Information Flow Control by Typing

Michael Hicks
University of Maryland

Data leaks

- A cause of data leaks in software is **not controlling output of secret data**
  - exposing sensitive information to untrusted parties
    - Login credentials stored in web forms sent over unencrypted connections
    - Not removing sensitive fields from database dumps
  - revealing information that correlates with secrets
    - Error message that indicates the legitimacy of a login ID
- Attackers can use stolen information directly or in service of other attacks

Injection attacks

- The root cause of many software attacks is **trusting unvalidated input**
  - form field used in constructed SQL query
    - should contain no SQL commands
  - source string of `strcpy`
    - size should be ≤ target buffer size
  - format string of `printf`
    - Should contain no format specifiers
- Attacks can cause data corruption and remote code execution

Example: 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 flow 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("%s\n",name); // OK: "%s\n" untainted
printf(name);    // FAIL: tainted ≠ untainted
```

Type Qualifiers for Flow Control

- Types have form ***q v***
  - where ***v*** is a basic type
  - and ***q*** is a flow control policy
- Example: avoiding injection attacks
  - ***q*** = **`tainted`** means data from the untrusted user;
  - ***q*** = **`untainted`** means it is trusted
  - Goal: Ensure tainted data not used as if untainted
- Can avoid data leaks too; will see this later

Extended Formalism for Micro-OCaml

$$
\begin{array}{lll}
\textbf{Exprs} & e ::= & x \mid v \mid e + e \mid \texttt{let } x = e \texttt{ in } e \\
& & \mid \texttt{eq0 } e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& & \mid \texttt{annot}(q, e) \mid \texttt{check}(q, e) \\
\textbf{BVals} & \omega ::= & n \mid \texttt{true} \mid \texttt{false} \\
\textbf{Quals} & q ::= & \texttt{tainted} \mid \texttt{untainted} \\
\textbf{Values} & v ::= & q(\omega) \\
\textbf{BTypes} & \nu ::= & \texttt{bool} \mid \texttt{int} \\
\textbf{Types} & t ::= & q\ \nu
\end{array}
$$

Normal value ω annotated with a qualifier q

Normal type ν annotated with a qualifier q

Encoding Examples

Example in C

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

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

Example in Micro-OCaml

```
let name = annot(tainted,1) in
let x = name in
check(untainted,x)
```

Typing Integers

- Integer constants have type q `int`

$$\frac{}{G \vdash q(n) : q\ \texttt{int}}$$

- Arithmetic expressions also have type q `int`

$$\frac{G \vdash e1 : q\ \texttt{int} \quad G \vdash e2 : q\ \texttt{int}}{G \vdash e1\ \texttt{+}\ e2 : q\ \texttt{int}}$$

- Qualifier q to be the same for both $e1$ and $e2$
  - We will revisit this

Typing Booleans

- Boolean constants have type q `bool`
- Equality checking does too
  - Qualifier inherited from target `int` expression

$$\frac{G \vdash e : q\ \texttt{int}}{G \vdash \texttt{eq0}\ e : q\ \texttt{bool}}$$

- Conditional: `bool` guard, same-typed branches

$$\frac{G \vdash e1 : q\ \texttt{bool} \quad G \vdash e2 : t \quad G \vdash e3 : t}{G \vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

  - Qualifier q on guard unrelated to one on branches
    - We will revisit this decision

Typing Qualifier Operations

- Annotation overrides any existing qualifier
  - Can be used to endorse existing data

$$\frac{G \vdash e : q'\ \nu}{G \vdash \texttt{annot}(q, e) : q\ \nu}$$

- Checking ensures an expression has the expected qualifier
  - E.g., that data definitely untainted

$$\frac{G \vdash e : q\ \nu}{G \vdash \texttt{check}(q, e) : q\ \nu}$$

Example: Rejects unsafe program

```
let name = annot(tainted,1) in
let x = name in
check(untainted,x)
```

```
name : tainted int
x : tainted int
```

Fails!

Quiz 1

- What is the type of the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = annot(tainted,3) in
if x then y else z
```

A. **`untainted int`**

B. **`tainted int`**

C. *type error*

Quiz 1

- What is the type of the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = annot(tainted,3) in
if x then y else z
```

A. `untainted int`

B. `tainted int`

C. ***type error – x is an int, but should be a bool***

Quiz 2

- What is the type of the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = annot(untainted,3) in
if eq0 x then y else z
```

A. **untainted int**

B. **tainted int**

C. *type error*

Quiz 2

- What is the type of the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = annot(untainted,3) in
if eq0 x then y else z
```

A. **untainted int**

B. **tainted int**

C. ***type error – qualifier of y and z must be same***

Quiz 3

- What is the type of **z** in the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = x + y in z
```

A. **untainted int**

B. **tainted int**

C. *type error*

Quiz 3

- What is the type of **z** in the following program?

