Software Security

Application-level sandboxing

Erik Poll

Radboud Universiteit Nijmegen

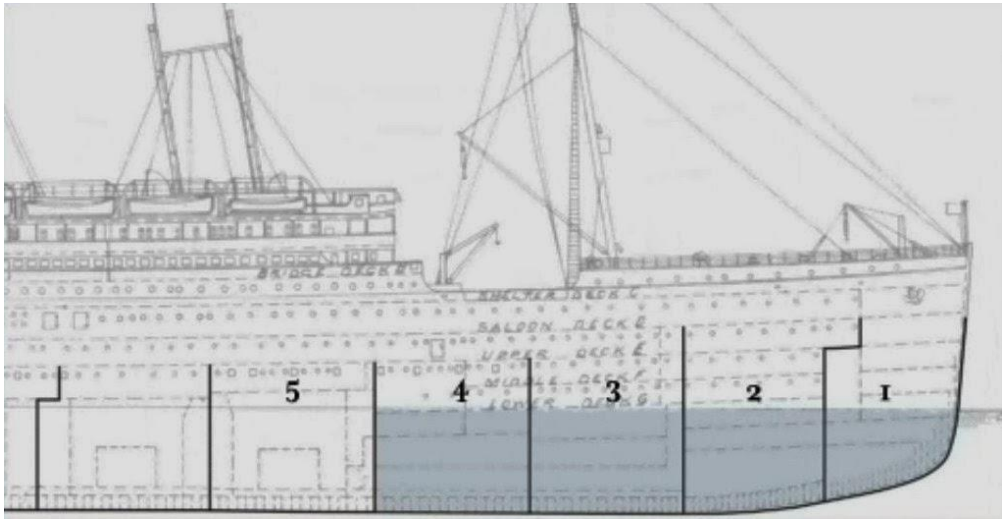
TRU/e Master in Cyber Security

Overview

1. Compartementalisation
2. Classic OS access control
   - compartementalisation *between* processes
   - Chapter 2 of lecture notes
3. Language-level access control
   - compartementalisation *within* a process
   - by sandboxing support in safe programming languages
     - notably Java and .NET
   - Chapter 4 of lecture notes
4. Hardware-based sandboxing
   - compartementalisation *within* a process, also for unsafe languages

1. Compartmentalisation

Compartmentalisation in ships



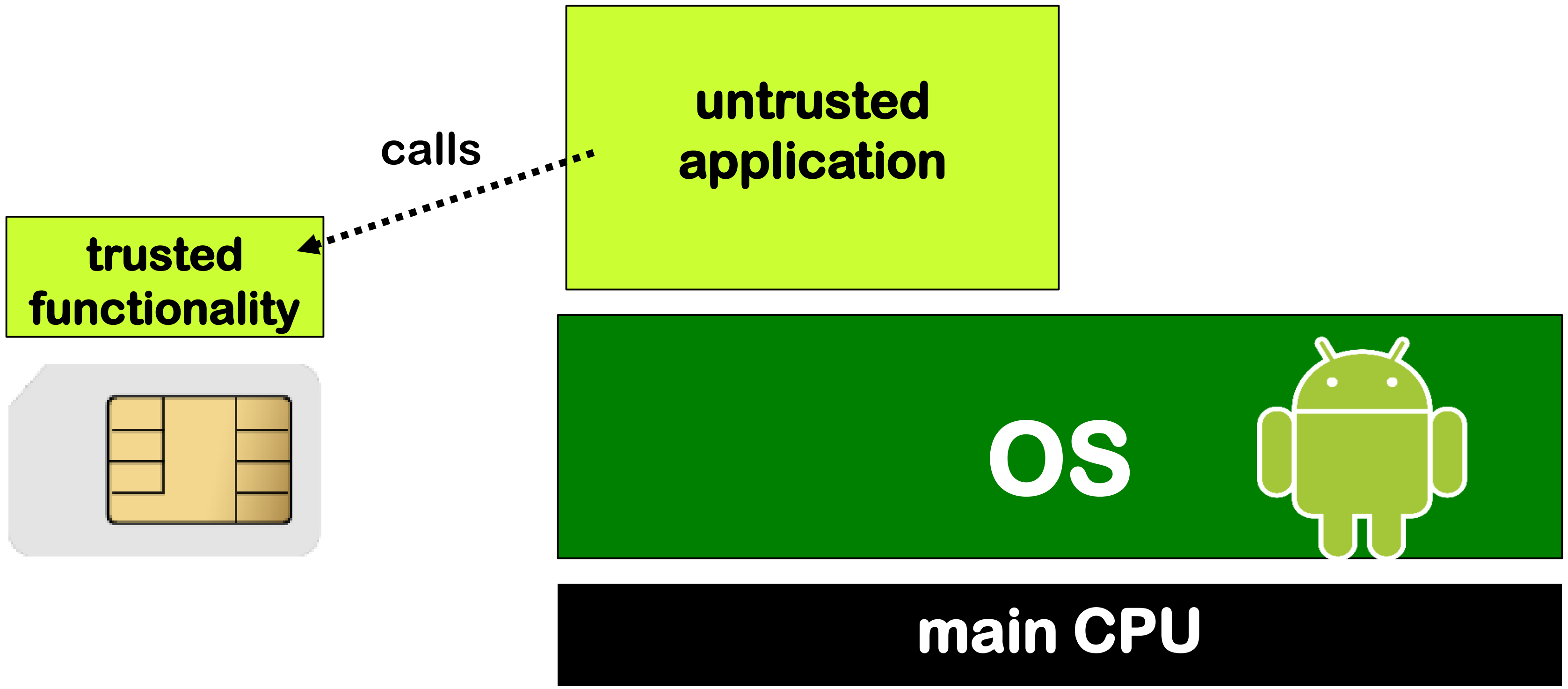
Compartmentalisation examples

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 2 OSs
- Inside a program
  - different ‘modules’ with different tasks

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)



Compartmentalisation for security

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 (Trusted Computing Base) for certain security-sensitive functionality

b. Reduces the impact of any security flaws.

Sandboxing

Sandboxing aka access control the standard way to provide compartmentalisation.

It involves

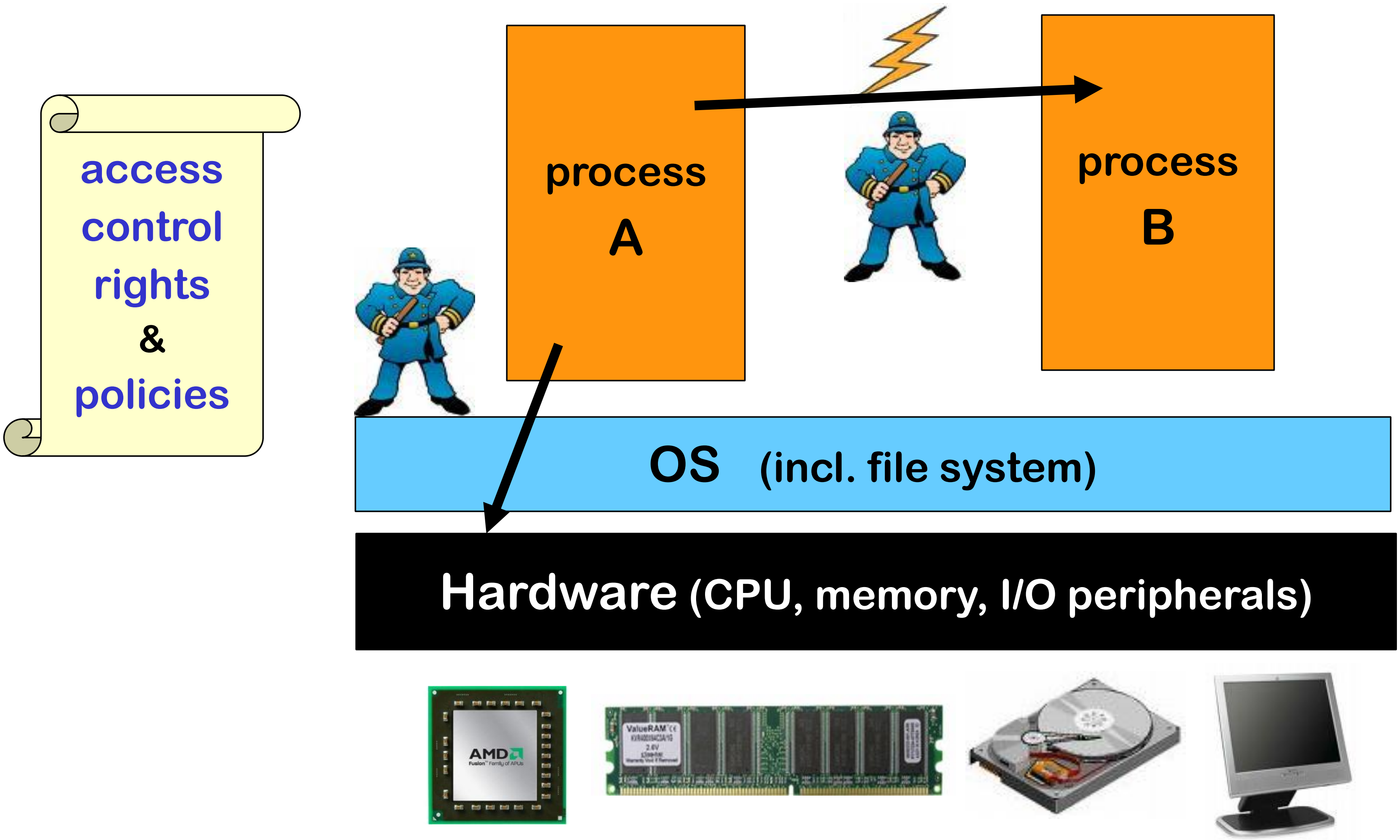
1. rights/permissions
2. parties (eg. users, processes, components)
3. policies that give rights to parties

