# INF226 – Software Security

Håkon Robbestad Gylterud

2019–08–21

# Requirements, assumptions and mechanisms

# Examples of requirements

- Availability of the service.
- Capacity
- Integrity of data
- Authenticity of data
- Recoverability

# Examples of mechanisms

- Choice of programming language
- Rate limiting
- Sanity checks on user inputs
- Access control lists
- Optimisation of algorithms
- Encryption

# Examples of assumptions

Assumptions pointing towards problems:

- User input cannot be trusted to have property `X`
- IP addresses can be spoofed
- Computer resources are finite
- Programmers write bugs

# Examples of assumptions

Assumptions pointing towards solutions:

- If the program checks the input, we know it has property `X`
- An attacker cannot guess a random 128 bit number.
- The semantics of the program.
- The type checker is correct.
- Internet routing is quite robust.

# Example

| Requirement | Mechanism | Assumptions |
|---|---|---|
| Website should have a high uptime. | | Server has limited capacity to process requests.<br>An attacker could send a lot of request. |

Problem: How to prevent legit users experiencing downtime because of an attacker?

# Example

| Requirement | Mechanism | Assumptions |
|---|---|---|
| Website should have a high uptime. | Optimise request handling | Server has limited capacity to process requests. An attacker could send a lot of request, **but not that many.** |

# Example

| Requirement | Mechanism | Assumptions |
|---|---|---|
| Website should have a high uptime. | IP-based rate-limiting | Server has limited capacity to process requests.<br>An attacker could send a lot of request.<br>**An attacker will have a limited number IP addresses.** |

# Example

| Requirement | Mechanism | Assumptions |
|---|---|---|
| Website should have a high uptime. | Require proof of work, for each request. | Server has limited capacity to process requests.<br>An attacker could send a lot of request.<br>**The work will acceptable for normal users, but not for attackers.** |

# Relationship to security

Making requirements = spelling out our intention

Making assumptions = spelling out our knowledge of the environment

# Relationship to security

Making requirements = spelling out our intention

Making assumptions = spelling out our knowledge of the environment

**Definition**

Software security is the ability of software to function according to intentions in an adversarial environment.

# Vulnerabilities and exploits

# Vulnerabilities and exploits

**Definition**

A **vulnerability** in a software is a circumstance in which the program *fails to be secure* (aka. behave according to intentions).

# Vulnerabilities and exploits

**Definition**

An **exploit** of a vulnerability is a procedure which upon execution leads to the circumstance described by the vulnerability, thus demonstrating the insecurity of the program.

# Vulnerabilities and exploits

Examples:

- Broken access control
- Buffer overflow vulnerabilities
- Injection vulnerabilities
- ...

# Remote code execution

The most serious vulnerabilities lead to the attacker being able to run any code on the victim machine.

# The Morris worm

In 1988, a worm exploiting a buffer overflow in `fingerd` spread across the internet.

- A bug in the code made the virus' hosts grind to a halt.
- Internet was partitioned for several days during the clean up.

Even today, twenty years later, buffer-overflow exploits remain some of the more common vulnerabilities.

# Buffer overflow

# Demo/discussion

```
#include <stdio.h>

int main(int argc, char* argv) {
    char buffer[8];
    int a = 3;
    fgets(buffer, 256 , stdin);
    printf("You entered: %s \n", buffer);
    printf("and a = %i \n", a);
}
```

