Security

Definition

System security is often defined via:
- **Security Properties**: what the system should guarantee (e.g., confidentiality, integrity, availability).
- **Attack Models**: what the system should protect against (e.g., unauthorized access, data breaches).

Security + Formal PL Methods

General PL problems are **pure** and what is optimal is often clear and well-defined.

Security in industry is **constrained** by budget and performance considerations. A best solution **might not be feasible** in practice.

Formal PL techniques can be used to **formalize** *security properties* and *attack models*.

Questions to consider when evaluating system security

- Model the **Target System**
- Model **Adversaries**
- Specify **Security Guarantees**
- Analyze effectiveness of **Approaches**

Techniques

- Model
- Reasoning

Information Flow

History

- Multi-level Security: split data into different levels of sensitivity, e.g., Top Secret, Secret, Confidential, Unclassified.
  - **BLP Model (Confidentiality only) - No Write Down**: a subject at a higher security level cannot write to an object at a lower security level, but a subject at a lower security level can write an object at a higher security level, compromising data integrity.
  - **Biba Model (Integrity only) - No Write Up**: a subject at a lower trust level cannot write an object at a higher trust level.
  - **Lattice Model**: formalized policies $(N, P, SC, \oplus, \rightarrow)$
    1. N: objects
    2. P: subjects
    3. **BLP**:
       - $SC = \{\text{secret}, \text{open}\}$
       - $\rightarrow = \{\text{open} \rightarrow \text{secret}\}$
       - equations: $\text{open} \oplus \text{secret} = \text{secret}$
    4. **Biba**:
       - $SC = \{\text{trusted}, \text{untrusted}\}$
       - $\rightarrow = \{\text{trusted} \rightarrow \text{untrusted}\}$
       - equations: $\text{trusted} \oplus \text{untrusted} = \text{untrusted}$
- **Modern Lattice Model**:
  - L: set of security labels
  - $\sqsubseteq$: a partial order on L specifying allowed information flow

  - TODO Expand ?

From Local to Global

- **Local** properties:
  - **BLP**: low users can't write high files; secrets can't be written to unclassified files
  - **Biba**: low integrity users can't write high integrity files; low integrity files can't be read by high integrity users
- **Global** property: **Non-interference**
  1. *Secrets* can't **interfere** with the **observation** of users who are not allowed to see them.
  2. *Untrusted data* can't **interfere** with the **operations** (**observation**) of trusted data.

In other words, for a system to be secure in the information flow sense, it must ensure that **secret input** does not flow to **public output**.

TODO what does this mean?

Non-interference

Let $P(I_{\text{pub}}, I_{\text{priv}}) = O_{\text{pub}}, O_{\text{priv}}$.
For any two executions of P with the **same public input I_{pub}**, the **public output O_{pub}** must be the **same**, **regardless of** the private input I_{priv}.

NOTE I revised the original expression.

Examples

Password Manager

Components:

- `report`: crash report
- `pwd`: user passwords
- `send`: crash report sending (Network API) function

Bad Example:

From the most naive one:

```
send pwd;
```

... to a complex one:

```
output := "";
for (i = 0; i < pwd.length; i++) {
  c = pwd[i];
  switch (c) {
    case 'a': output += "a"; break;
    case 'b': output += "b"; break;
    case 'c': output += "c"; break;
    // ...
  }
}
send output;
```

Information-flow Witness:

- `pwd` is a secret input, and `output` is a public output.
- Notice how `output` depends on `c` which in term depends on `pwd` on *Line 4-9*.

Strava heatmap around a military base

Strava leaked aggregated data about users' activities, which allowed adversaries to infer sensitive information about military personnel's movements.

Intuition: normally aggregation is a good privacy preserving technique, but in this case the aggregation leaks sensitive location information preserved through aggregation.

Eager Password Manager

```
c := input();
while (c != null) {
  if (c == password[i])
    c = input(); i++;
  else
    return fail;
}
return success;
```

TODO I failed to keep up with how this example is bad in terms of information flow 😭

Side-channel attacks

- Timing

```
F(x) {
  if (secret == 0 || x == 0)
    skip
  else
    complex_operation;
}
```

- Cache timing (meltdown attack):
  1. `secret` is sensitive data
  2. Program `P` tries to access a probe array `array[secret * PAGE_SIZE]`, which gets the page containing `secret` into the cache.
  3. Attacker scans the probe array after `P` runs to find the index with the shortest access time, which reveals the value of `secret` because its page is still in the cache line.

Permission-based access control

Global variables are accessible to every extension in Firefox.

- `FlashGot` reads global variable `files` and has **write permission**.
- `Greasemonkey` reads global variable `$exe` and has **execute permission**.
- `Attacker` writes to `files` and `$exe`. While it **does not have any permission**, it can still **download and execute code**.

Intuition:

1. Global state is bad
2. Permission control is too local to enforce global security properties.

Lessons taken: Information flow sense, permission should be *transitive*.

Information Flow Security

- **Confidentiality**: guard against data leaking to attackers
- **Integrity**: guard against data from attackers flowing to core components

Nondeterministic systems

Noninterference?

Let $P\left(I^{\text{pub}}, I^{\text{priv}}\right) = \Sigma_i\left(O_i^{\text{pub}}, O_i^{\text{priv}}\right)$.

For any two executions of P with the **same public input I_{pub}**, the **public output set of the first execution O_1^{pub}** must be a **subset** of **public output set of the second execution O_2^{pub}**, **regardless of** the private input I_{priv}.

NOTE I revised the original expression.

Problem

TODO

Enforcement measures

1. Type system: int S, int P. Whenever the type rejects the program, it means that the program **might** violate the security property. Note that this is an over-approximation.
2. Runtime monitors: monitor terminates the program in runtime if it violates the security property.