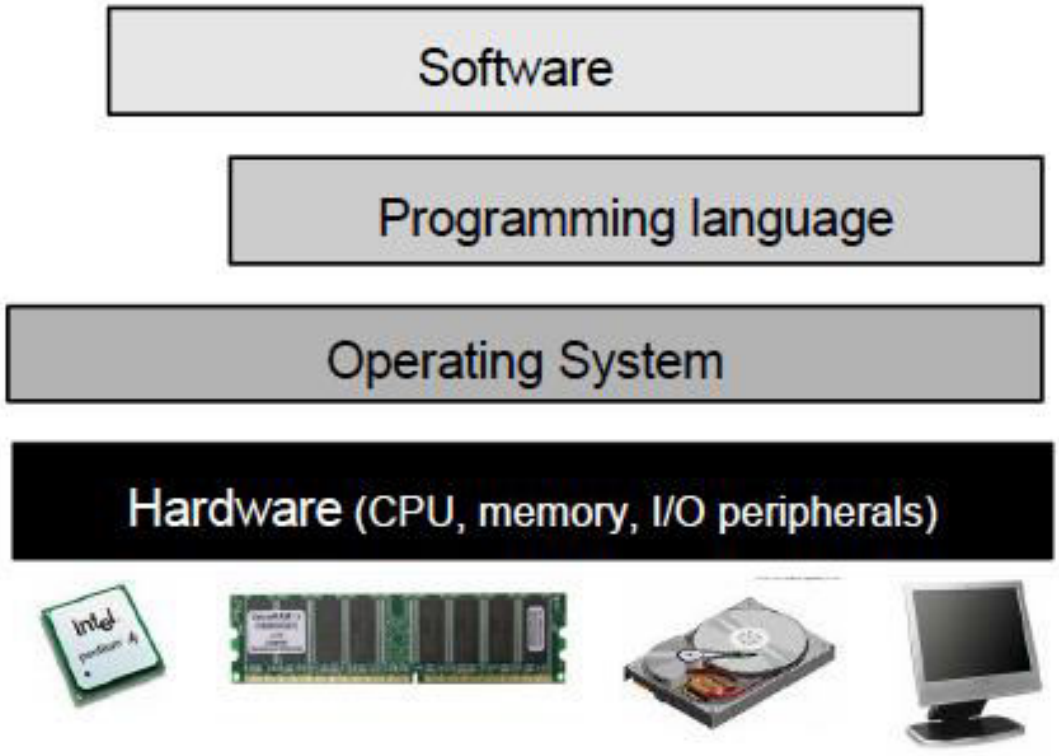