# The basic problem

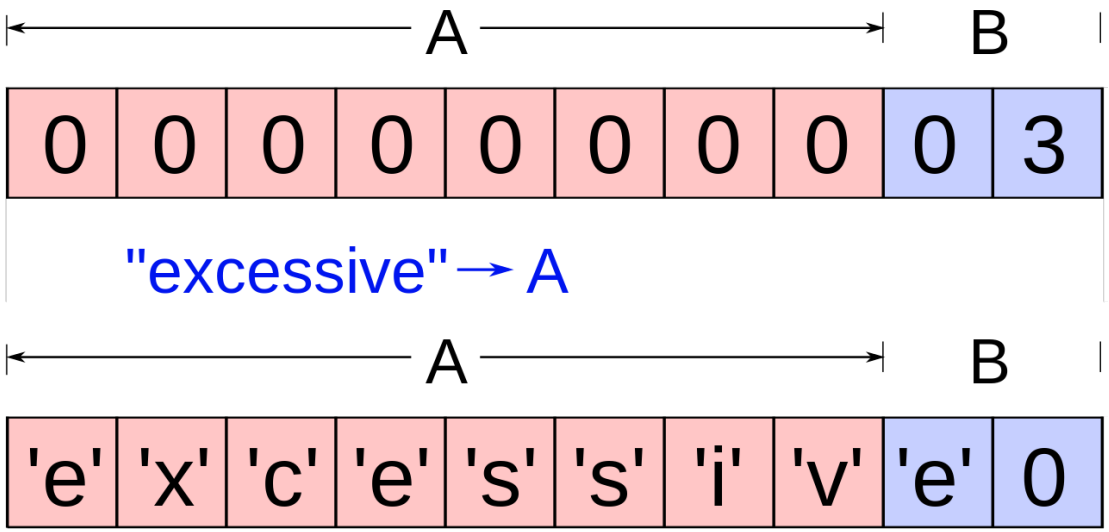



Figure 1: Buffer overflows!

# In code...

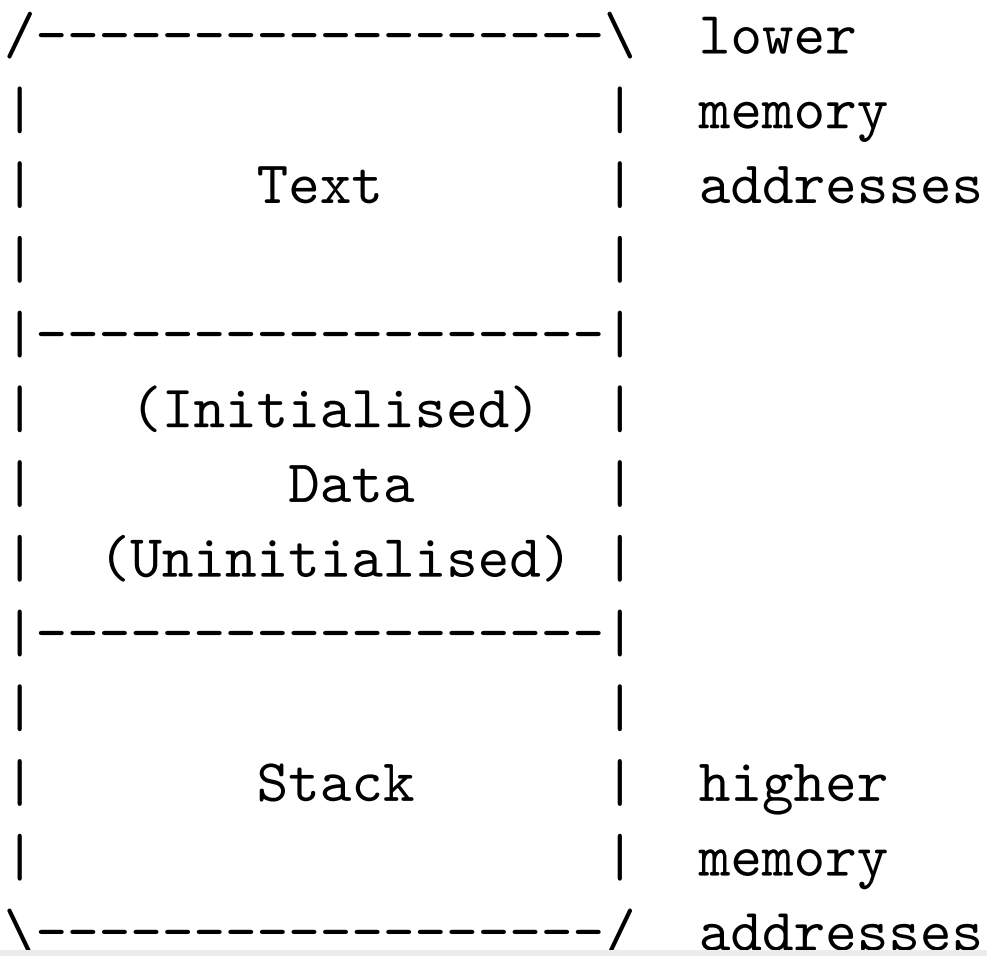
```
int main() {
    char a[] = "short";
    char b[] = "very long";
    // Copy b into a
    for (int i = 0 ; i < strlen(b) ; ++i)
      a[i] = b[i];
    printf("%s",a);
}
```

