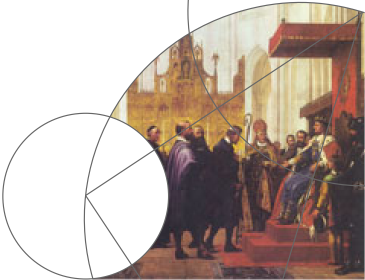
PWNIES INTRO WORKSHOP

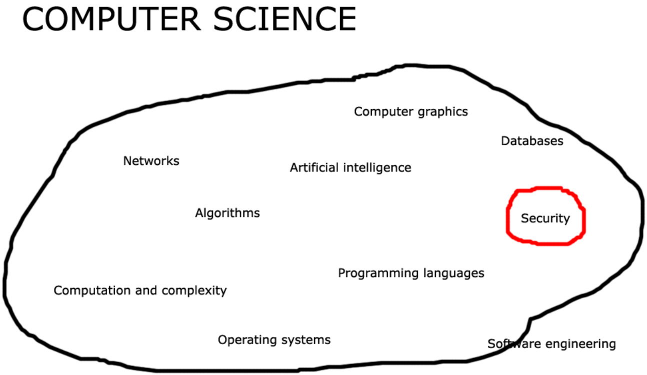
Day 1: Shellcode

Department of Computer Science, University of Copenhagen

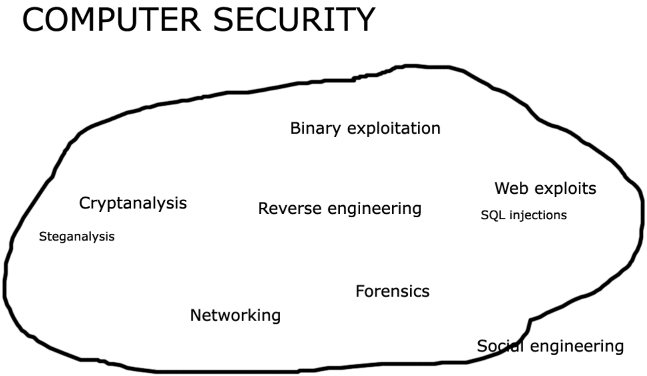


November 11, 2013

Computer and information security

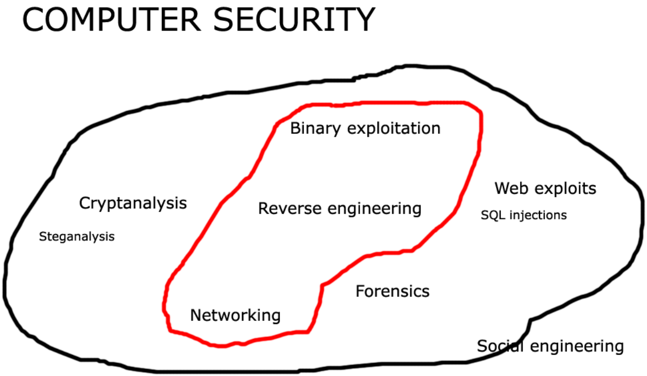


ZOOM; ENHANCE



Many of these are common categories in CTFs.

ZOOM; ENHANCE



Our focus in this workshop.

This week's program

- Shellcode and x86 assembly
- Reverse engineering and debugging
- Buffer overflow exploits
- Network-based exploits
- Advanced topics (format string, return-oriented programming)

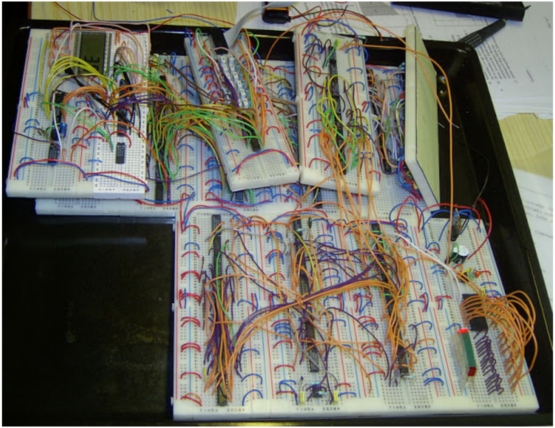
Today's program

0. Hardware and lowlevel programming
1. CPU, x86 assembler instructions, and shellcode
2. Memory and the stack
3. Improving your shellcode
4. Extra exercises

