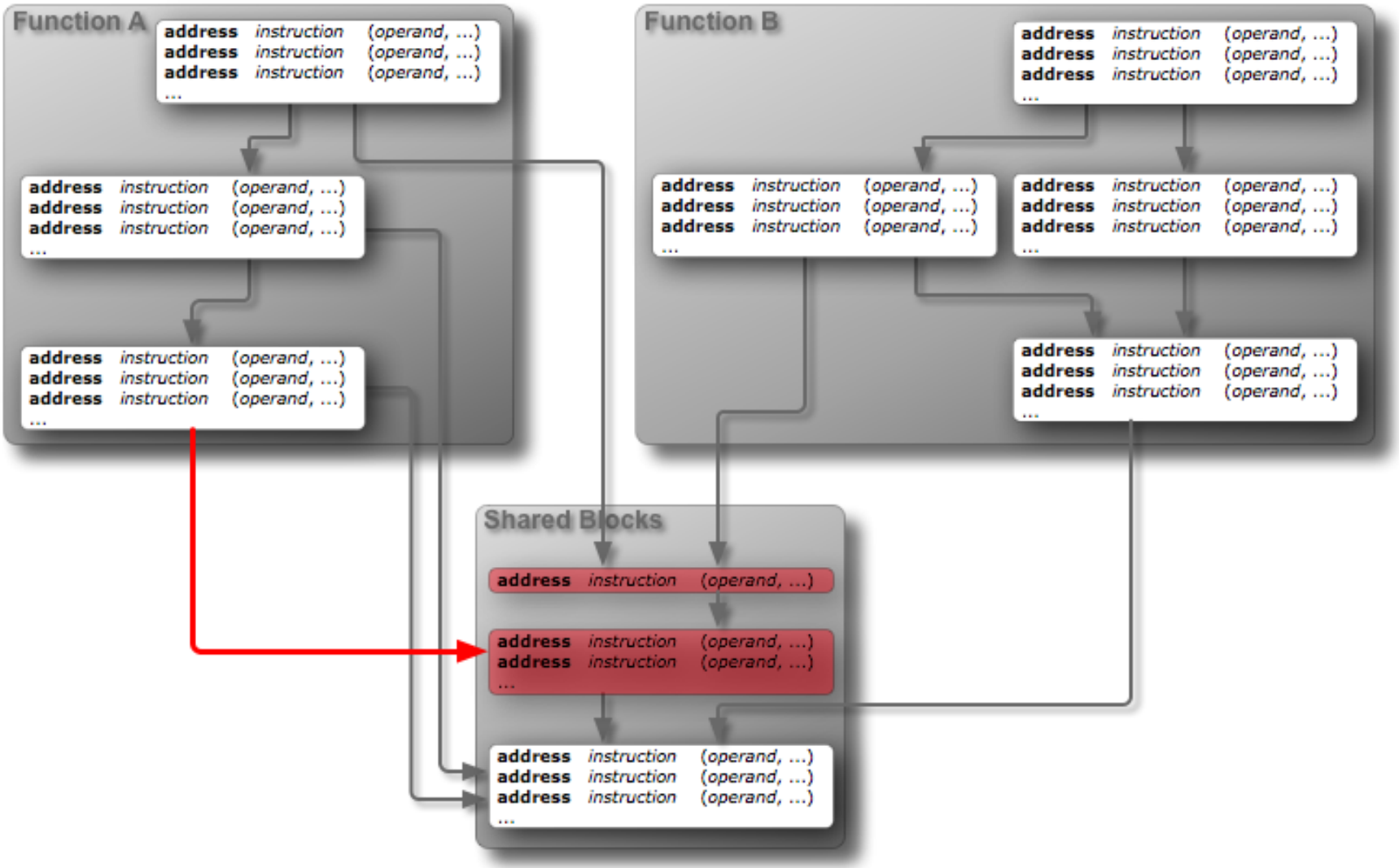
# Basic Block Sharing Problem

- Counter-intuitively, the same code can belong to two, otherwise logically different basic blocks.
- This is a side-effect of the basic-block sharing, caused when one of the functions has a branch targeting the shared code, while the other doesn't

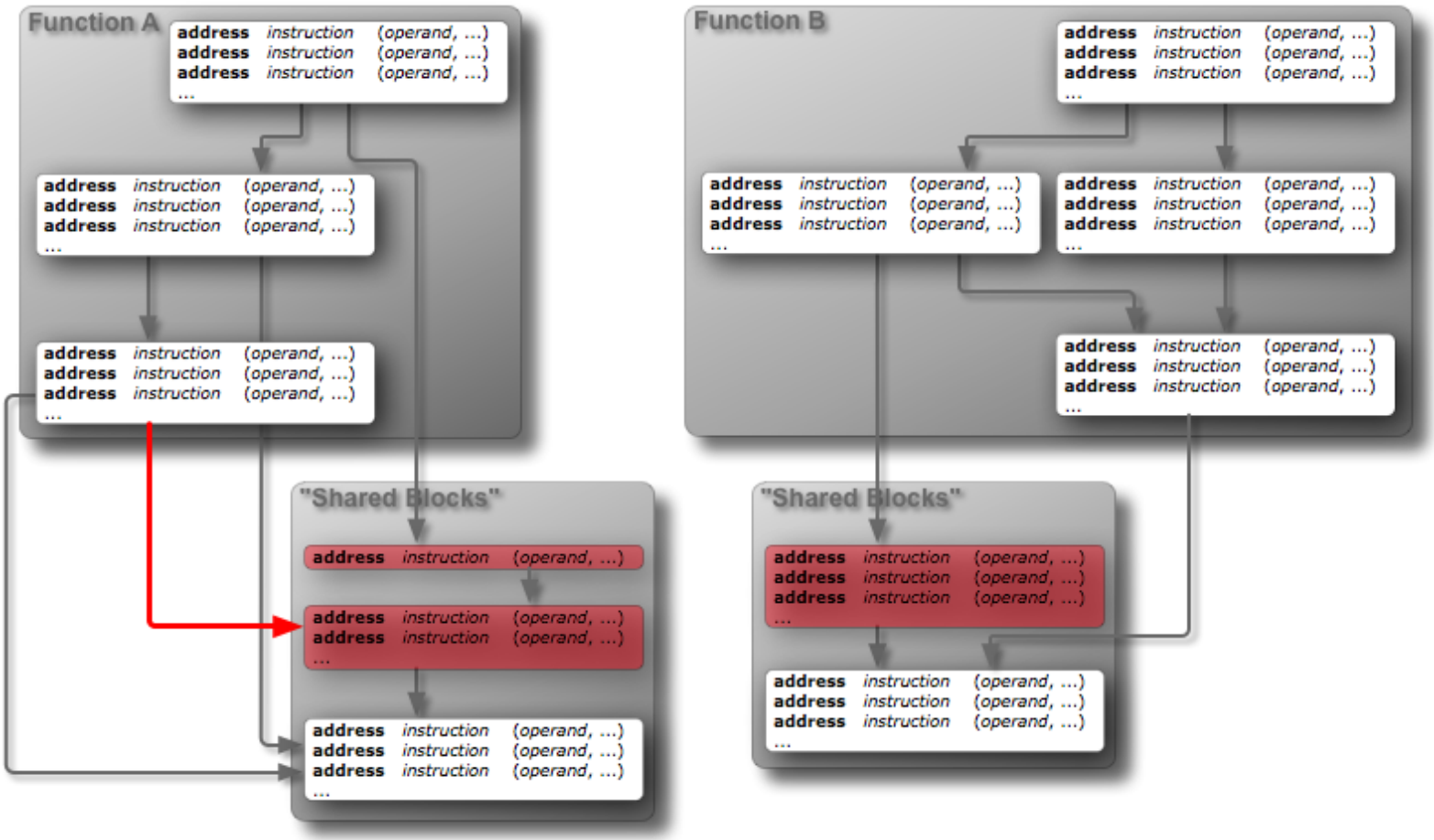
# Basic Block Sharing Illustrated



# Basic Block Sharing Problem Illustrated



# Basic Block Sharing. Alternative View



# More Disassembler Mucking

- Junk instructions
  - Used to waste analysis time
- Frequently encapsulated in blocks bordered with PUSHAD / POPAD
- Consider writing a script to replace PUSHAD.*POPAD with NOP instructions

# IDA Pro Vulnerability

- Buffer Overflow Vulnerability
  - Buffer overflow in PE import directory parsing
  - Easy to create an exploit for, however it breaks the loader
  - There is a way around this, we know because Pedram wrote it ;-)
- Patched in 4.7
  - Affects PEiD as well, which was also patched
- Full advisory
  - http://www.idefense.com/application/poi/display?id=189

# Overview of Tricks

- Invalid/malformed values in the header
- Misaligned sections, few applications correctly load them
- Relocation tricks
- TLS, running before the Entry Point

# Invalid Values

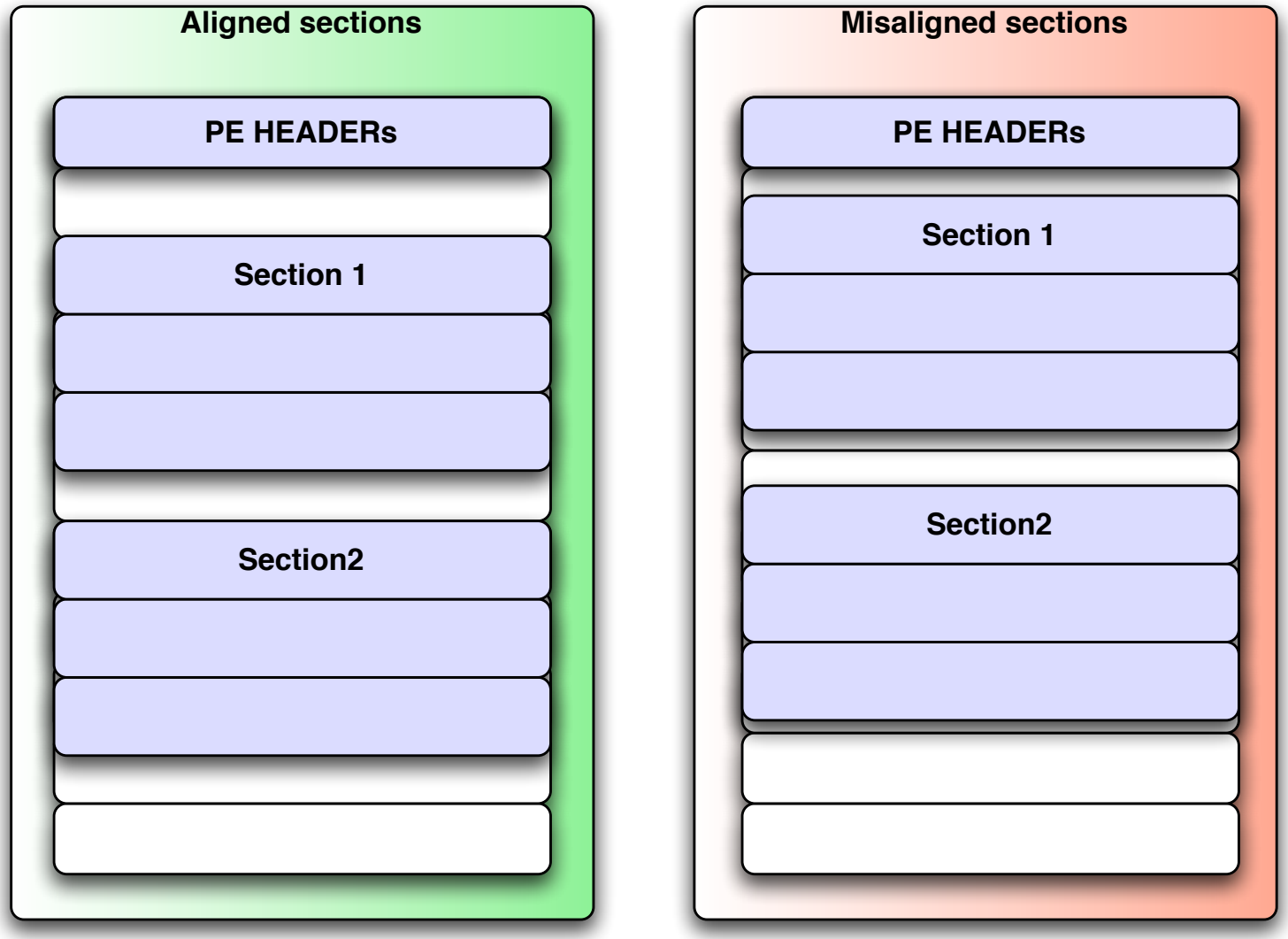
Malware can use invalid values in order to cause trouble to tools and hence, the analyst

- Uncommon ImageBase values
- Invalid data in LoaderFlags and NumberOfRvaAndSizes
- Large SizeOfRawDataValues [Scan of the Month 33]

# Misaligned Sections

- The Optional Header contains members describing:
  - The file aligment (FileAligment)
  - The memory alignment of the sections (SectionAlignment)
- The section´s starting file offset (PointerToRawData) is usually aligned to FileAligment
- However, it´s possible to specify an unaligned offset. Windows will round it down to the largest aligned value smaller than the given offset

# Misaligned Sections

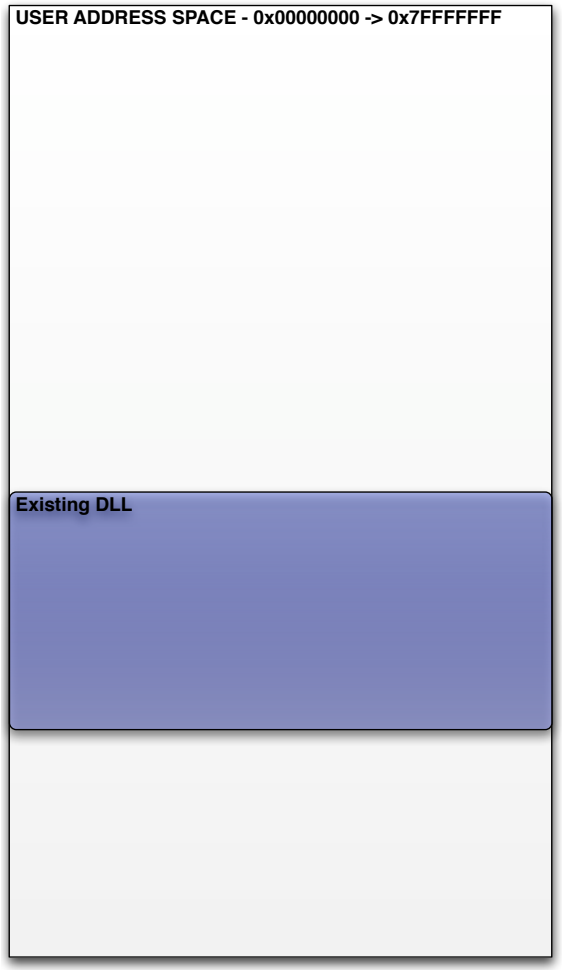


# Relocation Tricks

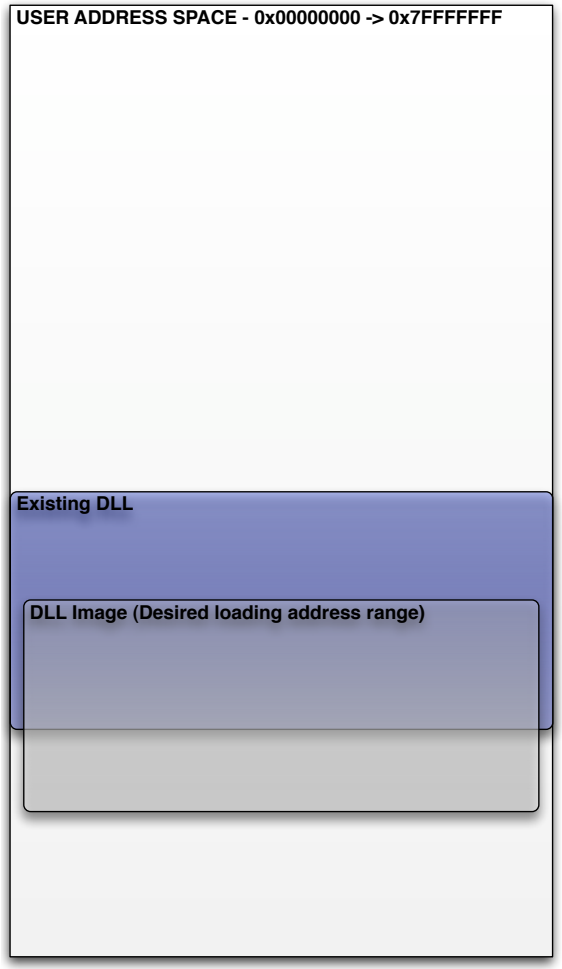
- Relocations are meant to provide for a mechanism to rebase an image
- Relocations are meant to patch references in the code pointing to locations that change if the image is rebased
- It is possible to use this to actually patch any data [Tricky Relocations]
- Abusing this technique allows an attacker to modify the image during load

*skape* has an excellent write-up on the topic in Uninformed 6 [Locreate]

