Reverse Engineering

Table of Contents

What is reverse engineering?

- Working out what a program does
- Malware analysis
- Finding exploits
- Cracking software
- Cheating in games

Some terms

Static analysis Working out what a program does by looking at the files

Dynamic analysis Working out what a program does by running it and examining and modifying it's memory

Decompiler Attempts to turn machine code or bytecode into a higher level language

Disassembler Attempts to turn machine code or bytecode into an assembly language

Debugger Attaches to a process and allows you to examine and modify it's memory

Setup

- Update r2
  sudo apt update && sudo apt upgrade radare2
- Set up 32 bit
  dpkg -add-architecture i386
  sudo apt update && sudo apt install libc6:i386
  libc6-dev-i386

Table of Contents

Non native binaries

- Some languages don't compile to native executables.
- They compile to bytecode which is executed in a virtual machine (emulated CPU).
- This is usually done to make the executable run cross platform easily (without needing to recompile).
- These languages include Java, .net (clr), python (.pyc), etc...
- They tend to be easier, and you can usually get something close to the full original source code back, with variable names.
- The generated code is not always perfect though, and sometimes can't be recompiled.

Some tools I

Java

JD-GUI, cfr, jad, Krakatau, Procyon- decompilers
javasnoop - debugger
r2 - can read java bytecode, not much good documentation though

Android (java)

apktool - disassembles dalvik bytecode to a higher level format called smali
dex2jar - converts dalvik bytecode to jvm bytecode

.net

dnspy - decompiler

Some tools II

Python

Uncompyle6 - decompiler

Native

radare2 - disassembler and debuger
binaryninja - diassembler, paid, free demo
ida - disassembler and debugger, paid, free demo
gdb - debugger (disassembly isn't great)
objdump - disassembler
snowman - decompiler
hex rays decompiler - decompiler, paid

Obfuscation

To make these challenges harder the code can be obfuscated
Some examples of what can be done:

- Making the logic hard to follow
- Replacing variable and function names with nonsense
  I reconmend looking for meaningful strings assuming they haven't been encoded
  Also looking for functions which you recognise e.g. modexp, rsa, ...
- Having some code in base64 (or other), decoding it, and executing it
  Replace the eval function with a print statement

Table of Contents

Types I

Every variable must have a specific type.

Variable declaration and assignment

```
int a;
a = 10;
// can do both at the same time
int a = 10;
```

Types II

Basic types

- int usually 32 bits (4 bytes)
- long usually 64 bits (8 bytes)
- char usually 1 byte
- float usually 32 bits
- double usually 64 bits
- bool true/false usually 1 byte (technically a macro for _Bool, need to include stdbool.h)

sizeof

The sizeof operator takes a datatype and returns it's size in bytes.
It can also take an expression and return the size of the result.

```
sizeof(int);
```

Functions

Definitions

```
bool greaterThan10(int num) {
    return num > 10;
}
```

Call

```
bool result = greaterThan10(20); // true
```

Every program must have a main function (some exceptions - e.g. libraries)

```
int main(int argc, char** argv) {
}
```

Basic flow control I

If statement

```
if (1 == 2) {
    puts("true");
}
else {
    puts("false");
}
```

While loop

```
int i = 0;
while (i < 5) {
    printf("%d", i);
    i++;
}
```

Basic flow control II

For loop

```
for(int i=0; i<5; i++) {
    printf("%d", i)
}
```

Pointers

- A pointer stores a memory address
- Address-of operator & gets a memory address
- Dereference operator * gets the value at a memory address
- If a pointer doesn't point to anything you can set it to NULL
- On most systems NULL is 0 which isn't a valid address

```
int a = 5;
int* p = &a; // stores address of a in p
int b = *p // b == 5
a++; // a == 6, b == 5
b = *p; // b == 6
```

Why pointers

- Pass by reference
  We can't return arrays from functions (except some circumstances, e.g. array is on the heap)
- Pointer arithmetic
  e.g. array indexing

Arrays

- Need to specify a type and size
- Strings are stored as arrays of characters which end with a nullbyte.
- The vaiable holds a pointer to the start of the array
- Array elements are stored next to each other in memory

```
int someArray[5] = {1,2,3,4,5};
char someString[6] = "12345" // size is +1 for nullbyte
```

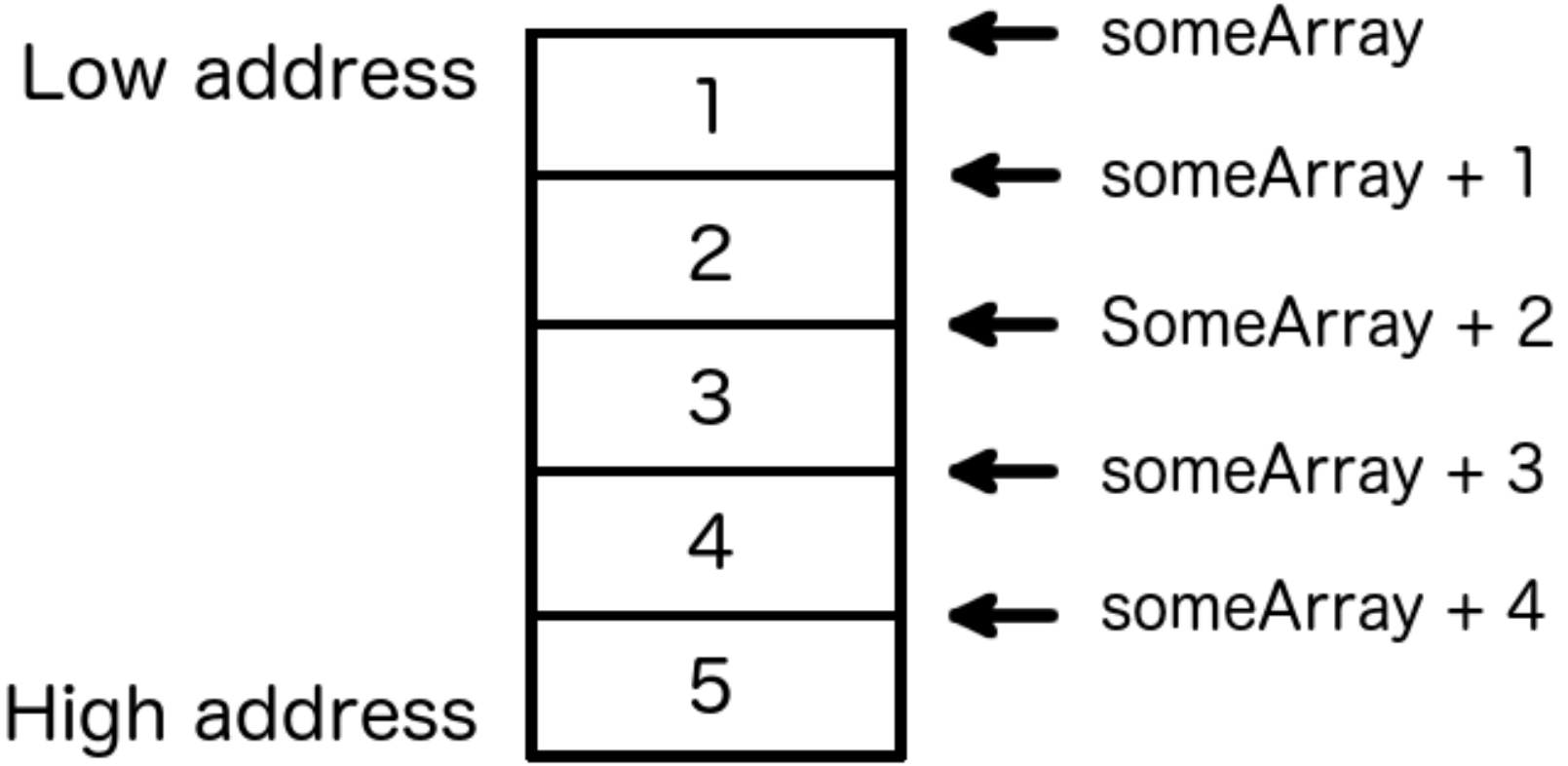
Pointer Arithmetic

Array indexing

```
int someArray[5] = {1,2,3,4,5};
printf("%d\n", someArray[2]); // is equivilent to
printf("%d\n", *(someArray+2));
```

- adding to a pointer actually adds a multiple of the type size.
- e.g. integers are (probably) 4 bytes, stack memory addresses also decrease towards 0, so `someArray+2` actually adds 2*4 = 8 to the memory address in someArray

Note: In ASM this would be 4, 8, 12, 16
Also note: Like many things, different compilers may do this differently.