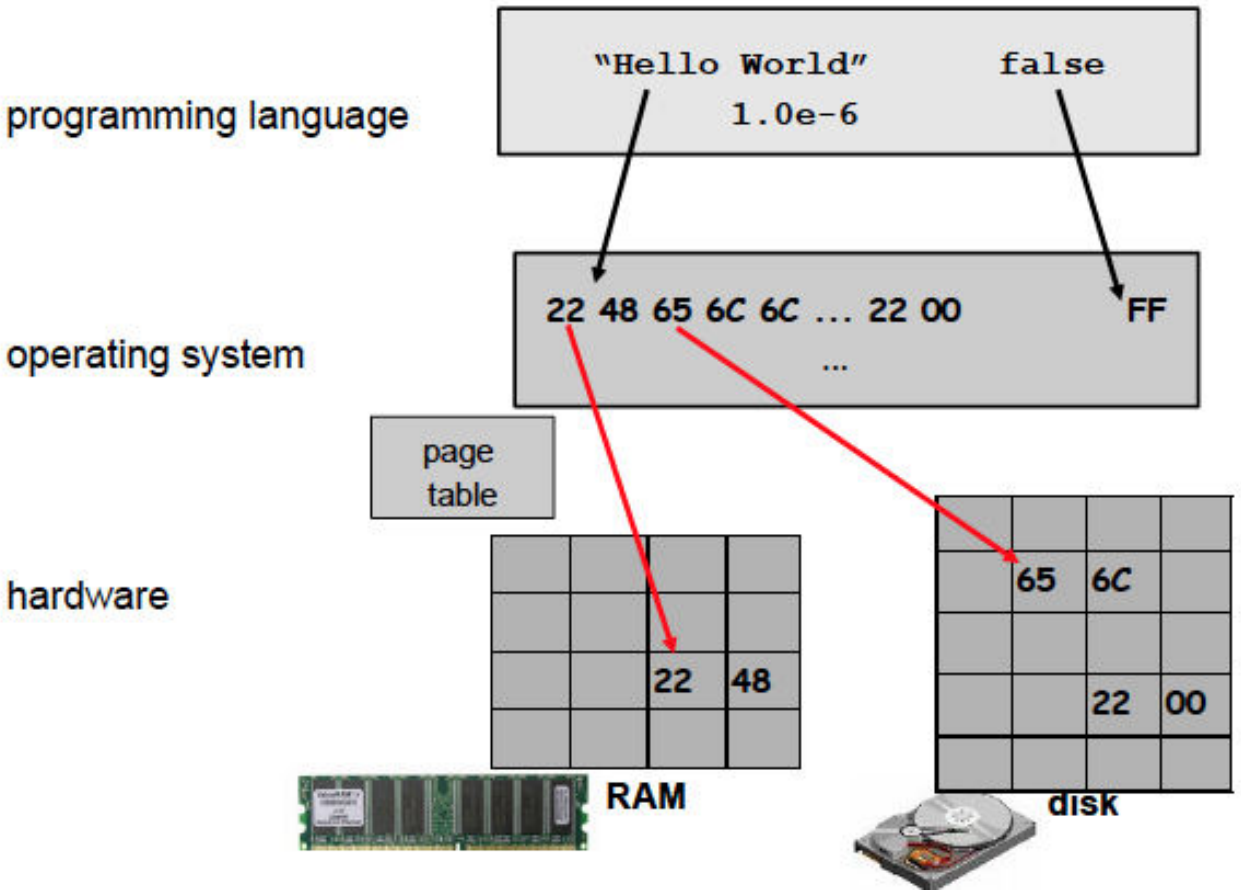
Language/OS Based Security

