4RA7L8M4

Binary analysis and application security

David Oswald and Eike Ritter

Introduction to Computer Security,

Based on a course by Tom Chothia

Data can be Code

- Lots of the attacks we have seen trick a program into accept data that is really code, e.g.,
  - SQL injection
  - XSS
  - Buffer overflow (next lecture)
- This is a very common way to attack systems.

Code is Data

- In this lecture we are going to do the opposite.

- Executable code can be written and edited, just like an other document.

- Ultimately, an attacker/analyst can do ***anything*** they want with a program.

Introduction

- Compiled code is really just data…
  … which can be edited and inspected.
- By examining low-level code, protections can be removed and the function of programs altered.
- Good protection tends to slow down this process, not stop it.

This lecture

- Java Byte code:
  - High level overview
  - Inspecting the byte code
  - Decompiling back to Java

- x86 assembly:
  - High level overview
  - Inspecting and altering binaries in IDA

Learning Objectives

- I ***don't*** want you to memorise assembly, or Java byte code commands.
- I ***do*** want you to have a understanding of how machine code works, and is compiled.
- I ***do*** want you to know what buffer overflow attacks are and how they work.
- I ***do*** want you to understand that an attacker can view and edit assembly.

Reasons For Reverse Engineering

- Analyse malware
- Debug memory errors
- Analyse legacy code
- Security audit

Live-Demo

"Anything that can go wrong, will go wrong"