Some string operations

Need to `#include <string.h>`

strlen

Gets the length of a string
Counts the number of characters before a nullbyte - O(n)

```
int len = strlen("some string");
```

strcmp

Compares strings
Returns 0 if equal

```
if (strcmp("abc", "def") == 0) {
    puts("equal");
}
```

IO I

Need to `#include <stdio.h>`

puts

Writes a string to stdout followed by a newline

```
puts("Hello, World!");
```

printf

Writes to stdout following a format string
Does not add a newline

```
printf("int: %d float: %f string: %s\n", 1, 2.0, "a string");
```

IO II

gets - DO NOT USE

Gets a line from stdin - line does not include newline character(s)

THIS IS INSECURE, USE FGETS

```
char myStr[20];
gets(myStr);
```

fgets

Gets either n-1 characters from stdin or a line (whichever is shorter)

Includes newline character(s)

```
char myStr[20];
fgets(mystr, 20, stdin);
fgets(mystr, sizeof(myStr), stdin); // better
```

A full program (series/1.c)

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

char password[] = "password";

bool testPassword(char* attempt) {
    return strcmp(attempt, password) == 0;
}

int main(int argc, char** argv) {
    printf("Enter password: ");

    char buff[200];
    fgets(buff, sizeof(buff), stdin); // get input
    buff[strlen(buff) - 1] = '\0'; // remove the newline

    if (testPassword(buff)) {
        puts("You win!");
    }
    else {
        puts("You fail");
    }
}
```

Command line arguments (codes/args.c)

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

char password[] = "password";

bool testPassword(char* attempt) {
    return strcmp(attempt, password) == 0;
}

int main(int argc, char** argv) {
    if (argc != 2) {
        puts("Give the password as a command line argument");
        exit(EXIT_FAILURE);
    }
    if (testPassword(argv[1])) { // 0 is name of program/command
        puts("You win!");
    }
    else {
        puts("You fail");
    }
}
```

Compiling

c is a compiled language - it needs to be compiled into a binary to execute

```
gcc mycode.c -o exename
```

If you use math.h

```
gcc mycode.c -lm -o execname
```

Some things I haven't included

- File IO (fopen, fclose, fprintf)
- Heap allocation (malloc, realloc, free)
- Function pointers
- Void pointers
- Undefined behaviour

Table of Contents

What is an executable

- Binaries are not just raw machine code
- They are stored in a format which says how to load them
- linux uses binaries in the Executable and Linkable Format (ELF)
- windows uses the portable executable format (PE)
- OSX uses the Mach object file format (Mach-O)

Loading

- The program on linux that links binaries is called ld
- When the kernel is told to run a file with a magic number that matches ELF (0x7F ELF) it gives it to ld

Some important things that need to happen

- Load segments of the binary into memory
- Load linked dynamic libraries (LD_LIBRARY_PATH)
- Call the main function

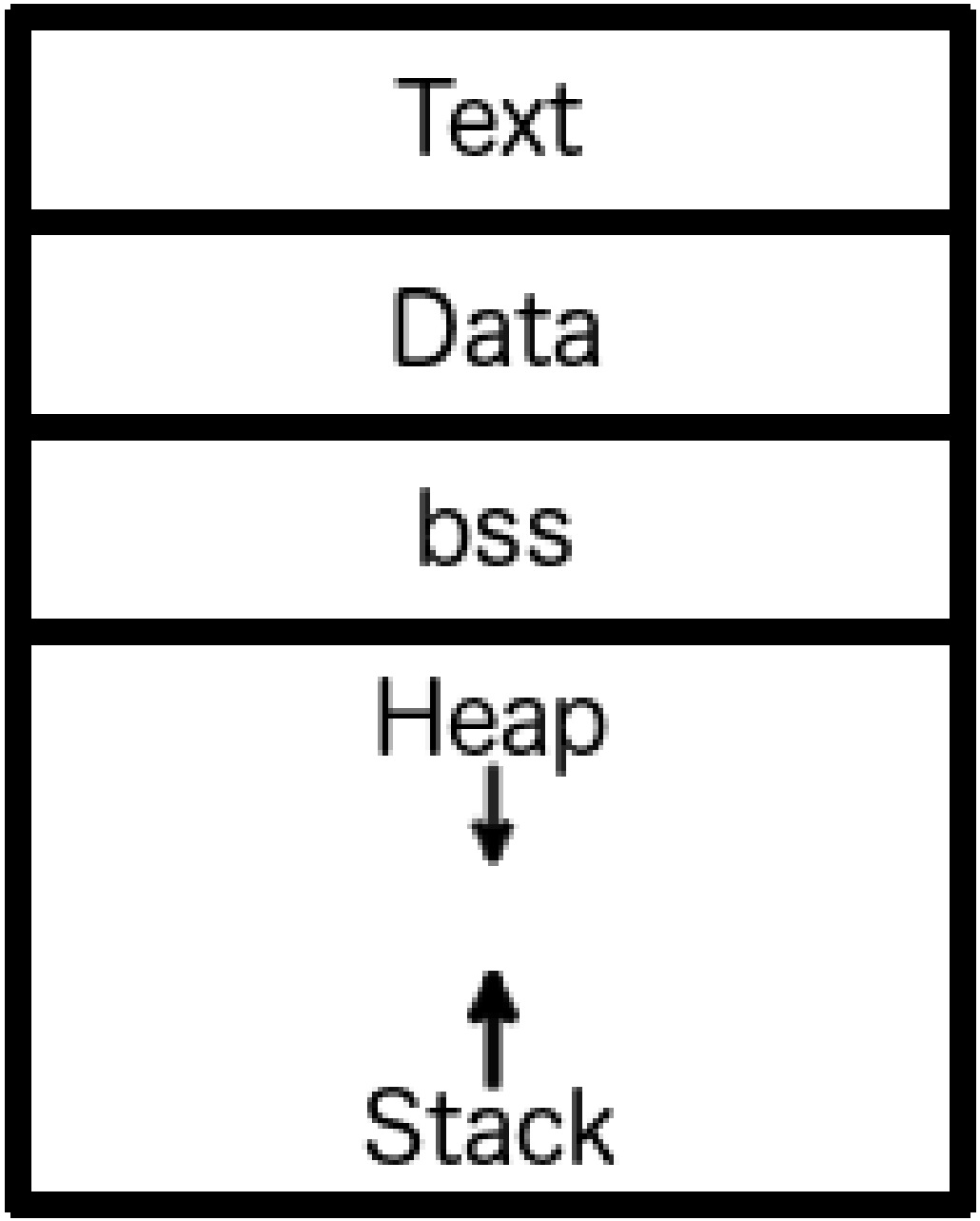
Some segments

- data - stores initialised global variables
- bss - stores uninitialised global variables (only the length is stored in the binary). Linux and windows initialise it to 0.
- text - stores the machine code

On my mac (Apple LLVM version 9.1.0 (clang-902.0.39.2)) string literals in functions seem to be stored in a data section. - a pointer is put on the stack
On kali (gcc version 7.2.0 (Debian 7.2.0-19)) they seem to be stored in the code as integers (8 chars in a 64 bit integer) and moved onto the stack.

Other memory locations in a running program

- stack - local variables, stack frames
- heap - malloc, realloc, free



Low address
000...