Today's program

0. **Hardware and lowlevel programming**
1. CPU, x86 assembler instructions, and shellcode
2. Memory and the stack
3. Improving your shellcode
4. Extra exercises

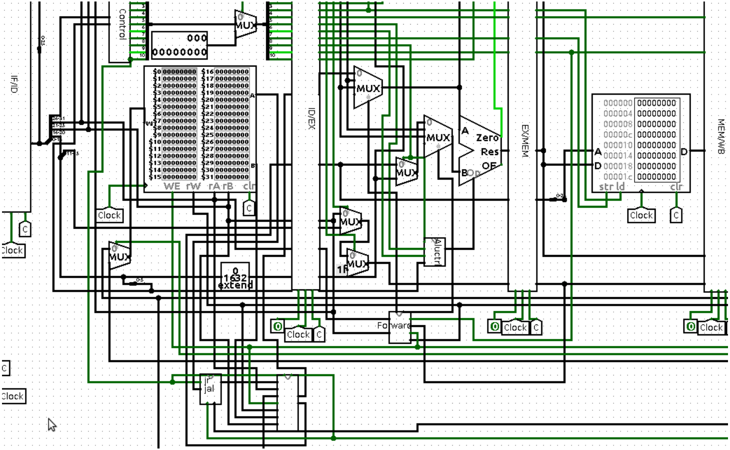
0. Hardware and lowlevel programming



At the base level, computers are just electronic components

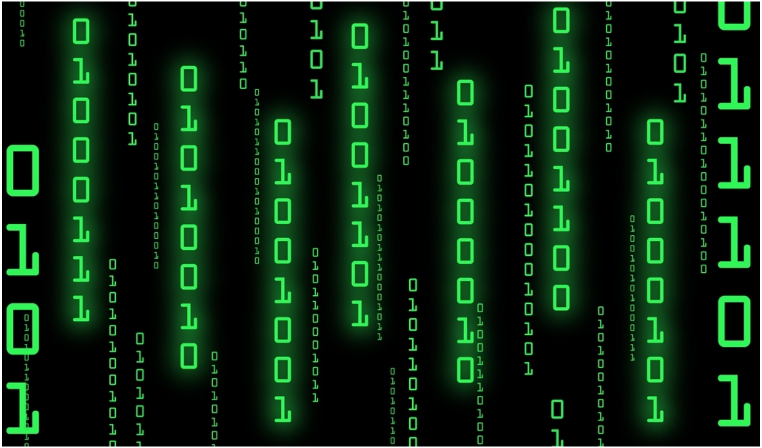
...

0. Hardware and lowlevel programming



. . . connected by logical circuits . . .

0. Hardware and lowlevel programming



...and mapped to 0's and 1's.

0. Hardware and lowlevel programming

- Everything the computer does is *really just numbers*.

0. Hardware and lowlevel programming

- Everything the computer does is *really just numbers*.
- Example: In the MIPS architecture, the sequence 00000000101011111000000000100000 is an instruction for the computer to add two numbers.

0. Hardware and lowlevel programming

- Everything the computer does is *really just numbers*.
- Example: In the MIPS architecture, the sequence 00000000101011111000000000100000 is an instruction for the computer to add two numbers.
- What if we change *one* bit, 0**1**000000101011111000000000100000?

0. Hardware and lowlevel programming

- Everything the computer does is *really just numbers*.
- Example: In the MIPS architecture, the sequence 00000000101011111000000000100000 is an instruction for the computer to add two numbers.
- What if we change *one* bit, 0**1**000000101011111000000000100000?
- The meaning changes entirely - now it's a `jump` instruction.

0. Hardware and lowlevel programming

- Everything the computer does is *really just numbers*.
- Example: In the MIPS architecture, the sequence 00000000101011111000000000100000 is an instruction for the computer to add two numbers.
- What if we change *one* bit, 0**1**000000101011111000000000100000?
- The meaning changes entirely - now it's a `jump` instruction.
- Going deeper: Binary sequences also represent numbers (the jump one is 1085243424), ascii values (@¯€), and more.

0. Hardware and lowlevel programming

- A CPU works by executing a flow of instructions.

0. Hardware and lowlevel programming

- A CPU works by executing a flow of instructions.
- Sometimes, a CPU can be 'tricked' into executing your instructions instead of the original program.

