Software Security

Application-level sandboxing

Erik Poll

Radboud Universiteit Nijmegen

TRU/e Master in Cyber Security

Sandboxing

Runtime access control aka sandboxing is one of the standard ways to provide security.

This involves rights and policies – to specify who is allowed to do what

1. conventional OS acccess control

access control
of applications and
between applications

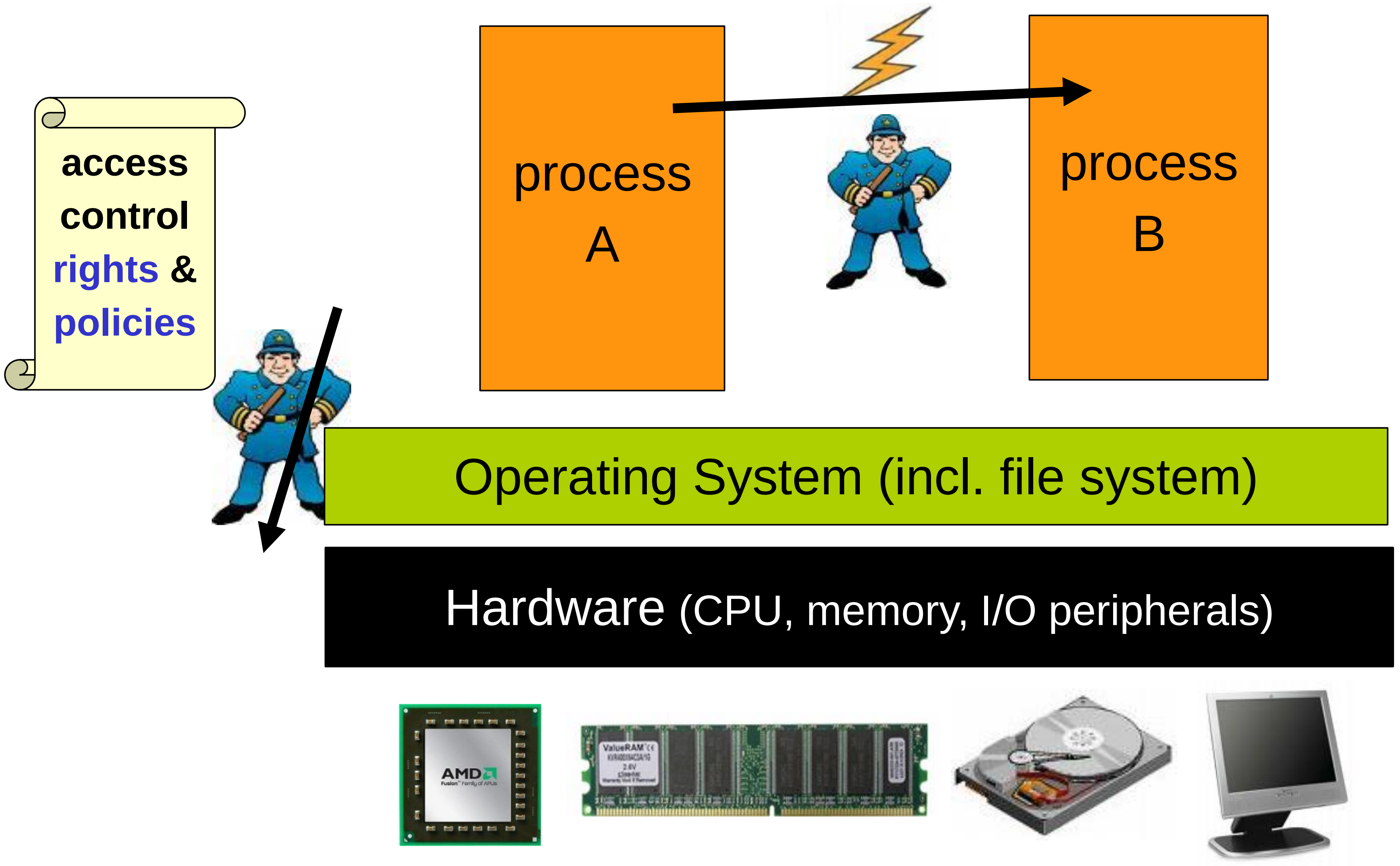
2. language-level sandboxing in safe languages
   - eg Java sandboxing using stackwalking
3. hardware-based sandboxing also for unsafe languages
   - eg safe enclaves using Intel SGX

access control
within an
application

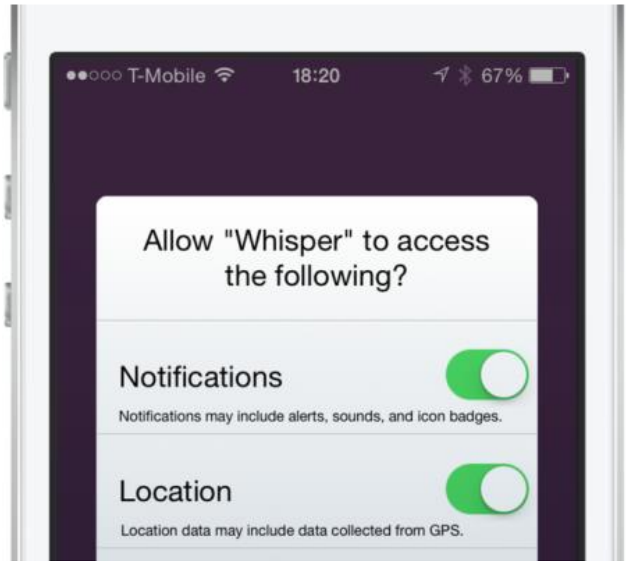
1. Operating System (OS) Access Control

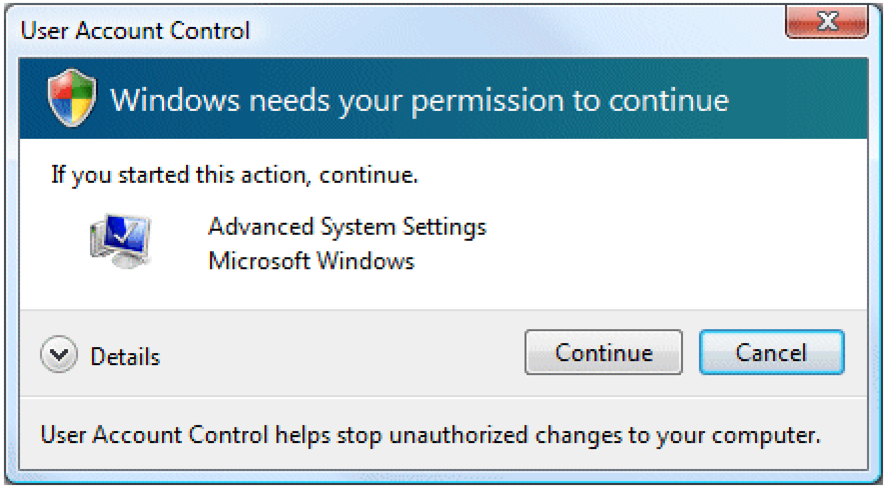
See also Chapter 2 of the lecture notes

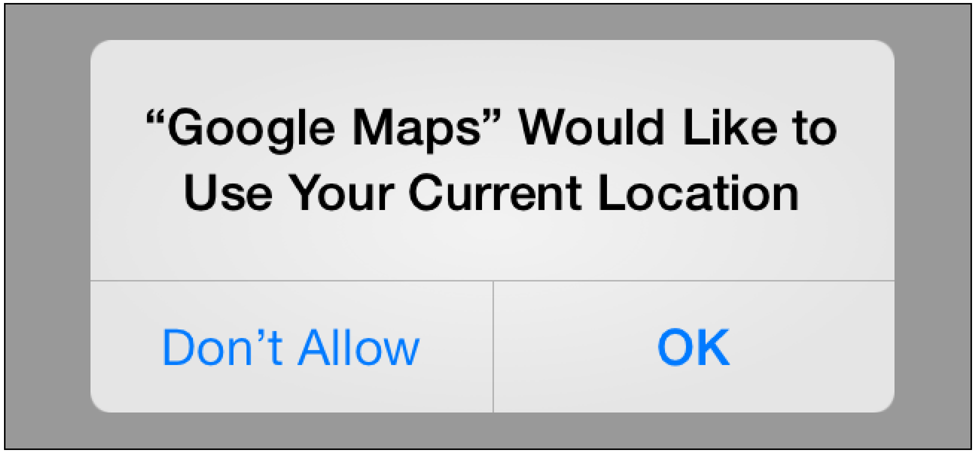
Classical OS-based security (reminder)



Signs of OS access control







Problems with OS access control

1. **Size of the TCB**

   The Trusted Computing Base for OS access control is **huge**
   so there *will* be software security flaws in the code.
   The only safe assumption: a malicious process on a typical OS (Linux, Windows, BSD...) *will* be able to get super-user/administrator rights.

2. **Complexity**

   The tools & languages for expressing access control are very complex, so people *will* make mistakes in access control policies and giving access control rights

3. **Expressivity / granularity**

   The OS cannot always provide the access control we want, because policies are not expressive enough or because OS access control is at the wrong 'level' to provide the level of granularity we want.

Note the fundamental conflict between the need for expressivity and the desire to keep things simple.

Complexity problem (resulting in *privilige escalation)*

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

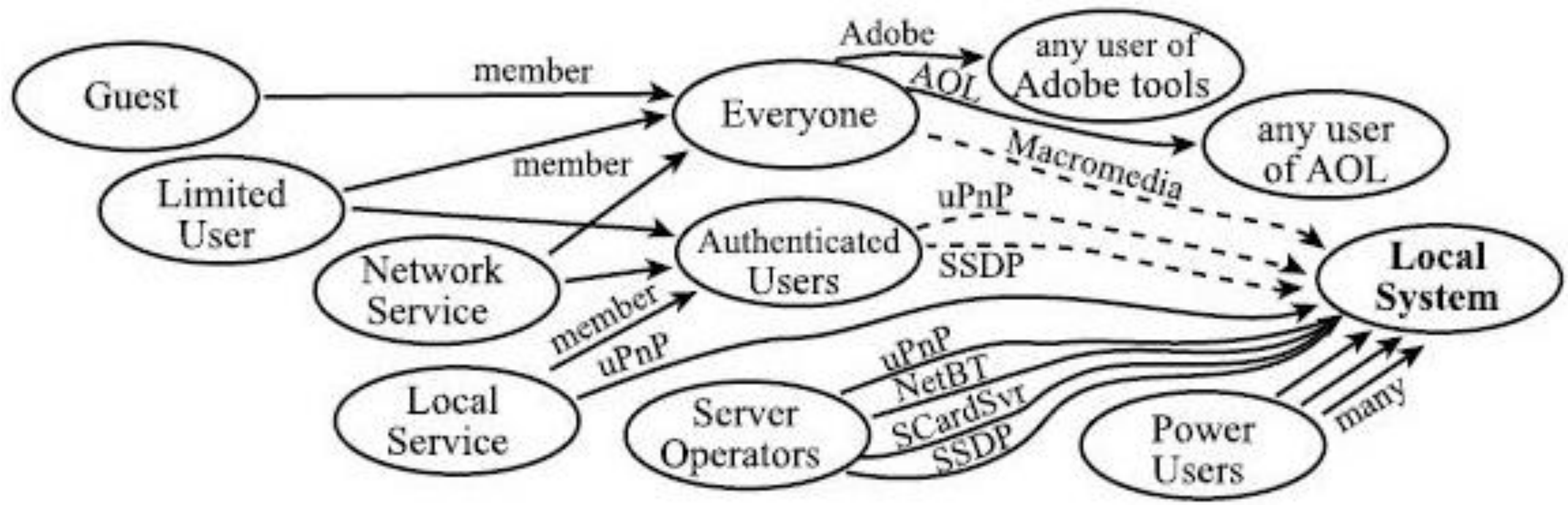
Windows XP uses 30 permissions, 9 categories of users, and 15 kinds of objects.