High address
fff...

```c
int a = 1; // data
int b; // bss
char str[5]; // bss

int main(int argc, char **argv) {
    char str2[10]; // stack
    char str3[] = "abcdef"; // data or stack?
    int i = 0; // stack
    for (;i<4;i++) {
        char c = str3[i]; // stack
    }
}
```

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

char password[] = "password";

bool testPassword(char* attempt) {
    return strcmp(attempt, password) == 0;
}

int main(int argc, char** argv) {
    printf("Enter password: ");

    char buff[200];
    fgets(buff, sizeof(buff), stdin); // get input
    buff[strlen(buff) - 1] = '\0'; // remove the newline

    if (testPassword(buff)) {
        puts("You win!");
    }
    else {
        puts("You fail");
    }
}
```

Strings

In this code the password string is stored in the binary in the data section. We can just find it with strings

```
root@kali:/m/s/c/R/codes > strings 1
/lib64/ld-linux-x86-64.so.2
libc.so.6
puts
stdin
printf
fgets
strlen
...
%b
=I
5B
AWAVI
AUATL
[]A\A]A^A_
Enter password:
You win!
You fail
;*3$"
password
...
```

Table of Contents

ltrace

ltrace prints all the calls to dynamically linked functions

```
root@kali:~ > ltrace ./1
printf("Enter password: ")
fgets(Enter password:
"\n", 200, 0x7fb1f3b64a00)
strlen("\n")
strcmp("", "password")
puts("You fail"You fail
)
+++ exited (status 0) +++
```

strace

strace prints all the system calls made by the program

```
root@kali:~ > strace ./1
execve("./1", ["./1"], 0x7ffc177e9330 /* 41 vars */) = 0
brk(NULL)                                       = 0x5597eb436000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=146319, ...}) = 0
mmap(NULL, 146319, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f780d532000
close(3)                                        = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\33\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1800248, ...}) = 0
...
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
write(1, "Enter password: ", 16Enter password: )        = 16
read(0,
"\n", 1024)                       = 1
write(1, "You fail\n", 9You fail
)               = 9
exit_group(0)                                   = ?
+++ exited with 0 +++
```

```
bool testPassword(char* attempt) {
    if (strlen(attempt) != 8)
        return false;
    if (attempt[0] != 'p')
        return false;
    if (attempt[1] != 'a')
        return false;
    if (attempt[2] != 's')
        return false;
    if (attempt[3] != 's')
        return false;
    if (attempt[4] != 'w')
        return false;
    if (attempt[5] != 'o')
        return false;
    if (attempt[6] != 'r')
        return false;
    if (attempt[7] != 'd')
        return false;
    return true;
}
```

Windows

On windows you can use procmon from sysinternals as strace
For ltrace I think I've used API monitor from rohitab.com

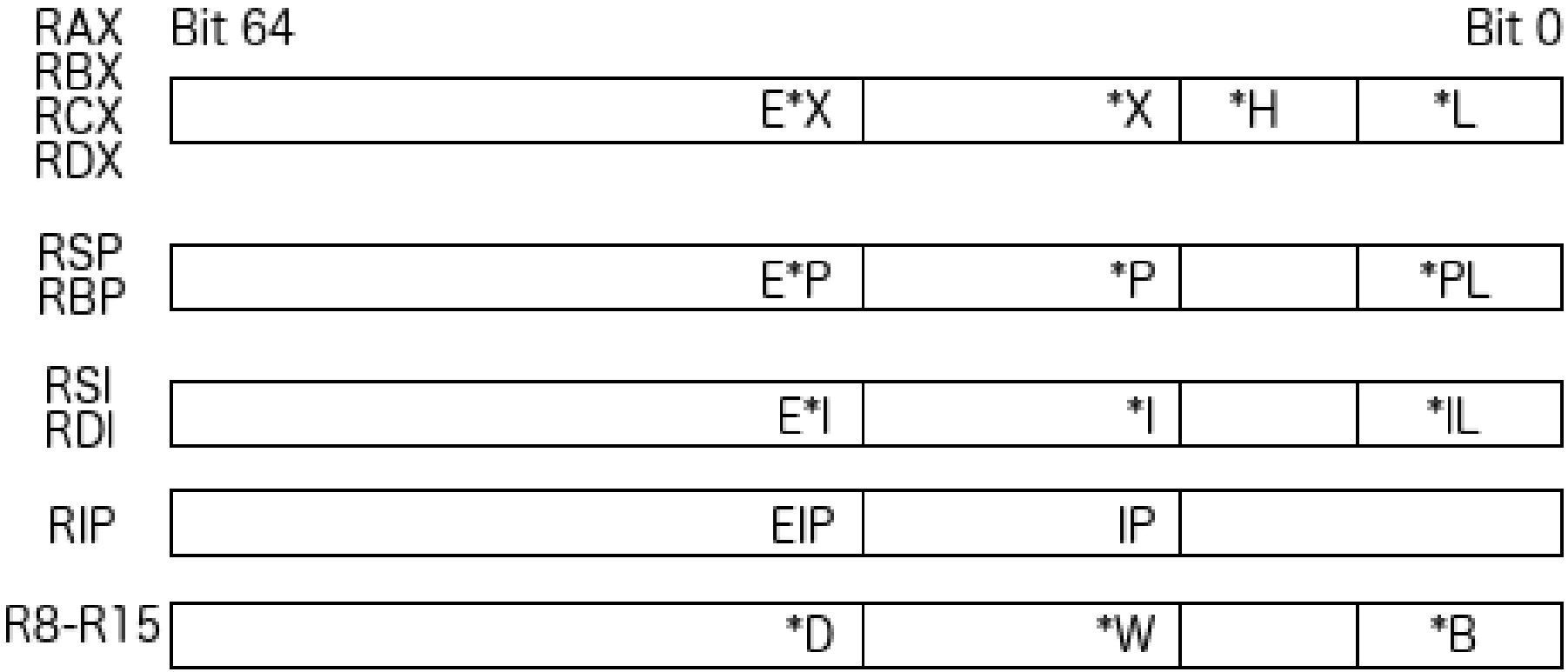
Table of Contents

x86

- Used by Intel and AMD
- Complex instruction set computer (CISC)
- Little endian architecture
  least significant byte first - ”reverse” order
  i.e. 16909060
  In big endian: 0x01020304
  In little endian: 0x04030201
- Instructions are variable length

Registers

- 64 bit registers extend 32 bit which extend 16 bit which extend 8 bit
- 64 bit also added new registers r8 - r15
- Most registers are general purpose and can be used for anything
- rip is the instruction pointer, it points to the current instruction being executed
- rbp is the stack base pointer, it points the start of the current stack frame
- rsp is the stack pointer, it points to the top of the stack
- the FLAGS/EFLAGS/RFLAGS register holds information about calculations


RAX
RBX
RCX
RDX
Bit 64
Bit 0
E*X
*X
*H
*L
RSP
RBP
E*P
*P
*PL
RSI
RDI
E*I
*I
*IL
RIP
EIP
IP
R8-R15
*D
*W
*B

Data types

Bytes are bytes, the only difference between a int, char, bool, etc... is the interpretation.
The size of a word in x86 assembly is 2 bytes (16 bits) (The actual hardware word size has changed, but ”word” in assembly still refers to 2 bytes for backwards compatibility)

- byte - 1 byte
- word - 2 bytes
- dword - double word, 4 bytes
- qword - quad word, 8 bytes

Basic syntax

- There are a few different syntaxes. I will use Intel's syntax
- `<instruction> <operand1>, <operand2>, <operand3>`
- The result of the expression is stored in the first argument

```
add rax, 1
```

- A literal value in an operand (like the above 1) is called an immediate value
- The number of arguments can vary. e.g. no args

```
nop
```

mov I

- To do things with data we need to move it into registers
- The mov instruction moves (actually copies) data around

```
mov rax, 1 ; put 1 in rax
mov rax, rbx ; copy what is in rbx into rax
```

Dereferencing pointers

- We can dereference pointers with brackets
- Need to specify the size

```
; copy a byte at the top of the stack into rax
mov rax, byte [esp]
; copy dword from ebx to the top of the stack
mov dword [esp], ebx
```

mov II

Array indexing

Assuming start of array is at rbp - 40 and holds 4 byte ints

```
mov rax, 2 ; index
mov rax, dword [rbp + rax*4 - 40]
```

Problem - moving smaller data into a larger register i.e. Can't do

