Rev101

spritzers - CTF team

spritz.math.unipd.it/spritzers.html

Disclaimer

All information presented here has the only purpose of teaching how reverse engineering works.

Use your mad skillz only in CTFs or other situations in which you are legally allowed to do so.

Do not hack the new Playstation. Or maybe do, but be prepared to get legal troubles (I’m looking at you, geohot).

Disclaimer

But seriously, if you do pls tell me. It’d be awesome.

Reversing in CTFs

In reversing challenges you have to understand how a program works, but you don’t have its source code.

You typically have to reverse an algorithm (encryption?) to get the flag.

Most of the time, solving a challenge is a bit time consuming but straightforward.

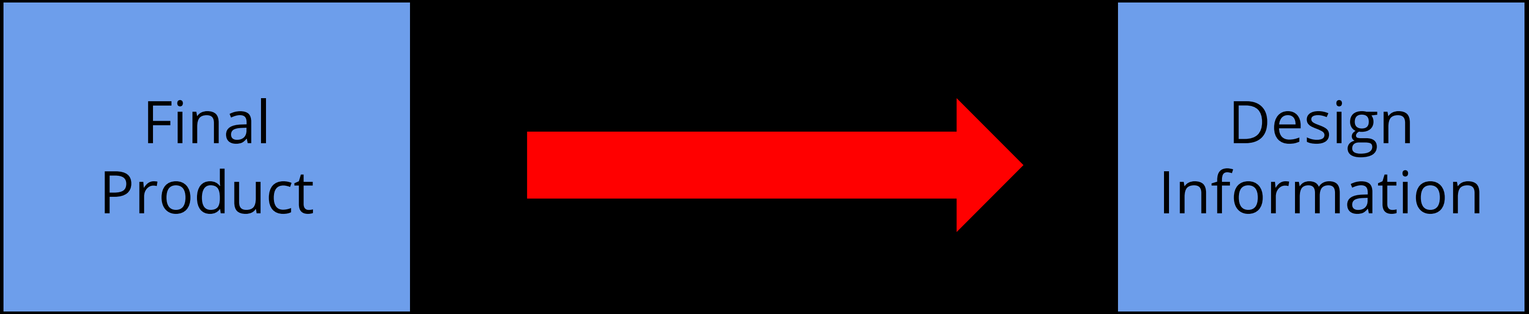
...Unless obfuscation is involved.

Reversing IRL

A lot of cool stuff, but legally it’s a gray area.

Reverse Engineering?

