SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

51.505 – Foundations of Cybersecurity

Week 3 - Information Flow

Created by **Martin Ochoa** (2017)
Modified by **Jianying Zhou** (2018)

Last updated: 28 Sept 2018

Recap

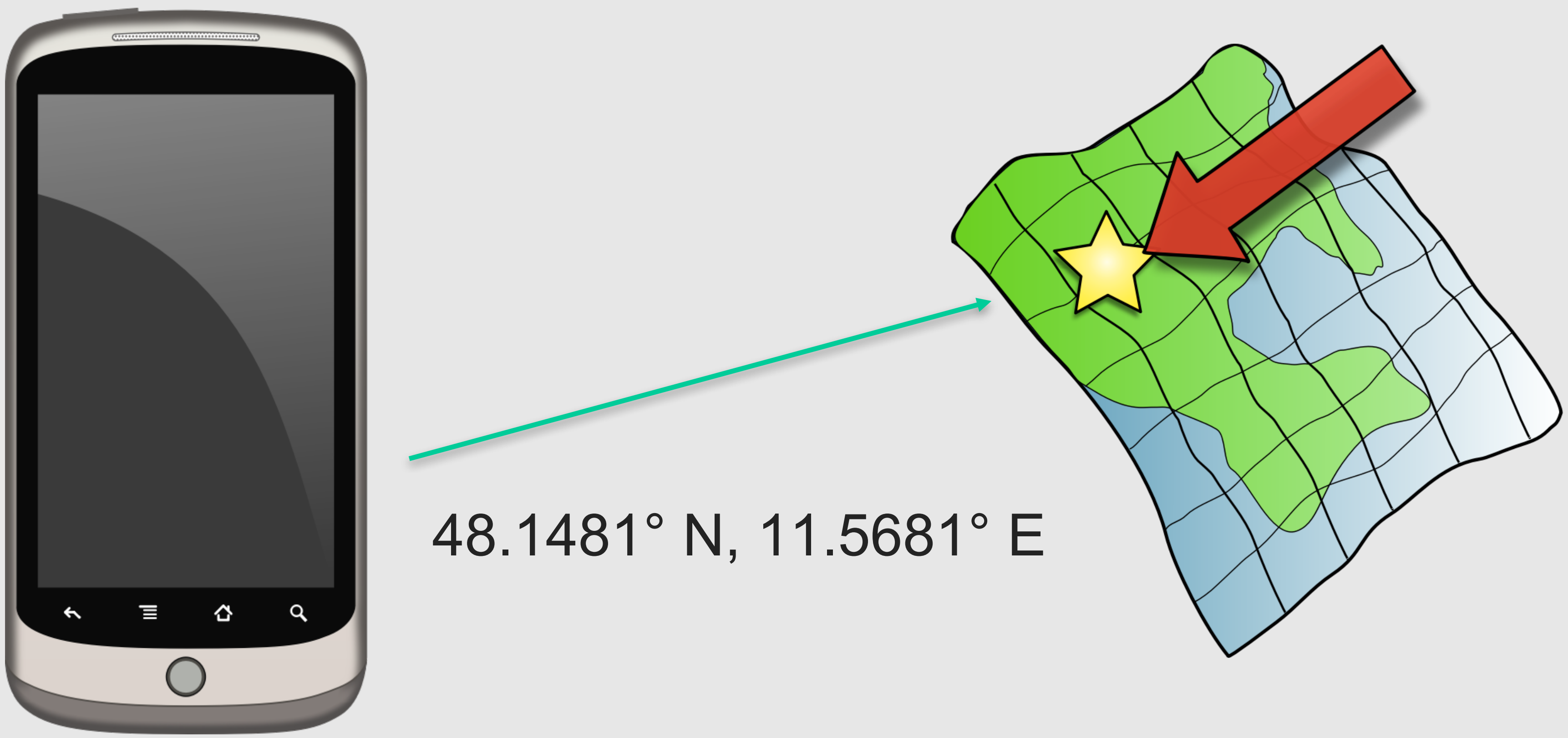
- Questions on last week's exercises?