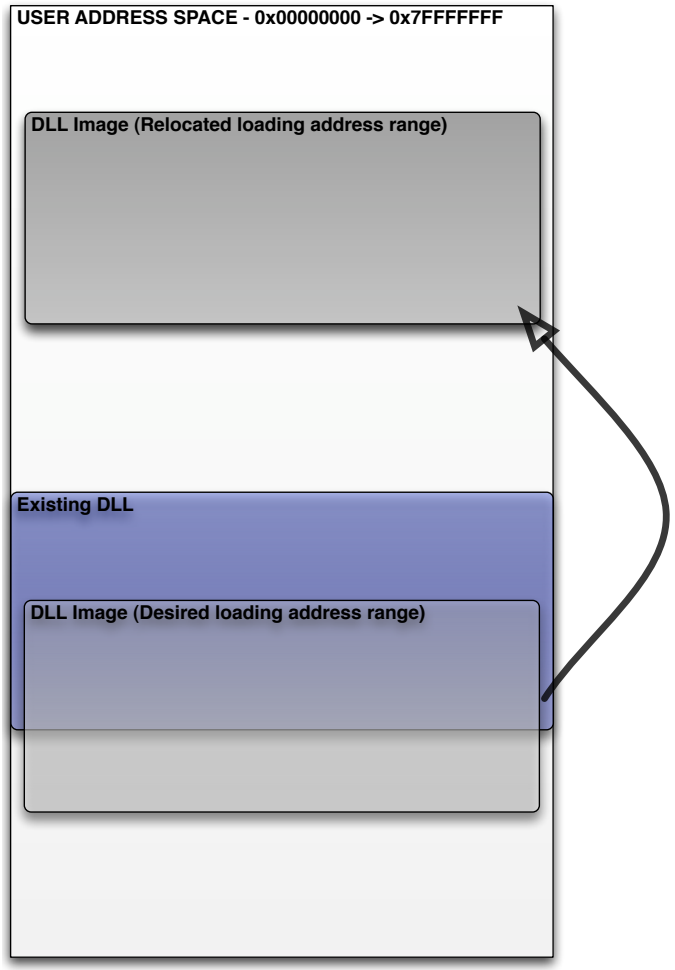
# Relocation of a DLL



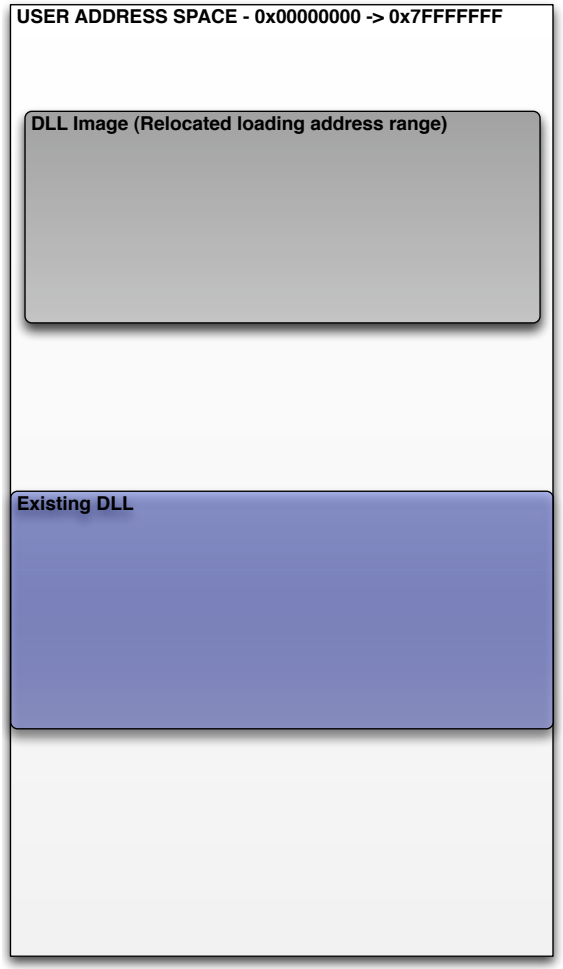
# Relocation of a DLL



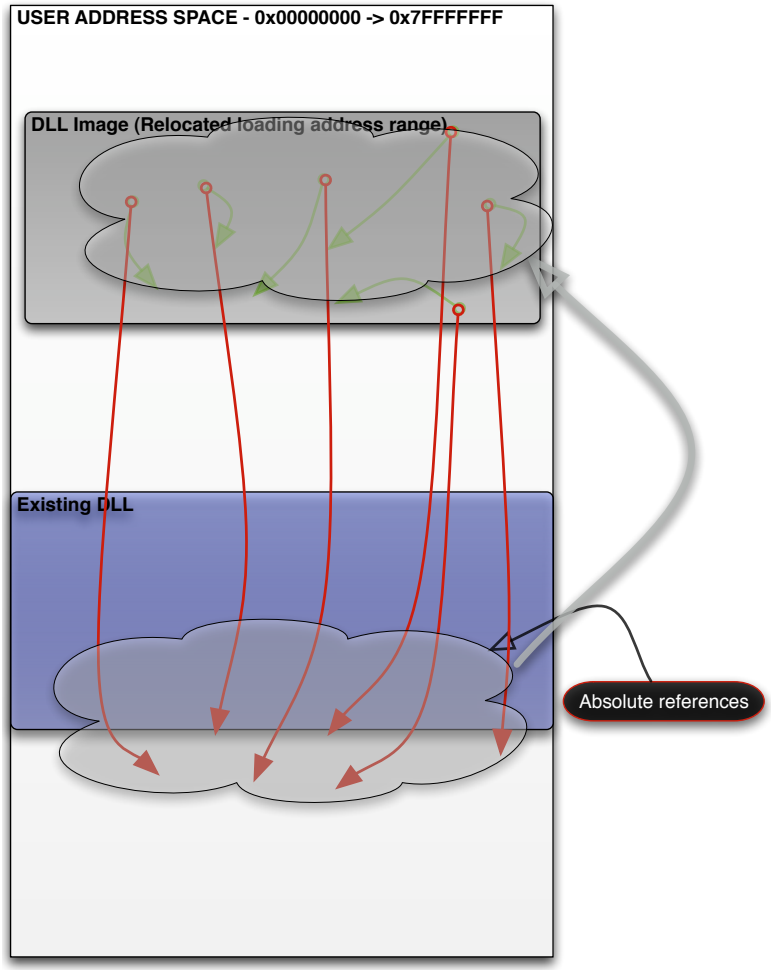
# Relocation of a DLL



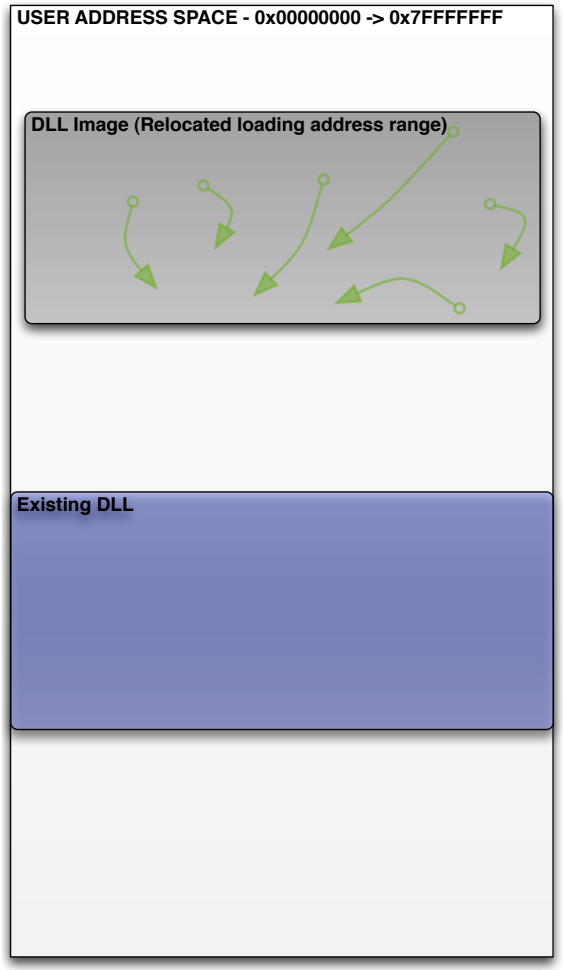
# Relocation of a DLL



# Relocation of a DLL



# Relocation of a DLL



# Virtual Machine Detection

- Some malware have routines for detecting VMWare
- Malicious code that becomes aware that it is being executed in a virtual machine environment may behave completely differently then otherwise expected
- Facts like these keep reverse engineers employed

# Generic Detection

- Timing
  - RDTSC
  - Inconsistencies in the MMU might alter access times as compared to a real machine
  - TLBs might also produce different memory access times
  - External timers, NTPD
- SIDT, SGDT (Redpill) to retrieve the base address of the IDT and GDT.
  - IDT value (hosts): 0x80FFFFFF Windows, 0xC0FFFFFF Linux
  - IDT value (guests): 0xFFXXXXXX VMWare, 0xE8XXXXXX VirtualPC
  - GDT value (hosts): 0xC0XXXXXX
  - The IDT test might fail on SMP systems. There's an IDT per processor

# VMWare Detection

- Trivial checks
  - Checking for VMWare Tools
  - Checking for the service VMTools
  - Spot the files in the filesystem, there are more than 50 files and folders with VMWare/vmx in their name
  - In a guest VM with VMWare Tools installed there will be more than 300 references in the Windows registry and more than 1500 text strings in memory containing VMware

# VMWare Detection

- Hardware
  - MAC address of the network adaptor: 00-05-69, 00-0C-29 or 00-50-56
  - Identifier of the graphics adaptor
- Communication channel
  - IN with EAX=0x564D5868 "VMXh", EBX=parameters, ECX=command, EDX=5658h ("VX", port number)
  - If the command is 0xA (request VMWare's verion) EBX will contain 0x564D5868 "VMXh" if it's a guest system
- There are ways of working around some of these

# VirtualPC Detection

- Device detection
- Communication channel
  - VirtualPC uses invalid instructions to communicate
  - 0F 3F x1 x2
  - 0F C7 C8 y1 y2
- Instructions longer than 15 bytes are invalid but VirtualPC does not raise the corresponding exception
- CPUID returns "ConnetixCPU"

# Other VMs

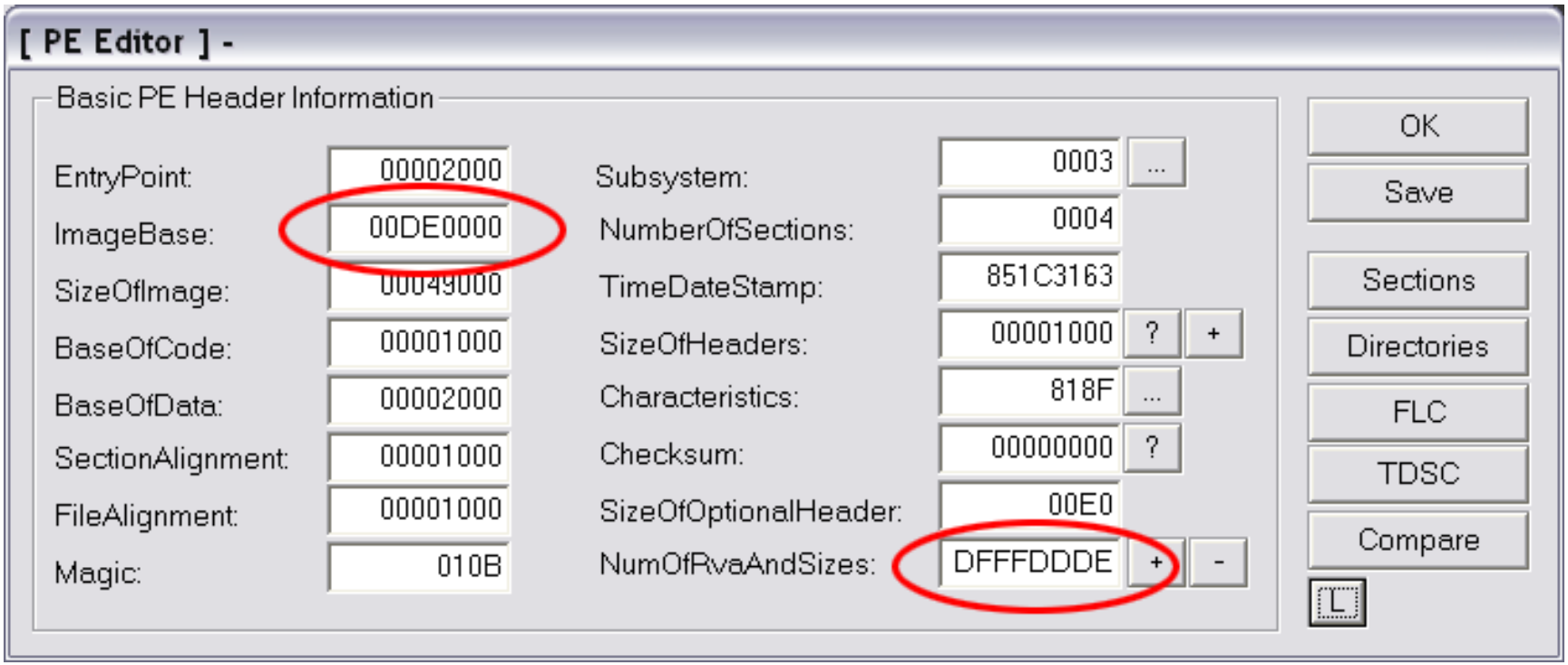
- Peter Ferrie wrote a detailed paper on attacks to all major virtual machine applications
  - [Attacks on More Virtual Machine Emulators]
- Also of interest are
  - [Methods for Virtual Machine Detection]
  - [On the Cutting Edge: Thwarting Virtual Machine Detection]
  - [VMM Detection Myths and Realities]

# Exercises

- Defeat IsDebuggerPresent() check using inline patching
- Verify that VMWare detection works
  - Check VM/Settings/Options/Advanced/DisableAcceleration and try again
- Help IDA disassemble the ASPack decode routine with x86-emu.
- Get 0x90.exe to load in IDA
- At your leisure review the rest of the ridiculous protection mechanisms presented and dissected in:
  - http://project.honeynet.org/scans/scan33/

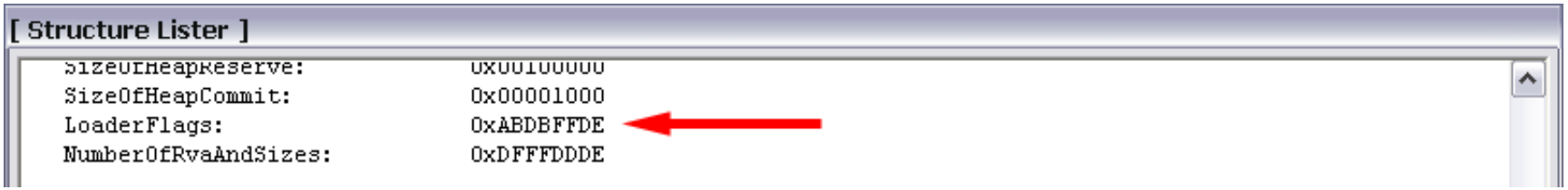
# Solution: PE Optional Header Trickery

- Binary from *Honeynet Project* Scan of The Month 33
- *ImageBase* normally 0x00400000
- Modification is simply a nuisance
- *NumberOfRvaAndSizes* normally 0x00000010



# Solution: PE Optional Header Trickery

- *LoaderFlags* normally *NULL*




- Debugger vulnerabilities discovered by *Nicolas Brulez*
  - *NumberOfRvaAndSizes* modification crashes *SoftICE*
  - *LoaderFlags* + *NumberOfRvaAndSizes* modifications crashes *OllyDBG*
- To fix
  - Set *NumberOfRvaAndSizes* to *0x00000010*
  - Optionally set *LoadFlags* to *0x00000000*

# Solution: PE Section Header Trickery

- Large *SizeOfRawData*
- Causes many tools to crawl or crash
  - Ex: *IDA Pro* will attempt to allocate massive memory chunk

[ Section Table ]

| Name | VOffset | VSize | ROffset | RSize | Flags |
|---|---|---|---|---|---|
| CODE | 00001000 | 00001000 | 00001000 | 00001000 | E0000020 |
| DATA | 00002000 | 00045000 | 00002000 | 00045000 | C0000040 |
| NicolasB | 00047000 | 00001000 | 00047000 | EFEFADFF | C0000040 |
| .idata | 00048000 | 00001000 | 00047000 | 00001000 | C0000040 |

- To fix
  - Set NicolasB section raw size to *0x00000000*
- Note: virtual size == raw size, binary is not compressed

# Outline

10 Executable (Un)Packing

11 Anti Reverse Engineering

12 Binary Diffing and Matching
- Binary Diffing
- Example in Malware Analysis
- Binary Matching
- Exercises

13 Advanced Malware Techniques

# What is it?

- The science of comparing similar binaries to pinpoint changes
  - Ex: Comparing the vulnerable and patched versions of a DLL
- Byte level comparisons fail for a number of reasons
  - Compiler optimizations
  - Branch inversion
  - A single added line of source code or structure variable can change register usage, instruction ordering, offsets etc

# Approaches to Bin Diffing

- Binary and function heuristics
- Graph isomorphism
- Graph heuristics
  - Halvar Flake's work is most notable in this field, has turned into a commercial product and is described in more detail in the coming slides

# Zynamics High Level Algorithm

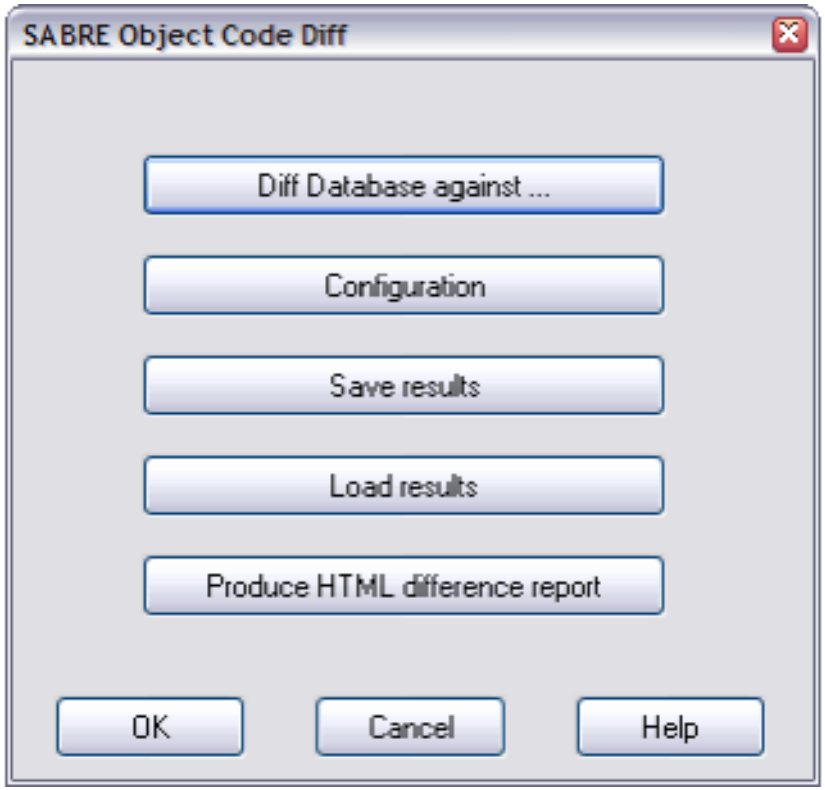
- Function level heuristics
  - Node, edge and call count
  - Small primes product
- Node level heuristics
  - Number of blocks in the shortest path to the entry / exit point, call count
  - Small primes product
- Instruction level heuristics
  - Distance to node entry / exit point
- Small primes product
  - Create a table of small prime numbers
  - Use the instruction opcode as an index into the table
  - Keep a running product
  - Wrap at long long

# Application in Malware Analysis

- Bin Diffing can be used to align functions between variants of malcode and port both names and comments
- Consider the situation where you have spent hours analyzing a binary
  - Analysis time can be saved by porting symbols
  - Analysis time can be saved by pinpointing the exact changes
- Fortunately for the malcode analysts, Bin Diffing malware samples tends to be easier than patch analysis

