Reverse-Engineering

Ethical Hacking

Alessandro Brighente
Eleonora Losiouk
Master Degree on Cybersecurity

UNIVERSITÀ DEGLI STUDI DI PADOVA

SPRITZ SECURITY & PRIVACY RESEARCH GROUP

Outline

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Reverse-engineering definition
- ELF format
- Assembly instructions
- Calling conventions

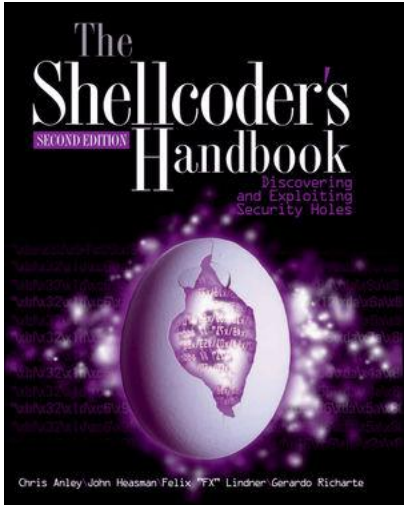
Outline

SPRITZ
Security & Privacy
Research Group

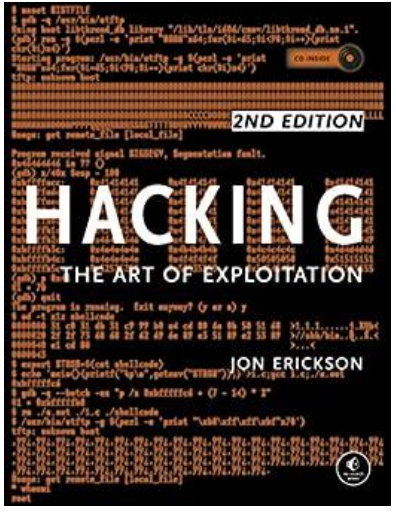
Università
degli Studi
di Padova

- Reverse-engineering definition
- ELF format
- Assembly instructions
- Calling conventions

Additional Material



The Shellcoder's Handbook: Discovering and Exploiting Security



Hacking: The Art of Exploitation 2nd Edition

Reverse-Engineering Environment

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Install a Virtual Machine (VirtualBox will do) and put all your tools there.
- Which OS?
  - **Kali**, if you don't know what you want
  - **Ubuntu**, if you want to be safe (more or less)
  - **Xubuntu**, for a lighter version

What's reversing?

Not limited to software, e.g, network protocols

What's reversing?

"[...] the process of analyzing a subject system to create representations of the system at a higher level of abstraction."

Chikofsky, Cross (1990)

- Why?
  - Missing or poor documentation
  - Opening up proprietary platforms
  - Security auditing
  - Curiosity

Compiling Software


```
int main() {
  puts("YAY");
  return 0;
}
```

Source code

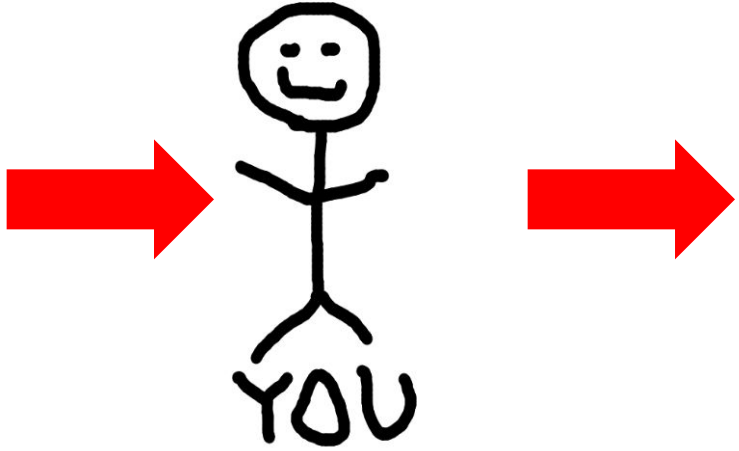

```
000100100100
...
```

Binary

Reversing Software

```
000100100100
...
```

Binary

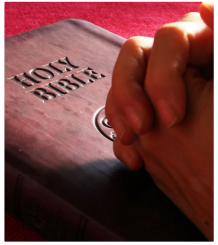



```
int main() {
  puts("YAY");
  return 0;
}
```