   – specifying who is allowed to do what
4. runtime monitoring to enforce policies

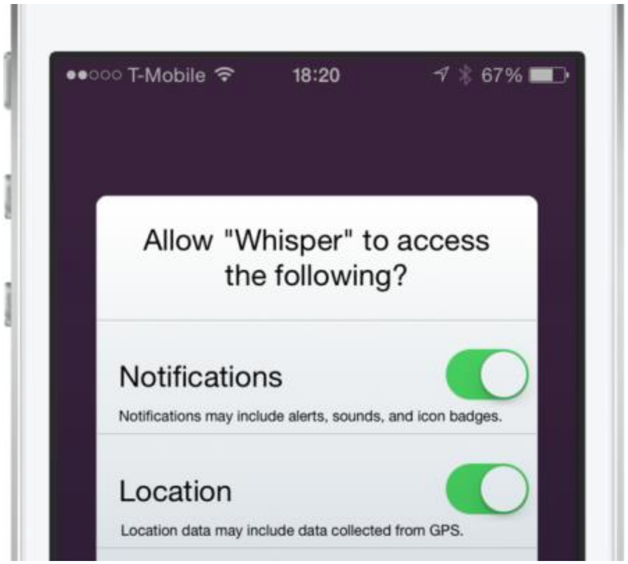
2. Operating System (OS) Access Control

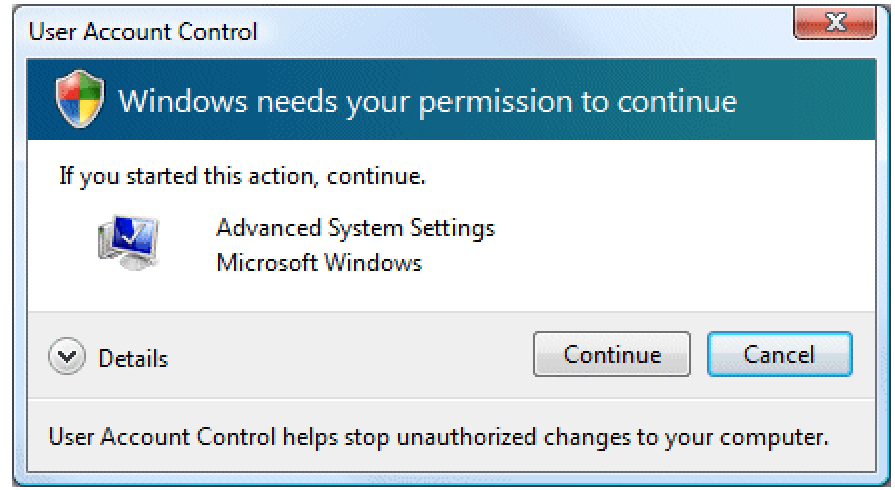
See also Chapter 2 of the lecture notes

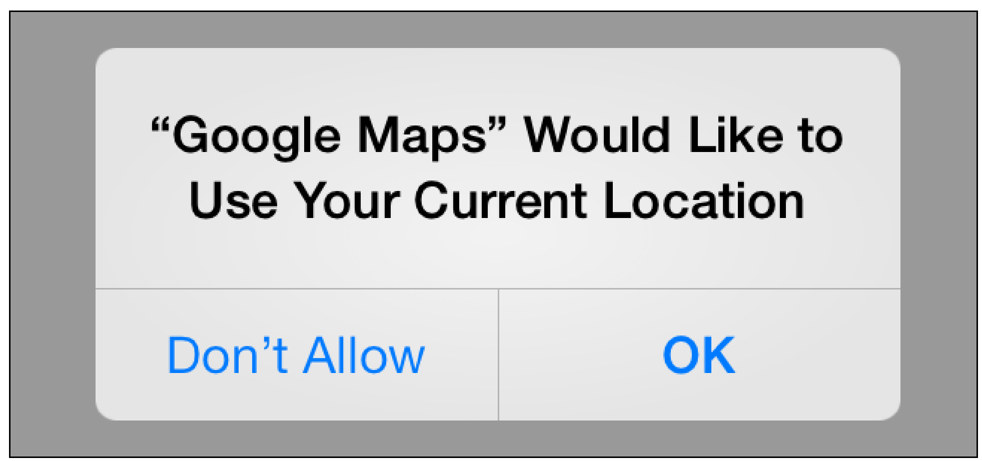
Classical OS-based security (reminder)



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 security flaws in the code.

   The only safe assumption: a malicious process on a typical OS (Linux, Windows, BSD, iOS, Android, ...) *will* be able to get superuser/root/administrator 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 problem (resulting in *privilege escalation)*

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

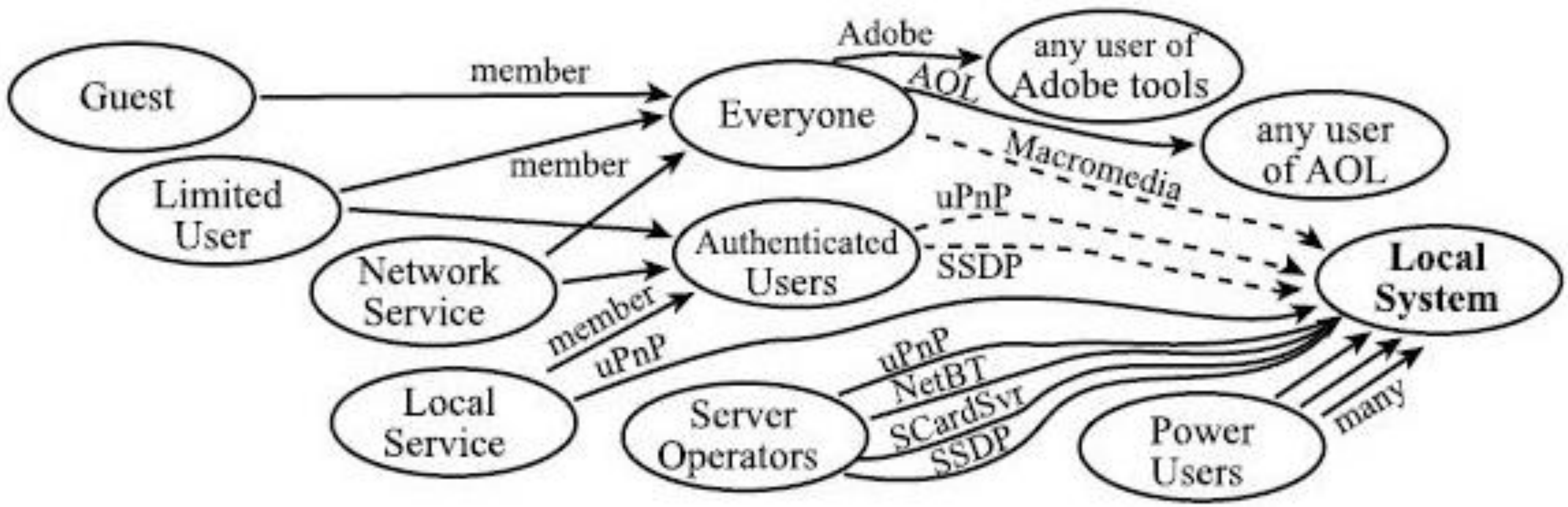
Windows XP uses 30 permissions, 9 categories of users, and 15 kinds of objects.

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*
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:



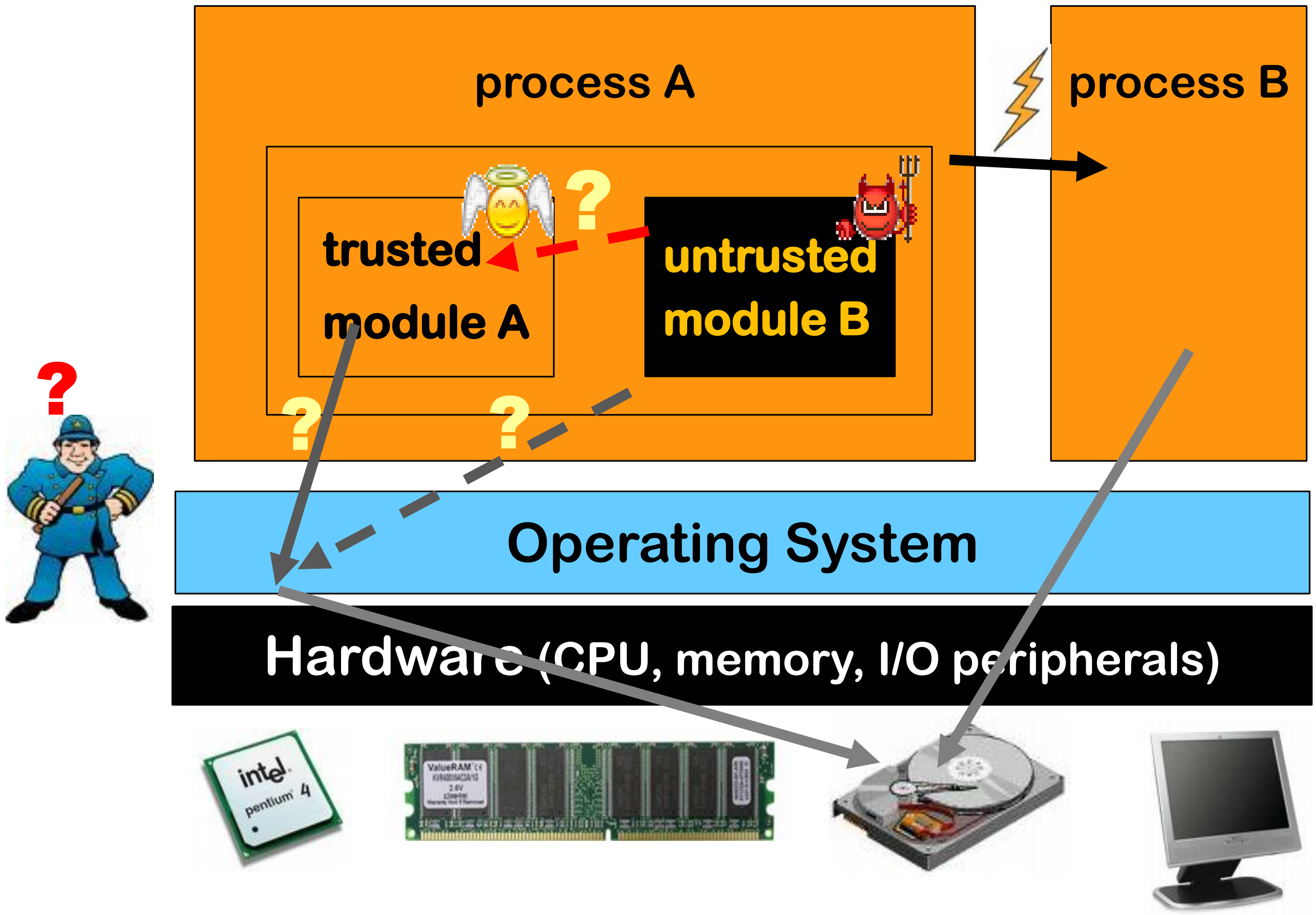
[S. Govindavajhala and A.W. Appel, Windows Access Control Demystified, 2006]

