Information security: definition and proof methods

Class 9, Monday 2015-10-26, 5:10–7:00 PM, WWH–312

Patrick Cousot
pcousot@cs.nyu.edu  cs.nyu.edu/~pcousot

Introduction

Information security

- Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction
- see http://en.wikipedia.org/wiki/Information_security

No illicit flow of information should be allowed in a program
- Two dimensions of information security:
  - Integrity: Valuable information should not be damaged by any computation
  - Confidentiality: Valuable information should not be revealed by any computation
- Confidentiality different from:
  - Secrecy: Secret information is not leaked to public listeners
  - Anonymity: A public observer cannot learn the identities of the participating principals even though actions might be known
Confidentiality

- **Confidentiality** is the term used to prevent the disclosure of information to unauthorized individuals or systems.
- For example, a credit card transaction on the Internet requires the credit card number to be transmitted from the buyer to the merchant and from the merchant to a transaction processing network. The system attempts to enforce confidentiality by encrypting the card number during transmission, by limiting the places where it might appear (in databases, log files, backups, printed receipts, and so on), and by restricting access to the places where it is stored. If an unauthorized party obtains the card number in any way, a breach of confidentiality has occurred.
- Confidentiality is necessary (but not sufficient) for maintaining the privacy of the people whose personal information a system holds.

Integrity

- In information security, **integrity** means that data cannot be modified undetectably.
- Example: integrity is violated when a message is actively modified in transit.
- Information security systems typically provide message integrity in addition to data confidentiality.

This is not the same thing as referential integrity in databases, although it can be viewed as a special case of Consistency as understood in the classic ACID model of transaction processing.

Availability/Accessibility

- For any information system to serve its purpose, the information must be available when it is needed.
- This means that the computing systems used to store and process the information, the security controls used to protect it, and the communication channels used to access it must be functioning correctly.
- **High availability** systems aim to remain available at all times, preventing service disruptions due to power outages, hardware failures, and system upgrades.
- Ensuring availability also involves preventing denial-of-service attacks.

Authenticity

- In computing, e-Business, and information security, it is necessary to ensure that the data, transactions, communications or documents (electronic or physical) are genuine.
- It is also important for **authenticity** to validate that both parties involved are who they claim they are.
Non-repudiation

- In law, **non-repudiation** implies one's intention to fulfill their obligations to a contract. It also implies that one party of a transaction cannot deny having received a transaction nor can the other party deny having sent a transaction.
- **Electronic commerce** uses technology such as **digital signatures** and **public key encryption** to establish authenticity and non-repudiation.

Content of this class

- **CIA**: **Confidentiality and Integrity** are handled in this class
- Time permitting, **availability** is handled later in the context of liveness properties (during execution something good ultimately happens in finite time)
- Time permitting, **authenticity** and **non-repudiation** are handled later, in the context of Cryptographic protocols, see [http://en.wikipedia.org/wiki/Cryptographic_protocol](http://en.wikipedia.org/wiki/Cryptographic_protocol)

Low and High Variables

- We classify variables $\forall = \mathbb{L} \times \mathbb{H}$ into **low variables** $x \in \mathbb{L}$ and **high variables** $y \in \mathbb{H}$
- The meaning of low/high is **application dependent** e.g. low = very important/high security/visible/private/untainted/correct/..., high = not much important/low security/hidden/public/tainted/incorrect/...
- **Non-interference** [1,2] informally states that
  
  "Low data of the program are not affected by / do not depend upon high data"

---

**[Counter-]examples**

- `int{H} x;` // high information  
- `int{L} y;` // low information  
- `String{L} z;` // low string  
- `x = y;` // OK  
- `y = x;` // BAD (*)  
- `x = z.size();` // OK  
- `z = Integer.toString(x);` // BAD (***)

(*) Low value of `y` is affected by / depends upon high value of `x`

(***) Low value of `z` is affected by / depends upon high value of `x`

---

**Non-interference**

- **Schematically**

  ![Diagram showing non-interference between high (H) and low (L) data]

- If an execution starts with equal low data then if and when it finishes, the low data are equal (so changing the high data cannot change the low ones)

  - **Confidentiality**: low = public, high = confidential
  - **Integrity**: low = trusted, high = untrusted