```
mov rax, ebx
```

Signed integers

- We use a format called two's complement
- store negative numbers as $2^N - x$ where N is the number of bits and x is the number to negate
- e.g. for -3 as a 8 bit integer $2^8 - 3$
- This is so that we can just add negative numbers normally
- for -3 + 4 adding the first 3 in the 4 overflows the -3 to 0. The last 1 in the 4 makes the result 1
- An equivalent way to calculate 2s complement is to invert all the bits and add 1
- the most significant bit of a signed integer is the sign bit, if it's 1 the number is negative

movsx and movzx

- To extend a smaller number (e.g. 32 bits to 64 bits) we need to know if it's signed or unsigned
- If it's signed we should extend it with the sign bit
- If it's unsigned we should extend it with 0
- movsx = move with sign extend
- movzx = move with zero extend

lea

Moves the address of data into a register

```
mov rax, 2 ; index
; move thing at arr[2] into rbx
mov rbx, dword [rbp + rax*4 - 40]
; move the address of arr[2] into rbx
lea rbx, [rbp + rax*4 - 40]
```

Effectively just does the calculation in the brackets, so it can be used for maths

Some basic maths

```
inc rax ; increment rax
dec rax ; decrement rax
neg rax ; negate rax (2s complement)
add rax, rbx ; add rbx into rax
sub rax, rbx ; subtract rbx from rax
xor rax, rbx ; xor rax and rbx
xor rax, rax ; set rax to 0
and rax, rbx ; and rax and rbx
or rax, rbx
...
```

Unsigned Multiplication I

Multiplication is a bit more complicated as the result can be bigger than the register.
How it works depends on the size of the operand

```
mul bl ; a byte
```

Does al * bl and stores the result in ax (note: ax is al and ah)

```
mul bx ; a word
```

Does ax * bx and stores the result in dx:ax (low part in ax and high part in dx)

Unsigned Multiplication II

```
mul ebx ; a dword
```

Does eax * ebx and stores the result in edx:eax

```
mul rbx; a qword
```

Does rax*rbx and stores the result in rdx:rax
So in most cases it multiplies the operand with the appropriate sized portion of rax and stores the result in same sized portions of rdx and rax.

Signed Multiplication

- imul instead of mul
- It can work like mul, but it also has another 2 addressing modes
- One lets you specify the destination and the other specifies the destination and the second source

Unsigned Division

- The div instruction does not create a floating point number.
- It calculates the quotient and remainder.
- The dividend (thing being divided) has a size twice as large as the divisor (thing divided by)
- 8 bit divisor
  dividend is in AX, quotient is put in Al, remainder in AH
- 16 bit divisor
  dividend is in edx:eax, quotient put in AX, remainder in DX
- 32 bit divisor
  dividend is in edx:eax, quotient put in eax, remainder in edx
- 64bit divisor
  dividend in rdx:rax, quotient put in rax, remainder in edx

Flow control I

- x86 does not have if statements and loops, it has conditional (and unconditional) jumps instead.
- When writing assembly we jump to labels. A disassembler will show memory address instead though.

Unconditional jump

```
label:
    add rax 1
jmp label
```

Note: this example is an infinite loop

Flow control II

The cmp instruction compares two numbers and sets RFLAGS/EFLAGS based on whether the numbers are higher, lower, equal, ... It is usually used before a conditional jump

Conditional jump

```
mov rax, 0
mov rbx, 1
cmp rax, rbx
je label ; jump if equal
...
label:
...
```

Flow control III

Some common conditional jumps

je - equal
jne - not equal
jl - less than
jg - greater than
jle - less than or equal to
jge - greater than or equal to
note: for signed comparisons b (below) is used instead of l and a (above) is used instead of g

If statement I

Note: I'm using .1., .2., etc... as placeholders for code

If

```
if (1 == 2) {
    .1.
}
.2.
```

```
mov rax, 1
; cmp doesn't take 2 immediates
cmp rax, 2
jne false
.1.
false:
.2.
```

If statement II

If - Else

```
if (1 == 2) {
    .1.
} else {
    .2.
}
.3.
```

```
mov rax, 1
cmp rax, 2
jne false
.1.
jmp end
false:
.2.
end:
.3.
```

Loop

```
int i = 0;
while (i < 5) {
    .1.
    i++
}
.2.
```

```
mov rax, 0
jmp test
start:
    .1.
    inc rax
test:
    cmp rax, 5
    jb start
.2.
```

Functions I

Need to deal with a few things

1. Passing arguments
2. Jumping to the function
3. Storing local variables
4. Returning a value
5. Going back where we came from
6. Also handling the fact that registers will change in the function

Let's deal with 2 and 5 first

- We handle this by storing information on the stack.

Functions II

- When calling the function it pushes the position of the **next** instruction onto the stack before jumping to it
- When returning it pops it back into the instruction pointer
- If we allocate stack space for local variables we also need to be able to restore the stack pointers
- That can be done by saving the old ebp and using the leave instruction at the end of the function

Functions III

call, ret and leave

`call myfunc`	`push eip` `jmp myfunc`
`ret`	`pop eip`
`leave`	`mov ebp, esp` `pop ebp`

Functions IV

- For local variables we will allocate some stack space and store them on the stack
- If we are going to change the stack pointer we also need to either save it on the stack or increment it again later
- We refer to the local variables using ebp and keep ebp constant. esp can change if we push more stuff on the stack.

Functions V

A basic function

```
myfunc:
    push ebp
    mov epb, esp
    sub esp 12 ; 3 ints
    ...
    pop ebp
    ret

call myfunc
```

There are different conventions for handling arguments and return values

cdecl I

- Used when compiling 32 bit programs with gcc, is common with 32 bit compilers
- Arguments are pushed onto the stack in right to left order
- Return value is put in eax (usually - different implementations)
- eax,ecx and edx should be saved by the caller (if needed) others should be saved by the callee
- The calling function cleans up the arguments on the stack

cdecl II

cdecl

