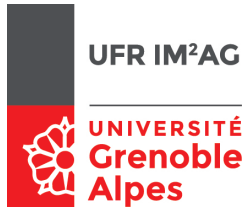


Software security, secure programming

Non Interference: a short summary

Master M2 Cybersecurity & MoSiG

Academic Year 2020 - 2021

Information-Flow

$\hookrightarrow$ retrieve how information “flows” inside a program

- more precisely:
  - *use/def* dependencies between variables
  - 2 kinds of flows:
    - **data-flow** (direct/explicit) through *assigments*
    - **control-flow** (indirect/implicit) through *if*, *while*, . . . statements
- classical code analysis technique:
  compilation/optimization, verification
- in practice
  - 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)

Non Interference

$\hookrightarrow$ 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 (e.g., isolation, enclaves)
  - taint analysis (e.g., vulnerability exploitability)
  - side-channels through shared resources (execution time, cache, . . .)
  - no use of unitialized variables
  - etc.
- in practice:
  - dedicated & refined information-flow analysis techniques
    (static and/or dynamic)
    $\Rightarrow$ # **tools** available . . .
  - $\exists$ secure coding patterns
    ex: **constant-time programming paradigm** [1] for timing/cache attacks

[1] see for instance `https://www.chosenplaintext.ca/articles/beginners-guide-constant-time-cryptography.html`

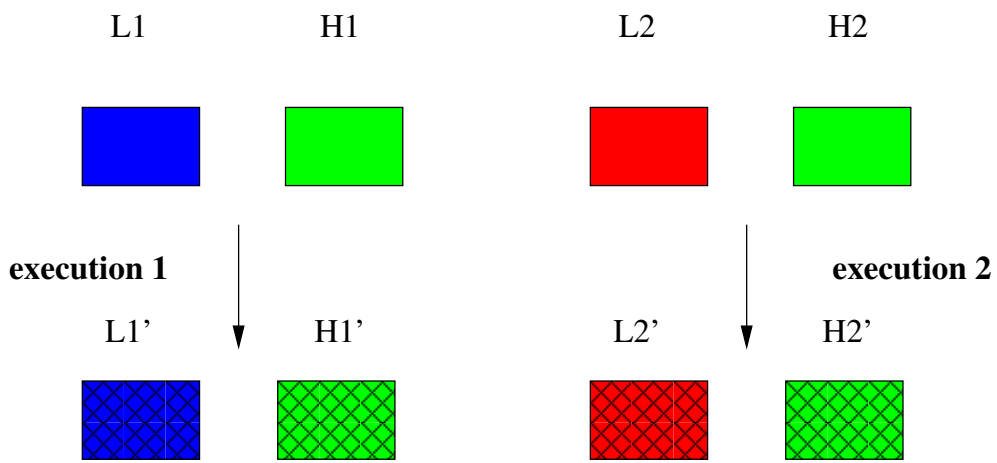
Non Interference: a general definition

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 . . .)

Access Control

A more **coarse-grain** property than non-interference
$\hookrightarrow$ check for information access (only) at the thread level

(not consider how sensitive data is **processed** accross # threads . . .)

(see E. Poll's slides)

As a (temporary) conclusion of part 1

- Mind your programming language
  - type safety, memory safety
  - wysinwyx
- A wide spectrum of **intruder models**
  (from *passive external observer* to *corrupted execution plateform*)
- $\exists$ well-know code vulnerabilities . . .
  but $\exists$ well-know **secure coding patterns** as well !
- Compilers and tools may help a lot !
  $\Rightarrow$ towards certified secure code generation & execution ?