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

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

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

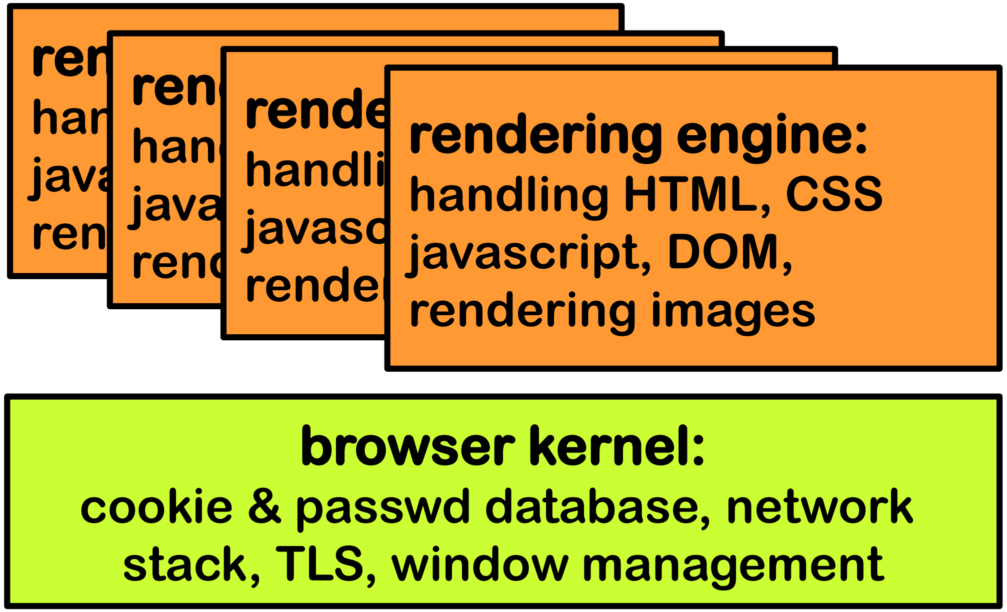
- A process has a fixed set of permissions. Usually, all permissions of the user who started it
- Execution with reduced permission set may be needed temporarily when executing untrusted or less trusted code. For this OS access control may be too coarse.

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: 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*

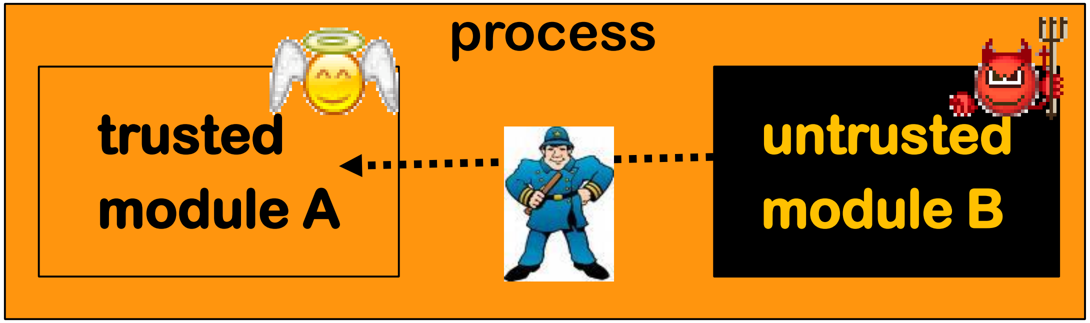
- (Complex!) rendering engine is black box for browser kernel
- Plugins also run as different processes
- Running a new process per domain can enforce the restrictions of the SOP (Same Origin Policy)
- *Advantage: TCB for certain operations drastically reduced*

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 just buggy)

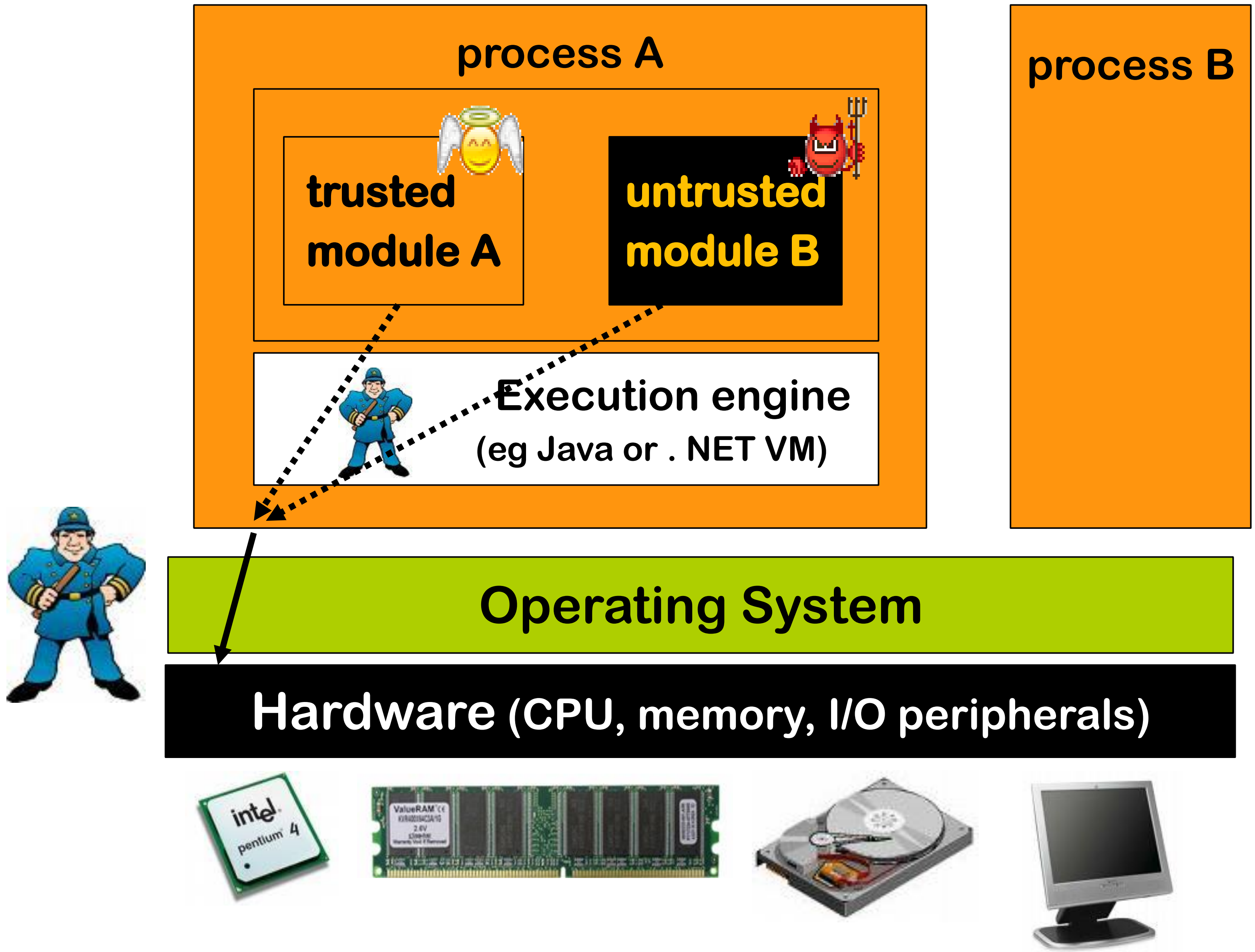
- *Without memory-safety, this is impossible. Why?*

