INF226: Buffer overflows

Håkon R. Gylterud

Motivating examples

Java

```
import java.util.Scanner;
public class ChooseList {
        static String[] buffer = { "apple",
                                   "orange",
                                   "pear",
                                   "banana",
                                   "tangerine"};

        public static void main(String[] args) {
                System.out.println("Chose something:");
                for(int i = 0; i < buffer.length ; ++i) {
                        System.out.println((i + 1) + ". " + buffer[i]);
                }
                int choice = (new Scanner(System.in)).nextInt();

                System.out.println("Here you go: One "
                                + buffer[choice-1] + " for you!");
        }
}
```

C

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        char* buffer[5] = { "apple",
                            "orange",
                            "pear",
                            "banana",
                            "tangerine"};
        char answer[5];
        int choice;
        printf("Please make a choice:\n");
        for(int i = 0 ; i < 5; ++i) {
                printf("%d. ",i+1);
                printf("%s\n", buffer[i]);
        }
        fgets(answer,5,stdin);
        choice = atoi(answer);
        printf("Here you go: One %s for you!\n",buffer[choice-1]);
        return 0;
}
```

Questions

- What happen if we give the input “42”?
- Why is the output different for Java and C?
- Can you explain exectly the output from the C program?

Programs, processes and the run-time

Programs and processes

The term "program" can refer to either:

- the source code, or
- the executable (more commonly).

While *a process* is a specific instance of the program which is executing.

Source code

```
if (request.getMethod().equals("POST")) {
    if (request.getParameter("newforum") != null) {
        System.err.println("Crating a new forum.");
        Maybe<Stored<Forum>> forum
            = inforum.createForum((new Maybe<String> (request.getParameter("name"))).get(), con);
        response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
        response.setHeader("Location", "/forum/" + forum.get().value.handle + "/");
        baseRequest.setHandled(true);
        return ;
    }
}
```

Executable file

Programs can be created from source code in different ways:

- Compiled to machine code.
- Compiled to virtual machine code (Java VM, $\cdots$).
- Scripts are exectuted by an interpreter.

Executing a program causes a **process** to be started.

Processes

The **operating system** (OS) manages the different processes running.

Each process has

- a Process ID (which identifies the process)
- **a (virutal) memory space**
- a lot of other metadata (UID,PID,file descriptors,$\cdots$)

How the memory space is managed is called the **runtime environment**, and depends on the compiler.

How the memory space is managed is called the **runtime environment**, and depends on the compiler.

In particular, the OS **does not know about**:

- Variables
- Types
- Objects
- Scopes
- Functions

All these must be emulated using bits of memory and machine code.

Example

The binary `01100010 01100001 01101110 01101011` could be:

- the string "bank", or
- the number "1650552427" (unsigned, 32 bits, big endian)
- or a number of other things!

Runtime enviroment

The memory space of a program typically contains:

- Execution instructions (an image of the program)
- The stack:
  - data needed function calls and local variables.
  - organised in “frames”.
- The heap which contains dynamically allocated data.

But more complex languages may use additional space for organising garbage collection, green threads, etc.

The difference between Java and C

In short the difference between C and Java is:

- C provides a very thin abstraction over the OS model of a process.
- Java provides a more robust abstraction.

The guarantees provided by Java are called “memory safety” and “type safety”.

Processes interacting with the environment

Processes make **system calls** to interact with the world beyond its memory space:

- Read/write to files
- Network access
- Time functions
- Inter-proccess communication

The OS receives the call and executes the action on behalf of the process.

Processes interacting with the environment

Processes make **system calls** to interact with the world beyond its memory space:

- Read/write to files
- Network access
- Time functions
- Inter-proccess communication

The OS receives the call and executes the action on behalf of the process.

The system call interface is a security boundary.

Tools

Tools

In order to work on first mandatory assignment, you will need:

- GNU/Linux (YMMV if you use other Unix derivatives)
- A C-compiler: `gcc`
- A debugger: `gdb`
- A dissasembler: `objdump -d`

Also useful:

- `xxd`: convert back and forth between hex and bytes.
- Python
- `pwntools`: A python library which makes everything easier.
- `strace`: Make a log of the system calls made by the process.

Buffer overflow

Example

