Software Security

Language-based Security: *'Safe'* programming languages

[Chapter 3 of lecture notes on language-based security]

Erik Poll

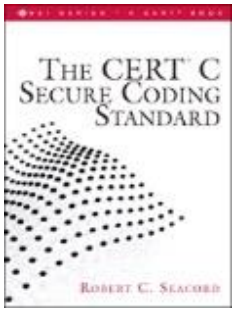
Radboud Universiteit Nijmegen

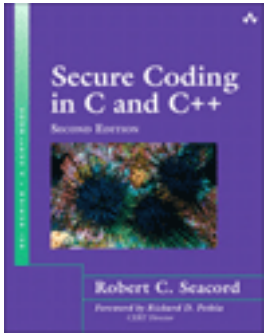
Producing more secure code

1. You can try to produce more secure C(++) code.

   Not just using SAST & DAST tools, but more importantly by reading – or making other people read – books like

   - CERT secure coding guidelines for C and C++
     at http://www.securecoding.cert.org





2. More structural way to improve security:
   improve the programming language

   – not just to prevent memory corruptions flaws, but other common problems too…

Language-based security

Security features & guarantees provided by programming language

- safety guarantees,
  incl. memory-safety, type-safety, thread-safety

  There are many flavours & levels of 'safety' here.
  Eg. different type systems give different notions of type-safety.

- forms of access control
  - visibility/access restrictions with eg. `public, private`
  - sandboxing mechanisms inside programs
- forms of information flow control

Some features depend on each other, eg

- type safety & just about anything else relies on memory safety
- sandboxing relies on memory & type safety

This week: safety. See course lecture notes, chapters 2 & 3

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
  - only having to review the public interface, in a code review

  These properties *require* some form of safety

'Safe' programming languages?

You can write insecure programs in ANY programming language.

Eg

- **You can forget or screw up forget input validation in any language**
- **Flaws in the program logic can never be ruled out**

Still...some safety features can be nice:

to *prevent* certain classes of bugs

or at least *mitigate their impact*

General idea behind safety

Under which conditions does

```
a[i] = (byte)b
```

make sense?

`a` must be a non-null byte array;
`i` should be a non-negative integer less then array length;
`b` should be (castable to?) a byte

Two approaches

1. the programmer is responsible for ensuring these conditions
   "unsafe" approach
2. the language is responsible for checking this
   "safe" approach

Heated debates about th pros & cons highlight tension between

flexibility, speed and control vs safety & security

But execution speed ≠ speed of development of secure code

and maybe programmers are more expensive the 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 a `boolean` is either `true` or `false`, and never `23` or `null`
   - Programmer doesn't have to care if `true` is represented as `0x00` and `false` as `0xFF` or vice versa

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' & 'unsafer' languages

Warning: this is overly simplistic, as there are many dimensions of safety

Functional languages such as Haskell are safe because data is immutable (no side-effects)

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)

} 'unsafe'; some undefined semactics

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

} 'safe'

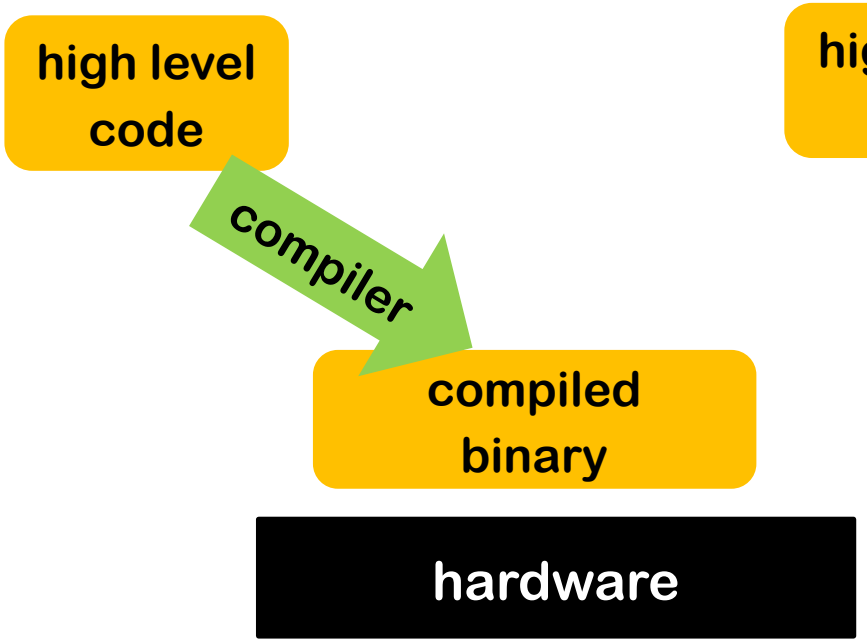
Safety: how?