Confidentiality & Integrity: Non-interference

- Non-interference: an alternative formulation of security policy models.
  - ✓ A strict separation of subjects requires that all channels, not merely those designed to transmit information, must be closed.
- A precise definition attempt by Goguen/Meseguer (1982):
  - ✓ Elegantly capture both confidentiality and integrity notions.
- Thoroughly studied in the literature from different perspectives, playing a role in recent attacks (side-channels).

J. Goguen and J. Meseguer. “Security Policies and Security Models”. IEEE S&P 1982.

Information Flow



- For instance, a Map app wants your coordinates for providing a service.
- Therefore the app needs access to the current location.
- Are the coordinates transmitted to untrusted third parties afterward?

Practical Interpretation of Unwanted Flows

- Exploiting a vulnerability that alters data is an <u>integrity</u> violation.

- An attack that leaks information violates <u>confidentiality</u>.

Interference

- Single system with 2 users:
  - ✓ Each has own virtual machine.
  - ✓ Holly at system high, Lara at system low so they *cannot communicate directly*.
- CPU shared between VMs based on load:
  - ✓ Form a <u>*covert channel*</u> through which Holly, Lara can communicate.

Interference

- Think of it as something used in "indirect” communication.
  - ✓ <u>Covert channel</u>: Holly interferes with the CPU utilization, and Lara detects it.
  - ✓ Example: at a fixed interval, if Holly runs his program, “transmitting” a 1-bit to Lara; If not, “transmitting” a 0-bit to Lara.
  - ✓ Violating *-property.

Model

- System as <u>state machine</u>:
  - ✓ Subjects $S = \{ s_i \}$
  - ✓ States $\Sigma = \{ \sigma_i \}$
  - ✓ Outputs $O = \{ o_i \}$
  - ✓ Commands $Z = \{ z_i \}$
  - ✓ State transition commands $C = S \times Z$

- Inputs:
  - ✓ Encode either as selection of commands or in state transition commands.

Functions

- State transition function T: $C \times \Sigma \to \Sigma$

  ✓ Describe effect of executing command *c* in state σ.

- Output function P: $C \times \Sigma \to O$

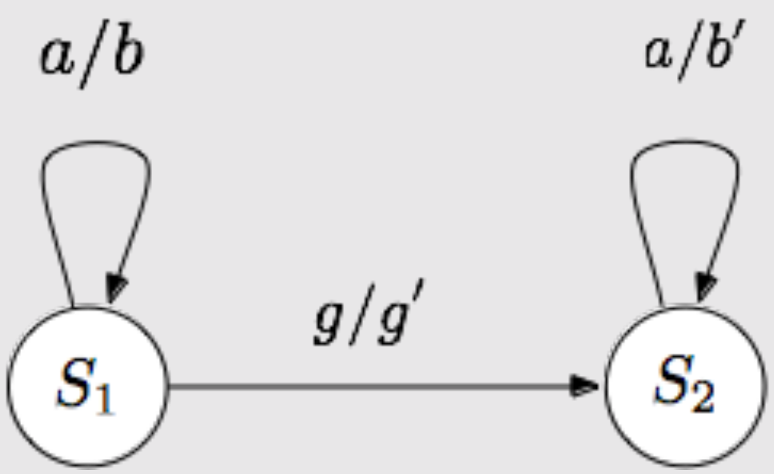
  ✓ Output of machine when executing command *c* in state *s.*

- Initial state is $\sigma_{0.}$

Example Semantics

- <u>Finite state machine</u> (also called Mealy machine)
  - ✓ *Step (T)* takes an input event and changes the state.
  - ✓ *Output (P)* takes an input event and shows the output event associated.

- T(a, S1) = S1, T(g, S1) = ?
- P(a, S1) = b, P(a, S2) = ?