---

**Duality in non-interference**

- “Low behavior of the program is not affected by any high security data” [Goguen and Meseguer, 1982]
- Dual interpretation for integrity and confidentiality

\[
(H_1, L_1) \approx_L (H_2, L_2) \implies L_1 = L_2
\]

---

**Example: injection flaw in web-applications**

![Diagram showing web application injection flaw]

```
public void submitQuery(String userName) {
    String query = "SELECT id FROM users WHERE name = "+ userName + "();
    execute(query);
}
```
Formal definition of non-interference

- Let $P$ be a program on variables classified into $\mathbb{V} = L \times H$ with trace semantics $S$

- States have the form $<c, l, h> \in \mathbb{C} \times L \times H$ where
  - $c \in \mathbb{C}$ is the control state
  - $l \in L \rightarrow \mathbb{V}$ is the low memory state
  - $h \in H \rightarrow \mathbb{V}$ is the high memory state

Formal definition of non-interference (cont’d)

- Definition: $P$ with trace semantics $S$ has the non-interference property for $\mathbb{V} = L \times H$, if and only if
  $$\forall t_1, t_2 \in S^+: t_1 \equiv^\downarrow t_2 \Rightarrow t_1 \equiv^\uparrow t_2$$

- Executions with identical low data on entry have identical low data on exit whichever are their respective high data on entry.

- Low data on exit cannot depend on high data on entry

Runtime checks

- A runtime check consists in enlarging the state and adding checks to verify a property during execution.

- An example is buffer overrun checks automatically added to the code by compilers (using appropriate compilation options).

- Runtime checks can only observe one execution up to the point of check.
Non-interference cannot be checked at runtime

- Non-interference must observe all executions two by two and so cannot be checked at runtime.
- All non-interference runtime checks are necessarily incomplete (necessary but not sufficient)

Confidentiality

- Confidentiality: low = public, high = confidential
- Execution traces with identical public inputs have identical public outputs
- It follows that public outputs cannot be influenced by confidential data
- In particular public outputs cannot export any information on private data
- Example: a mailer should only send messages to addressees
**Counter-examples**

- `int{H} x;` // confidential information
  `int{L} y;` // public information
  `String{L} z;` // public string
  
  `x = y;` // OK
  `y = x;` // BAD
  `x = z.size();` // OK
  `z = Integer.toString(x);` // BAD

**Implicit flows**

- `bool{H} b;` // confidential boolean
  `int{L} n;` // public integer
  
  `n := 0;` // OK
  `if (b)`
    `{ n = 1; }` // BAD, reveals value of `b`

- The confidential information in `b` flows to the public information in `n` through control although `b` and `n` never appear in the same basic statement

- Changing `b {H}` will change `n {L}` so confidentiality is broken!

**Restrictions on observables**

- These definitions of confidentiality and non-interference are based on the assumption that data are only observed on entry and exit of execution.

- `bool{H} b;` // confidential boolean
  `int{L} n;` // public integer
  
  `if (b)`
    `{ n := 0; }
  else
    `{ n := 0; }`; // OK (constant final value)

- Low data on exit do not depend on entry high data.

**Formal definition of confidentiality**

- Let `P` be a program on variables classified into `V = Public x Confidential` with trace semantics `S`. `P` has the confidentiality property for `V = Public x Confidential`, if and only if

  \[
  \forall \langle c_0, p_0, q_0 \rangle - a_0 \rightarrow s_1 - a_1 \rightarrow s_2 \ldots <c_m, p_n, q_n> \in S: \\
  \forall \langle c'_0, p'_0, q'_0 \rangle - a'_0 \rightarrow s'_1 - a'_1 \rightarrow s'_2 \ldots <c'_m, p'_m, q'_m> \in S: \\
  p_0 = p'_0 \implies p_n = p'_m
  \]
**Intuition for the formal definition of confidentiality**

- Executions with identical public inputs have identical public outputs
- So changing the confidential inputs does not change the public outputs
- And so, public outputs cannot depend on confidential inputs hence cannot reveal confidential information