```
#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

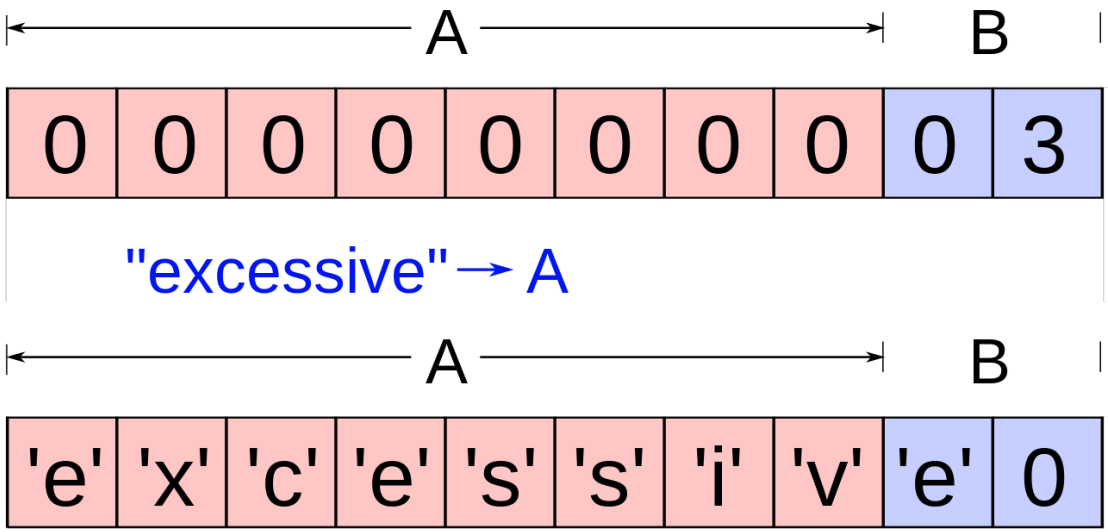


Figure 1: Buffer overflows!

In code...

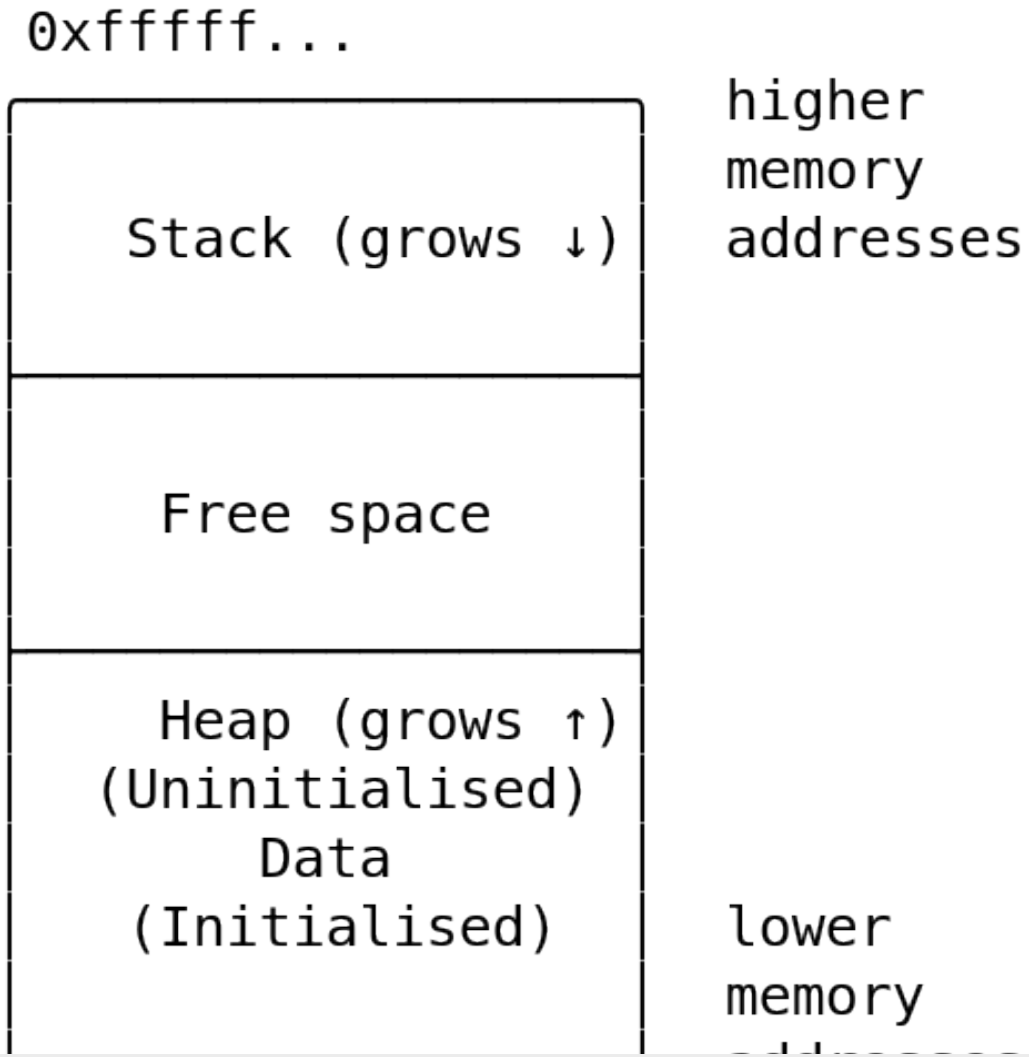
```
int main() {
    char a[] = "short";
    char b[] = "very long";
    // Copy b into a
    for (int i = 0 ; i < strlen(b) ; ++i)
      a[i] = b[i];
    printf("%s",a);
}
```