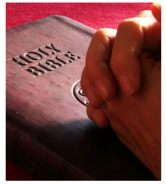
Source code

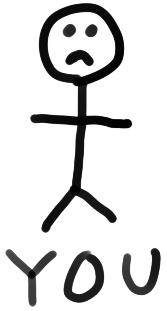
Reversing Software - The Truth


```
00010010
...
```




```
mov eax, 3
call func
ret
```






```
int main() {
  puts("YAY");
  return 0;
}
```

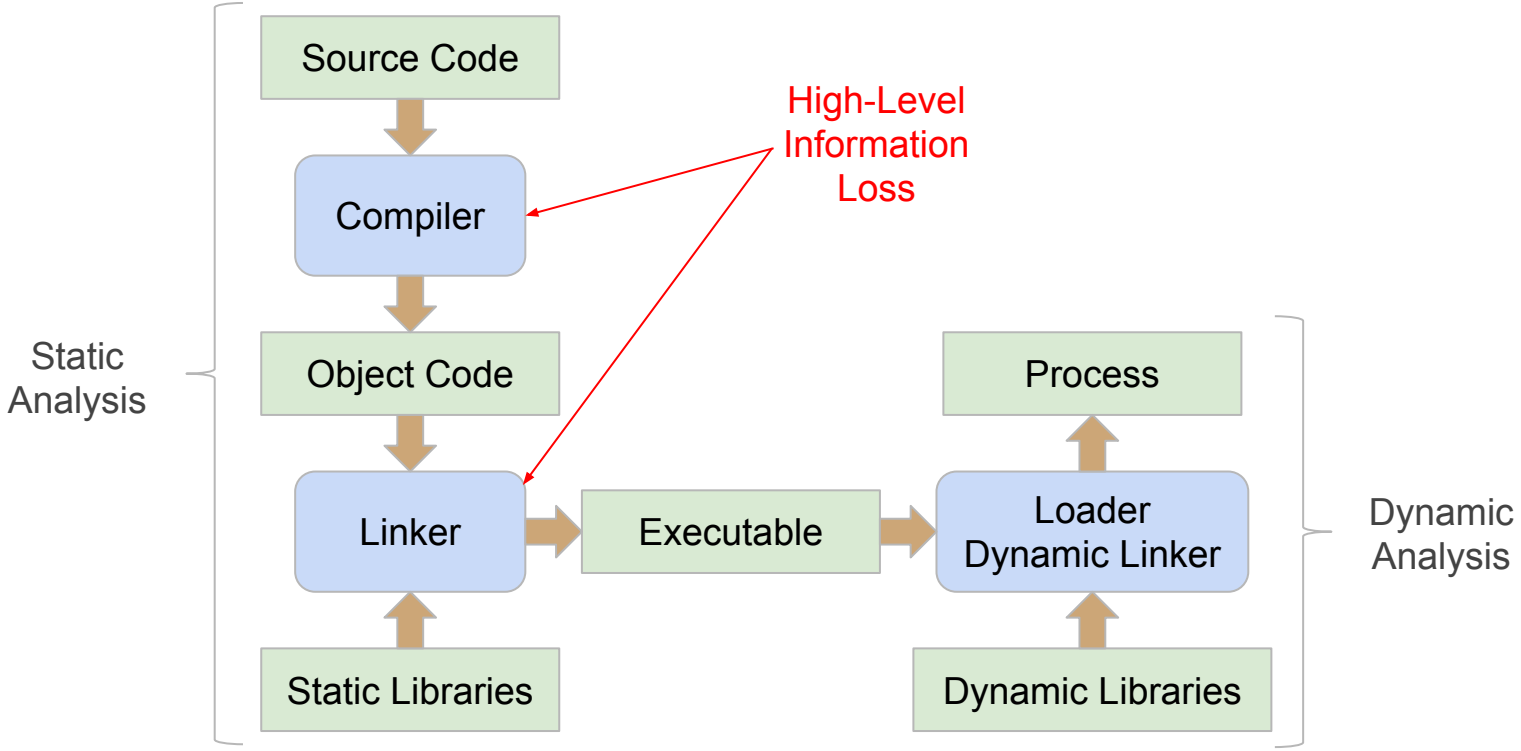
Why is it relevant?