**Integrity**

- Integrity: low = trusted, high = untrusted
- Execution traces with identical trusted inputs have identical trusted outputs
- It follows that trusted outputs cannot be influenced by untrusted data
- In particular untrusted inputs cannot import any information on trusted data, in particular cannot modify them
- Example: a mailer should not modify delivered messages

**Formal definition of integrity**

- A program on variables $V$ has the non-interference property for $V = \text{Trusted} \times \text{Untrusted}$, if and only if
  \[
  \forall \langle c_0, t_0, u_0 \rangle \rightarrow a_0 \rightarrow s_1 \rightarrow a_1 \rightarrow s_2 \cdots \langle c_n, t_n, u_n \rangle \in S: \\
  \forall \langle c'_0, t'_0, u'_0 \rangle \rightarrow a'_0 \rightarrow s'_1 \rightarrow a'_1 \rightarrow s'_2 \cdots \langle c'_m, t'_m, u'_m \rangle \in S: \\
  t_0 = t'_0 \Rightarrow t_n = t'_m
  \]
- Executions with identical trusted inputs have identical trusted output whichever are their respective untrusted inputs.
- Trusted outputs cannot depend on untrusted inputs
Example

- A search algorithm should preserve the integrity of the searched array/list/file/etc
- A database query should not change the database content
- A virus violates the computer system integrity

Formal verification of non-interference

\[ \{ P \} C \{ Q \} \]

- The notation should \( \{ P \} C \{ Q \} \) formalize the non-interference property, which we depict as follows:

\[
\begin{array}{c}
H \\
\downarrow \\
L \\
\end{array}
\]

- This can be done in Hoare logic by considering the input \( <H, L> \) as auxiliary mathematical variables:

\[
\{ P(H,L) \land H=H \land L=L \} C \{ Q(H,L,H,L) \}
\]
Example

• \{ H > L \land H=H \land L=L \}  
  \begin{align*}  
  H &= H+1; \\
  L &= L+1; \\
  \{ H > L \land H=H+1 \land L=L+1 \} 
  \end{align*}

• \( P(H,L) \triangleq (H > L) \)  
  \( Q(H,L,H,L) \triangleq (H > L \land H=H+1 \land L=L+1) \)

• The initial values should be fresh variables (not appearing in the program text nor in the assertions)

• \( Q \) is a relation between the input \( H,L \) and output values \( H,L \) of the program variables \( H \) and \( L \).

\{\{ P \}\} C \{\{ Q \}\}

- We write \( P(L,H,L,H) \) for \( P(L,H) \land H=H \land L=L \)
- The picture

\begin{center}  
\begin{tabular}{c|c}  
  & C \\  \hline  
  L & H \\
  \hline  
  L & H \\
\end{tabular}
\end{center}

is now in Hoare logic

\begin{align*}  
\end{align*}

• \( P \) is an hypothesis on the input values and \( Q \) is a relation between input values (under hypothesis \( P \)) and output values

Example (cont’d)

\begin{align*}  
\{ H > L \land H=H \land L=L \}  
  \begin{align*}  
  H &= H+1; \\
  L &= L+1; \\
  \{ H > L \land H=H+1 \land L=L+1 \} 
  \end{align*}

\[ Q(H,L,H,L) \land Q(H',L,H',L') \]  
\begin{align*}  
H &> L \land H=H+1 \land L=L+1 \land H'>L' \land H'=H'+1 \land L'=L+1 \\
\implies [L=L']  
\end{align*}

\begin{align*}  
\end{align*}

i.e. there is no interference since changing the high input (from \( H \) to \( H' \)) does not change the low output (\( L \)).
Soundness

• We must prove that if the premise of the rule [NI] holds then the conclusion of the rule does hold.

• **Proof:** Consider any two traces of the semantics $S$ of $C$:

  $<c_0, L, H> \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots <c_n, L, H>$

  $<c'_0, L', H'> \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} s'_2 \ldots <c'_m, L', H'>$

  such that $L = L' \land P(L, H) \land P(L, H')$.

• By definition of $\{P\} C \{Q\}$, this implies that $Q(L, H) \land Q(L', H')$ does hold.

• From the premise, we derive $L = L'$ proving, by definition of non-interference, that $\{P\} C \{Q\}$. □

Completeness

• We must prove that if the conclusion of the rule [NI] holds then the premise of the rule does hold.

