Software Security

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

(continued)

Erik Poll

Radboud Universiteit Nijmegen

Recap last week: language-based security

- Safety guarantees *in* the programming language can improve security, incl.
  - omitting insecure features
    - eg pointer arithmetic, malloc & free, …
  - adding secure features
    - eg checks for array bounds & non-null, **types**, having exceptions, convenient security APIs (eg for access control)
- Safety guarantees checked by mixture of
  - compile-time checks (or load-time)
    - for typing (incl. static casts) , uncaught exceptions, …
  - run-time checks
    - for array bounds, non-nullness, dynamic casts

Safe arithmetic

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

NB *depends* on programming language; can be defined or undefined

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**
   - **(experimental) support in some functional programming languages**

Typing:

how do we know that the type system is *sound*?

Type confusion bugs

If the type system is not sound (as in C/C++), type confusion is an important source of security flaws

Eg look for CVEs of category CWE-843 Access of Resource Using Incompatible Type ('Type Confusion') or mentioning ’type confusion’at https://nvd.nist.gov/search

CWE-843 is relatively new, so older type confusions bugs will be classified under different CWEs

How do we know a type system is sound? (1)

- Representation independence (for booleans)

  it does not matter if we represent `true` as 0 and `false` as 1 (or 0xFF), or vice versa

  - if we execute a given program with either representation, the result is guaranteed to be the same

We could test this or *prove* it:

> *Given a formal mathematical definition of the programming language, we could prove that it does not matter how* `true` *and* `false` *are represented for all programs*

Similar properties should hold for all datatypes.

More on such formal methods in Program Verification with Types and Logic by Robbert Krebbers

How do we know type system is sound? (2)

Give two formal definitions of the programming language

- a typed operational semantics,, which records and checks type information at runtime.

  Effectively, a 'paranoid' operational semantics

- an untyped operational semantics, which does not

and prove their equivalence for all well-typed programs.

Or, in other words, prove the equivalence of

- a defensive execution engine (which continuously (re)checks *all* type information at runtime) and
- a normal execution engine which does not

for any program that passes the type checker.

People have formalised the semantics and type system of eg Java, using theorem provers (Coq, Isabelle/HOL), to prove such results.

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

Ongoing evolution to richer types: non-null vs nullable

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

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

- to prevent null pointer bugs or detect (some/all?) of them earlier, at compile time
- to improve efficiency (and remove runtime non-null checks, but dangerous, esp. in concurrent setting)

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.

Other language-based guarantees

- 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 provides a more systematic distinction between mutable and immutable data to promote the use of immutable data structures

Thread-safety

&

Aliasing

Problems with threads (ie. lack of thread safety)

- Two concurrent execution threads both execute the statement

```
x = x+1;
```

where `x` initially has the value 0.

What is the value of `x` in the end?

  – Answer: `x` can have value 2 or 1

- The root cause of the problem is a data race:
  `x = x+1` is *not* an atomic operation, but happens in two steps - reading `x` and assigning it the new value - which may be interleaved in unexpected ways

- Why can this lead to security problems?

Think of internet banking, and running two simultaneous sessions with the same bank account… *Do try this at home!* ☺

Weird multi-threading behaviour in Java

```
class A {
    private int i ;
    A() { i = 5 ;}
    int geti() { return i; }
}
```

Can geti() ever return something else than 5?

Yes!

Thread 1, initialising x

```
static A x = new A();
```

Thread 2, accessing x

```
j = x.geti();
```

You'd think that here x.geti() returns 5 or throws an exception, depending on whether thread 1 has initialised x

Hence: x.geti() in thread 2 can return 0 instead of 5

Execution of thread 1 takes in 3 steps

1. allocate new object m
2. m.i = 5;
3. x = m;

the compiler or VM is allowed to swap the order of these statements, because they don't affect each other

Weird multi-threading behaviour in Java

```
class A {
    private final int i ;
    A() { i = 5 ;}
    int geti() { return i;}
}
```

Now geti() always return 5.