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = x + y in z
```

A. **untainted int**

B. **tainted int**

C. ***type error – x and y must have same qualifier***

Too Restrictive

```
int puts(tainted char *str);
```

```
untainted char *x = "error";
puts(x);
```

```
let x = annot(untainted,1) in
check(tainted,x)
```

```
x : untainted int
```

Fails!

- Rejecting this program is too harsh
  - If `puts` can accept tainted data, surely it can also accept trustworthy data

Subtyping

- Type ***t1*** is a subtype of ***t2***, written $\vdash$ ***t1*** $\leq$ ***t2***
  - if values of type ***t1*** can be used where those of type ***t2*** are expected

$$\frac{G \vdash e : t1 \quad \vdash t1 \leq t2}{G \vdash e : t2}$$

  - Common idea from OO inheritance: If class T1 extends T2, then T1 can be used wherever T2 can

- Rule can be applied where needed

Qualifiers and subtyping

- Untainted data can be used as if tainted

$$\frac{\vdash q1 \leq q2}{\vdash q1\ \nu\ \leq q2\ \nu} \qquad \frac{}{\vdash q \leq q}$$

$$\frac{}{\vdash \texttt{untainted} \leq \texttt{tainted}}$$

- Qualifier checks allow for sub-qualifiers

$$\frac{G \vdash e : q'\ \nu \quad \vdash q' \leq q}{G \vdash \texttt{check}(q, e) : q'\ \nu}$$

Now Accepted

```
let x = annot(untainted,1) in
check(tainted,x)
```

```
x: untainted int
OK: untainted ≤ tainted
```

```
let x = annot(untainted,1) in
let y = annot(tainted,2) in
let z = x + y in …
```

```
x: untainted int
y: tainted int
z: tainted int
```

Use of subtyping

$$\frac{\vdash \textbf{untainted} \leq \textbf{tainted} \qquad \Gamma \vdash \textbf{x} : \textbf{untainted int}}{\Gamma \vdash \textbf{x} : \textbf{tainted int}} \qquad \Gamma \vdash \textbf{y} : \textbf{tainted int}$$

$$\Gamma \vdash \textbf{x + y} : \textbf{tainted int}$$

Extended Semantics

- We need to extend the operational semantics to include qualifiers
  - Just a technical device for proving type safety
- Basic rules: ignore qualifiers q in values $q(\omega)$

$$\frac{A;\ e \Rightarrow q(0)}{A;\ \texttt{eq0}\ e \Rightarrow q(\texttt{true})}$$

$$\frac{A;\ e1 \Rightarrow q(\texttt{true}) \qquad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

Extended Semantics

- Compute qualifiers

$$\frac{A;\ e1 \Rightarrow q1\,(n1) \qquad q \text{ is } q1{+}q2 \qquad A;\ e2 \Rightarrow q2\,(n2) \qquad n3 \text{ is } n1{+}n2}{A;\ e1 + e2 \Rightarrow q\,(n3)}$$

- ***q*** is the *join* of ***q1*** and ***q2***
  - aka *least upper bound*
  - ***q*** **+** ***q*** **=** ***q***
  - **tainted + *q* = tainted**
  - ***q*** **+ tainted = tainted**

Extended Semantics

- Enforce checks

$$\frac{A;\ e \Rightarrow q'(\omega) \quad \vdash q' \leq q}{A;\ \texttt{check}(q, e) \Rightarrow q'(\omega)}$$

  - Run-time check that qualifier of value is compatible with the one required

$$\frac{A;\ e \Rightarrow q'(\omega)}{A;\ \texttt{annot}(q, e) \Rightarrow q(\omega)}$$

  - Overrides value's qualifier with the annotation

Type Safety: All Checks Succeed

- Suppose G ⊢ A and G ⊢ ***e*** : ***t***

  Then there exists ***v*** such that A;***e*** ⇒ ***v***

- Corollary: All **`check(`*q*`,`*e*`)`** terms succeed
  - Required qualifier ***q*** compatible with true qualifier ***q′***

- So: no tainted values where untainted required
  - Corollary: Can "erase" run-time qualifiers ***q*** because they have no impact on evaluation
    - No special compiler/run-time support required

Dynamic Taint Analysis

- Enforcement by static type checking/inference can be too imprecise
  - Just as static type checking is less precise than dynamic type checking
- Several languages have taint mode
  - Ruby, PHP, Perl, Python; useful for web apps
- This is basically a direct implementation of the semantics we have shown
  - Values coupled with (run-time) taint qualifier
  - Libraries extended to use annot/check

Taint Mode in Scripting Languages

- Several languages have taint mode
  - Ruby, PHP, Perl, Python; useful for web apps
  - Values coupled with (run-time) taint qualifier
  - Libraries extended to use annot/check
- Example in Ruby