Example 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. Permission SERVICE_CHANGE_CONFIG allows *changing the executable* associated with a service
3. But... it *also* allows to change *the account under which it runs*, incl. to Local System, which gives maximum 'root' privileges.
4. Many services mistakenly grant SERVICE_CHANGE_CONFIG to all Authenticated Users...

Unintended privilige escalation in Windows XP

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



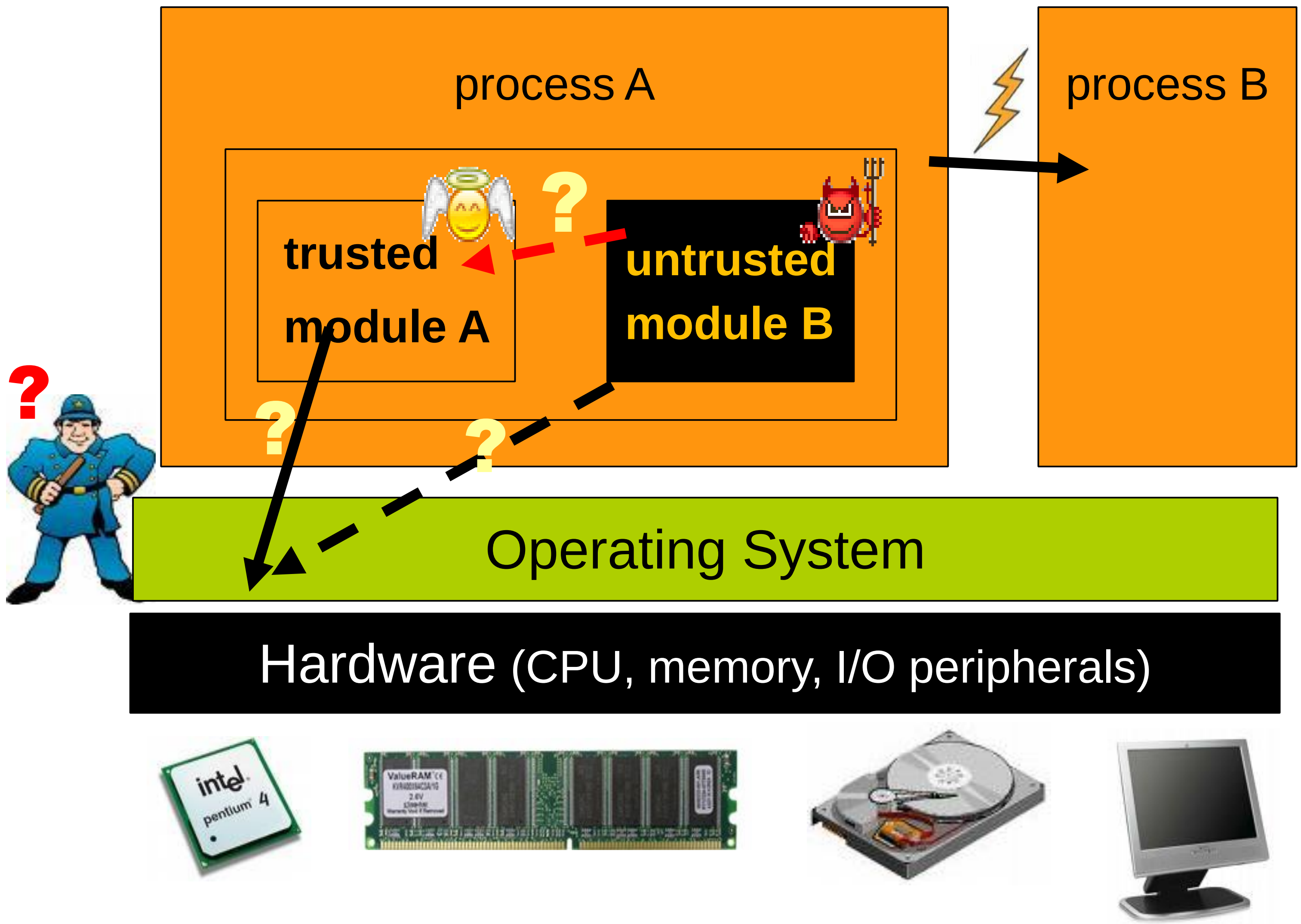
[S. Govindavajhala and A.W. Appel, Windows Access Control Demystified, 2006]

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

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

Limits in granularity

The OS cannot distinguish components *within* a process, so cannot differentiate access control for these, or 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
  - But OS can fine-tune this, eg demanding access for additional permissions at runtime
- Execution with reduced permission set may be needed temporarily when executing untrusted/less trusted code.
  For this OS access control is too coarse

One solution:

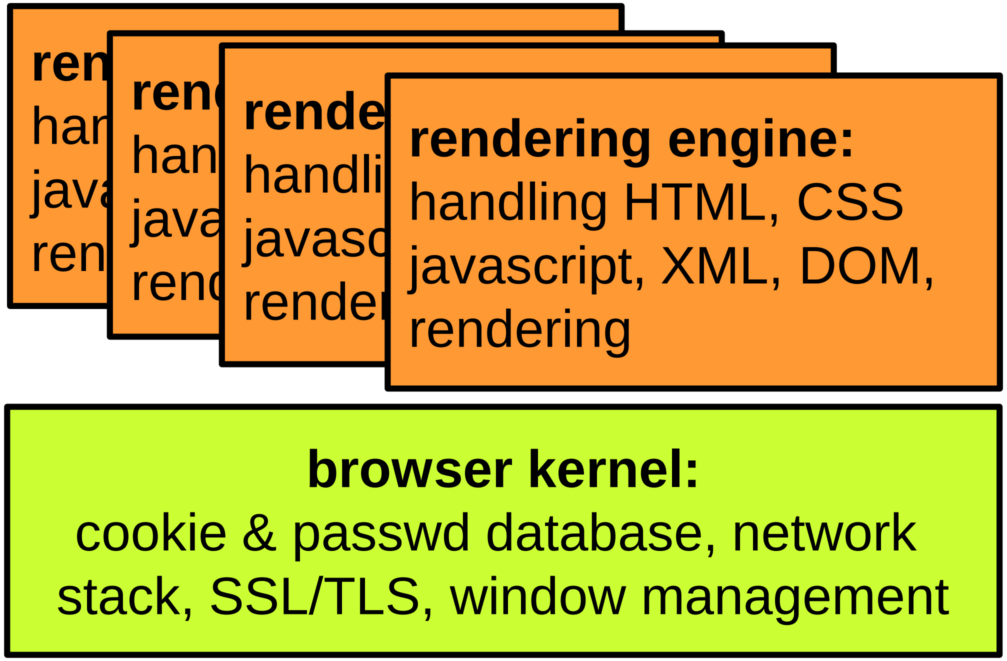
split a process into multiple processes,

with different access rights

Note: this can also reduce the size of the TCB, as some large & untrusted components can run with reduced rights

Example: compartementalisation in Chrome

The Chrome browser process is split into multiple OS processes



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
- plugins also run as different processes
- ***Advantage: size of the TCB drastically reduced***

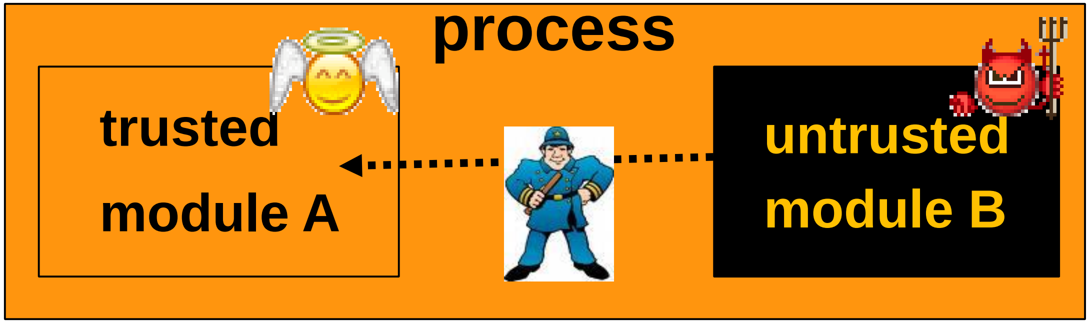
Other browsers now do the same thing

2. 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 simply buggy)