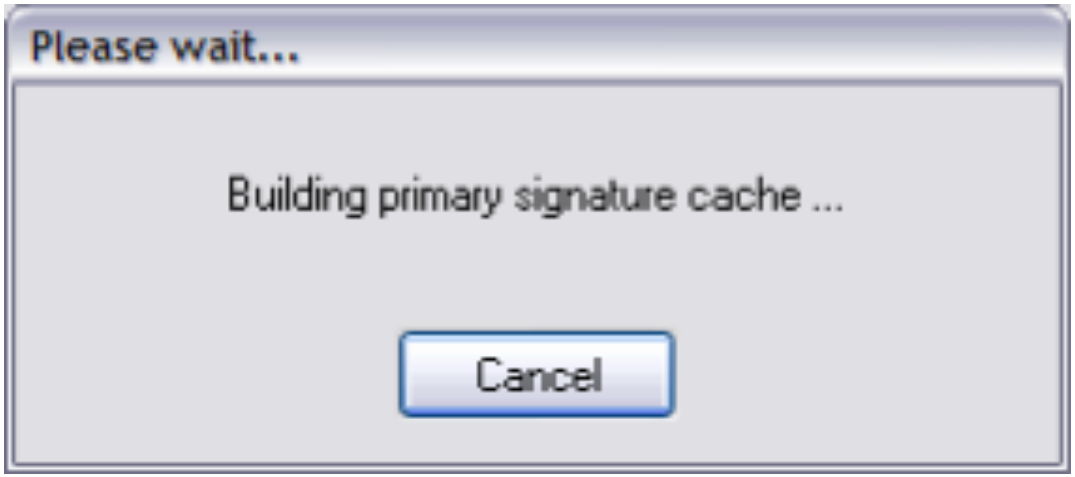
# Invoking BinDiff

- *BinDiff* presents the following dialog when the plug-in is invoked in *IDA*



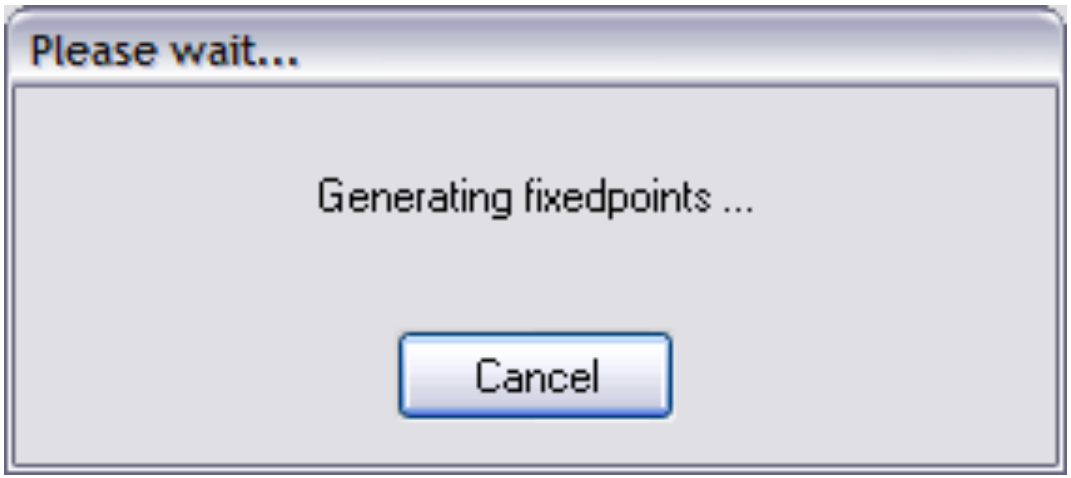
# BinDiff Primary Signature Cache

- *BinDiff* will first generate signatures out of the current *IDB* and the one selected to diff against



# BinDiff Fixed Points

- *BinDiff* will then generate points from which to start matching and will go through several rounds of processing until the set of functions to match is exhausted



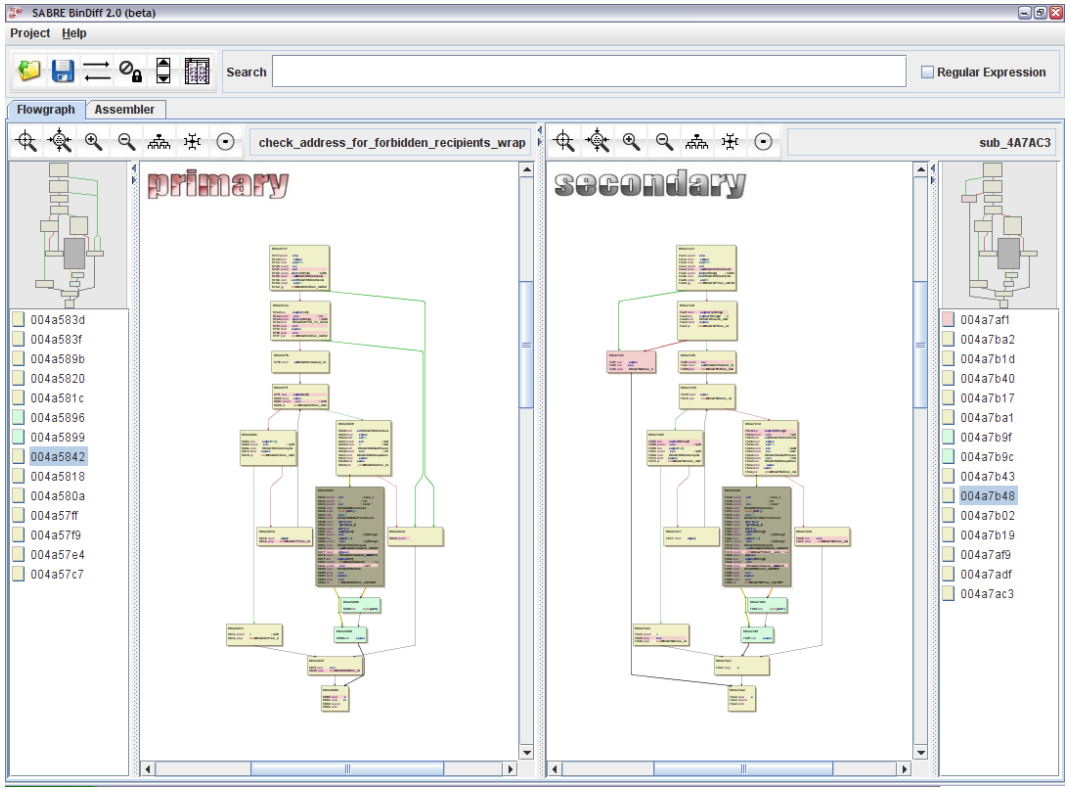
# BinDiff Results

- Once *BinDiff* has completed the matching, it will present three windows, one with the functions that couldn't be matched in the current *IDB*, another with the unmatched functions in the second *IDB* and finally a window showing the matched functions

| Cha... | Function 1 EA | Function 1 Name | Function 2 EA | Function 2 Name |
|---|---|---|---|---|
| No | 405bf4 | create_key | 405bf4 | sub_405BF4 |
| No | 404ee0 | create_mutex | 404ee0 | sub_404EE0 |
| No | 403d10 | decrement_mutex | 403d10 | sub_403D10 |
| No | 406464 | deobfuscate | 406464 | sub_406464 |
| No | 40ae30 | download_execute_and_exit | 40ae30 | download_url |
| No | 405d94 | enum_keys_and_get_info | 405d94 | sub_405D94 |
| No | 405878 | find_next_file_time | 405878 | sub_405878 |
| No | 407e10 | find_port_and_bind | 407e10 | sub_407E10 |
| No | 406370 | get_file_attributes | 406370 | sub_406370 |
| No | 407294 | get_foreground_window_title | 407294 | sub_407294 |
| No | 406398 | get_os_info | 406398 | sub_406398 |
| No | 405960 | get_os_version | 405960 | sub_405960 |
| No | 4058dc | get_time_of_files | 4058dc | sub_4058DC |
| No | 405b48 | get_type | 405b48 | sub_405B48 |
| No | 4041ac | get_val_if_not_zero | 4041ac | sub_4041AC |
| No | 4060a0 | get_win_dir | 4060a0 | sub_4060A0 |
| No | 40c330 | handle_irc_commands | 40c330 | sub_40C330 |

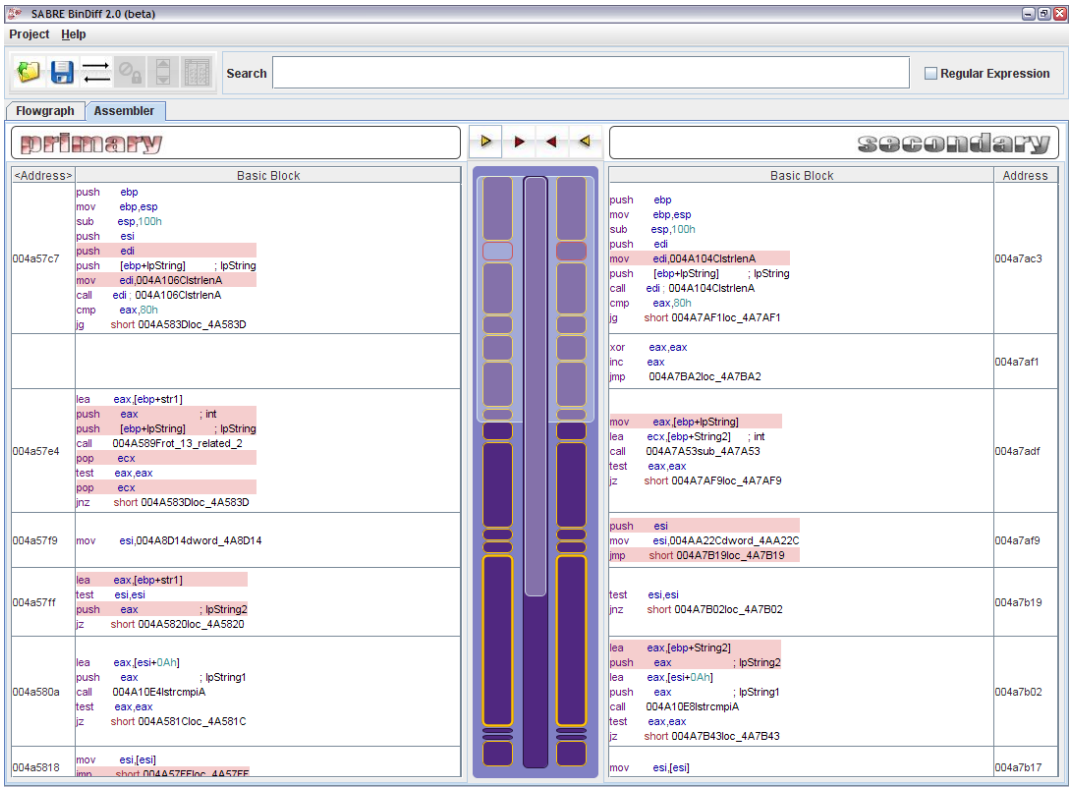
# Close Inspection

- It's also possible to investigate basic block and instruction level changes in order to try to discover the specific changes



# Close Inspection

- BinDiff 2 also offers a line by line assembly view of the changes



# Phylogenetic Classification

- Binary comparison via graph similarities
  - Same heuristics as binary diffing
- Application of clustering analysis to create taxonomy
- Define a similarity function that yields: 0 <= value <= 1
  - Near 0 indicates a low degree of resemblance
  - Near 1 indicates a high degree of resemblance

$$\sigma(A,B) = \frac{|A| \cdot |B|}{|A \bigcup B|^2}$$

- By maintaining large trees of malware, new and unidentified malware can be immediately assigned into a family for classification and porting

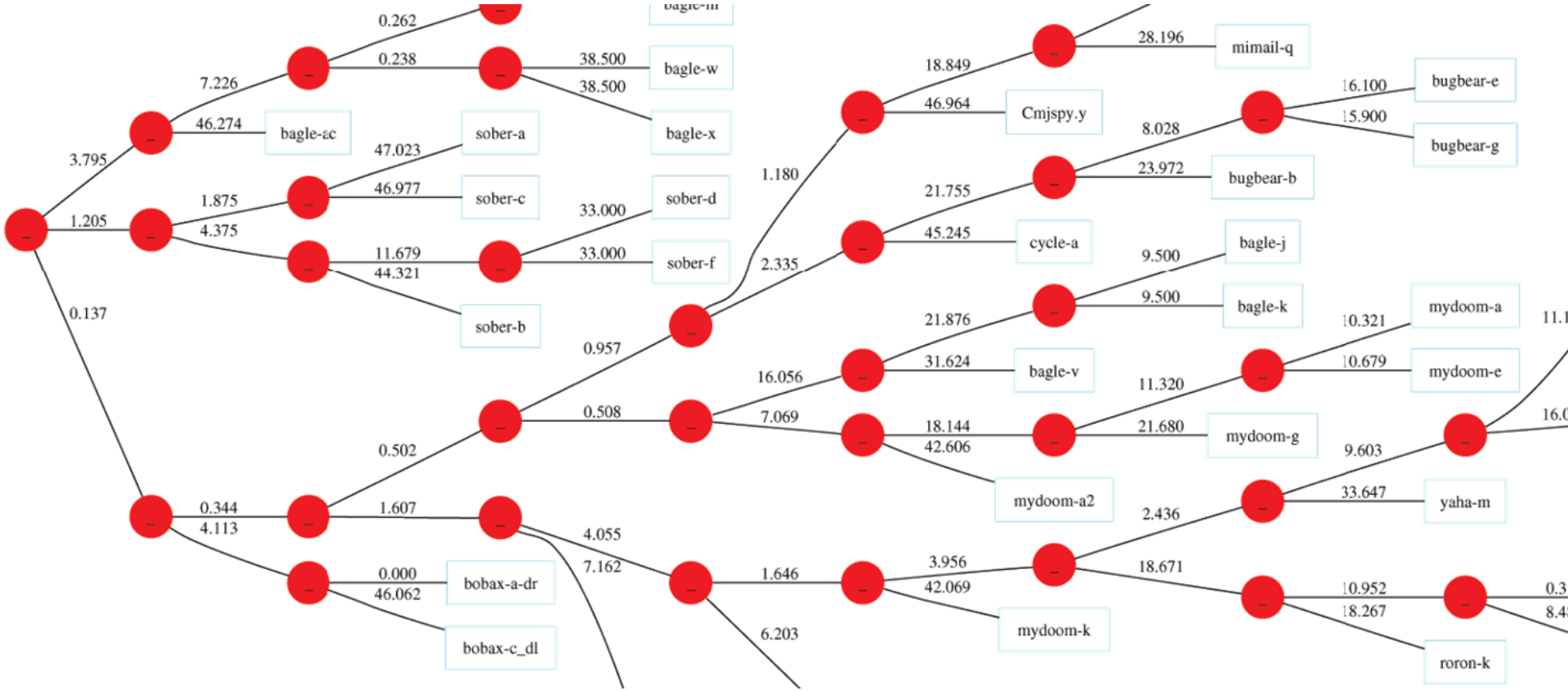
# Phylogenetic Classification

- Using the previously defined equation, distance matrices can be generated:

| | mimail.a | mimail.b | mimail.c | mimail.d | mimail.e | mimail.f |
|---|---|---|---|---|---|---|
| mimail.a | 0 | 90.8 | 85.4 | 87.4 | 75.0 | 75.0 |
| mimail.b | 90.8 | 0 | 84.7 | 88.0 | 74.3 | 74.3 |
| mimail.c | 85.4 | 84.7 | 0 | 81.5 | 81.3 | 81.3 |
| mimail.d | 87.4 | 88.0 | 81.5 | 0 | 72.3 | 72.3 |
| mimail.e | 75.0 | 74.3 | 81.3 | 72.3 | 0 | 95.4 |
| mimail.f | 75.0 | 74.3 | 81.3 | 72.3 | 95.4 | 0 |

# Phylogenetic Classification

- X-tree clustering algorithm can be applied to distance matrices



# Exercise

- Port work from Mydoom.D to Mydoom.M
- Any notable changes?
- Examine the differences between the dropped backdoors

# Outline

10 Executable (Un)Packing

11 Anti Reverse Engineering

12 Binary Diffing and Matching

13 Advanced Malware Techniques
- Advanced Malware Techniques
- Anti-Detection/Obfuscation Measures
- Runtime Hiding Techniques
- Counter-Measures

# Detection Techniques and Counter-Measures

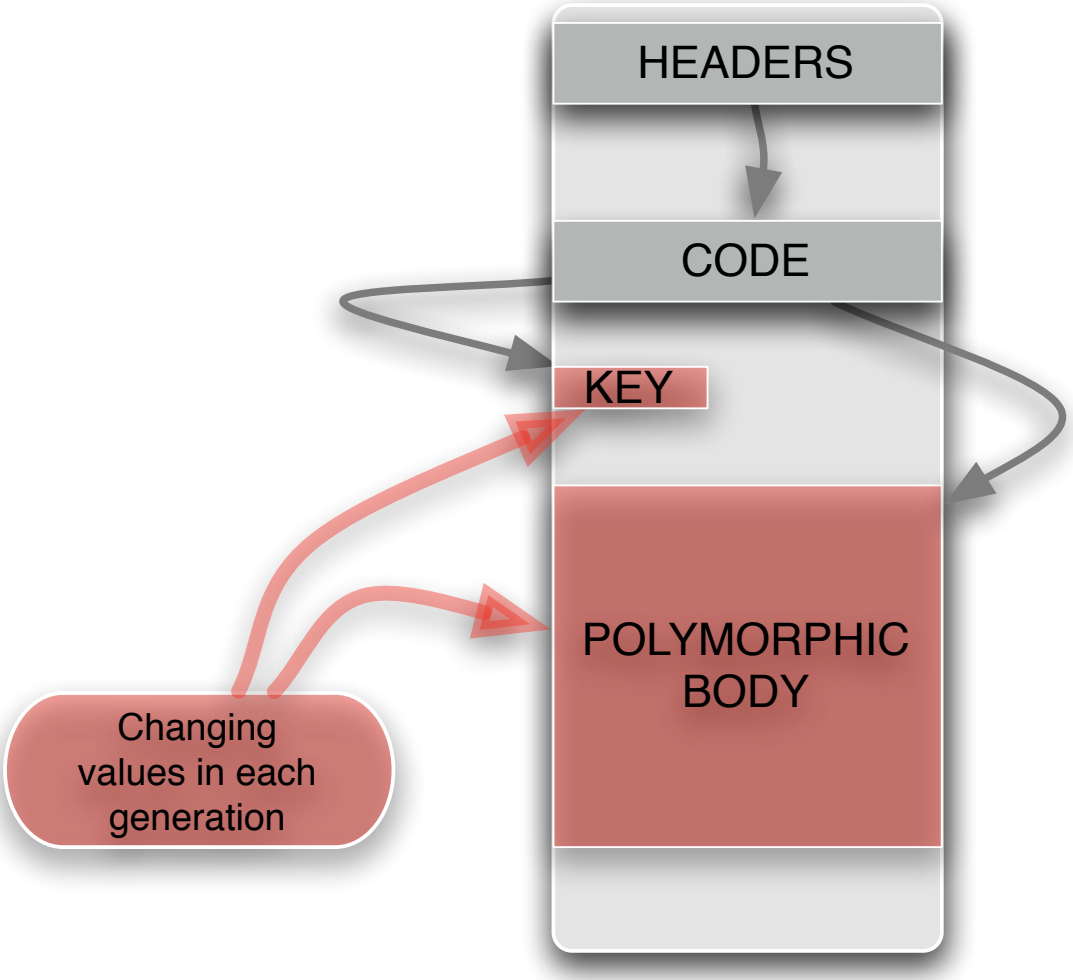
In order to detect malicious malware samples most of the antivirus industry relies on signature matching for detecting malware

- A set of techniques will aim at making the signature based matching useless against identifying malware: polymorphism, server side polymorphism, metamorphism
- Others just aim at making analysis difficult or painful: code obfuscation, virtual machines, entry point obfuscation