```
x1 = "a string”    => "a string"
x1.tainted?        => false
y1 = ENV["HOME"]   => "/Users/mwhicks"
y1.tainted?        => true
if y1 =~ /…/ then y1.untaint end
y1.tainted?        => false
```

- http://phrogz.net/programmingruby/taint.html

Summary: Typing

$$\frac{}{G \vdash q(n) : q\ \texttt{int}} \qquad \frac{}{G \vdash q(\texttt{true}) : q\ \texttt{bool}}$$

$$\frac{G \vdash e1 : q\ \texttt{int} \quad G \vdash e2 : q\ \texttt{int}}{G \vdash e1\ \texttt{+}\ e2 : q\ \texttt{int}}$$

$$\frac{G \vdash e1 : q\ \texttt{bool} \quad G \vdash e2 : t \quad G \vdash e3 : t}{G \vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

$$\frac{G \vdash e : q'\ v}{G \vdash \texttt{annot}(q, e) : q\ v} \qquad \frac{G \vdash e : q\ \texttt{int}}{G \vdash \texttt{eq0}\ e : q\ \texttt{bool}}$$

$$\frac{G \vdash e : q\ v}{G \vdash \texttt{check}(q, e) : q\ v}$$

Summary: Semantics

$$\frac{A;\ e \Rightarrow q(0)}{A;\ \texttt{eq0}\ e \Rightarrow q(\texttt{true})}$$

$$\frac{A;\ e1 \Rightarrow q(\texttt{true}) \qquad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

$$\frac{\begin{array}{ll} A;\ e1 \Rightarrow q1(n1) & q \text{ is } q1{+}q2 \\ A;\ e2 \Rightarrow q2(n2) & n3 \text{ is } n1{+}n2 \end{array}}{A;\ e1 + e2 \Rightarrow q(n3)}$$

$$\frac{A;\ e \Rightarrow q'(\omega) \qquad \vdash q' \leq q}{A;\ \texttt{check}(q, e) \Rightarrow q'(\omega)}$$

$$\frac{A;\ e \Rightarrow q'(\omega)}{A;\ \texttt{annot}(q, e) \Rightarrow q(\omega)}$$

Secrecy

- Can apply the qualifier approach to avoid leaks
  - ***q*** = **`secret`** means data is sensitive;
  - ***q*** = **`public`** means it is *not* sensitive
  - **`public` ≤ `secret`**

```
int puts(public char *str);
-----------------------------------------
secret char *pass = “1234”;
puts(pass); // FAIL: secret > public
```

Problem: Implicit Flows

```
let x = annot(secret,…) in
let y =
  if x then annot(public,true)
  else      annot(public,false) in
check(public,y)
```

- This code type checks because the conditional always produces a public value
  - But it leaks the secret! x and y will be equal
- This is an *implicit* flow because it arises via *control* flow, not data flow
  - Qualifiers only consider data flow

Desired Property: **Noninterference**

Secret inputs have no effect on public outputs

- ***e*** enjoys noninterference iff
  - For all A1 and A2 such that A1 ~ A2
  - A1;***e*** ⇒ ***v1*** and A2;***e*** ⇒ ***v2*** implies ***v1*** ~ ***v2***
- ***v1*** ~ ***v2*** iff
  - ***v1*** = **`public`**(***ω***) and ***v2*** = **`public`**(***ω***)
    - ***v1*** and ***v2*** are the **same**, if they are public
  - ***v1*** = **`secret`**(***ω***) and ***v2*** = **`secret`**(***ω'***)
    - ***v1*** and ***v2*** need ***not*** be the same, if they are secret
- A1 ~ A2 iff A1(x) ~ A2(x) for all x

Example violates noninterference

e is

