Software Security

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

(continued)

Erik Poll

Radboud Universiteit Nijmegen

Safe(r) programming languages

Last week

- memory-safety 2 kinds: to ensure only 'legal' memory access,
  or also ensure only access to initialized memory
- type-safety: ensuring a different kind of 'legal' memory access

Today

- safe(r) integer arithmetic
- type-safety continued: type confusion
- visibility / encapsulation
- more expressive type systems
- thread safety, aliasing & immutability
- compartmentalisation

Safe arithmetic

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

What would be unsafe or safe(r) approaches?

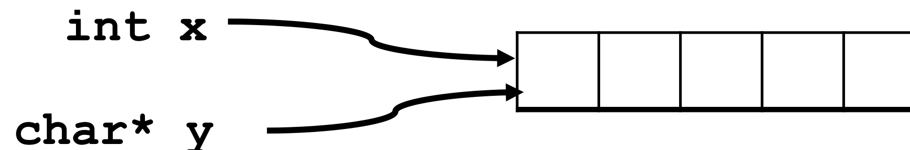
1. *Unsafest approach* : leaving this as undefined behavior
   – eg C and C++
2. *Safer approach* : specifying how over/underflow behaves
   – eg based on 32 or 64 bit two-complements behaviour
   – eg Java and C#
3. *Safer still* : integer overflow results in an exception
   – eg checked mode in C#
4. *Safest*: have infinite precision integers, 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 as far as I know.

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;
     ...
}
```

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;
 }
}
```

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.

Other language-based (type) 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 provide a more systematic distinction between mutable and immutable data to promote the use of immutable data structures

In functional programming languages data structures are always immutable.

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
In some languages (eg. Java) `x` can have any value

- 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:

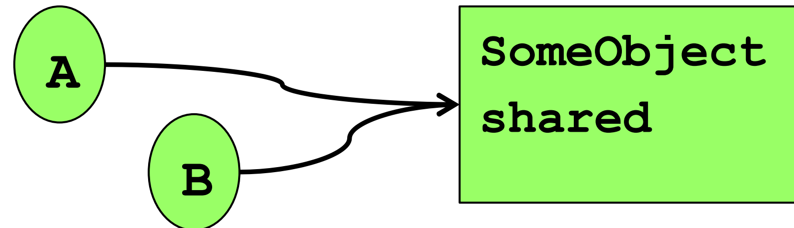
  Even purportedly "safe" programming languages can have very weird behaviour in presence of concurrency

- 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# !

```
1  public void f(char[] x){
2    if (x[0] != 'a') { throw new Exception(); }
3    // Can we assume that x[0] is the letter 'a'  here?
4    // No!! Another concurrent execution thread could
5    //       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.

```
1   public void f(String x){
2    if (x.charAt(0) != 'a') { throw new Exception(); }
3    // We CAN assume that x[0] is the letter 'a‘ here?
4    // Yes, as Java Strings are immutable
5      ...
```

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

Non-atomic check and use
aka
TOCTOU (Time of Check, Time of Use)
or
Race conditions

A classic source of (security) problems

- Race condition aka data race is a common type of bug in concurrent programs
  - Basically: two execution threads mess with the same data or object (program variable, file, ...) at the same time
  - Not necessarily a *security* bug, but it can be...

- Non-atomic check and use

  aka TOCTOU (Time Of Check, Time of Use)

  is a closely related type of security flaw

  Problem: some precondition required for an action is invalidated between the time it is checked and the time the action is performed
  - Typically, this precondition is access control condition
  - Typically, it involves some concurrency

Classic UNIX race condition

`lpr -r`

- Print utility with `-r` option to remove file after printing
- Could be used to delete arbitrary files

  How?

  1. User executes `lpr -r symlink`
     where `symlink` is a symbolic link
  2. OS checks that user has permission to read & delete this file
  3. While the file is printing move the link is moved, eg to `/etc/passwd`
  4. after printing `lpr`, which has *root permission*, deletes `/etc/passwd`

  Root of the problem: time between check (2) and use (4)

Learning from past mistakes?

`lpr -r` is a classic security flaw from the 1970s, but similar flaws happen decades later

CVE-2003-1073
A race condition in the `at` command for Solaris 2.6 through 9 allows local users to delete arbitrary files via the -r argument with .. sequences in the job name, then modifying the directory structure after at checks permissions to delete the file and before the deletion actually takes place

Combination of race condition with failure to check that file names do not contain ..

Another classic: `mkdir` on Unix

- `mkdir` creates a new directory/folder
- This program executes as root
  - in Linux terminology, it is `setuid root`

- It creates new directory *non-atomically*, in several steps:
  1. enter super-user mode
  2. creates the directory, with owner is root
  3. sets the owner, to whoever invoked `mkdir`
  4. exit super-user mode

- Attack: by creating a symbolic link between steps 2 and 3, attacker can own any file

Example race condition