States & Outputs

- *States:* T is inductive in first argument, as

$$T(c_0, \sigma_0) = \sigma_1;\ T(c_{i+1}, \sigma_{i+1}) = T(c_{i+1}, \mathrm{T}(c_i, \sigma_i))$$

- Let C^* be set of possible sequences of commands in C

- $T^*: C^* \times \Sigma \rightarrow \Sigma$ and
$$c_s = c_0, \ldots, c_n \Rightarrow T^*(c_s, \sigma_0) = T(c_n, \ldots, T(c_0, \sigma_0))$$

- Similar definition for *outputs* P and P^*

Example: 2-bit Machine

- 2 bits of state info: *H* (high), *L* (low)
  - ✓ System state is (*H*, *L*) where *H*, *L* are 0, 1
- 2 users: Heidi (high), Lucy (low)
  - ✓ Heidi can read both high and low bit info.
  - ✓ Lucy can only read low bit info.
- 2 commands: *xor0*, *xor1*
  - ✓ Do *xor* on both bits with 0, 1.
  - ✓ Operations affect *both* state bits regardless of whether Heidi or Lucy issues it.

Example: 2-bit Machine

- S = { Heidi, Lucy }
- Σ = { (0,0), (0,1), (1,0), (1,1) }
- C = { *xor0*, *xor1* }

State transition function:

	Input States (H, L)			
	(0,0)	(0,1)	(1,0)	(1,1)
xor0	(0,0)	(0,1)	(1,0)	(1,1)
xor1	(1,1)	(1,0)	(0,1)	(0,0)

Example: 2-bit Machine

- Let $\sigma_0 = (0,1)$
- 3 commands applied:
  - ✓ Heidi applies *xor0*
  - ✓ Lucy applies *xor1*
  - ✓ Heidi applies *xor1*
- c_s = ((Heidi,*xor0*), (Lucy,*xor1*), (Heidi,*xor1*))
- Output P^* = 011001
  - ✓ Shorthand for sequence (0,1) (1,0) (0,1)

Projection

- $T^*(c_s,\sigma_i)$: sequence of state transitions for a system
- $P^*(c_s,\sigma_i)$: corresponding outputs
- $proj(s, c_s, \sigma_i)$: set of outputs in $P^*(c_s,\sigma_i)$ that subject *s* is authorized to see
  - ✓ In same order as they occur in $P^*(c_s,\sigma_i)$
  - ✓ Projection of outputs for *s*
- **Intuition**: Removing outputs that *s* cannot see.

Example

- $\sigma_0 = (0,1)$, c_s = ((Heidi,xor0), (Lucy,xor1), (Heidi,xor1)) = 011001
- *proj*(Heidi, c_s, σ_0) = 011001

✓ Heidi can see both high and low bit outputs.

- *proj*(Lucy, c_s, σ_0) = 101

✓ Lucy cannot see high bit outputs.

Purge

- $G \subseteq S$, G is a group of subjects
- $A \subseteq Z$, A is a set of commands
- $\pi_G(c_s)$: *subsequence of* c_s *with all elements* (s,z), $s \in G$ *being deleted*
- $\pi_A(c_s)$: *subsequence of* c_s *with all elements* (s,z), $z \in A$ *being deleted*
- $\pi_{G,A}(c_s)$: *subsequence of* c_s *with all elements* (s,z), $s \in G$ *and* $z \in A$ *being deleted*

Example

- $\sigma_0 = (0,1)$, $c_s = ((Heidi,xor0), (Lucy,xor1), (Heidi,xor1)) = 011001$
- $\pi_{Lucy}(c_s) = (Heidi,xor0), (Heidi,xor1)$
- $\pi_{Lucy,xor1}(c_s) = (Heidi,xor0), (Heidi,xor1)$
- $\pi_{Lucy,xor0}(c_s) = (Heidi,xor0), (Lucy,xor1), (Heidi,xor1)$
- $\pi_{Heidi}(c_s) = (Lucy,xor1)$
- $\pi_{\mathrm{Heidi},xor1}(c_s) = (\mathrm{Heidi}, xor0), (\mathrm{Lucy}, xor1)$
- $\pi_{\mathrm{Heidi},xor0}(c_s) = \pi_{xor0}(c_s) = (\mathrm{Lucy}, xor1), (\mathrm{Heidi}, xor1)$
- $\pi_{xor1}(c_s) = (\mathrm{Heidi}, xor0)$

Non-interference

- **Intuition:** Set of outputs Lucy can see corresponds to set of inputs she can see, there is no interference.