# Polymorphism

- Polymorphic malware generates different binaries as it spreads
- Performs relatively simple instruction substitutions or scrambling
- Changes key/algorithm in every iteration
- Code keeps some similar patterns across generations
- Not truly problematic, live analysis is usually an effective measure. High-level behavior seldom changes

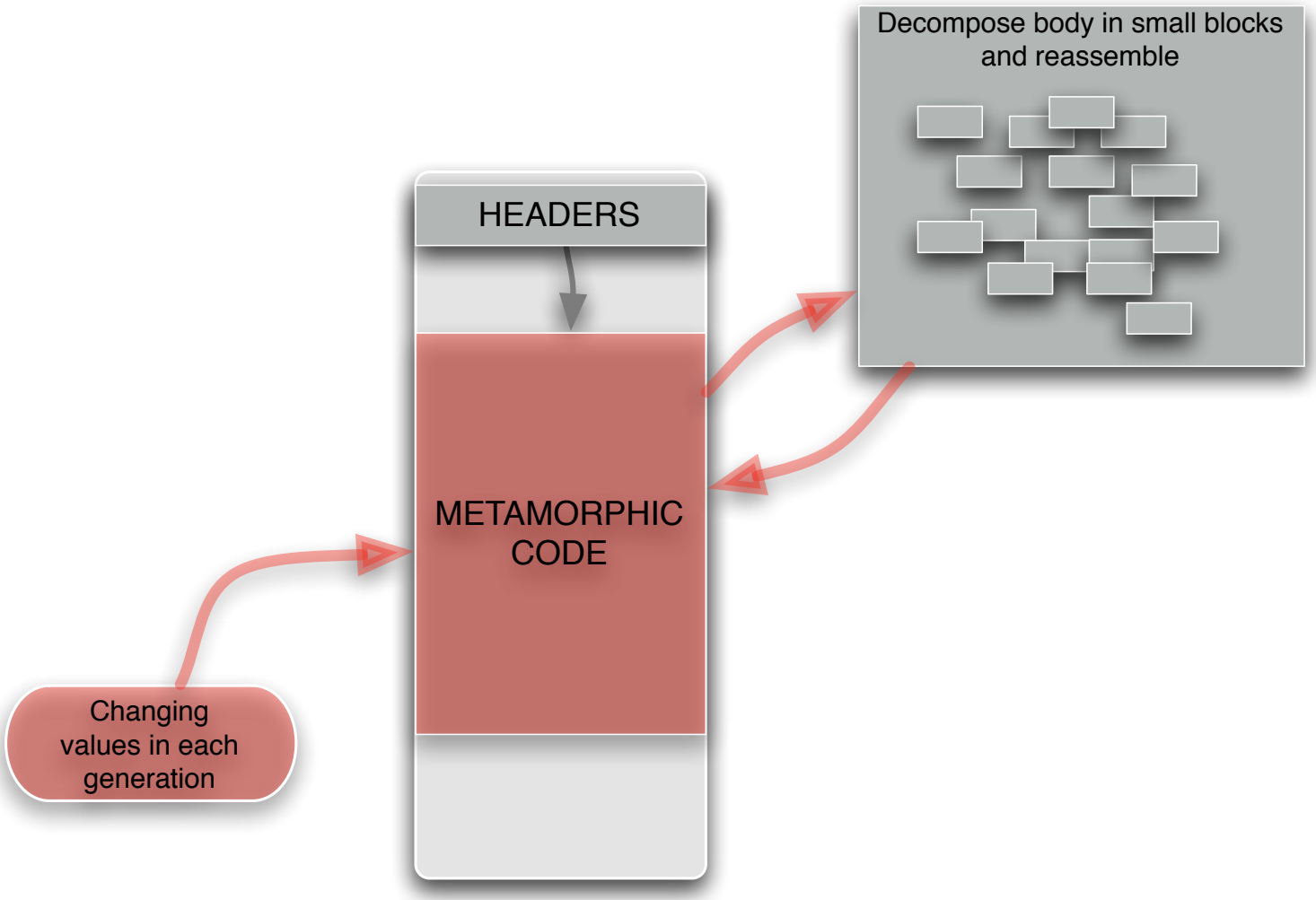
# A View of Polymorphism



# Metamorphism

- Metamorphic malware doesn't just scramble the code but actually generates new code
- Disassembles and reassembles itself, performing the same actions using different code
- Goes beyond simple instruction substitution
- Signature based detection has a tough time

# A View of Metamorphism



# Examples of Metamorphism

[Zmist Opportunities]

[The Viral Darwinism of W32.Evol]

[Simile/MetaPHOR, Striking Similarities] (Metamorphic Permutating High-Obfuscating Reassembler)

# Entry Point Obfuscation

- Used by file infectors
- Hides the entry point to the malicious code
- Finds some functionality of the host application that it's always executed
- Then it diverts the execution flow to its own code, eventually returning to the original code
- The Polip family is a good example

Try writing a small IDAPython script to automatically find the entrypoints to the malicious code

# Virtual Machines

- Aims at making analysis difficult
- Removes the logic of the code by implementing a virtual architecture on which the final code is developed
- An analyst must first unfold this machine's architecture in order to understand the higher level code

# Examples of Virtual Machine Technology

- Themida

http://www.oreans.com/themida.php

- VMProtect

**Source Code**

http://www.polytech.ural.ru/ (in Russian)

- StarForce

http://www.star-force.com/

- T2 Conference Challenge and a few other "crackmes"

# Rootkit Technology

## Overview

Rootkits aim at making themselves and other executables undetectable to the underlying system. For malware, the main purpose of rootkit technology is to remain on the infected host as long as possible without being detected and therefore disinfected

- Rootkits normally run in kernel mode
- Commonly they hook *API* functionality and filter those events that would expose them
- *DKOM* (*Direct Kernel Object Manipulation*) is a more advanced technique by which rootkits alter executive objects in order to remove any traces of them running

# Hypervisor Technology

*From Wikipedia*

A hypervisor in computing is a scheme which allows multiple operating systems to run, unmodified, on a host computer at the same time. The term is an extension of the earlier term supervisor, which was commonly applied to operating system kernels in that era.

# Hypervisor Technology

- Both *Intel* and *AMD* have their own hardware virtualization extensions
- *AMD*'s is currenly codenamed *Pacifica*
- *Intel*'s is known as *VT* or *Vanderpool*
- Both technologies are already available on consumer products.
- *Microsoft's Virtual PC* and *Virtual Server*, *Parallels Workstation*, *TRANGO*, *VMware* and *Xem* already employ the virtualization extensions if available
- *Blue Pill* uses undocumented features in *Pacifica* to take over the system

http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html

# Poly/Meta morphism

- Polymorphism can be fought with emulation and live analysis techniques
- Metamorphism and VM techniques are trickier
- Behavioral analysis is effective as long as the malware in question performs distinctive actions using common APIs

# Rootkits

- To detect active rootkits one of the common approaches is to attempt to gather the same information through different channels. For instance:
  - Using the operating system's standard *API*s
  - Using low level techniques, thus bypassing the *API*s
  - Attempting to obtain handles to every possible process
- Proceeding to compare the two sets of results. Disparities between them will tend to indicate that something might trying to hide
- Detecting hardware-virtualized malware might actually be impossible if the implementation has no bugs... ;-)

# Part IV

# Analysis and Custom Development

# Outline

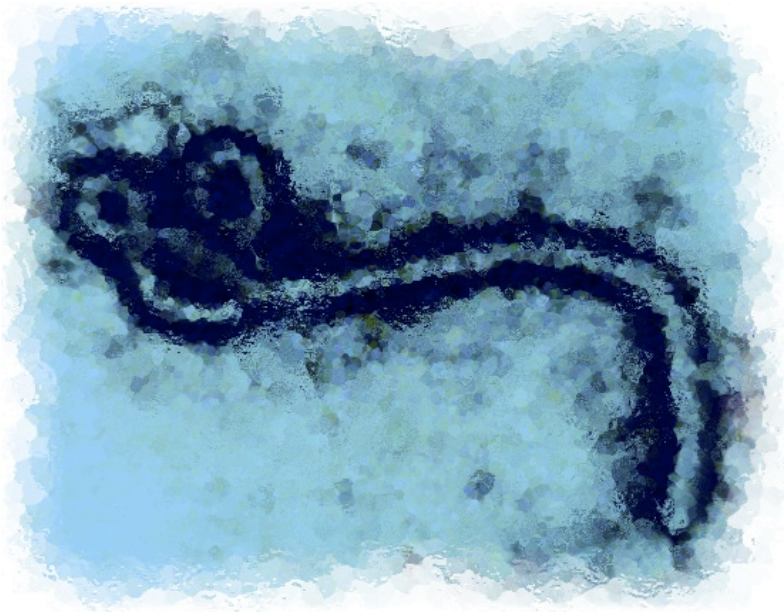
14 Analysis
- Analysis I
- Analysis II

15 IDA Python

16 PEFile and PyDasm

17 PaiMei

# The Target



- SHA1: *c9f10fa5135e3f10c1fb942d12bb8f267e4203d0*
- MD5: *04c94ee7122a2844e12afe0928806fa0*

# Is it Packed?

- If so, generally there are no visible strings.

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`

# Is it Packed?

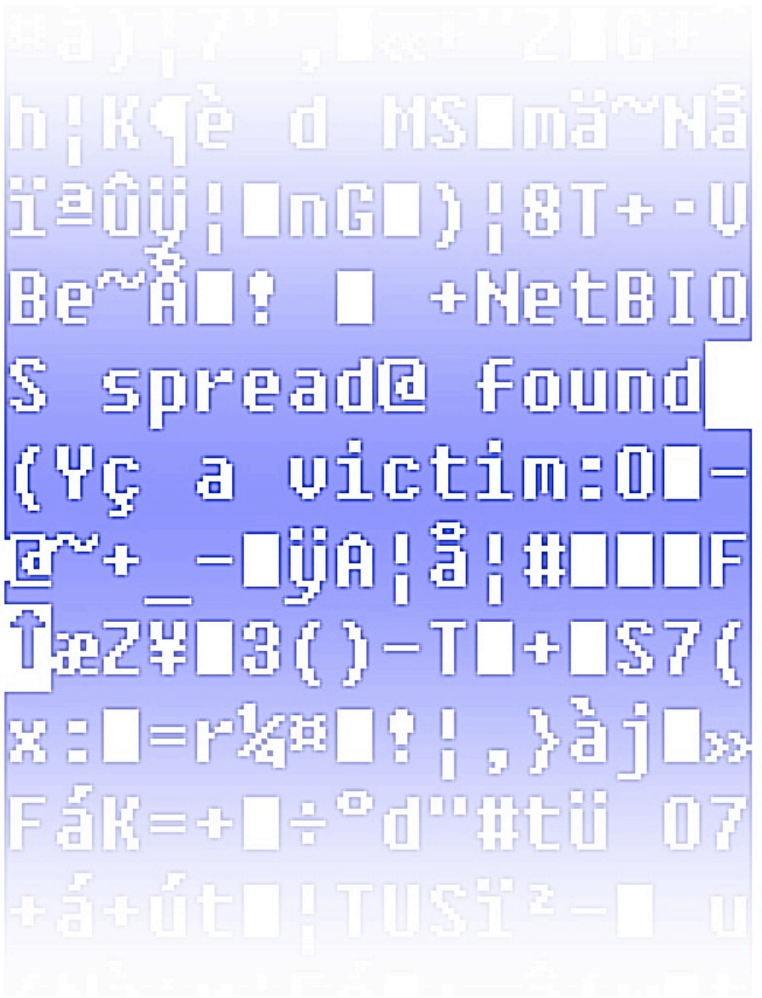
- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`
- Recall that we can utilize statistical tests as an indicator

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`
- Recall that we can utilize statistical tests as an indicator
  - A high entropy indicates a *very-likely-to-be-compressesed* data section

# What is it Packed With?

- Can we guess the packer?
- Does *PEiD* help?
- Does a hexeditor expose any clues?



# First Look in IDA

Notice that opening the file in *IDA* results in complaints about a missing import table. The navigation bar reveals very little code. What we see is the *UPX* unpacking code with its typical start and finish.

```
.UPX1:0041B9BF                public start
.UPX1:0041B9BF        start   proc near
.UPX1:0041B9BF 000            nop
.UPX1:0041B9C0 000            popa
.UPX1:0041B9C1 -20            mov     esi, offset dword_412000
.UPX1:0041B9C6 -20            lea     edi, [esi-11000h]
.UPX1:0041B9CC -20            mov     dword ptr [edi+126F8h], 1941BFB7h
.UPX1:0041B9D6 -20            push    edi
.UPX1:0041B9D7 -1C            or      ebp, 0FFFFFFFFh
.UPX1:0041B9DA -1C            jmp     short loc_41B9EA
.UPX1:0041B9DA
```

# First Look in IDA

Notice that opening the file in *IDA* results in complaints about a missing import table. The navigation bar reveals very little code. What we see is the *UPX* unpacking code with its typical start and finish.

```
.UPX1:0041BB0B -14          add     ebx, 4
.UPX1:0041BB0E -14          jmp     short loc_41BAF1
.UPX1:0041BB0E
.UPX1:0041BB10      ; ---------------------------------------
.UPX1:0041BB10
.UPX1:0041BB10      loc_41BB10:
.UPX1:0041BB10 -14          call    dword ptr [esi+1B9FCh]
.UPX1:0041BB10
.UPX1:0041BB16
.UPX1:0041BB16      loc_41BB16:
.UPX1:0041BB16 -14          pusha
.UPX1:0041BB17 00C          jmp     near ptr unk_40F32C
.UPX1:0041BB17
.UPX1:0041BB17      start   endp
```

# Bypassing UPX

- Despite its simplicity, *UPX* is very commonly used

# Bypassing UPX

- Despite its simplicity, *UPX* is very commonly used
- Set a breakpoint at the jump to the *OEP* (not in the destination)

# Bypassing UPX

- Despite its simplicity, *UPX* is very commonly used
- Set a breakpoint at the jump to the *OEP* (not in the destination)
- Continue the process and let the unpacker do its work

# Bypassing UPX

- Despite its simplicity, *UPX* is very commonly used
- Set a breakpoint at the jump to the *OEP* (not in the destination)
- Continue the process and let the unpacker do its work
- Once the breakpoint is hit we can step to the next instruction to find the unpacked code

# OEP Before and After

```
.UPX0:0040F32C     unk_40F32C db    ? ;
.UPX0:0040F32D                db    ? ;
.UPX0:0040F32E                db    ? ;
.UPX0:0040F32F                db    ? ;
.UPX0:0040F330                db    ? ;
.UPX0:0040F331                db    ? ;
.UPX0:0040F332                db    ? ;
.UPX0:0040F333                db    ? ;
.UPX0:0040F334                db    ? ;
.UPX0:0040F335                db    ? ;
.UPX0:0040F336                db    ? ;
.UPX0:0040F337                db    ? ;
.UPX0:0040F338                db    ? ;
.UPX0:0040F339                db    ? ;
.UPX0:0040F33A                db    ? ;
.UPX0:0040F33B                db    ? ;
.UPX0:0040F33C                db    ? ;
```

# OEP Before and After

```
.UPX0:0040F32C                  public start
.UPX0:0040F32C      start       proc near
.UPX0:0040F32C
.UPX0:0040F32C      var_12C = dword ptr -12Ch
.UPX0:0040F32C      var_128 = dword ptr -128h
.UPX0:0040F32C      var_124 = dword ptr -124h
.UPX0:0040F32C      var_120 = dword ptr -120h
.UPX0:0040F32C      var_11C = dword ptr -11Ch
.UPX0:0040F32C      var_118 = dword ptr -118h
.UPX0:0040F32C      var_114 = dword ptr -114h
.UPX0:0040F32C      var_14  = dword ptr -14h
.UPX0:0040F32C
.UPX0:0040F32C 000            push    ebp
.UPX0:0040F32D 004            mov     ebp, esp
.UPX0:0040F32F 004            add     esp, 0FFFFFED4h
.UPX0:0040F335 130            xor     eax, eax
.UPX0:0040F337 130            mov     [ebp+var_12C], eax
.UPX0:0040F33D 130            mov     [ebp+var_128], eax
.UPX0:0040F343 130            mov     [ebp+var_120], eax
.UPX0:0040F349 130            mov     [ebp+var_124], eax
.UPX0:0040F34F 130            mov     [ebp+var_11C], eax
.UPX0:0040F355 130            mov     [ebp+var_118], eax
.UPX0:0040F35B 130            mov     [ebp+var_14], eax
.UPX0:0040F35E 130            mov     eax, offset unk_40F264
.UPX0:0040F363 130            call    sub_404DAC
```

# Dumping the Image

- We need to save the unpacked image to disk for analysis

# Dumping the Image

- We need to save the unpacked image to disk for analysis
- The *OllyDump* plugin can do this for us

# Dumping the Image

- We need to save the unpacked image to disk for analysis
- The *OllyDump* plugin can do this for us
- *OllyDump* can also rebuild the *IAT*

# Dumping the Image

- We need to save the unpacked image to disk for analysis
- The *OllyDump* plugin can do this for us
- *OllyDump* can also rebuild the *IAT*
- *ProcDump* and *ImpRec* are other tools for accomplishing the job

# Dumping the Image

- We need to save the unpacked image to disk for analysis
- The *OllyDump* plugin can do this for us
- *OllyDump* can also rebuild the *IAT*
- *ProcDump* and *ImpRec* are other tools for accomplishing the job
- Now we are ready to jump back to IDA

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

- A good amount of code is not properly recognized, we will need to manually define it

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

- A good amount of code is not properly recognized, we will need to manually define it
- Function ends are often not found properly, we must manually fix these

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

- A good amount of code is not properly recognized, we will need to manually define it
- Function ends are often not found properly, we must manually fix these
- There will be tables of pointers interleaved within the code, these are *Delphi*'s structures

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

- A good amount of code is not properly recognized, we will need to manually define it
- Function ends are often not found properly, we must manually fix these
- There will be tables of pointers interleaved within the code, these are *Delphi*'s structures
- *IDA* might not recognize library functions, therefore we might need to spot these manually

# The Analysis

This executable has a range of interesting features and there are certain considerations to take into account during the analysis

- A good amount of code is not properly recognized, we will need to manually define it
- Function ends are often not found properly, we must manually fix these
- There will be tables of pointers interleaved within the code, these are *Delphi*'s structures
- *IDA* might not recognize library functions, therefore we might need to spot these manually
- A good approach is to linearly scan the binary (if it's not too large)

# Exploring the Binary

One of the first functions, called (at *40F37Bh*) is interesting.
We can get to it in two ways. One is exploring from the beginning of the binary and the second is by following backwards references like the following

```
.UPX0:0040AC62 11AC          mov     eax, ds:off_412828
.UPX0:0040AC67 11AC          mov     eax, [eax]
.UPX0:0040AC69 11AC          call    eax
```

# Exploring the Binary

```
.UPX0:0040AC62 11AC          mov     eax, ds:off_412828
.UPX0:0040AC67 11AC          mov     eax, [eax]
.UPX0:0040AC69 11AC          call    eax
```

