

# Software Security

## *'Safe'* programming languages

Erik Poll

# Safe(r) programming languages

You can write insecure programs in ANY programming language

Still...some languages are safer than others by

to **ruling out** certain classes of bugs

making them **less likely**

make them **easier to spot**

or **mitigating their impact**



# Language-based security

Security features & guarantees provided by programming language

- Safety guarantees, incl. *memory* safety  
*initialisation* safety  
*type* safety  
*thread* safety

Many *kinds & levels*

- Access control & modularisation

Eg. *visibility/access restrictions* with eg. *public, private*

Some safety features are interdependent, eg

- *type safety* - and just about anything else - relies on *memory safety*
- *memory safety* relies on *type safety*

# Other ways the programming language can help

A programming language can also help security by

- offering good APIs/libraries, eg.
  - APIs with parametrised queries/prepared statements for SQL
  - more secure string libraries for C
- offering convenient language features
  - esp. **exceptions**, to simplify handling error conditions
- making assurance of the security easier, by
  - being able to understand code in a modular way
  - e.g. only having to review the public interface, in a code review

# General idea behind safety

Under which conditions does

`a[i] = (byte)b`

make sense?

`a` must be a non-null byte array pointing to allocated memory ;  
`i` should be a non-negative integer less than array length;  
`b` should be (castable to?) a byte and have been initialised

Two approaches

1. the programmer is responsible for ensuring these conditions  
“unsafe” approach
2. the language is responsible for ensuring them

Heated debates about pros & cons highlight tension between

flexibility, speed and control vs safety & security

But execution speed  $\neq$  speed of development of secure code  
and programmer's time may be more expensive than CPU cycles

# Safe programming languages

## Safe programming languages

- impose some **discipline or restrictions** on the programmer
- offer some **abstractions** to the programmer,  
with associated **guarantees**

This takes away some freedom & flexibility from the programmer,  
but hopefully extra safety and easier understanding makes it worth this.

# Attempts at a general definition of safety

A programming language can be considered *safe* if

1. You can trust the abstractions provided by the programming language

The programming language enforces these abstractions and guarantees that they cannot be broken

- Eg programmer does not need to know how Strings are represented

2. Programs have a precise & well defined semantics (ie. meaning)

- More generally, leaving things **UNDEFINED** in any specification is asking for security trouble

3. You can understand the behaviour of programs in a modular way

# 'safer' & 'uns safer' languages



*This is overly simplistic; there are many dimensions of safety*

Functional languages such as Haskell are safe because **data is immutable (no side-effects)** which also makes them thread-safe

# Dimensions & levels of safety

There are many dimensions of safety

memory-safety, type-safety, thread-safety, arithmetic safety;  
guarantees about (non)nullness, about immutability, about the  
absence of aliasing,...

For each dimension, there can be many levels of safety

Eg, in increasing level of safety, going outside array bounds may:

1. **let an attacker inject arbitrary code**
  2. ***possibly* crash the program (or else corrupt some data)**
  3. ***definitely* crash the program**
  4. **throw an exception, which the program can catch to handle the issue gracefully**
  5. **be ruled out at compile-time**
- 
- } 'unsafe';  
some undefined  
semantics
- } 'safe'

# Safety: how?

Mechanisms to provide safety include

- **compile time checks**, eg **type checking**
- **runtime checks**, eg **array bounds checks**, checks for **nullness**, **runtime type checks**, ...
  - for things that cannot be guaranteed by compile time checks
- **automated memory management**
  - so programmer does not have to `free()` heap-allocated data

Safe programming languages often use an **execution engine** (aka **interpreter** aka **runtime** aka **platform**) do the things above

Eg **Java Virtual Machine (VM)** or **Java Runtime Environment (JCRE)**

# Memory-safety



## America's Cyber Defense Agency

NATIONAL COORDINATOR FOR CRITICAL INFRASTRUCTURE SECURITY AND RESILIENCE

Search



[Topics](#) ▾ [Spotlight](#) [Resources & Tools](#) ▾ [News & Events](#) ▾ [Careers](#) ▾ [About](#) ▾