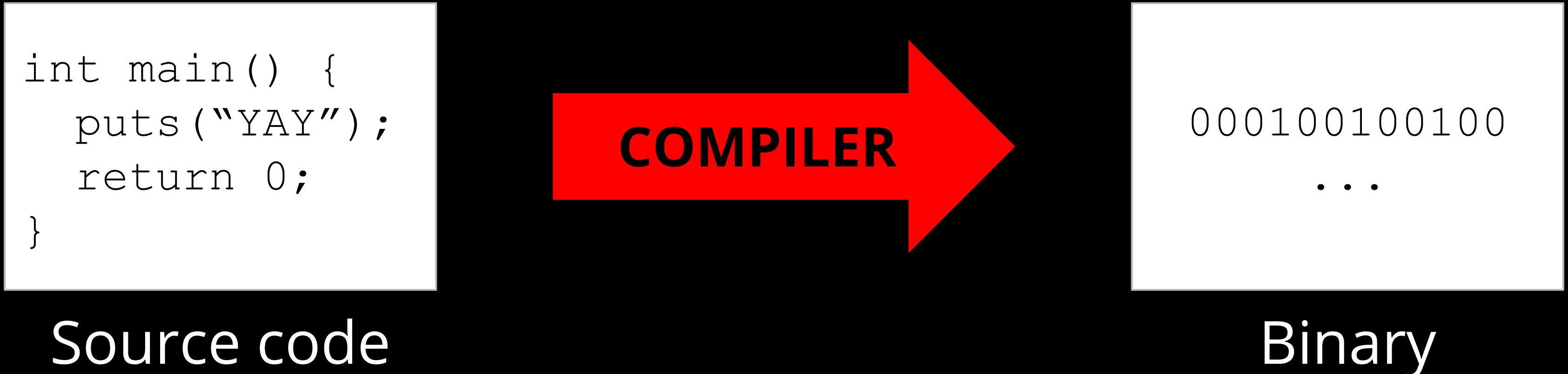
What it is



Not limited to software

(Binary) Software Reverse Engineering

Compiling Software



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

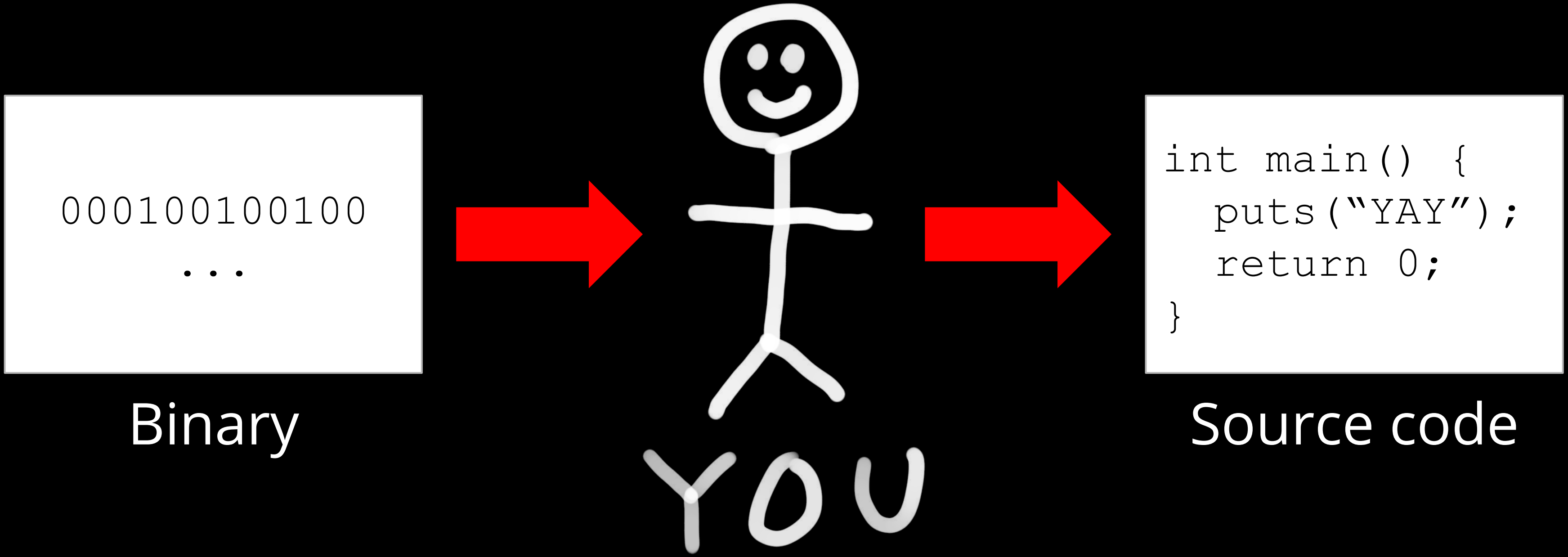
Source code

COMPILER

```
000100100100
     ...
```

Binary

Reversing Software



```
000100100100
...
```

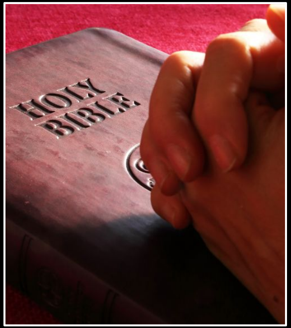
Binary

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

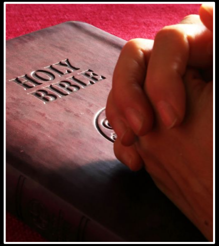
Source code

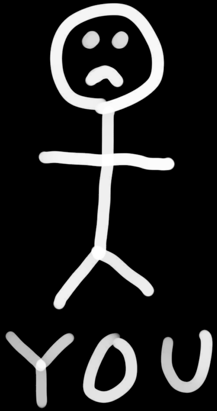
Reversing Software - The Truth

```
00010010
...
```

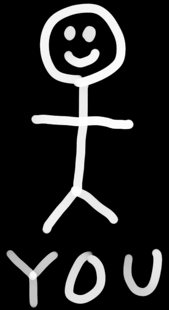


```
mov eax, 3
call func
ret
```





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



Why is it relevant?

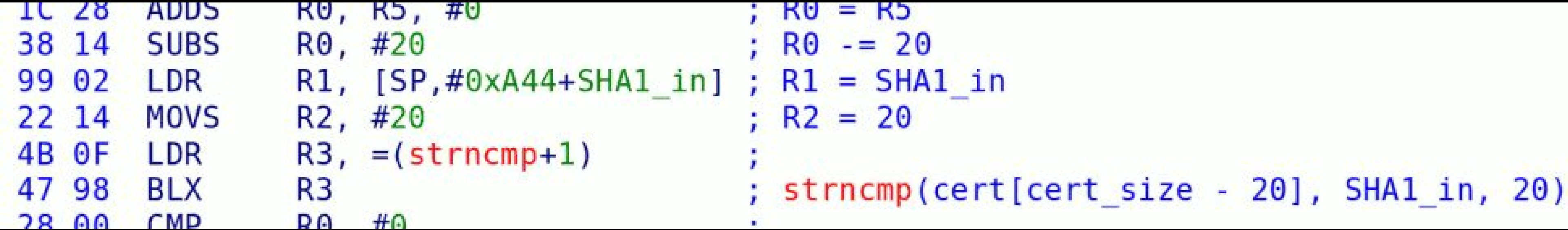
- You don’t always have access to source code
- Vulnerability assessment
- Malware analysis
- Pwning
- Algorithm reversing (default WPA anyone?)
- Interoperability (SMB/Samba, Windows/Wine)
- Hacking embedded devices

Can’t I just use a decompiler?

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

Why should I do it?

- Sometimes it’s fun



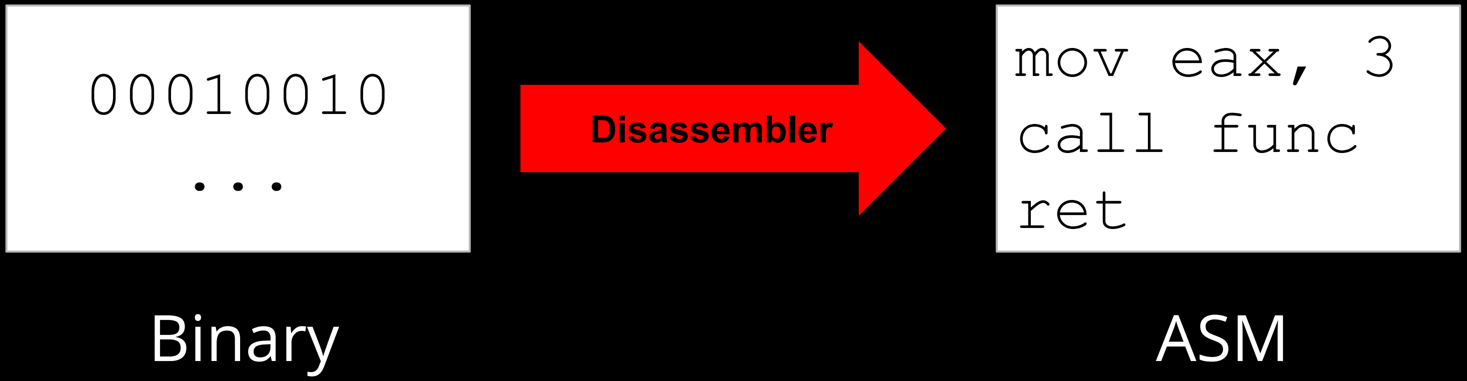
```
1C 28  ADDS    R0, R5, #0             ; R0 = R5
38 14  SUBS    R0, #20                ; R0 -= 20
99 02  LDR     R1, [SP,#0xA44+SHA1_in] ; R1 = SHA1_in
22 14  MOVS    R2, #20                ; R2 = 20
4B 0F  LDR     R3, =(strncmp+1)       ;
47 98  BLX     R3                     ; strncmp(cert[cert_size - 20], SHA1_in, 20)
28 00  CMP     R0, #0                 ;
```

This is straight from the Wii’s game signature checking.

(Credits: https://hackmii.com/)