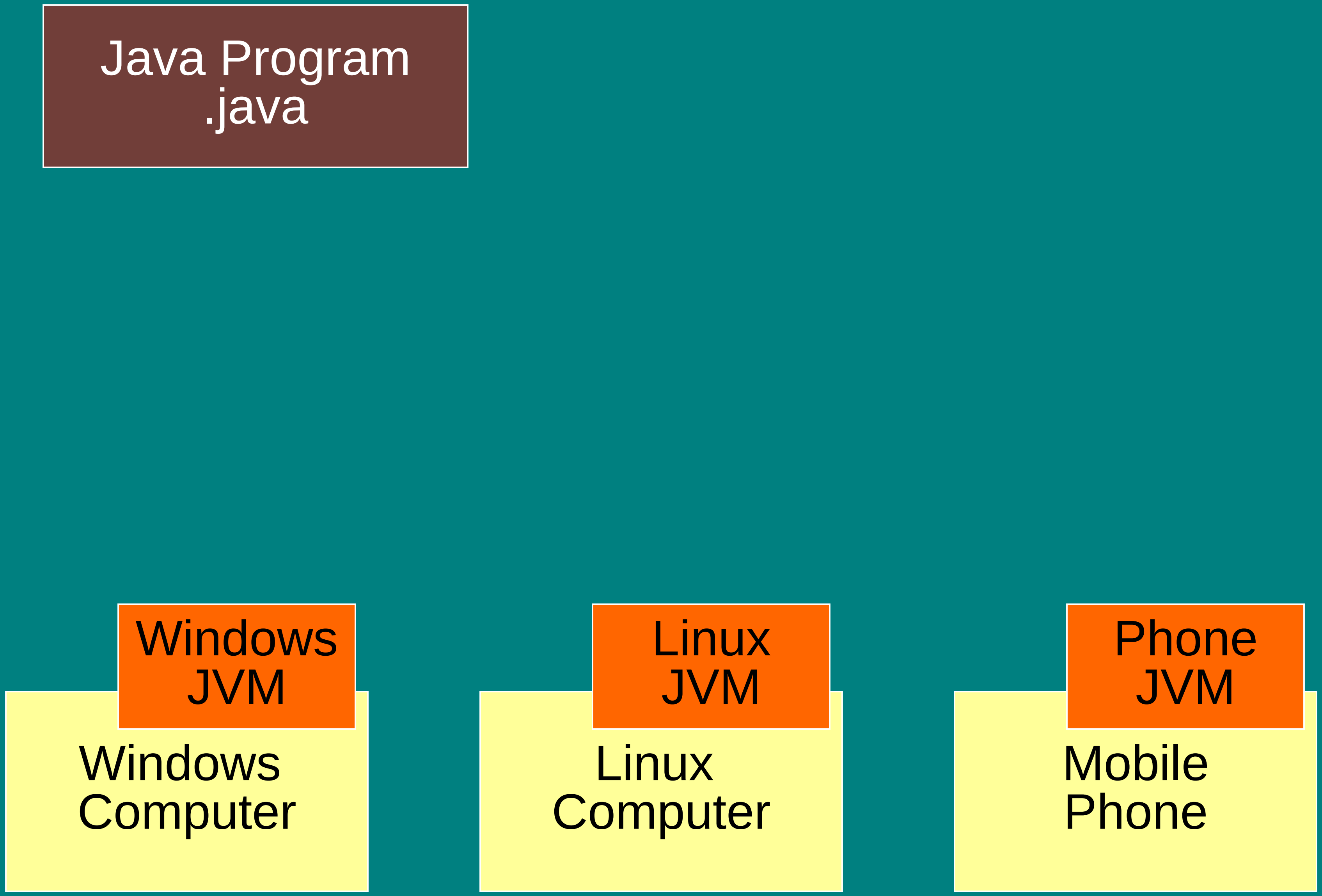
A password checker in Java

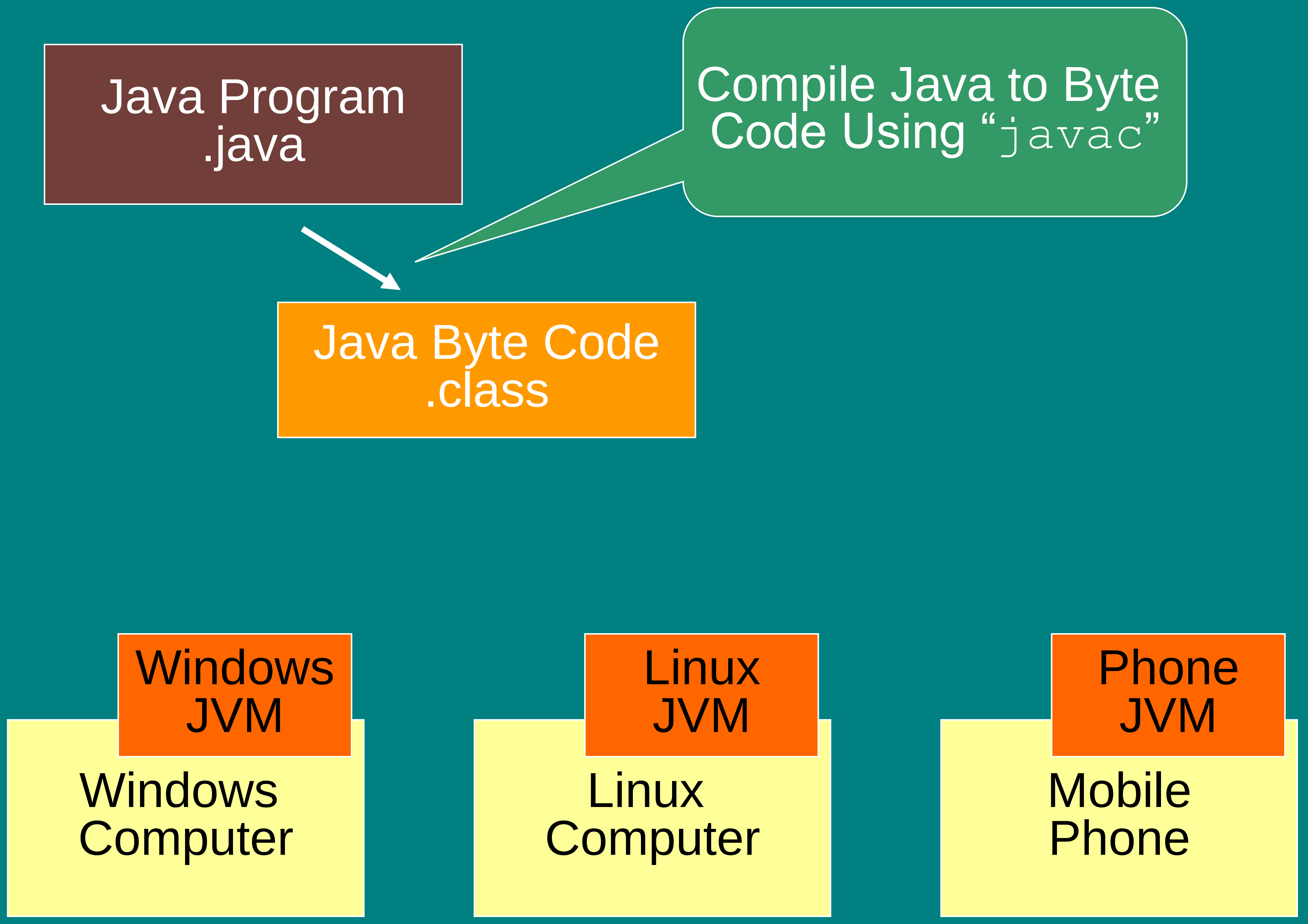
Java Program
.java

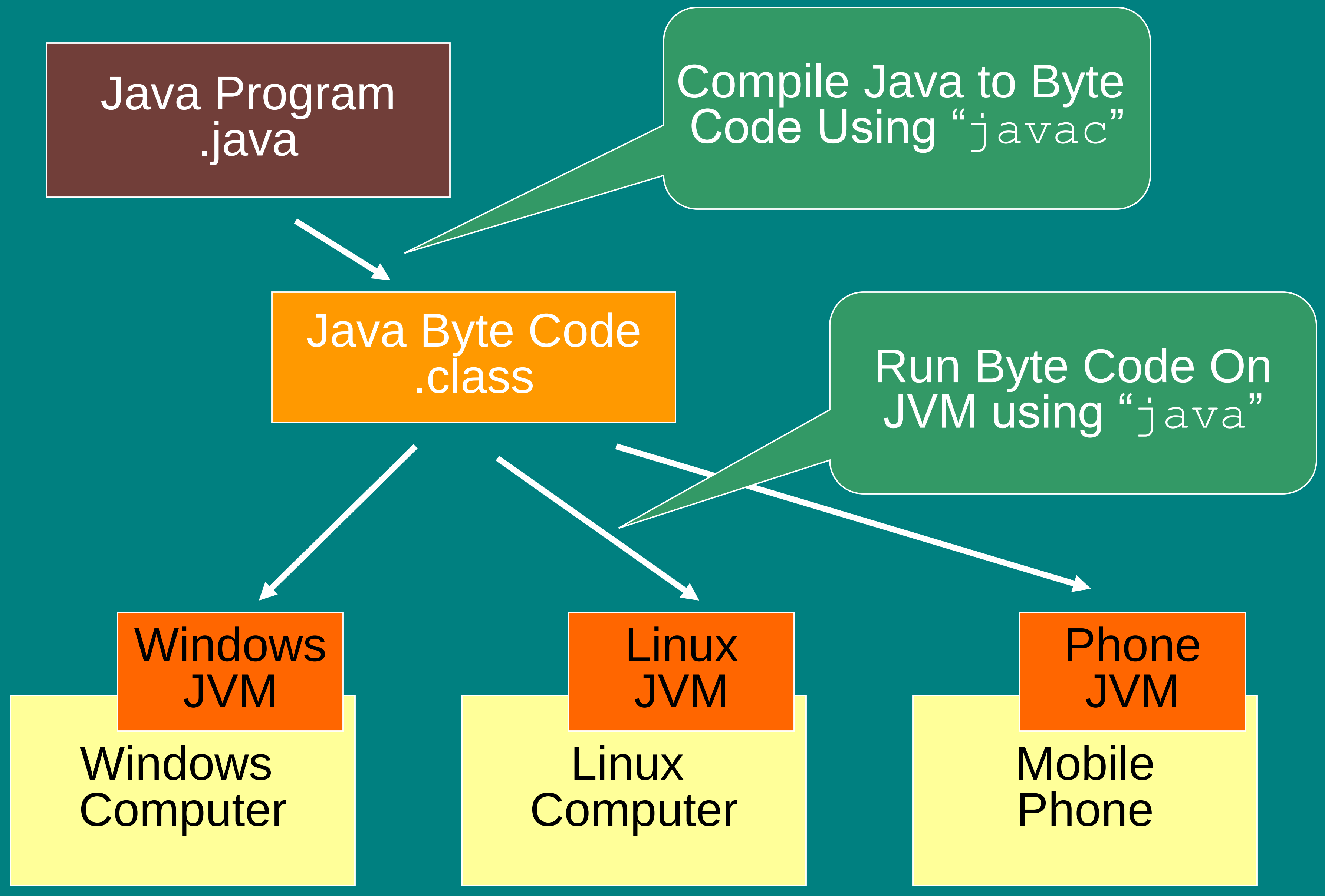
Windows
Computer

Linux
Computer

Mobile
Phone


Java Program
.java
Windows
JVM
Windows
Computer
Linux
JVM
Linux
Computer
Phone
JVM
Mobile
Phone


Java Program
.java
Compile Java to Byte Code Using "javac"
Java Byte Code
.class
Windows JVM
Windows Computer
Linux JVM
Linux Computer
Phone JVM
Mobile Phone



Java Program
.java
Compile Java to Byte Code Using "javac"
Java Byte Code
.class
Run Byte Code On JVM using "java"
Windows JVM
Windows Computer
Linux JVM
Linux Computer
Phone JVM
Mobile Phone

Java Byte Code

- Java compiles to *Java Byte Code*.
  – Type: “javap -c <ClassName>” to see the byte code.