# Buffer overread

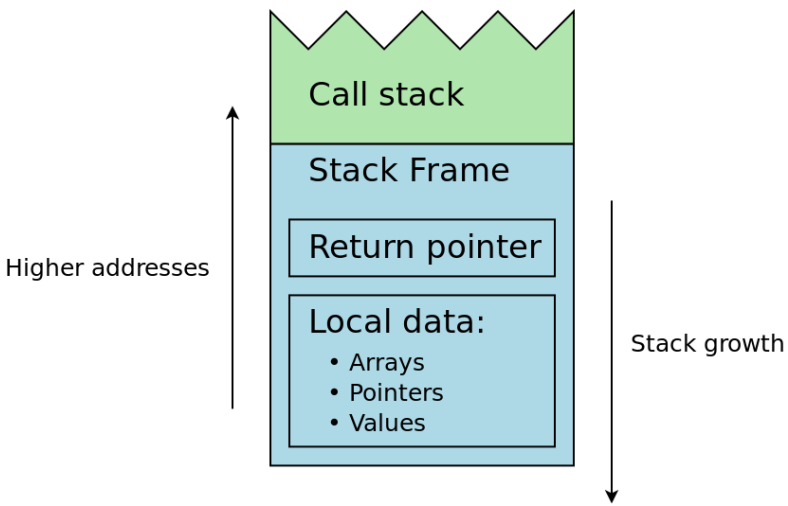
The simplest mistake one can make in an unsafe language is reading outside the bounds of a buffer (array).

Example: The "Heartbleed" bug in libssl was caused by not bounds checking the TLS heart-beat signal before responding.

# Memory layout of a C program



# The call stack




The primary purpose of the call stack is to store return addresses for function calls: When a function is called:

- a return pointer is pushed on the stack.

When the function is done

- the return pointer is popped from the stack

. . . and program flow is returned to the caller, following the return pointer.

# Shell-code

The easiest way to exploit a buffer overflow bug:

- Fill the buffer with attack code
- Overwrite the return pointer to point into the array.

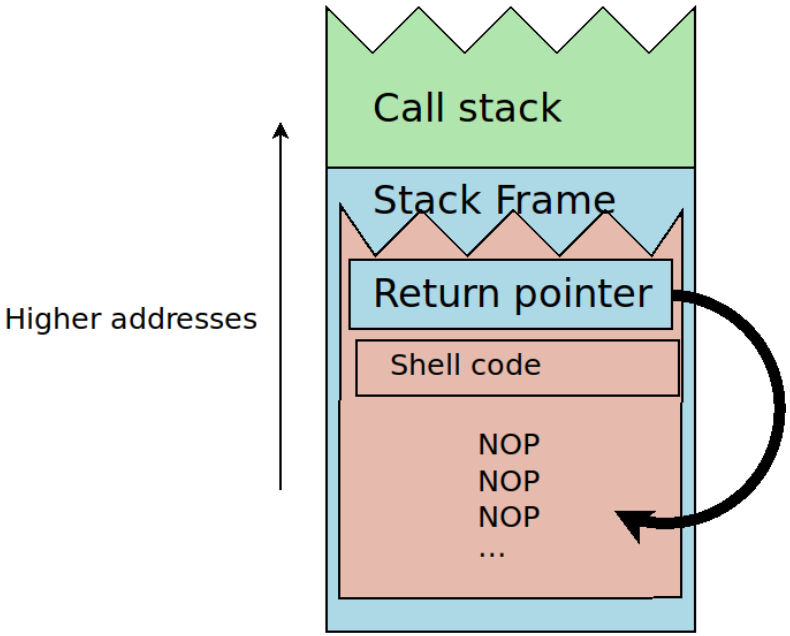
The attack code often spawns a shell (**shell code**), which gives the attacker RCE on the machine.

# NO-OP sled

**Difficulty**: Attacker does not know the address of the buffer.

# NO-OP sled

**Difficulty**: Attacker does not know the address of the buffer.




Attacker solution:

- Fill most of the buffer with NO-OPs (a NO-OP sled) and
- put shell-code at the end of the buffer.

If the attacker guesses any address in the NOP part, execution slides to the shell-code.

