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 'legal' memory access,
  or also ensure access to initialized memory

- type-safety
- safe(r) integer arithmetic

Today

- visibility / encapsulation
- immutability (of data structures)
- compartmentalisation

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

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 `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:

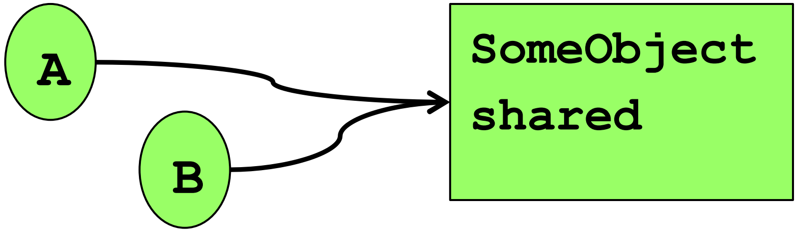
  Even purportedly "safe" programming languages can have very weird behaviour in presence of concurrency

- 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# !

```
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.

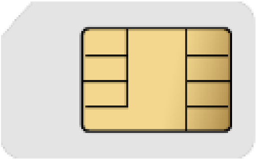
```
public void f(String x){
  if (x.charAt(0) != 'a') { throw new Exception(); }
  // We CAN assume that x[0] is the letter 'a` here?
  // Yes, as Java Strings are immutable
     ...
```

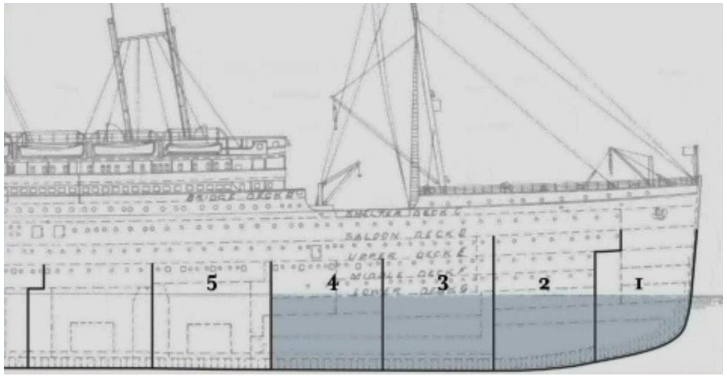
Another thread with a reference to the same string *cannot* change the value (or ‘contents’) of the string, as Java strings are immutable.

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

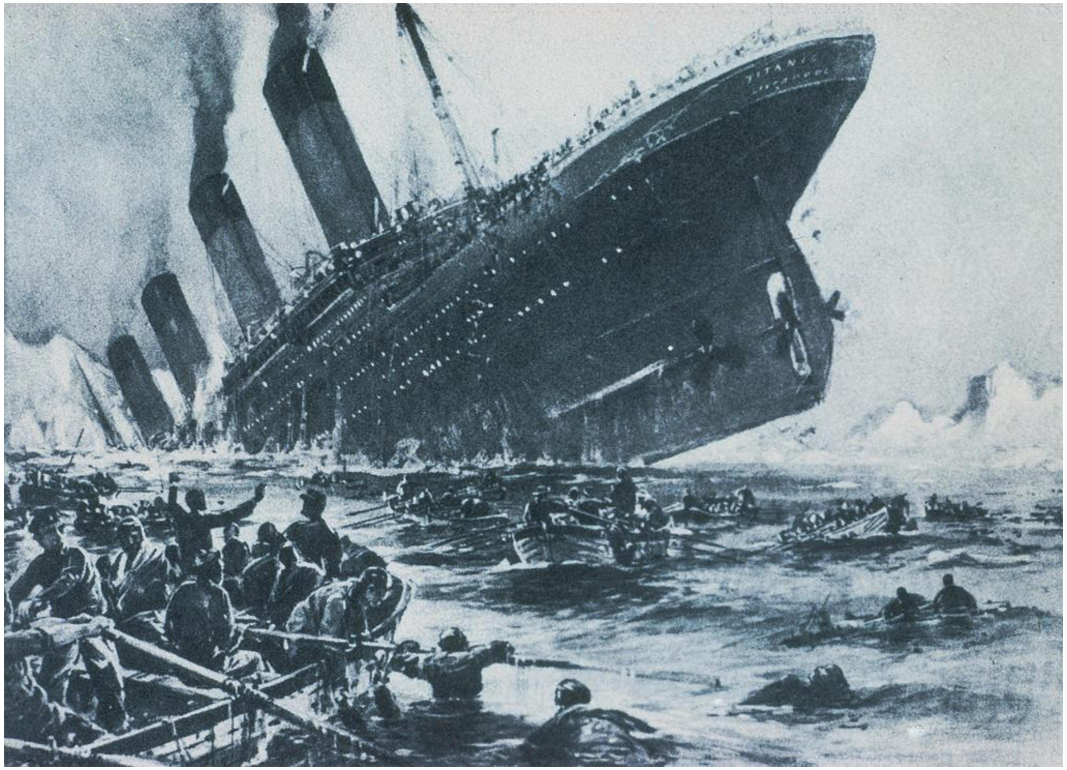
Compartmentalisation
aka
(application-level) sandboxing

Examples





Titanic



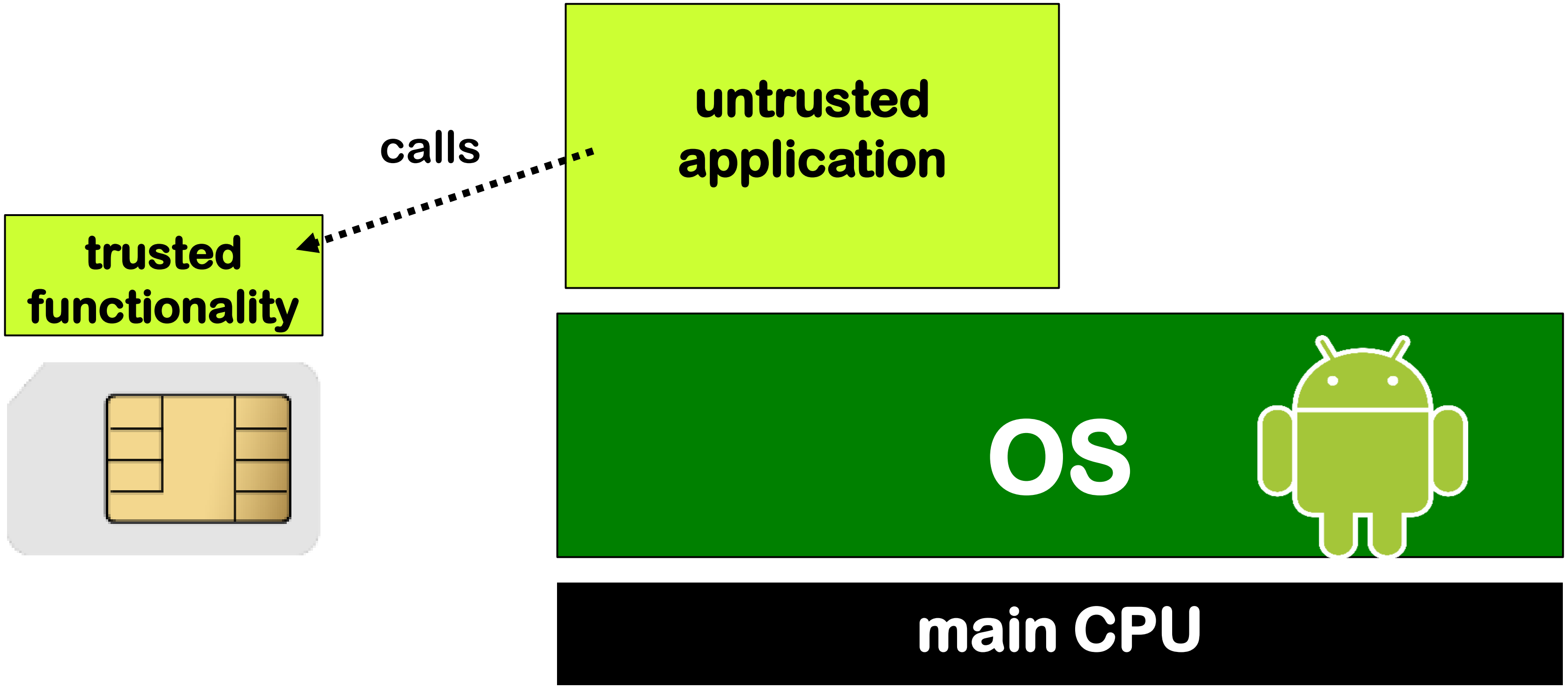
Does this mean compartmentalising is a bad idea?

No, but the attacker model was wrong.

- **Making vessel double-hulled would have been a better form of compartmentalising.**

Compartmentalisation example: SIM card in phone

A SIM provides some trusted functionality (with a small TCB) to a larger untrusted application (with a larger TCB)



untrusted application

calls

trusted functionality

OS

main CPU

Compartmentalisation examples

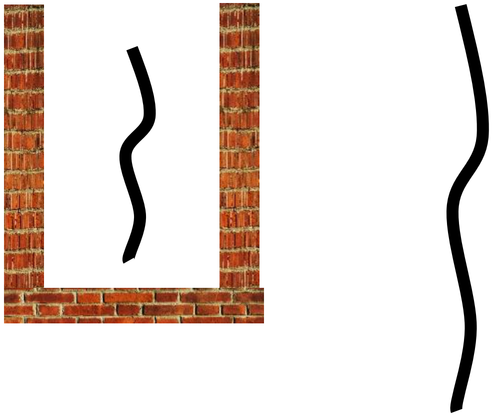
Compartmentalisation can be applied on many levels

- In an organisation
  - eg terrorist cells in Al Qaida or extreme animal rights group
- In an IT system
  - eg different machines for different tasks
- On a single computer, eg
  - different processes for different tasks
  - different user accounts for different task
  - use virtual machines to isolate tasks
  - partition your hard disk & install two OSs
- Inside a program / application / app / process
  - different ‘modules’ with different tasks

Focus of today

Isolation vs CIA (Confidentiality, Integrity & Availability)