- Every computer must have its own Java Virtual Machine (JVM) which runs the byte code.

- Every different OS must have its own JVM

Live-Demo

"Anything that can go wrong, will go wrong"

A for loop in Java

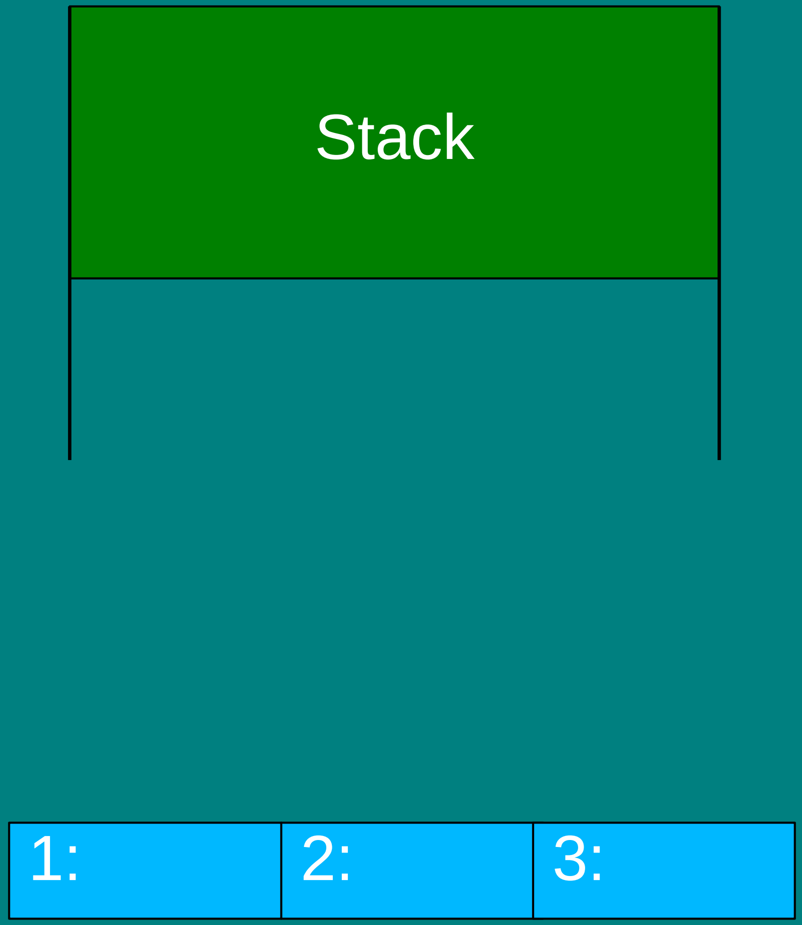
```
0: iconst_1
1: istore_1
2: iconst_1
3: istore_2
4: iload_2
5: iconst_4
6: if_icmpge     26
9: iload_1
10: iload_2
11: iadd
12: istore_1
13: getstatic  #7 // Field java/lang/System.out
16: iload_1
17: invokevirtual #13 // println:(I)V
20: iinc          2, 1
23: goto          4
26: return
```

A Stack Machine

A stack machine has a stack to hold data and a small number of registers.

Data pushed onto the stack or “popped” off.

The registers are fast, but there are only a few of them.



Java Byte Code

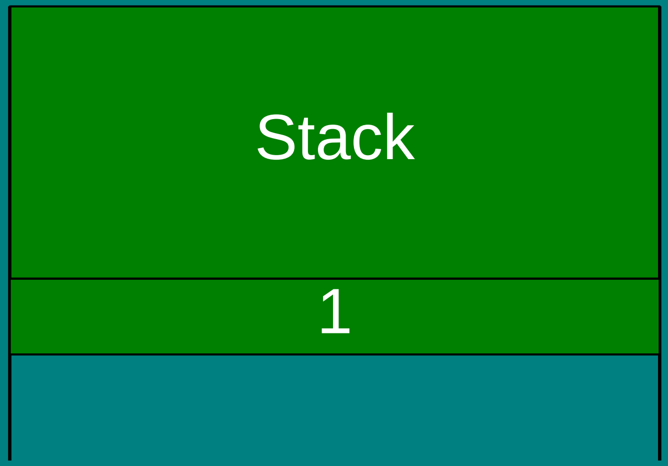
- iconst_0 : push 0 onto the stack
- istore_1: pop the top of the stack as variable 1
- goto: jump to line:
- iload_1: push variable 1 onto the stack
- iadd: add the top two numbers on the stack.
- if_icmpge: if 1st item on stack =< 2nd jump
- Ifeq: if 1st item on stack > 2nd jump to line

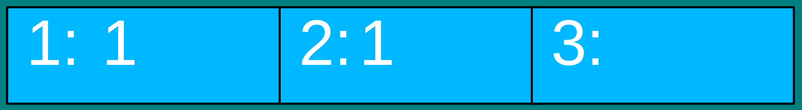
A Stack Machine

Example code starts off by loading 0s into registers 1 and 2.

These are i & j in the code.

0: iconst_1
1: istore_1
2: iconst_1
3: istore_2





A Stack Machine

Next the code checks the for loop guard:

4: iload_2
5: iconst_4
6: if_icmpge 26

The program doesn't jump

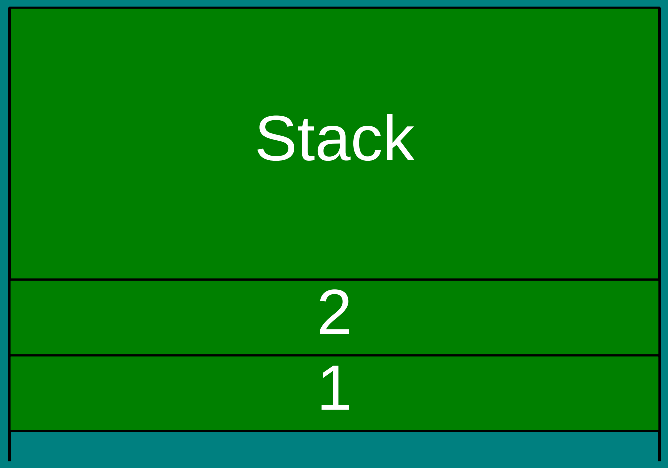
Stack
1
4

1: 1	2:1	3:

A Stack Machine

The for loop body.