The Tools

Disassembler



```
00010010
...
```

Disassembler

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

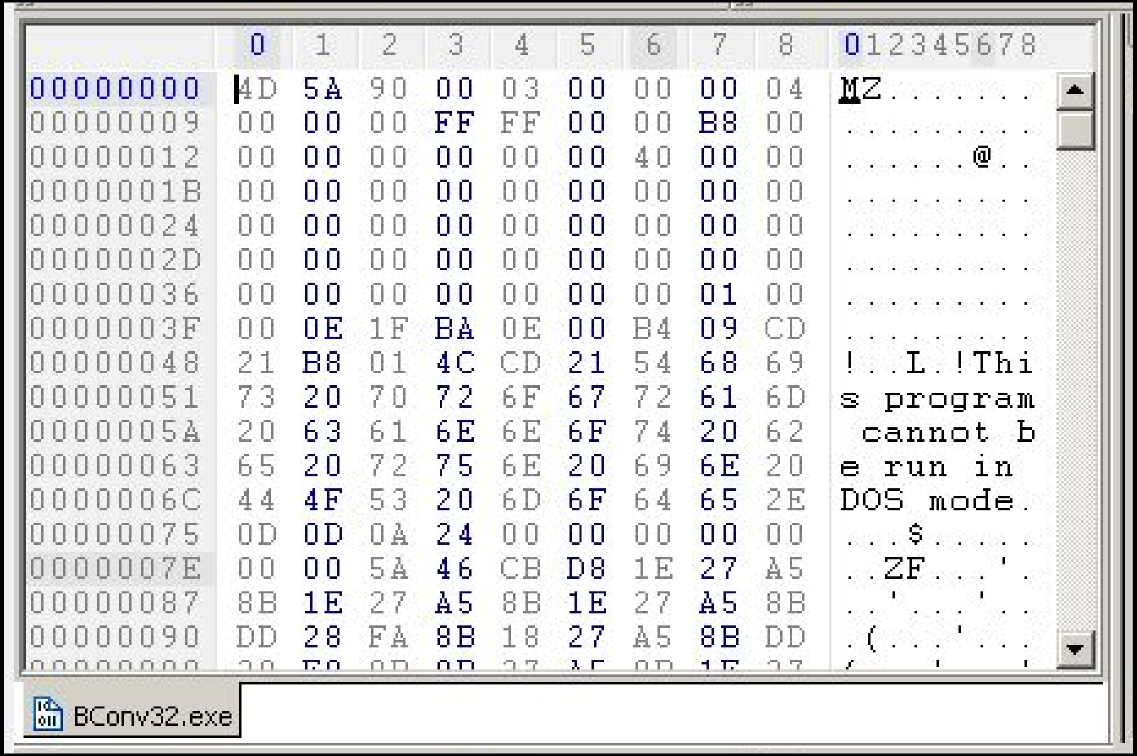
Binary

ASM

Disassembler

- **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
- **Objdump**
  - Seriously, don’t

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  .(...'...
```

Hex Editor

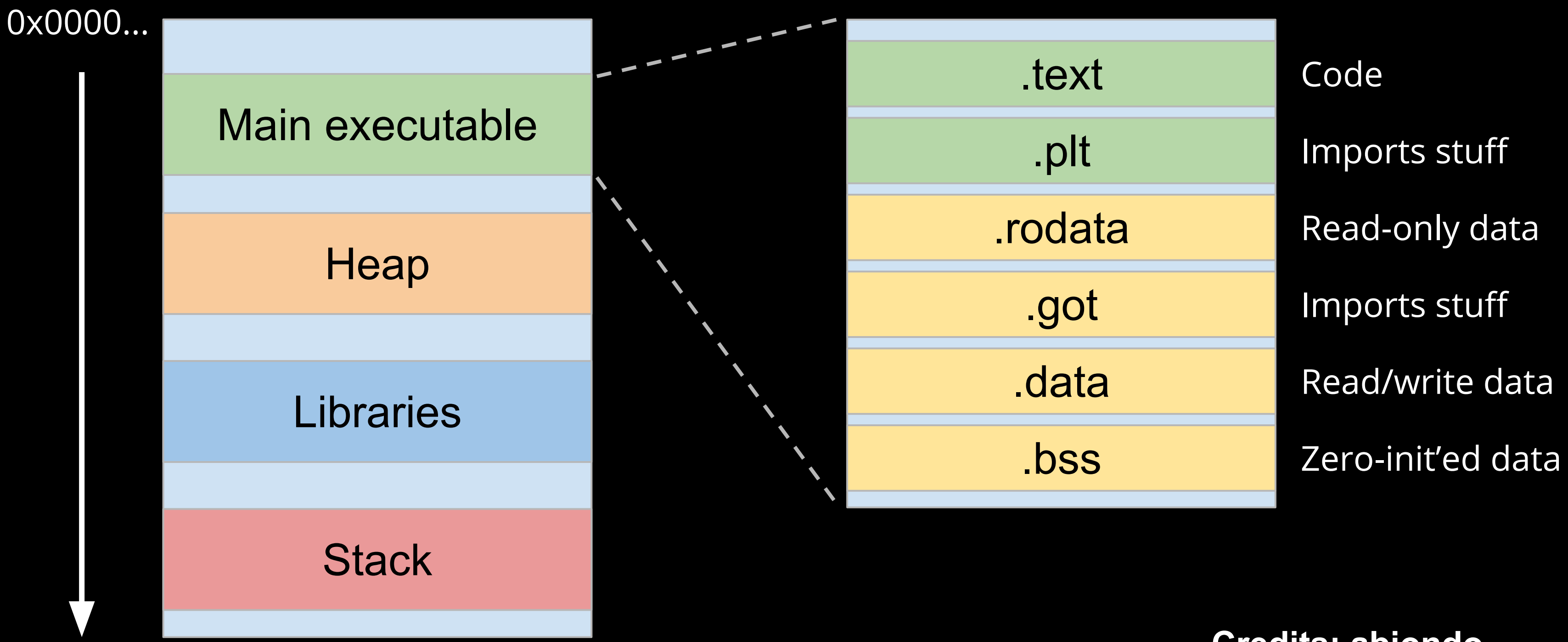
- Patch programs
- Inspect file formats
- Change content of files

Many different options here (hexedit, biew, etc...)

Introduction to x86 ASM
(yay)

I DON'T EVEN SEE THE CODE
ALL I SEE IS BLONDE, BRUNETTE, REDHEAD
imgflip.com

Quick recap: a process' memory

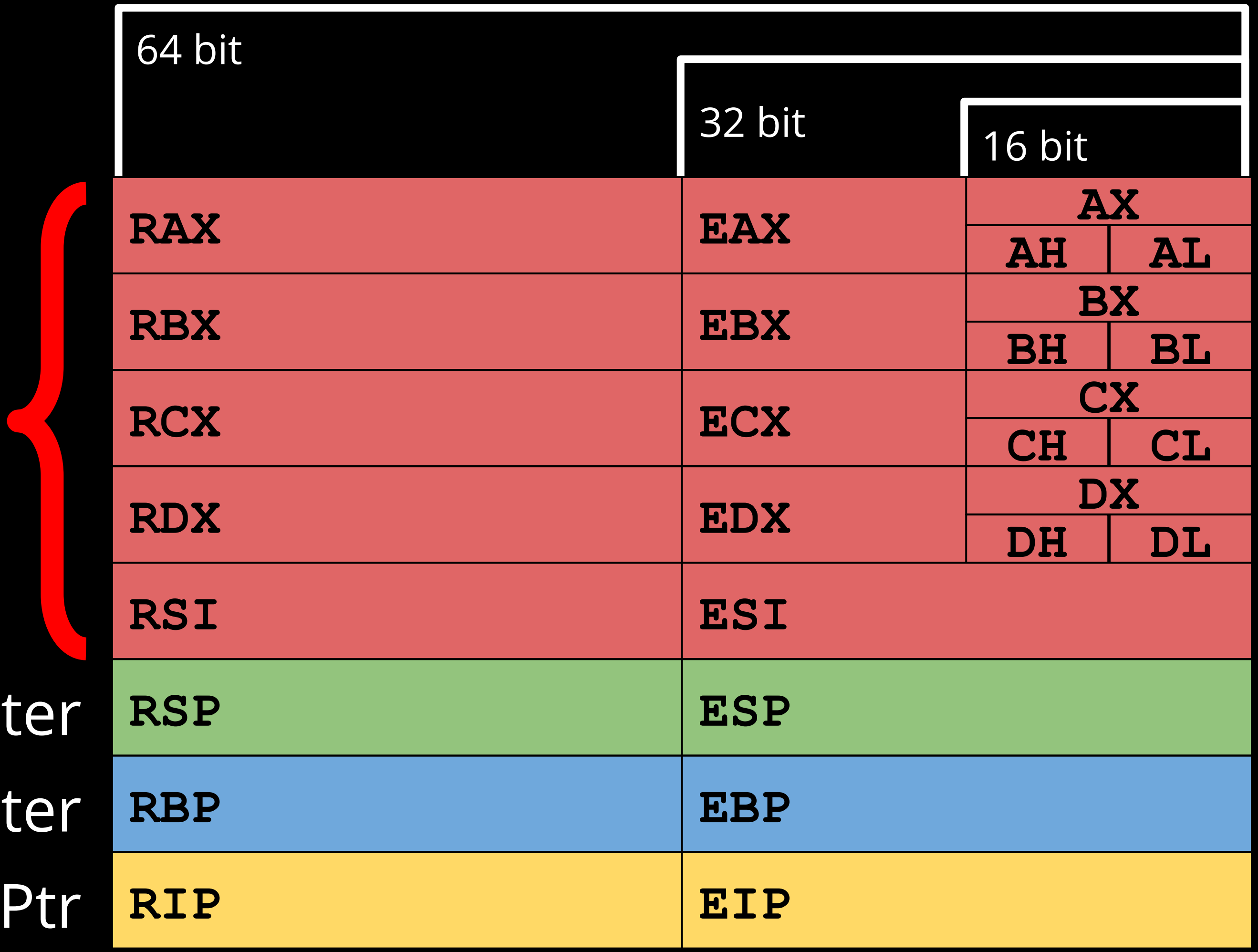


Credits: abiondo

Introduction to x86 ASM

- Only architecture supported by IDA/Binja demo :(
- Your computer probably runs on x86_64
  - x86 still supported
  - 32 bit vs 64 bit
- This is **NOT** supposed to be a complete ASM lesson (booooring)

(some) x86_64 Registers



	64 bit	32 bit	16 bit
General Purpose	RAX	EAX	AX AH AL
	RBX	EBX	BX BH BL
	RCX	ECX	CX CH CL
	RDX	EDX	DX DH DL
	RSI	ESI	
Stack Pointer	RSP	ESP	
Base Pointer	RBP	EBP	
Instruction Ptr	RIP	EIP	

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 ESP and put <src> onto the stack (push)
- PUSH EAX
    - ESP -= 4
    - *ESP = (dword) EAX
- PUSH CX
    - ESP -= 2
    - *ESP = (word) CX

Instructions - POP ‹dst›

- <dst> takes the value on top of the stack, ESP 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 - Example