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- You don’t always have access to source code
- Vulnerability assessment
- Malware analysis
- Pwning
- Algorithm reversing
- Interoperability (SMB/Samba, Windows/Wine)
- Hacking embedded devices

A program's lifecycle



Outline

- Reverse-engineering definition
- ELF format
- Assembly instructions
- Calling conventions

Executables

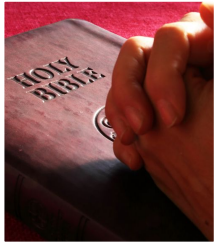
- OS-specific format
  - e.g. **ELF** (*nix), PE (Windows), Mach-O (MacOS, iOS)
- Generally, **same format used for programs and libraries**
- Made of sections that will be memory-mapped
  - e.g. .**text**, .(ro)data, .bss
- Specifies imports from dynamic libraries
  - e.g. **GOT/PLT** (ELF), IAT (PE)
- **Loading methods**:
  - Fixed address
  - Relocation
  - Position-independent

Executable and Linkable Format (ELF)

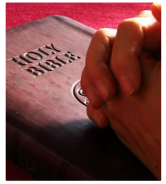
- Introduced in **System V Release 4**, used by most Unix-like OSes
  - Executables, object code, shared libraries, core dumps
- Designed to be **flexible**, **extensible** and **cross-platform**
- **Program headers** describe segments (i.e. virtual mappings)
- **Section headers** describe sections and how to load them into segments
- Supports **relocation** (i.e., connecting symbolic references with symbolic definitions)

Reversing Software - The Truth

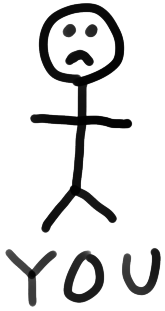

```
00010010
...
```




```
mov eax, 3
call func
ret
```



Disassembling



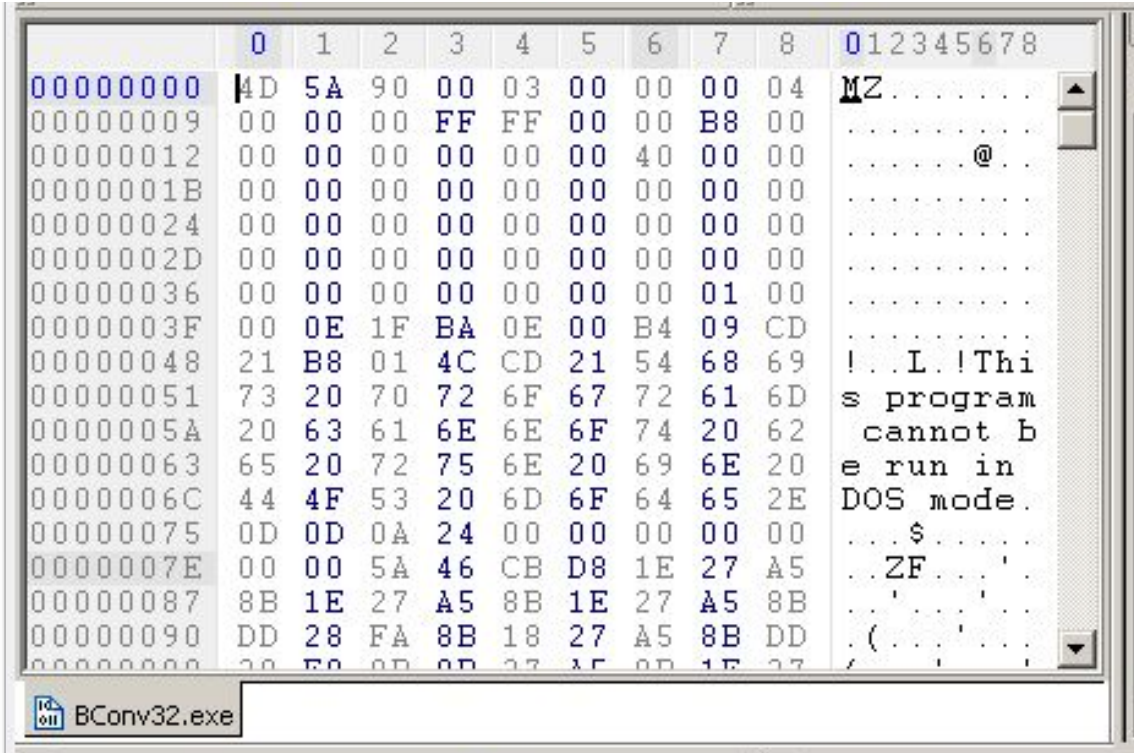
Decompiling

```
int main() {
  puts("YAY");
  return 0;
}
```