9: iload_1
10: iload_2
11: iadd
12: istore_1
13: getstatic …
16: iload_1
17: invokevirtual ...



A Stack Machine

The loop continues:

```
   ...
   4: iload_2
   5: iconst_4
   6: if_icmpge     26
   …
   …
  20: iinc         2, 1
  23: goto          4
  26: return
```

Stack

2

4

1: 2	2: 2	3:

A Stack Machine

The loop continues:

```
 ...
 4: iload_2
 5: iconst_4
 6: if_icmpge     26
 …
 …
20: iinc         2, 1
23: goto          4
26: return
```

1: 4	2: 3	3:

A Stack Machine

The loop continues:

```
    ...
 4: iload_2
 5: iconst_4
 6: if_icmpge     26
    …
    …
20: iinc          2, 1
23: goto          4
26: return
```

Stack
4
4

1: 7	2: 4	3:

Decompilation

- Wouldn’t it be much easier to work with the source code, rather than the byte code?
- JD-GUI is a Java de-compiler, it transforms Java Byte Code into Java Code.
- Not perfect, e.g. confuses 0,1 and true, false.

Live-Demo

"Anything that can go wrong, will go wrong"

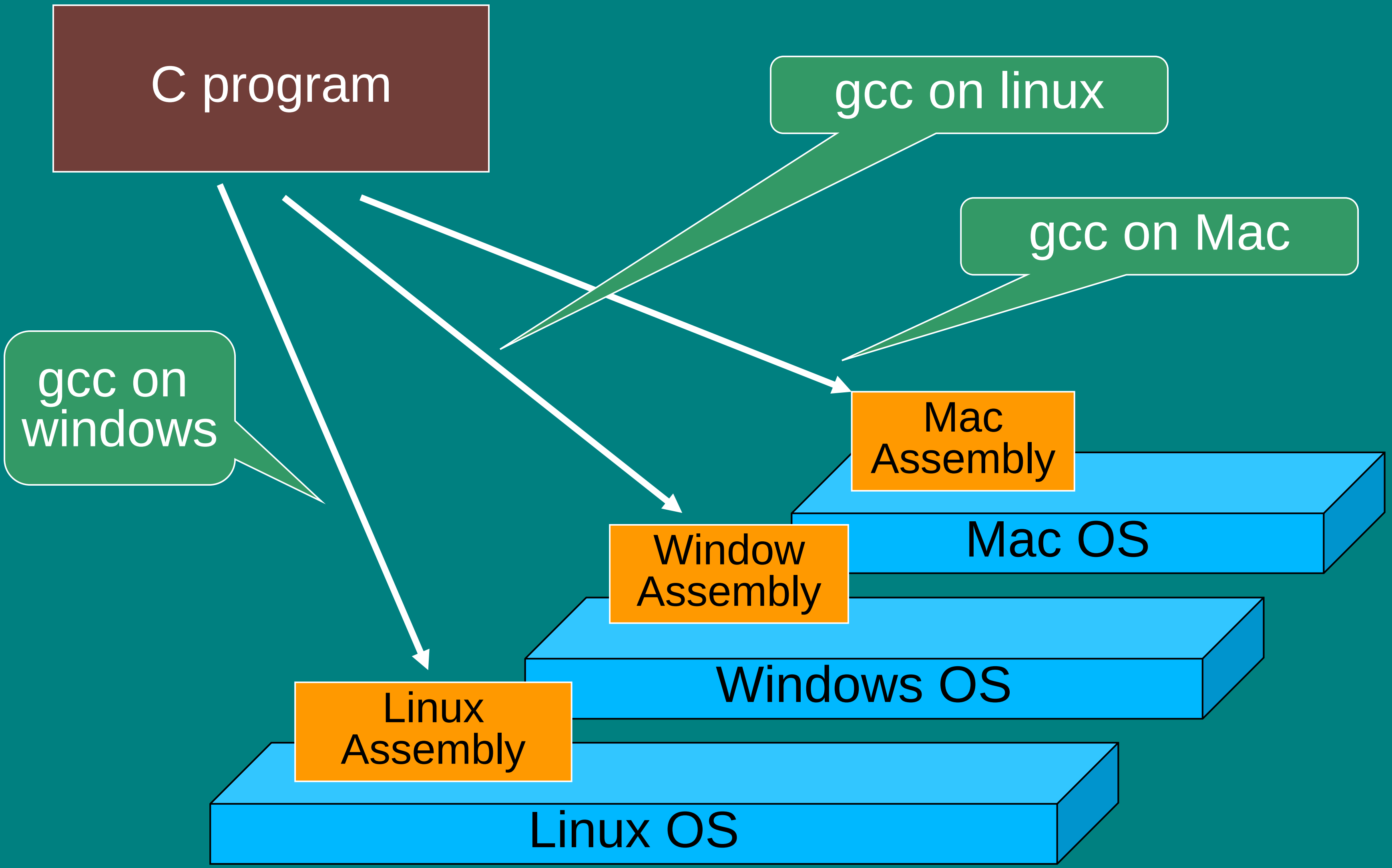
A password checker (2) in Java

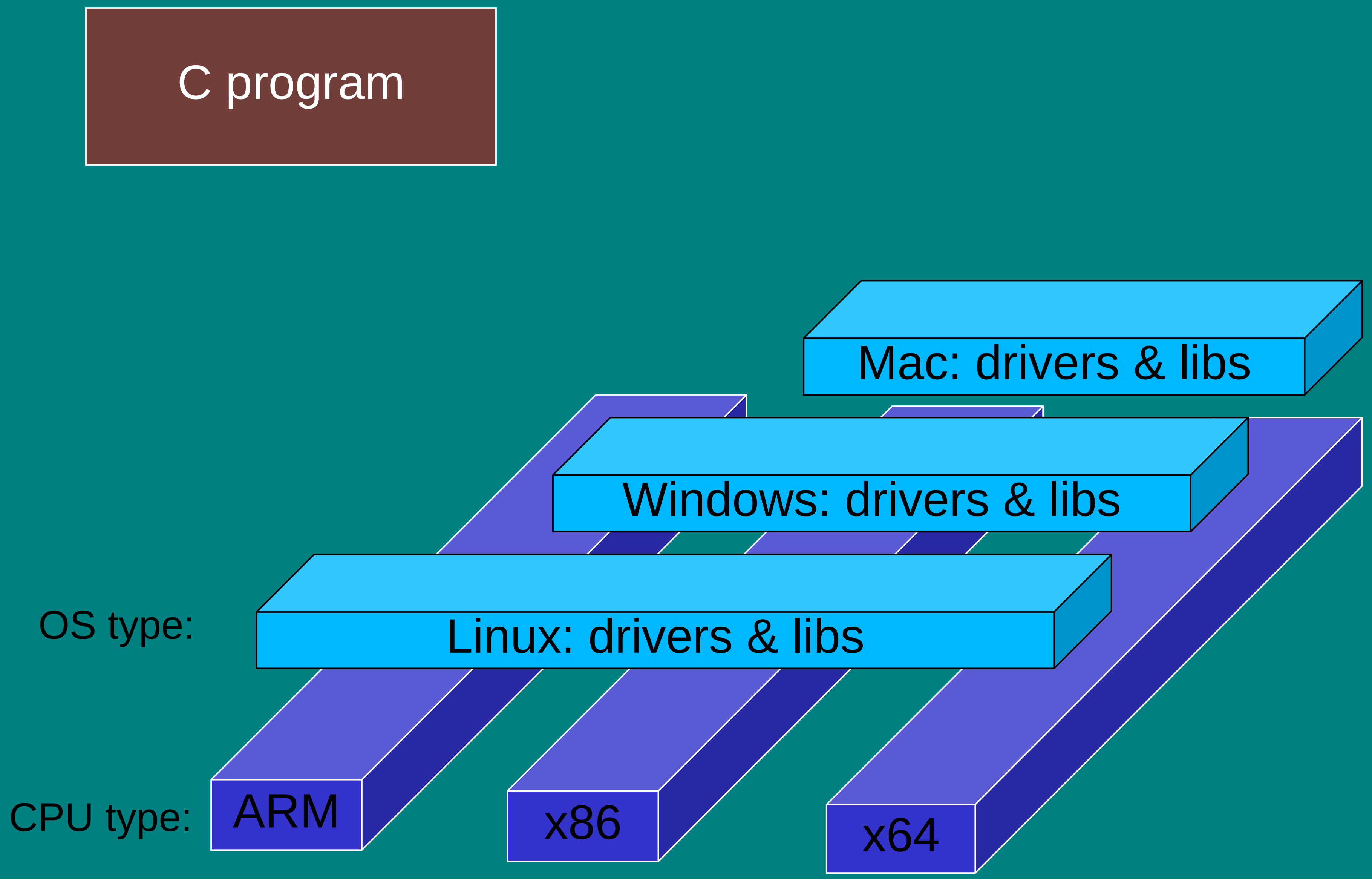
Bypassing the password check.

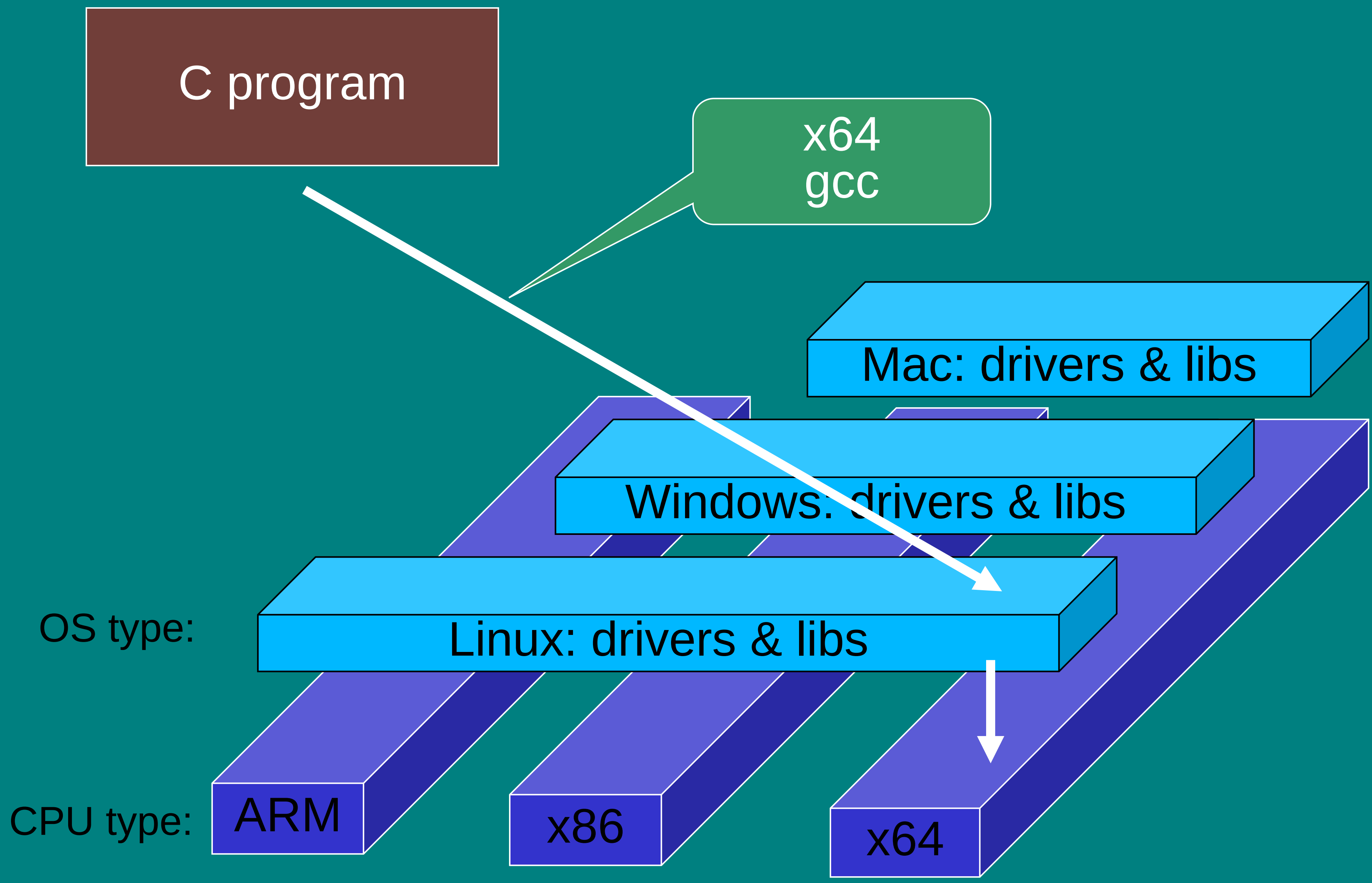
- De-compilation makes it much easier to understand what a program is doing.
- It also makes it easy to alter and recompile the code.
- All code that is used to protect the code can be removed.