```
MOV RAX, 666

SUB RAX, 666

=>

ZF = 1

SF = 0

CF = 0
```

Flags - Example

```
MOV RAX, 123

SUB RAX, 666

=>

ZF = 0

SF = 1

CF = 1
```

Flags - Example

```
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

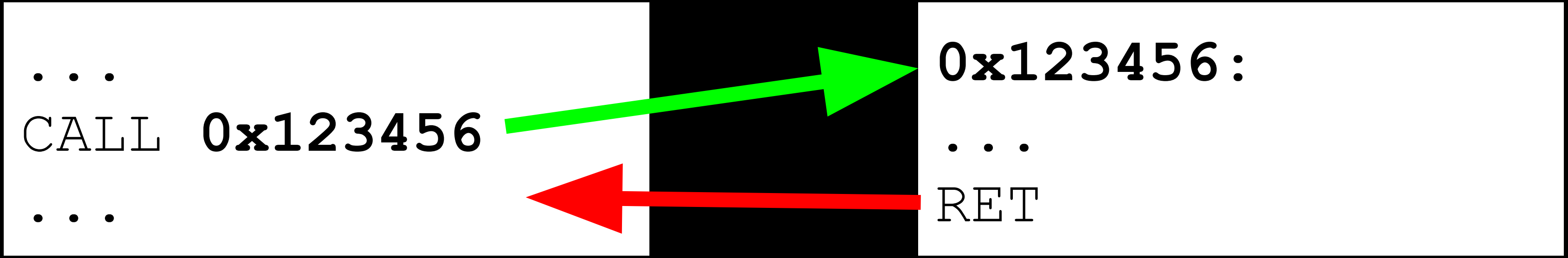
Instructions - CALL <dst>

- 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



```
...
CALL 0x123456
...
```

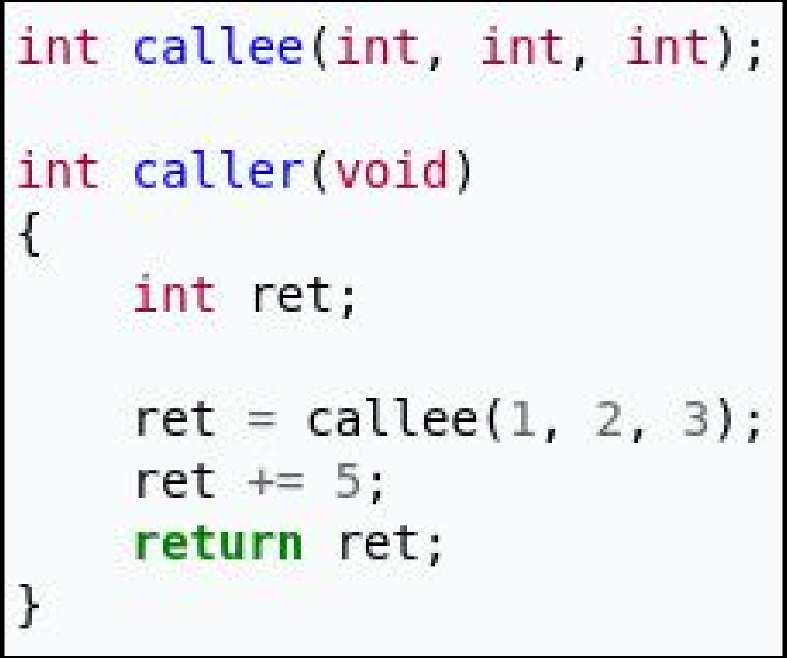
```
0x123456:
...
RET
```

How are function parameters passed around?