Techniques

- **Executable information**
  - file, readelf, PEview, hexedit
- **Static** analysis doesn't run the executable
  - Disassembly
    - objdump, IDA, radare, Hopper, Binary Ninja
  - Decompilation
    - Ghidra
  - Abstract interpretation
  - Symbolic execution
- **Dynamic** analysis runs the executable
  - Debugging
    - gdb, WinDbg, OllyDbg, Immunity Debugger, qira, ...
  - Dynamic binary instrumentation

Hex Editor



	0	1	2	3	4	5	6	7	8	012345678
00000000	4D	5A	90	00	03	00	00	00	04	MZ.....
00000009	00	00	00	FF	FF	00	00	B8	00	.....
00000012	00	00	00	00	00	00	40	00	00	.....@..
0000001B	00	00	00	00	00	00	00	00	00	.....
00000024	00	00	00	00	00	00	00	00	00	.....
0000002D	00	00	00	00	00	00	00	00	00	.....
00000036	00	00	00	00	00	00	00	01	00	.....
0000003F	00	0E	1F	BA	0E	00	B4	09	CD	.....
00000048	21	B8	01	4C	CD	21	54	68	69	!..L.!Thi
00000051	73	20	70	72	6F	67	72	61	6D	s program
0000005A	20	63	61	6E	6E	6F	74	20	62	cannot b
00000063	65	20	72	75	6E	20	69	6E	20	e run in
0000006C	44	4F	53	20	6D	6F	64	65	2E	DOS mode.
00000075	0D	0D	0A	24	00	00	00	00	00	...$.....
0000007E	00	00	5A	46	CB	D8	1E	27	A5	..ZF...'.
00000087	8B	1E	27	A5	8B	1E	27	A5	8B	..'...'..
00000090	DD	28	FA	8B	18	27	A5	8B	DD	.(...'...

BConv32.exe

Hex Editor

- **Patch** programs
- **Inspect** file formats
- **Change** content of files

Many different options here (hexedit, biew, vim, etc…)

Tools

- IDA Pro (https://www.hex-rays.com/products/ida/)
  - GUI
  - Industry standard
  - $$$$$
- Binary Ninja (https://binary.ninja/)
  - GUI
  - Very nice scripting features + has “undo” functionality
  - $$
- Radare2 (https://github.com/radare/radare2)
  - CLI (experimental GUI @ https://github.com/radareorg/cutter/releases)
  - Opensource
- Ghidra
  - NSA reversing tool
- Objdump
  - Seriously, don’t

Can’t I just use a decompiler?

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Can speed up the reversing, but...
- Decompiling is (generally) **undecidable**
- **Fails** in many cases
- Sometimes you want to work at the ASM level (pwning)

Outline

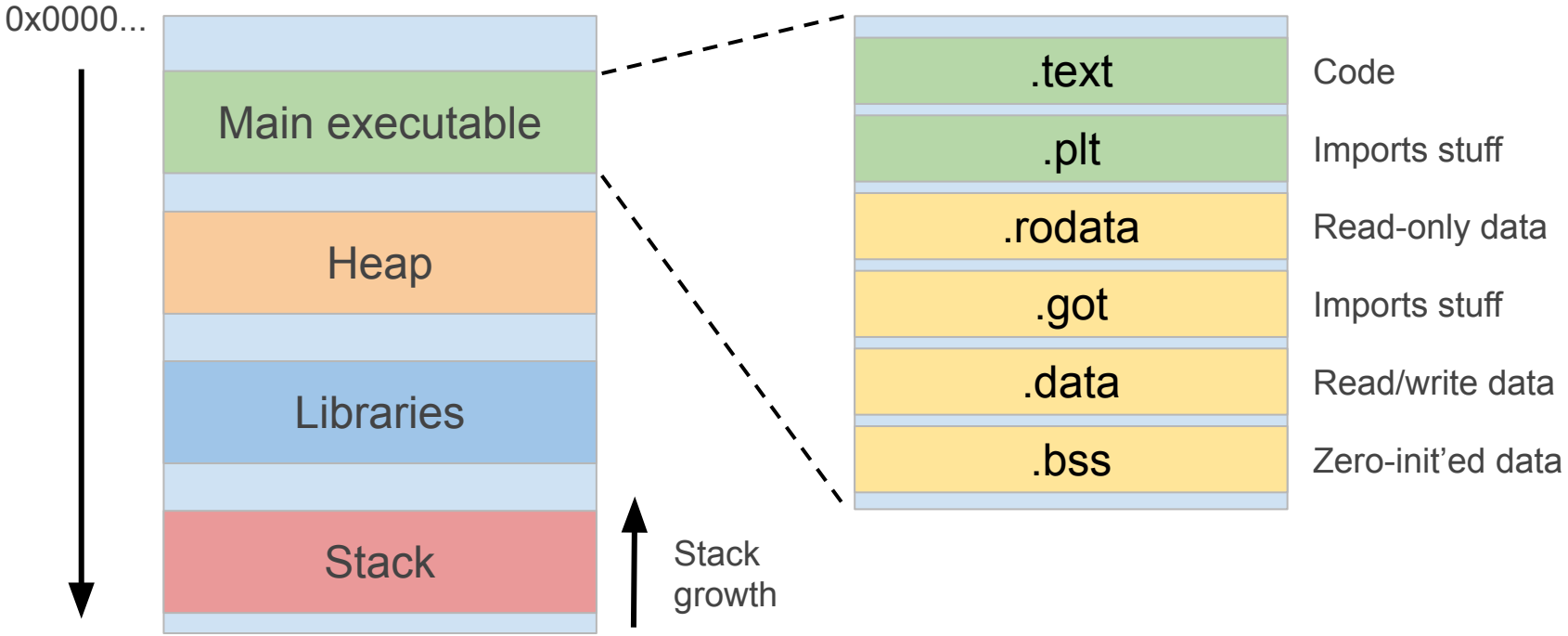
SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