```
if x then annot(public,true)
else      annot(public,false)
```

- A1 = **x** : **secret(true)**, A2 = **x** : **secret(false)**
  - A1 ~ A2 since secrets need not be equal

- A1;**e** ⇒ **public(true)**, A2;**e** ⇒ **public(false)**
  - But **public(true)** ≁ **public(false)**

Noninterference by Typing

- Suppose G ⊢ A1 and G ⊢ A2 and A1 ~ A2

  G ⊢ ***e*** : ***t*** implies there exists ***v1***, ***v2*** such that

  A1;***e*** ⇒ ***v1*** and A2;***e*** ⇒ ***v2*** and ***v1*** ~ ***v2***

- Requires changing the rule for conditionals
  - Must join the qual of the guard with that of branches

$$\frac{G \vdash e1 : q1\ \texttt{bool} \quad q = q1{+}q2 \qquad G \vdash e2 : q2\ v \qquad G \vdash e3 : q2\ v}{G \vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : q\ v}$$

Bad example now rejected

```
let z = if x then annot(public,true)
        else      annot(public,false) in
check(public,z)
```

$$\frac{G \vdash \texttt{x} : \texttt{secret bool} \quad \texttt{secret=secret+public} \qquad G \vdash \texttt{e2} : \texttt{public bool} \quad G \vdash \texttt{e3} : \texttt{public bool}}{G \vdash \texttt{if x then e2 else e3} : \texttt{secret bool}}$$

where
G = x:secret bool e1 = annot(public,true) e2 = annot(public,false)

- **z** thus has type **secret bool**,
  and so **check(public,z)** fails to check

Implicit Flows and Tainting

- Implicit flows also affect tainted/untainted data
  - Not just public/secret data

```
let x = annot(tainted,…) in
  if x then annot(untainted,true)
  else      annot(untainted,false)
```

- Also: expression `table[x]` where `x` is ***tainted*** and `table[i]` is ***untainted***, for all `i`
  - Used to implement `tolower(c)` library routine

Other Sources of Implicit Flow

- Arrays
  - expression `table[x]` where `x` is ***secret*** and `table[i]` is ***public***, for all `i`
- Side effects (like memory writes, output)

```
if x then puts("yo"); // else do nothing
```

- Exceptions
  - `table[x]` where `x` is outside the bounds of the array, resulting in an exception
  - Dereferencing a secret pointer `x` that could be null

Proving Noninterference

- We can prove noninterference by induction on the evaluation derivation A;***e*** ⇒ ***v***
  - We consider the rules that were composed together to form the evaluation of ***e***
- We can assume the desired result holds for evaluations that are subderivations of A;***e*** ⇒ ***v***
  - So if ***e*** is **`if`** ***e1*** **`then`** ***e2*** **`else`** ***e3*** then we know that since A;***e*** ⇒ ***v*** we must have that A;***e1*** ⇒ ***v1***
    - where ***v1*** is either ***q*****`(true)`** or ***q*****`(false)`**
  - So, we can assume ***e1*** is noninterfering by induction.
- We may use other sophisticated techniques

False Alarms

- Static IFC often imprecise. Simple example:

```
let x = annot(secret,…) in
  if x then annot(public,true)
  else      annot(public,true)
```

  - There is no leak because **both branches return true**. But will be flagged as problematic.
- Many other sources of false alarms
  - Flow insensitivity, path insensitivity, non-consideration of calling context (e.g., particular arguments to functions), …
- Some leaks inconsequential (e.g., termination)

Noninterference, dynamically

- Can we enforce noninterference in the style of dynamic type checking?
  - Idea: maintain a runtime "program counter" qualifier
  - This is the join of qualifiers of conditional guards
  - It affects the qualifier of computed values

```
let x = annot(secret,…) in
y[0] := annot(public,false);
if x then y[0] := annot(public,true);
// BAD: y[0] == x
```

assignment rejected because x's qualifer is `secret`

  - Ugh: doesn't reject when x is false

Application to Realistic Software

- CQual/JQual: whole-program type inference
  - For legacy C/Java programs
  - CQual/JQual infers most annotations
    - If inference fails then there is a possible tainting violation
- Similar approach based on flow analysis
  - Aka static taint analysis
  - WALA (JOANA, Pidgin), SOOT; many others for Android (TaintDroid, DroidSafe, …). Also LLVM.
- Information Flow Control (IFC)
  - Jif: Type-based, Java dialect
  - LIO: Dynamic, Haskell library

Further Reading

- What is noninterference (and how do we enforce it)?
  - Blog post: https://www.pl-enthusiast.net/2015/03/03/noninterference/
- JIF homepage
  - http://www.cs.cornell.edu/jif/
- Language-based Information Flow Security, by Sabelfeld and Myers
  - http://www.cse.chalmers.se/~andrei/jsac.pdf
- Expressing Security policies
  - http://www.pl-enthusiast.net/2016/08/10/expressing-security-policies/