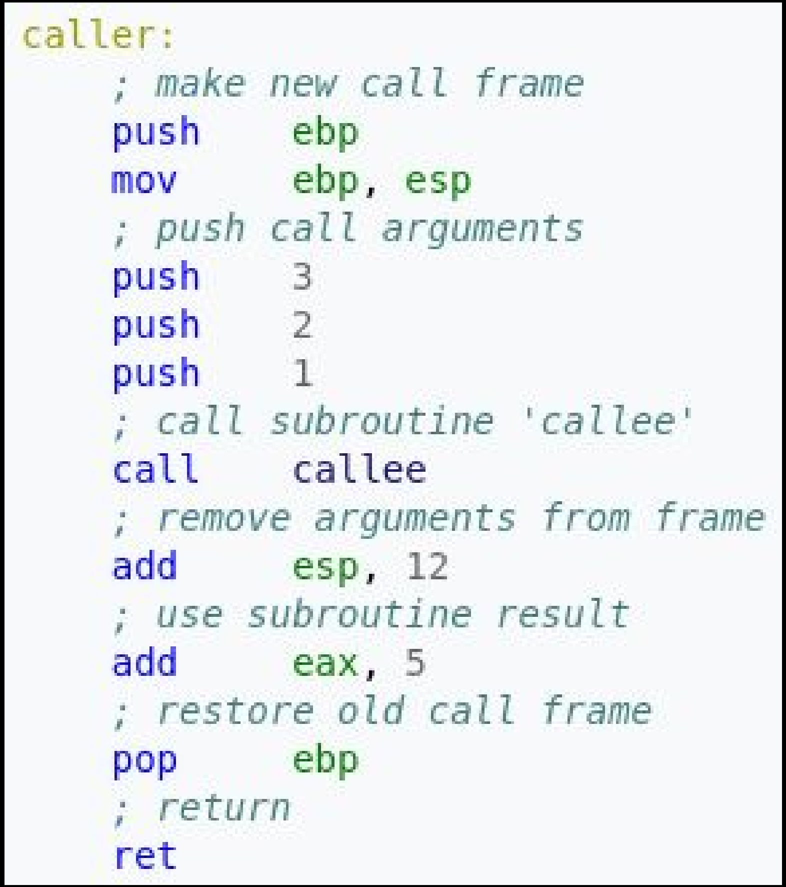
- On x86, there are many **calling conventions**
- Sometimes parameters are passed in registers
- Sometimes on the stack
- Return value usually in **RAX/EAX**
- You should take some time to look at them

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

Calling Convention - cdecl

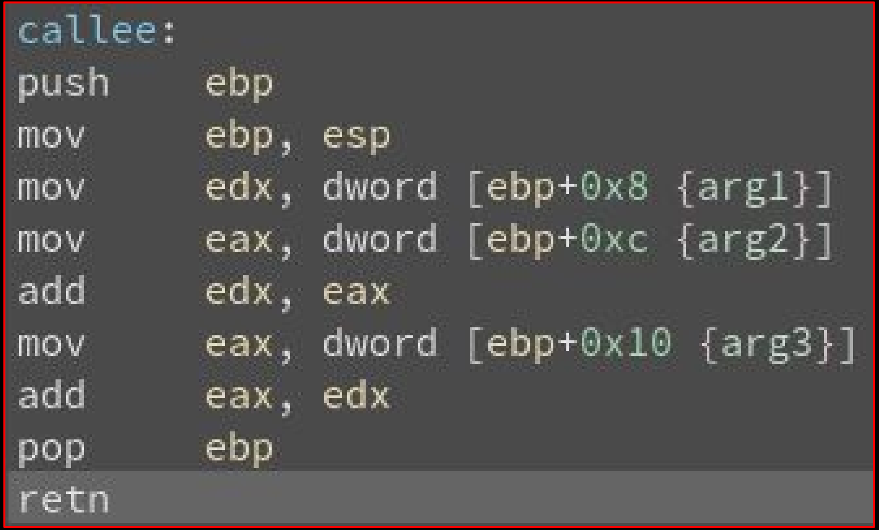


```c
int callee(int, int, int);

int caller(void)
{
    int ret;

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

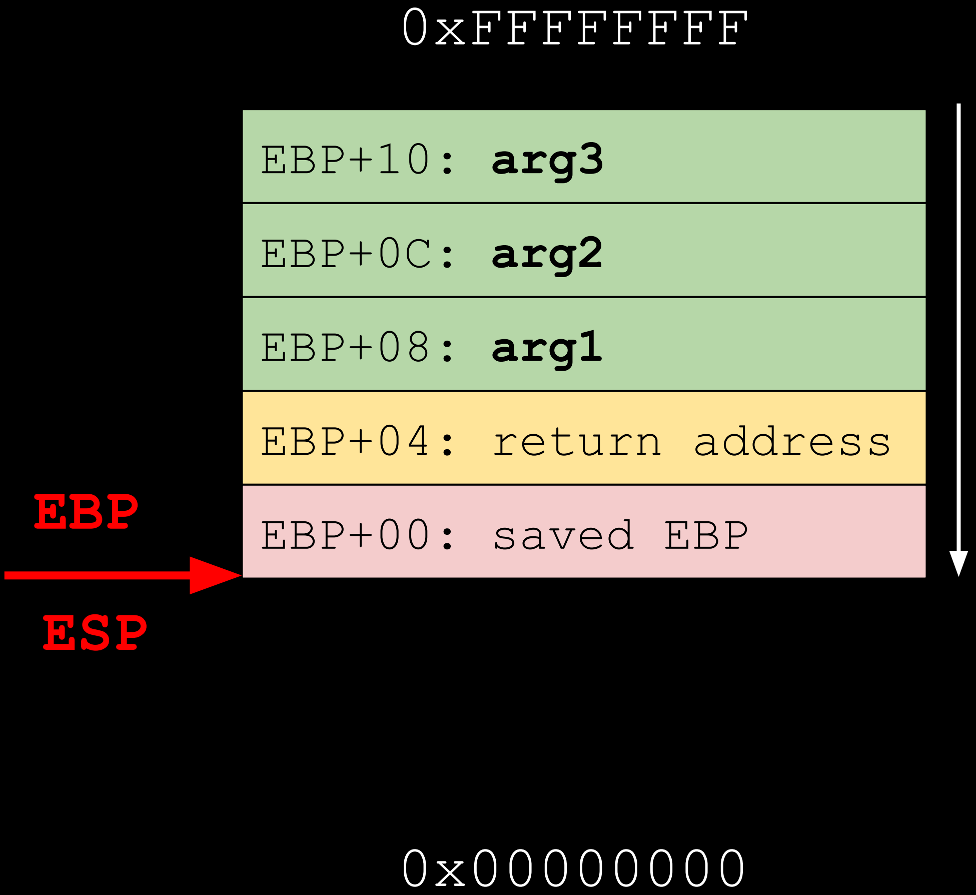