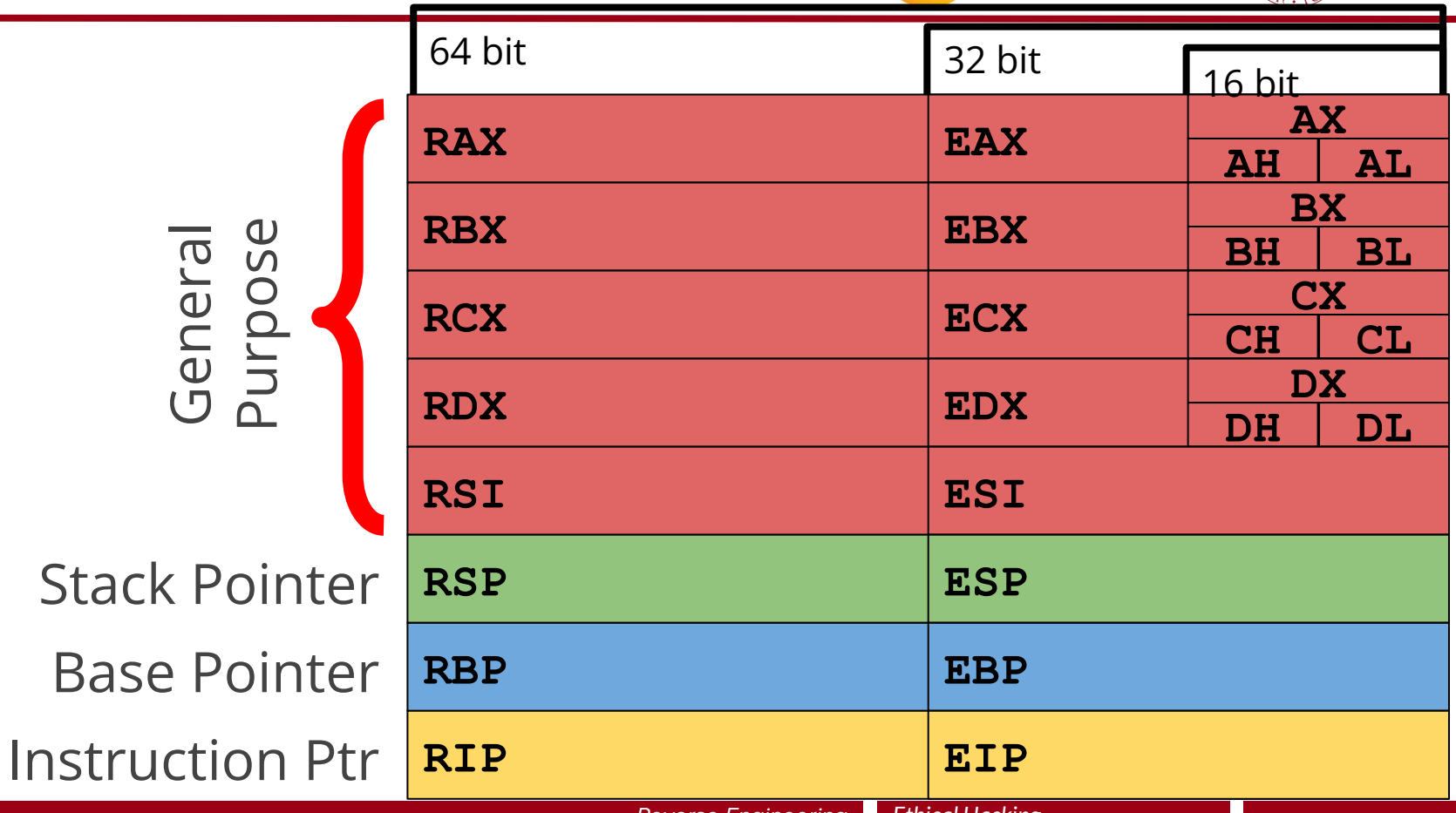
- Reverse-engineering definition
- ELF format
- Assembly instructions
- Calling conventions