  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

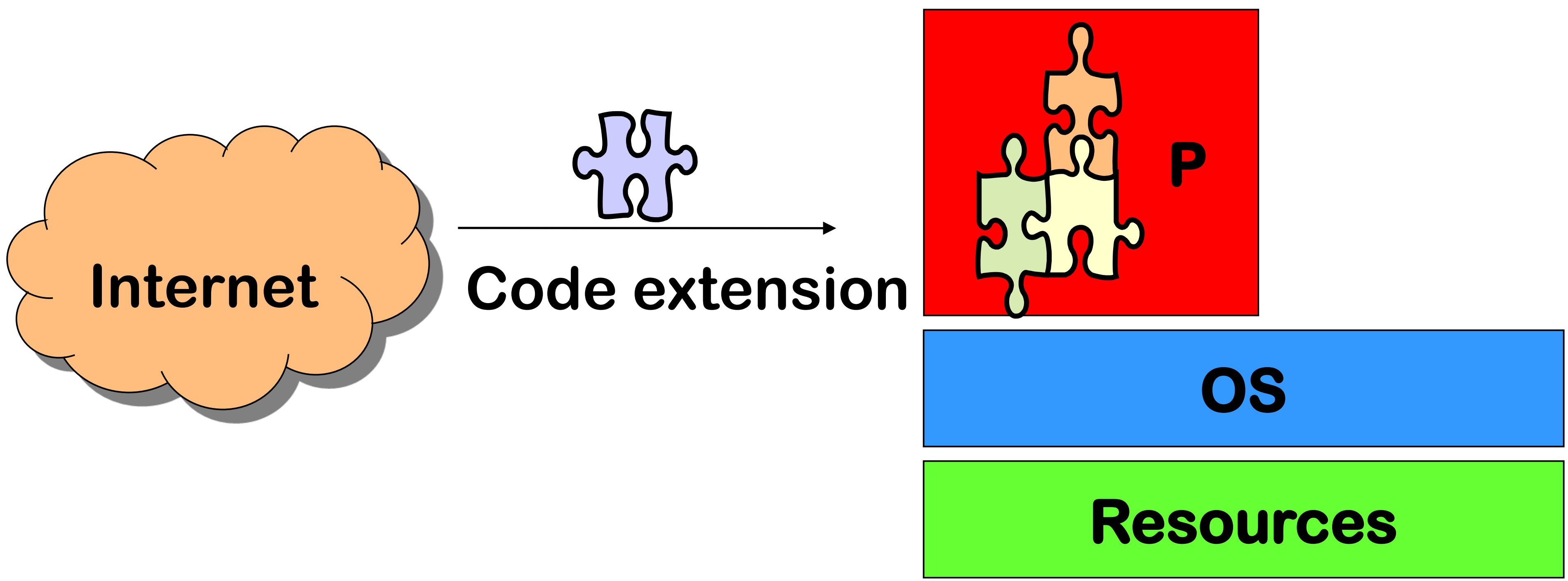
Language-level sandboxing



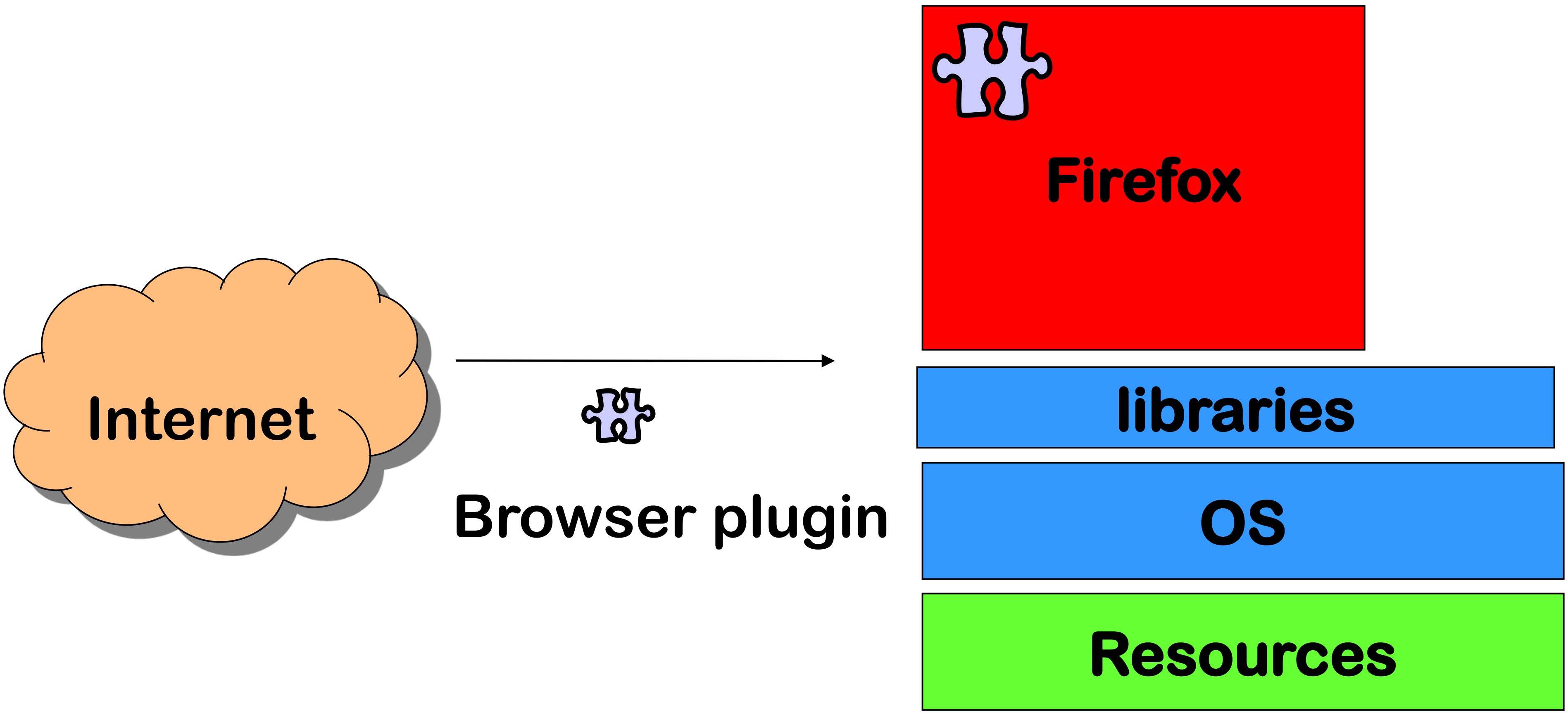
Extensible applications

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

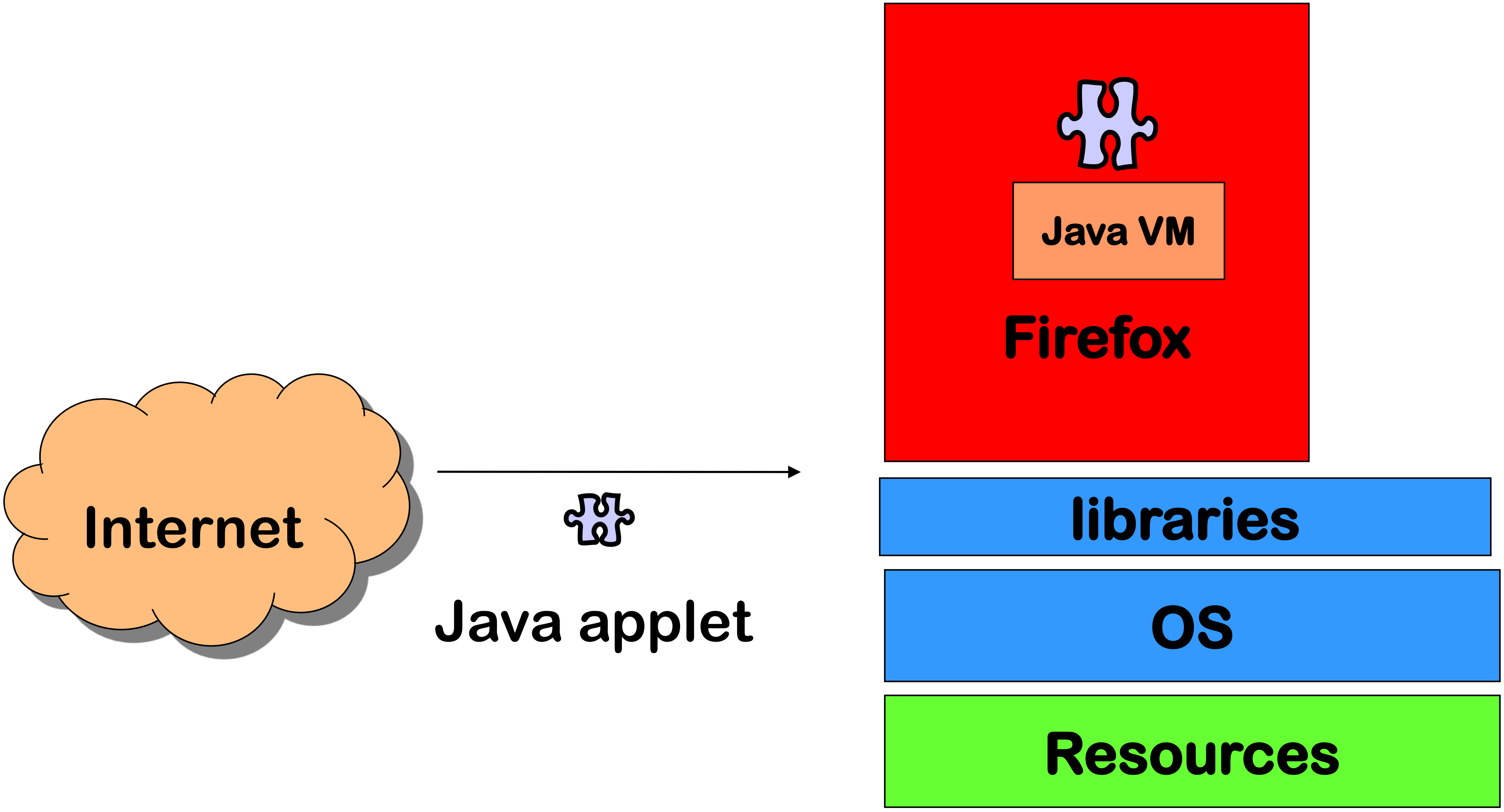
This is especially the case for extensible applications, where at runtime an application can extend itself