```asm
caller:
    ; make new call frame
    push    ebp
    mov     ebp, esp
    ; push call arguments
    push    3
    push    2
    push    1
    ; call subroutine 'callee'
    call    callee
    ; remove arguments from frame
    add     esp, 12
    ; use subroutine result
    add     eax, 5
   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



0xFFFFFFFF

EBP+10: **arg3**

EBP+0C: **arg2**

EBP+08: **arg1**

EBP+04: return address

EBP+00: saved EBP

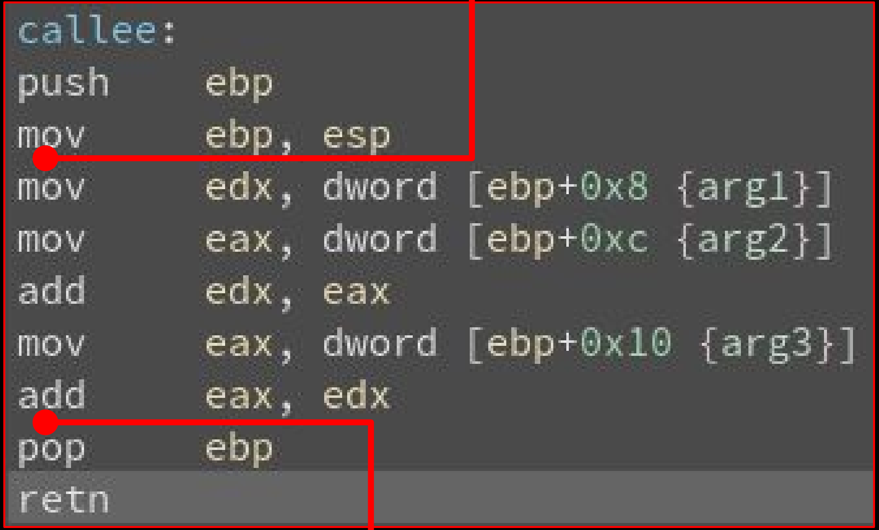
EBP

ESP

0x00000000

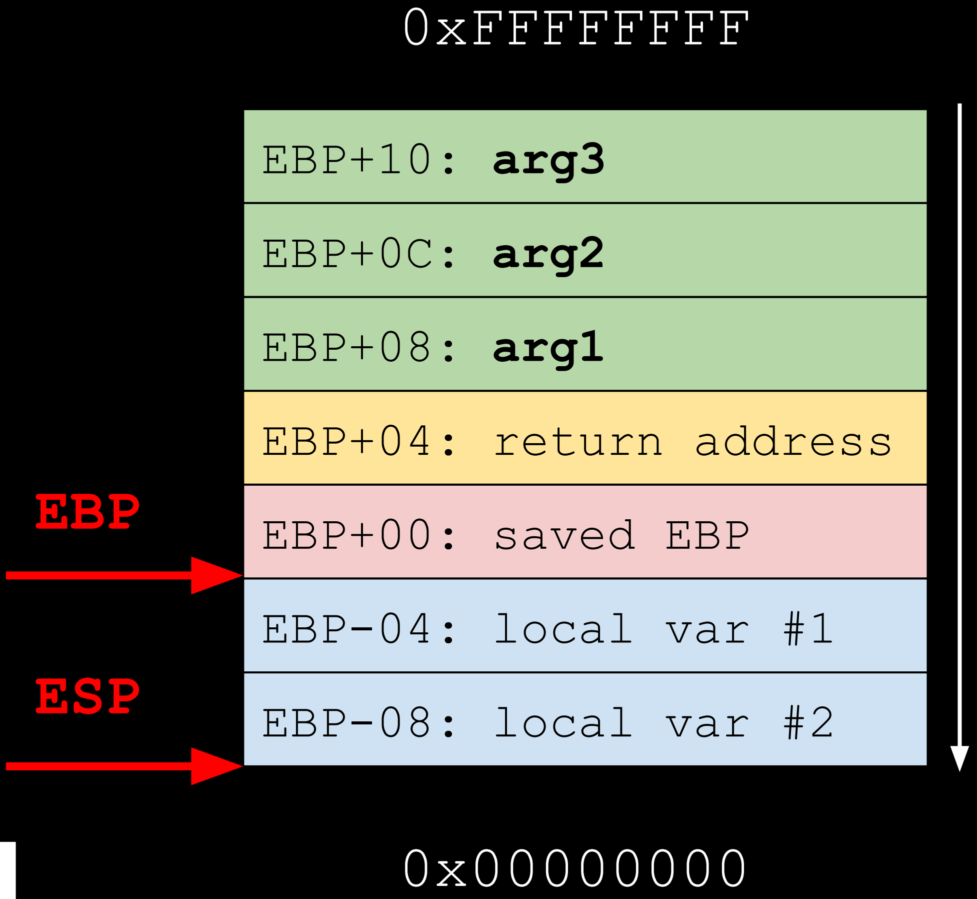
Calling Convention - cdecl - Local vars

```
sub esp, 8
```



```
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
```

```
mov esp, ebp
```