Buffer overread

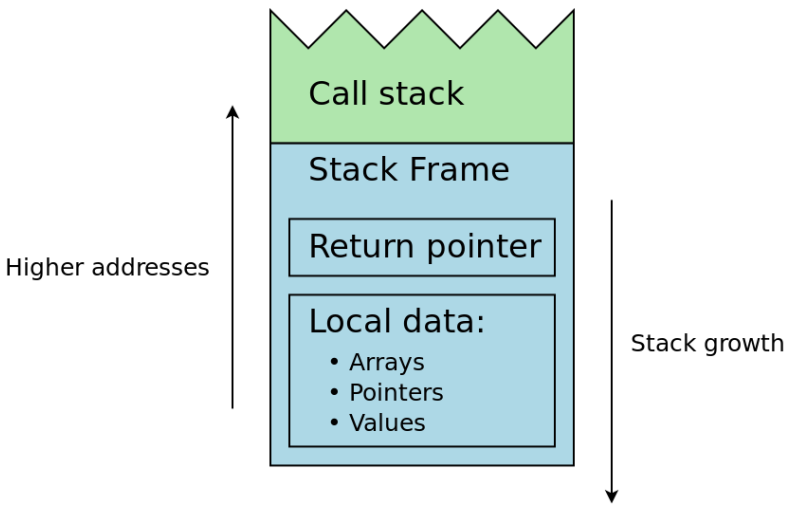
The simplest mistake one can make in an unsafe language is reading outside the bounds of a buffer (array).

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.

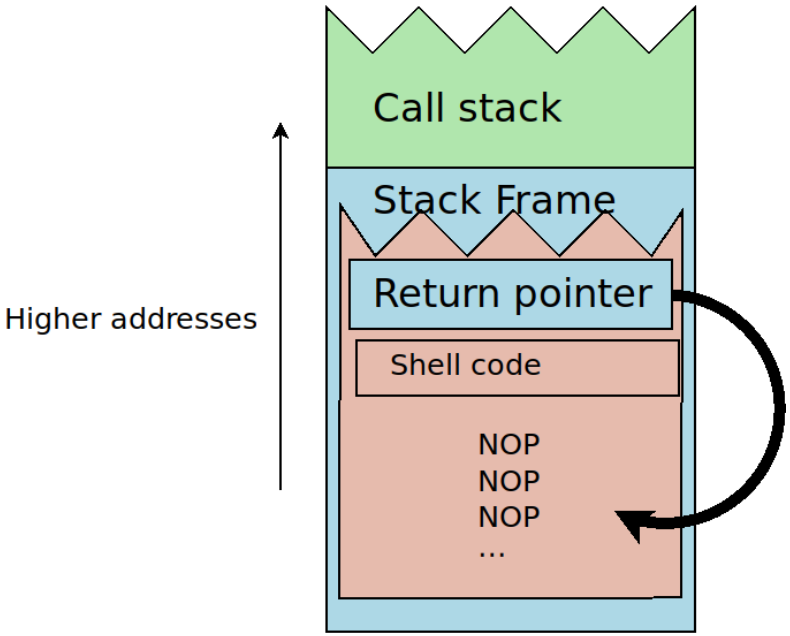
The attack code often spawns a shell (**shell code**), which gives the attacker RCE on the machine.