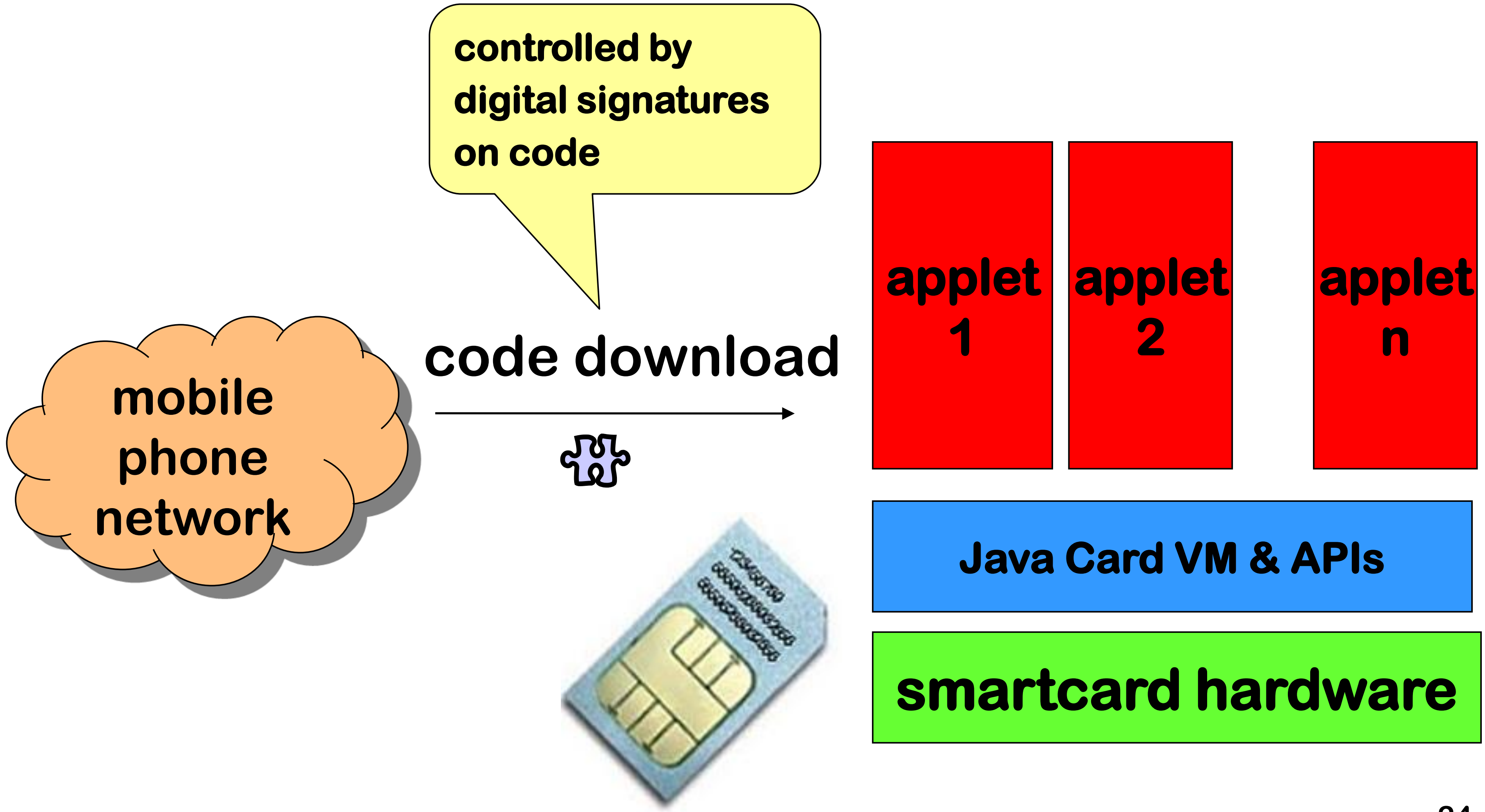
Example: browser plugin



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

- 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

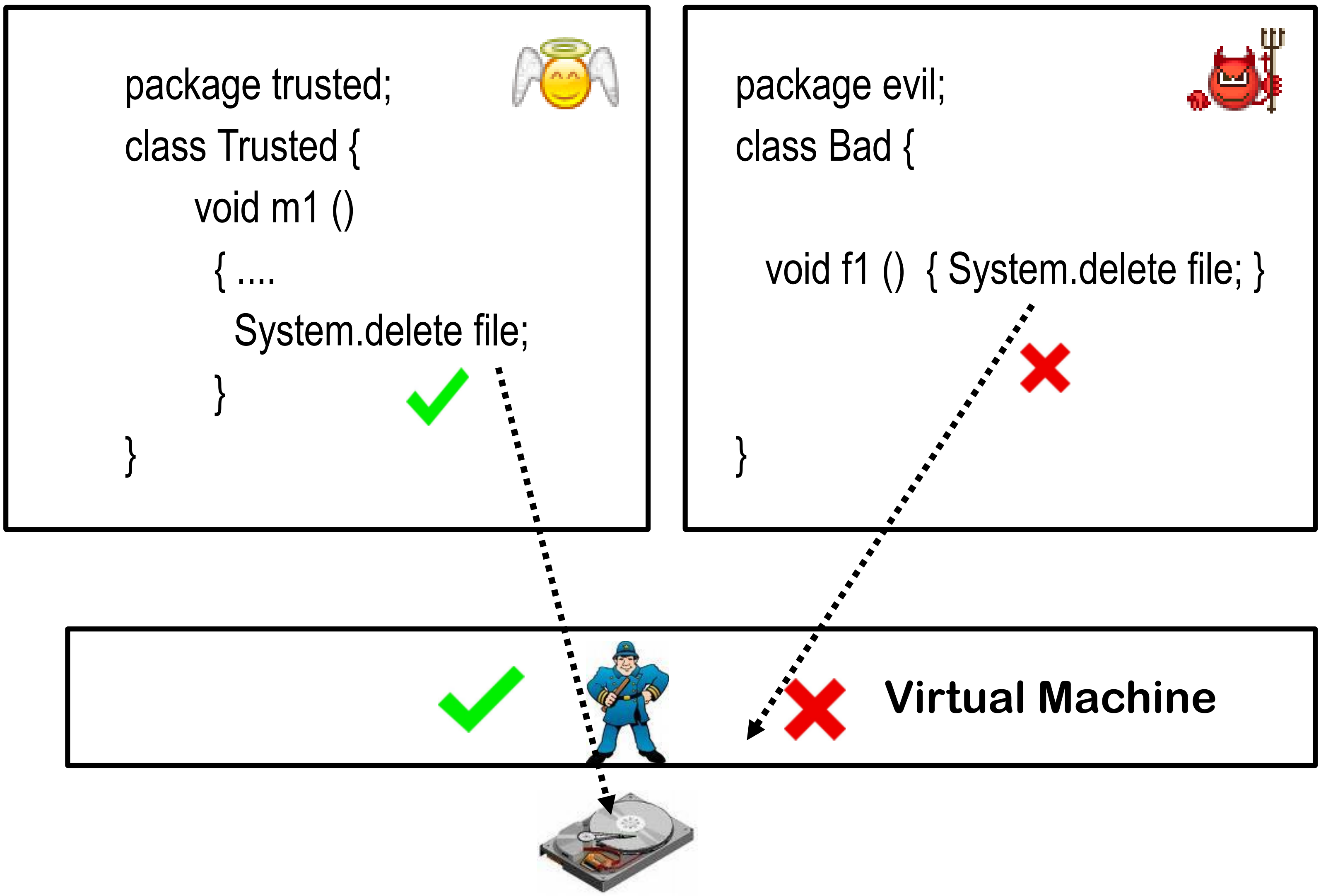
- Protection domains based on evidence

  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

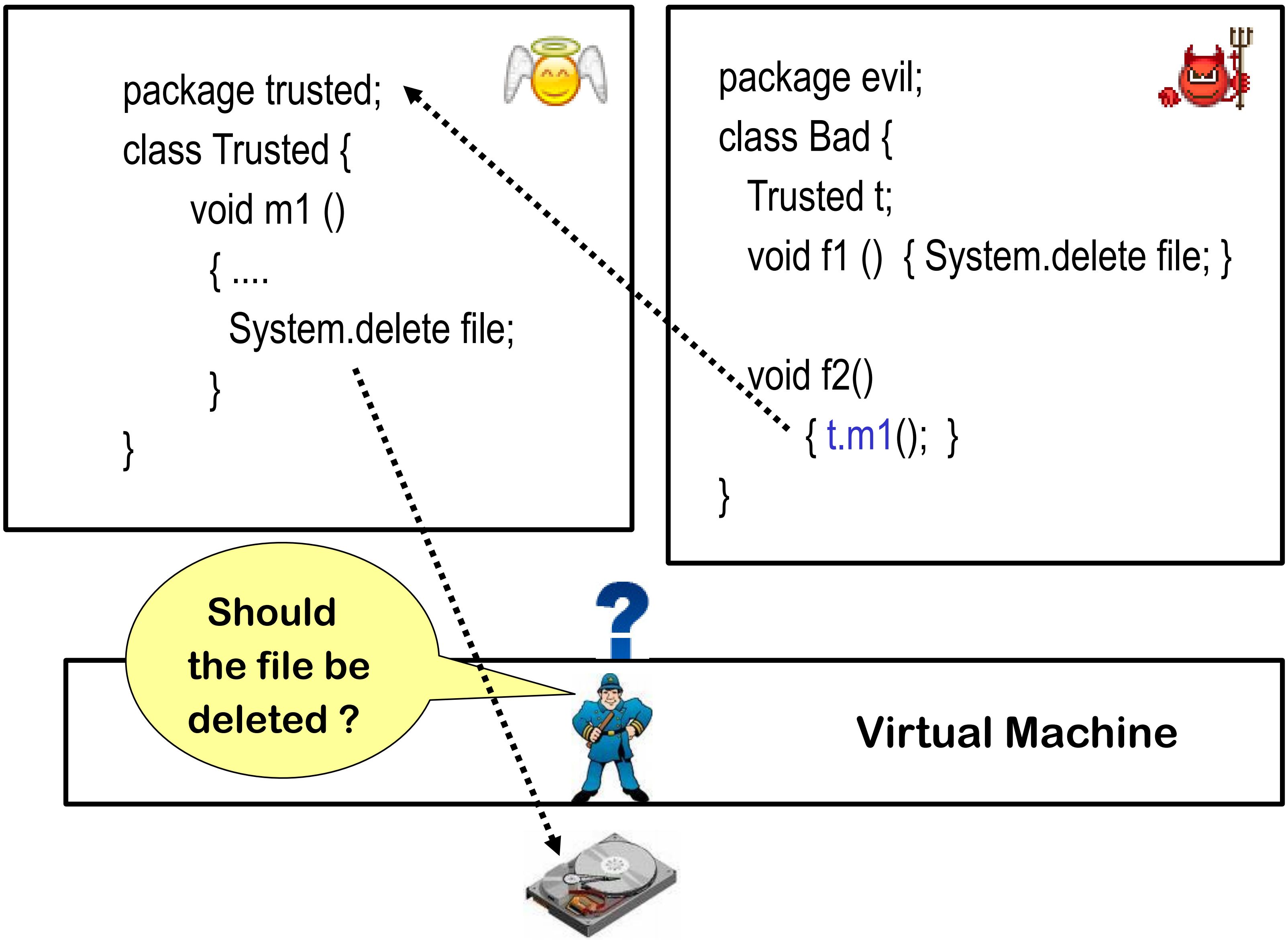
- When loading a component, the Virtual Machine (VM) consults the security policy and remembers the permissions

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.


package trusted;
class Trusted {
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 <u>top frame</u> on the stack has permission
2. only allow action if <u>all frames</u> 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
  public void safeMethod(File f) {
        .... // lots of checks on f;
     if all checks are passed, then dele
        // 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