```
int addNums(int a, int b, int c) {
    int l1 = 2;
    int l2 = 3;
    return a+b+c + l1 + l2;
}
addNums(1,2,3);
```

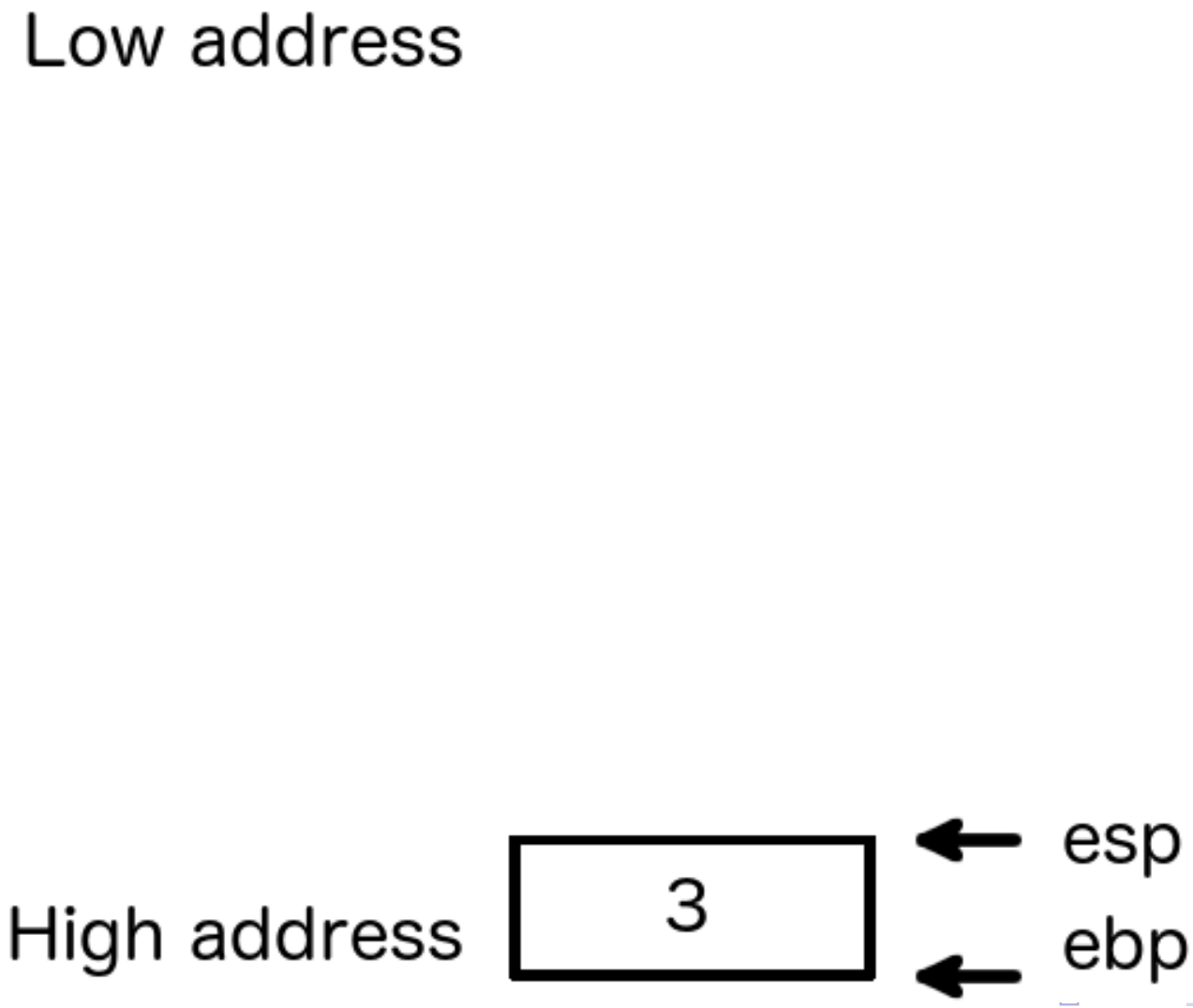
```
addNums:
    push ebp
    move ebp, esp
    sub esp, 8
    mov dword [ebp - 4], 2
    mov dword [ebp - 8], 3
    mov edx, dword [ebp + 8]
    mov eax, dword [ebp + 12]
    add edx, eax
    mov eax, dword [ebp + 16]
    add edx, eax
    mov eax, dword [ebp - 4]
    add edx, eax
    mov eax, dword [ebp - 8]
    add eax, edx
    leave
    ret