0. Hardware and lowlevel programming

- A CPU works by executing a flow of instructions.
- Sometimes, a CPU can be 'tricked' into executing your instructions instead of the original program.
- Today, you'll learn how to write these instructions in the Intel x86 architecture.

Today's program

0. Hardware and lowlevel programming
1. **CPU, x86 assembler instructions, and shellcode**
2. Memory and the stack
3. Improving your shellcode
4. Extra exercises

1.0 CPU, x86 assembler instructions, and shellcode

- A CPU can only execute simple instructions like `add` or `mov`.

1.0 CPU, x86 assembler instructions, and shellcode

- A CPU can only execute simple instructions like `add` or `mov`.
- Usually one line of c code will translate into several lines of assembler.

1.1 registers

- A register is a 32 bit memory inside the CPU.

1.1 registers

- A register is a 32 bit memory inside the CPU.
- EAX, EBX, ECX, EDX (EDI, ESI) are general purpose registers.

1.1 registers

- A register is a 32 bit memory inside the CPU.
- EAX, EBX, ECX, EDX (EDI, ESI) are general purpose registers.
- ESP, EBP are used for memory mannagement.

1.1 registers

- A register is a 32 bit memory inside the CPU.
- EAX, EBX, ECX, EDX (EDI, ESI) are general purpose registers.
- ESP, EBP are used for memory mannagement.
- EIP points to the next instruction.

1.1 registers

- A register is a 32 bit memory inside the CPU.
- EAX, EBX, ECX, EDX (EDI, ESI) are general purpose registers.
- ESP, EBP are used for memory mannagement.
- EIP points to the next instruction.
- EFLAGS will not be discussed.

1.1 registers

32	24	16	8
E_X			
		_X	
		_H	_L

1.2 Example: mov instruction:

mov a, b

a = b

- mov register, value
  moves value (like 5) into register (like eax)
- mov register a, register b
  moves the content of reg. b into reg. a
- mov register a, [pointer]
  moves the content on the address "pointer" into reg. a

1.3 syscall:

Stops the instruction execution, and lets the operating system execute a function.
http://syscalls.kernelgrok.com/

- eax containes the syscall number.
- ebx, ecx and edx contains the arguments.
- 0x80 makes the call.

1.4 compiling, linking, and executing a program:

compile:	nasm -f elf32 <name>.asm
link:	ld -melf_i386 -o <name> <name>.o
run	./<name>

Replace <name> with the filename of your program.

1.5 x86 assembler template

```
[bits 32]
section .text
global _start



_start:
mov eax, str



section .data
str: db 'lolhej', 0x0
```

1. CPU, x86 assembler instructions, and shellcode

```
 ----------------------------------
| Exercise: Write a program        |
|           that executes /bin/sh! |
 ----------------------------------
              \
              (\^)
               o O\______-
               \_/      \\
                 \ ____ /\\
                 //\ ||\\ \\
                 ||\\|| \\
```

Today's program

0. Hardware and lowlevel programming
1. CPU, x86 assembler instructions, and shellcode
2. **Memory and the stack**
3. Improving your shellcode
4. Extra exercises

2. Memory and the stack

Memory on x86 is very complex. Long story short:

- Memory ranges from `0x00000000` to `0xffffffff`.

2. Memory and the stack

Memory on x86 is very complex. Long story short:

- Memory ranges from `0x00000000` to `0xffffffff`.
- Every address contains 1 byte, but are normally used in chunks of 4 bytes (a dword).

2. Memory and the stack

Memory on x86 is very complex. Long story short:

- Memory ranges from `0x00000000` to `0xffffffff`.
- Every address contains 1 byte, but are normally used in chunks of 4 bytes (a dword).
- x86 is little endian (the 'least significant byte' is saved first). The number `0xDECAFBAD` consists of `0xAD`, `0xFB`, `0xCA`, `0xDE`, *in that order*!

2. Memory and the stack

Memory on x86 is very complex. Long story short:

- Memory ranges from `0x00000000` to `0xffffffff`.
- Every address contains 1 byte, but are normally used in chunks of 4 bytes (a dword).
- x86 is little endian (the 'least significant byte' is saved first). The number `0xDECAFBAD` consists of `0xAD`, `0xFB`, `0xCA`, `0xDE`, *in that order*!
- The addresses are local for your program.

2. Memory and the stack

Example:

0x000c	...
0x0008	...
0x0004	0xDECAFBAD
0x0000	...