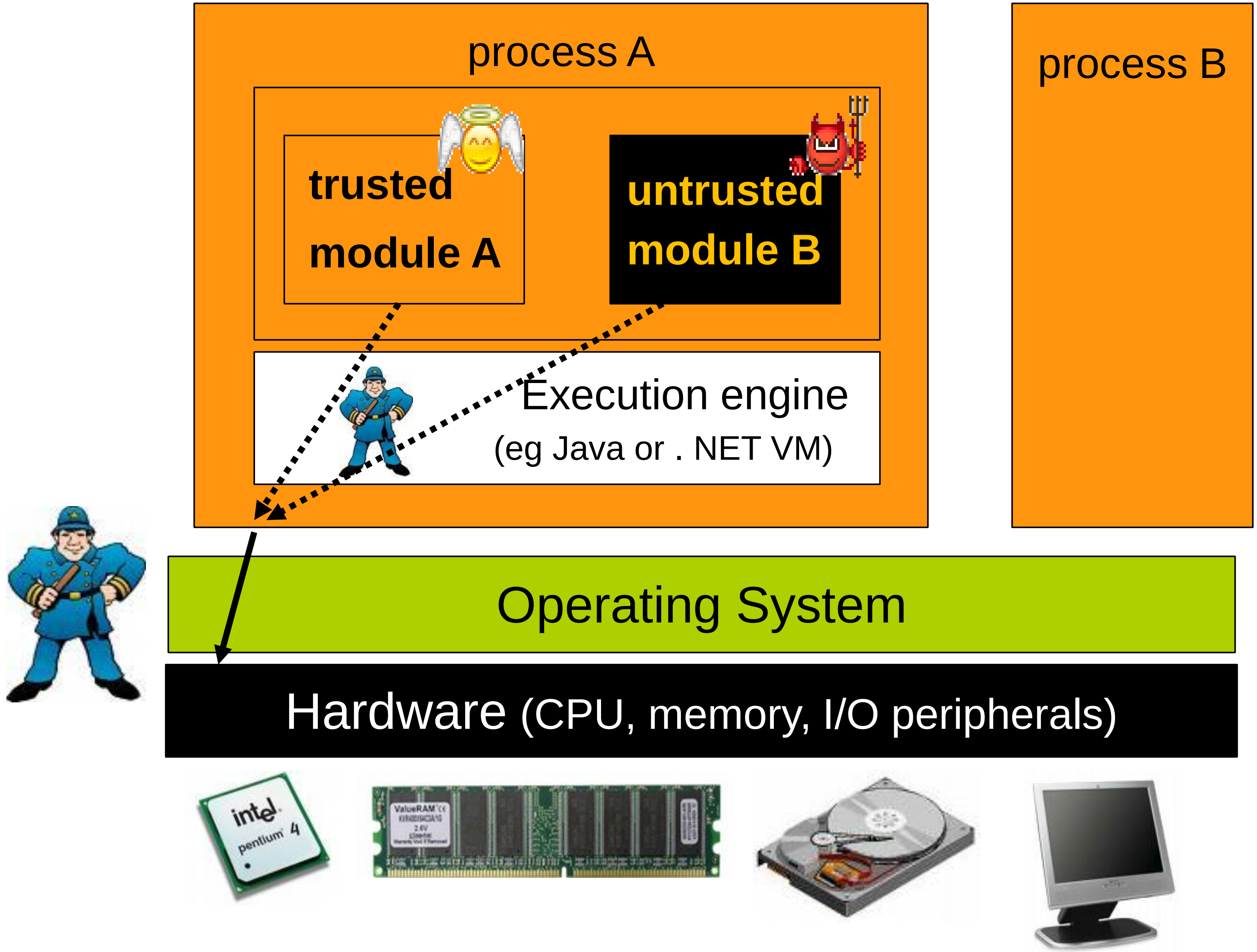
- *Without memory-safety, this is impossible. Why?*

  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

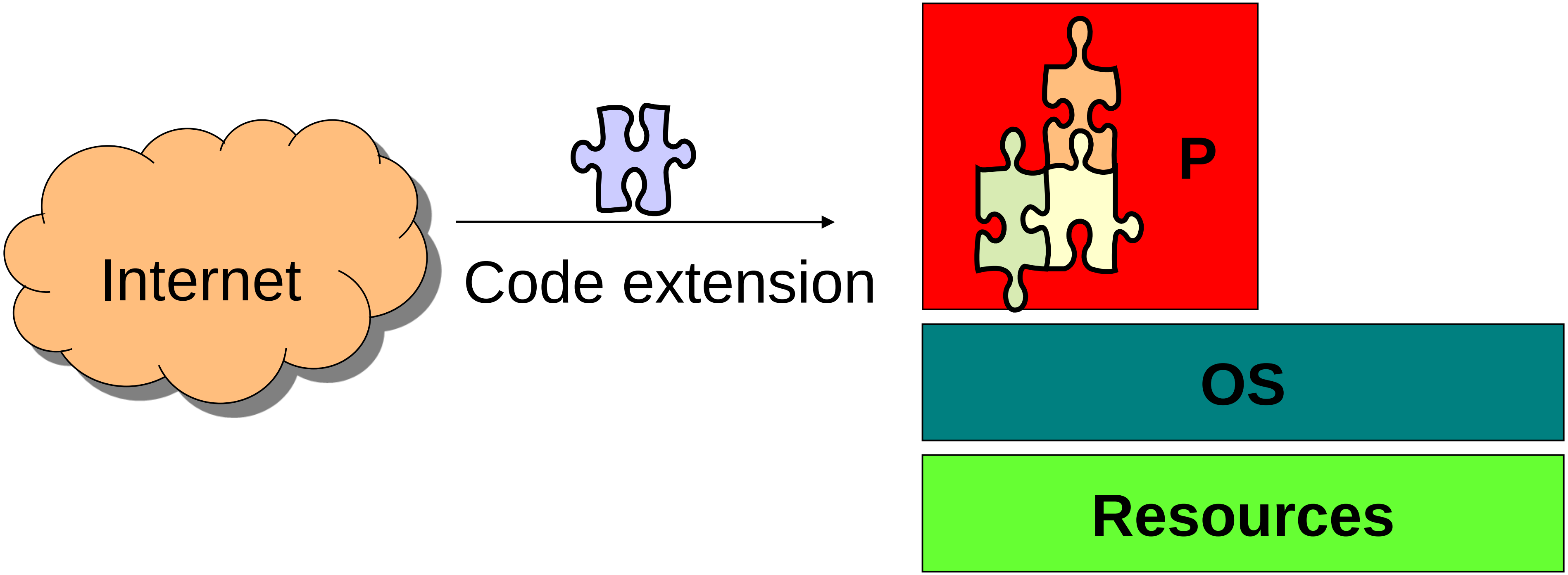
Language-level sandboxing



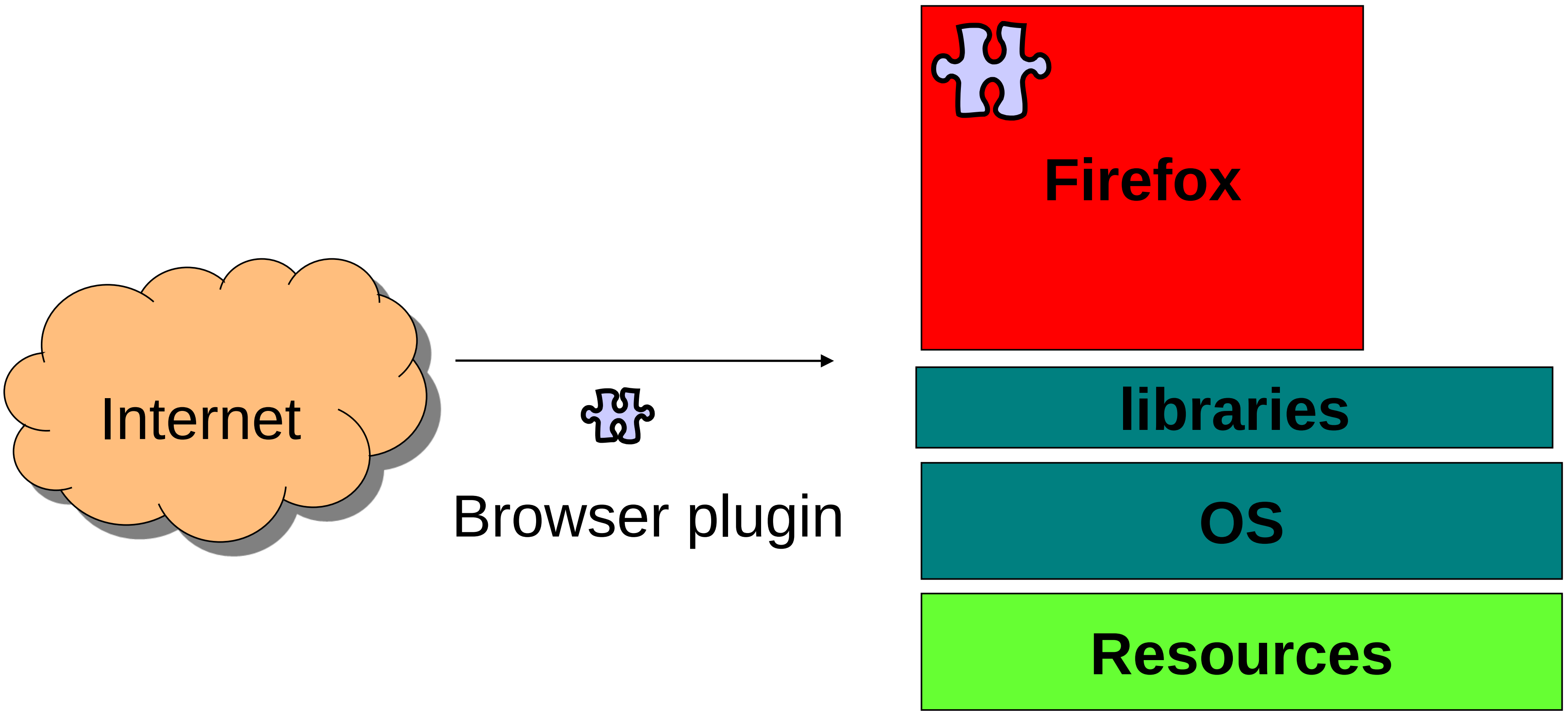
Extensible applications

Sandboxing individual parts of a program is useful if you trust some parts less than others