# Exploring the Binary

```
.UPX0:0040AC62 11AC          mov     eax, ds:off_412828
.UPX0:0040AC67 11AC          mov     eax, [eax]
.UPX0:0040AC69 11AC          call    eax
```

- Following the offset *off_412828* we will find ourselves at *413D44h*. This address has two references and is actually written to by one of them (*DATA XREF: sub_406764+288*)

# Exploring the Binary

```
.UPX0:0040AC62 11AC          mov     eax, ds:off_412828
.UPX0:0040AC67 11AC          mov     eax, [eax]
.UPX0:0040AC69 11AC          call    eax
```

- Following the offset *off_412828* we will find ourselves at *413D44h*. This address has two references and is actually written to by one of them (*DATA XREF: sub_406764+288*)
- Inspecting the reference we see this and other addresses are being written to, all of which are called

# Exploring the Binary

```
.UPX0:0040AC62 11AC          mov     eax, ds:off_412828
.UPX0:0040AC67 11AC          mov     eax, [eax]
.UPX0:0040AC69 11AC          call    eax
```

- Following the offset *off_412828* we will find ourselves at *413D44h*. This address has two references and is actually written to by one of them (*DATA XREF: sub_406764+288*)
- Inspecting the reference we see this and other addresses are being written to, all of which are called
- Hence we have more imports to fix. This time the program seems to be using a hash based approach, where it's difficult to tell the import from the value fed

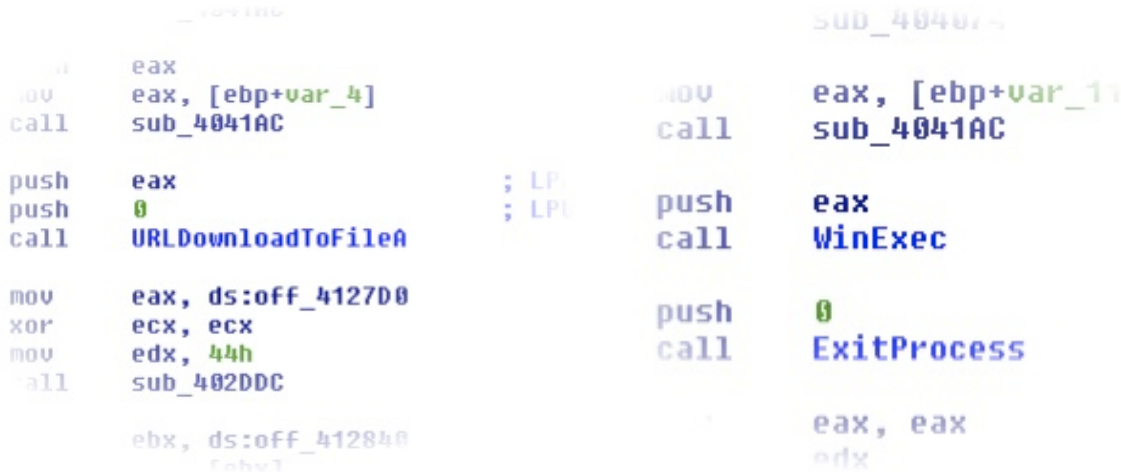
# Resolving the Imports

If we trace the function with *OllyDbg* we can see that a good set of *API* functions are imported

```
.UPX0:004069E0 030          mov     [ebx], eax
.UPX0:004069E2 030          mov     eax, 0A7733ACDh
.UPX0:004069E7 030          call    resolve_import
.UPX0:004069E7
.UPX0:004069EC 030          mov     ds:send, eax
.UPX0:004069F1 030          mov     eax, 0A0F5FC93h
.UPX0:004069F6 030          call    resolve_import
.UPX0:004069F6
.UPX0:004069FB 030          mov     ds:WSAStartUp, eax
.UPX0:00406A00 030          mov     eax, 5E568BBh
.UPX0:00406A05 030          call    resolve_import
.UPX0:00406A05
.UPX0:00406A0A 030          mov     ds:socket, eax
```

# A Quick Glimpse of the Imports

- Of special interest are the functions from the networking *API*
- Combined with a look at the imports gives us dire forecast
- Further inspection reveals that this is indeed malicious



# Interesting Code Construct

```
.UPX0:00406086 000          mov     eax, eax
.UPX0:00406088 000          push    ebx
.UPX0:00406089 004          push    esi
.UPX0:0040608A 008          mov     esi, edx
.UPX0:0040608C 008          mov     ebx, eax
.UPX0:0040608E 008          push    esi
.UPX0:0040608F 00C          push    ebx
.UPX0:00406090 010          call    ds:__WSAFDIsSet
.UPX0:00406090
.UPX0:00406096 008          cmp     eax, 1
.UPX0:00406099 008          sbb     eax, eax
.UPX0:0040609B 008          inc     eax
.UPX0:0040609C 008          pop     esi
.UPX0:0040609D 004          pop     ebx
.UPX0:0040609E 000          retn
```

# Branchless Logic

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0

# Branchless Logic

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)

# Branchless Logic

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1

# Branchless Logic

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1
- Otherwise if eax is greater than 0 we have eax = eax - eax+0 = 0

# Branchless Logic

```
cmp     eax, 1
sbb     eax, eax
inc     eax
pop     esi
pop     ebx
retn
```

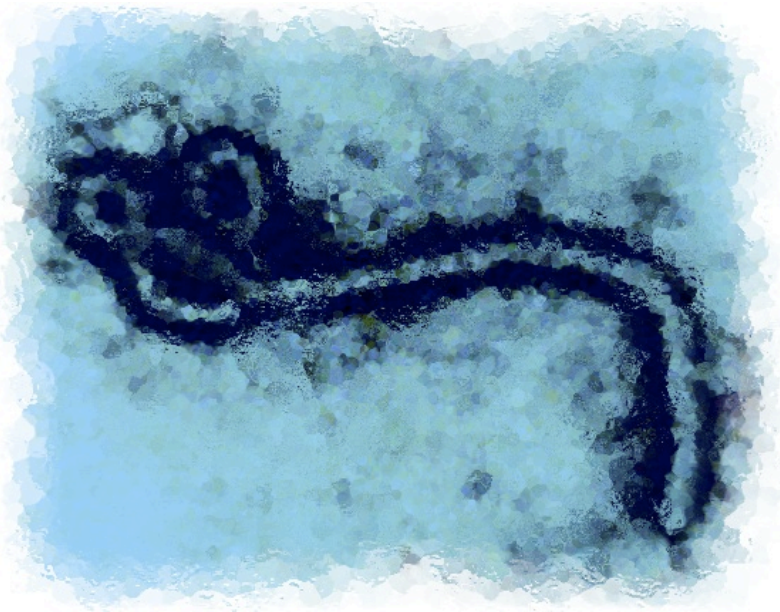
- cmp eax, 1 will set the carry flag (CF) if eax is 0
- sbb eax, eax does eax = eax - (eax+CF)
- Therefore if eax was 0 we have eax = 0 - (0+1) = -1
- Otherwise if eax is greater than 0 we have eax = eax - eax+0 = 0
- The increment will set the possible eax values to 1 or 0

# Hotspots

Points of interest in the binary are

- Import resolution *40F37Bh*
- Deobfuscation function *406464h*
- *NetBios* spreading *409408h*
- Connect to *Mydoom*'s backdoor *409E50*
- Download and execute *40AE30h*
- Build system info summary *40BA14h*
- Handle *IRC* commands *40C330h*

# The Target Revealed



- Kaskpersky: *Backdoor.Win32.Gobot.w*
- McAfee: *Exploit-Mydoom*
- Symantec: *W32.Gobot.A*
- Trend Micro: *BKDR_GOBOT.W*

- SHA1: *c9f10fa5135e3f10c1fb942d12bb8f267e4203d0*
- MD5: *04c94ee7122a2844e12afe0928806fa0*

# The Target



- SHA1: *f30ad924a0d35b13d2057b1bb6305ad5a8ac8fa2*
- MD5: *0af7b122bb722fb679c97fdb8cf85d23*

# Is it Packed?

- If so, generally there are no visible strings.

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`

# Is it Packed?

- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`
- Recall that we can utilize statistical tests as an indicator

# Is it Packed?

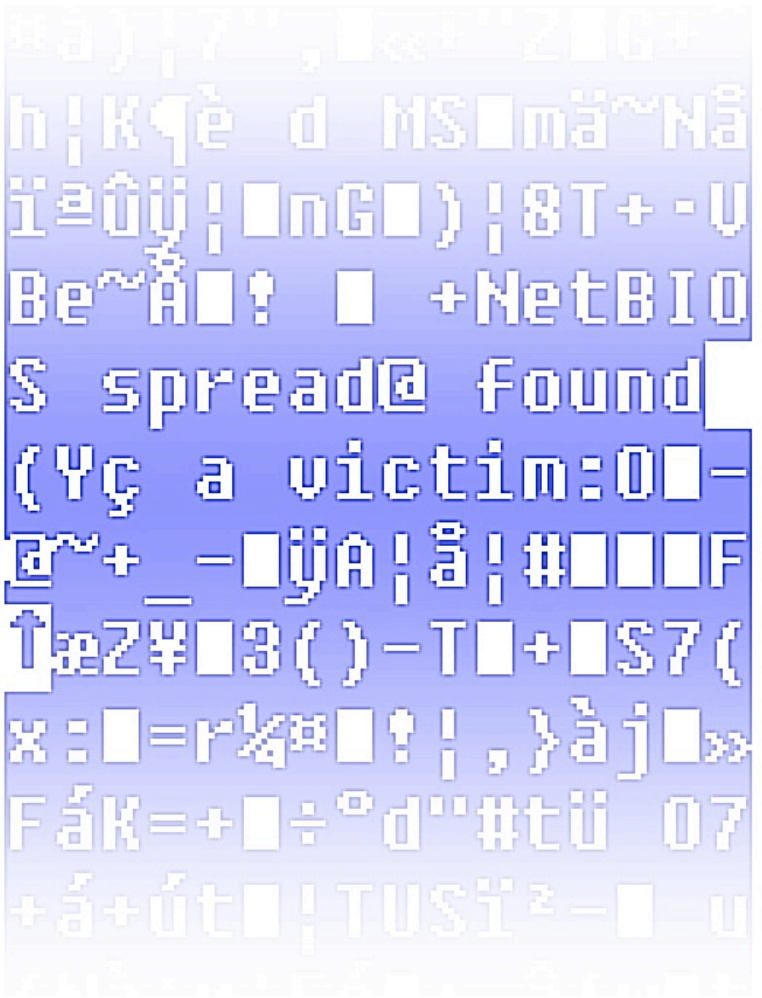
- If so, generally there are no visible strings.
  - Remember: Unicode strings have a non-occidental charset
- Most common imports will not be present
- Perhaps only `LoadLibrary` and `GetProcAddress`
- Recall that we can utilize statistical tests as an indicator
  - A high entropy indicates a *very-likely-to-be-compressesed* data section

# What is it Packed With?

- Can we guess the packer?
- Does *PEiD* help?
- Does a hexeditor expose any clues?

# What is it Packed With?

- Can we guess the packer?
- Does *PEiD* help?
- Does a hexeditor expose any clues?



# First look in IDA (1)

Here we can see a basic obfuscation technique. Upon execution it will first go through an *XOR* loop to reveal some additional code. Execution continues at the unveiled code

```
.yvs_:0041D000                public start
.yvs_:0041D000        start:
.yvs_:0041D000                push    59E0F1h
.yvs_:0041D005                pop     eax
.yvs_:0041D006                nop
.yvs_:0041D007                nop
.yvs_:0041D008                mov     esi, (offset loc_41D01D+1)
.yvs_:0041D00D                mov     edx, 1432
.yvs_:0041D012                nop
.yvs_:0041D013
.yvs_:0041D013        XOR_loop:                               ; CODE XRE
.yvs_:0041D013                xor     [edx+esi], eax
.yvs_:0041D016                sub     edx, 2
.yvs_:0041D019                sub     edx, 2
.yvs_:0041D01C                nop
.yvs_:0041D01D
.yvs_:0041D01D        loc_41D01D:                             ; DATA XRE
.yvs_:0041D01D                jnz     short XOR_loop
.yvs_:0041D01D
.yvs_:0041D01F                nop
```

# First look in IDA (2)

This is the result after the deobfuscation is finished

```
.yvs_:0041D000            push    59E0F1h
.yvs_:0041D005            pop     eax
.yvs_:0041D006            nop
.yvs_:0041D007            nop
.yvs_:0041D008            mov     esi, 41D01Eh
.yvs_:0041D00D            mov     edx, 598h
.yvs_:0041D012            nop
.yvs_:0041D013
.yvs_:0041D013        loc_41D013:
.yvs_:0041D013            xor     [edx+esi], eax
.yvs_:0041D016            sub     edx, 2
.yvs_:0041D019            sub     edx, 2
.yvs_:0041D01C            nop
.yvs_:0041D01D            jnz     short loc_41D013
.yvs_:0041D01D
.yvs_:0041D01F            nop
.yvs_:0041D020            nop
.yvs_:0041D021            nop
.yvs_:0041D022            call    start
```

# Lack of imports

We can now follow the newly available code in *IDA*. We soon realize that there's not much we can tell as there are no imported symbols available

```
.yvs_:0041D27C 018          call    dword ptr [ebx+4]
.yvs_:0041D27C
.yvs_:0041D27F 010          mov     edx, [ebp+arg_4]
.yvs_:0041D282 010          mov     ecx, [edx+20h]
.yvs_:0041D285 010          mov     [ecx], eax
.yvs_:0041D287 010          mov     eax, [ebp+arg_4]
.yvs_:0041D28A 010          push    dword ptr [eax+28h]
.yvs_:0041D28D 014          push    esi
.yvs_:0041D28E 018          call    dword ptr [ebx+4]
.yvs_:0041D28E
.yvs_:0041D291 010          mov     edx, [ebp+arg_4]
.yvs_:0041D294 010          mov     ecx, [edx+20h]
.yvs_:0041D297 010          add     ecx, 4
.yvs_:0041D29A 010          mov     [ecx], eax
.yvs_:0041D29C 010          lea     eax, [edi+0Dh]
.yvs_:0041D29F 010          push    eax
.yvs_:0041D2A0 014          push    esi
.yvs_:0041D2A1 018          call    dword ptr [ebx+4]
```

# *OllyDbg* to the Rescue

*OllyDbg* can help in this case.

# *OllyDbg* to the Rescue

*OllyDbg* can help in this case.

- Tracing to the interesting locations, we can see where the addresses are actually pointing to

```
0041D263  8B0A          mov     ecx, dword ptr ds:[edx]
0041D265  894B 04       mov     dword ptr ds:[ebx+4], ecx
0041D268  57            push    edi
0041D269  FF13          call    dword ptr ds:[ebx]          kernel32.LoadLibrary
0041D26B  8BF0          mov     esi, eax
0041D26D  85F6          test    esi, esi
0041D26F  0F84 8C000000 je      0af7b122.0041D301
0041D275  8B45 0C       mov     eax, dword ptr ss:[ebp+C]
Stack ds:[0012FF84]=7C801D77 (kernel32.LoadLibraryA)
```

# *OllyDbg* to the Rescue

*OllyDbg* can help in this case.

- Tracing to the interesting locations, we can see where the addresses are actually pointing to
- The binary is resolving the imported symbols by itself

```
0041D291  8B55 0C     mov     edx, dword ptr ss:[ebp+C]
0041D294  8B4A 20     mov     ecx, dword ptr ds:[edx+20]
0041D297  83C1 04     add     ecx, 4
0041D29A  8901        mov     dword ptr ds:[ecx], eax
0041D29C  8D47 0D     lea     eax, dword ptr ds:[edi+D]
0041D29F  50          push    eax
0041D2A0  56          push    esi
0041D2A1  FF53 04     call    dword ptr ds:[ebx+4]
0041D2A4  8943 08     mov     dword ptr ds:[ebx+8], eax
0041D2A7  8D57 1A     lea     edx, dword ptr ds:[edi+1A]
0041D2AA  52          push    edx
0041D2AB  56          push    esi
0041D2AC  FF53 04     call    dword ptr ds:[ebx+4]
0041D2AF  8943 0C     mov     dword ptr ds:[ebx+C], eax      kernel32.GetTempFile
0041D2B2  8D4F 2B     lea     ecx, dword ptr ds:[edi+2B]
0041D2B5  51          push    ecx
eax=7C8606DF (kernel32.GetTempFileNameA)
Stack ds:[0012FF90]=00000000
```

# *OllyDbg* to the Rescue

*OllyDbg* can help in this case.