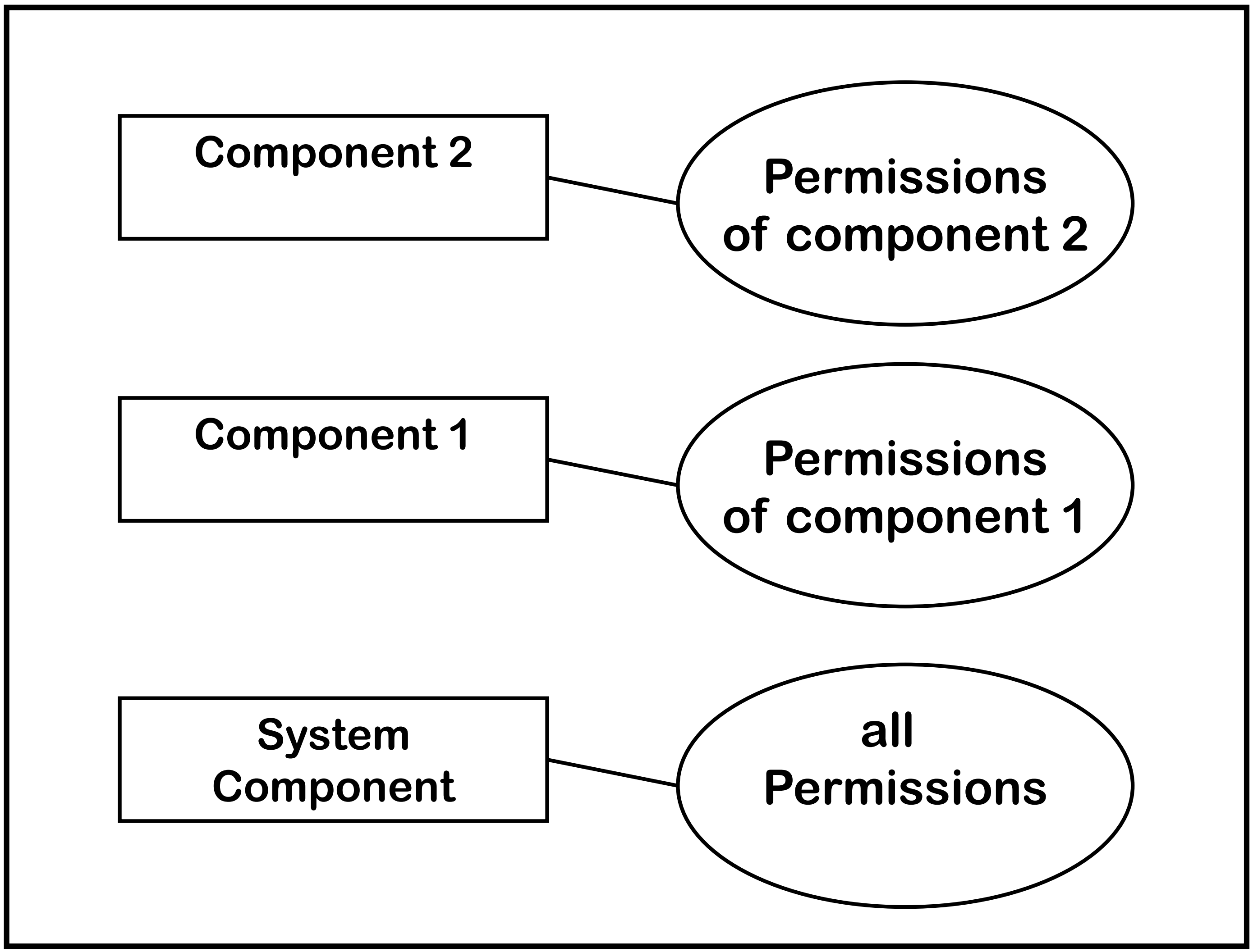
Hence: stackwalking

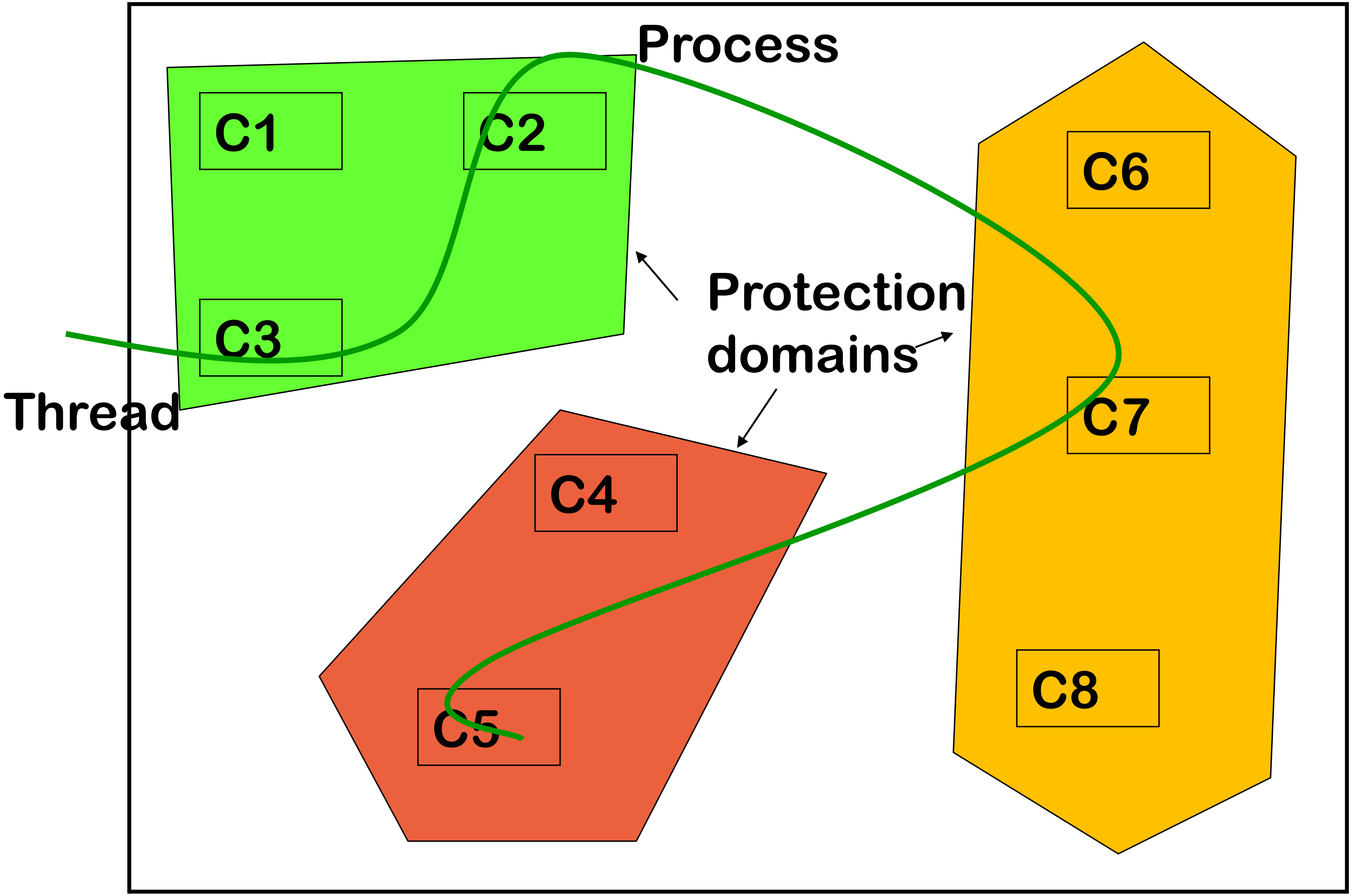
Stack walking

- 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

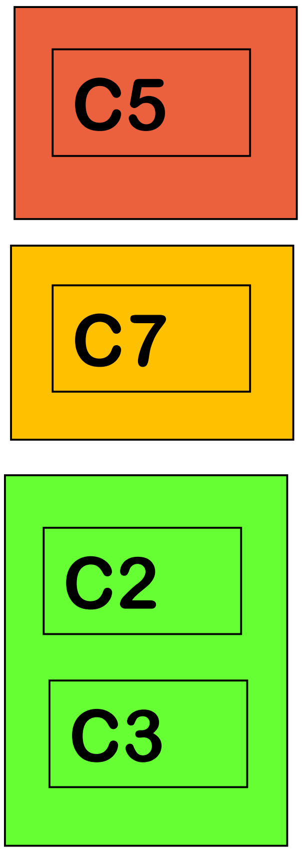
Components and permissions in VM memory




Process
C1
C2
C3
Thread
Protection
domains
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 givie or ask the privileges *really* needed, and *only when* they are really needed)

Stack walking: algorithm

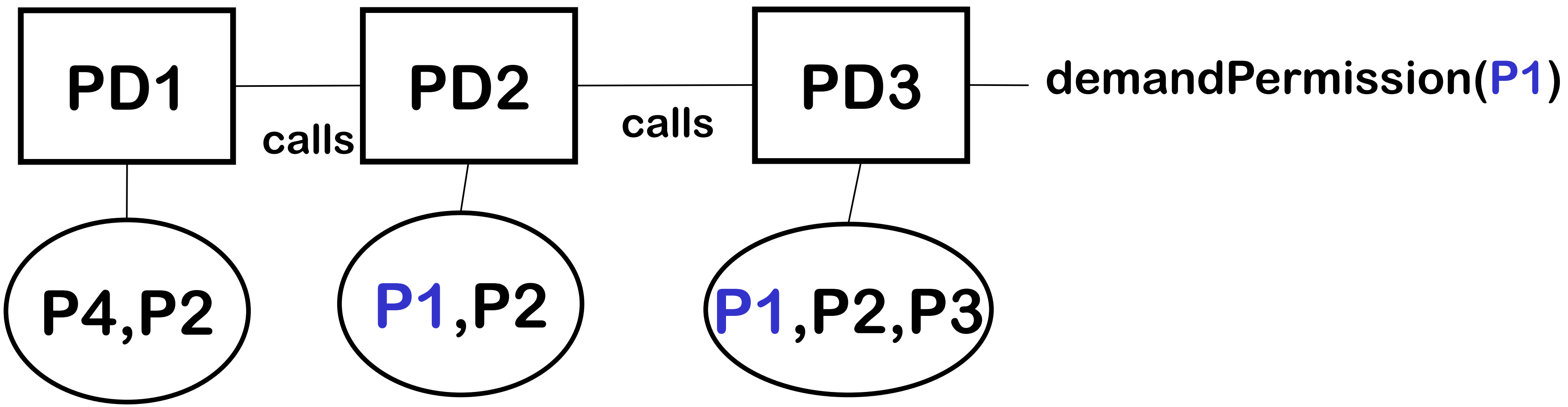
On creating new thread:

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

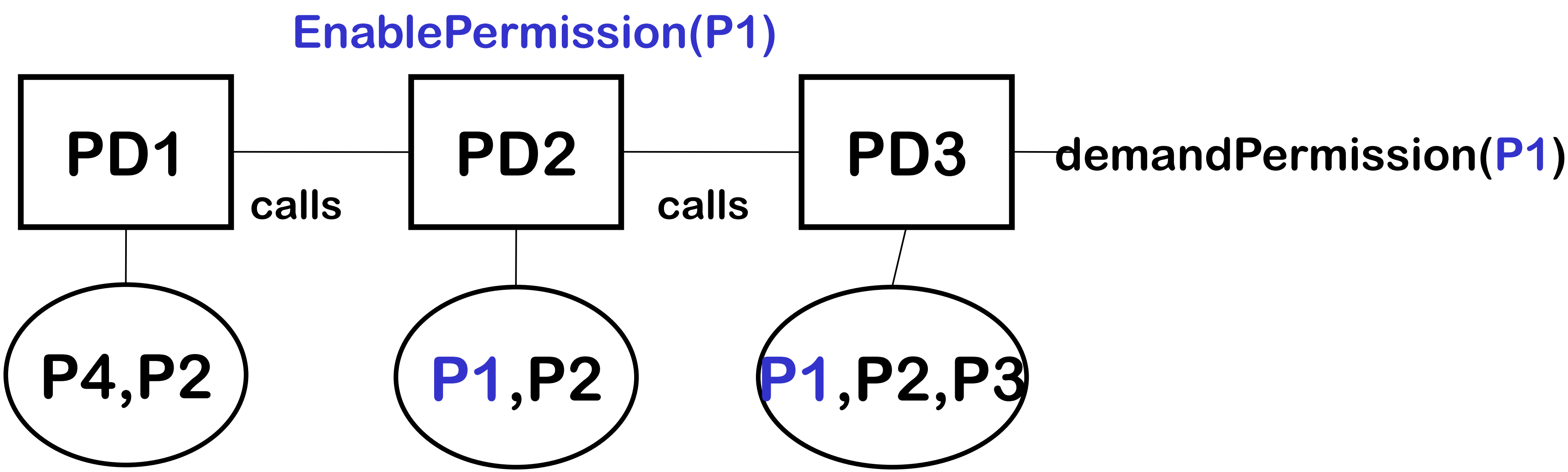
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