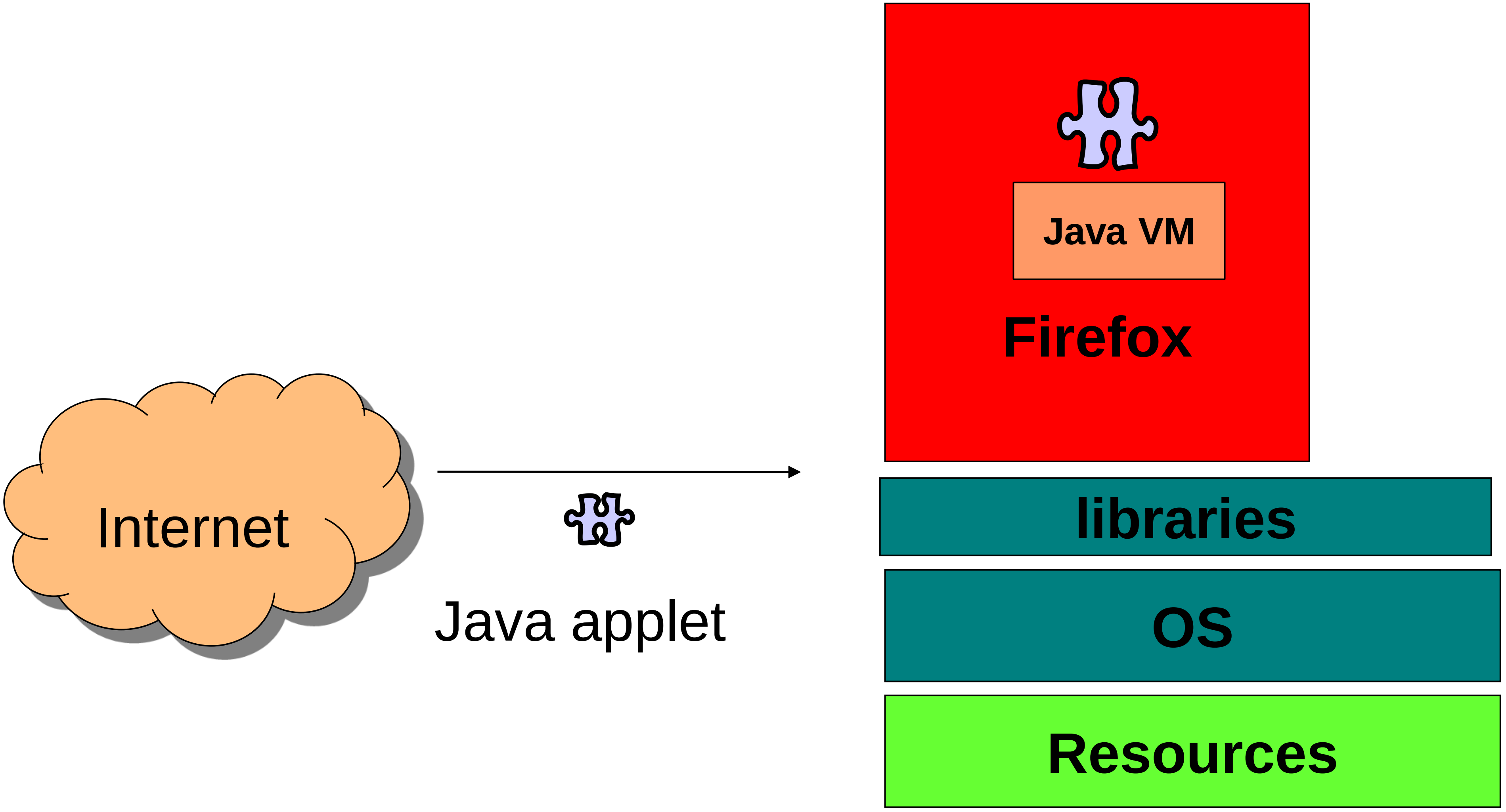
This is especially the case for extensible applications, where at runtime an application can extend itself



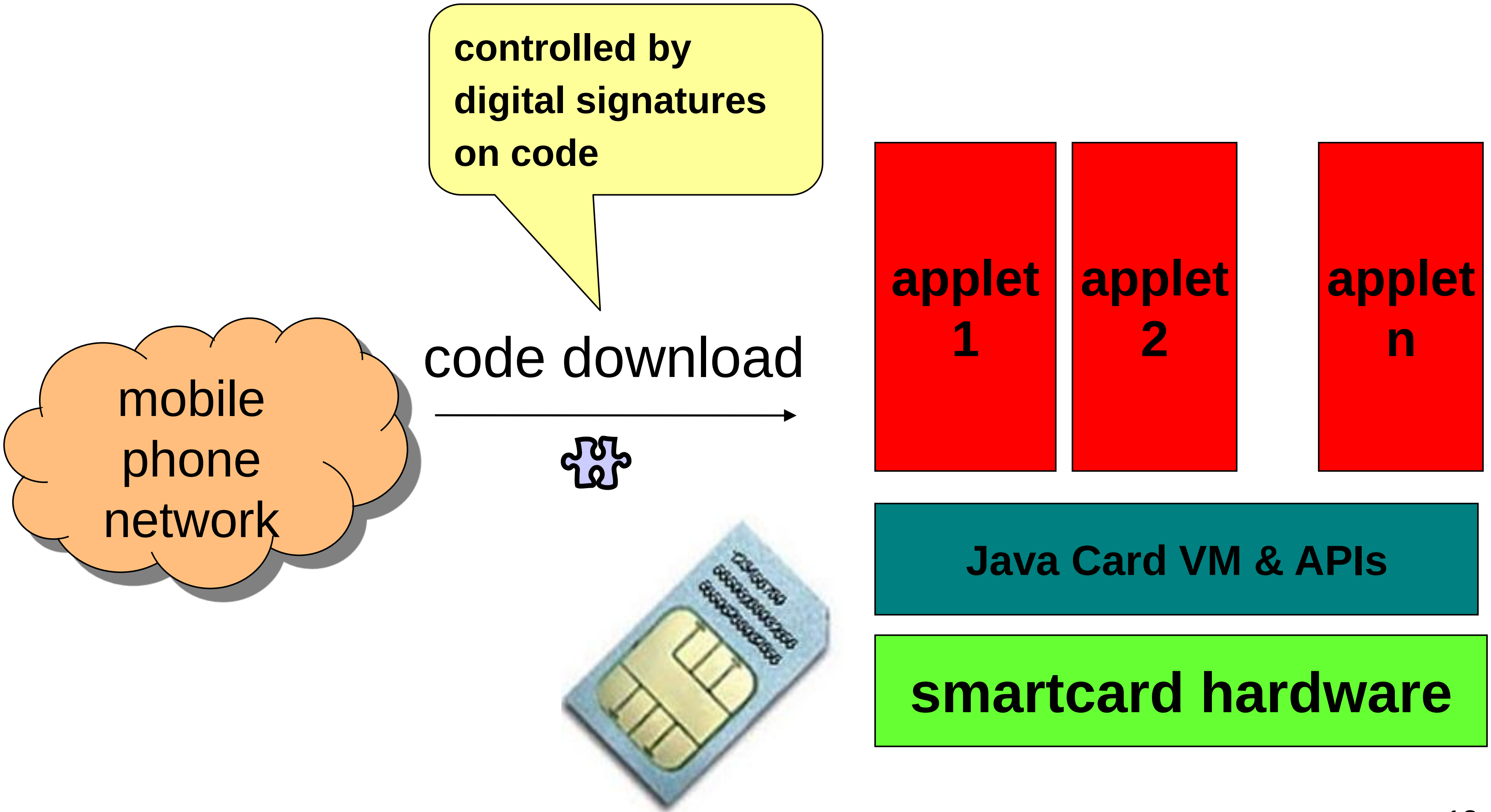
Example: browser plugin



Example: Java applet



Example: JavaCard smartcard



Sand-boxing with code-based access control

- Language platforms such as Java and .NET provide code-based access control which treats different parts of a program differently
  - on top of the user-based access control of the OS

- Ingredients for such access control, as usual
  1. permissions
  2. components or 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

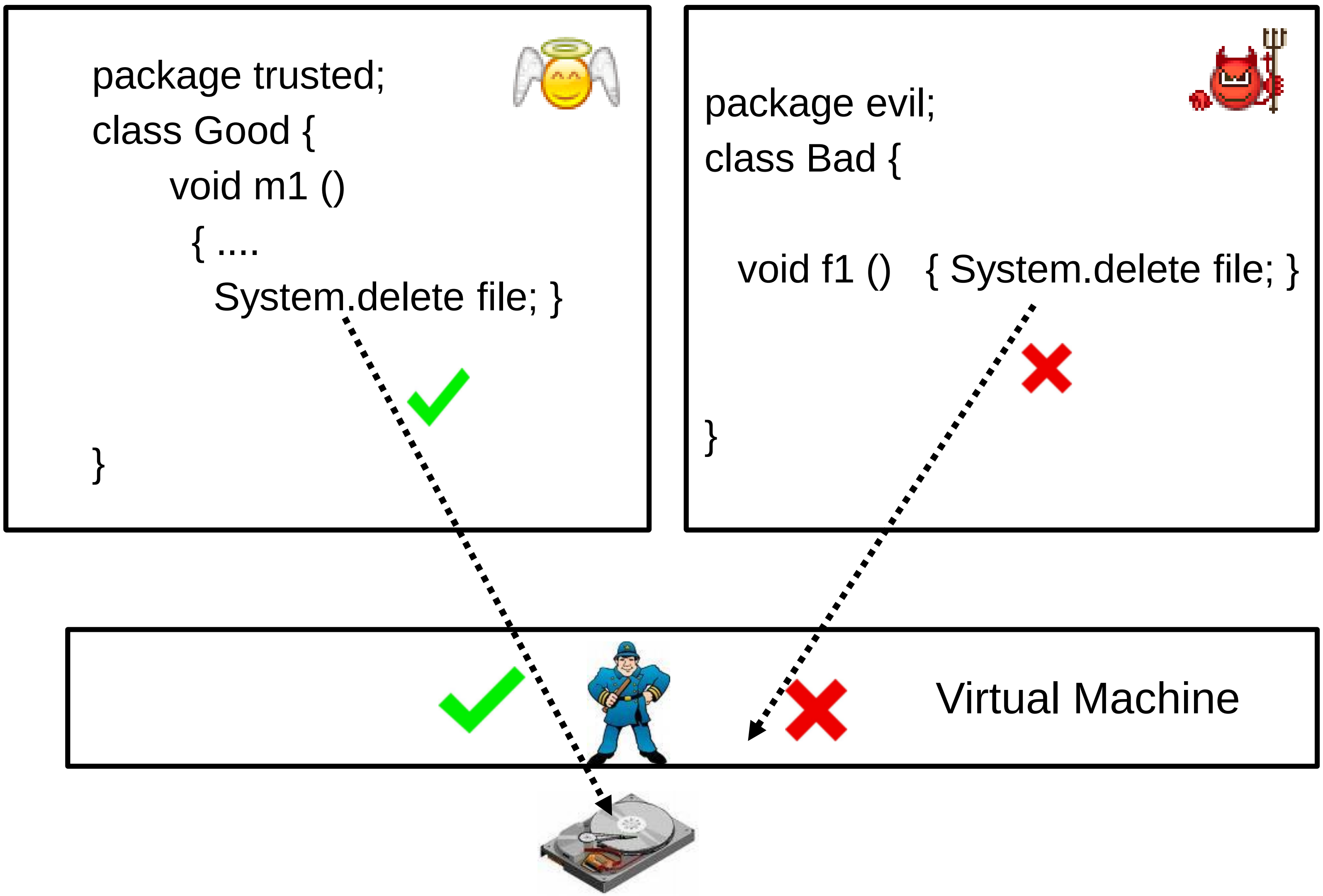
permissions

- 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.