- Tracing to the interesting locations, we can see where the addresses are actually pointing to
- The binary is resolving the imported symbols by itself

```
0012FF7C  81FBEFE0
0012FF80  00001FE0
0012FF84  7C801D77  kernel32.LoadLibraryA
0012FF88  7C80AC28  kernel32.GetProcAddress
0012FF8C  7C8221CF  kernel32.GetTempPathA
0012FF90  7C8606DF  kernel32.GetTempFileNameA
0012FF94  7C801A24  RETURN to kernel32.CreateFileA
0012FF98  7C80180E  kernel32.ReadFile
0012FF9C  7C810F9F  kernel32.WriteFile
0012FFA0  7C810DA6  kernel32.SetFilePointer
0012FFA4  7C809B77  kernel32.CloseHandle
0012FFA8  7C8092AC  kernel32.GetTickCount
0012FFAC  7C80B357  kernel32.GetModuleFileNameA
0012FFB0  77DD761B  advapi32.RegOpenKeyExA
0012FFB4  77DD7883  RETURN to advapi32.RegQueryValueExA
0012FFB8  77DD6BF0  advapi32.RegCloseKey
0012FFBC  0012FFF0
```

# Resolving Imports

- Two functions are used *LoadLibrary* and *GetProcAddress*

```
HMODULE WINAPI LoadLibrary
(
    LPCTSTR lpFileName
);

FARPROC WINAPI GetProcAddress
(
    HMODULE hModule, LPCSTR lpProcName
);
```

# Imports fixed...

- The imported symbols appear mainly to relate to file access operations
- We can track the flow further in *OllyDbg* and conveniently comment the *API* call names in *IDA* for later reference
- We can also locate the table of addresses and write a small script to do it for us. More on that later
- After some tracing it is possible to see that the file being executed is opened, seeked and then its contents written to disk after yet another *XOR*'ing loop

# Main Deobfuscation Function

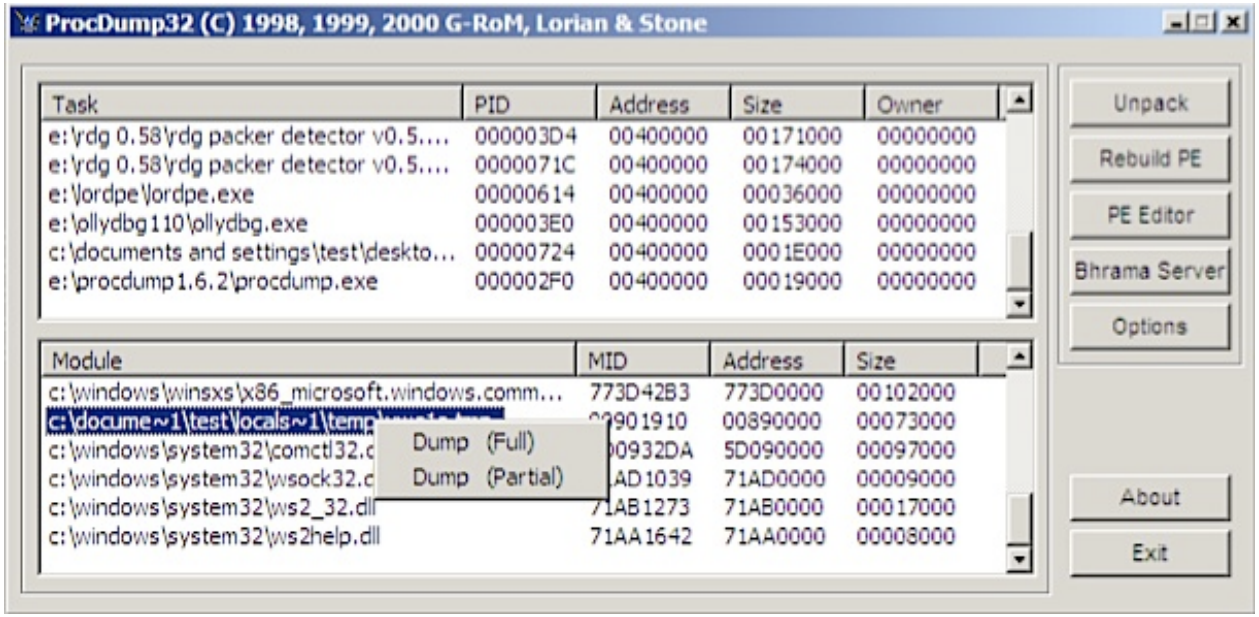
```
.yvs_:0041D58F 004          push    ebx
.yvs_:0041D590 008          mov     ebx, [ebp+xor_key]
.yvs_:0041D593 008          mov     edx, [ebp+buffer]
.yvs_:0041D596 008          mov     ecx, [ebp+buffer_size]
.yvs_:0041D599 008          test    ecx, ecx
.yvs_:0041D59B 008          jns     short start_loop
.yvs_:0041D59B
.yvs_:0041D59D 008          add     ecx, 3
.yvs_:0041D5A0
.yvs_:0041D5A0      start_loop:
.yvs_:0041D5A0 008          sar     ecx, 2
.yvs_:0041D5A3 008          xor     eax, eax
.yvs_:0041D5A5 008          cmp     ecx, eax
.yvs_:0041D5A7 008          jle     short exit
.yvs_:0041D5A7
.yvs_:0041D5A9
.yvs_:0041D5A9      XOR_loop:
.yvs_:0041D5A9 008          xor     [edx+eax*4], ebx
.yvs_:0041D5AC 008          inc     eax
.yvs_:0041D5AD 008          cmp     ecx, eax
.yvs_:0041D5AF 008          jg      short XOR_loop
.yvs_:0041D5AF
.yvs_:0041D5B1
.yvs_:0041D5B1      exit:
.yvs_:0041D5B1 008          pop     ebx
.yvs_:0041D5B2 004          pop     ebp
.yvs_:0041D5B3 000          retn    0Ch
```

# Now What?

- After the file is entirely dumped to disk, it is loaded as a *DLL* (Typical *LoadLibrary*, *GetProcAddress*, *call* sequence)
- If we take a look at the dumped file we can see that, yet again, it's packed. But this time it is a *DLL*
- The unpacking code is in the *DLLEntryPoint* (automatically called when doing invoking *LoadLibrary*)
- The main sample will resolve the *Initiate* procedure and call it

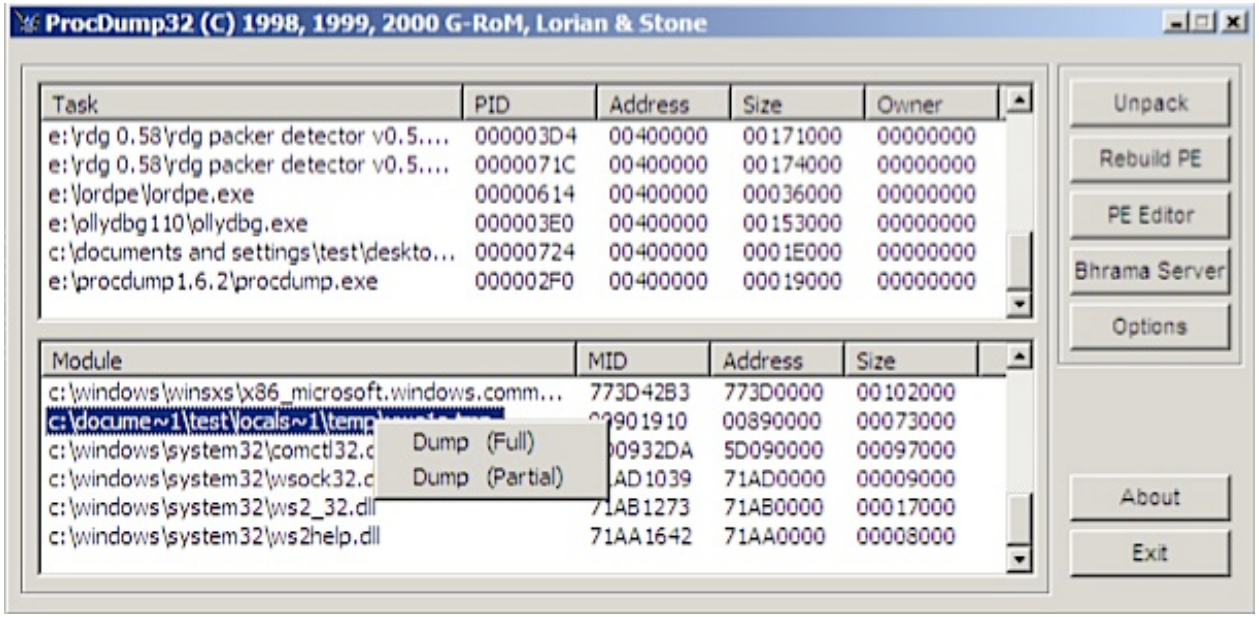
# Dumping a DLL

- If we want the unpacked contents of the binary we can use *ProcDump* to dump the *DLL* from memory



# Dumping a DLL

- If we want the unpacked contents of the binary we can use *ProcDump* to dump the *DLL* from memory
- Now we have the code but not the imports
  - And fixing the imports for a *DLL* is not trivial



# Fixing Imports

- If we followed all along with *OllyDbg*, it is possible to find the imported *API* calls becase *UPX* did the work of resolving them for us when the *DLL* was loaded

# Fixing Imports

- If we followed all along with *OllyDbg*, it is possible to find the imported *API* calls becase *UPX* did the work of resolving them for us when the *DLL* was loaded
- We will track the execution to find the table of imported symbols

# Fixing Imports

- If we followed all along with *OllyDbg*, it is possible to find the imported *API* calls becase *UPX* did the work of resolving them for us when the *DLL* was loaded
- We will track the execution to find the table of imported symbols
- We will then extract that info and use it in *IDA*

# Fixing Imports (1)

The execution is followed until the dropped *DLL* is loaded and its *Initiate* exported function called

```
0041D3BC   8B5D 0C          mov     ebx, dword ptr ss:[ebp+C]
0041D3BF   FF75 08          push    dword ptr ss:[ebp+8]
0041D3C2   FF13             call    dword ptr ds:[ebx]
0041D3C4   85C0             test    eax, eax
0041D3C6   74 1F            je      short 0af7b122.0041D3E7
0041D3C8   8B55 14          mov     edx, dword ptr ss:[ebp+14]
0041D3CB   81C2 F5000000    add     edx, 0F5
0041D3D1   52               push    edx
0041D3D2   50               push    eax
0041D3D3   FF53 04          call    dword ptr ds:[ebx+4]
0041D3D6   85C0             test    eax, eax
0041D3D8   74 0D            je      short 0af7b122.0041D3E7
0041D3DA   FF75 10          push    dword ptr ss:[ebp+10]
0041D3DD   FFD0             call    eax                          ywc1C.Initiate
0041D3DF   84C0             test    al, al
0041D3E1   74 04            je      short 0af7b122.0041D3E7
0041D3E3   B0 01            mov     al, 1
0041D3E5   EB 02            jmp     short 0af7b122.0041D3E9
0041D3E7   33C0             xor     eax, eax
eax=00891FBC (ywc1C.Initiate)
```

# Fixing Imports (2)

- We step into and try to find an instance where a call to an imported symbol is made
- *OllyDbg* will indicate this by displaying a *DLL.Function* name on the right side of the disassembly

```
008D562A  64:8921           mov     dword ptr fs:[ecx], esp
008D562D  803D BDA18E00 0(  cmp     byte ptr ds:[8EA1BD], 0
008D5634  74 0A             je      short ywc1C.008D5640
008D5636  68 ACA58E00       push    ywc1C.008EA5AC
008D563B  E8 C0D70000       call    ywc1C.008E2E00          jmp to ntdll.RtlEnterCritic
008D5640  83C3 07           add     ebx, 7
008D5643  83E3 FC           and     ebx, FFFFFFFC
008D5646  83FB 0C           cmp     ebx, 0C
```

# Fixing Imports (3)

Following the jump we find what we wanted. The table with all the symbols!

```
008E2DD0 -FF25 E8F28E00  jmp     dword ptr ds:[8EF2E8]   kernel32.CopyFileA
008E2DD6 -FF25 ECF28E00  jmp     dword ptr ds:[8EF2EC]   kernel32.CreateEventA
008E2DDC -FF25 F0F28E00  jmp     dword ptr ds:[8EF2F0]   kernel32.CreateFileA
008E2DE2 -FF25 F4F28E00  jmp     dword ptr ds:[8EF2F4]   kernel32.CreateMutexA
008E2DE8 -FF25 F8F28E00  jmp     dword ptr ds:[8EF2F8]   kernel32.CreateProcessA
008E2DEE -FF25 FCF28E00  jmp     dword ptr ds:[8EF2FC]   kernel32.CreateThread
008E2DF4 -FF25 00F38E00  jmp     dword ptr ds:[8EF300]   ntdll.RtlDeleteCriticalSection
008E2DFA -FF25 04F38E00  jmp     dword ptr ds:[8EF304]   kernel32.DeleteFileA
008E2E00 -FF25 08F38E00  jmp     dword ptr ds:[8EF308]   ntdll.RtlEnterCriticalSection
008E2E06 -FF25 0CF38E00  jmp     dword ptr ds:[8EF30C]   kernel32.EnumCalendarInfoA
008E2E0C -FF25 10F38E00  jmp     dword ptr ds:[8EF310]   kernel32.ExitProcess
008E2E12 -FF25 14F38E00  jmp     dword ptr ds:[8EF314]   kernel32.ExitThread
008E2E18 -FF25 18F38E00  jmp     dword ptr ds:[8EF318]   kernel32.FileTimeToDosDateTime
008E2E1E -FF25 1CF38E00  jmp     dword ptr ds:[8EF31C]   kernel32.FileTimeToLocalFileTime
008E2E24 -FF25 20F38E00  jmp     dword ptr ds:[8EF320]   kernel32.FindClose
008E2E2A -FF25 24F38E00  jmp     dword ptr ds:[8EF324]   kernel32.FindFirstFileA
008E2E30 -FF25 28F38E00  jmp     dword ptr ds:[8EF328]   kernel32.FindNextFileA
008E2E36 -FF25 2CF38E00  jmp     dword ptr ds:[8EF32C]   kernel32.FindResourceA
008E2E3C -FF25 30F38E00  jmp     dword ptr ds:[8EF330]   kernel32.FormatMessageA
008E2E42 -FF25 34F38E00  jmp     dword ptr ds:[8EF334]   kernel32.FreeEnvironmentStringsA
```

# Being Inventive (1)

- Now that we have found all the imported symbols for the *DLL*, we want to be able to port it over to our disassembly

# Being Inventive (1)

- Now that we have found all the imported symbols for the *DLL*, we want to be able to port it over to our disassembly
- One possible approach is to use *IDAPython* and some C&P'ing

# Being Inventive (2)

- Lets copy the entire list from *OllyDbg* and paste it in a decent editor with regex substitution support

# Being Inventive (2)

- Lets copy the entire list from *OllyDbg* and paste it in a decent editor with regex substitution support
- Now some regex magic...

```
([0-9ABCDEF]+).*jmp.*; (.*)
MakeName(\1, '\2')
```

# Being Inventive (3)

- ...and turn it into something sexier

```
MakeName(0x008E2DA0, 'advapi32.RegCloseKey')
MakeName(0x008E2DA6, 'advapi32.RegCreateKeyExA')
MakeName(0x008E2DAC, 'advapi32.RegFlushKey')
MakeName(0x008E2DB2, 'advapi32.RegOpenKeyExA')
...
```

# Being Inventive (3)

- ...and turn it into something sexier

```
MakeName(0x008E2DA0, 'advapi32.RegCloseKey')
MakeName(0x008E2DA6, 'advapi32.RegCreateKeyExA')
MakeName(0x008E2DAC, 'advapi32.RegFlushKey')
MakeName(0x008E2DB2, 'advapi32.RegOpenKeyExA')
...
```

- Copy the converted text into *IDAPython*'s notepad and execute it

# Being Inventive (3)

- ...and turn it into something sexier

```
MakeName(0x008E2DA0, 'advapi32.RegCloseKey')
MakeName(0x008E2DA6, 'advapi32.RegCreateKeyExA')
MakeName(0x008E2DAC, 'advapi32.RegFlushKey')
MakeName(0x008E2DB2, 'advapi32.RegOpenKeyExA')
...
```

- Copy the converted text into *IDAPython*'s notepad and execute it
- Now, we will have all the names in our dumped *DLL*

# Main Deobfuscation Function

```
.data:008E305E     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_WideCharToMultiByte. PRESS KEYPAD
.data:008E3064     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_WriteFile. PRESS KEYPAD "+" TO EX
.data:008E306A     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32__lopen. PRESS KEYPAD "+" TO EXPAN
.data:008E3070     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_lstrcatA. PRESS KEYPAD "+" TO EXP
.data:008E3076     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_lstrcmpiA. PRESS KEYPAD "+" TO EX
.data:008E307C     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_lstrcpyA. PRESS KEYPAD "+" TO EXP
.data:008E3082     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_lstrcpynA. PRESS KEYPAD "+" TO EX
.data:008E3088     ; [00000006 BYTES: COLLAPSED FUNCTION kernel32_lstrlenA. PRESS KEYPAD "+" TO EXP
.data:008E308E             align 10h
.data:008E3090     ; [00000006 BYTES: COLLAPSED FUNCTION mpr_WNetCloseEnum. PRESS KEYPAD "+" TO EXP
.data:008E3096     ; [00000006 BYTES: COLLAPSED FUNCTION mpr_WNetEnumResourceA. PRESS KEYPAD "+" TO
.data:008E309C     ; [00000006 BYTES: COLLAPSED FUNCTION mpr_WNetOpenEnumA. PRESS KEYPAD "+" TO EXP
.data:008E30A2             align 4
.data:008E30A4     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSAAsyncGetHostByName. PRESS KEYPAD
.data:008E30AA     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSAAsyncGetServByName. PRESS KEYPAD
.data:008E30B0     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSAAsyncSelect. PRESS KEYPAD "+" TO
.data:008E30B6     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSACancelAsyncRequest. PRESS KEYPAD
.data:008E30BC     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSACleanup. PRESS KEYPAD "+" TO EXP
.data:008E30C2     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSAGetLastError. PRESS KEYPAD "+" T
.data:008E30C8     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_WSAStartup. PRESS KEYPAD "+" TO EXP
.data:008E30CE     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_closesocket. PRESS KEYPAD "+" TO EX
.data:008E30D4     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_connect. PRESS KEYPAD "+" TO EXPAND
.data:008E30DA     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_gethostbyname. PRESS KEYPAD "+" TO
.data:008E30E0     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_getservbyname. PRESS KEYPAD "+" TO
.data:008E30E6     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_ntohs. PRESS KEYPAD "+" TO EXPAND]
.data:008E30EC     ; [00000006 BYTES: COLLAPSED FUNCTION WS2_32_inet_addr. PRESS KEYPAD "+" TO EXPA
```

# Is That All?

- You might think that the unpacking is now complete... unfortunately that's only partially true

# Is That All?

- You might think that the unpacking is now complete... unfortunately that's only partially true
- If we follow the flow further we will eventually find that it does not exit

# Is That All?

- You might think that the unpacking is now complete... unfortunately that's only partially true
- If we follow the flow further we will eventually find that it does not exit
- Instead we find ourselves in *UPX* unpacking code...

# Is That All?

- You might think that the unpacking is now complete... unfortunately that's only partially true
- If we follow the flow further we will eventually find that it does not exit
- Instead we find ourselves in *UPX* unpacking code...
- ... unpacking again?