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

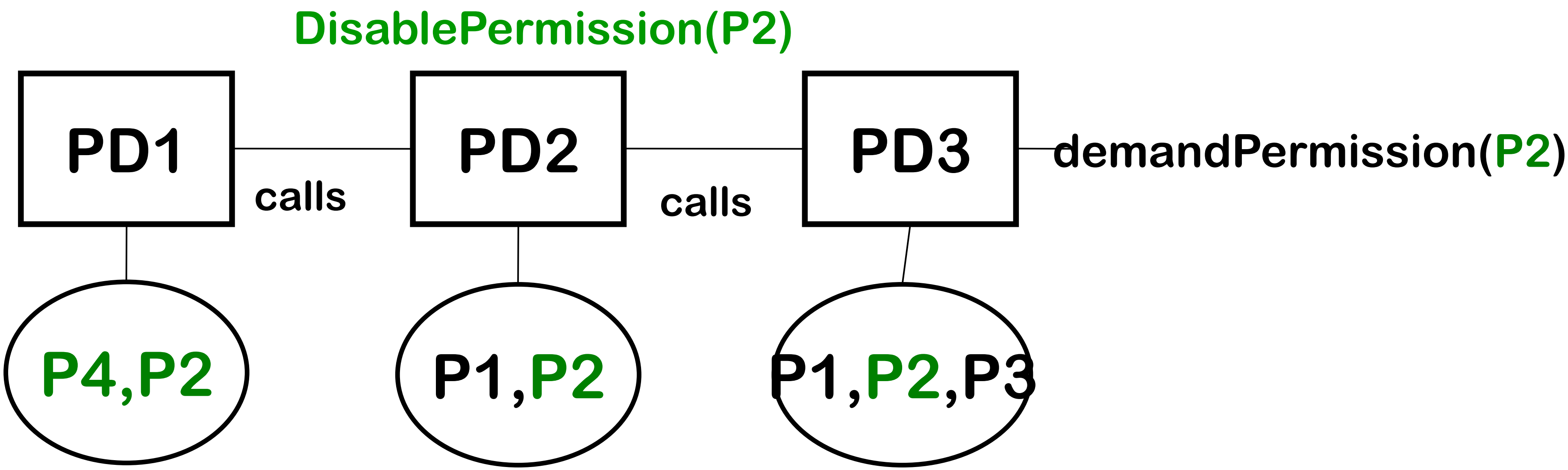
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

DemandPermission(P1) succeeds

Stack walk modifiers: examples



Will DemandPermission(P2) succeed ?

DemandPermission(P2) fails

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 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;
    .... }
```

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 (FileDeletionPermi
        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!

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 OS sandboxing by splitting the code across different processes (as e.g. Chrome does)**
- **An alternative approach:**
  use sandboxing support provided by underlying hardware,
  to impose memory access restrictions inside a process

Example: security-sensitive code in larger program

secret.c

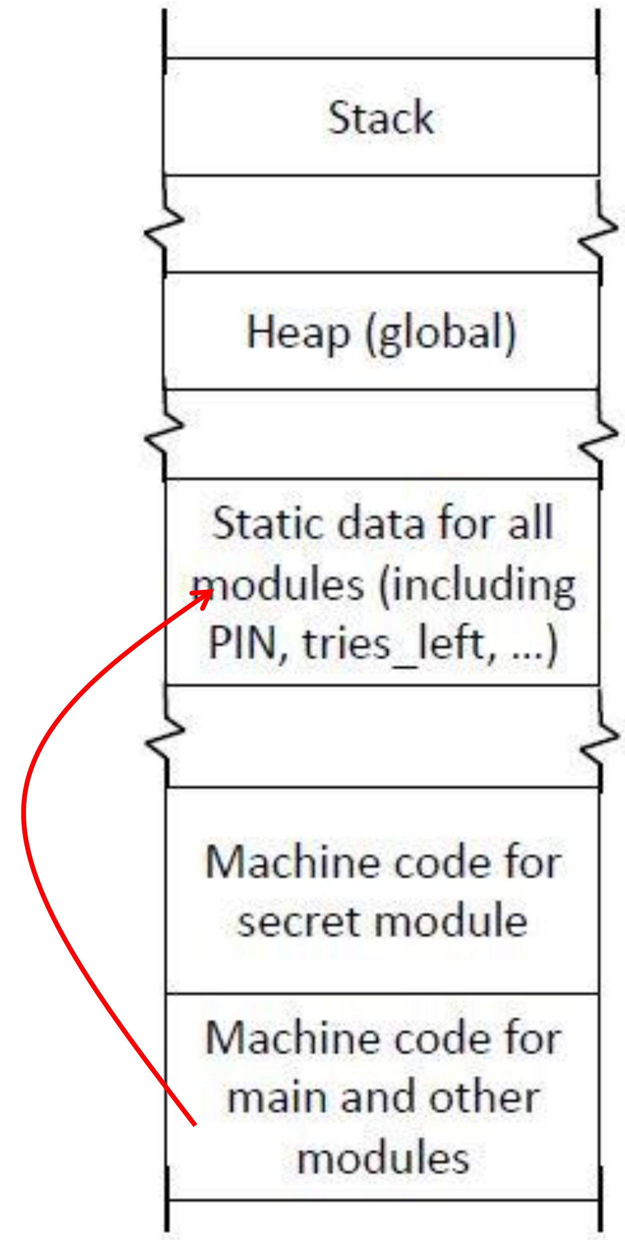
```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

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

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```

Bugs or **malicious code** ***anywhere*** **in the program could access the high-security data**