- Your computer probably runs on x86_64
  - x86 still supported
  - **32 bit vs 64 bit**

Quick recap: a (Linux) process' memory



X86_64 Registers



Instructions - MOV <dst>, <src>

- Copy <src> into <dst>
- MOV EAX, 16
  - EAX = 16
- MOV EAX, [ESP+4]
  - EAX = *(ESP+4)
- MOV AL, ‘a’
  - AL = 0x61

Instructions - LEA <dst>, <src>

- Load Effective Address of <src> into <dst>
- Used to access elements from a buffer/array
- Used to perform simple math operations
- LEA ECX, [EAX+3]
  - ECX = EAX + 3
- LEA EAX, [EBX+2*ESI]
  - EAX = EBX+2*ESI

Instructions - PUSH <src>

- Decrement RSP and put <src> onto the stack (push)
- PUSH EAX
  - ESP -= 4
  - *ESP = (dword) EAX
- PUSH CX
  - ESP -= 2
  - *ESP = (word) CX
- PUSH RAX
  - RSP -= 8
  - *RPS = (qword) RAX

Instructions - POP <dst>

- <dst> takes the value on top of the stack, RSP gets incremented
- POP EAX
  - EAX = *ESP
  - ESP += 4
- POP CX
  - CX = *ESP
  - ESP += 2

PUSH/POP example

PUSH EAX
POP EBX
=
MOV EBX, EAX

Instructions - ADD <dst>, <src>

- <dst> += <src>
- ADD EAX, 16
  - EAX += 16
- ADD AH, AL
  - AH += AL
- ADD ESP, 0x10
  - Remove 16 bytes from the stack

Instructions - SUB <dst>, <src>

- <dst> -= <src>
- SUB EAX, 16
  - EAX -= 16
- SUB AH, AL
  - AH -= AL
- SUB ESP, 0x10
  - Allocate 16 bytes of space on the stack

Flags

- x86 instructions can modify a special register called FLAGS
- FLAGS contains 1-bit flags:
  - Ex: OF, SF, ZF, AF, PF, and CF
- ZF = Zero Flag
- SF = Sign Flag
- CF = Carry Flag

Flags

- Zero Flag
  - set if the result of last operation was zero
- Sign Flag
  - set if the result of last operation was negative (dst - src <s 0)
- Carry Flag
  - set if integer underflow (dst <u src)
- See https://stackoverflow.com/questions/8965923/carry-overflow-subtraction-in-x86

Flags

MOV RAX, 666

SUB RAX, 666

=>

ZF = 1
SF = 0
CF = 0

Flags

MOV RAX, 123

SUB RAX, 666

=>

ZF = 0
SF = 1
CF = 1

MOV AL, 0xFF

SUB AL, 0x01

=>

ZF = 0

SF = 1 (-1 - 1 = -2 < 0)

CF = 0 (255 - 1 = 254 > 0)

Instructions - CMP <dst>, <src>

- CoMPare
- Perform a SUB but throw away the result
- Used to set flags
- CMP EAX, 13
  - EAX value doesn't change
  - TMP = EAX - 13
  - Update the FLAGS according to TMP

Instructions - JMP <dst>

- JuMP to <dst>
- JMP RAX
  - Jump to the address saved in RAX
- JMP 0x1234
  - Jump to address 0x1234