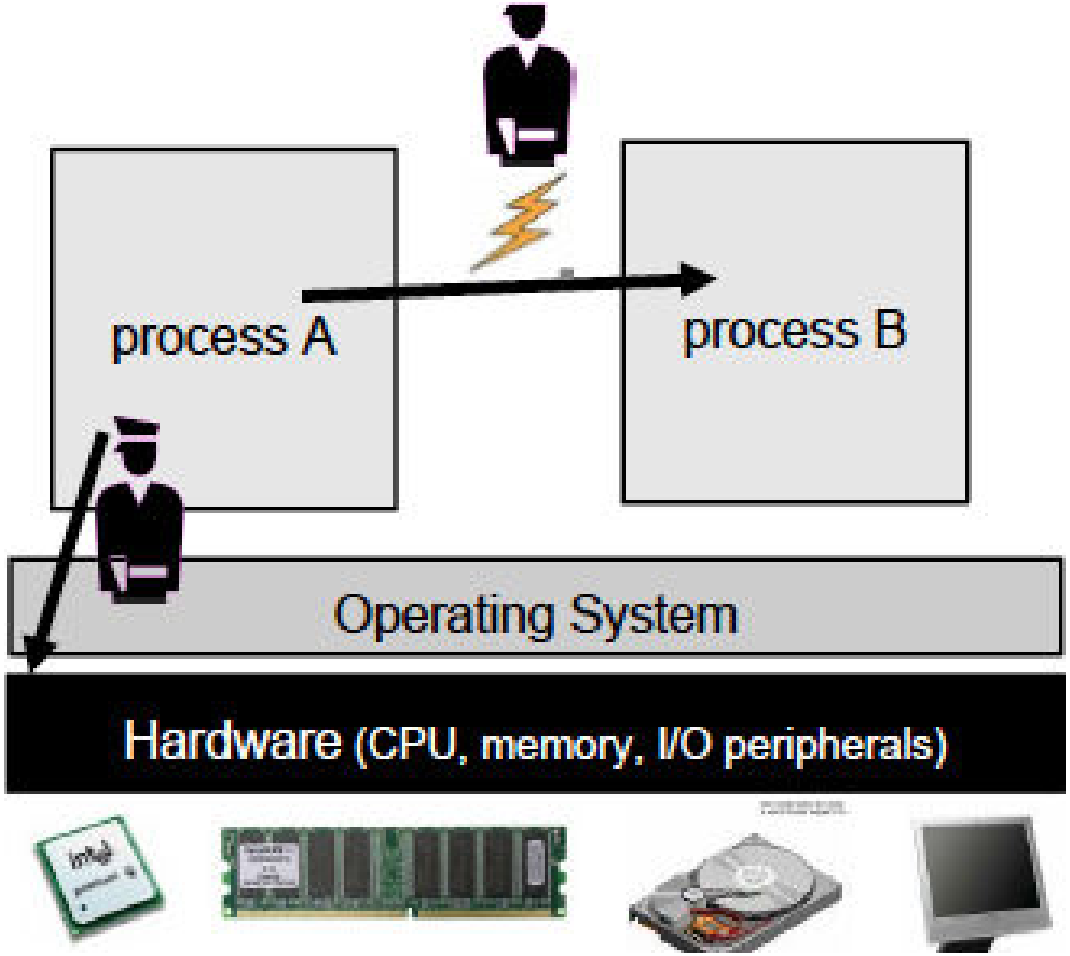
Infrastructure



Security Abstraction Layers



OS Security Control



Abstraction Imperfections

- OS: Each process has exclusive access to its own CPU and memory – Illusion!!
- A process may be able to detect that some of its memory is in fact swapped out to disk, based on the response time of certain operations.
  - Guess a password – early OS (TENEX)
  - Tarball problem in Solaris

Other factors

- Software Bugs
- Complexity
- Safe programming Languages
  - Trusted Abstractions
  - Vs
  - Well defined precise semantics

Safety and Security

- Memory Safety
  - pointer arithmetic;
  - unconstrained casting (e.g., casting a floating-point number to an array);
  - lack of array bounds checks;
  - programmer-controlled de-allocation (as this allows dangling pointers to memory that has been de-allocated).

Stronger Notion of Memory Safety

- Guarantees: programs never read uninitialised memory.
- Weaker definition of memory safety --programs are guaranteed to only access memory locations that they are permitted to access—
  - one can argue whether accessing uninitialised memory should also count as access that is not permitted.
- One consequence of this is that programs cannot observe the old content of physical memory that other processes used previously.
  - Rules out possibility of spying other programs in the usual way
  - Consequence: iprograms behave more deterministically

Type Safety

- Type: Impose restrictions on the programmer that avoid meaningless programs.
  - many safe languages are typed.
- Not all safe languages are typed.
  - LISP is a safe, but untyped language

Type safety

- Type soundness or safety or strong typing: i guarantee that executing well-typed programs can never result in type errors.
  - I.e., in a type-safe language we can rely on the abstractions offered by the type system.
- Other PL Features
  - Visibility (public, private…)
  - Constant values and immutable data structures

Ensuring Type Safety

- Overcome fundamental flaws: establish mathematically the correctness of type systems -- manual, theorem provers …
- What does it means for a type system to be safe: difficulty either informally or formally
  - Operational semantics
  - Representation independence
    - Abstraction provided cannot be broken – see example

Example: Representation Independence

- In a type-safe language it should be impossible to find out if the boolean value true is represented as 1 (or maybe a 32- or 64-bits word ending with a 1) and false as 0, or the other way around.
- If we define two semantics of a type-safe language, one where true is represented as 1 and false as 0 and one where true is represented as 0 and false as 1, we should be able to prove that for any well-typed program the semantics are equivalent regardless of which of these two representations is used.