• **Proof:** Assume $\{P\} C \{Q\}$.

• By definition of $\{P\} C \{Q\}$, $\{P\} C \{Q\}$ holds.

• By definition of non-interference, for any two traces of the semantics $S$ of $C$

  $<c_0, L, H> \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots <c_n, L, H>$

  $<c'_0, L', H'> \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} s'_2 \ldots <c'_m, L', H'>$

  we have $L = L'$ so $[P(L, H) \land P(L, H') \land Q(L, H) \land$

A non-interference structural logic

• **Hoare logic** is a sound and complete structural logic to prove invariance properties by induction on the syntax of programs.

• Similarly, we would like to design (if possible) a non-interference structural logic to prove non-interference by induction on the syntax of programs.

• To do so, we instantiate the sound and complete non-interference rule [NI] by induction on the syntax of programs.
Non-interference proof method

- \{P(L,H,L,H)\} C \{Q(L,H,L,H)\} for \{skip\}, \{input\}, \{block\}, \{
- \forall L,H,H': [Q(L,H,H') \land Q(L,H,L,H)] \implies L'=L 


Non-interference proof method (cont’d)

- \forall L,H,H',L': Q(L,H,L',H) \implies L=L' 

Example (cont’d)

- \{H > L \land H=L \land L=L\} 
- H := H+1; 
- \{H > L+1 \land H=L \land L=L\} 
- L := L+1; 
- \{H > L \land H=L+1 \land L=L\} 

\forall L,H: [\exists L': P(L,H,L',H) \land L=f(L',H)] 

Non-interference for while loops

- \{Q\} C \{R\} \text{ and } \{P\} \text{ while } B (\{Q\} C \{R\} \{S\}) \text{ does not imply } \{P\} \text{ while } B (\{Q\} C \{R\} \{S\})

- \text{Counter-example:}
  
  \begin{align*}
  \text{int}\{h\} \text{ h; } & \quad \text{// confidential integer} \\
  \text{int}\{l\} \text{ l; } & \quad \text{// public integer} \\
  l := 0; \\
  \text{while}\ (h > 0) \\
  \{ h := h - 1; \\
  l := l + 1 \} 
  \end{align*}

- \text{The final value of } l \text{ reveals the initial value of } h 
  \text{(although one iteration of the loop body doesn’t)}
Non-interference proof method (cont’d)

- We need to use an instance of the general rule $[\text{NI}]$

\[\{P\} \text{ while } B (\{Q\} C \{R\}) \{S\}\]


\[\{P\} \text{ while } B (\{Q\} C \{R\}) \{S\}\]

- As for the if...then...else... case, the absence of interference in the immediate sub-component (i.e. the loop body) does not imply non-interference for the whole component (i.e. while loop) (because of the test that may leak information)

B is assumed to have no side-effect (does not modify any variable)

Soundness

- The soundness proof consists in showing that each of the rules is an instance of the general non-interference rule $[\text{NI}]$

Incompleteness

- Consider the program

```
if H { L := 0; }
else {L := 1;};
/* (H \land L=0) \lor (\neg H \land L=1) (**) */
L = -1;
/* L = -1 */
```

- There is a violation of non-interference at (**) so the non-interference proof rules do not apply, although there is no violation of non-interference on program exit

- Contrary to invariance, non-interference cannot always be proved by induction on the program syntax

Formal verification of confidentiality

- Instanciate the non-interference rules to

Confidentiality: low = public, high = confidential

Formal verification of integrity

- Instanciate the non-interference rules to

Integrity: low = trusted, high = untrusted
Explicit flows and side channels

- In our definition of non-interference, it is assumed that (malevolent) observers can only access the initial and final values of variables.
- Of course, more information can be obtained by observing so-called side channels such as memory content, execution times, power consumption, electromagnetic fields of computer displays, etc.
- For the formal proofs to be correct, all information that can be leaked has to be cumulated in the final value of variables.

Example of side channel

- By observing the execution time of the following program:

```java
var l, h
if h = 1 then
    (* do some time-consuming work *)
    l := 0
```

it is possible to know whether the secret variable `h` is 1 or not.
- The can be taken into account by a public implicit auxiliary variable counting the numbers of program steps executed.