2. Memory and the stack

Example:

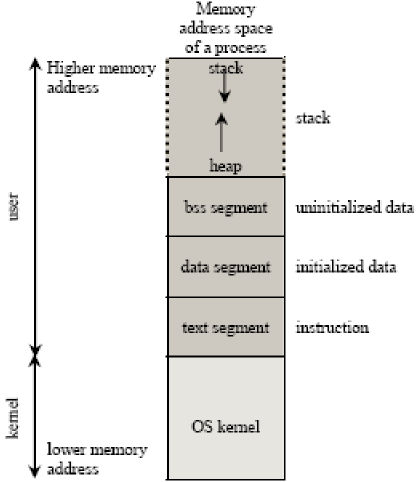
0x000c	...
0x0008	...
0x0004	0xDECAFBAD
0x0000	...

Zooming in at address 0x0004:

0x0007	0xDE
0x0006	0xCA
0x0005	0xFB
0x0004	0xAD

2. Memory and the stack

x86 memory is divided into segments:



2. Memory and the stack

- We have made some assembler code which spawns a shell.

2. Memory and the stack

- We have made some assembler code which spawns a shell.
- Problem: We are using labels (why is this a problem?).

2. Memory and the stack

- We have made some assembler code which spawns a shell.
- Problem: We are using labels (why is this a problem?).
- Solution: We have plenty of space in dynamic memory! It's time to look at the stack.

2. Memory and the stack

- The stack is a 'first in, last out' structure that grows *downwards*.

2. Memory and the stack

- The stack is a 'first in, last out' structure that grows *downwards*.
- ESP, the stack pointer, points to the top of the stack.

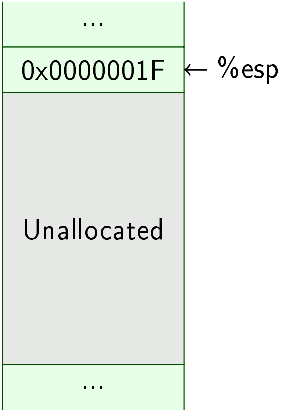
2. Memory and the stack

- The stack is a 'first in, last out' structure that grows *downwards*.
- ESP, the stack pointer, points to the top of the stack.
- The `push` instruction adds an element (4 bytes) to the top of the stack and decrements ESP.

2. Memory and the stack

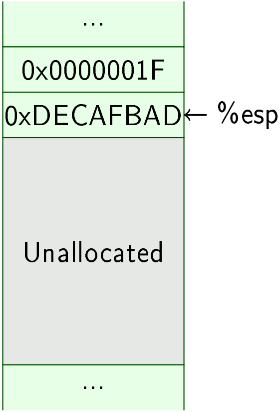
- The stack is a 'first in, last out' structure that grows *downwards*.
- ESP, the stack pointer, points to the top of the stack.
- The push instruction adds an element (4 bytes) to the top of the stack and decrements ESP.
- The pop instruction increments ESP by 4 (effectively removing the 4 bytes at the top).

2. Memory and the stack



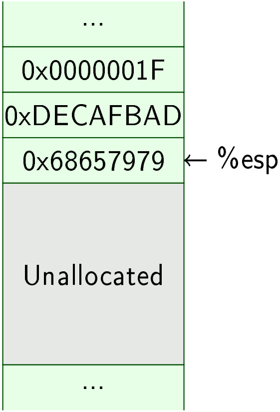

```
->  push 0xDECAFBAD
    push 'heyy'
    push byte 0x42
    pop eax
```

2. Memory and the stack



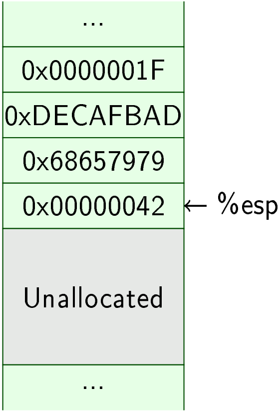

```
    push 0xDECAFBAD
->  push 'heyy'
    push byte 0x42
    pop eax
```

2. Memory and the stack



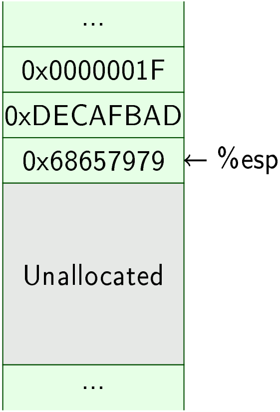

```
    push 0xDECAFBAD
    push 'heyy' ('h' = 0x68. Check man ascii.)
->  push byte 0x42
    pop eax
```