Safety concerns in low-level languages

- Control-flow safety: the guarantee that programs only transfer control to correct program points. Programs should not jump to some random location.
- Stack safety: the guarantee that the run-time stack is preserved across procedure calls, i.e., that procedures only make changes at the top frame of the stack, and leave lower frames on the stack, of the callers, untouched.

Safety Arithmetic

- No overflow
- No underflow
- exceptions

Thread safety

- Safety concerns the execution of multi-threaded or concurrent programs
- Security vulnerabilities when there is concurrent access to some resource, namely so-called TOCTOU (Time-of-Check, Time-of-Use) errors, also known as `Non-Atomic Check and Use' .

Thread Safety

- A procedure, function, method - or generally, some piece of code - is called thread-safe if it can be safely invoked by multiple threads at the same time.
- In an object-oriented language, a class is called thread-safe if multiple execution threads can simultaneously invoke methods on the same instance of that class.
- one could define a programming language to be thread-safe if its semantics is well-defined in the presence of multi-threading.
  - Even a supposedly safe programming language such as Java is inherently unsafe when it comes to concurrency
  - if a Java program contains data races, it may exhibit very strange behaviour, and one cannot make any guarantees about the semantics whatsoever

Language Based Access Control

- Language Platforms:
  - Modern programming language platforms (or `frameworks'), such as Java or .NET, provide an execution engine that is responsible for executing code, and
  - an API that exposes functionality of the platform itself and of the underlying operating system.

Language Platforms (1)

- The API provides basic building blocks for programs (for example, java.lang.Object), an interface to the underlying operating system (for example System.out.println), and provides some components responsible for the security functionality of the platform (for example the java.lang.ClassLoader and java.lang.SecurityManager).

Language Platforms (2)

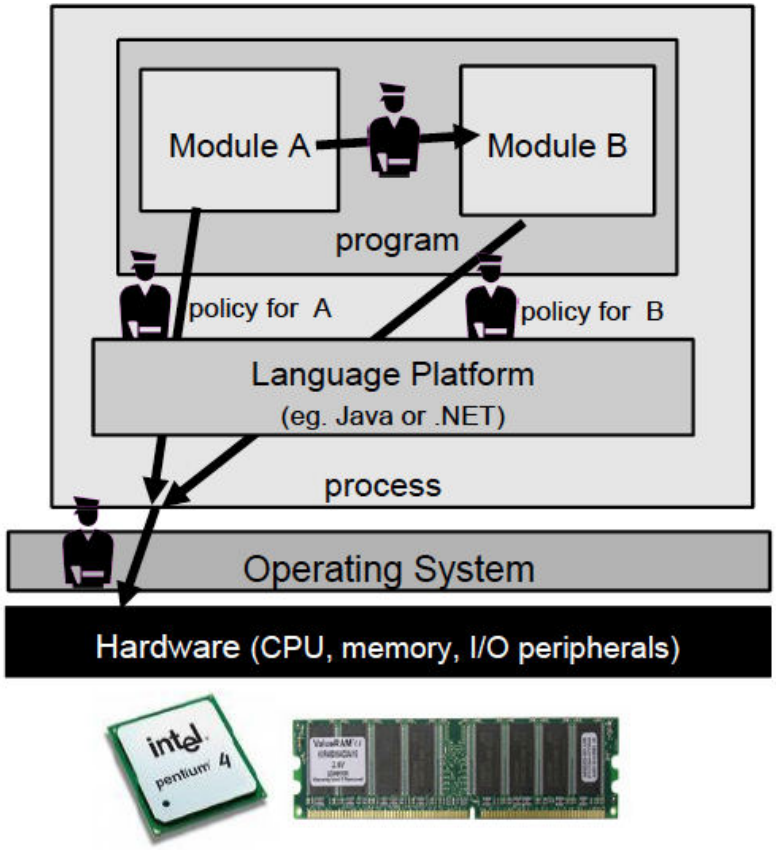
- The Java platform is officially called the Java Runtime Environment, commonly abbreviated to the Java Runtime.
- It consists of the Java Virtual Machine and an implementation of the Java API.
- The .NET framework also consists of a virtual machine, the Common Language Runtime or CLR, and a library, the .NET Framework class library.