push 3
push 2
push 1
call addNums
```

Low address

High address

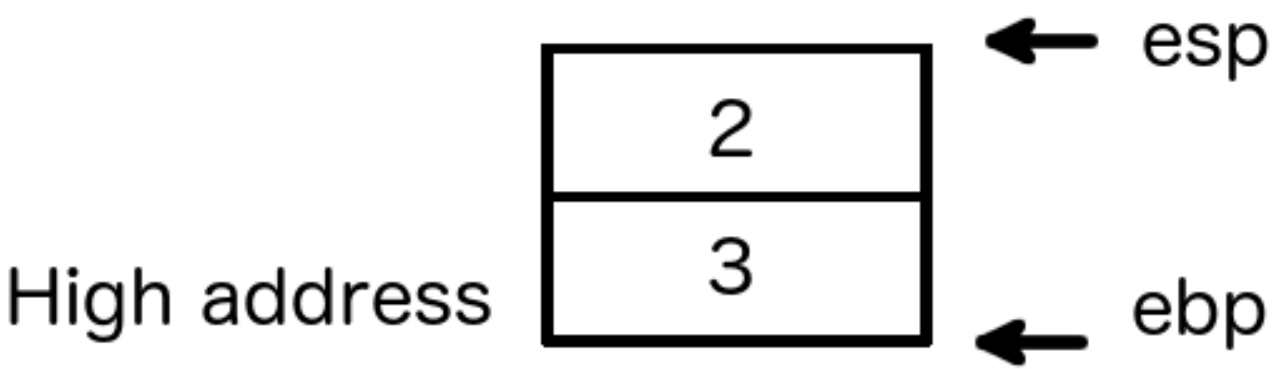
esp, ebp

push 3



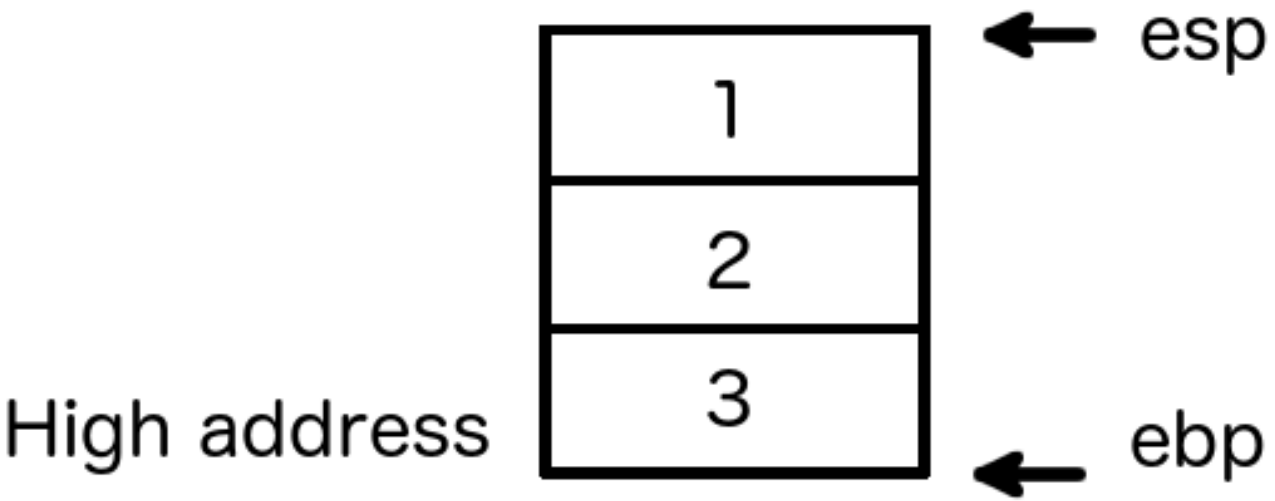
push 2

Low address



push 1

Low address

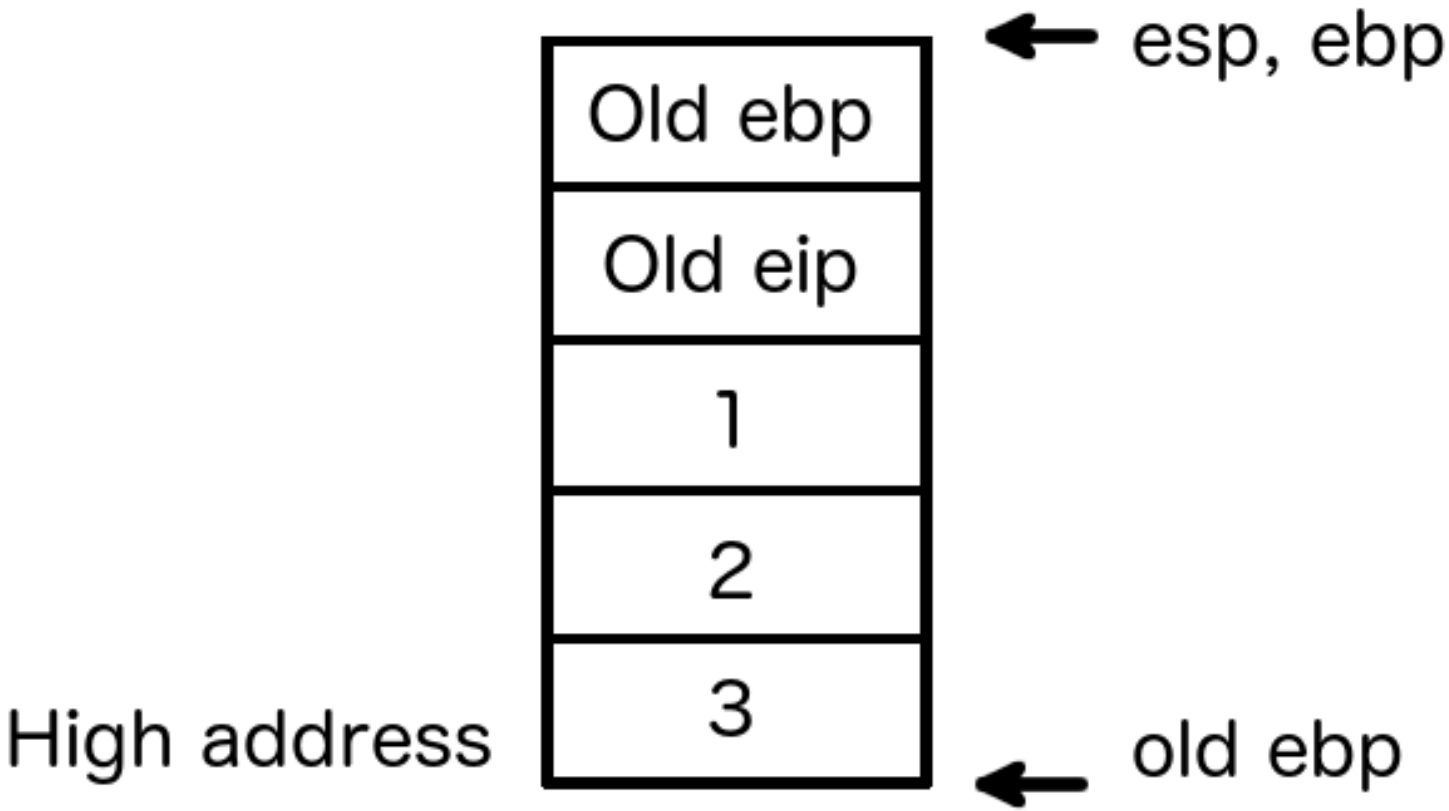


call addNums

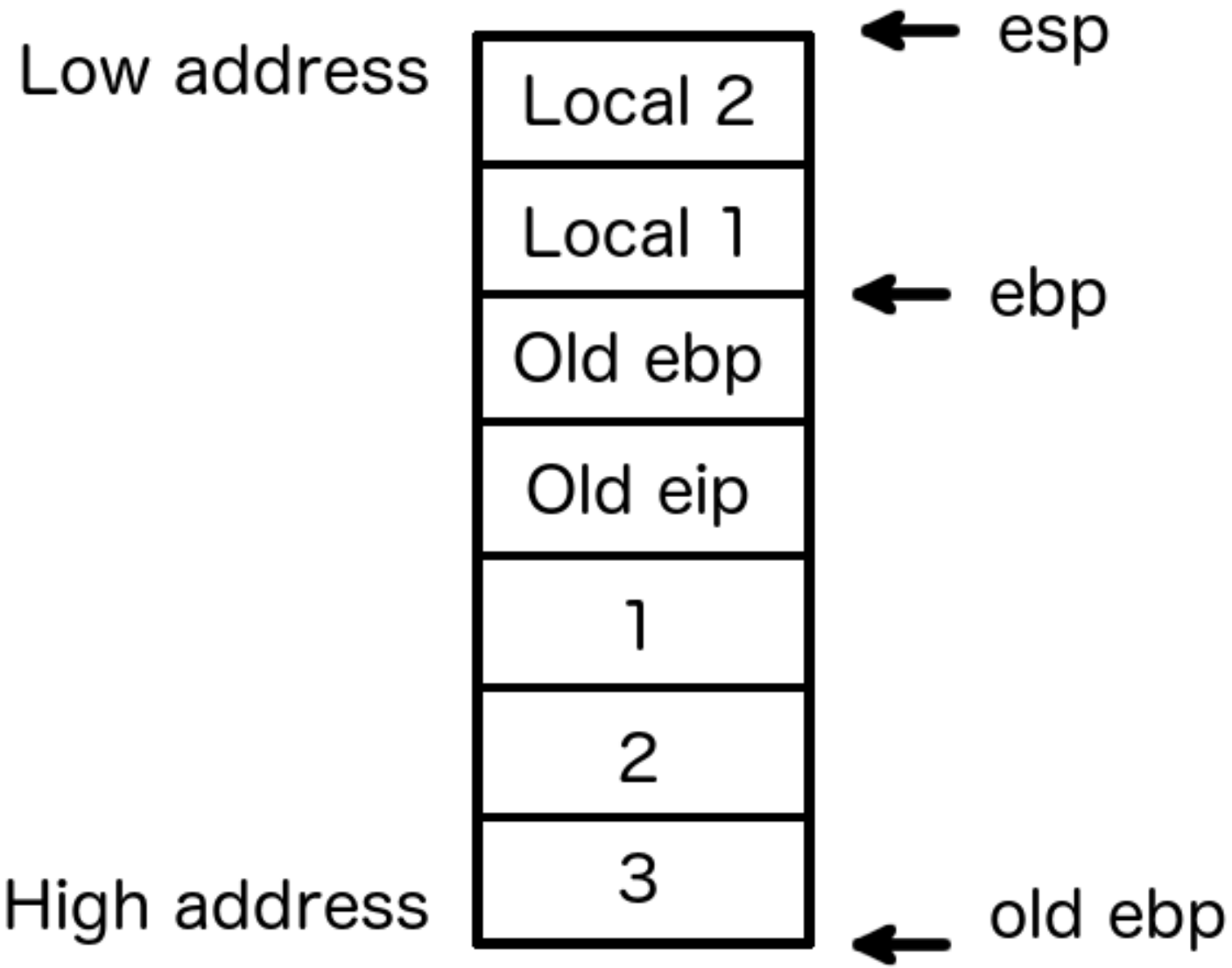
Low address

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

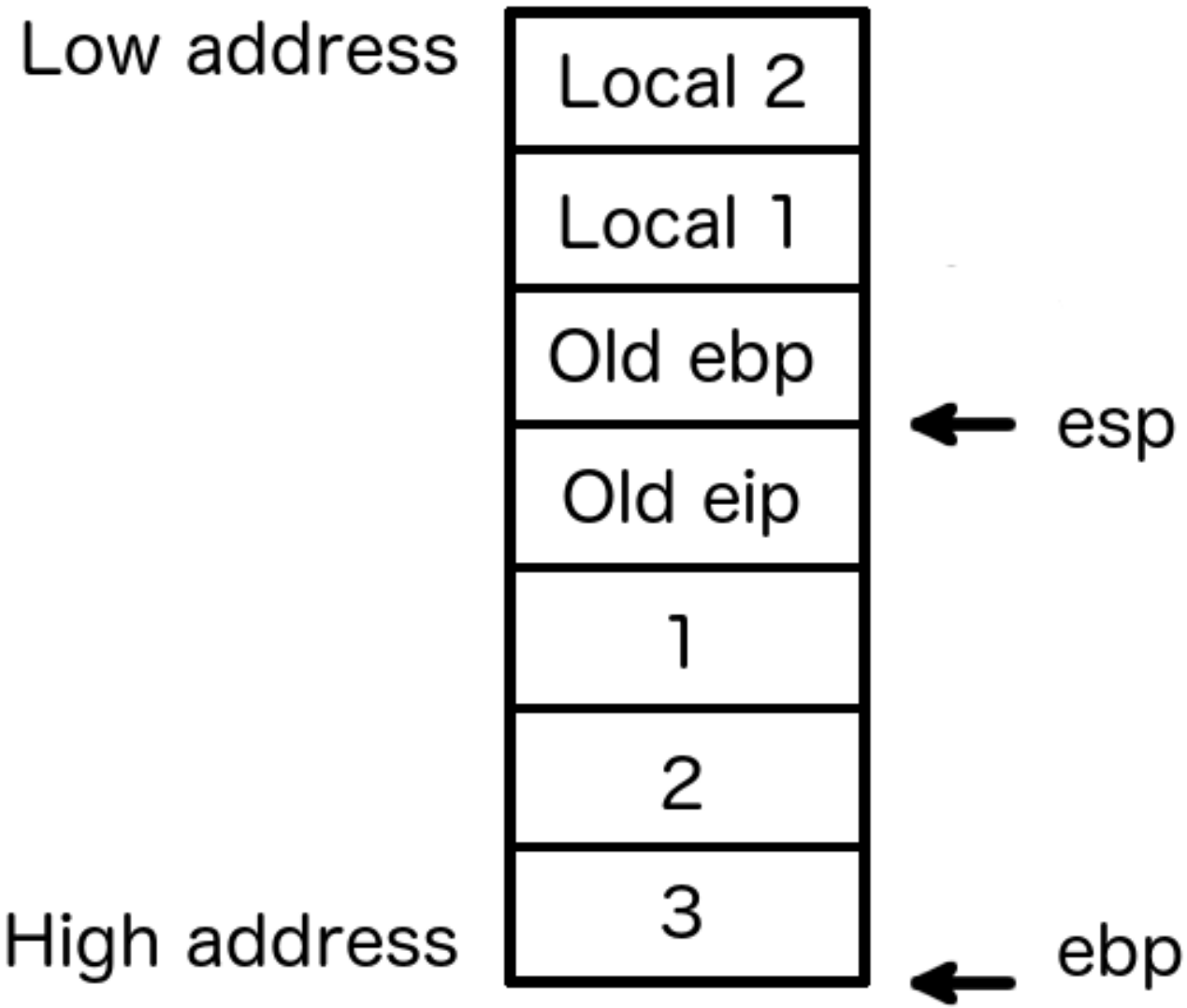
Low address



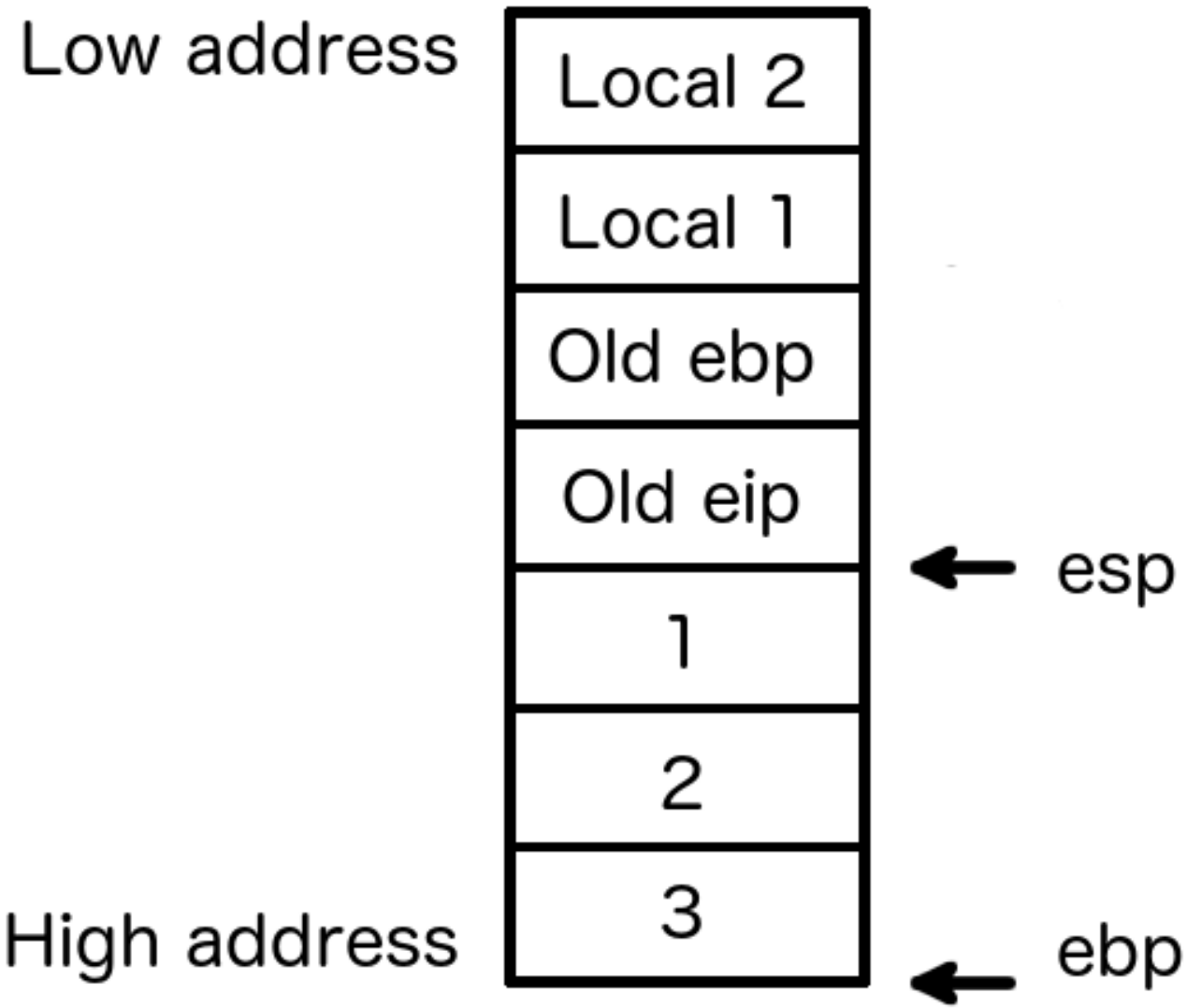
sub esp 8


Low address
Local 2
Local 1
Old ebp
Old eip
1
2
3
High address
esp
ebp
old ebp

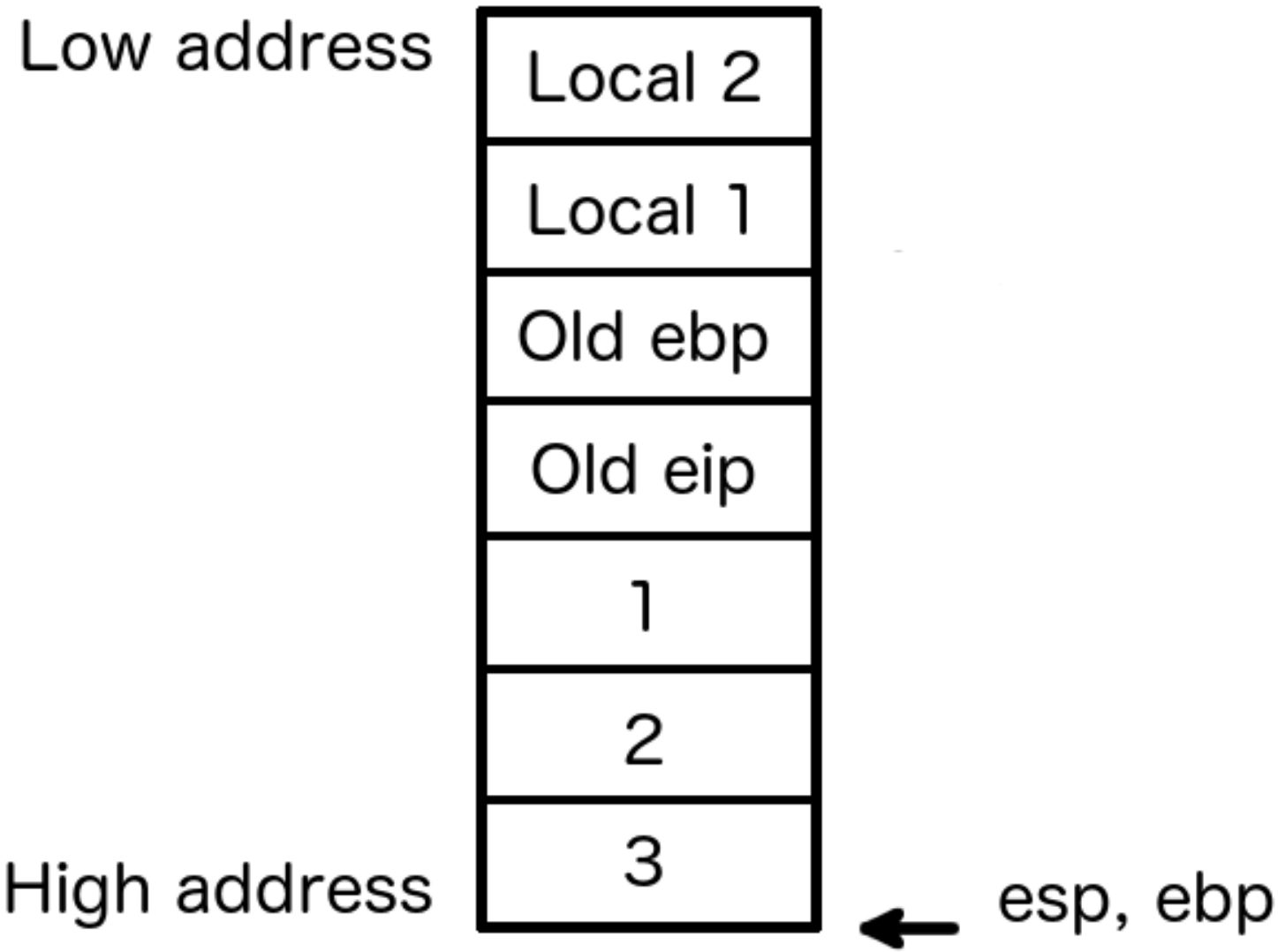
leave (mov esp, ebp
pop ebp)



ret


Low address
Local 2
Local 1
Old ebp
Old eip
esp
1
2
3
High address
ebp

add esp, 12


Low address
Local 2
Local 1
Old ebp
Old eip
1
2
3
High address
esp, ebp

System V AMD ABI

- 64 bit calling convention
- Used by linux, freeBSD, MacOS
- Registers are faster than stack memory
- The 64 bit architecture has more registers
- First 6 arguments are stored in rdi, rsi, rdx, rcx, r8 and r9
- Others are put on the stack (RTL)
- Return value in rax
- If the callee uses the registers rbx, rbp, and r12-r15 it needs to save and restore them before returning
- Others need to be saved by the caller

Windows function calls

- for 32 bit windows mostly uses stdcall which is similar to cdecl.
  The main difference is that the callee removes the arguements from the stack
  This can be done by giving ret an operand e.g. ret 4
- fastcall and vectorcall are also used sometimes. They put some of the arguments into registers.
- Microsoft x64 calling convention
  - First four args in rcx, rdx, r8 and r9. Others on stack (RTL)
  - caller must **always** allocate 32 bytes of ”shadow space” on the stack before calling the function
  - rax, rcx, rdx, r8, r9, r10, r11 are saved by the caller
  - rbx, rbp, rdi, rsi, rsp, r12, r13, r14, r15 are saved by the callee

Stack alignment

- Variables on the stack may not be directly after each other in memory
- The compiler may put padding in between some variables to align them on certain boundaries
- How it's aligned can depend on compiler version. I think it can also be set to suit your hardware
- This also happens in structs

Syscalls I

- External resources, e.g. files, stdin, stdout, etc... are managed by the kernel
- To use them we need to call a function in the kernel
- `int 80` is ~~a rapper~~ an older way of making syscalls
  The syscall number is put in eax and the args into ebx,ecx,edx,esi,edi
  Then `int 80`
  The return value is put in rax
  There are faster methods
- for 32 bit you can use sysenter
- for 64 bit you can use syscall

Syscalls II

64 bit linux syscall numbers:
https://filippo.io/linux-syscall-table/
32 bit linux syscall numbers:
https://syscalls.kernelgrok.com/
C programs don't usually make syscalls directly, the standard library wraps them.

Table of Contents

radare2

- collection of tools
  - rax2 - expression evaluator
  - rabin2 - shows information about binaries
  - radiff2 - diffs 2 binary files
  - rasm2 - assembler and disassembler
  - ragg2 - compiler
  - rahash2 - hash each block in a file
  - radare2 - main binary that combines all the others
- the -d commandline arg enables debugging
- the -w commandline arg allows binary patching
- commands use memonics
- use the ? command to get help
- ? also evaluates expressions

radare2 commands I

a - analysis

- aa - analyse all
- aaa - does aa and autonames functions
- af - analyse functions
  - af ([name]) ([addr]) - analyse functions starting at addr
  - afvn [old_name] [new_name] - rename an argument/variable (analyse function variable name)
  - afl - list functions (analyse function list)
- ax - list xrefs
- axt - find references to address

radare2 commands II

d - debugging

- db - breakpoints
  - db - list breakpoints
  - db <addr> - add breakpoint ad addr
  - db -<addr> - remove breakpoint at addr
  - dbd <addr> - disable breakpoint
  - dbe <addr> - enable breakpoint
  - dbs <addr> - Toggle breakpoint
- dc - continue
- ds - step
- ds <num> - step number of instructions
- dso - step over
- dsf - step to end of frame (step out)
- dr - print registers
- dr <register>=<val> - set a register

Visual mode

- V (capital) enters visual mode
- Pressing ? gives some help
- You can change view with p and P
- You can run a command with :
- q quits
- Pressing V again opens the control flow graph (CFG)

Reverse Debugging

- Radare 2 has a reversible debugger, you can revert to a previous state of the program
- use dts+ to save the current state and start recording
- dsb - step back
- dcb - continue back

RE tips I

- Understand xor (probably other binary operations as well)
- Learn to recognize hex numbers in the printable ASCII range
- Step through code you don't understand and look at the changes to the registers and stack
- Maybe do the same to code you do understand to make sure you are correct

RE tips II

- If the password is assembled completely before being compared you can easily pull it from memory
- Try searching for the failure message - the checking code is often close (or has an xref close)
- If you can find the success/failure messages try working backwards from there
- If you can't find an exact password try to reduce the search space and bruteforce
- Write it down

Table of Contents

Anti reversing

- Remove helpful information from the binary
- Make the disassembly a mess
- Stop debuggers attaching to the process
- Stop tools being able to open the file

Stripping the binary (stripped)

