UFR IM²AG

UNIVERSITÉ Grenoble Alpes

UGA Université Grenoble Alpes

UE SCLAM

Sécurité Logicielle

Information Flow, Non Interference, Sandboxing . . .

Master M2 Cybersécurité et Informatique Légale

Academic Year 2021 - 2022

Information levels

Several information levels “coexist” inside an execution platform:

- from different users (including root/admin/...)
- from different processes/threads/applets (e.g., web browser)
- from different input sources (trusted/untrusted, confidential/public)
- etc.

$\Rightarrow$ a lattice, with **lower** and **higher** values from a given informatin level

$\hookrightarrow$ how to avoid **unexpected** interferences between the corresponding information flows ?

Security properties to preserve/enforce

confidentiality:
$\hookrightarrow$ no information leakage from higher data

integrity:
$\hookrightarrow$ no information deletion/rewritting of higher data

Examples:

- sensitive shared plateform level data (e.g., caches, etc.)
- sensitive OS level data (e.g., passwords, resource management, etc.)
- external data, owned by other users/threads
- sensitive internal application data (e.g., crypto keys, nonces, etc.)
- sensitive program execution level memory locations
  (e.g., canaries, return adresses, etc.)

Attacker model

- knows the **code** (executable $\rightarrow$ assembly, source ?)
- **observe** outputs + low variables + part of the execution plateform . . .
- **controls** inputs + low variables + part of the execution plateform . . .
- may observe other **side-channels**

$\Rightarrow$ may direct program execution through controled inputs

- to produce/increase leakage of higher values
- to break integrity (of higher data, of code execution, etc).

Rk: could even imagine interactive/adaptative multi-step attack strategies !

How information may flow ?

- Inside an application, *use/def* dependencies between variables
  - **data-flow** (direct/explicit) through *assigments*
  - **control-flow** (indirect/implicit) through *if*, *while*, ... statements, exceptions, etc.

- Through *side channels*
  - execution time, termination
  - power consumption
  - micro-architecture level (shared) resources: caches, intruction pipelines, branch prediction, etc.
  - others ?

- Between processes/threads
  - sockets, remote calls
  - shared resources (and race conditions !)

Protection against (unwanted) information flows

- Hardware mechanisms (enclaves, etc.)
- OS primitives and access control mechanisms, Virtual Machines
- Language level facilities and libraries (crypto, etc.)
- Coding rules (input sanitization, *constant-time* programming)
- Compiler options (to enforce protection at the executable code level) ?
- some tools ...
  - static analysis: type systems $\rightsquigarrow$ fix-point computations
    $\rightarrow$ not decidable, (over-)-approximation, not complete
  - runtime instrumentation/monitoring techniques (tags, extra checks)
    $\rightarrow$ not sound (may miss existing flows)

But:
protection mechanisms always rely on a TCB **(Trusted Computing Base)**

Non Interference: a general definition (1/2)

$\hookrightarrow$ check information flow partitions inside a program

- more precisely:
  no *influence* of variable/statement of one class to another
  influence = read and/or write and/or execute

- numerous applications in security:
  - confidentiality/integrity
  - taint analysis (e.g., vulnerability exploitability)
  - side-channels through shared resources (execution time, cache, . . .)
  - no use of unitialized variables
  - etc.

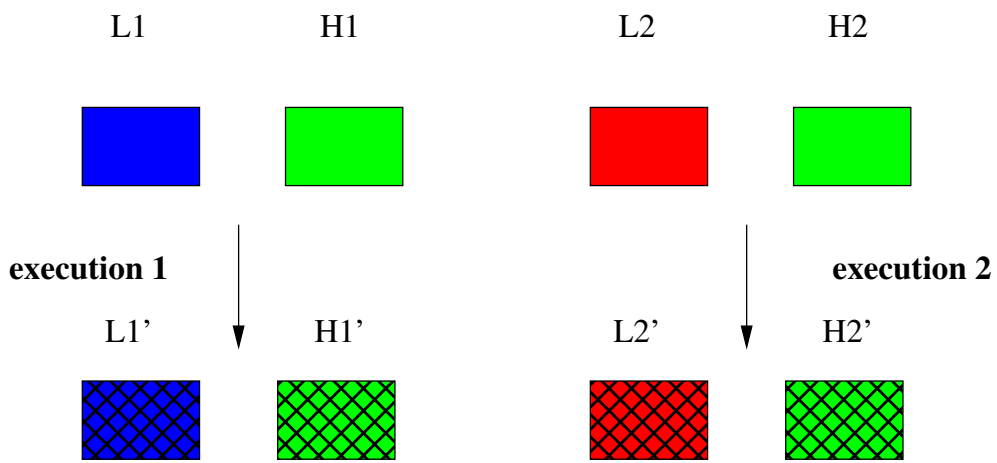
Non Interference: a general definition (2/2)

No influence between data/statement of class L w.r.t. data of class H

Given:

- a variable partition in 2 classes H and L
- memory states M1=(L1, H1) and M2=(L2, H2) s.t. H1 $\equiv$ H2 and L1 $\neq$ L2

Then, executions from M1 and M2 lead to
memory states M'1=(L'1, H'1) and M'2=(L'2, H'2) s.t. H'1 $\equiv$ H'2



Rk: hyper property
(models are **sets** of execution sequences, not single ones . . .)

In the following . . .

1. Intra-application information flow (see E. Poll' slides)

2. Side channels (next slides below)

3. Sandboxing and access control (see E. Poll' slides)

Information leakage through side channels

Are these programs secure?

In both cases,

- some additional information about the secret is leaked by the time or the instruction cache
- by interacting iteratively with the application, the adversary is able to improve his knowledge

```
void compare(int l, int  s){
  if (s<l)
      {write_log(``too large'');}     // 1 sec.
  else
      {some_computation();}           // 2 sec.
}
```

- Attacker can binary search on s using l and the leaked output

```
int pwdCheck(char *l, char*  pwd){
  unsigned i;
  for (i=0; i<B_Size; i++)
       if (l[i]!=pwd[i])
            {return 0;}
  return 1;
}
```

- Segment Oracle Attack : Attacker can brute-force individual characters

Side channels

Information leakage through:

- (implicit) shared resources: caches, hidden registers, etc.
- physical observations: time, power consuption, etc.

A same cause: the use of **high** variables to control

- (global) memory accesses (e.g, arrays) $\rightsquigarrow$ data cache attacks
- execution control flow $\rightsquigarrow$ instruction cache or branch prediction attacks
- time-dependent (assembly level) instructions

constant-time[1] programming:
a set of coding rules to protect against such attacks . . .

See for instance:

- A beginner's guide to constant-time cryptography
- (Intel) Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations
- Some Cryptocoding rules

[1] not related to time complexity !

Some research directions regarding side channels

- **Quantifying** the information leakage
  leaking always one single (same !) bit of a crypto key is less critical than leaking once the whole key . . .

- **Quantifying** the "control level" of an attacker
  how much can she/he influence the execution, at which *cost* ?

- Consider **regular outputs** when computing the leakage
  e.g., password checking may (raisonnably !) delivers at least a boolean output . . .

- Improve **automatic detection** of side channels (hyper-property checking)

- Automatic **code transformation** to *constant time* mode, dedicated programming languages, etc.