Isolation is a very useful security property for programs and processes (i.e. program in execution)



'isolation' can be understood in CIA terms, as

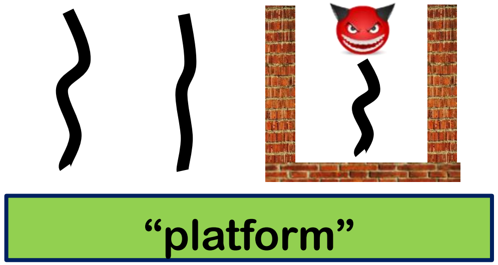
confidentiality and integrity of both data and code,

but conceptually less clear

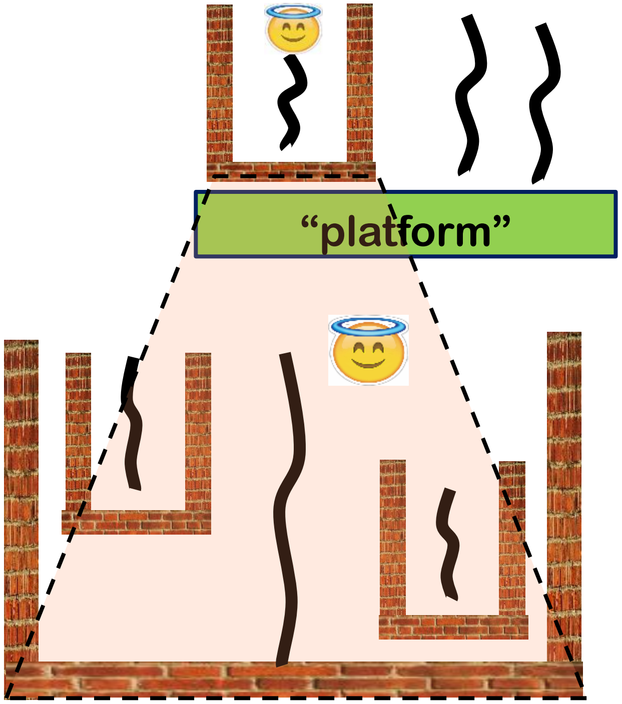
Two use cases for compartments

Compartmentalisation is good to isolate different trust levels

1. to contain a untrusted process from attacking others
    - aka sandboxing



2. to protect a trusted process from outside attacks



    - Here, it makes sense to apply it recursively

Compartmentalisation

Important questions to ask about any form of compartmentalisation

- What is the Trusted Computing Base (TCB) ?
  - Compartmentalising critical functionality inside a trusted process reduces the TCB for that functionality inside that process, but increases the TCB with the TCB of the enforcement mechanism
- Can the compartmentalisation be controlled by policies?
  - How expressive & complex are these policies?
  - Expressivity can be good, but resulting complexity can be bad…
- What are input & output channels?
  - We want exposed interfaces to be as simple, small, and just powerful enough
- Are there any hidden channels? Eg timing behaviour
  - These can be used deliberately, as covert channels, or exist by accident, as side channels

Access control

Some compartments offer access control that can be configured

It involves

1. Rights/permissions
2. Parties (eg. users, processes, components)
3. Policies that give rights to parties

   – specifying who is allowed to do what
4. Runtime monitoring to enforce policies,
   which becomes part of the TCB

Compartmentalisation for security design

1. Divide systems into **chunks** – aka compartments, components,…

   Different compartments for different tasks

2. Give **minimal access rights** to each compartment

   aka **principle of least privilege**

3. Have **strong encapsulation** between compartments

   so flaw in one compartment cannot corrupt others

4. Have **clear and simple interfaces** between compartments

   exposing minimal functionality

Benefits:

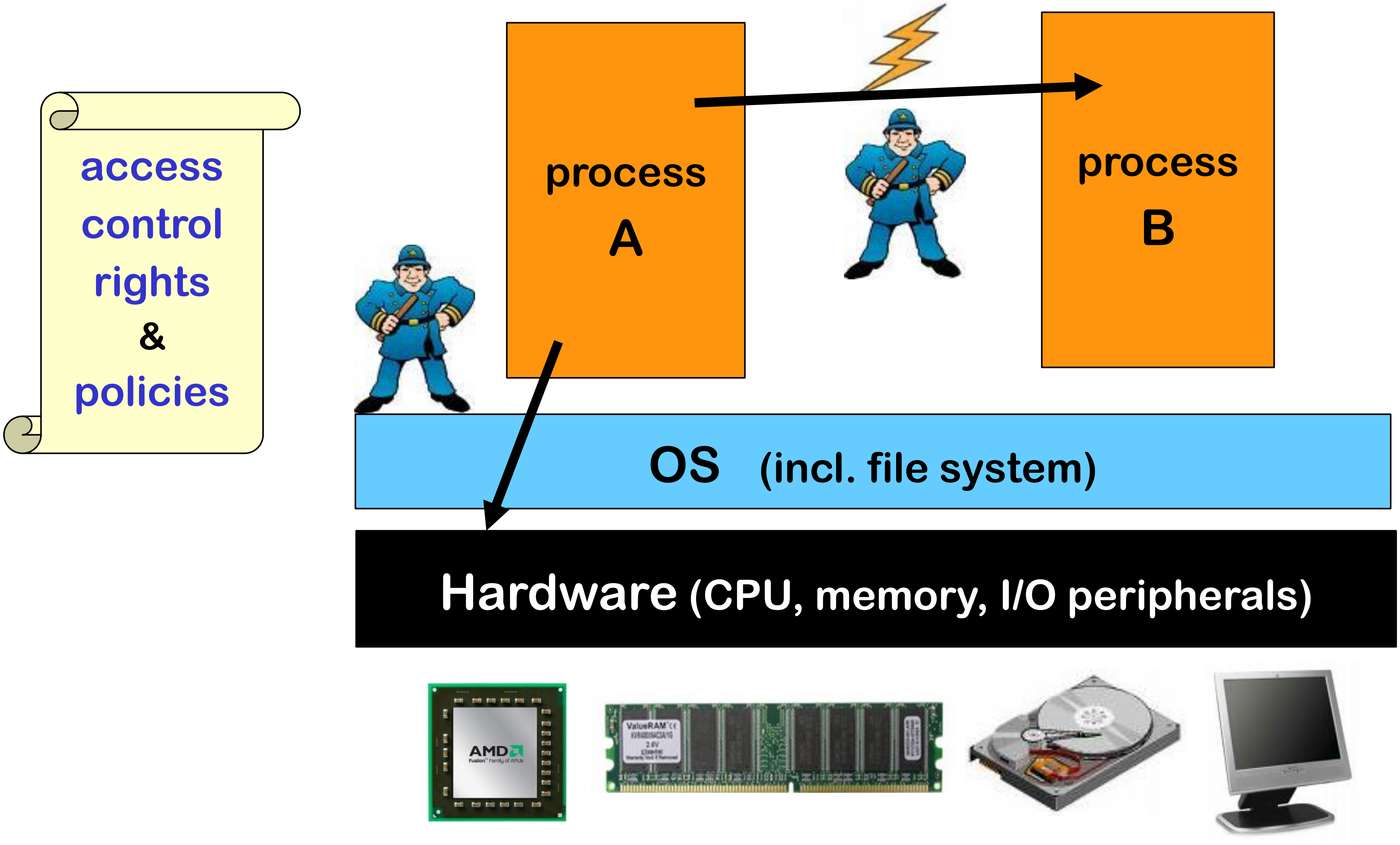
a. **Reduces TCB** for certain security-sensitive functionality

b. **Reduces the impact** of any security flaws.

Operating System (OS) Access Control