```
const char *filename="/tmp/erik";
if (access(filename, R_OK)!=0) {
   ... // handle error and exit;
}
// file exists and we have access
int fd open (filename, O_RDONLY);
...
```

Between calls to `access` and `open` the file might be removed, or a symbolic link in the path might be reset!

Race condition & file systems

Interaction with the file system is common source of TOCTOU issues

Signs of trouble:

- Access to files using filenames rather than file handles or file descriptors
  - filenames may point to different files at different moments in time
- Creating files or directories in publicly accessible places, for instance `/tmp`
  - especially if these have predictable file names

Spot the race condition!

```
public class SimpleServlet extends HttpServlet {
  private String query;
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                 throws ServletException, IOException {
    try { Connection conn =
            DriverManager.getConnection("jdbc:odbc ... ");
        query = "INSERT INTO roles" + "(userId, userRole)" +   "VALUES "  + "('"  +
              request.getParameter("userId") + "','" +
              "'standard')";
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(query);
        } catch ...
  }
```

Spot the race condition!

```
public class SimpleServlet extends HttpServlet {
  private String query;
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                throws ServletException, IOException {
    try { Connection conn =
            DriverManager.getConnection("jdbc:odbc ... ");
         query = "INSERT INTO roles" + "(userId, userRole)" +   "VALUES "  + "('"  +
              request.getParameter("userId") + "'," +
              "'standard')";
        Statement stmt = conn.createStatement();
        stmt.executeUpdate(query);
        } catch ...
  }
```

Concurrent calls of doGet will act on the *same* Servlet object and hence use the *same* instance field `query`

Spot the race condition!

```
public class SimpleServlet extends HttpServlet {
  private String query;
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                throws ServletException, IOException {
   String query;
   try { Connection conn =
           DriverManager.getConnection("jdbc:odbc ... ");
     query = "INSERT INTO roles" + "(userId, userRole)" +   "VALUES "  + "('"  +
             request.getParameter("userId") + "','" +
             "'standard')";
     Statement stmt = conn.createStatement();
     stmt.executeUpdate(query);
     } catch ...
  }
```

Fix: now every (possibly concurrent) call of doGet has its own `query` field

One account. All of Google.

Sign in to continue to Gmail

Email

Password

Sign in

Stay signed in

Need help?

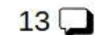
Create an account

Edge & Safari GUI bug [CVE-2018-8383]

Security

Safari, Edge fans: Is that really the website you think you're visiting? URL spoof bug blabbed

Egghead says Apple has yet to patch spoofing vulnerability

By Shaun Nichols in San Francisco 11 Sep 2018 at 05:01 13  SHARE ▼

URL in address bar can be spoofed with a race condition:

JavaScript code loads legitimate page; changes address bar, but over non-existent port; and then quickly loads another page

https://www.theregister.co.uk/2018/09/11/safari_edge_spoofing/

https://youtu.be/Ni2XzF5-ixY

https://youtu.be/dGJSsK55nfQ