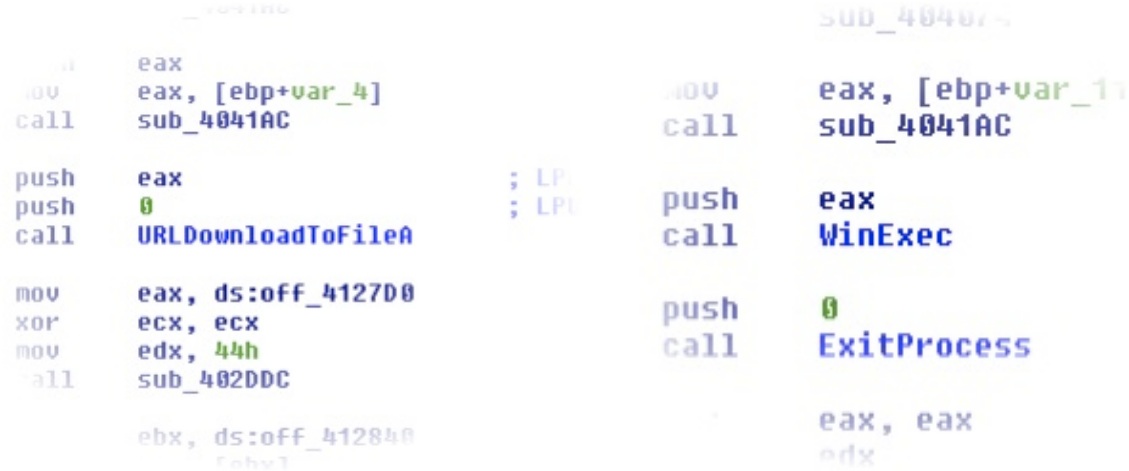
# The Follow Up

- *UPX* is pretty simple to deal with
- We just move to the end of the unpacking code and look at the jump
- It's indeed *UPX* and after the unpacking is done execution will continue within the binary
- If we breakboint the jump and let it unpack we can dump the new executable

```
text:0041BB09            mov     [ebx], eax
text:0041BB0B            add     ebx, 4
text:0041BB0E            jmp     short loc_41BAF1
text:0041BB0E
text:0041BB10    ; ---------------------------------------
text:0041BB10
text:0041BB10    loc_41BB10:
text:0041BB10            call    dword ptr [esi+1B9FCh]
text:0041BB10
text:0041BB16
text:0041BB16    loc_41BB16:
text:0041BB16            pusha
text:0041BB17            jmp     start
text:0041BB17
```

# The 2nd Unpacked Component

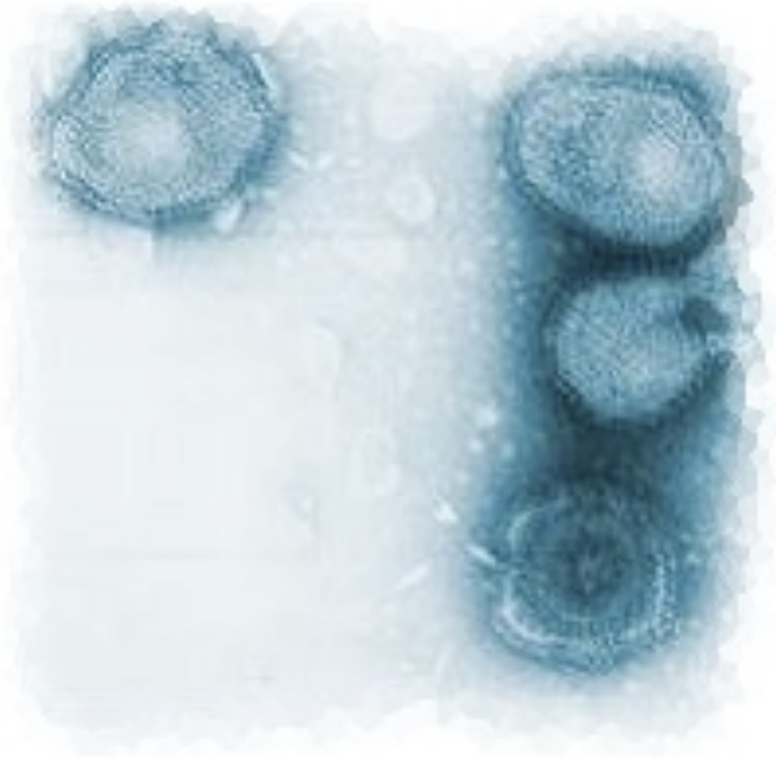
After some inspection it's possible to find that this new element is certainly malicious (as is the *DLL*)



# Final Outcome

- What's the explanation?
- The second sample is a Bot, specifically *Backdoor.Win32.Gobot.w* or *W32.Gobot.A*, depending on who you ask
- The first sample was *Parite*
- *Parite* is a polymorphic file infector and we just saw it in action

# The Target Revealed



- Kaskpersky: *Virus.Win32.Parite.b*
- McAfee: *W32/Pate.b*
- Symantec: *W32.Pinfi*
- Trend Micro: *PE_PARITE.A*

- SHA1: *f30ad924a0d35b13d2057b1bb6305ad5a8ac8fa2*
- MD5: *0af7b122bb722fb679c97fdb8cf85d23*

# Outline

14 Analysis

15 IDA Python
- Overview
- Examples
- Exercises

16 PEFile and PyDasm

17 PaiMei

# What is it?

- IDAPython extends IDA with Python
- The whole IDC function set and the plugin API are available
- Python is an incredibly powerful scripting language
- The blend allows for extreme flexibility when writing custom analysis tools
- *ida2sql*, for instance, was written using IDAPython and exports nearly the whole IDB to an SQL database for advanced and automated data-mining

# Resources

- First introduced in the paper "Digital Genome Mapping - Advanced Binary Malware Analysis"

[Digital Genome Mapping]

- Available for download from:

http://d-dome.net/idapython

- Good article on OpenRCE: "Introduction to IDAPython"

[Introduction to IDAPython]

- ida2sql is available at:

http://dkbza.org/ida2sql.html

# Reading a string

Reading and printing a string from the current cursor location:

```
ea = ScreenEA()

s = ""
while True:
    b = Byte(ea)
    if b == 0:
        break
    s += chr(b)
    ea += 1

print s
```

# XOR-ing a string

```
ea = ScreenEA()

s = ""
while True:
    b = Byte(ea)
    if b == 0:
        break
    b = b^0xff
    PatchByte(ea, b)
    ea += 1

print s
```

# Enumerating Segments

```
for seg in Segments():
    print '%08x-%08x' % (seg, SegEnd(seg))
```

# Enumerating Functions

```
for f in Functions(start_addr, end_addr):
    print '%s: %08x-%08x' % (Name(f), f, FindFuncEnd(f))
```

# Enumerating Heads

```
for h in Heads(ScreenEA(), ScreenEA()+100):
    print hex(h)
```

# Collecting Function Chunks

- Some compilers generate optimized code containing features such as
  - Functions share basic blocks
  - The layout of *basic blocks* within the binary is affected by the likelihood of them being run
- *IDA* handles the resulting "fragmented" functions by collecting all the parts in chunks
- The functions *func_tail_iterator_t()*, *func_iter.main()*, *func_iter.chunk()*, *func_iter.next()* allow to retrieve them

# Collecting Function Chunks (2)

```
function_chunks = []
#Get the tail iterator
func_iter = idaapi.func_tail_iterator_t(idaapi.get_func(ea))

# While the iterator's status is valid
status = func_iter.main()
while status:
 # Get the chunk
 chunk = func_iter.chunk()
 # Store its start and ending address as a tuple
 function_chunks.append((chunk.startEA, chunk.endEA))
 # Get the last status
 status = func_iter.next()
```

# Mydoom Backdoor DLL Extraction

- Locate the DLL bytes in memory
- Locate the DLL decoder
- Write an IDA Python script to decode and save the DLL to disk

# Mydoom String De-obfuscation

- Locate some obfuscated strings
- Determine the obfuscation method
- Write an IDA Python script to restore the original strings

# Outline

14 Analysis

15 IDA Python

16 PEFile and PyDasm
- Overview
- pefile
- pydasm
- Exercises

17 PaiMei

# Purpose

- Python libraries that allow for scripted Windows binary inspection
- These tools allow for great automation of various analysis tasks
- *pefile* is a PE Format Python parsing module
- *pydasm* is an *x86* Python disassembler module
  - pydasm is used by PaiMei (we may talk about this later)

# pefile

http://dkbza.org/pefile.html

- *pefile* is a cross-platform pure Python module intended for handling PE Files
- Actively maintained by Ero
- It should be able to process any file that can be open by the Windows loader
- Usage requires some basic understanding of the layout of a PE file
- You can use *pefile* to explore nearly every single feature of a PE file

# Loading a PE file

- Loading a file is as easy as creating an instance of the PE class with a path to the PE file

```
pe = pefile.PE('path/to/file')
```

- Alternatively, you can pass a PE file as raw data

```
pe = pefile.PE(data=python_string)
```

# Inspecting the Headers

```
>>> import pefile

>>> pe = pefile.PE('notepad.exe')
>>> hex(pe.OPTIONAL_HEADER.ImageBase)
['0x1000000L']

>>> hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint)
['0x6AE0L']

>>> hex(pe.OPTIONAL_HEADER.NumberOfRvaAndSizes)
['0x10L']

>>> hex(pe.FILE_HEADER.NumberOfSections)
['0x3']
```

# Inspecting the Sections

```
>>> for section in pe.sections:
...  print (section.Name,
       hex(section.VirtualAddress),
       hex(section.Misc_VirtualSize),
       section.SizeOfRawData )
...
('.text', '0x1000L', '0x6D72L', 28160L)
('.data', '0x8000L', '0x1BA8L', 1536L)
('.rsrc', '0xA000L', '0x8948L', 35328L)
```

# Think it's Packed?

```
import math

def H(data):
    if not data:
        return 0

    entropy = 0
    for x in range(256):
        p_x = float(data.count(chr(x)))/len(data)
        if p_x > 0:
            entropy += - p_x*math.log(p_x, 2)

    return entropy
```

# Think it's Packed? (Unpacked)

```
>>> for section in pe.sections:
...  print section.Name, H(section.data)
...
.text 6.28370964662
.data 1.39795676336
.rsrc 5.40687515641
```

# Think it's Packed? (ASPack)

```
>>> pe2 = pefile.PE('notepad-aspack.exe')
>>> for section in pe2.sections:
...  print section.Name, H(section.data)
...
.text 7.98363149339
.data 4.68226874255
.rsrc 6.09026175185
.aspack 5.90609875421
.adata 0
```

# Think it's Packed? (UPX)

```
>>> pe3 = pefile.PE('notepad-upx.exe')
>>> for section in pe3.sections:
...  print section.Name, H(section.data)
...
UPX0 0
UPX1 7.83028313969
.rsrc 5.59212256596
```

# Imports

```
>>> for entry in pe.DIRECTORY_ENTRY_IMPORT:
...   print entry.dll
...   for imp in entry.imports:
...     print '\t', hex(imp.address), imp.name
...
comdlg32.dll
        0x10012A0L PageSetupDlgW
        0x10012A4L FindTextW
        0x10012A8L PrintDlgExW
[snip]
SHELL32.dll
        0x1001154L DragFinish
        0x1001158L DragQueryFileW
```

# pydasm

http://dkbza.org/pydasm.html

- *pydasm* is a cross-platform Python module wrapping jt's libdasm
- Combined with *pefile*, you get a good base for developing a mini-IDA wannabe

# Disassembling

```
>>> import pydasm
>>> i = pydasm.get_instruction('\x90', pydasm.MODE_32)
>>> pydasm.get_instruction_string(
        i, pydasm.FORMAT_INTEL, 0)

['nop ']

>>> i = pydasm.get_instruction(
        '\x8B\x04\xBD\xE8\x90\x00\x01', pydasm.MODE_32)
>>> pydasm.get_instruction_string(
        i, pydasm.FORMAT_INTEL, 0)

['mov eax,[edi*4+0x10090e8]']
```

# The Instruction Object

```
>>> pprint.pprint(dir(i))
['__doc__','__module__',
 'dispbytes', 'extindex',
 'flags', 'fpuindex',
 'immbytes', 'length',
 'mode','modrm',
 'op1', 'op2', 'op3',
 'opcode', 'ptr',
 'sectionbytes', 'sib',
 'type']
```

# The Operand Object

```
>>> pprint.pprint(dir(i.op1))
['__doc__', '__module__',
 'basereg', 'dispbytes',
 'displacement', 'dispoffset',
 'flags', 'immbytes',
 'immediate', 'immoffset',
 'indexreg', 'reg',
 'scale', 'section',
 'sectionbytes', 'type']
```

# pefile+pydasm

```
>>> ep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
>>> ep_ava = ep+pe.OPTIONAL_HEADER.ImageBase
>>> data = pe.get_memory_mapped_image()[ep:ep+100]
>>> offset = 0
>>> while offset < len(data):
...     i = pydasm.get_instruction(
              data[offset:], pydasm.MODE_32)
...     print pydasm.get_instruction_string(
              i, pydasm.FORMAT_INTEL, ep_ava+offset)
...     offset += i.length
```

# pefile+pydasm (Output)

```
push byte 0x70
push dword 0x1001888
call 0x1006ca8
xor ebx,ebx
push ebx
mov edi,[0x100114c]
call edi
cmp word [eax],0x5a4d
jnz 0x1006b1d
mov ecx,[eax+0x3c]
add ecx,eax
cmp dword [ecx],0x4550
jnz 0x1006b1d
movzx eax,[ecx+0x18
```

# pefile Exercises

- Load a *DLL* and print the imported and exported symbols
- Load a *PE* file and check whether the entry point is in the last section and whether such section is smaller than the others
- Load a *PE* file and dump the first 256 bytes, starting from the entry point, to disk
  - Think of a way of using this metadata as a packer or executable signature

# pydasm Exercises

- Building on the examples in the slides, create a disassembler that follows references and try to disassemble as far as you can
- Create a histogram from the mnemonics of the disassembled code
- Think of ways to use the histogram (or more advanced statistical techniques) to build a classifier based on entry point code

# Outline

14 Analysis

15 IDA Python

16 PEFile and PyDasm

17 PaiMei
- Overview
- Command Line Tools
- GUI and Tools
- Exercises

# The Man



# What is it?

- It's a win32 reverse engineering framework
- Written entirely in Python
- Think of PaiMei as an RE swiss army knife
- Already proven effective for a number of tasks
  - Fuzzer assistance
  - Code coverage tracking
  - Data flow tracking
  - A beta tester used it to solve the T2'06 RE challenge

**My hopes and dreams**

That with community support and contributions, PaiMei can do for RE dev what Metasploit does for exploit dev

# Motivation: Rapid Development

- Avoid the learning / re-learning curve of various SDKs
- Develop in a higher level language
  - Easy management of arbitrary data structures
  - Less code
  - Less debugging of the actual tool
- Build data representation into the framework, as opposed to an after-thought
  - Of course, coming from Pedram, this translates into graphing

# Motivation: Homogenous Environment

- Making tools and languages talk to one another is tedious
  - IDA vs. OllyDbg vs. MySQL
  - C/C++ vs. Python
- Centralized tool creation vs. the old school:
  - Launch debugger
  - Run plug-in
  - Save output to disk
  - Parse output through Perl into IDC
  - Import into IDA

# Core Components

## PyDbg

A pure Python win32 debugger abstraction class

## pGRAPH

An abstraction library for representing graphs as a collection of nodes, edges and clusters

## PIDA

A binary abstraction library, built on top of pGRAPH, for representing binaries as a collection of functions, basic blocks and instructions

# Extended Components

## Utilities

A set of abstraction classes for accomplishing various repetitive tasks

## Console

A pluggable WxPython GUI for quickly and efficiently rolling out your own sexy RE tools

## Scripts

Individual scripts built on the framework

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

## Example API

```
enumerate_processes()
enumerate_modules()
enumerate_threads()
attach()
load()
suspend_thread()
resume_thread()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

## Example API

```
bp_set_hw()
bp_set()
bp_set_mem()
bp_del_hw()
bp_del()
bp_del_mem()
bp_is_ours_mem()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

## Example API

```
read()
write()
virtual_alloc()
virtual_query()
smart_dereference()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

**Example API**

```
process_snapshot()
process_restore()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

**Example API**

```
stack_unwind()
seh_unwind()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

**Example API**

```
set_callback()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

**Example API**

```
disasm()
disasm_around()
```

# PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

## Example API

```
flip_endian()
flip_endian_dword()
func_resolve()
hex_dump()
to_binary()
to_decimal()
```

# PyDbg: Example

## Abstracted interface allows for painless development

```
from pydbg import *
from pydbg.defines import *

def handler_breakpoint (pydbg):
    # ignore the first windows driven breakpoint.
    if pydbg.first_breakpoint:
        return DBG_CONTINUE

    print "ws2_32.recv() called from thread %d @%08x" % \
        pydbg.dbg.dwThreadId,
        pydbg.exception_address)

    return DBG_CONTINUE

dbg = pydbg()

# register a breakpoint handler function.
dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)
dbg.attach(XXXXX)

recv = dbg.func_resolve("ws2_32", "recv")
dbg.bp_set(recv)

pydbg.run()
```

# PyDbg: Random Idea Implementation

## The problem

I want to solve the F-Secure T2'06 challenge ... but I'm lazy.

1. Open the binary in IDA
2. Locate password read and process exit
3. Set breakpoints on both
4. The first time a password is read, snapshot
5. When the exit is reached, restore
6. Read the buffer address off the stack
7. Change the password
8. Continue

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

## Example API

```
add_node()
add_edge()
del_node()
del_edge()
```

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

## Example API

```
find_node()
find_edge()
edges_from()
edges_to()
```

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

## Example API

```
graph_cat()
graph_sub()
graph_up()
graph_down()
graph_intersect()
graph_proximity()
```

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

**Example API**

```
render_graph_graphviz()
render_graph_gml()
render_graph_udraw()
```

# pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Why do we need this library?

# PIDA

- Extends from pGRAPH to represent binaries as a graph of graphs
- PIDA databases are propogated by an IDA Python script pida_dump.py
  - This is important, I will show it to you in a second
- The database is serialized to a zlib compressed .pida database
- PIDA enumerates basic blocks and discovers RPC routines
- The .pida database can later be loaded independent of IDA
- All the aforementioned graph functionality is available for (ab)use
- Quick demo

# PIDA: Contrived Example

## Again, abstracted interface allows for painless development

```
import pida import *

module = pida.load("some_file.pida")

# render a function graph in uDraw format for the entire module.
fh = open("graphs/functions.udg", "w+")
fh.write(module.render_graph_udraw())
fh.close()

# step through each function in the module:
for function in module.functions.values():
    # if we found the function we are interested in:
    if function.name == "some_function":
        # step through each basic block in the function.
        for bb in function.basic_blocks.values():
            print "\t%08x - %08x" % (bb.ea_start, bb.ea_end)
            # print each instruction in each basic block.
            for ins in bb.instructions.values():
                print "\t\t%s" % ins.disasm

        # render a GML graph of this function.
        fh = open("graphs/function.gml", "w+")
        fh.write(function.render_graph_gml())
        fh.close()
```

# PIDA: Contrived Example

## ...Continued

```
# if we found the second function we are interested in.
if function.ea_start == 0xdeadbeef:

    # render a uDraw format proximity graph.
    fh = open("graphs/proximity.udg", "w+")

    # look 3 levels up and 2 levels down.
    prox_graph = module.graph_proximity(function.id, 3, 2)
    fh.write(prox_graph.render_graph_udraw())
    fh.close()
```

Together, PIDA and PyDbg offer a powerful combination for building a variety of tools. Consider for example the ease of re-creating Process Stalker on top of this platform.