Instructions - Jxx <dst>

- Conditional jump
- Used to control the flow of a program (ex.: IF expressions)
- JZ/JE => jump if ZF = 1
- JNZ/JNE => jump if ZF = 0
- JB, JA => Jump if <dst> Below/Above <src> (unsigned)
- JL, JG => Jump if <dst> Less/Greater than <src> (signed)
- Many others
- See http://unixwiz.net/techtips/x86-jumps.html

Jxx - Example: Password length == 16?

```
MOV RAX, password_length

CMP RAX, 0x10

JZ  ok

JMP exit

ok:

...print ‘yay’...
```

Jxx - Example: Given number >= 11?


```
MOV RAX, integer_user_input

CMP RAX, 11

JB  fail

JMP ok


fail: ...print ‘too short’...

ok: ...print ‘OK’...
```

Instructions - XOR <dst>, <src>

- Perform a bitwise XOR between <dst> and <src>
- XOR EAX, EBX
  - EAX ^= EBX
- Truth table:

	0	1
0	0	1
1	1	0

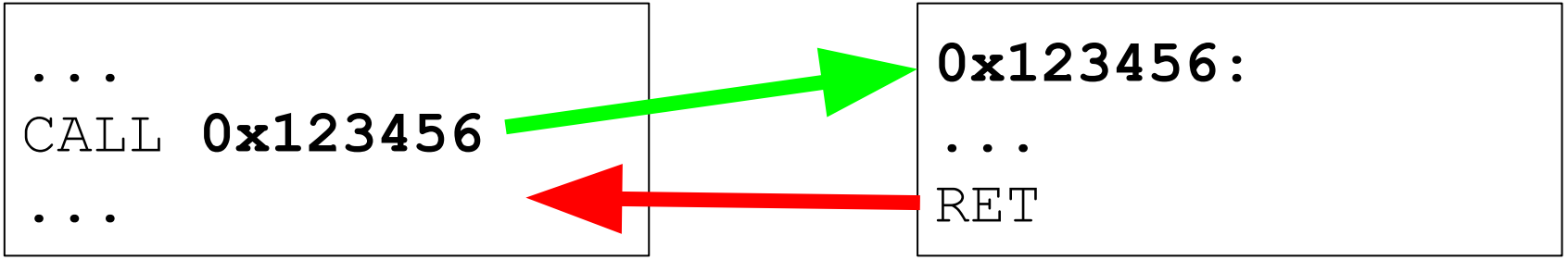
Instructions - CALL <dst>

- CALL a subroutine
- CALL 0x123456
  - Push return address on the stack
  - RIP = 0x123456
- Function parameters passed in many different ways

Instructions - RET

- RETurn from a subroutine
- RET
  - Pop return address from stack
  - Jump to it

CALL / RET



Outline

SPRITZ
Security & Privacy
Research Group

Università
degli Studi
di Padova

- Reverse-engineering definition
- ELF format
- Assembly instructions
- Calling conventions

How are function parameters passed around?

- On x86, there are many calling conventions
- Sometimes parameters are passed in registers
- Sometimes on the stack
- Return value usually in RAX/EAX
- You should take some time to look at them

https://en.wikipedia.org/wiki/X86_calling_conventions

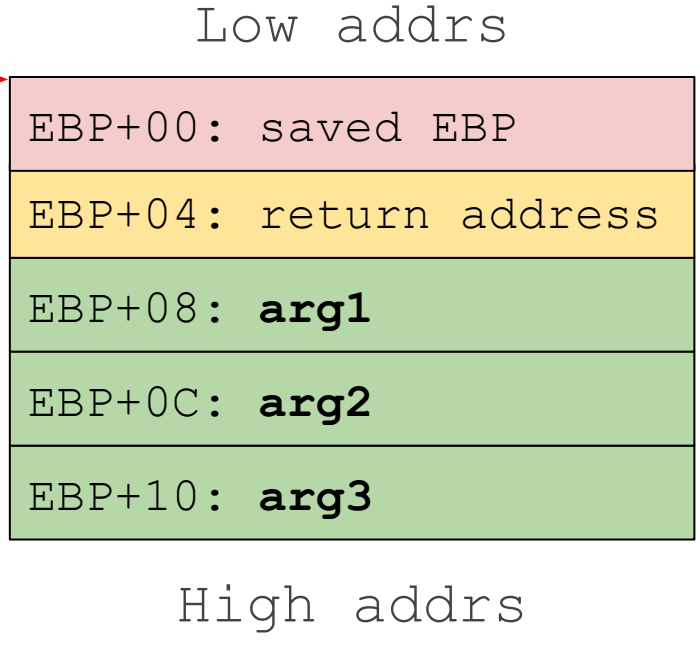
Calling Convention - cdecl


```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
    ; make new call frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call    callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
    ; restore old call frame
    pop     ebp
    ; return
    ret
```