See also Chapter 2 of the lecture notes

Classical OS-based security (reminder)



access
control
rights
&
policies

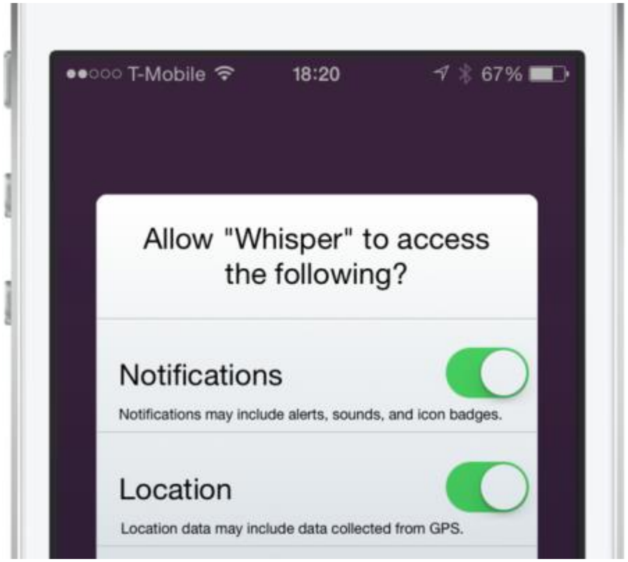
process
A

process
B

OS (incl. file system)

Hardware (CPU, memory, I/O peripherals)

Signs of OS access control



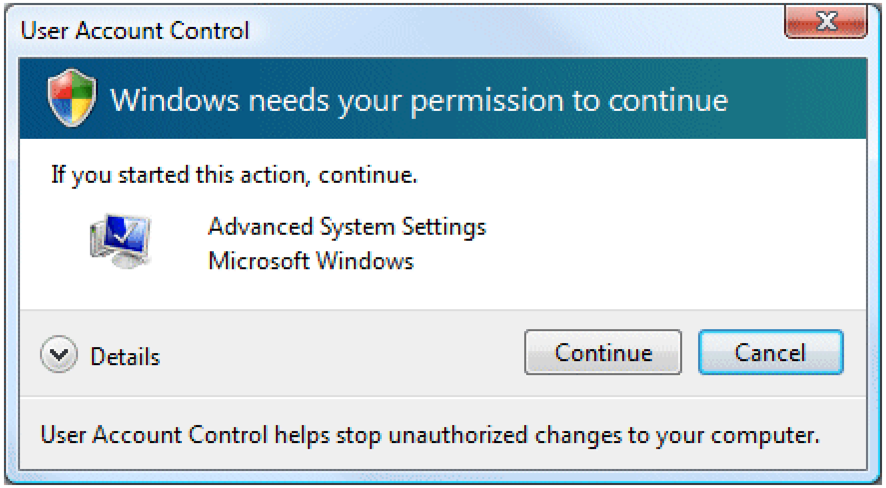
Allow "Whisper" to access the following?

Notifications

Notifications may include alerts, sounds, and icon badges.

Location

Location data may include data collected from GPS.

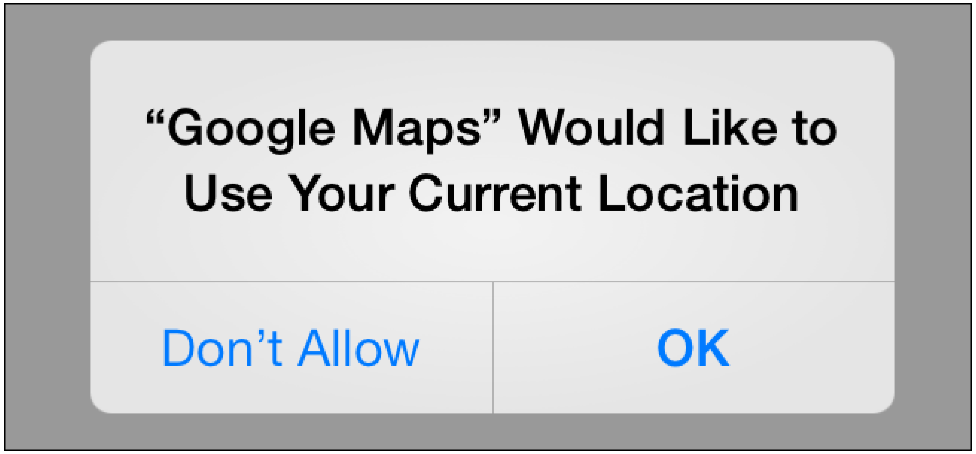

User Account Control

Windows needs your permission to continue

If you started this action, continue.

Advanced System Settings
Microsoft Windows

User Account Control helps stop unauthorized changes to your computer.


“Google Maps” Would Like to Use Your Current Location

Problems with OS access control

1. Size of the TCB
   The Trusted Computing Base for OS access control is **huge** so there *will* be security flaws in the code.

   The only safe assumption: a malicious user process on a typical OS (Linux, Windows, BSD, iOS, Android, ...) *will* be able to get root rights.

2. Too much complexity
   The languages to express access control policy are very complex, so people *will* make mistakes

3. Not enough expressivity / granularity
   Eg the OS cannot do access control *within* process, as processes as the ‘atomic’ units

Note: fundamental conflict between the need for expressivity

and the desire to keep things simple

Example: complexity (resulting in *privilege escalation)*

UNIX access control uses 3 permissions (`rwx`) for 3 categories of users (`owner,group,others`), for files & directories.

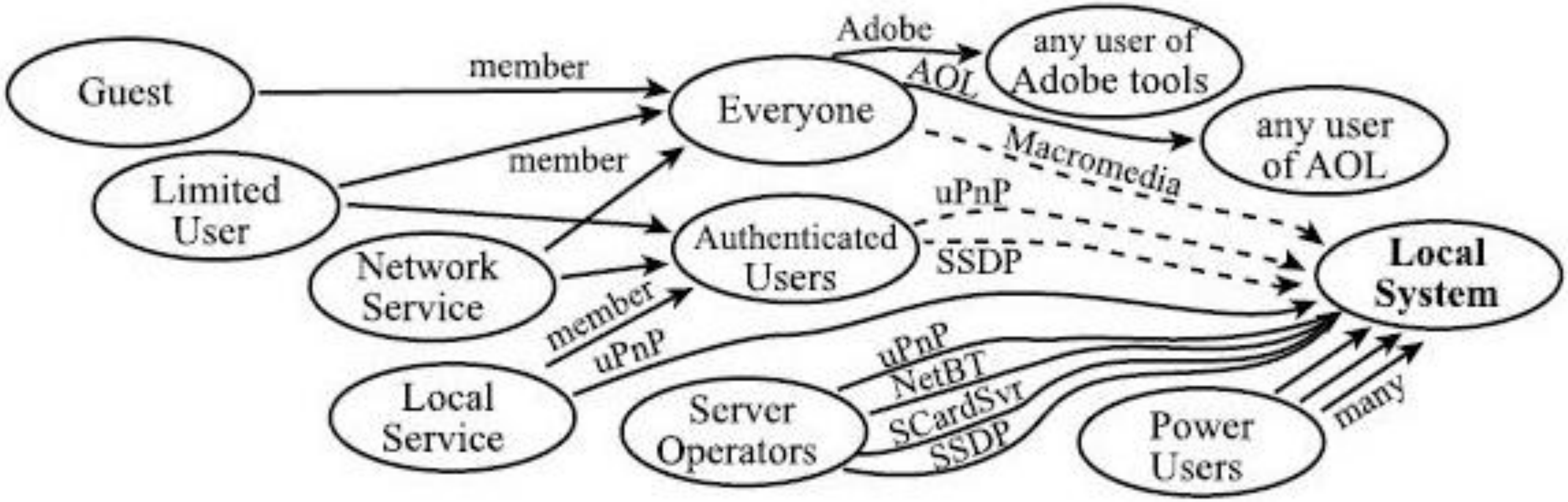
Windows XP uses 30 permissions, 9 categories of users, and 15 kinds of objects.

Example common configuration flaw in XP access control, in 4 steps:

1. Windows XP uses `Local Service` or `Local System` services for privileged functionality (where UNIX uses `setuid` binaries)
2. The permission `SERVICE_CHANGE_CONFIG` allows *changing the executable associated with a service* (say a printer driver)
3. But... it *also* allows to change *the account under which it runs*, incl. to `Local System`, which gives maximum root privileges.
4. Many configurations mistakenly grant `SERVICE_CHANGE_CONFIG` to all `Authenticated Users`...

Privilege escalation in Windows XP