# PIDA: Real World Example

Locate all functions within a binary that open a file and display the execution path from the entry point to the call of interest...

```
# for each function in the module
for function in module.functions.values():
    # create a downgraph from the current routine and locate the calls to [Open|Create]File[A|W]
    downgraph = module.graph_down(function.ea_start, -1)
    matches   = [node for node in downgraph.nodes.values() if re.match(".*(create|open)file.*", \
                node.name, re.I)]
    upgraph   = pgraph.graph()

    # for each matching node create a temporary upgraph and add it to the parent upgraph.
    for node in matches:
        tmp_graph = module.graph_up(node.ea_start, -1)
        upgraph.graph_cat(tmp_graph)

    # write the intersection of the down graph from the current function and the upgraph from
    # the discovered interested nodes to disk in gml format.
    downgraph.graph_intersect(upgraph)

    if len(downgraph.nodes):
        fh = open("%s.gml" % function.name, "w+")
        fh.write(downgraph.render_graph_gml())
        fh.close()
```

# Utilities

- Classes for further abstracting frequently repeated functionality:
  - Code Coverage
  - Crash Binning
  - Process Stalker
  - uDraw Connector

# Utility: Code Coverage

- Simple container for storing code coverage data
- Supports persistant storage to MySQL or serialized file
- You can use this class to keep track of where you have been
- Examples:
  - Process Stalker
  - Individual fuzzer test case tracking

# Utility: Crash Binning

- Simple container for categorizing and storing "crashes"
- Stored crashes are organized in bins by exception address
- The in-house version of this class goes one step further by categorizing on path as well (stack unwind)
- The `crash_synopsis()` routine generates detailed crash reports:
  - Exception address, type and violation address
  - Offending thread ID and context
  - Disassembly around the exception address
  - Stack and SEH unwind information
- This class is extremely useful for fuzzer monitoring
  - *ex:* 250 crashes vs. 248 crashes at x and 2 crashes at y
- *Note to Pedram*: Mention the Excel file format exploit "fuzzer"

# Utility: Process Stalker

- Abstracted interface to Process Stalking style code coverage
- Currently only being used by the pstalker GUI module
- A command line interface can be easily built
- The class handles all the basics:
  - Re-basing and setting breakpoints in the main module
  - Re-basing and setting breakpoints in loaded libraries
  - Recording, with or without context data, hit breakpoints
  - Monitoring for access violations
  - Exporting (through the code coverage class) to MySQL

# Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

# Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

**Example API**

```
graph_new()
graph_update()
```

# Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

**Example API**

```
focus_node()
layout_improve_all()
scale()
open_survey_view()
```

# Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

## Example API

```
change_element_color()
window_background()
window_status()
window_title()
```

# Utility: uDraw(Graph) Connector

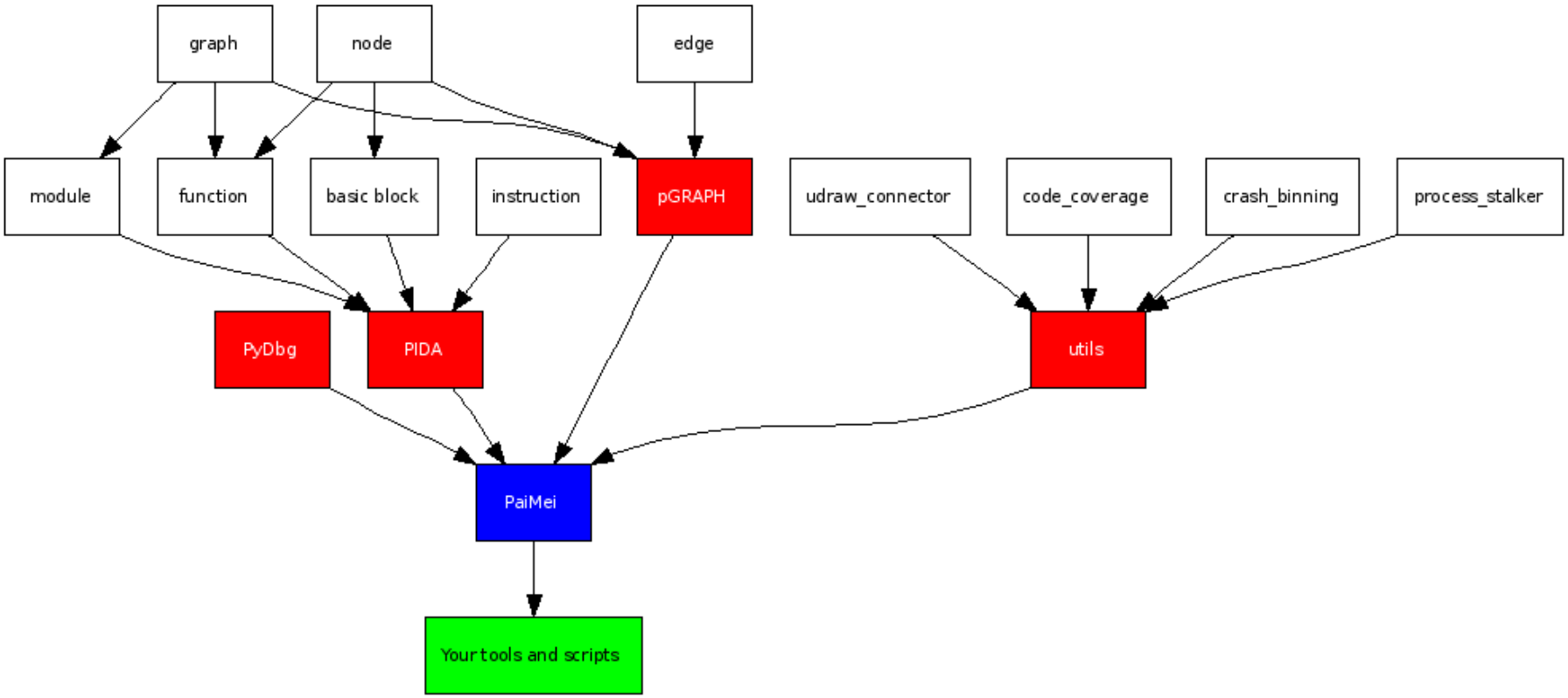
Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

**Example API**

```
set_command_handler()
```

# How it All Ties Together



# GUI Overview

- Some complex tools are not suitable for the command line
- The PaiMei console provides a base for new GUI modules
- Development for the framework is well documented (I think)
- Allows you to focus your efforts on the tool
- GUI modules follow the naming convention PAIMEIxxxx

# GUI General layout

- Modules are independent of one another
  - Though you can push / pull data between them
- Each module represented by a notebook icon
- Entire right pane is controlled by the module
- Left status bar displays console wide messages
- Right status bar is owned by the current module
- *Connections* menu establishes connectivity to MySQL and uDraw
- *Advanced* menu exposes log window clearing and CLI
- The CLI (Command Line Interface) is a full Python interpreter and allows you to interact with any portion of the console.
  - Explicitly documented module member variables are listed on the right-hand side of the CLI

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

```
procedure("pedram", 26)
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions

```
procedure("pedram", 26)
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list

```
procedure("pedram", 26)
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. `PUSH` numeric arguments directly

```
procedure("pedram", 26)
```

```
PUSH 26
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. PUSH numeric arguments directly
4. Allocate space for string arguments and PUSH address

```
procedure("pedram", 26)
```

```
PUSH 26
PUSH 0x12345678
```

```
0x12345678: "pedram"
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. `PUSH` numeric arguments directly
4. Allocate space for string arguments and `PUSH` address
5. Write the `CALL` instruction

```
procedure("pedram", 26)
```

```
PUSH 26
PUSH 0x12345678
CALL procedure
```

```
0x12345678: "pedram"
```

# DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. `PUSH` numeric arguments directly
4. Allocate space for string arguments and `PUSH` address
5. Write the `CALL` instruction
6. Write an `INT 3` to regain control

```
procedure("pedram", 26)
```

```
PUSH 26
PUSH 0x12345678
CALL procedure
INT 3
```

```
0x12345678: "pedram"
```

# DPC: Usage

- Once attached you are given a command prompt
- Any Python statement is valid
- `dbg` references current PyDbg instance
- Convenience wrappers exist for memory manipulaton
  - `alloc(), free(), free_all(), show_all()`
- Assigned variables are not persistant!
  - Use `glob` for that
  - `print glob` to display what you have assigned
- `dpc(procedure, *args, **kwargs)`
  - kwargs are for fast call support
- Took me less than 30 minutes to write the 1st version of this tool

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| | | |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| 48-53 | 53 | 6 |
| | | |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| 48-53 | 53 | 6 |
| 54-59 | 59 | 6 |
| | | |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| 48-53 | 53 | 6 |
| 54-59 | 59 | 6 |
| 60-61 | 61 | 2 |
| | | |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| 48-53 | 53 | 6 |
| 54-59 | 59 | 6 |
| 60-61 | 61 | 2 |
| 62-67 | 67 | 6 |
| | | |

# DPC: Example One

## Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

| Input Range | Return | Δ |
|---|---|---|
| 25-29 | 29 | 6 |
| 30-31 | 31 | 2 |
| 32-37 | 37 | 6 |
| 38-41 | 41 | 4 |
| 42-43 | 43 | 2 |
| 44-47 | 47 | 4 |
| 48-53 | 53 | 6 |
| 54-59 | 59 | 6 |
| 60-61 | 61 | 2 |
| 62-67 | 67 | 6 |
| 68-71 | 71 | 4 |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| paimei | apple | 50 | 0x00000001 |
| | | | |
| | | | |
| | | | |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| paimei | apple | 50 | 0x00000001 |
| paimei | pear | 69 | 0xFFFFFFFF |
| | | | |
| | | | |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| paimei | apple | 50 | 0x00000001 |
| paimei | pear | 69 | 0xFFFFFFFF |
| pai | paimei | 666 | 0xFFFFFFFF |
| | | | |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| paimei | apple | 50 | 0x00000001 |
| paimei | pear | 69 | 0xFFFFFFFF |
| pai | paimei | 666 | 0xFFFFFFFF |
| paimei | paimei | 31337 | 0x00000000 |
| | | | |

# DPC: Example Two

Here's another one...

| Arg 1 | Arg 2 | Arg 3 | Return |
|---|---|---|---|
| paimei | eyebrow | 25 | 0x00000001 |
| paimei | apple | 50 | 0x00000001 |
| paimei | pear | 69 | 0xFFFFFFFF |
| pai | paimei | 666 | 0xFFFFFFFF |
| paimei | paimei | 31337 | 0x00000000 |
| pai | paimei | 3 | 0x00000000 |

# DPC: (Quick) Live Demo



# OllyDbg Connector

- PyDbg is designed for mostly non-interactive functionality
- This two-part tool adds live graphing functionality to OllyDbg
- Part 1: Receiver
  - Socket server for OllyDbg
  - Receives module name, base address and offset from plug-in
  - Socket client to uDraw(Graph)
  - Loads specified PIDA database and generates graph
- Part 2: Connector
  - Registers hotkeys for transmitting location to receiver
  - , Step into and xmit current location
  - . Step over and xmit current location
  - / Xmit current location

# OllyDbg Connector: Live Demo



# Stack Integrity Monitor

- Tracking down the source of a complete stack smash is tedious
- I had to do a bunch one day so I wrote a 150 line PyDbg script
- How it works:
  - Instantiate a debugger object and attach to the target program
  - Set a breakpoint where we want the trace to start, this can be as simple as setting a break on recv()
  - Once the breakpoint is hit, set the active thread to single step
  - When a CALL instruction is reached, copy the stack and return addresses to an internal "mirror" list
  - When a RET instruction is reached, walk through the "mirror" list and verify that the values match the actual stack
  - When the last saved return address is reached, pop it off the internal "mirror" list

# Stack Integrity Monitor: Before

```
[INVALID]:41414141 Unable to disassemble at 41414141 from thread 568 caused
          access violation when attempting to read from 0x41414141

CONTEXT DUMP
  EIP: 41414141 Unable to disassemble at 41414141
  EAX: 00000001 (         1) -> N/A
  EBX: 0259eedc (  39448284) -> AAAAAAAAAAAAAA....AAAAAAAAAAAAAAAA (stack)
  ECX: 00000000 (         0) -> N/A
  EDX: ffffffff (4294967295) -> N/A
  EDI: 00000000 (         0) -> N/A
  ESI: 0259f102 (  39448834) -> AAAAAAAAAAAAAA....AAAAAAAAAAAAAAAA (stack)
  EBP: 00000001 (         1) -> N/A
  ESP: 0259e2d4 (  39445204) -> AAAAAAAAAAAAAA....AAAAAAAAAAAAAAAA (stack)
  +00: 41414141 (1094795585) -> N/A
  +04: 41414141 (1094795585) -> N/A
  +08: 41414141 (1094795585) -> N/A
  +0c: 41414141 (1094795585) -> N/A
  +10: 41414141 (1094795585) -> N/A
  +14: 41414141 (1094795585) -> N/A

disasm around:
        0x41414141 Unable to disassemble
```

# Stack Integrity Monitor: After

```
0259fc24: TmRpcSrv.dll.65741721
0259e7b4: StRpcSrv.dll.65671190
0259e7a8: Eng50.dll.61181d8c
0259e790: Eng50.dll.611819a0
0259e564: Eng50.dll.61181a50
0259e2d0: Eng50.dll.61190fa4 --> 41414141
0259e03c: Eng50.dll.61190fd2

STACK INTEGRITY VIOLATON AT: Eng50.dll.61194b8e
analysis took 35 seconds
```

Examining the vicinity of the last return address in the list, we find:

```
61190FC7 lea edx, [esp+288h+szShortPath]
61190FCB push esi
61190FCC push edx
61190FCD call _wcscpy
61190FD2 add esp, 8
```

# Proc Peek

- This two-part tool was designed for discovering *low hanging fruit* vulnerabilities
  - Which, believe it or not, is quite effective
- The first half of the tool is a static reconnaissance phase
  - *proc_peek_recon.py*
- The second half of the tool is a run-time analysis phase
  - *proc_peek.py* and *PAIMEIpeek*

**General philosophy**

With minimal effort, generate a list of locations that can be easily monitored and *checked off*. This approach is great for 1st phase auditing and can be handed off to an intern.

# Proc Peek: proc_peek_recon.py

- IDA Python script
- Looks for *interesting* locations, or peek points
  - Inline `memcpy()` and `strcpy() routines`
  - Calls to API that accept format string tokens
    - Ignoring ones that do not contain `%s`
  - Calls to potentially *dangerous* API such as `strcat()`, `strcpy()`, etc...
- Discovered peek points are written to a file or database

# Proc Peek: proc_peek.py

- PyDbg based script (a bit dated)
- Attach to the target process
- Set breakpoints on each peek point
- When a breakpoint is hit:
  - Present the user with relevant information
  - Prompt for action: *ignore*, *continue*, *make notes*
- Supports automated keyword searching (Hoglund: *Boron tagging*)
- Also features Winsock recv() tracking

# Proc Peek: PAIMEIpeek

- Also PyDbg based, the GUI version of the last script
- Less of an interactive tool, more of a data sampling utility
- Again, set breakpoints on each peek point
- When a breakpoint is hit:
  - Record the time and contextual information to the database
  - Search the context for user-specified keywords

# PAIMEIpeek: Demo



# PAIMEIdocs

- HTML documentation browser
- Use the control bar at the top to load general or developer specific documentation
- Not all that exciting

# PAIMEIexplore

- The *hello world* of the console
- The in-house version has a bit more functionality
- To use:
  - Load a PIDA database
  - Double click the PIDA database
  - Browse through the explorer tree
  - Select a function to display disassembly
  - Connect to uDraw
  - Graph a function through the right-click context menu

# PAIMEIfilefuzz

- File fuzzing and exception monitoring tool built on PaiMei
- Developed by Cody Pierce
- Loads a target file
- Generates mutations based at specified offset / range, variable length and byte values
  - More advanced features include, additive mutations
- Supports mid-session pause and resume
- Features predictable completion time and run-time statistics
- In-house experimental features:
  - Auto file discovery
  - Auto handler discovery
  - Auto fuzz
  - ie: Give it a laptop and go

# PAIMEIdiff

- A binary diffing tool built on PaiMei
- Being developed by Peter Silberman
- Still an early beta and not currently distributed
- Heuristic based diffing engine (like Zynamics BinDiff)
- The goal of the module is to allow the user to deeply control the diffing algorithm
- Customized algorithms can be saved for later use
- This will likely lead to job specific sets:
  - Malware analysis
  - Generic patch diffing
  - Microsoft patch diffing
  - Etc...

# PAIMEIdiff: Supported Heuristics

Some of these were gleaned from the Zynamics Security white papers:

- API calls
- Argument and variable sizes
- Constants
- Control flow
- CRC
- Name
- NECI (graph heuristics)
- Recursive calls
- Size
- Small Prime Product (SPP)
- "Smart" MD5
- Stack frame
- String references

# PAIMEIpstalker

- Code coverage recording tool
- This is the "next generation" of Process Stalker
- All metadata is stored to MySQL
- Three step approach:
  - Setup data sources
  - Capture code coverage data
  - Explore captured data
- Filtering support allows you to pinpoint interesting code locations

# Basic Exercises

- Write a PIDA script to locate utility functions
- Use DPC to decipher one of the located utility functions
- Write a PIDA script to find all paths to a specific API
- Write a PyDbg script to sample data crossing a specific point
- For a more in-depth project, see the next two slides

# Malware Profiler

- I will never get around to this, so someone else do it
- Post unpacking / PIDA conversion, static analysis tool
- Step through the call chains within the binary
  - Mark common sequences with a high level label
  - Automatically extract information such as mutex name, startup keys, etc..
- Can help narrow analysis areas, ie:
  - Glean what you can through live analysis
  - Automatically tag and command statically recognized code sequences
  - What you are left with will be the more interesting sections
- The tool should be driven by XML configuration files (next slide)

# Malware Profiler: Continued

## Theorized example XML

```
<classification name="SMTP Engine">
    <API name="htons">
        <argument index=1>25</argument>
    </API>
</classification>
<classification name="Address Harvesting">
    <API name="FindFirstFile()"></API>
    <API name="FindNextFile()"></API>
    <API name="MapViewOfFile()"></API>
    <string match="regex">
        [^@]+@[^\.]+\.com
    </string>
</classification>
<classification name="Startup Entry">
    <API name="RegCreateKeyEx">
        <argument index=1>
            HKEY_LOCAL_MACHINE
        </argument
        <argument index=2>
            <string match="regex">\run|\runonce</string>
        </argument>
    </API>
</classification>
```

# Part V

# Appendix

# Outline

18 Appendix
- References
- Slide Count

# 18 Appendix

- References
- Slide Count

# References I


Alexander Sotirov, Determina Security Research
Reverse Engineering Microsoft Binaries
CansecWest 2006


Communications of the ACM vol. 17 no. 7, July 1974


Open Reverse Code Engineering http://www.openrce.org


Microsoft Portable Executable and Common Object File Format Specification
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF


Portable Executable File Format  A Reverse Engineer View
http://www.CodeBreakers-Journal.com

# References II


Peter Ferrie, Zmist Opportunities, Virus Bulletin March 2001; http://pferrie.tripod.com/vb/zmist.pdf


Peter Ferrie, Striking Similarities, Virus Bulletin May 2002; http://pferrie.tripod.com/vb/simile.pdf


Ero Carrera & Gergely Erdelyi, Digital Genome Mapping - Advanced Binary Malware Analysis, Virus Bulletin Conference 2004; http://dkbza.org/data/carrera_erdelyi_VB2004.pdf


Ero Carrera, Introduction to IDAPython, https://www.openrce.org/articles/full_view/11


Nicolas Brulez, Scan of the Month 33: Anti Reverse Engineering Uncovered, http://www.honeynet.org/scans/scan33/nico/

# References III


Tricky Relocations, Peter Szor, Virus Bulletin, April 2001, page 8
http://peterszor.com/resurrel.pdf


Locreate: An Anagram for Relocate
http://uninformed.org/?v=6&a=3&t=sumry


The Viral Darwinism of W32.Evol
https://www.openrce.org/articles/full_view/27


Tiny PE, solareclipse;
http://www.phreedom.org/solar/code/tinype/


Andrs et al. Methods for Virtual Machine Detection. (2006)


Liston et al. On the Cutting Edge: Thwarting Virtual Machine Detection. (2006)

# References IV

 Garnkel et al. Compatibility is Not Transparency: VMM Detection Myths and Realities. (2007)

 Ferrie et al. Attacks on More Virtual Machine Emulators. (2007)

# Total Slide Count

**500**