Termination

- By definition a program **terminates** if and only if its semantics has no infinite execution trace (so all executions are finite).
- This does **not** mean that execution time is **bounded** (i.e. the traces of the semantics have a finite bound on their lengths).
Termination proof method (cont’d)

• To prove that a while loop does terminate, we show that a so-called integer-valued variant function of the program variables $\nu \in \mathbb{V} \rightarrow \mathbb{Z}$:

- Strictly decreases at each loop iteration
- Remains positive within the loop

Example

Let $x$ and $x'$ be the integer values of variable $x$ at some iteration and at the next one. So $x \geq 2$.

If $x$ is odd $\nu(x) = x+1$ and $x'=x+1$ is even so $\nu(x') = x'-1 = (x+1)-1 = x$ satisfies $\nu(x') < \nu(x)$.

If $x$ is even $x=2k$ then $\nu(x) = x-1$ and $x'=x/2$ so $U(x') = x'-1$ or $\nu(x') = x'+1$, so in both cases $\nu(x') \leq x'+1 = x/2+1 < x-1 = \nu(x)$ since $x/2-1 < k-1 < x-1$ if $x=2k \geq 2$ so $k > 0$.

Example

```
int x;
if (x >= 1) {
    while (x <> 1) {
        if even(x)
            x = x / 2
        else
            x = x + 1;
    }
}
```

$\nu(x) = ???$

```
int x;
if (x >= 1) {
    while (x <> 1) {
        if even(x)
            x = x / 2
        else
            x = x + 1;
    }
}
```

$\nu(x) = \text{if } (x \leq 0 ) \text{ then } 0$

else $x-1$
Soundness of the termination proof method

- Let \( C := \text{while } B \lbrace C' \rbrace \) be a while loop and \( u \in V \mapsto \mathbb{Z} \) be a variant function.
- Assume, by reductio ad absurdum, that there exists an initial value \( X_0 \) of the variables \( V \) for which the loop does not terminate.
- Let \( X_i \) be the value of the variables \( V \) after the \( i \)-th loop iteration. We have \( u(X_0) > u(X_1) > \ldots > u(X_i) > u(X_{i+1}) > \ldots \) since the value \( u \) of the program variables strictly decreases at each loop iteration.
- Moreover \( \forall i \in \mathbb{N}: u(X_i) \geq 0 \) since the value \( u \) of the program variables remains positive within the loop.
- This is in contradiction with the fact that \( \mathbb{N} \) ordered by \( \leq \) is well-founded (i.e. has no infinite strictly decreasing chain). □

Natural numbers are not sufficient

- In general we need a larger well-founded set than \( \langle \mathbb{N}, \geq \rangle \) (for unbounded non-determinism)
- Counter example:
  
  ```c
  while (x<>0) {
    if (x > 0) {
      x := -abs(random())
    } else {
      x := x+1
    }
  }
  ```

  Let \( u(x) = \text{if } x>0 \text{ then } +\infty \text{ else } -x \) where \( u \in \mathbb{Z} \mapsto \mathbb{N} \cup \{+\infty\} \) in the loop and \( +\infty > \ldots > n+1 > n > \ldots > 2 > 1 > 0 \) is well-founded.

Completeness (cont’d)

- Let \( P \) be a program which always terminates. Its semantics \( S \) has only finite traces.
- We can define the transition relation \( \tau \) between a state and its possible successors:
  
  \[
  \tau := \{ <s,s'> \mid \exists s_0 - a_0 \rightarrow \ldots s_i - a_i \rightarrow s' = s_{i+1} \ldots s_n \in S \}
  \]
- The relation \( \tau \) is well-founded, i.e., by definition, there is no infinite sequence \( s_0, \ldots, s_n, \ldots \) such that \( \tau(s_0, s_1) \land \tau(s_1, s_2) \land \ldots \land \tau(s_n, s_{n+1}) \land \ldots \)

Completeness (cont’d)

- Proof: Assume \( \tau \) not to be well-founded. Then, we have infinitely many traces \( t_0, t_1, t_2, \ldots \) of \( S \) of the form
  
  \[
  \tau(s_0, s_1) \tau(s_1, s_2) \ldots \tau(s_n, s_{n+1})
  \]
  
  \[
  s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_2} s_2 \ldots s_n \xrightarrow{a_n} s_{n+1} \ldots
  \]
  
  - The program has finitely many actions \( a_0, a_2, \ldots, a_n, \ldots \) so someone, say \( a_i \), must repeat infinitely often. Since the program text is finite, that action \( a_i \) must be within a loop.
  