Language Platforms (3): Access Control

- The platform also performs access control where it treats different parts of the code (components) differently.

- Separation between components and access control per component.

Security controls at programming platform level

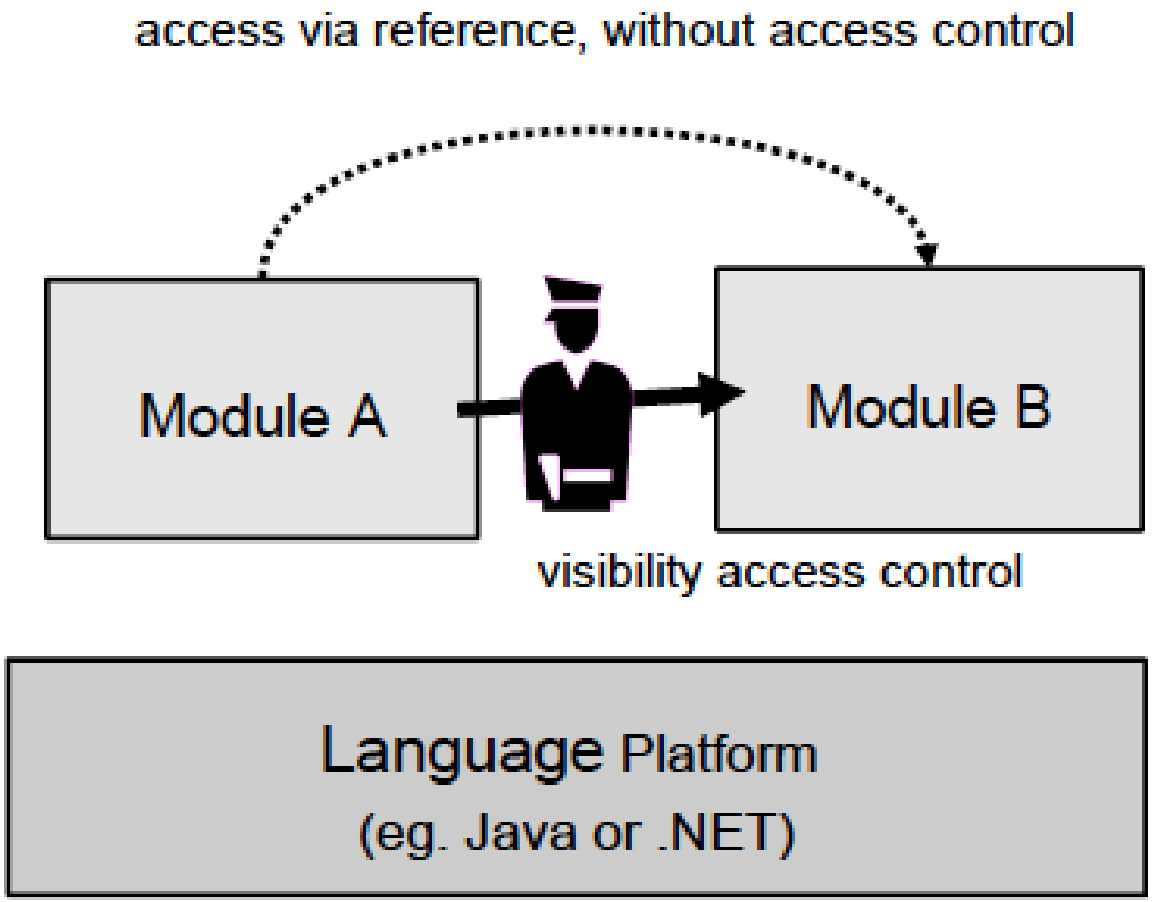


Layers

- Two layers of access contro:
  - One by the language platform
  - One by the operating system -- nice example of Defence in Depth .
- One could in principle get rid of the whole operating system, or `hide' it under the language platform, so that the language platform is the only interface to the hardware for users and application programs.
- Example: JavaCard smartcards.