[Home](#) / [Resources & Tools](#) / [Resources](#)

SHARE: [f](#) [X](#) [in](#) [✉](#)

# The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

**Publish Date:** December 06, 2023

**RELATED TOPICS:** [CYBERSECURITY BEST PRACTICES](#)



## America's Cyber Defense Agency

NATIONAL COORDINATOR FOR CRITICAL INFRASTRUCTURE SECURITY AND RESILIENCE

Search



[Topics](#) ▾ [Spotlight](#) [Resources & Tools](#) ▾ [News & Events](#) ▾ [Careers](#) ▾ [About](#) ▾

[Home](#) / [Resources & Tools](#) / [Resources](#)

SHARE: [f](#) [X](#) [in](#) [✉](#)

# Exploring Memory Safety in Critical Open Source Projects

**Publish Date:** June 26, 2024

Authors:  
Tara Tarakiyee  
Hugo van de Pol

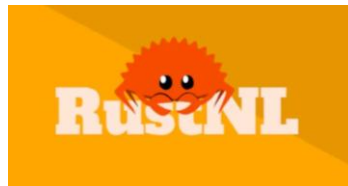
Improving  
Europe's  
cybersecurity  
posture  
through  
memory  
safety

# Rust



Real momentum building behind Rust

as the memory-safe language for low-level 'systems' software



<https://rustnl.org>

# RustWeek 2027

May 24-29 — Utrecht, NL

The world's biggest Rust conference includes one day of workshops, two days of talks, a hackathon, social activities, and more!

[Week schedule](#)[Tickets](#)

# Memory-safety – with/without initialisation safety

There are two flavours of memory safety:

A programming language is **memory-safe** if it guarantees that

1. **programs never access unallocated or de-allocated memory**
  - hence also: no segmentation faults at runtime
2. possibly also **initialisation safety**  
**program never access *uninitialised* memory**

# Memory safety

Unsafe language features that break memory safety

1. no array bounds checks
2. pointer arithmetic
3. null pointers, *if these cause undefined behaviour*
4. manual memory management with eg. malloc, new & free

Manual memory management can be avoided by

- a) not using the heap at all (eg in MISRA C)
- b) automating memory management

# Automated memory management

## 1. Garbage collection

Java, C#, Haskell

garbage collector periodically traces which objects are reachable from the root objects; unreachable objects are de-allocated

+ invisible to the programmer

-- periodic unpredictable overhead; some variants make it predictable

## 2. Automatic Reference Counting (ARC)

Objective C, Swift

Compiler adds code to keep track of how many references there are to an object; when count reaches zero, objects are de-allocated

+ no unpredictable delays

-- not invisible to programmer esp. for cyclical data structures

## 3. Ownership

Rust, Swift

Objects de-allocated when the owner 'dies'

+ no unpredictable delays

-- not invisible to the programmer

they need to be aware of ownership;

as with ARC, cyclical data structures need special attention

Smart pointers provide ARC & Ownership in C++

# Remaining buffer overflow issues in Java or C#

Buffer overflows can still exist, namely:

1. **in native code**, eg for graphics libraries
2. for C#, in code blocks declared as **unsafe**
3. through **bugs in the Virtual Machine (VM)** implementation, which is typically written in C++....
4. through **bugs in the implementation of the type checker** or, worse, **bugs in the type system (unsoundness)**

The VM (incl. the type checker aka bytecode verifier) is part of the **Trusted Computing Base (TCB)** for memory and type-safety

Hence 3 & 4: bugs in this TCB can break these properties.