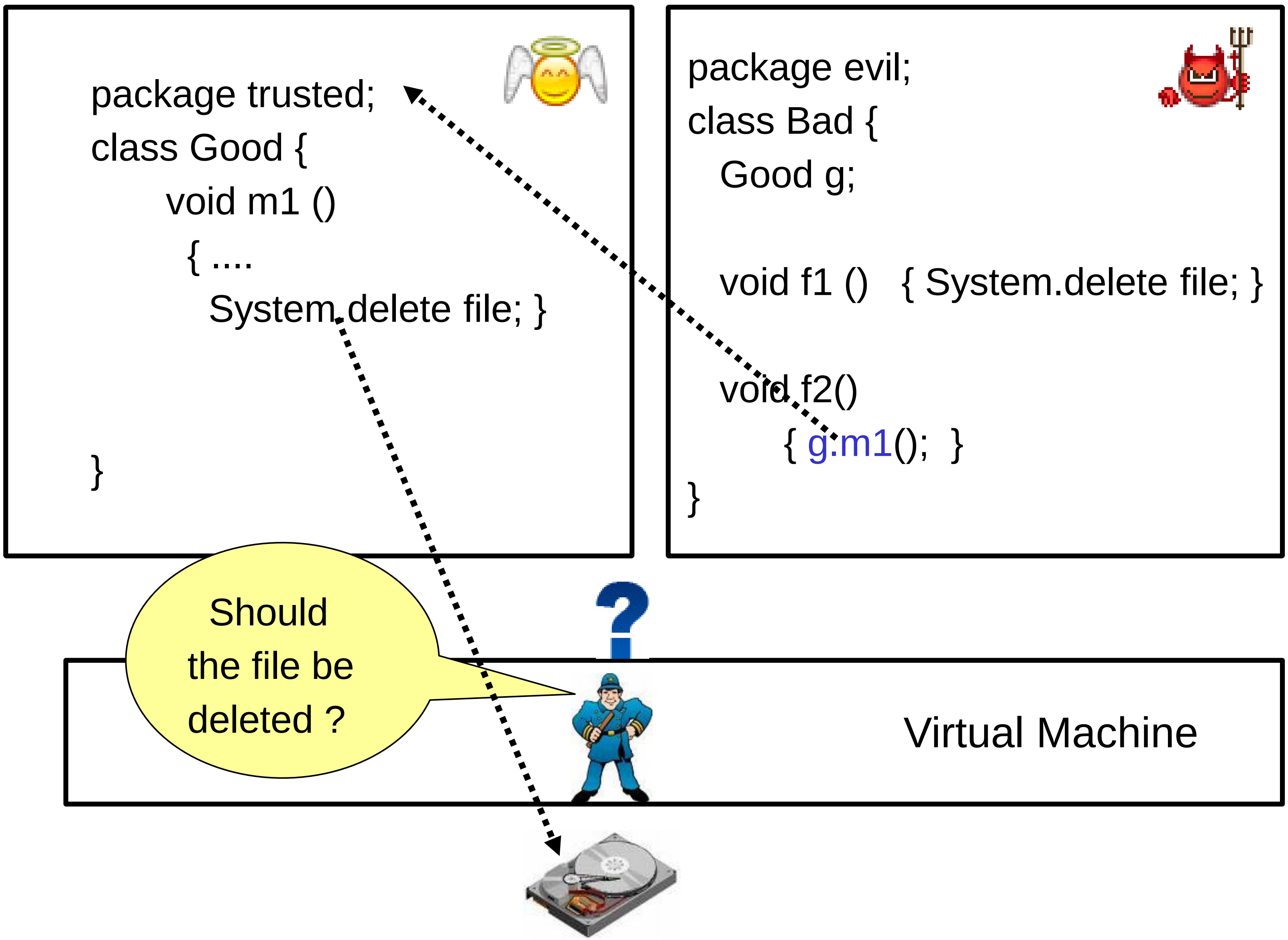
protection domains

- 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


package trusted;
class Good {
void m1 ()
{ ....
System.delete file; }
}
package evil;
class Bad {
void f1 () { System.delete file; }
}
Virtual Machine

Complication: methods calls



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 Good{
    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 restrict access to dangerous functionality?

```
Class Good{
    private 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
}
```

Making the unsafe method *invisible* to untrusted code helps, but is error-prone.. Some code in a big trusted package might indirectly expose access to this function. **Hence: stackwalking**

Stack walking

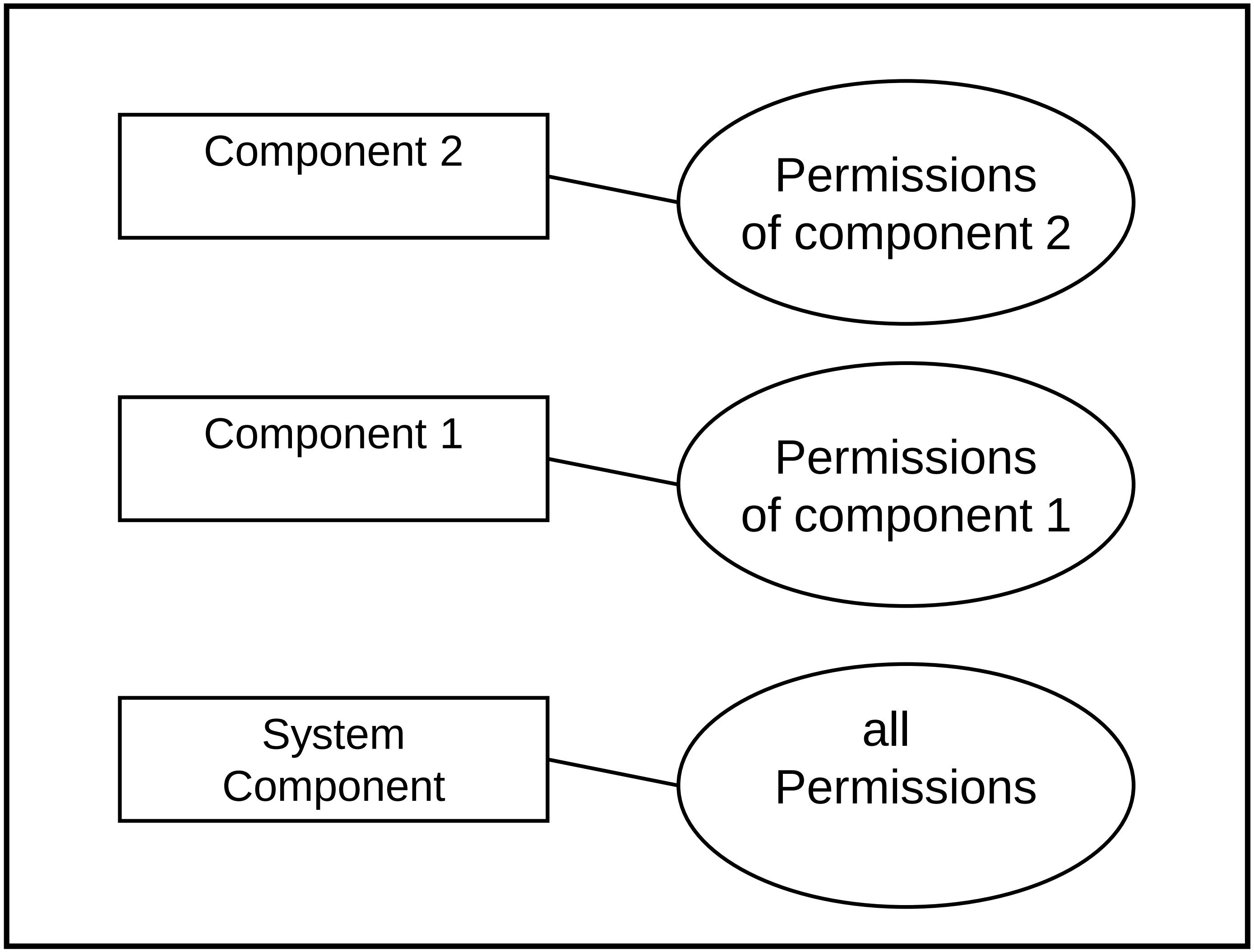
- Every resource access or sensitive operation protected by a demandPermission(P) call for an appropriate permission P
  - no access without asking permission!

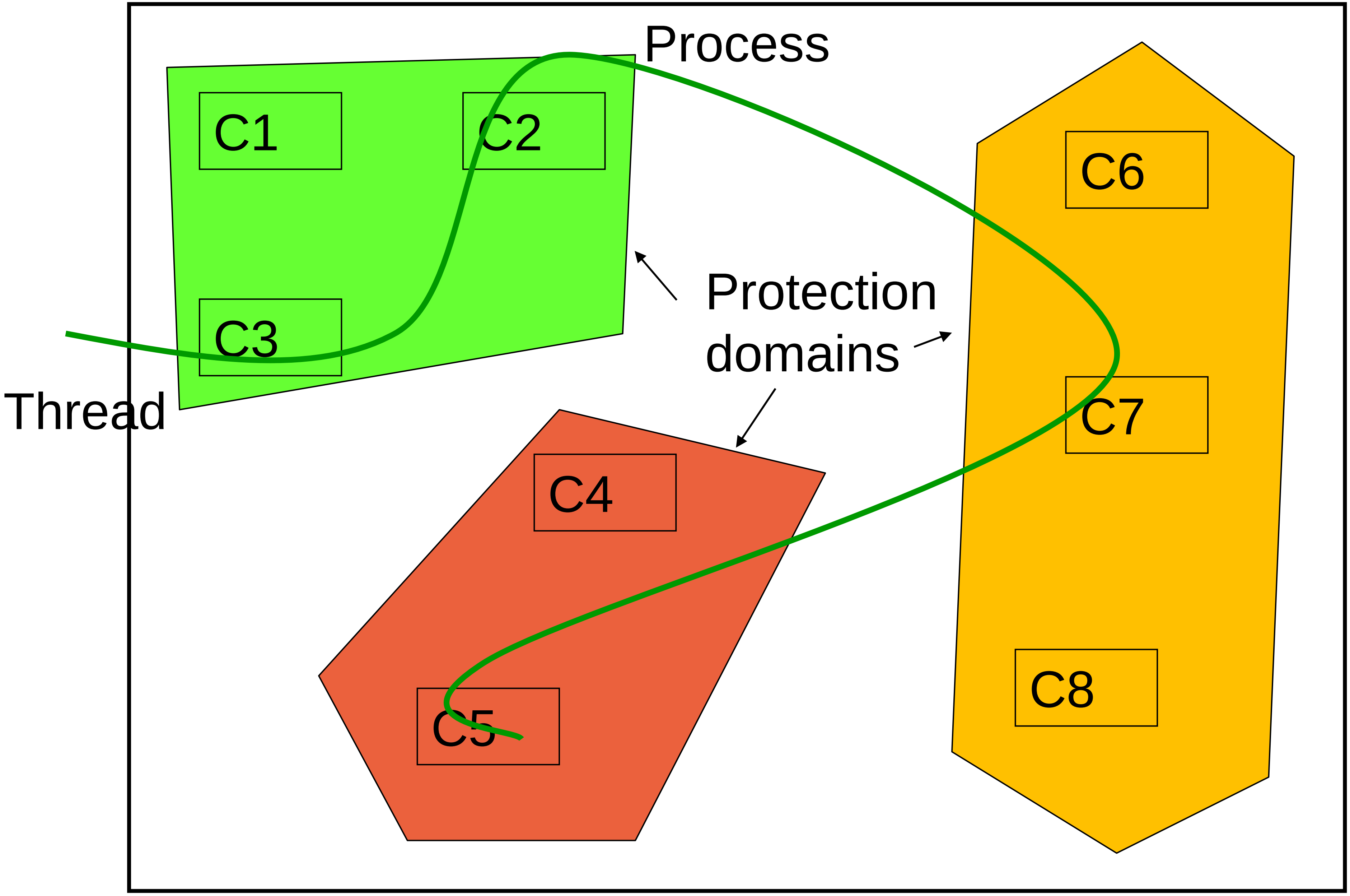
- The algorithm for granting permission is based on *stack inspection* aka *stack walking*