- We can tell gcc not to include information about function and variable names
- gcc -s -fvisbility=hidden

Static linking (static)

- The binary still contains information about libraries it is linked to and what it calls in them
- This is needed for dynamic linking
- If we statically link the binary we can remove it
- glibc (GNU's c library implementation) does not support static linking
- We can use musl instead
- sudo apt install musl-tools musl-dev
- musl-gcc -s -fvisibility=hidden -static

Removing strings (nostrings)

- We (badly) hid the password string earlier so it couldn't be found with strings
- But we didn't hide the success and failure messages
- They can give information about where the checking function may be
- Let's just use a simple xor "encryption"
- This is bad though - it's quick to brute force so someone could compute all 255 versions of the binary and strings them all
- Use something closer to a real encryption algorithm

Fake main function (fakemain)

- The program does not actually start executing at main
- Some other stuff is done first to link dynamic libraries and handle passing command line args to main
- gcc allows you to create a function with __attribute__((constructor))
- This function will be called before main
- Put the real code in here and exit before main is called
- Put fake code in main

Stopping ptrace (ptrace)

- ptrace is the systemcall debuggers use to interact with the bianry
- the linux kernel only allows one process to debug it with ptrace
- so we can just make it ptrace itself
- see: man ptrace
- In windows you can check if the process is being debugged. IsDebuggerPresent is common
  https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx
  https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/determining-if-a-debugger-is-attached

SIGTRAP (sigtrap)

- Debuggers use the SIGTRAP signal for breakpoints
- They don't pass the signal on to to program they are debugging
- So we can put important code in a signal handler and SIGTRAP the binary
- The important code will never be called as the debugger intercepted the signal

Jumping into the middle of instructions (instructions)

- The disassembly view can't show overlapping instructions
- This is not normally an issue as instructions don't normally overlap
- But this can be valid code
- See code for an example

Table of Contents

Resources

- These are all **free**
- RE4B
- Intel developer manuals.
  You probably want part 2 of the 4 volume set
- Programming Linux: Anti-Reversing Techniques
  Some of the code samples with this presentation are based on code from this book
- Radare2 Book
- Wikibooks x86 assembly
- Wikibooks x86 disassembly
- NoREPls
- Malware Reversing - Burpsuite Keygen
- crackmes.one