- Formally: $G, G' \subseteq S$, $G \neq G'$; $A \subseteq Z$; Users in G executing commands in A are non-interfering with users in G' iff for all $c_s \in C^*$, and for all $s \in G'$,

$$proj(s, c_s, \sigma_i) = proj(s, \pi_{G,A}(c_s), \sigma_i)$$

written A,G :| G'

Example

- $\sigma_0 = (0,1)$, $c_s = ((Heidi,xor0), (Lucy,xor1), (Heidi,xor1)) = 011001$
- $G = \{ Heidi \}$, $G' = \{ Lucy \}$, and $A = \emptyset$.
- $\pi_{Heidi,\ A}(c_s) = (Lucy,xor1)$
- $proj(Lucy, \pi_{Heidi,\ A}(c_s), \sigma_0) = proj(Lucy, xor1, \sigma_0) = 0$
- $proj(Lucy, c_s, \sigma_0) = 101 \neq proj(Lucy, \pi_{Heidi,\ A}(c_s), \sigma_0) = 0$
- The statement { Heidi } :| { Lucy } is false.
- **Intuition:** commands issued to change the *H* bit also affect the *L* bit.

Information Flow

- Access controls can constrain the rights of a user, but they cannot constrain the flow of information about a system.
- When a system has a security policy regulating information flow, the system must ensure that the information flows do not violate the constraints of the policy.

Language-based Security

- Consider the following property on a program *P* (which implies non-interference).
- Let *h* and *l* two variables in *P*.
- Example of *P*:

```
l = h
```

- Is it non-interferent?

Language-based Security

- Consider the following property on a program *P* (which implies non-interference).

- Let *h* and *l* two variables in *P.*

- Example of *P:*

  $h = l$

- What about now?

Language-based Security

- Consider the following property on a program *P* (which implies non-interference).

- Let *h* and *l* two variables in *P*.

- Example of *P*:

```
l = 0
if (h == 1)
        l = 1
else
        l = 0
```

- Now?

Implicit Flows

- Why are these kind of properties interesting?
  - ✓ Implicit flows!

- It is in general not enough to track assignments to guarantee confidentiality. For instance:

```
h = 16 digit credit card number
l = 1111111111111111
l = f(h)

f(h):
for i=0 …length_bits(h)
   if h[i]  == 1:
      l[i] = 0
```

Basics

- Bell-LaPadula Model embodies information flow policy.
    - ✓ Given compartments A, B, information can flow from an object in A to a subject in B iff B *dom* A.
- Variables x, y are assigned compartments $\underline{x}, \underline{y}$ as well as values.
    - ✓ If $\underline{x} = A$ and $\underline{y} = B$, and A *dom* B, then $y := x$ is allowed but not $x := y$.

Entropy

- *Uncertainty* of a value, as measured in *bits*.
- Example: X is the value of fair coin toss; X could be heads or tails, so 1 bit of uncertainty.
  - ✓ Therefore entropy of X is $H(X) = 1$
- Formal definition: random variable X, values $x_1, \ldots, x_n$; so $\Sigma_i\, p(X = x_i) = 1$

$$H(X) = -\Sigma_i\, p(X = x_i) \lg p(X = x_i)$$

Conditional Entropy

- X takes values from { $x_1, \ldots, x_n$ }
  - ✓ $\Sigma_i\, p(X{=}x_i) = 1$
- Y takes values from { $y_1, \ldots, y_m$ }
  - ✓ $\Sigma_i\, p(Y{=}y_i) = 1$
- Conditional entropy of X given $Y{=}y_j$ is:
  - ✓ $H(X \mid Y{=}y_j) = -\Sigma_i\, p(X{=}x_i \mid Y{=}y_j)\, lg\, p(X{=}x_i \mid Y{=}y_j)$
- Conditional entropy of X given Y is:
  - ✓ $H(X \mid Y) = -\Sigma_j\, p(Y{=}y_j)\, \Sigma_i\, p(X{=}x_i \mid Y{=}y_j)\, lg\, p(X{=}x_i \mid Y{=}y_j)$

