Flow Analysis

Tainted Flow Analysis

- The **root cause** of many attacks is **trusting unvalidated input**
  - Input from the user is **tainted**
  - Various data is used, assuming it is **untainted**
- Examples expecting untainted data
  - source string of `strcpy` (≤ target buffer size)
  - format string of `printf` (contains no format specifiers)
  - form field used in constructed SQL query (contains no SQL commands)

Recall: Format String Attack

- Adversary-controlled format string

```
char *name = fgets(…, network_fd);
printf(name);    // Oops
```

  - Attacker sets name = `“%s%s%s”` to crash program
  - Attacker sets name = `“…%n…”` to write to memory
    - Yields code injection exploits

- These bugs still occur in the wild
  - Too restrictive to forbid non-constant format strings

The problem, in types

- Specify our requirement as a *type qualifier*

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

  - **tainted** = possibly controlled by adversary
  - **untainted** = must not be controlled by adversary

```
tainted char *name = fgets(…,network_fd);
printf(name); // FAIL: tainted ≠ untainted
```

Analysis problem

- **No tainted data flows**: For all possible inputs, prove that tainted data will never be used where untainted data is expected
  - **untainted** annotation: indicates a **trusted sink**
  - **tainted** annotation: an **untrusted source**
  - *no annotation* means: not sure (analysis figures it out)
- A solution requires inferring **flows** in the program
  - What **sources can reach what sinks**
  - If any flows are *illegal*, i.e., whether a **tainted** source *may flow to* an **untainted** sink
- We will aim to develop a *sound* analysis

Legal Flow

```
void f(tainted int);
untainted int a = …;
f(a);
```

`f` accepts **tainted** *or* **untainted** data

untainted ≤ **tainted**

Illegal Flow

```
void g(untainted int);
tainted int b = …;
g(b);
```

`g` accepts *only* **untainted** data

tainted ⋠ **untainted**

Allowed flow as a **lattice**

untainted < **tainted**

Analysis Approach

- Think of **flow analysis** as a kind of **type inference**
  - If no qualifier is present, we must infer it
- Steps:
  - **Create** a **name** for each missing qualifier (e.g., **α**, **β**)
  - For each statement in the program, **generate constraints** (of the form $q_1 \leq q_2$) on possible solutions
    - Statement `x = y` generates constraint $qy \leq qx$ where qy is `y`'s qualifier and qx is `x`'s qualifier
  - **Solve the constraints** to produce solutions for **α**, **β**, etc.
    - A solution is a *substitution* of qualifiers (like **tainted** or **untainted**) for names (like **α** and **β**) such that all of the constraints are legal flows
- If there is **no solution**, we (may) have an **illegal flow**

Example Analysis

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);
```

```
char *name = fgets(…, network_fd);
char *x = name;
printf(x);
```

α

β

$\text{tainted} \leq \alpha$

$\alpha \leq \beta$

$\beta \leq \text{untainted}$

First constraint requires α = tainted

To satisfy the second constraint implies β = tainted

But then the third constraint is illegal: tainted ≤ untainted

Illegal flow!

No possible solution for α and β