Mechanisms to provide safety include

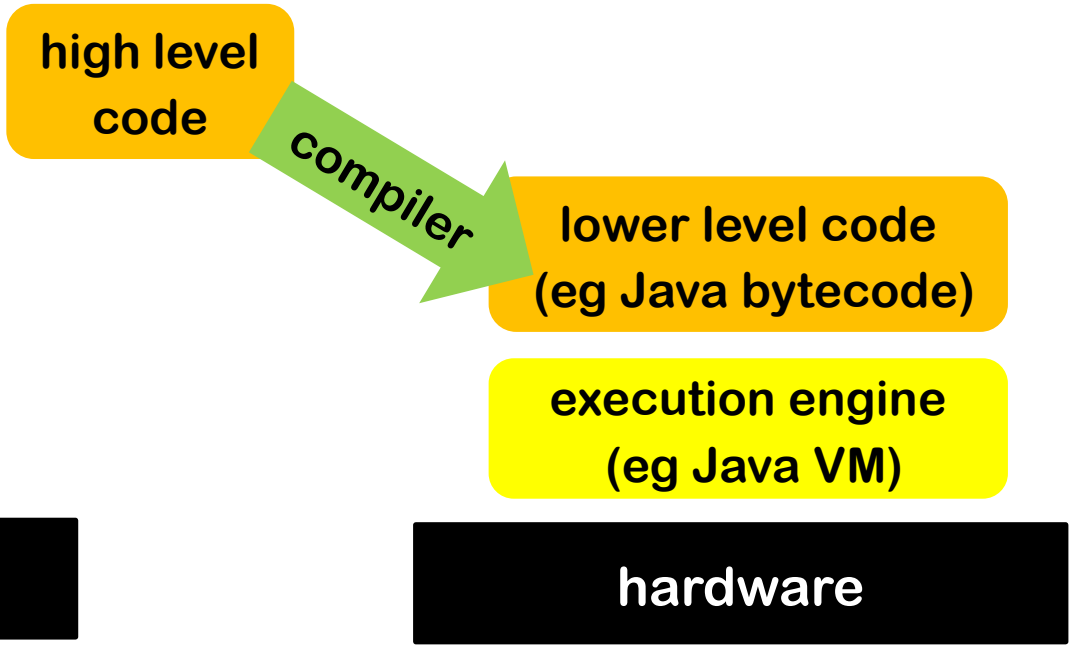
- compile time checks, eg type checking
- runtime checks, eg array bounds checks, checks for nullness, runtime type checks, ...
  - for things that cannot be guaranteed by compile time cheks
- automated memory management using a garbage collector
  - so programmer does not have to `free()` heap-allocated data
- using an execution engine, to do the things above
  - Eg the Java Virtual Machine (VM), which
    - runs the bytecode verifier (bcv) to type-check code,
    - performs some runtime checks
    - periodically invokes the garbage collector

Compiled vs interpreted code

Compiled binary runs on bare hardware



Execution engine (aka 'runtime') isolates code from hardware



Any defensive measures have to be compiled into the code.

The programming language / platform still 'exists' at runtime, and the execution engine can provide checks at runtime

Memory-safety

Memory-safety – two different flavours

A programming language is memory-safe if it guarantees that

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

Here

1. means we could switch off OS access control to memory.
   Assuming there are no bugs in our execution engine...
2. means we don't have to zero out memory before de-allocating it to avoid information leaks (within the same program).
   Again, assuming there are no bugs in our execution engine...

Memory safety

Unsafe language features that break memory safety

- not having array bounds checks
- allowing pointer arithmetic
- null pointers, *but only if these cause undefined behaviour*

Null pointers in C

Common (and incorrect!) folklore:

dereferencing a NULL pointer will crash the program.

But, the C standard only guarantees

the result of dereferencing a null pointer is undefined.

So it *may* crash the program, but 

might happen

See the CERT Secure Coding guidelines for C

https://www.securecoding.cert.org/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointer

for discussion of a security vulnerability in a PNG library caused by a null dereference that didn't crash (on ARM processors).