Stack inspection first implemented in Netscape 4.0,

then adopted by Internet Explorer, Java, .NET

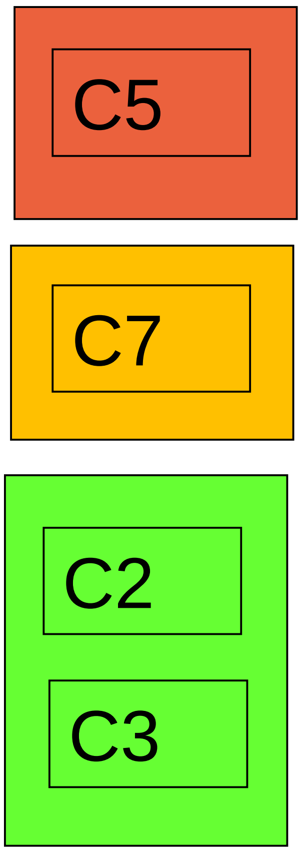
Components and permissions in VM memory




Process
C1
C2
C3
Protection
domains
Thread
C4
C5
C6
C7
C8

Stack walking: basic concepts

Suppose thread T tries to access a resource



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

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

Stack walking

Basic algorithm is *too restrictive* in some cases

E.g.

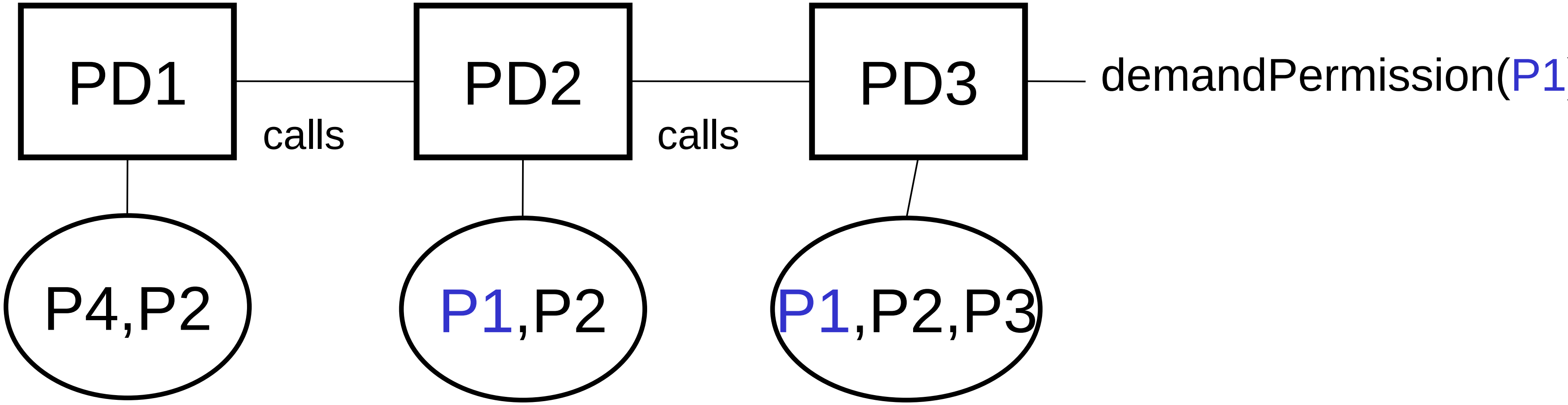
- allowing an untrusted component to delete some specific files
- giving a partially trusted component the right to open speciallay marked windows (eg. security pop-ups) without giving it the right to open arbitrary windows
- giving an app the right to phone certain phone numbers (eg. only domestic ones, 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 real privilege* (ie. only giving or asking the privileges *really* needed, and *only when* they are really need)

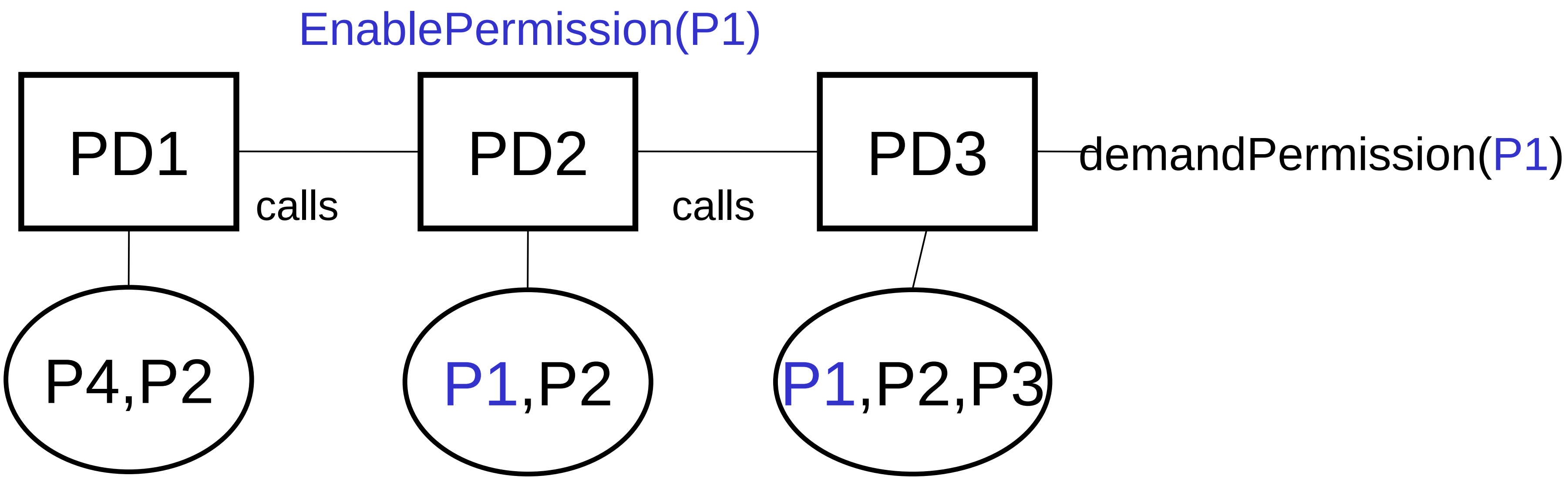
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

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

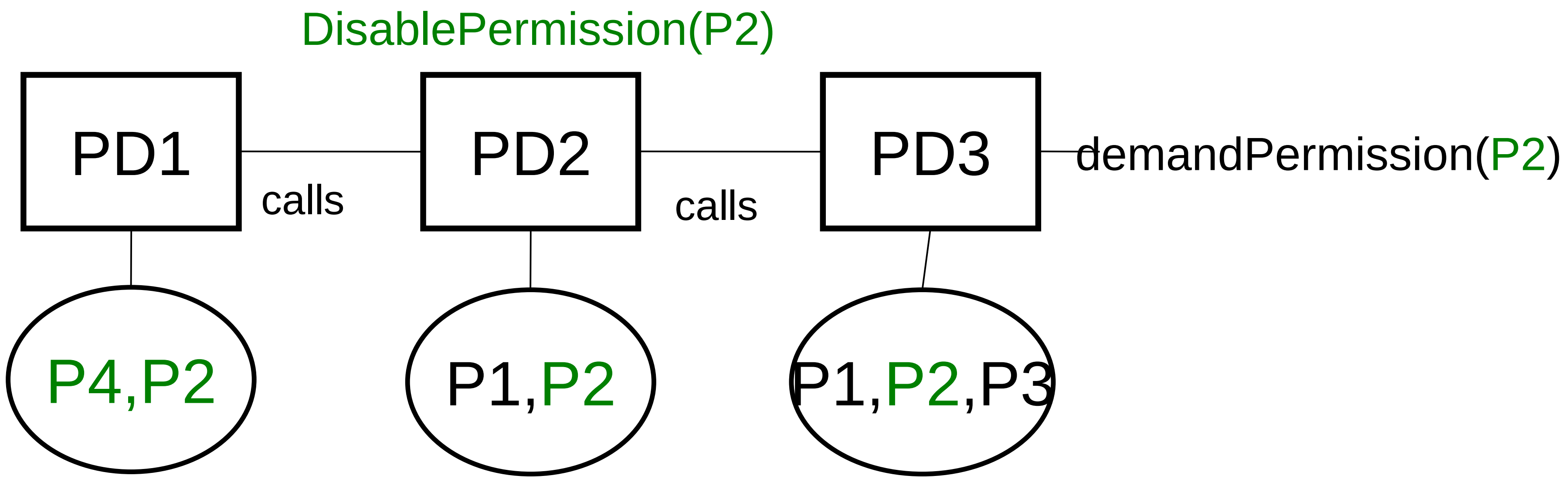
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

DemandPermission(P1) succeeds

Stack walk modifiers: examples



Will DemandPermission(P2) succeed ?