  - It follows that there is an infinite execution of this loop \( s_i \rightarrow a_i \rightarrow s_{i+1} \ldots s'_i \rightarrow a_i \rightarrow s'_{i+1} \ldots s''_i \rightarrow a_i \rightarrow s''_{i+1} \ldots \) which never terminates, a contradiction. □
Completeness (cont’d)

- We can now define the function
  - \( \nu(s) = 0 \) when \( \forall s': \neg \tau(s, s') \) (so \( s \) can only appear at the end of traces of \( S \), if ever).
  - \( \nu(s) = \sup\{\nu(s') \mid \tau(s, s')\} + 1 \) (for states \( s \) appearing within a trace, not at the end).
- The function \( \nu \) is well defined (on ordinals).

Proof

This follows from the fact that \( \nu \) is defined by induction on the well-founded relation \( \tau \) (see appendix where this mathematical theorem is proved).

- \( \nu \) is a variant function

Proof: By definition, \( \nu \) is ordinal-valued and decreases along each step of traces since \( s_0 \rightarrow a_0 \rightarrow \ldots s = s_i \rightarrow a_i \rightarrow s' = s_{i+1} \rightarrow \ldots s_n \in S \) implies \( \tau(s, s') \) so \( \nu(s) > \nu(s') \).

Accessibility (optional)

System is responsive to requests

OS attacks: attempts to destroy availability (perhaps by cutting off network access)
Fault tolerance: system can recover from faults (failures), remain available, reliable
Benign faults: not directed by an adversary (usual province of fault tolerance work)
Malicious or Byzantine faults: adversary can choose time and nature of fault
  - Byzantine faults are attempted security violations
  - Usually limited by not knowing some secret keys (password)
Liveness properties

“Something good eventually happens”
A typical example is:

Given any trace $s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_2 \rightarrow s_2 \ldots \ s_n \rightarrow a_n \rightarrow s_{n+1} \ldots$ such that $P(s_0) \land P(s_1) \land \ldots \land P(s_i)$ holds, there exists $n > i$ such that $Q(s_n)$ holds.

Examples:
• When receiving a message the email server will eventually send it (so the email server will not stop running)
• Violated by denial of service attacks
• Can’t be enforced purely at run time

Proving liveness

• Prove that given any trace $s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_2 \rightarrow s_2 \ldots \ s_n \rightarrow a_n \rightarrow s_{n+1} \ldots$ such that $P(s_0) \land P(s_1) \land \ldots \land P(s_i)$ holds, there exists $n > i$ such that $Q(s_n)$ holds.
• First, find an invariant $I(s)$ satisfied by reachable states $s$ satisfying $P(s)$
• Second, consider all traces $T$ starting in states $I(s)$ and finishing in states $s'$ satisfying $Q(s')$ (instead of empty control for usual termination)
• Prove the traces $T$ finite (using a variant function)

Information security enforcement

• We have defined various forms of information security (CIA, etc).
• We have given proof methods for proving that programs respect information security specifications (i.e. for CI on one hand [non-interference] and A on the other end [termination and liveness proofs])
• We would like to have means of enforcing programs to respect information security specifications (by program design instead of proving it a posteriori)
• This is the objective of security policies
Computer security policies

- A computer security policy is a set of rules that programs (hardware and software) should satisfy for a computer system to be secure according to some definition of security (e.g. CIA), see http://en.wikipedia.org/wiki/Computer_security_policy

- Examples (to be studied in forthcoming classes):

Enforcing computer security policies

- The computer security rules can sometimes be checked at runtime (see class on reference monitors)
- Some rules cannot be checked at runtime so have to be checking statically, before execution:
  - Can be proved to be satisfied manually (error-prone)
  - Better be checked statically e.g. by automatic flow analysis, type systems, etc., which is the subject of next classes on permission-based access control.