Declaring a private field as final fixes this particular problem

- this is a totally ad-hoc fix; the JVM spec includes some ad-hoc restrictions on the initialisation of `final` fields

- A revision of the Java Memory Model specifies how compilers & VM (incl. underlying hardware) can deal with concurrency, in 2004.
- The API implementation of String was only fixed in Java 2 (aka 1.5)

Data races and thread-safety

- A program contains a data race if two execution threads simultaneously access the same variable and at least one of these accesses is a write

  NB data races are highly non-deterministic, and a pain to debug!

- thread-safety = the behaviour of a program consisting of several threads can be understood as an interleaving of those threads
- In Java, the semantics of a program with data races is effectively undefined, i.e. only programs without data races are thread-safe

  Moral of the story:

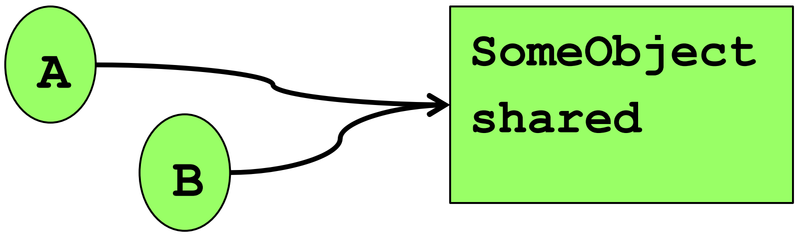
  Even purportedly "safe" programming languages can have very weird behaviour in presence of concurrency

- The programming language Rust aims to guarantee the absence of data races, i.e. thread-safety, at the language level
- Other modern programming language are also introducing features to help with thread safety, e.g. `@ThreadLocal` annotations in Kotlin

Why things often break in C(++), Java, C#, ...

Dangerous combination: ALIASING & MUTATION

Aliasing: two threads or objects A and B both have a reference to the same object `shared`



This is the root cause of many problems, not just with concurrency

1. in concurrent (aka multi-threaded) context: data races
   - Locking objects (eg `synchronized` methods in Java) can help, but: expensive & risk of deadlock
2. in single-threaded context: dangling pointers
   - Who is responsible for free-ing `shared` ? A or B?
3. in single-threaded context: broken assumptions
   - If `A` changes the `shared` object, this may break `B`'s code, because `B`'s assumptions about `shared` are broken

References to mutable data are dangerous

In multi-threaded programs, aliasing of mutable data structures can be problematic, as the referenced data can change,

- **even in safe programming languages such as Java or C# !**

```
public void f(char[] x){
  if (x[0] != 'a') { throw new Exception(); }
  // Can we assume that x[0] is the letter 'a'  here?
  // No!! Another concurrent execution thread could
  //       change the content of x at any moment
```

If there is aliasing, another thread can modify the content of the array at any moment.

References to *im*mutable data are *less* dangerous

In a multi-threaded program, aliasing of immutable data structures are safer.

```
public void f(String x){
  if (x.charAt(0) != 'a') { throw new Exception(); }
  // We CAN assume that x[0] is the letter 'a‘ here?
  // Yes, as Java Strings are immutable
    ...
```

Another thread with a reference to the same string *cannot* change the value (or ‘contents’) of the string, as Java strings are immutable.

Kotlin has annotation `@SharedImmutable` to explicitly mark objects as being immutable & (therefore) safe to share

Spot the security flaw

```
package java.lang;

   public class Class {

     private String[] signers; /** List of signers of this code */

     public String[] getSigners()  { return signers;   }

     ...
```

Class-object leaks a reference to security-critical, internal, mutable data structure `signers`

Implementation of the class `Class` in JDK1.1.1 was broken in this way (aka Magic Cloak attack)

Security flaw in code signing check (Magic Coat)

```
package java.lang;
   public class Class {
     private String[] signers;
     public String[] getSigners()    { return signers;   }
   ...
```

How can this bug be fixed ?

`getSigners` should clone the array and return a clone

Could it be prevented at language-level ?

- By having immutable arrays
- By a type system for alias control
  - by new modifier `private_and_unshareable`