Unintended privilege escalation due to misconfigured access rights of standard software packages in Windows XP:



[S. Govindavajhala and A.W. Appel, Windows Access Control Demystified, 2006]

Moral of the story (1) : **KEEP IT SIMPLE**

Moral of the story (2) : **If it is not simple, check the details**

`chroot` jail

`chroot` - change root - is nice example of compartmentalisation (of file system) in UNIX/Linux. It is coarse but simple.

- restricts access of a process to a subset of file system, ie. changes the root of file system for that process
- Eg running an application you just downloaded with

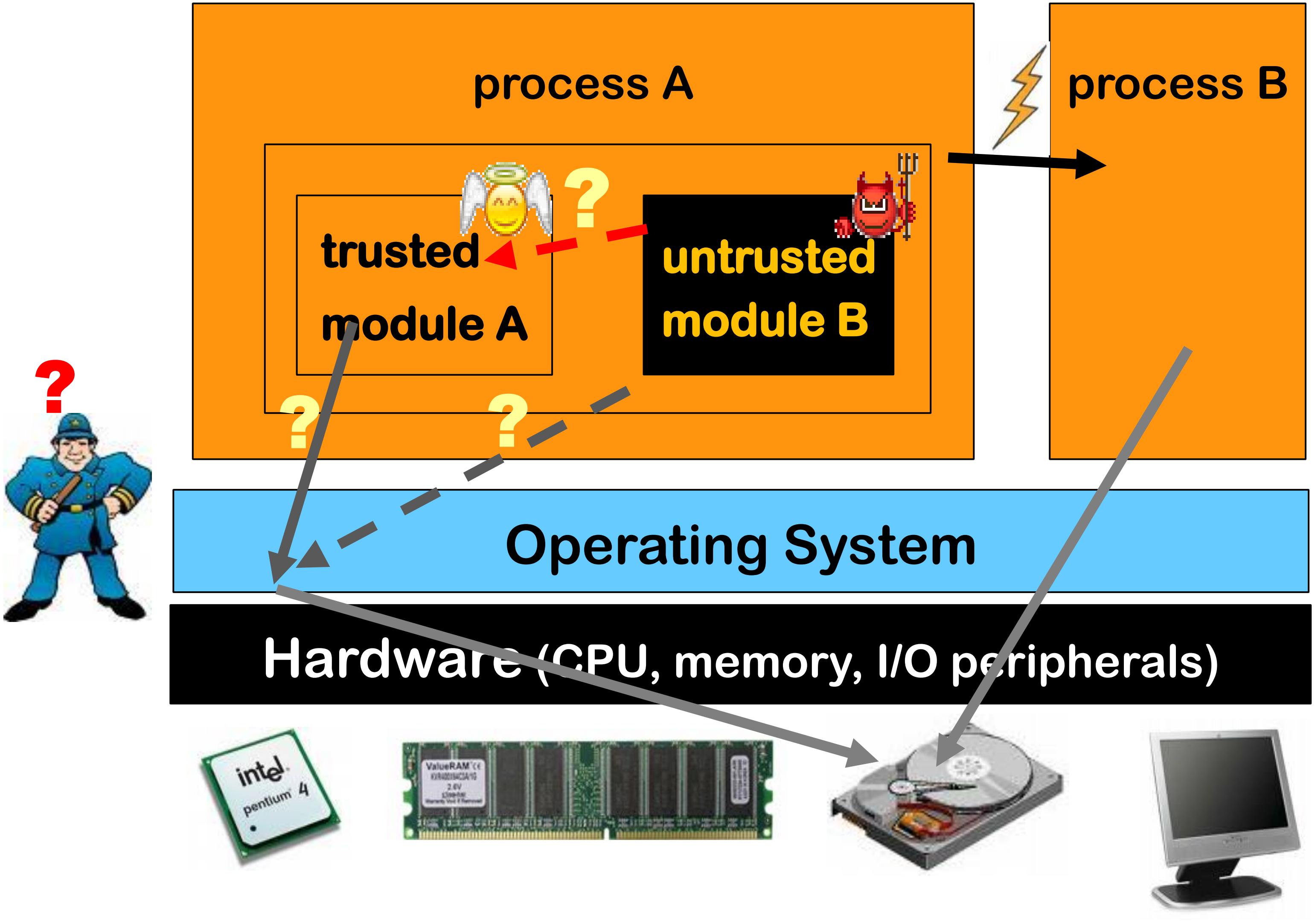
```
chroot /home/sos/erik/trial ; /tmp
```

restricts access to just these two directories

- Using traditional OS access control permissions for this would be very tricky! It would require getting permissions right all over the file system.

Limits in granularity

OS can't distinguish components *within* process, so can't differentiate access control for them, or do access control between them



Limitation of classic OS access control

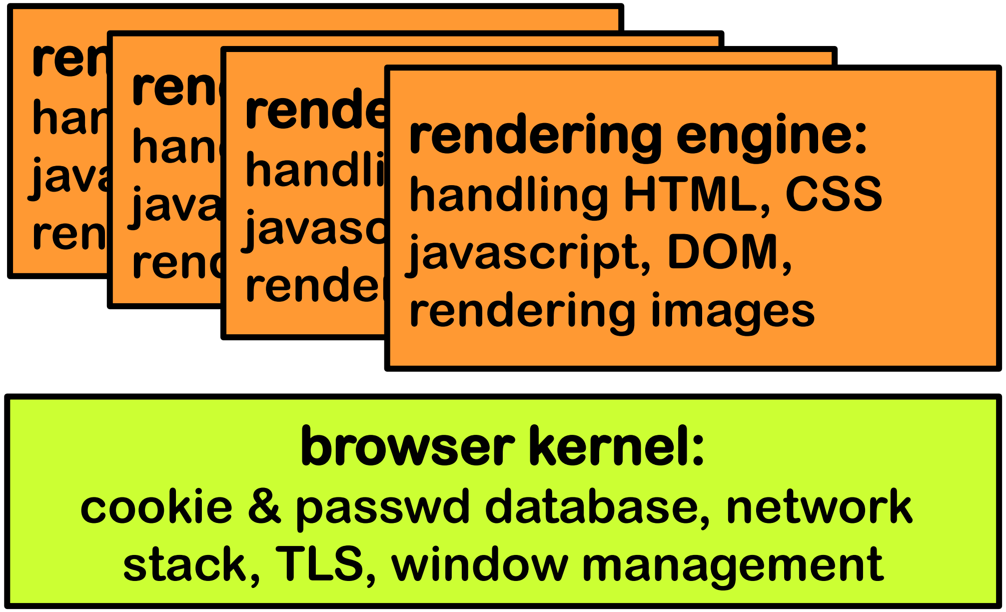
- A process has a fixed set of permissions. Usually, all permissions of the user who started it
- Execution with reduced permission set may be needed temporarily when executing untrusted or less trusted code. For this OS access control may be too coarse.

Remedies/improvements

  - Allowing users to drop rights when they start a process
  - Asking user approval for additional permissions at run-time
  - Using different user accounts for different applications, as Android does
  - Split a process into multiple processes with different access rights

Example: compartmentalisation in Chrome

Chrome browser process was split into multiple OS processes



rendering engine:
handling HTML, CSS
javascript, DOM,
rendering images

browser kernel:
cookie & passwd database, network stack, TLS, window management

One rendering engine per tab,
plus one for trusted content
(eg HTTPS certificate warnings)

No access to local file system and to each other

One browser kernel
with *full user privileges*

- (Complex!) rendering engine is black box for browser kernel
- Running a new process per domain can enforce the restrictions of the SOP (Same Origin Policy)
- *Advantage: TCB for certain operations drastically reduced*

More compartmentalisation in browsers