# Type-safety

# Guarantees that programming language can provide

```
public class Demo{  
    static private string greeting = "Hello";  
    final static int ANSWER = 42;  
  
    static void Main (string[] args){  
        foreach (string name in args){  
            Console.WriteLine(sayHello(name));  
        }  
    }  
  
    public static string sayHello(string name){  
        return greeting + name;  
    }  
}
```

greeting only accessible  
in class Demo

CONST will *always* be 42

sayHello will always  
return a string

sayHello will always be called  
with 1 parameter of type string

about **typing**, **immutability** & **visibility**

# Types

- **Types** assert invariant properties of program elements. Eg
  - This variable will always hold an integer
- **Type checking** verifies these assertions. This can be done
  - at compile time (static typing) or
  - at runtime (dynamic typing)or a combination.
- **Type safety** (aka **type soundness** or **strong typing**)  
guarantee that these assertions hold at run-time

# Type safety & representation independence

*Can we tell if `true` is represented as `0x00` and `false` as `0xFF` or vice versa?*

*Can we tell if `Strings` are represented as null-terminated character arrays or as records with a length field?*

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| h | e | l | l | o | \0 |
| 5 | h | e | l | l | o  |

If we can't tell how data structures are represented 'under the hood' then we have representation independence

representation independence => type safety

# Combinations of memory & type safety

Programming languages can be

- memory-safe, typed, and type-safe:
  - Java, C#, Rust, Go
    - some with loopholes to allow unsafety
  - Functional languages such as Haskell, ML, Clean, F#
- memory-unsafe, typed, and type-unsafe
  - C, C++

Not type sound: using pointer arithmetic or unsafe union types, you can break any type guarantees

- memory-safe and untyped (unusual)
  - LISP, Prolog

Memory-safety and type-safety are related

- without memory safety, ensuring type safety is impossible
- without some type safety, memory safety is usually impossible

# Richer type systems: non-null vs nullable

A type system can distinguish **non-null** and **nullable** types

```
public @NonNull String hello = "hello";
```

- to improve efficiency
- to prevent null pointer bugs or detect them earlier, at compile time

Support for this has become more mainstream:

- In Java you can use type annotations `@Nullable` and `@NonNull`
- C# supports nullable types, written as `A?` or `Nullable<A>`
- **Scala**, **Rust**, **Kotlin**, **Swift**, and **Ceylon** have non-null vs nullable aka **option(al)** types
- `__nullable` and `__nonnull` annotations in C/C++ code

Such languages may make references **non-null by default**

# Run & compile time checks in Java or C#

Ruled out at language-level, by combination of

- **compile-time typechecking** (**static** checks)
  - or at **load-time**, by bytecode verifier (bcv)
- **runtime checks** (**dynamic** checks)

What runtime checks are performed when executing the code below?

```
public class A extends Super{  
    protected int[] d;  
    private A next;  
  
    public A() { d = new int[3]; }  
    public void m(int j) { d[0] = j; }  
    public setNext(Object s)  
        next = (A)s;  
    }  
}
```

**Runtime checks for**  
1) non-nullness of d and  
2) array bound.

**runtime check for**  
type (down)cast

## Run & compile time checks in Java or C#

*Can we omit the array bound check in `m(int j)` ?*

*Given that `d` has length 3 ?*

*No, because a subclass of `A` can change the value of `d`*

```
public class A extends Super{  
    protected int[] d;  
    private A next;  
  
    public A() { d = new int[3]; }  
    public void m(int j) { d[0] = j; }  
    public setNext(Object s)  
        next = (A)s;  
}  
}
```

Runtime check  
if 0 is within  
array bounds

# Breaking type safety with unsafe union types

```
struct user{
    bool is_enrolled;
    bool has_paid_tuition_fee;
    union {
        char* name; // for users that have no uid yet
                    // "OR"
        int uid;    // for users that have a uid
    };
};

...
struct user u;
u.is_enrolled = true; u.uid = 250012;
printf("Username is %s \n", u.name);
```

- **Unions** aka **variants** in C/C++ and some other languages are **type unsafe**  
Behaviour of code above is unpredictable & depends on underlying data representations; it may allow pointer arithmetic
- There are also **type-safe union/variant** constructs;  
then accessing a union requires code for all cases.  
Eg the class template `std::variant` in C++ is type-safe