DemandPermission(P2) fails

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

Using stack walking to restrict access to functionality

```
Class Good{
    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;

.... }

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

m2 is insecure, because byte arrays are mutable; an attacker could change the value of filename after the checks, in a multi-threaded execution

Programming language platform vs OS

Note the similarity between

- a method call in which some permissions are enabled
- a Linux **`setuid root`** program or Windows Local System Service that can be started by any user but runs in administrator mode

Both are trusted components that elevate the privileges of their clients

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

In any code review, such code requires extra attention!

Hardware-based sandboxing

- also for unsafe languages

Sandboxing in unsafe languages

- Unsafe languages cannot provide sandboxing at language level
- An application written in an unsafe language could still use sandboxing at the level of the OS (like eg. Chrome does)
  - ie. by splitting the code across different OS processes
- An alternative approach:
  use sandboxing support provided by underlying hardware
- Additional benefit: drastically reducing the size of TCB, esp. keeping the main OS outside of the TCB when executing security-sensitive code.
  - less flexible that eg Java sandboxing,
    but more secure by having a smaller TCB:
    - the "platform", incl. VM and OS, no longer in the TCB

Example: security-sensitive code in larger program

secret.h

```
int get_secret(int provided_pin)
```

secret.c

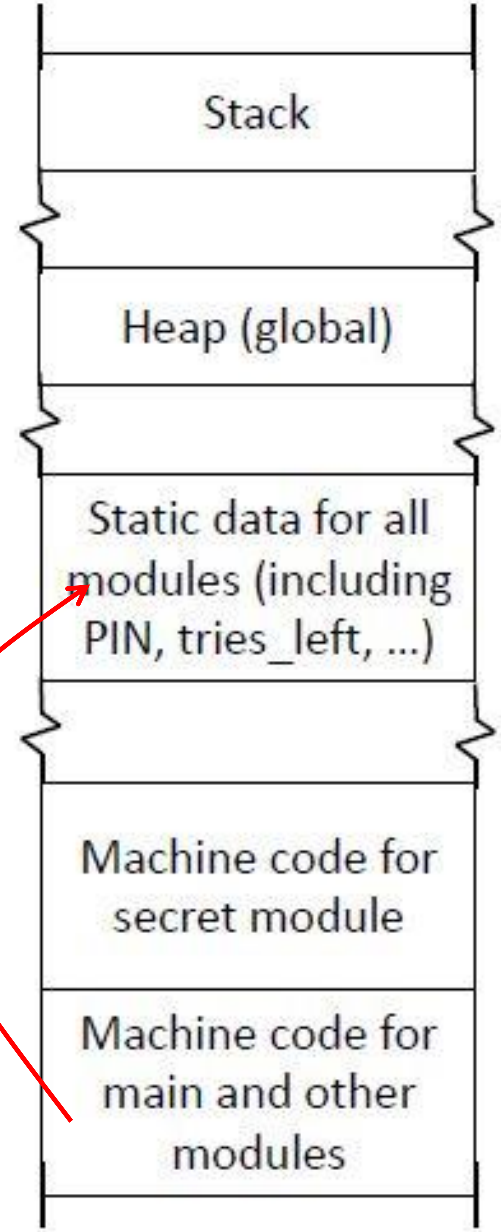
```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
     tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
...
}
```

(b) Other modules of the program

bugs or malicious code could access secret data



Example from [N. van Ginkel et al, Towards Safe Enclaves, HotSpot 2016]

Isolating security-sensitive code with secure enclaves

secret.h

```
int get_secret(int provided_pin)
```

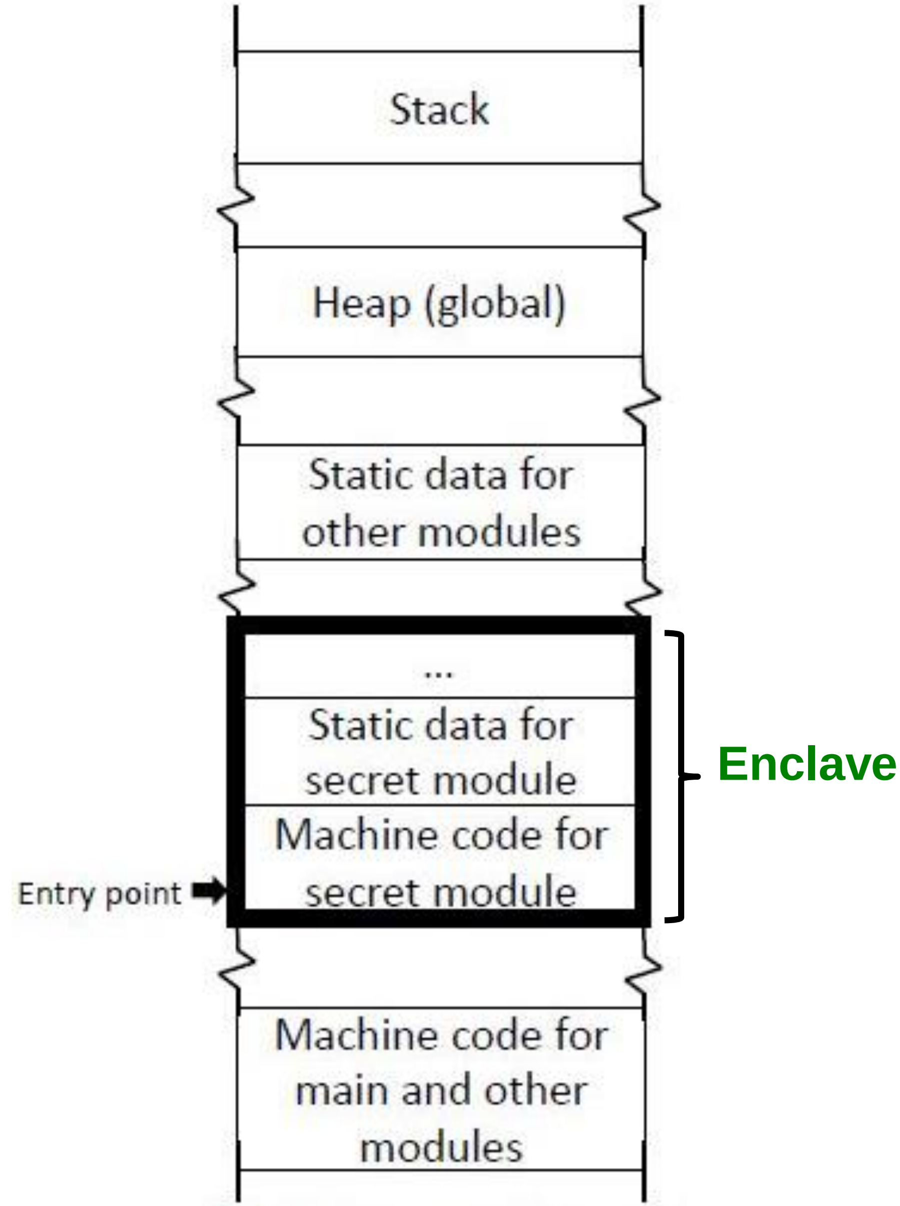
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
     tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
...
}
```

(b) Other modules of the program



Example from [N. van Ginkel et al, Towards Safe Enclaves, HotSpot 2016]

Isolating security-sensitive code with secure enclaves

secret.h

```
int get_secret(int provided_pin)
```

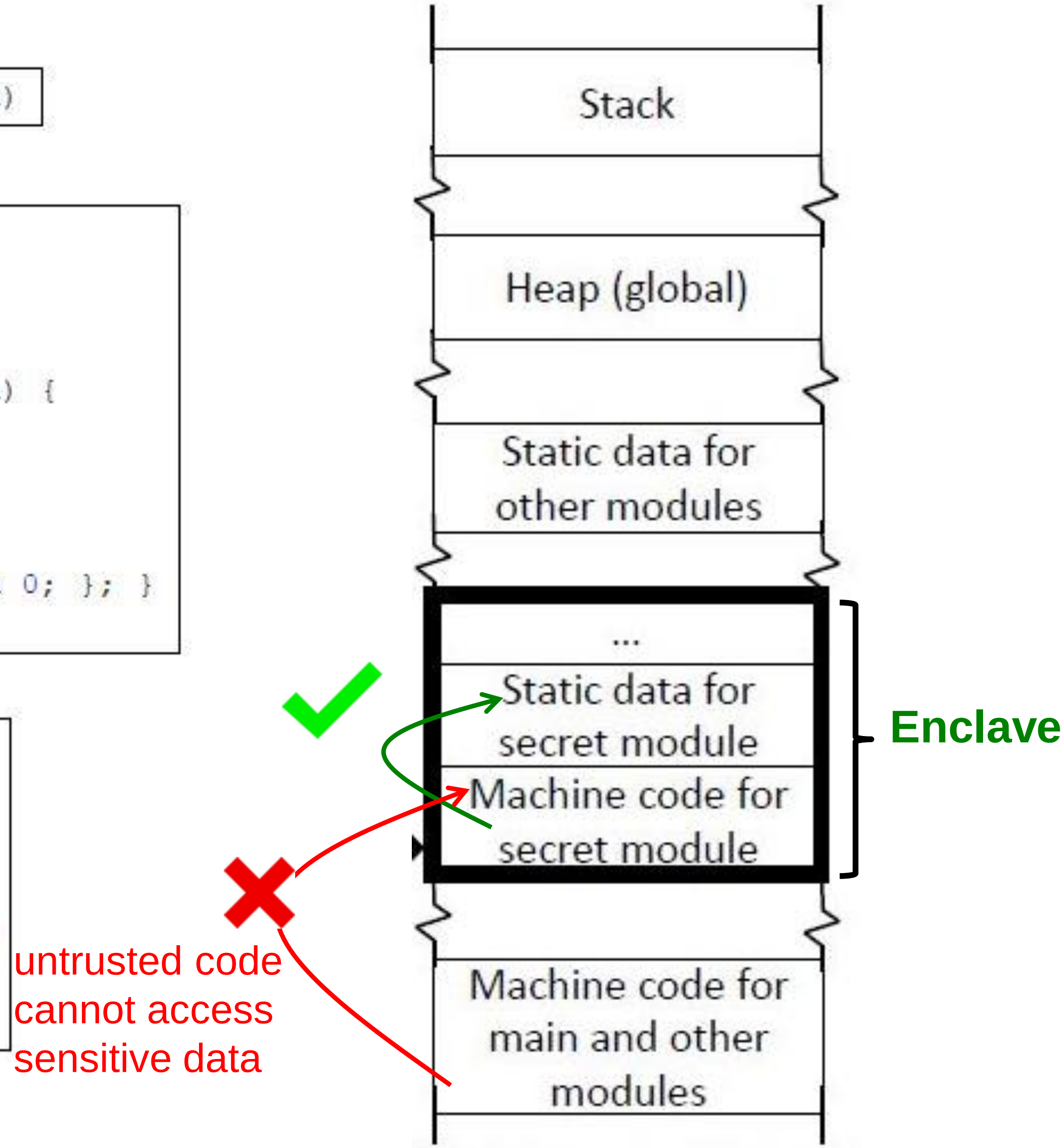
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
     tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
…
}
```

(b) Other modules of the program



Example from [N. van Ginkel et al, Towards Safe Enclaves, HotSpot 2016]

Isolating security-sensitive code with secure enclaves

secret.h