There are more forms of compartmentalisation and sandboxing inside browsers:

- SOP (Samen Origin Policy)
- CSP (Content Security Policy)
- sandboxing for iframes

Also, Microsoft Edge recently (2021) introduced Super Duper Secure Mode (SDSM) to remove some complexity, eg disabling JIT and to enable some additional memory protection mechanisms, eg CET (Control flow Enforcement Technology)

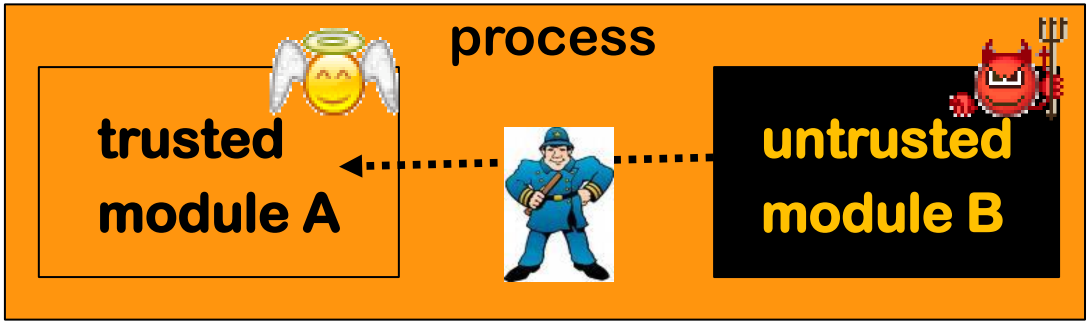
https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

Language-level access control

Chapter 4 of the lecture notes

Access control at the language level

In a safe programming language, access control can be provided *within* a process, at language-level, because interactions between components can be restricted & controlled



This makes it possible to have security guarantees in the presence of untrusted code (which could be malicious or just buggy)

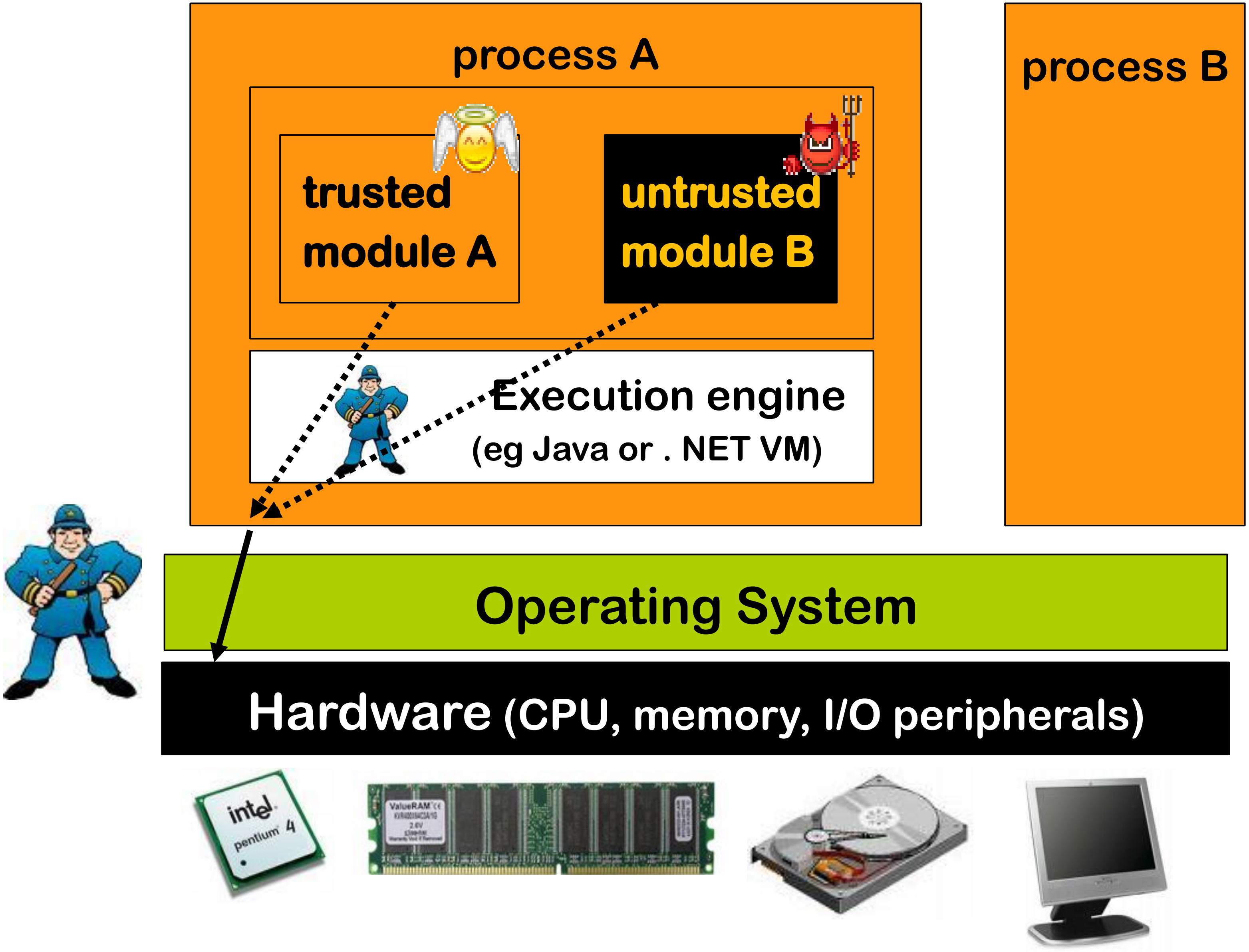
- *Without memory-safety, this is impossible. Why?*

  Because B can access any memory used by A

- *Without type-safety, it is hard. Why?*

  Because B can pass ill-typed arguments to A's interface

Language-level sandboxing is layer on top of OS sandboxing



process A

trusted module A

untrusted module B

Execution engine

(eg Java or . NET VM)

process B

Operating System

Hardware (CPU, memory, I/O peripherals)

Sand-boxing with code-based access control

Use cases

- using code from some untrusted or less trusted library
  - ie protection from supply chain attacks
- concentrating security-sensitive functionality is small module
  - smaller code base => smaller chance of bugs
  - put best programmers on this module
  - do more quality assurance for this module
    (more design reviews, more testing, more code reviews, ...)

Sand-boxing with code-based access control

Language platforms such as Java and .NET provide code-based access control

- this treats different parts of a program differently
- on top of the user-based access control of the OS

Ingredients for this access control, as for any form of access control

1. permissions
2. components (aka protection domains)
   - in traditional OS access control, this is the user ID
3. policies
   - which gives permissions to components, ie. *who* is allowed to do *what*

Code-based access control in Java

Example configuration file that expresses a policy