Why do we need language based access control?

- Modern applications: composed of code from different sources, not all equally trustworthy.
- Example:
  - A Java applet: a Java program downloaded over the web and executed in a web browser;
  - clearly you do not want to execute this code with all the access rights of the user, or even the access rights of the web browser.
  - For example, your web browser can access the hard disk, but you don't want to give some applet on any web page that right


access via reference, without access control
Module A
Module B
visibility access control
Language Platform
(eg. Java or .NET)

Language-Based Access Control and Safety

- Language-based access control is impossible in a programming language that is not memory safe. Without memory safety, there is no way to guarantee separation of memory used by different program components
- Is it possible for one to achieve language-based access control without type safety
- (Figure:) code in component A should be prevented from accessing data belonging to component B.
- Stronger still, it should be prevented from accessing data of the underlying language platform since any data used in enforcing the access control by the language platform could be corrupted

Realizing Language Based Access Control

- Requires policies to specify access rights, a policy language to express, and a mechanism to enforce them.
- Sandboxing – policies to assign permissions to components
  - Code source – directory, or Internet or communicated
- Stack Walking:
  - Method invokation – complicates the interpretation of policies
  - Thread T tries to access resource, access is only permitted if all components on call stack have the right to access it
- One major Issue: Alias Control of mutable objects

Code-based versus process-based access control

- Language-based access control :
  - Based on origin of the code and the associated visibility restrictions

- OS: based on the process identity

Security via Information Flow Control

- IFC - a more expressive category of security properties than traditionally used in access control.
- It is more expressive than access control at the level of the operating system or at the level of the programming
- Information Flow does take into account what you are allowed to do with data that you have read, and where this information is allowed to flow
  - For write access, it not only controls which locations you can write to, but also controls where the value that you write might have come from.

An Example

- APP: First locate the nearest hotel say using Google maps, and then book a room there via the hotel's website with his credit card.
  - Location data will have to be given to Google, to let it find the nearest hotel.
  - The credit card information will have to be given to the hotel to book a room.
- These should be the policy of the APP

Decentralization and Declassification

- Password Example
- Vickery Auction
- EasyChair Conference System
- Confused Deputy Problem

Password Example

- Endorse (guess, new_password);
- if declassify(password= guess) then

  result= success

  else result=failure

Language Based Security in Practice

- Type Safety -- tools
- Source code analysis tools: perform some form of IFC
- Source code scanning tools: analyse code at compile time to look for possible security flaws.
  - simplest versions just do a simple syntactic check to look for dangerous expressions,
  - Deeper analysis: Flow analysis

Language Based Security in Practice

- Deeper Analysis: IFC
  - Focus on integrity rather than confidentiality.
  - Check for tainting: arguments of HTTP POST or GET requests in web applications –
    - tool will try to trace how tainted data is passed through the application and flag a warning if tainted data ends up in dangerous places
    - for example, arguments to SQL commands without passing through input validation routines.
  - Tool has to know which routines should be treated as input validation routines (take tainted data as input and produce untainted data)
- Dynamic tainting

Run Time Monitoring

- Enforcing IFC - Harder
- Flow properties are harder to enforce by run-time monitoring than access control

- Run Time Monitoring through RWFM Model
- Static Certification
  - JIF Status
  - Certifying Python Programs (Abhishek Singh, MTECH Thesis, 2016) via RWFM

TARFILE ISSUE

1. To find out owner & group permissions for the tar file to be created, the password file was consulted (password file tells which group a user belongs to). I.e., the file was read from disk, and stored in RAM and then released.

2. Then the tarfile was created. For this RAM memory was allocated. Because of the way memory allocation worked, this memory included the memory that had just before been used for the password file. And because memory is not wiped on allocation or de-allocation, this memory still contained contents of the password file.

3. Size of a tar file is always a multiple of a fixed block size. Unless the actual contents size is precisely a multiple of this block size, the block at the end will not be completely filled. The remainder of this block still contained data from the password file.

TARFILE