Binaries

- Binaries are written in assembly
- Much lower level than Java byte code
- Assembly compiled for one type of machine won’t run on another
- But the same techniques apply


C program
gcc on linux
gcc on Mac
gcc on windows
Mac Assembly
Mac OS
Window Assembly
Windows OS
Linux Assembly
Linux OS


C program
Mac: drivers & libs
Windows: drivers & libs
OS type:
Linux: drivers & libs
CPU type:
ARM
x86
x64


C program
x64
gcc
Mac: drivers & libs
Windows: drivers & libs
Linux: drivers & libs
OS type:
CPU type:
ARM
x86
x64

IDA pro

- IDA pro is an Interactive DisAssembler.
- It helps a human understand binaries.
- This is the standard tool for malware binary analysis, security analysis of firmware and reverse engineering.
- There is are free & demo versions: http://www.hex-rays.com/
- NSA released (open-source) Ghidra – very powerful as well, decide for yourself

Live-Demo

"Anything that can go wrong, will go wrong"

Opening a binary in IDA

Some x86 Commands

PUSH: add to top of stack

POP: read and remove from top of stack

CALL: execute a function

JMP: jump to some code (like writing to EIP)

RET, RETN, RETF: end a function and restart calling code.

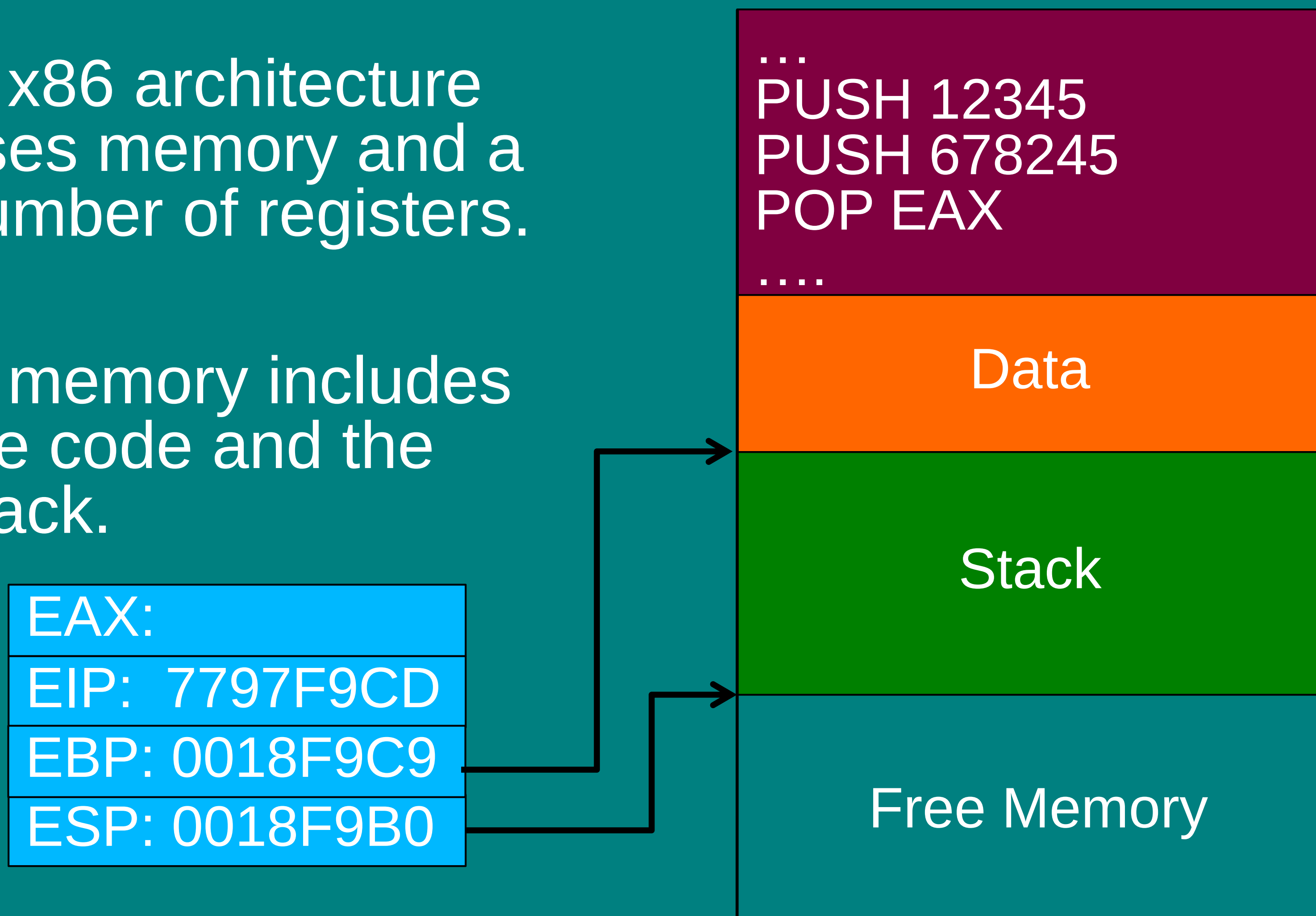
MOV: move value between registers

MOV r1,r2 = PUSH r2
POP r1

x86

The x86 architecture uses memory and a number of registers.

The memory includes the code and the stack.



Common Pattern 1

Data is moved to a register, operation is called, result stored in memory location or register.

```
mov     eax, [esp+1Ch]
add     [esp+18h], eax
```

- Value at [esp+1Ch] is moved to register eax,
- It is added to the value at [esp+18h]
- The result is stored at [esp+18h]

Flags

After an arithmetic operation flags are set.

- ZF: Zero flag
  - Set to 1 if result is 0
- SF: Sign flag
  - Set to 1 if result is negative
- OF: Overflow flag:
  - Set to 1 if operation overflowed.

Compare and Test

Compare and tests will set these flags, with no other affect.