# Return Oriented Programming

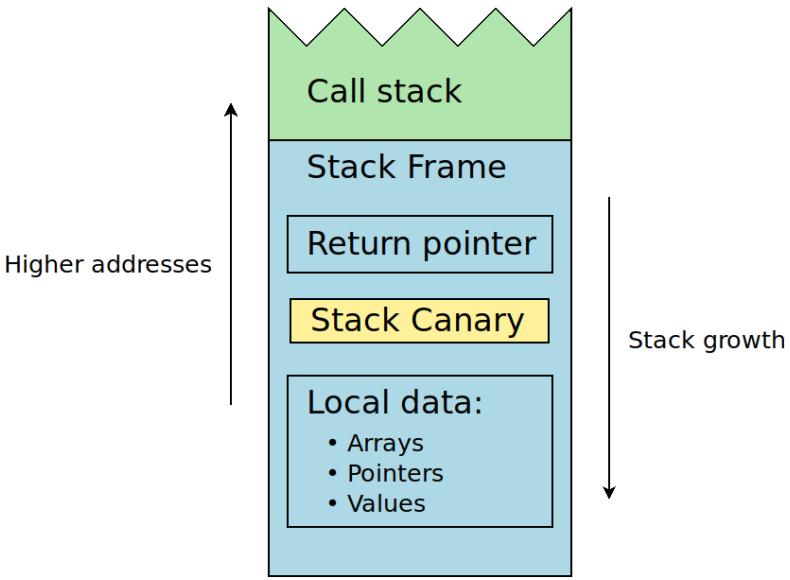
*Return Oriented Programming* (ROP) is an exploit technique using preexisting code in the program or libraries instead of uploaded shell code.

# Mitigations

How to prevent catastrophic failure?

- Write better C code.
- Static analysis.
- Stack canaries.
- W^X.
- Address space layout randomisation.

# Stack Canaries




A **stack canary** is a random integer value written after the function return pointer on the stack.
When the function returns the integer value is checked to detect if it (and thus the return pointer) has been overwritten since function call initiated.

# Address Space Layout Randomisation

*Address Space Layout Randomisation* (ASLR) refers to the practise of randomising the layout when allocating memory in the system.

**Purpose**: Making it difficult for an attacker exploiting a buffer overflow to guess the location of functions and libraries.

# Rearranging the stack

To prevent important values, arrays are put before other variables on the stack.

# W^X

Memory allocations can give the allocated memory different properties:

- Writable
- Executable

W^X (write xor executable) means that the operating system enforces that writable memory cannot be executable.

- Prevents loading shell code into writable buffers.
- Does not prevent ROP.

# Prevention

Best practice to avoid buffer overflows:

- Use memory safe languages
- Use memory-safe abstractions in unsafe languages (say vectors or smart pointers in C++)
- Use the compiler's abilities
- Run static analysers to identify potential bugs

# Memory safety

# Memory safety

A programming language is **memory safe** if each part of the program is only given access to memory locations for which they are given explicit permission.

# Example

The code in a function could access:

- Arguments from the caller: `f(x,y,z)`
- Local variables.
- Global variables.

Not, for instance, local variables of other functions.

# Breaking memory safety

- Pointer arithmetic
- Unconstrained casting
- No bounds-check on array access
- Unsafe de-allocation (dangling pointers, double free)

# Languages

We can distinguish between memory safe languages and languages with direct access to pointer arithmetic.

Memory safe:

- Java/C# (bounds check on arrays, runs on virtual machine).
- Most scripting languages (Python, JavaScript,...)
- Most functional languages (Scheme,ML,Haskell,...)

Not memory safe:

- Assembly
- C
- C++

# Achieving memory safety

Automated memory management:

- Garbage collection (LISP, Java, Haskell, Go, $\cdots$)
- Resource allocation is initialisation (RAII) and borrows checker (Rust).

# Undefined behaviour

# Undefined behaviour

**Undefined behaviour** is code which behaviour is unspecified by the language standard.

Example: In C, dereferencing `NULL` is undefined behaviour.

# Example of undefined behaviour

From the Linux kernel:

```
1 unsigned int tun_chr_poll(struct file *file,poll_table *wait){
2     struct tun_file *tfile = file->private_data;
3     struct tun_struct *tun = __tun_get(tfile);
4     struct sock *sk = tun->sk;
5     if (!tun) return POLLERR;
...
   }
```

If tun is `NULL` line 4. gives undefined behaviour.

A compiler could for example drop line 5, leading to a security vulnerability. CVE-2009-1897