```
grant
 codebase "http://www.cs.ru.nl/ds", signedBy "Radboud",
 { permission
     java.io.FilePermission "/home/ds/erik","read";
 };

grant
 codebase "file:/.*"
 { permission
     java.io.FilePermission "/home/ds/erik","write";
 }
```

protection domains

Protection domains

- Protection domains based on evidence
  1. Where did it come from?
     - where on the local file system (hard disk) or where on the internet
  2. Was it digitally signed and if so by who?
     - using a standard PKI

- When loading a component, the Virtual Machine (VM) consults the security policy and remembers the permissions

Permissions

- Permissions represent a right to perform some actions. Examples:
  - `FilePermission(name, mode)`
  - `NetworkPermission`
  - `WindowPermission`
- Permissions have a set semantics, so one permission can be a superset of another one.
  - E.g. `FilePermission("*", "read")`
    includes `FilePermission("some_file.txt", "read")`
- Developers can define new custom permissions.

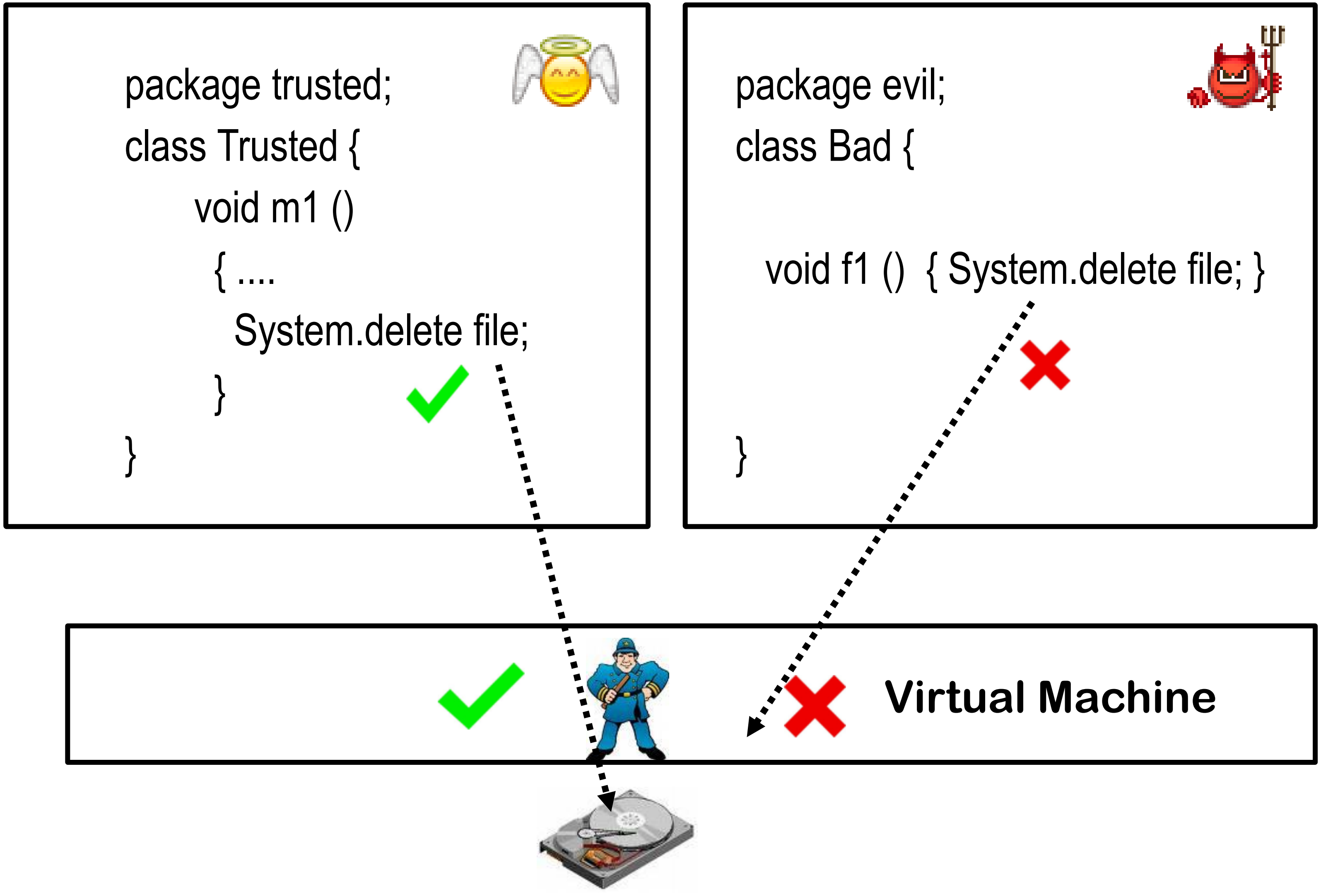
Last week: code-based access control in Java

Example configuration file that expresses a policy