2. Memory and the stack




```
    push 0xDECAFBAD
    push 'heyy'
    push byte 0x42
->  pop eax
```

2. Memory and the stack



`push 0xDECAFBAD`

`push 'heyy'`

`push byte 0x42`

`pop eax` (`EAX` now contains `0x42`)

2. Memory and the stack

```
 -----------------------------------------
| Exercise: Write a program that executes |
|           /bin/sh without using labels! |
| Hint:     Strings are null-terminated.  |
 -----------------------------------------
              \
              (\^)
               o O\_____-
               \_/      \\
                 \ ____ /\\
                 //\ ||\\ \\
                 ||\\|| \\
```

Today's program

0. Hardware and lowlevel programming
1. CPU, x86 assembler instructions, and shellcode
2. Memory and the stack
3. **Improving your shellcode**
4. Extra exercises

3. Improving your shellcode

- xor eax, eax
  - set eax to 0.

3. Improving your shellcode

- xor eax, eax
  - set eax to 0.
- mul ecx
  - edx:eax = eax*ecx

3. Improving your shellcode

- xor eax, eax
  - set eax to 0.
- mul ecx
  - edx:eax = eax*ecx
- mov al, 5
  - move 5 to the least significant byte in eax.

3. Improving your shellcode

```
 ------------------------------------------
| Exercises:                                |
| a. Write shellcode that executes /bin/sh |
|    without null bytes!                    |
| b. Optimize your shellcode to using as    |
|    few bytes as possible!                 |
 ------------------------------------------
             \
             (\^)
              o O\_____-
              \_/      \\
                \ ____ /\\
                //\ ||\\ \\
                ||\\|| \\
```

Today's program

0. Hardware and lowlevel programming
1. CPU, x86 assembler instructions, and shellcode
2. Memory and the stack
3. Improving your shellcode
4. **Extra exercises**

4. Extra exercises!

```
 --------------------------------------------
| You've found a vulnerability in a service  |
| run by <insert evil>, allowing you to      |
| execute shellcode.                         |
| You're interested in orienting yourself    |
| to find potentially interesting files...   |
|                                            |
| Write shellcode that executes /bin/ls with |
| '-l' as argument!                          |
 --------------------------------------------
             \
             (\^)
              o O\_____-
              \_/      \\
                \ ____ /\\
                //\ ||\\ \\
                ||\\|| \\
```

4. Extra exercises!

```
 ---------------------------------------------
| Huh. It seems they have a file called       |
| gold.txt lying around. Let's check it out.  |
|                                             |
| Write shellcode that writes the contents    |
| of a text file to standard output!          |
| Hint: sys_open, sys_read, sys_write.        |
|       For testing, create your own gold.txt |
|       file.                                 |
 ---------------------------------------------
             \
             (\^)
              o O\_____-
              \_/      \\
                \ ____ /\\
                //\ ||\\ \\
                ||\\|| \\
```

4. Extra! Compare instruction

cmp a, b
a ? b
compares a and b, the result is stored in a special register, and can be used by instructions like jeq, jne, jge, jle

- cmp register, value
  compares the value of register and value
- cmp register a, register b
  compares the value register a and b
- add register a, [register b + x]
  compares the value of register a and the value on the address of register b + x

4. Extra! Conditional jump instruction

j_ _ label
Used after compare, if the condition is true then the instruction pointer is moved to the address given by label.

- ja: jump if above
- jb: jump if below
- jeq: jump if equal
- jne: jump if not equal
- jae: jump if above or equal
- jbe: jump if below or equal
- jmp: always jump (does not have to be after a compare)

Compare and conditional jump instructions can be used to form loops.

4. Extra exercises!

```
 ----------------------------------------------
| a. Write a program that calculates the       |
|    factorial of some number n.               |
|    (For now, hardcode the number n in your   |
|    program. We'll get to function calls      |
|    another day.)                             |
| b. Write a program that calculates the nth   |
|    fibonacci number for some number n.       |
 ----------------------------------------------
             \
             (\^)
              o O\______-
              \_/       \\
                \ ____ /\\
                //\ ||\\ \\
                ||\\|| \\
```