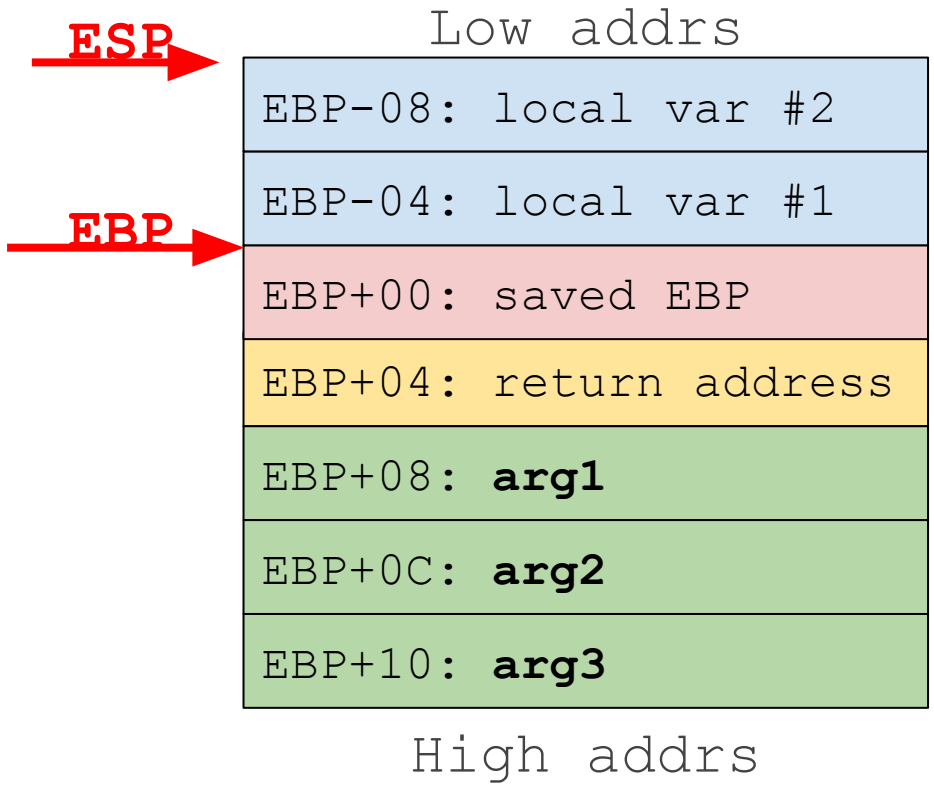
Calling Convention - cdecl


```
callee:
push    ebp
mov     ebp, esp
mov     edx, dword [ebp+0x8 {arg1}]
mov     eax, dword [ebp+0xc {arg2}]
add     edx, eax
mov     eax, dword [ebp+0x10 {arg3}]
add     eax, edx
pop     ebp
retn
```



Calling Convention - cdecl


```
callee:
push    ebp
mov     ebp, esp
mov     edx, dword [ebp+0x8 {arg1}]
mov     eax, dword [ebp+0xc {arg2}]
add     edx, eax
mov     eax, dword [ebp+0x10 {arg3}]
add     eax, edx
pop     ebp
retn
```



Calling Convention - SystemV AMD64

- Arguments in registers: rdi, rsi, rdx, rcx, r8, r9
- Further args on stack, like cdecl
- Red-zoning: leaf function with frames <= 128 bytes do not need to reserve stack space

```c
int callee(int, int, int);

int caller(void)
{
    int ret;

    ret = callee(1, 2, 3);
    ret += 5;
    return ret;
}
```

```asm
caller:
    ; set up stack frame
    push rbp
    mov  rbp, rsp
    ; set up arguments
    mov  edi, 1
    mov  esi, 2
    mov  edx, 3
    ; call subroutine 'callee'
    call callee
    ; use subroutine result
    add  eax, 5
    ; restore old stack frame
    pop  rbp
    ; return
    ret
```