```
grant
 codebase "http://www.cs.ru.nl/ds", signedBy "Radboud",
 { permission
    java.io.FilePermission "/home/ds/erik","read";
 };

grant
 codebase "file:/.*"
 { permission
    java.io.FilePermission "/home/ds/erik","write";
 }
```

protection domains



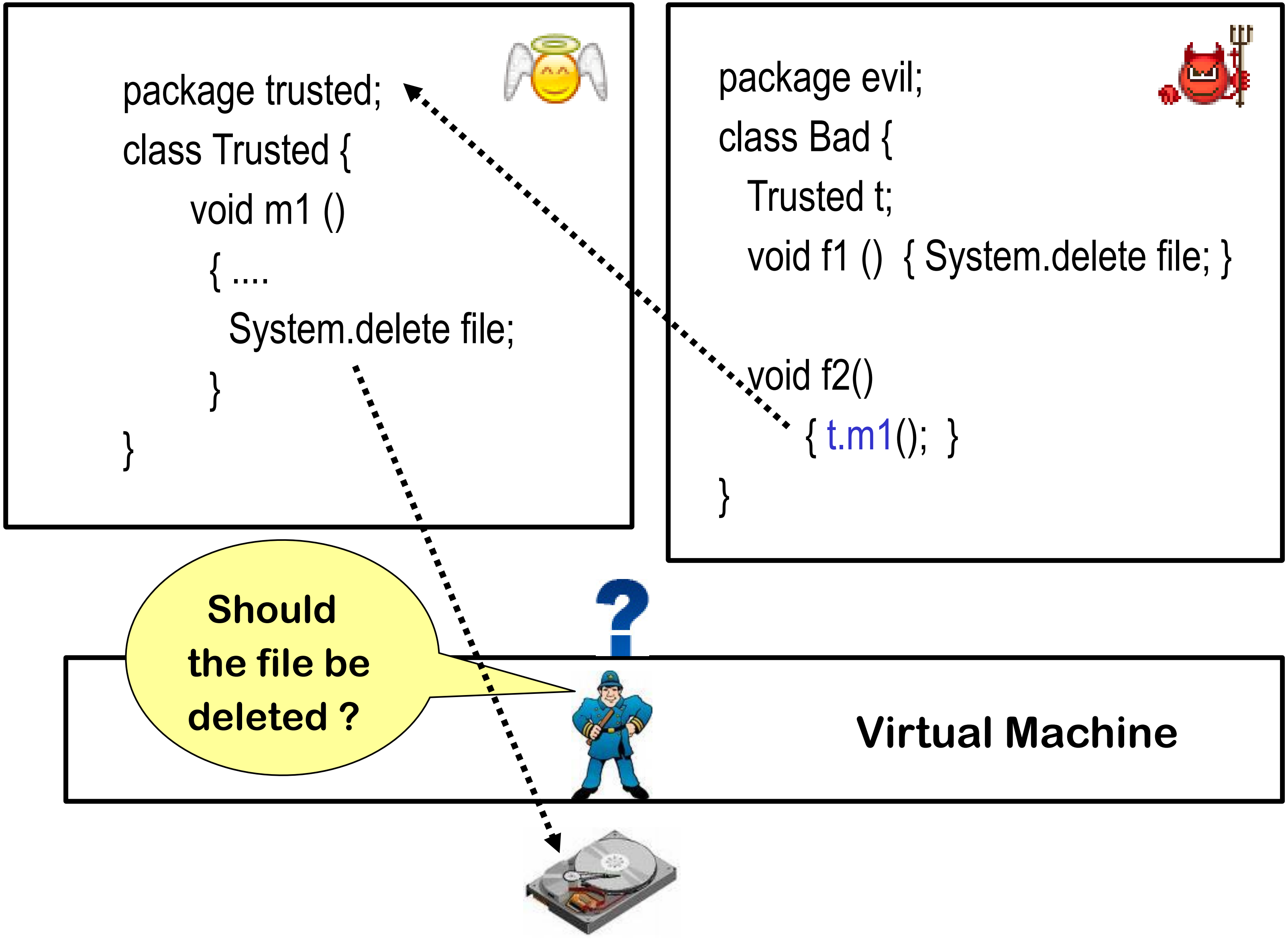
```
package trusted;
class Trusted {
      void m1 ()
       { ....
         System.delete file;
       }
}
```