```
int get_secret(int provided_pin)
```

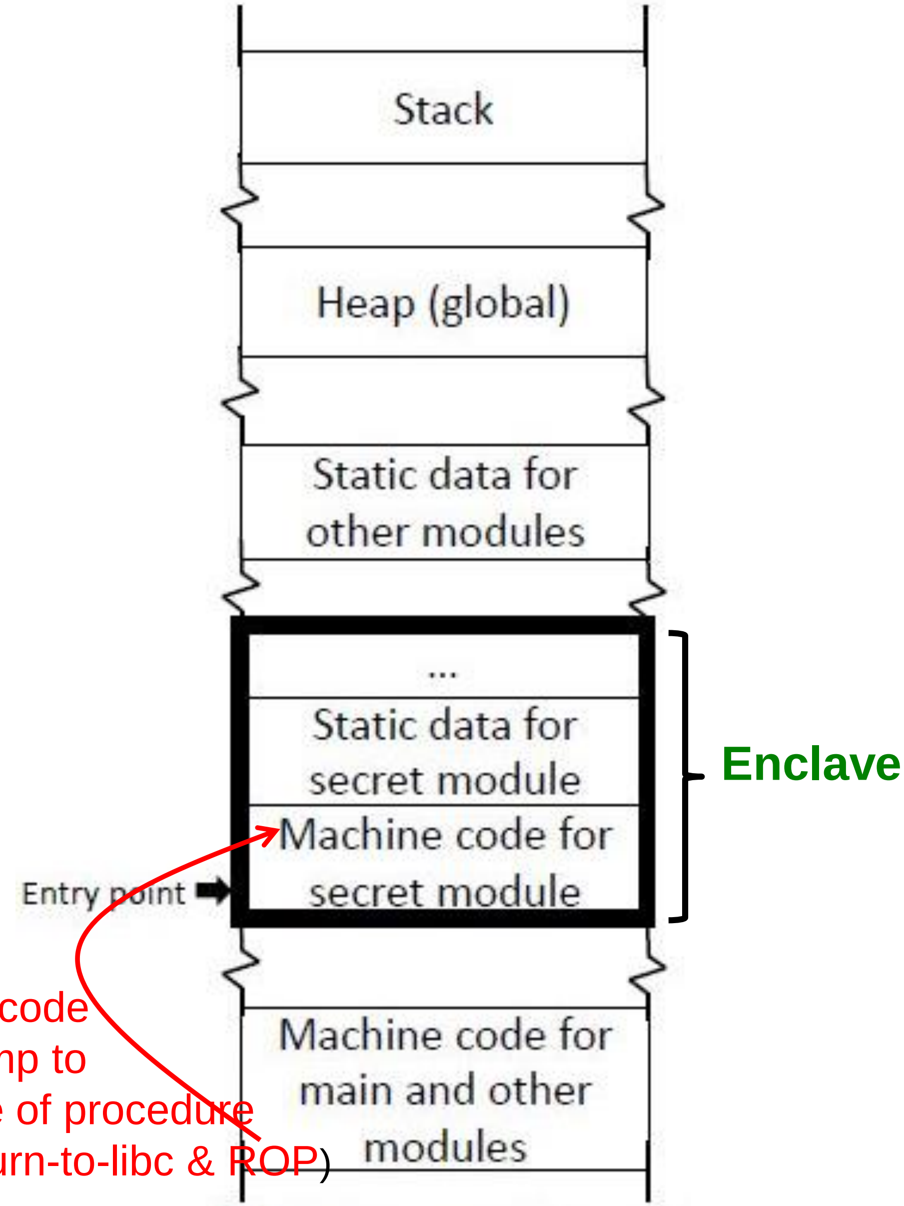
secret.c

```
static int tries_left = 3;
static int PIN = 1234;
static int secret = 666;

int get_secret(int provided_pin) {
 if (tries_left > 0) {
   if (PIN == provided_pin) {
     tries_left = 3;
     return secret;}
   else { tries_left-- ; return 0; }; }
else return 0; }
```

(a) The secret module

```
#include<stdio.h>
#include "secret.h"
// includes for other modules

void main() {
// code for main functionality
...
}
```

(b) Other modules of the program



untrusted code
cannot jump to
the middle of procedure
(recall return-to-libc & ROP)

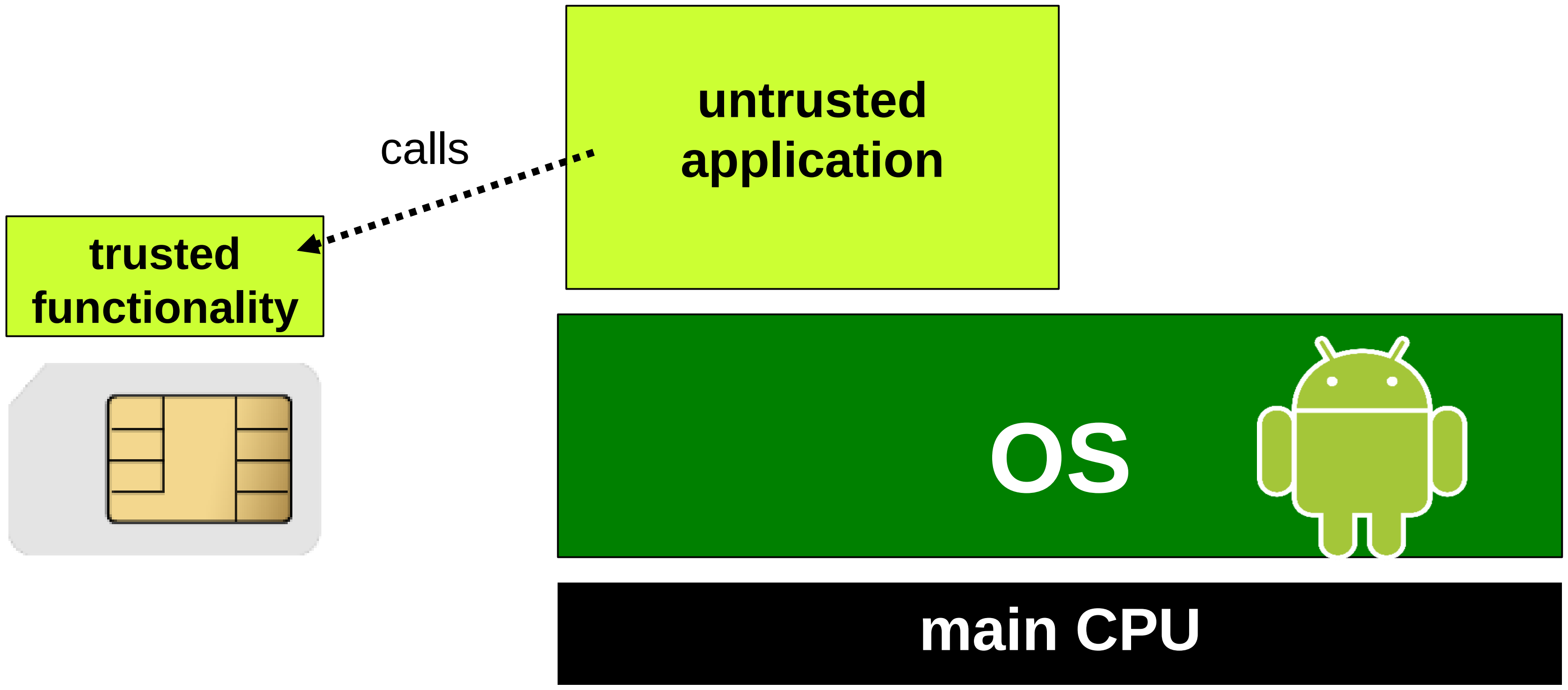
Example from [N. van Ginkel et al, Towards Safe Enclaves, HotSpot 2016]

Secure enclaves

- Enclaves isolates part of the code together with data
  - Code outside the enclave cannot access the enclave's data
  - Code outside the enclave can only jump to valide entry points for the code in the enclave
- Less flexible than stack walking:
  - code in the enclave cannot inspect the stack as the basis for security decisions
  - not such a rich collection of permissions, and programmer cannot define his own permissions
- More secure, because
  - OS & VM are not in the TCB
  - also some protection against physical attacks is possible

Analogy: SIM card in phone

A SIM also provide a secure enclave for providing some trusted functionality (with a small TCB) to a larger untrusted application (with a larger TCB)



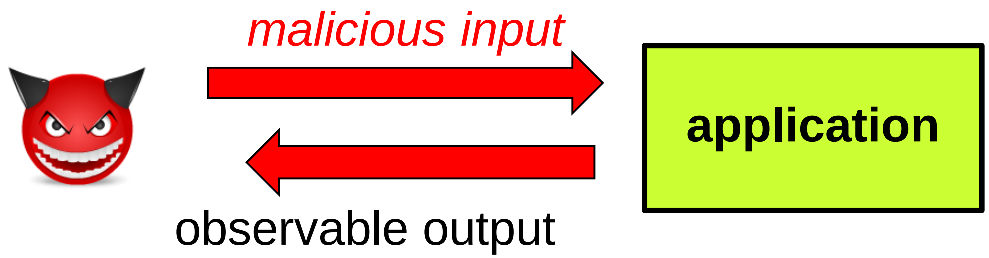
Realising safe enclaves

Different hardware-based mechanisms proposed to provide the isolation for secure enclaves aka protected modules. incl.

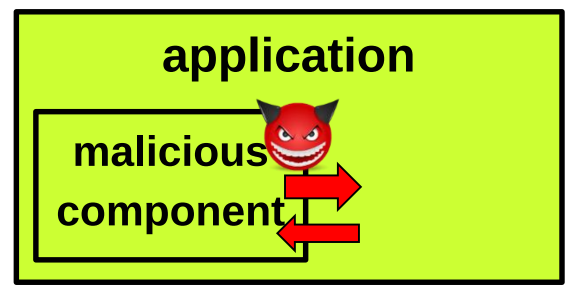
1. Flicker: processor switches to a different mode, suspending the main OS, with the help of a TPM
2. Physically separate hardware, eg SIM card or Secure Element in phone
3. Using Trusted Execution Enviroments (TEEs), where processor can run in two modes, to offer a secure & an insecure world
   – eg **Intel SGX** and **ARM Trustzone**
4. Using processor that can do memory access control based on the value of the program counter: execution-aware memory protection (discussed as buffer overflow countermeasure)
   – more lightweight approach than TEE

Different attacker models for software

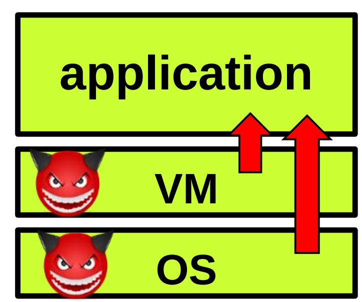
1. **I/O attacker**



2. **malicious code attacker**
   inside the application



3. **the platform level attacker**
   'inside' the platform under the application



Java sandboxing protects against 2, SGX enclaves also against 3

In all cases, the application itself will *still* have to make sure it exposes only the right functionality, correctly & securily (eg. with all input validation in place)

Recap

- Language-based sandboxing is a way to do access control within a application: *different access right for different parts of code*

  We want this

  – to reduce the TCB for some functionality provided by that application

  – *when we run code from many sources on the same VM and don't trust all of them equally*

  – *to limit code review to small part of the code*

  – *...*

- Safe programming language like Java offer language mechanisms for this

- Hardware-based sandboxing can also achieve this also for unsafe programming languages

  – has much smaller TCB: OS and VM are no longer in the TCB

  – but a less expressive & flexible mechanism