- CMP   a b
  - calculates a-b then sets flags
- TEST a b
  - does a bitwise “and”: a /\ b then sets flags

Jump Commands

- Jump if equal, Jump if zero
  - JE,JZ address
  - Jumps to address if ZF = 1
- Jump if not equal, Jump if not zero
  - JNE,JNZ address
  - Jumps to address if ZF =/= 0
- Jump if less than
  - JL address
  - Jump to address if SF=1 and OF=/=1

Common Pattern 2

Data is compared using “cmp” or “test”, then a jump is made based on the result.

```
cmp     dword ptr [esp+1Ch], 3
jle     short loc_80483DF
```

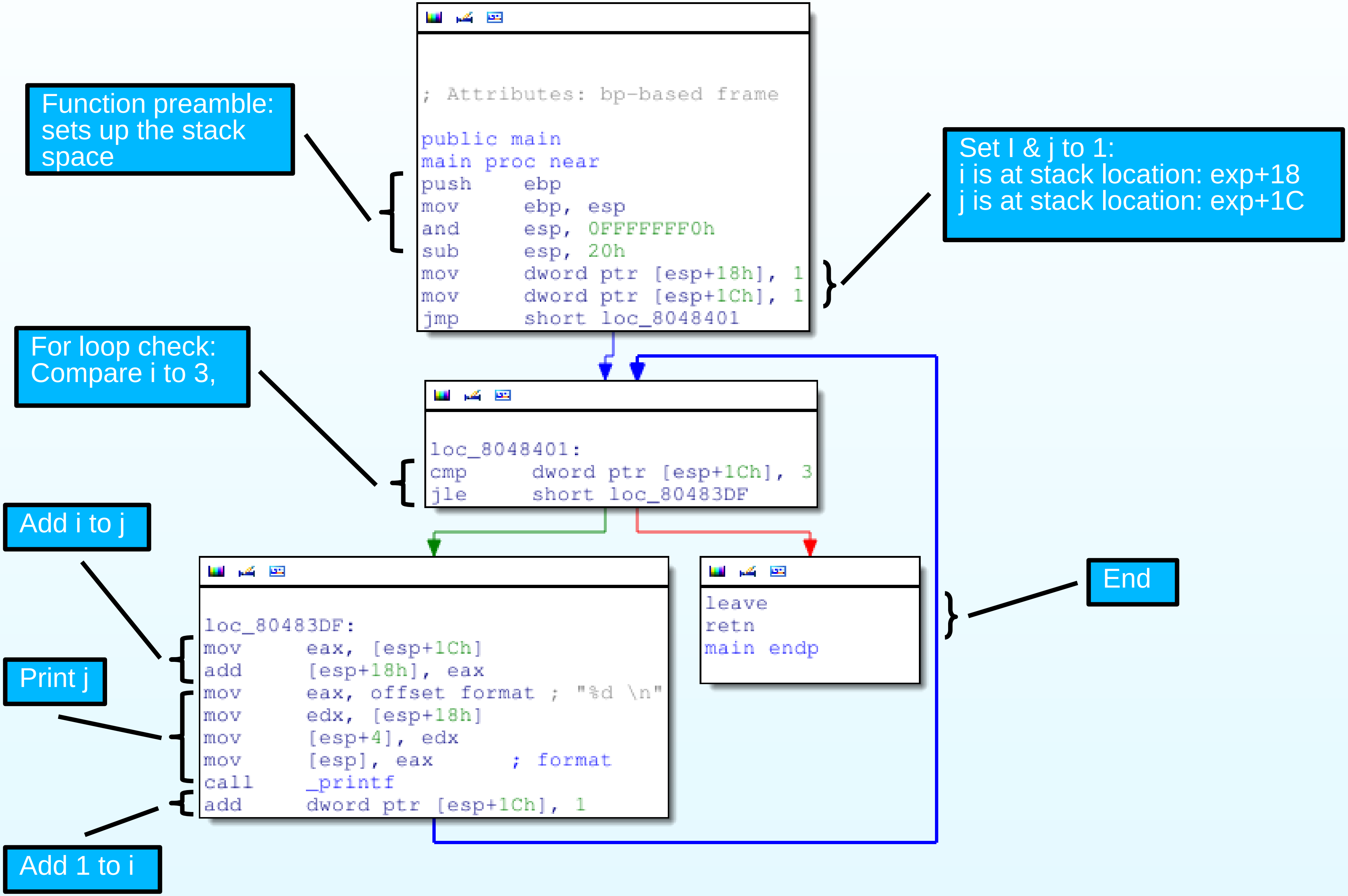
- Value [esp+1Ch] – 3 is calculated (not stored)
- If it is less than or equal to zero, the program jumps to location “loc_80483DF”
- Otherwise it continues to the next command.

Common Pattern 3

- Data is loaded onto the stack
- Function is called that uses these values,
- The result will be pointed to by eax

```
mov     [esp+4], eax     ; s2
mov     dword ptr [esp], offset s1 ; "exit"
call    _strncmp
```

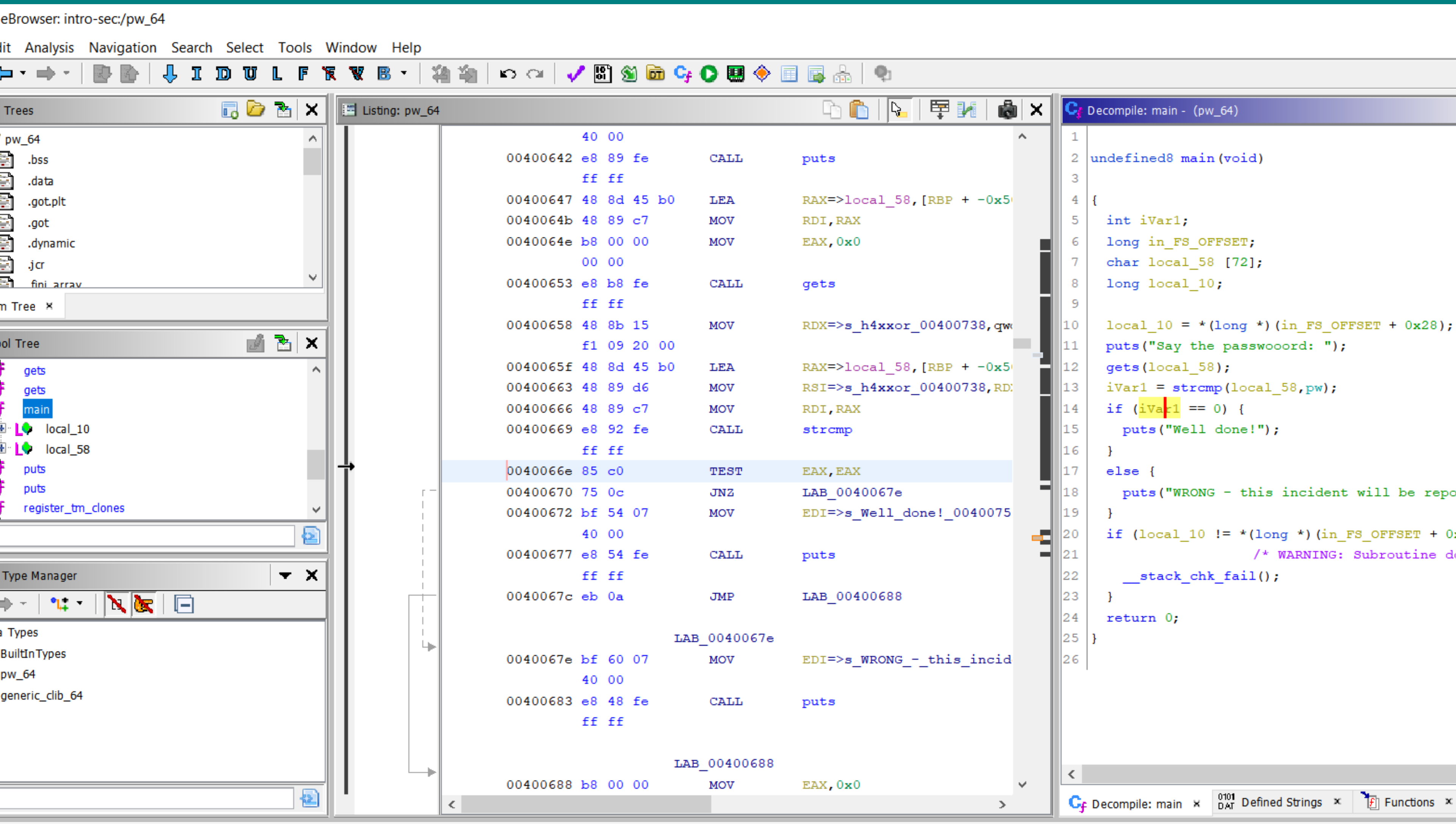
- Value in eax is moved to [esp+4]
- “exit” is put on top of the stack
- String compare is called on these.
- The result will be returned in the eax register