```
package evil;
class Bad {

   void f1 ()  { System.delete file; }

}
```

Complication: methods calls



```
package trusted;
class Trusted {
      void m1 ()
       { ....
         System.delete file;
       }
}
```

```
package evil;
class Bad {
  Trusted t;
  void f1 ()  { System.delete file; }

  void f2()
      { t.m1();  }
}
```

Complication: method calls

There are different possibilities here

1. allow action if top frame on the stack has permission
2. only allow action if all frames on the stack have permission
3. ....

Pros? Cons?

1. is very dangerous: a class may accidentally expose dangerous functionality

2. is very restrictive: a class may want to, and need to, expose some dangerous functionality, but in a controlled way

More flexible solution: stackwalking aka stack inspection

Exposing dangerous functionality, (in)securely

```
Class Trusted{
  public void unsafeMethod(File f){
     delete f; } // Could be abused by evil caller
  public void safeMethod(File f) {
        .... // lots of checks on f;
     if all checks are passed, then delete f;}
          // Cannot be abused, assuming checks are bullet-proof
  public void anotherSafeMethod(){
     delete "/tmp/bla"; }
          // Cannot be abused, as filename is fixed.
          //  Assuming this file is not important..
}
```

Using visibility to control access?

```
Class Trusted{
  private void unsafeMethod(File f){
     delete f; } // Could be abused by ev
  public void safeMethod(File f) {
        .... // lots of checks on f;
     if all checks are passed, then delet
          // Cannot be abused, assuming checks are bullet-proof
  public void anotherSafeMethod(){
     delete "/tmp/bla"; }
          // Cannot be abused, as filename is fixed.
          //  Assuming this file is not important..
}
```

Making the unsafe method private & hence *invisible* to untrusted code helps, but is error-prone. Some public method may call this private method and indirectly expose access to it

Hence: stackwalking

Stack walking

- Every resource access or sensitive operation protected by a demandPermission(P) call for an appropriate permission P
  - no access without asking permission!
- The algorithm for granting permission is based on *stack inspection* aka *stack walking*

Stack inspection first implemented in Netscape 4.0,
then adopted by Internet Explorer, Java, .NET

Stack walking: basic concepts

Suppose thread T tries to access a resource

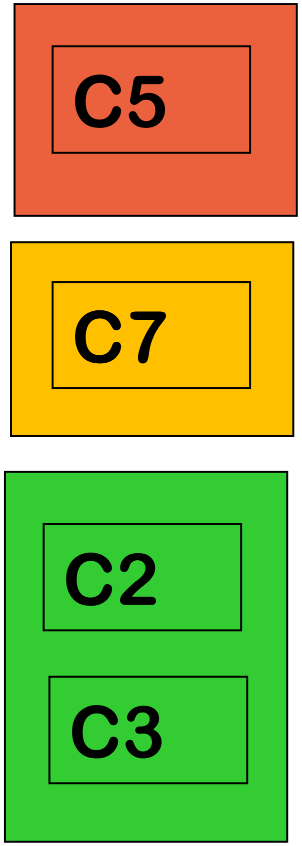
Basic algorithm:

access is allowed iff

__ALL__ components on the call stack have the right to access the resource

ie

- rights of a thread is the *intersection* of rights of all outstanding method calls



C5

C7

C2

C3

Stack for thread T:
C5 called by C7
called by C2 and C3

Stack walking

Basic algorithm is *too restrictive* in some cases

E.g.

- Allowing an untrusted component to delete some specific files
- Giving a partially trusted component the right to open specially marked windows (eg. security pop-ups) without giving it the right to open arbitrary windows
- Giving an app the right to phone certain phone numbers (eg. only domestic ones, or only ones in the mobile's phonebook)

Stack walk modifiers

- Enable_permission(P):
  - means: don't check my callers for this permission, I take full responsibility
  - This is essential to allow *controlled* access to resources for less trusted code

- Disable_permission(P):
  - means: don't grant me this permission, I don't need it
  - This allows applying the *principle of least privilege* (ie. only givie or ask the privileges *really* needed, and *only when* they are really needed)

Stack walking: algorithm

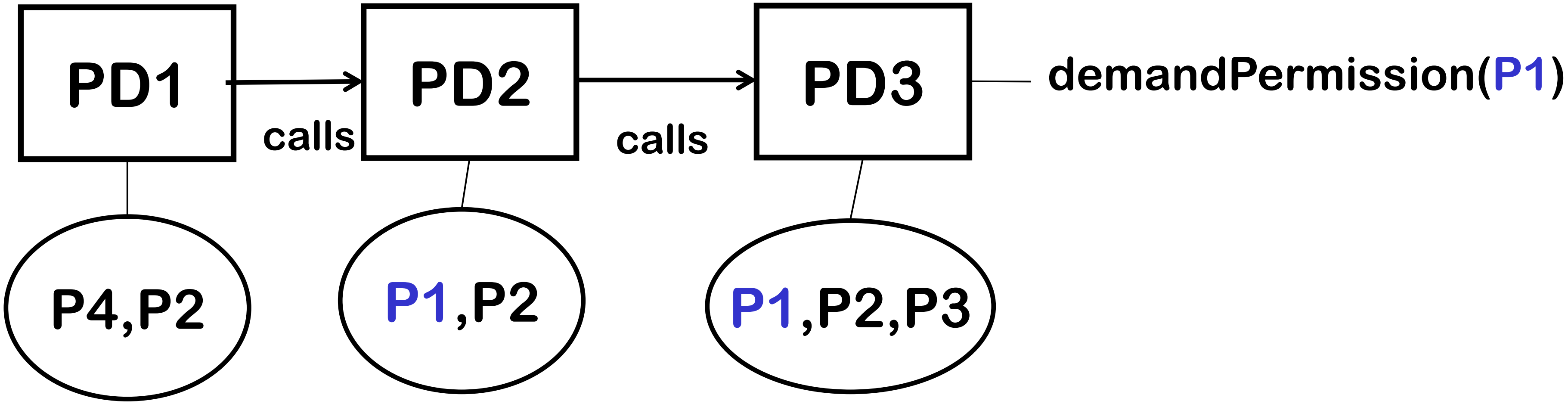
On creating new thread:

new thread inherit access control context of creating thread

DemandPermission(P) algorithm:

1. for each caller on the stack, from top to bottom:
   if the caller
   a) lacks Permission P: throw exception
   b) has disabled Permission P: throw exception
   c) has enabled Permission P: return
2. check inherited access control context

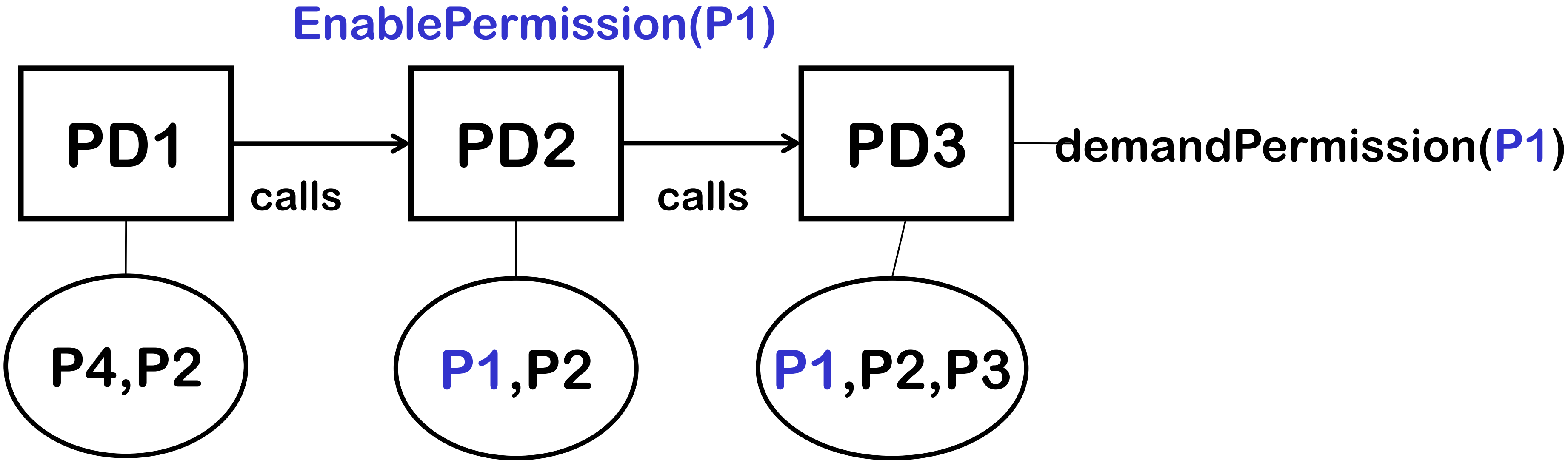
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

DemandPermission(P1) fails because PD1 does not have Permission P1

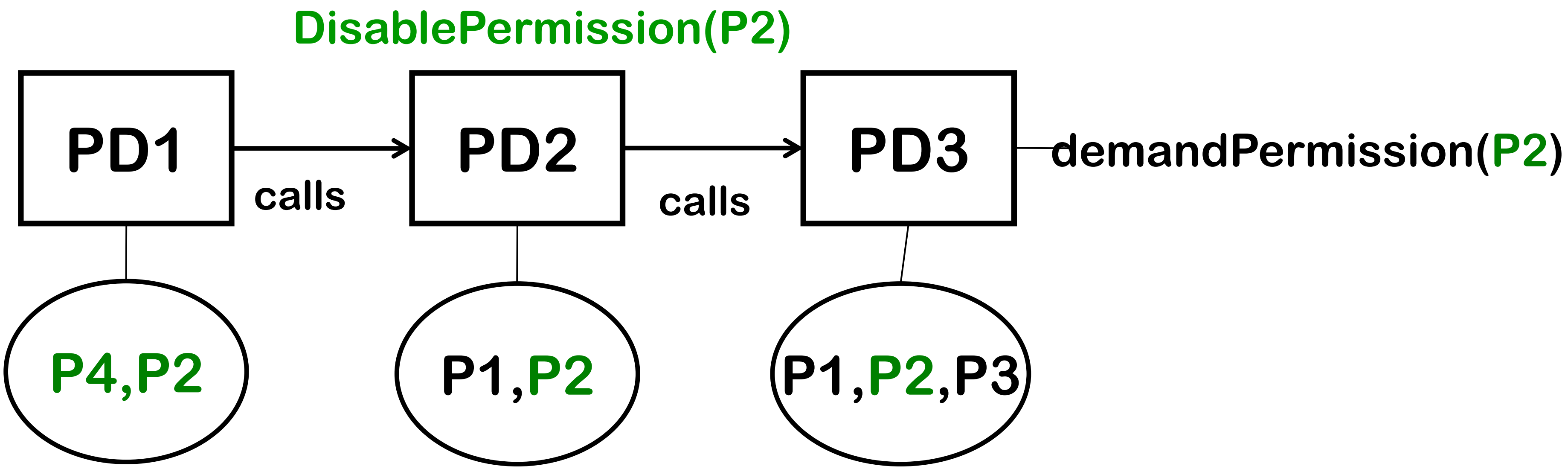
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

DemandPermission(P1) succeeds

Stack walk modifiers: examples



Will DemandPermission(P2) succeed ?

DemandPermission(P2) fails

Using stack walking to restrict access to functionality

```
Class Trusted{
   public void unsafeMethod(File f){
      delete f; }
   public void safeMethod(File f) {
      ... // lots of checks on f;
      enablePermission (FileDeletionPermission);
      delete f;}
   public void anotherSafeMethod(){
      enablePermission (FileDeletionPermission);
      delete “/tmp/bla”; }
}
```

“I take full responsibility for my callers”

Typical programming pattern

The typical programming pattern in privileged components,

esp. in public methods accessible by untrusted code:

```
public methodExposingScaryFunctionality (A a, B b){
    ....; do security checks on arguments a and b
    enable privileges (P1,P2);
    do the dangerous stuff that needs these privileges;
    disable privileges (P1,P2);
    .... }
```

in keeping with the principle of least privilege

Spot the security flaw?

```
Class Good{
  public void m1 (String filename) {
      lot of checks on filename;
      enablePermission (FileDeletionPermission);
      delete filename;}

   public void m2(byte[] filename){
      lot of checks on filename;
      enablePermission (FileDeletionPermission);
      delete filename;}
}
```

TOCTOU attack (Time of Check, Time of Use)

```
Class Good{

  public void m1 (String filename) {

       lot of checks on filename;

       enablePermission (FileDeletionPermission);

       delete filename;}


   public void m2(byte[] filename){

       lot of checks on filename;

       enablePermission (FileDeletionPermission);

       delete filename;}

}
```

m1 is secure, because Strings are immutable (assuming there are no TOCTOU vulnerabilities in the underlying file systems, eg due to symbolic links)

m2 is insecure, because byte arrays are mutable; attackers can could change the value of filename after the checks, in a multi-threaded setting

Need for privilege elevation

Note the similarity between

- Methods which enable some permissions
  - which temporarily raise privileges
- Linux `setuid root` programs or Windows Local System Services
  - which can be started by any user, but then run in admin mode
- OS system calls invoked from a user program
  - which cause a switch from user to kernel model

All are trusted services that elevate the privileges of their clients

- hopefully in a secure way...
- if not: privilege escalation attacks

In any code review, such code obviously requires extra attention!