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

Isolating security-sensitive code with secure enclaves

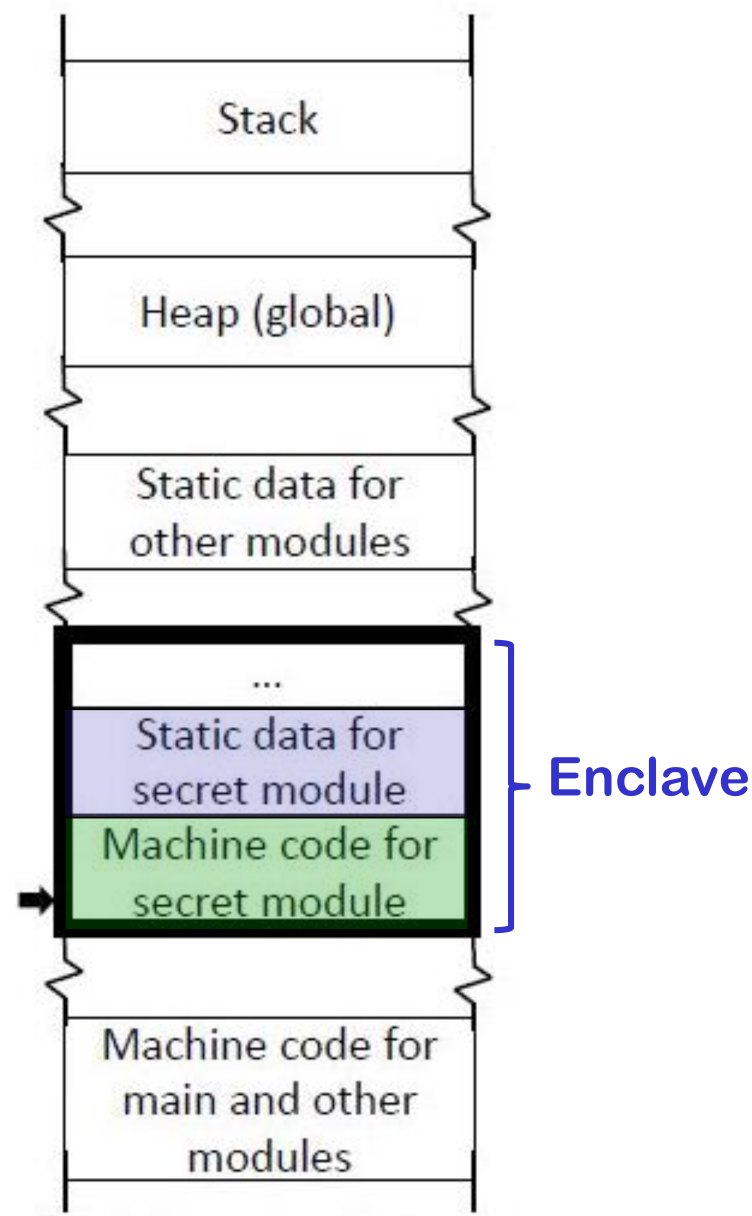
secret.c

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

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

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```



Isolating security-sensitive code with secure enclaves

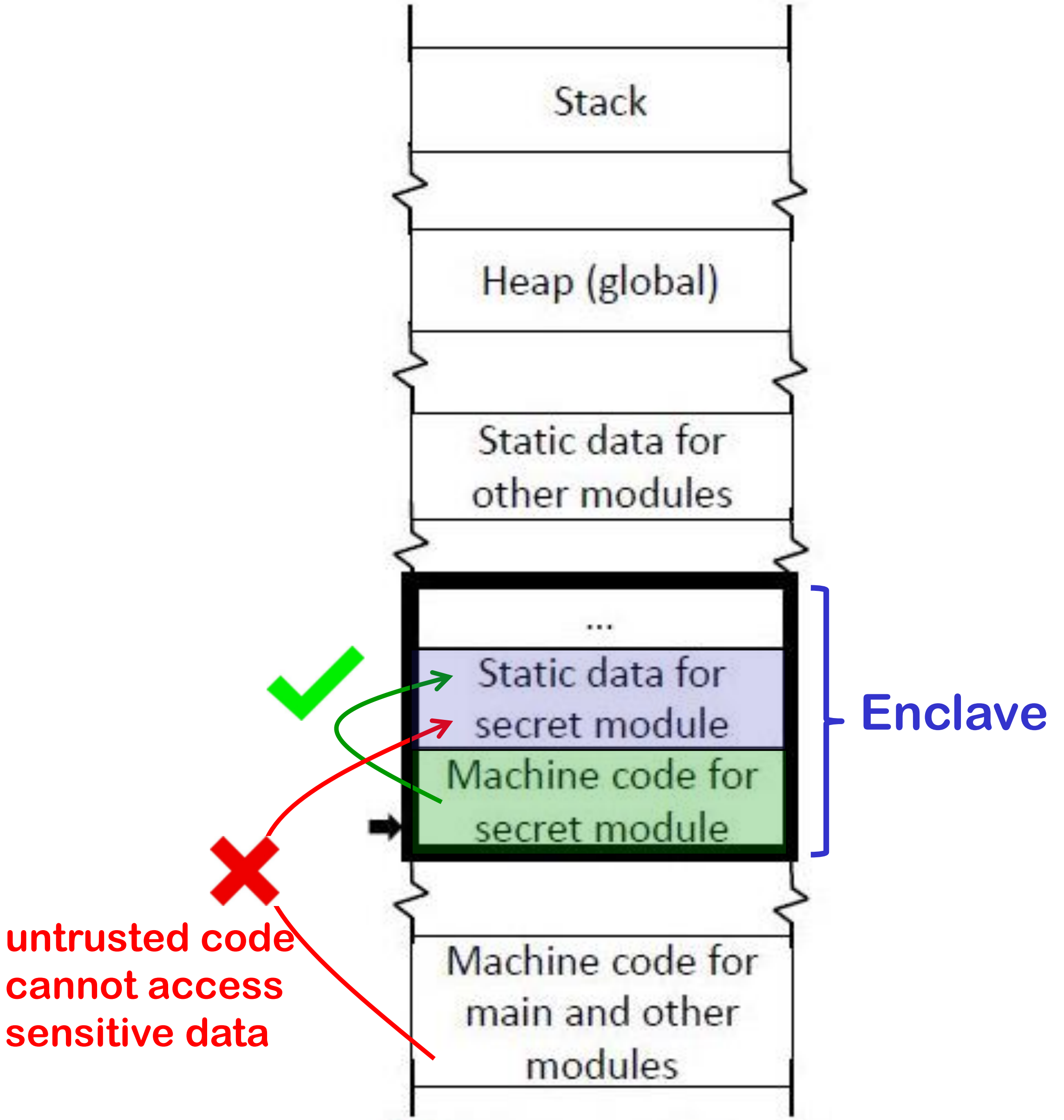
secret.c

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

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

main.c

```
# include "secret.h"
… // other modules
void main () {
…
}
```



Isolating security-sensitive code with secure enclaves

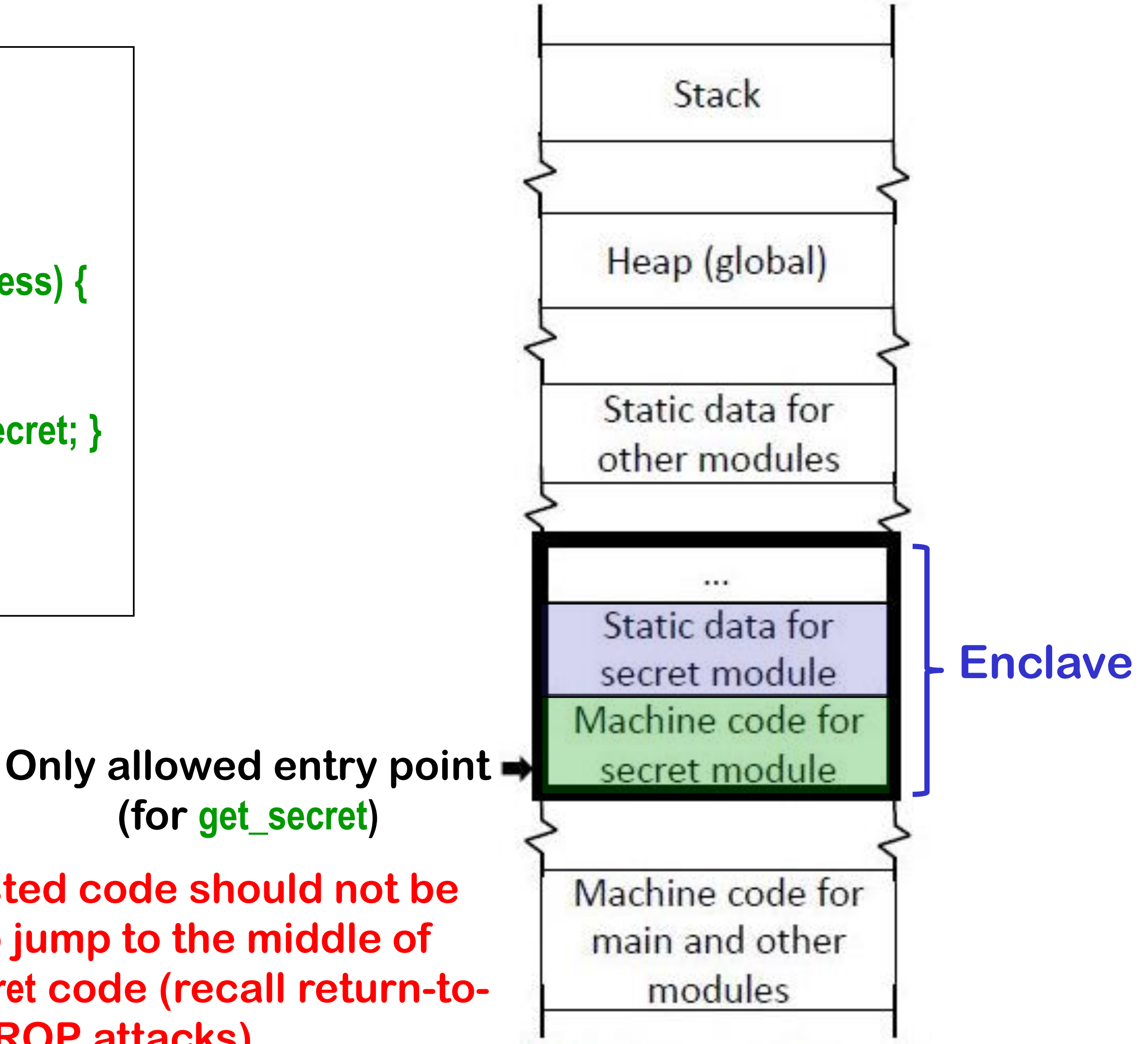
secret.c

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

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

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```



Untrusted code should not be able to jump to the middle of get_secret code (recall return-to-libc & ROP attacks)

Secure enclaves

- Enclaves isolates part of the code together with its data
  - Code outside the enclave cannot access the enclave's data
  - Code outside the enclave can only jump to valid entry points for code inside 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 & Java VM (Virtual Machine) are not in the TCB
  - Also some protection against physical attacks is possible

Enclaves using Intel SGX

Intel SGX provides hardware support for enclaves

- protecting confidentiality & integrity of enclave's code & data
- providing a form of Trusted Execution Enviroment (TEE)

This not only protects the enclave from the rest of the program, but also from the underlying Operating System!

- Hence example use cases include
  - Running your code on cloud service you don't fully trust: cloud provider cannot read your data or reverse-engineer your code
  - DRM (Digital Rights Management): decrypting video content on user's device without user getting access to keys
- Some concerns about Intel's business model & level of control: will only code signed by Intel be allowed to run in enclaves?

Execution-aware memory protection

A more light-weight approach to get secure enclaves

- access control based on the value of the program counter, so that some memory region can only be accessed by a specific part of the program code
- This provides similar encapsulation boundary inside process as SGX
  - Eg. crypto keys can be made only accessible from the module with the encryption code
  - The possible impact of an buffer overflow attack is the rest of the code is then reduced

[Google, US patent 9395993 B2, July 2016]

[Koeberl et al., TrustLite: A security architecture for tiny embedded devices, *European Conference on Computer Systems*. ACM, 2014]

Spot the defect!

secret.c

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

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

Repeated calls will cause integer underflow of tries_left, given attacker infinite number of tries

main.c

```
# include "secret.h"
… // other modules
void main () {
…
}
```

Moral of the story (this bug):

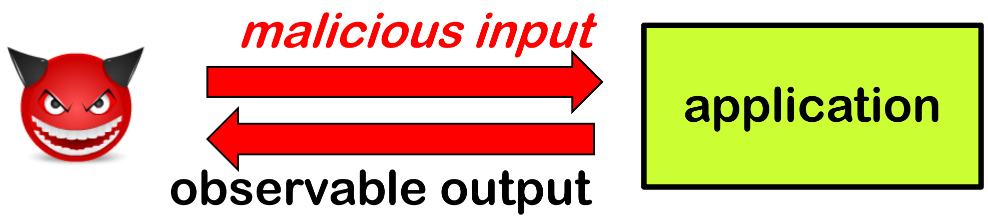
- You can still screw things up
- You have to be very careful writing security-sensitive enclave code

But:

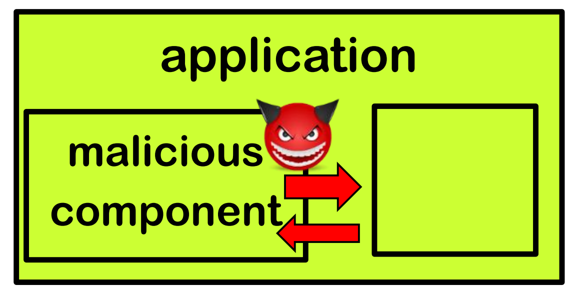
- Screwing up anywhere else in the program can not leak the PIN

Different attacker models for software

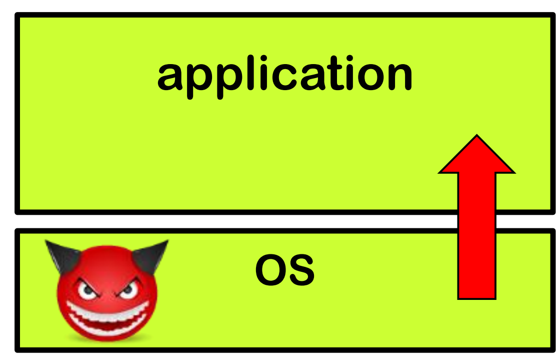
1. **I/O attacker**



2. **Malicious code attacker**
   inside the application
   - Java sandbox & SGX protect against this



3. **Platform level attacker**
   inside the platform, 'under' the application
   - SGX also protects against this



In all cases, the application itself *still* has to ensure it exposes only the right functionality, correctly & securely (eg. with all input validation in place)

Recap

- Conventional OS acccess control

  access control *of* applications and *between* applications

- Language-level sandboxing in safe languages
  - eg Java sandboxing using stackwalking
  - Java VM & OS in the TCB
- Hardware-supported enclaves in unsafe languages
  - eg Intel SGX enclaves
  - underlying OS possibly not in the TCB

  access control *within* an application

Recap

- Language-based sandboxing is a way to do access control within a application: *different access right for different parts of code*
  - This reduces the TCB for some functionality
  - This may allows us to limit code review to small part of the code
  - This allows us to run code from many sources on the same VM and don't trust all of them equally
- Hardware-based sandboxing can also achieve this also for unsafe programming languages
  - Much smaller TCB: OS and VM are no longer in the TCB
  - But less expressive & less flexible
    - No stackwalking or rich set of permissions