0xFFFFFFFF

EBP+10: **arg3**

EBP+0C: **arg2**

EBP+08: **arg1**

EBP+04: return address

EBP+00: saved EBP

EBP-04: local var #1

EBP-08: local var #2

0x00000000

Other useful instructions

NOP - Single-byte instruction that does nothing

RET - Return from a function

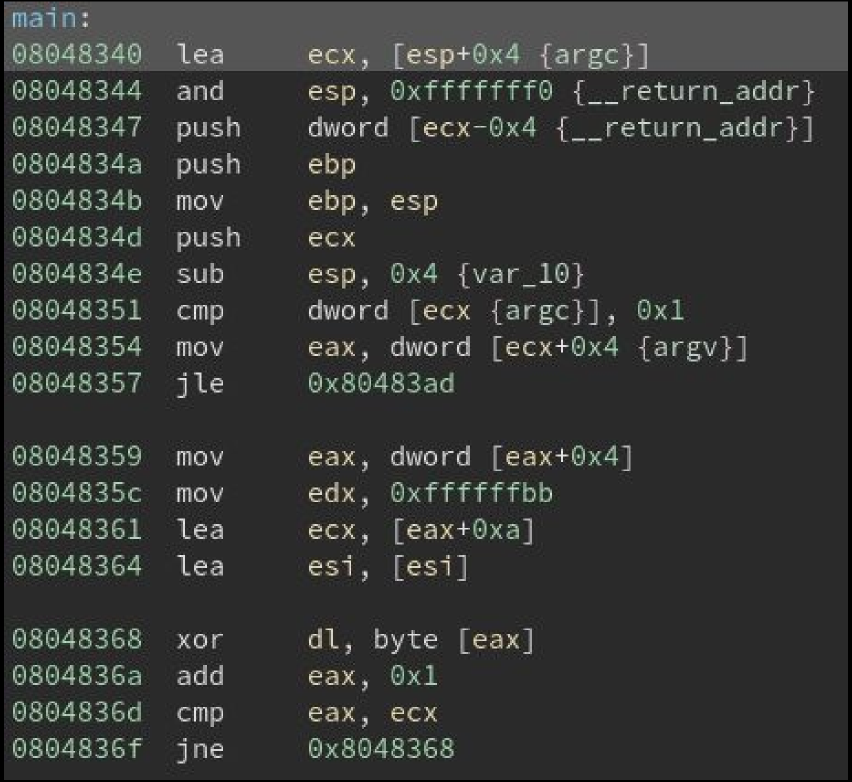
MOVZX - Move and zero extend

MOVSX - Move and sign extend

Now the (slightly) less boring part :D

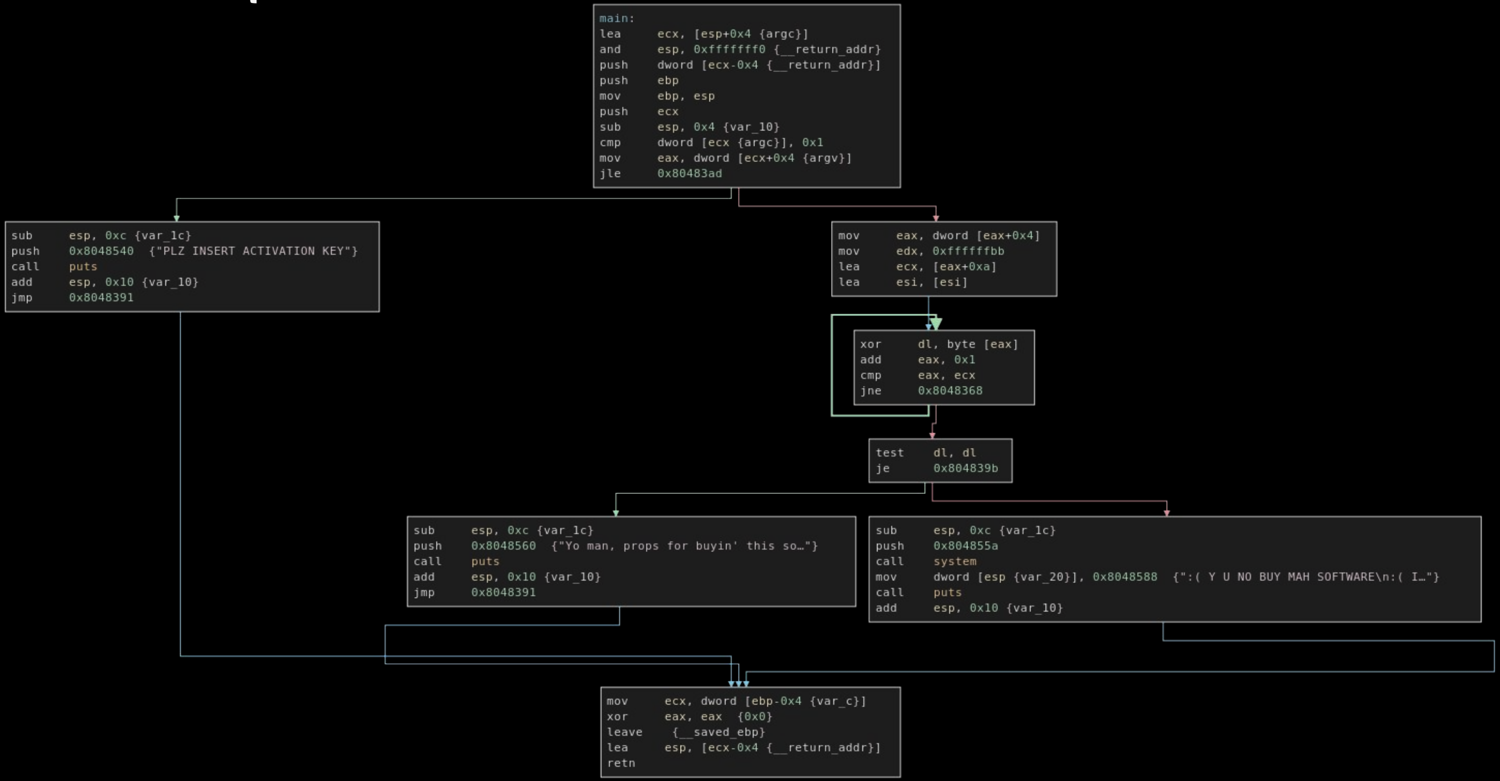
...a small introduction to reversing and binja

ASM - Linear View



```
main:
08048340  lea     ecx, [esp+0x4 {argc}]
08048344  and     esp, 0xfffffff0 {__return_addr}
08048347  push    dword [ecx-0x4 {__return_addr}]
0804834a  push    ebp
0804834b  mov     ebp, esp
0804834d  push    ecx
0804834e  sub     esp, 0x4 {var_10}
08048351  cmp     dword [ecx {argc}], 0x1
08048354  mov     eax, dword [ecx+0x4 {argv}]
08048357  jle     0x80483ad

08048359  mov     eax, dword [eax+0x4]
0804835c  mov     edx, 0xffffffbb
08048361  lea     ecx, [eax+0xa]
08048364  lea     esi, [esi]

08048368  xor     dl, byte [eax]
0804836a  add     eax, 0x1
0804836d  cmp     eax, ecx
0804836f  jne     0x8048368
```

ASM - Graph View (CFG)



```
main:
lea     ecx, [esp+0x4 {argc}]
and     esp, 0xfffffff0 {__return_addr}
push    dword [ecx-0x4 {__return_addr}]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 0x4 {var_10}
cmp     dword [ecx {argc}], 0x1
mov     eax, dword [ecx+0x4 {argv}]
jle     0x80483ad
```

```
sub     esp, 0xc {var_1c}
push    0x8048540  {"PLZ INSERT ACTIVATION KEY"}
call    puts
add     esp, 0x10 {var_10}
jmp     0x8048391
```

```
mov     eax, dword [eax+0x4]
mov     edx, 0xffffffbb
lea     ecx, [eax+0xa]
lea     esi, [esi]
```

```
xor     dl, byte [eax]
add     eax, 0x1
cmp     eax, ecx
jne     0x8048368
```