## Breaking type safety with casts

```
class DiskQuota {
    private:
        int MinBytes;
        int MaxBytes;
};

void EvilCode(DiskQuota* quota) {
    // use pointer arithmetic to access
    // the quota object in any way we like!
    ((int*)quota)[1] = MAX_INT;
}
```

For a C(++) program we can make *no guarantees whatsoever* in the presence of untrusted code

So

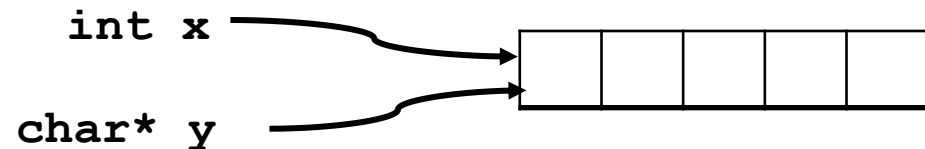
- a buffer overflow in some library can be fatal
- in a code review we have to look at *all* code to make guarantees

# Breaking type safety?

Type safety is an extremely **fragile** property:

one tiny flaw brings the whole type system crashing down

Data values and objects are just blobs of memory. If we can create **type confusion**, by having **two references with different types pointing the same blob of memory**, then *all* type guarantees are gone.



Example: type confusion attack on Java in Netscape 3.0:

```
public class A[] { ... }
```

Its Java execution engine confused the type **A[]** with the type **array of A**

This is an **input validation** problem:

[ and ] should not be allowed in class names

# Type confusion attacks

```
public class A{  
    private Object x;  
    ...  
}
```

What if we could compile **B** against **A**  
but we run it against **A**?

*We can do pointer arithmetic again!*

If Java Virtual Machine would allow such  
so-called *binary incompatible* classes  
to be loaded, the whole type system  
would break.

```
public class A{  
    public int x;  
    ...  
}  
  
public class B{  
    void setX(A a) {  
        a.x = 12;  
    }  
}
```

# Custom types for security

Programmer can define **custom types for their own security needs**

*What data type would you use for an email adress?*

Obvious answer is `String`, but better alternatives are

`ValidEmailAddress`

`ValidEmailAddressOfExistingUser`

`ValidEmailAddressOfAuthenticatedUser`

We can use types to track all kinds of safety guarantees, or lack thereof:

- this variable holds untrusted user data
- this variable holds a valid URL, but it may be user supplied
- this variable holds data that is user data has been HTML-escaped so that it safe to use inside an HTML context

More on **input sanitisation & validation** in a few weeks

## Other language guarantees: **visibility & immutability**

- **visibility:** public, private, etc
  - eg private fields not accessible from outside a class
- **immutability**

- of **primitive values (ie constants)**

- in Java: `final int i = 5;`
- in C(++): `const int BUF_SIZE = 128;`

Beware: meaning of `const` is confusing for C(++) pointers & objects!

- of **objects**

- In Java, for example `String` objects are immutable

Scala, Rust, Ceylon, and Kotlin provide a distinction between mutable and immutable data to promote the use of immutable data structures

In functional programming languages data structures are always immutable.

## Other language guarantees: **safe(r) arithmetic**

What happens if `i=i+1` ; overflows?

*What would be unsafe or safe(r) approaches?*

1. *Unsafest approach* : leaving this as undefined behavior
  - eg **C** and **C++**
2. *Safer approach* : specifying how over/underflow behaves  
for instance based on 32 or 64 bit two-complements behaviour
  - eg **Java** and **C#**
3. *Safer still* : integer overflow results in an exception
  - eg **checked mode in C#**
4. *Safest*: have infinite precision integers, so overflow never happens

**Python** and functional programming languages like **Haskell** have infinite precision integers.

There have been experiments with infinite precision reals, but no mainstream programming languages provide these AFAIK.