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.

DEMO

Mitigations

Mitigations

How to prevent catastrophic failure?

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

Mitigations

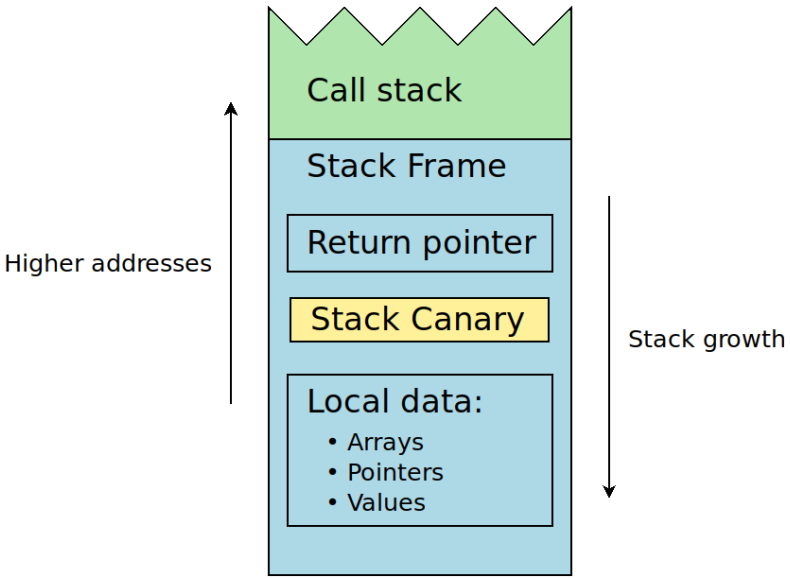
How to prevent catastrophic failure?

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

Later:

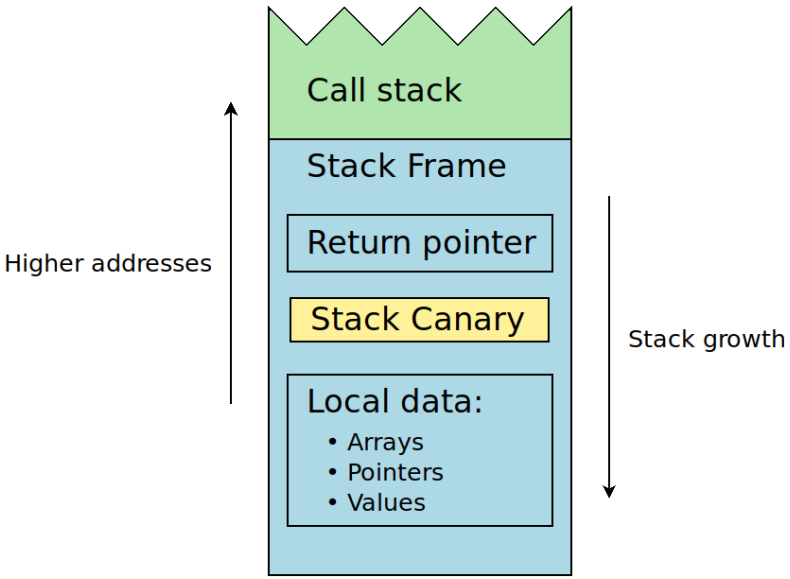
- Priviledge separation – how to limit the consequences of an remote code exploit.
- Static analysis.

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.

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.

However, if the attacker can somehow read the memory, they could get the value of the canary.

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 being overwritten, arrays are put before other variables on the stack:

1. Arrays (grow ↑)
2. Values

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
- Enable the protections allowed by the system
- Do not override protection mechanisms for convenience