Entropy & Information Flow

- *c* is a sequence of commands taking a system from state *s* to state *t.*
- *x* and *y* are objects in the system; x_s and y_s are values at state *s*.
- The command sequence *c* causes a *flow of information from x to y* if
  - ✓ $H(x_s \mid y_t) < H(x_s \mid y_s)$
- If y_s does not exist in *s*, then $H(x_s \mid y_s) = H(x_s)$.

Example 1

- Command is $x := y + z$; where:
  - ✓ $0 \le y \le 7$, equal probability
  - ✓ $z = 1$ with prob. $1/2$, $z = 2$ or 3 with prob. $1/4$ each
- s is state before command executed; t, after; so
  - ✓ $H(y_s) = H(y_t) = -8(1/8) \lg (1/8) = 3$
  - ✓ $H(z_s) = H(z_t) = -(1/2) \lg (1/2) - 2(1/4) \lg (1/4) = 1.5$
- If you know x_t, y_s can have at most 3 values (about z), so
  - ✓ $H(y_s \mid x_t) = -3(1/3) \lg (1/3) = \lg 3$
- $H(y_s \mid x_t) = \lg 3 < H(y_s) = 3$ → information flows from y to x.

Example 2

- Command is

```
if x = 1 then y := 0 else y := 1;
```

where:

x, y equally likely to be either *0* or *1*

- $H(x_s) = 1$ as x can be either *0* or *1* with equal probability.

- $H(x_s \mid y_t) = 0$ as if $y_t = 1$ then $x_s = 0$ and vice versa.
  - ✓ Thus, $H(x_s \mid y_t) = 0 < H(x_s) = 1$

- So information flows from x to y.

Implicit Flow of Information

- Information flows from x to y without an *explicit* assignment of the form $y := f(x)$.

  ✓ $f(x)$ is an arithmetic expression with variable x.

- Example from previous slide:

  ✓
  ```
  if x = 1  then y := 0
            else y := 1;
  ```

- So must look for implicit flows of information to analyze program.

Notation

- $\underline{x}$ means class of *x.*
  - ✓ In Bell-LaPadula based system, same as "label of security compartment to which *x* belongs".
- $\underline{x} \leq \underline{y}$ means "information can flow from an element in class of *x* to an element in class of *y.*
  - ✓ Or, "information with a label placing it in class *x* can flow into class *y*".

Compiler-based Mechanisms

- Detect unauthorized information flows in a program during compilation.

- Analysis not precise, but secure.
  - ✓ Not precise: A secure path of information flow may be marked as unauthorized (false positive).
  - ✓ Secure: No unauthorized path along which information could flow remains undetected.

- A set of statements is *certified* with respect to information flow policy if flows in that set of statements do not violate that policy.

Example

```
if x = 1 then y := a
              else y := b;
```

- Information flows from x and a to y, or from x and b to y.
- *Certified* only if $\underline{x} \leq \underline{y}$ and $\underline{a} \leq \underline{y}$ and $\underline{b} \leq \underline{y}$
  - ✓ Note flows for *both* branches must be true unless compiler can determine that one branch will *never* be taken.

Array Elements

- Information <u>flowing out</u>:

  ```
  … := a[i]
  ```

  ✓ Values of *i*, *a*[*i*] both affect result, so class is *max{ <u>a[i]</u>, <u>i</u> }.*

- Information <u>flowing in</u>:

  ```
  a[i] := …
  ```

  ✓ Only value of *a[i]* affected, so class is *<u>a[i]</u>.*

Assignment Statements

$x := \underline{y} + z;$

✓ Information flows from y, z to x, so this requires $max\{\ \underline{y}, \underline{z}\ \} \leq \underline{x}$.