Function preamble: sets up the stack space
Set I & j to 1:
i is at stack location: exp+18
j is at stack location: exp+1C
For loop check: Compare i to 3,
Add i to j
Print j
Add 1 to i
End
; Attributes: bp-based frame
public main
main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+18h], 1
mov dword ptr [esp+1Ch], 1
jmp short loc_8048401
loc_8048401:
cmp dword ptr [esp+1Ch], 3
jle short loc_80483DF
loc_80483DF:
mov eax, [esp+1Ch]
add [esp+18h], eax
mov eax, offset format ; "%d \n"
mov edx, [esp+18h]
mov [esp+4], edx
mov [esp], eax ; format
call _printf
add dword ptr [esp+1Ch], 1
leave
retn
main endp

Live-Demo

"Anything that can go wrong, will go wrong"

Analysing password checker in C with IDA


eBrowser: intro-sec:/pw_64
lit Analysis Navigation Search Select Tools Window Help
Trees
pw_64
.bss
.data
.got.plt
.got
.dynamic
.jcr
m Tree
ol Tree
gets
gets
main
local_10
local_58
puts
puts
register_tm_clones
Type Manager
Types
BuiltInTypes
pw_64
generic_clib_64
Listing: pw_64
40 00
00400642 e8 89 fe CALL puts
ff ff
00400647 48 8d 45 b0 LEA RAX=>local_58,[RBP + -0x5
0040064b 48 89 c7 MOV RDI,RAX
0040064e b8 00 00 MOV EAX,0x0
00 00
00400653 e8 b8 fe CALL gets
ff ff
00400658 48 8b 15 MOV RDX=>s_h4xxor_00400738,qw
f1 09 20 00
0040065f 48 8d 45 b0 LEA RAX=>local_58,[RBP + -0x5
00400663 48 89 d6 MOV RSI=>s_h4xxor_00400738,RD
00400666 48 89 c7 MOV RDI,RAX
00400669 e8 92 fe CALL strcmp
ff ff
0040066e 85 c0 TEST EAX,EAX
00400670 75 0c JNZ LAB_0040067e
00400672 bf 54 07 MOV EDI=>s_Well_done!_0040075
40 00
00400677 e8 54 fe CALL puts
ff ff
0040067c eb 0a JMP LAB_00400688
LAB_0040067e
0040067e bf 60 07 MOV EDI=>s_WRONG_-_this_incid
40 00
00400683 e8 48 fe CALL puts
ff ff
LAB_00400688
00400688 b8 00 00 MOV EAX,0x0
Decompile: main - (pw_64)
undefined8 main(void)
{
int iVar1;
long in_FS_OFFSET;
char local_58 [72];
long local_10;
local_10 = *(long *)(in_FS_OFFSET + 0x28);
puts("Say the passwooord: ");
gets(local_58);
iVar1 = strcmp(local_58,pw);
if (iVar1 == 0) {
puts("Well done!");
}
else {
puts("WRONG - this incident will be repo
}
if (local_10 != *(long *)(in_FS_OFFSET + 0
/* WARNING: Subroutine d
__stack_chk_fail();
}
return 0;
}
Decompile: main
Defined Strings
Functions

Live-Demo

"Anything that can go wrong, will go wrong"

Analysing password checker in C with Ghidra

A few words of warning

- Above was for 32 bit, these day a lot of programs are 64 bit
- No fundamental differences, but note:
  - Registers: 32-bit `eax` vs 64-bit `rax` etc.
  - Function calls stack vs registers
- Intel vs AT&T assembly syntax:

`mov eax, 5` vs. `mov $5, %eax`

Live-Demo

"Anything that can go wrong, will go wrong"

Patching a password checker in C

Live-Demo

"Anything that can go wrong, will go wrong"

Patching a game

Common Techniques

- Look for strings
- Identify key tests and check the values in the register using a debugger
- Swap JEQ and JNEQ etc.
- Jump over the instructions that perform checks (replace with NOP)

Defenses

- Dynamically construct the code
  - Attacker can run code
- Encrypt the binary
  - Your program must include the key in plain text, so the attacker can find it
- Obfuscate the code, e.g. mix data and code, so it's not clear which is which
  - Can slow down attacks by months or years! (e.g. Skype)

Defense

- Require online activation:
  Activation can be completely disabled, users don't like this.

- Require online content, e.g. WoW, BlueRay

- Hardware-based protection, i.e. store and *run* part of the code in tamper-resistant hardware.

Examples

- You can find IDA, jd-gui and some example files in the dan directory
- Username: dan, password: dan!dan
- Not assessed, but highly recommended.
- Feel free to ask questions about this in lab sessions.

Summary

- Machine code can be inspected and edited.
- Many tools exist to inspect, debug and decompile code.
- Most software protection can be removed.
- But slowing this down by months or years can save a business.