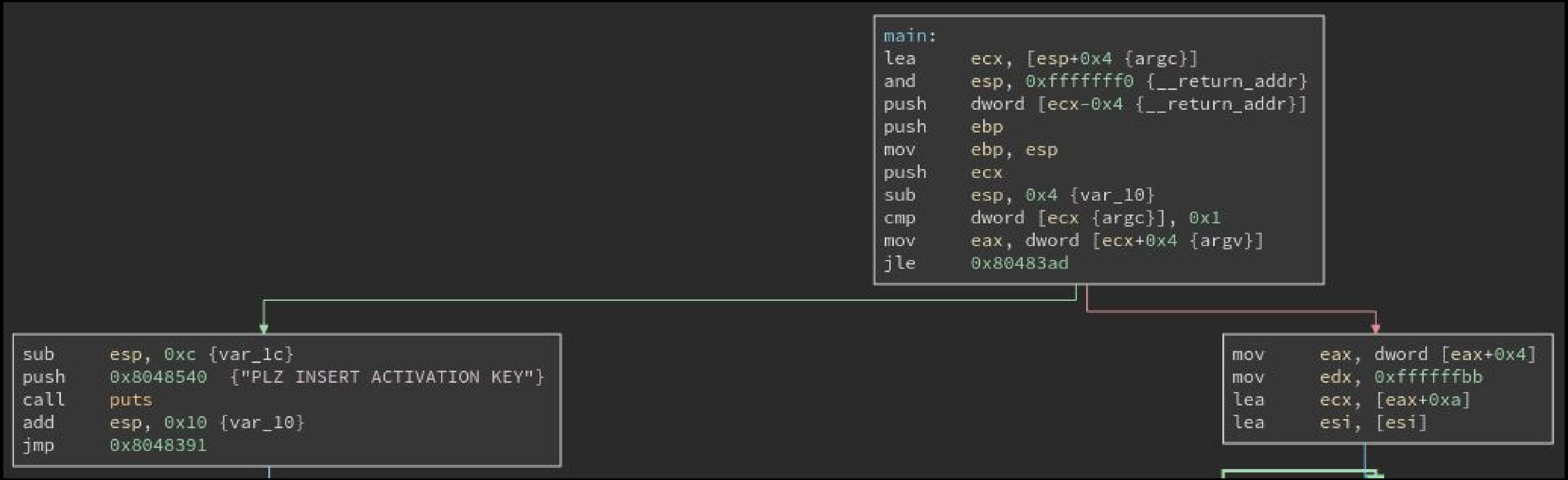
```
test    dl, dl
je      0x804839b
```

```
sub     esp, 0xc {var_1c}
push    0x8048560  {"Yo man, props for buyin' this so…"}
call    puts
add     esp, 0x10 {var_10}
jmp     0x8048391
```

```
sub     esp, 0xc {var_1c}
push    0x804855a
call    system
mov     dword [esp {var_20}], 0x8048588  {":( Y U NO BUY MAH SOFTWARE\n:( I…"}
call    puts
add     esp, 0x10 {var_10}
```

```
mov     ecx, dword [ebp-0x4 {var_c}]
xor     eax, eax  {0x0}
leave    {__saved_ebp}
lea     esp, [ecx-0x4 {__return_addr}]
retn
```

Graph View - IF

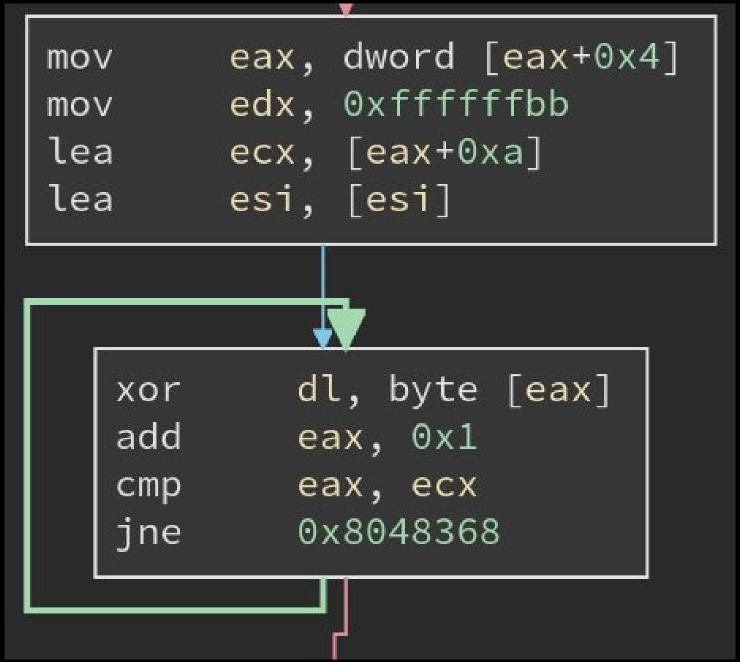


```
main:
lea     ecx, [esp+0x4 {argc}]
and     esp, 0xfffffff0 {__return_addr}
push    dword [ecx-0x4 {__return_addr}]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 0x4 {var_10}
cmp     dword [ecx {argc}], 0x1
mov     eax, dword [ecx+0x4 {argv}]
jle     0x80483ad
```

```
sub     esp, 0xc {var_1c}
push    0x8048540  {"PLZ INSERT ACTIVATION KEY"}
call    puts
add     esp, 0x10 {var_10}
jmp     0x8048391
```

```
mov     eax, dword [eax+0x4]
mov     edx, 0xffffffbb
lea     ecx, [eax+0xa]
lea     esi, [esi]
```

Graph View - Loop



```
mov     eax, dword [eax+0x4]
mov     edx, 0xffffffbb
lea     ecx, [eax+0xa]
lea     esi, [esi]
```

```
xor     dl, byte [eax]
add     eax, 0x1
cmp     eax, ecx
jne     0x8048368
```

Binja - Some shortcuts

g - Go to address / symbol

<spacebar> - Switch between linear and graph view

n - Rename symbol

y - Change symbol type

; - Comment (super useful!)

***** - Follow pointer

Welcome to ~~cracking~~ reversing 101

crackme v0

- You are given an expensive program
- But you don’t have any money
- You don’t need the license
- You can patch the license check so that every number is correct

DEMO

crackme v1

- Same program
- We don’t want to patch the binary
- We want to build a keygen

DEMO

crackme_remote

- Similar to crackme
- Running on spritz ctf
- Find a valid key to get the flag
- `CRACKME_FLAG=ASD ./crackme_remote`
- `nc 207.154.238.179 5222`

The End

Some pointers

- https://www.hex-rays.com/products/ida/index.shtml
- https://binary.ninja/
- http://www.radare.org/r/
- https://github.com/radareorg/cutter/releases
- http://hopperapp.com/ (only for Mac)
- https://github.com/wtsxDev/reverse-engineering
- https://azeria-labs.com/