Saltzer and Schroeder computer security principles

<table>
<thead>
<tr>
<th>Principle</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Economy of mechanism</td>
<td>The system should be as simple as possible.</td>
</tr>
<tr>
<td>2 Fail-safe defaults</td>
<td>The default is denial of access.</td>
</tr>
<tr>
<td>3 Complete mediation</td>
<td>Every access decision must be checked.</td>
</tr>
<tr>
<td>4 Open design</td>
<td>The design must be open to review.</td>
</tr>
<tr>
<td>5 Separation of privilege</td>
<td>Sensitive tasks should not be completed by a single individual.</td>
</tr>
<tr>
<td>6 Least privilege</td>
<td>Users should not possess extraneous privileges.</td>
</tr>
<tr>
<td>7 Least common mechanism</td>
<td>The fewer the number of users sharing a mechanism, the less problematic a user damaging the mechanism will be.</td>
</tr>
<tr>
<td>8 Psychological acceptability</td>
<td>The security interface must be easy to use, or it will not be used correctly.</td>
</tr>
</tbody>
</table>


The least privilege principle

- In a computing environment, every module (such as a process, a user, or a program) must be able to access only such information and resources that are necessary to its legitimate purpose [Saltzer and Schroeder, 1975]

- Example:
  - Grant a text editor the permission to access the file system
  - Do not grant a text editor the permission to open a socket connection
Problems in enforcing the least privilege principle

- An authorization policy must be neither too permissive nor too restrictive
  - Too permissive:
    - Violation of the Principle of Least Privilege
    - Program exposed to security attacks
  - Too restrictive
    - The policy-enforcement mechanism will generate run-time authorization failures
    - Security problems may arise

The principle of complete mediation

- Every access to any resource must be mediated by an appropriate authorization check [Saltzer and Shroeder, 1975]

Problems in enforcing the principle of complete mediation

- Enforcement is system-specific
  - Different systems have different resources that need to be protected
  - Different systems have different protection mechanisms
- The authorization check for a particular resource must check for authorization appropriately
- Authorization caching can cause violations of the Principle of Complete Mediation

Conclusion
Conclusion

• We have reviewed the basic concepts of language-based computer security: non-interference and liveness to cover CIA (Confidentiality, Integrity and Availability).

• We have introduced proof methods to show that programs satisfy information security specifications.

• More elaborate notions of CIA do exist such as probabilistic or information theory based information leaking.

Probabilistic confidentiality

• $\texttt{bool\{H\} b; // confidential boolean}$
  \[
  \texttt{int\{L\} n; // public integer}
  \]
  \[
  \texttt{seed(0); // OK}
  \]
  \[
  n := \text{random();}
  \]
  \[
  \text{if (b)}
  \]
  \[
  \{ n := \text{random(); } \} // reveals value of b?
  \]

• Depending on the properties of the random number generator, the value of $b$ may be revealed or not.

• With a standard pseudo-random number generator, the sequence of successive draws is known from the seed and so the value of $b$ is perfectly known.

• In general, probabilistic reasoning is required to prove non-interference.

(Facultative) Homework

• Prove that the following program does not terminate
  \[
  i := 1; \text{ while (}i <> 0\text{) } \{ i := i+1; \}
  \]
  when the variable $i$ takes its values in the integers $\mathbb{Z}$.

• Prove that this program does terminate on a computer with modulo arithmetics (where integers are between bounds $\text{min\_int}$ and $\text{max\_int}$ with $\text{min\_int} - 1 = \text{max\_int}$ and $\text{max\_int} + 1 = \text{min\_int}$).

• This shows that mathematical algorithms and computer programs may be different.
Bibliography

Basic papers in language-based security


Basic papers in language-based security


Chris Andreae, James Noble, Shane Markstrom, Todd Millstein. A framework for pluggable type systems. 21st ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA’06), to appear, October 2006.


Basic papers in language-based security


Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualiﬁers. ACM Conference on Programming Language Design and Implementation (PLDI’02), pages 1–12, June 2002.


Basic papers in language-based security


Basic papers in language-based security


Basic papers in language-based security


Basic papers in language-based security


Basic papers in language-based security


The End