More generally:

$\underline{y} := f(x_1, \ldots, x_n)$

✓ The relation $max\{\ \underline{x}_1, \ldots, \underline{x}_n\ \} \leq \underline{y}$ must hold.

Compound Statements

`x := y + z; a := b * c - x;`

- ✓ First statement: *max*$\{ \underline{y}, \underline{z} \} \le \underline{x}$
- ✓ Second statement: *max*$\{ \underline{b}, \underline{c}, \underline{x} \} \le \underline{a}$
- ✓ So, both must hold (i.e., be secure).

More generally:

S_1; … S_n;

- ✓ Each individual S_i must be secure.

Conditional Statements

```
if x + y < z then a := b
              else d := b * c - x;
```

- ✓ $b \leq a$, $\{ b, c, x \} \leq d$
- ✓ The statement executed reveals information about x, y, z (<u>condition</u>), so $max\{ \underline{x}, \underline{y}, \underline{z} \} \leq min\{ \underline{a}, \underline{d} \}$.

More generally:

```
if f(x1, …, xn) then S1 else S2;
```

- ✓ S_1, S_2 must be secure.
- ✓ $max\{ \underline{x}_1, \ldots, \underline{x}_n \} \leq min\{\underline{y} \mid y$ target of assignment in $S_1, S_2 \}$.

Iterative Statements

```
while i < n do
            begin a[i] := b[i];
                  i := i + 1;
            end;
```

- ✓ Same ideas as for “if”, but must terminate.

More generally:

while $f(x_1, \ldots, x_n)$ *do* S;

- ✓ Loop must terminate. Why ?
- ✓ *S* must be secure.
- ✓ $max\{\underline{x}_1, \ldots, \underline{x}_n\} \leq min\{\underline{y} \mid y$ target of assignment in $S\}$.

Infinite Loops

```
y := 0;
while x = 0 do
        (* nothing *);
y := 1;
```

- If $x = 0$ initially, infinite loop.
- If $x = 1$ initially, terminates with y set to *1*.
- No explicit flows, but implicit flow from x to y.
- However, hard to detect whether the loop will terminate at compile time.

Execution-based Mechanisms

- Detect and stop flows of information that violate policy.
  - ✓ Done at *run time*, not compile time.
  - ✓ Before $y = f(x_1, \ldots, x_n)$ is executed, verify that

$$max\{ \underline{x}_1, \ldots, \underline{x}_n \} \leq y$$

- Obvious approach: check explicit flows.

Execution-based Mechanisms

- Problem: Implicit flows complicate checking.
- Assume for security, $\underline{x} \leq \underline{y}$

```
if x = 1 then y := a;
```

- Explicit flow: cause a flow from *x* to *y*. Okay.
- Implicit flow: when $x \neq 1$, $\underline{x}$ = High, $\underline{y}$ = Low, $\underline{a}$ = Low
  - ✓ The implicit flow will not be checked.
  - ✓ The statement may be incorrectly certified.

Key Points

- Non-interference:
  - ✓ Alternative formulation of security policy models.
  - ✓ Assert a strict separation of subjects -- all channels, not merely those designed to transmit information, must be closed.

- Information Flow:
  - ✓ The amount of information flowing (entropy), and the way it flows.
  - ✓ *Explicit* vs *implicit* flows (side-channels)
  - ✓ *Compiler-based mechanism* assesses the flow of information in a program with respect to a given information flow policy.
  - ✓ *Execution-based mechanism* checks flows at run time. Either allow the flow to occur or block it.

Exercises & Reading

- Classwork (Exercise Sheet 3): due on Fri Sept 28, 10:00 PM
- Homework (Exercise Sheet 3): due on Fri Oct 5, 6:59 PM
- Reading: MB [Ch8 (without 8.2.2 - 8.2.4, 8.3 - 8.5), Ch16 (without 16.2, 16.3.2.5, 16.3.4,16.4.1,16.4.2)]

End of Slides for Week 3