Memory safety

Unsafe language features that break memory safety

- **no array bounds checks**
- **pointer arithmetic**
- **null pointers, *but only if these cause undefined behaviour***
- **manual memory management**

Manual memory management can be avoided by

1. **not using the heap at all (eg in MISRA C), or**
2. **automating it with a garbage collector**
    1. **Garbage collection first used in LISP in 1959, and went mainstream with Java in 1995**
3. **There are ways to automate memory management without a garbage collection, eg. using ownership type systems, as in Rust**

Type-safety

Types

- Types assert invariant properties of program elements. Eg
  - This variable will always hold an integer
  - This function will always return an object of class X (or one of its subclasses)
  - This array will never store more than 10 items

  NB there is a *wide range of expressivity* in type systems!

- Type checking verifies these assertions. This can be done
  - at compile time (static typing) or
  - at runtime (dynamic typing)

  or a combination.

- Type soundness (aka type safety or strong typing)
  A language is type sound if the assertions are guaranteed to hold at run-time

Type information and – ideally - *guarantees*

```
public class Demo{
   static private string greeting = "Hello";
   final static int CONST = 43;

   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 43

sayHello will always return a string

sayHello will always be called with 1 parameter of type string

Type-safety

Type-safety programming language guarantees that programs that pass the type-checker can only manipulate data in ways allowed by their types

- **So you cannot multiply booleans, dereference an integer, take the square root of reference, etc.**

  NB: this removes lots of room for undefined behaviour

- **For OO languages: no “Method not found” errors at runtime**

Combinations of memory & type safety

Programming languages can be

- memory-safe, typed, and type sound:
  - Java, C#, Rust, Go
    - though some of these have loopholes to allow unsafety
  - Functional languages such as Haskell, ML, Clean, F#
- memory-safe and untyped
  - LISP, Prolog, many interpreted languages
- memory-unsafe, typed, and type-unsafe
  - C, C++

  Not type sound: using pointer arithmetic in C, you can break any guarantees the type system could possibly make

More generally: without any memory safety, ensuring type safety is impossible.

Example – breaking type soundness in C++

```
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;

}
```

NB 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

Ruling out buffer overflows 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

Remaining buffer overflow issues in Java or C#

Buffer overflows can still exist, namely:

1. in native code
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 byte code verifier) is part of the ***Trusted Computing Base (TCB) for memory and type-safety***,

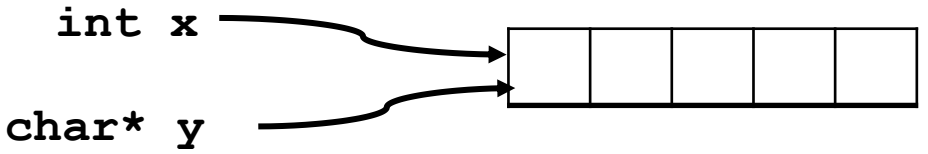
Hence 3 & 4: bugs in this TCB can break these properties.

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[]{ ... }
```

Netscape's Java execution engine confused this type `A[]` with the type array of `A`

Root cause: [and] should not be allowed in class names

So this is an input validation problem!

Type confusion attacks

```
public class A{

    public Object x;

     ...

}
```

```
public class A{

  public int x;

  ...

}

public class B{

 void setX(A a) {

  a.x = 12;

 }

}
```

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.

Safe 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
   - eg 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 & reals, 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.

How *rich* aka *expressive*
can we make type systems?

Ongoing evolution to richer types: non-null vs nullable

Many ways to enrich type systems further, eg

- Distinguish non-null & possibly-null (aka nullable) types

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

  - to improve efficiency
  - to prevent null pointer bugs or detect (some/all?) of them earlier, at compile time
- Support for this has become mainstream:
  - C# supports nullable types written as `A?` or `Nullable<A>`
  - In Java you can use type annotations `@Nullable` and `@NonNull`
  - Scala, Rust, Kotlin, Swift, and Ceylon have non-null vs nullable aka option(al) types
- Typically languages then take the approach that references are non-null by default (as PREfast did)

Ongoing evolution to richer type systems: aliasing & information flow

- **Alias control**
  restrict possible interferences between modules due to aliasing.
  - **More on the risk of aliasing later this lecture**
- **Information flow**
  controlling on the way tainted information flows through an implementation.
  - **More on type systems for information flow in later lectures.**