SQL injection countermeasures

The underlying issue

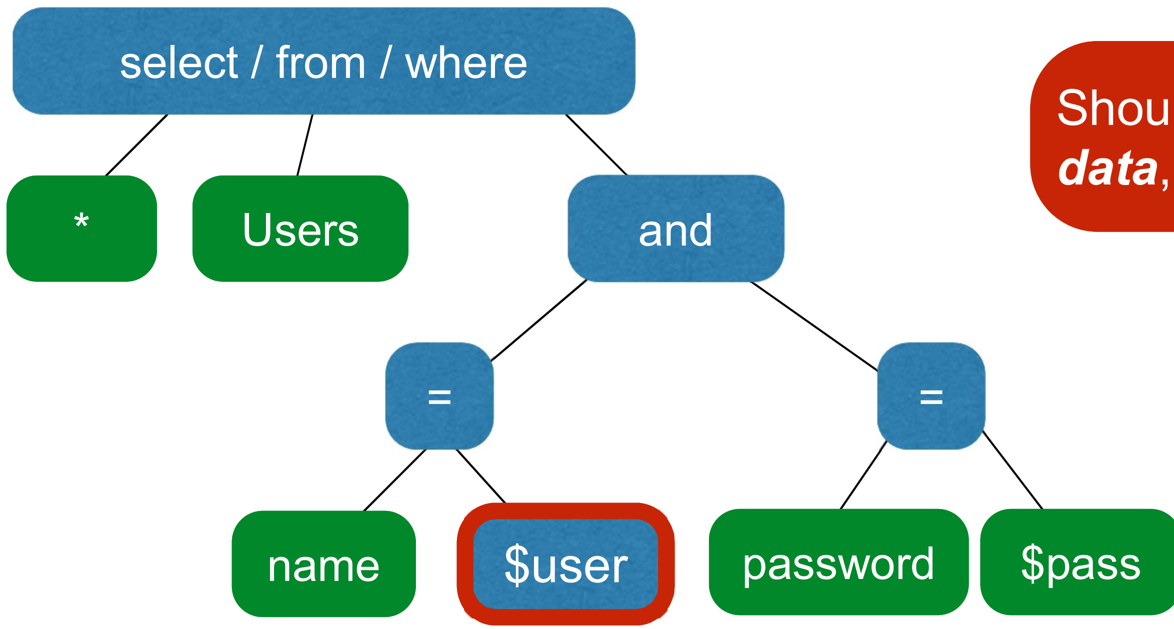
```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```

- This one string combines the **code** and the **data**
  - Similar to buffer overflows

When the boundary between code and data blurs, we open ourselves up to vulnerabilities

The underlying issue

```
$result = mysql_query("select * from Users
        where(name='$user' and password='$pass');");
```



Should be ***data***, not ***code***

Prevention: Input validation

- We require input of a certain form, but we cannot guarantee it has that form, so we must **validate it**
  - Just like we do to avoid buffer overflows
- Making input trustworthy
  - **Check** it has the expected form, reject it if not
  - **Sanitize** by modifying it or using it such that the result is correctly formed

Sanitization: Blacklisting

`'  ;  --`

- **Delete** the characters you don't want
- **Downside**: "`Lupita Nyong'o`"
  - You want these characters sometimes!
  - How do you know if/when the characters are bad?
- **Downside**: How to know you've ID'd all bad chars?

Sanitization: Escaping

- **Replace** problematic characters with safe ones
  - Change `'` to `\'`
  - Change `;` to `\;`
  - Change `-` to `\-`
  - Change `\` to `\\`
- Hard by hand, there are many libs & methods
  - `magic_quotes_gpc = On`
  - `mysql_real_escape_string()`
- **Downside**: Sometimes you want these in your SQL!
  - And escaping still may not be enough

Checking: Whitelisting

- Check that the user input is **known to be safe**
  - E.g., integer within the right range
- Rationale: Given invalid input, **safer to reject than fix**
  - “Fixes” may result in wrong output, or vulnerabilities
  - Principle of fail-safe defaults
- **Downside**: Hard for rich input!
  - How to whitelist usernames? First names?

Sanitization via escaping, whitelisting, blacklisting is HARD.

Can we do better?

Sanitization: Prepared statements

- Treat user data according to its *type*
  - Decouple the code and the data

```
$result = mysql_query("select * from Users
       where(name='$user' and password='$pass');");
```

```
$db = new mysql("localhost", "user", "pass", "DB");

$statement = $db->prepare("select * from Users
       where(name=? and password=?);");
```

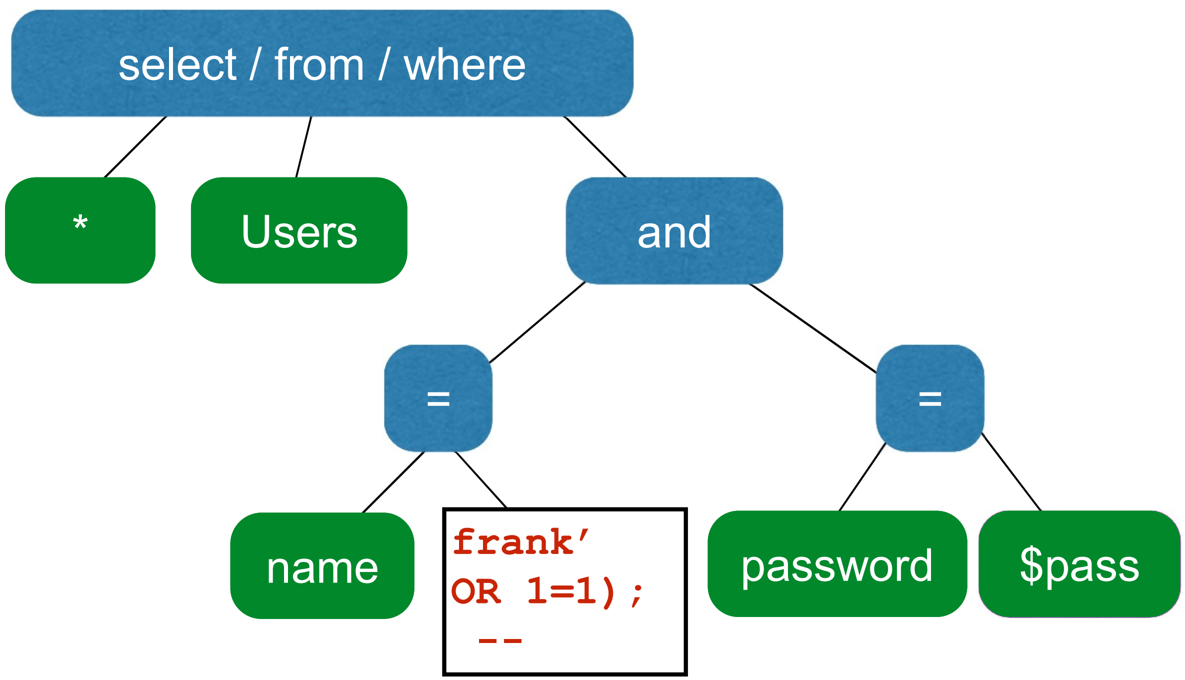
Bind variables

```
$statement->bind_param("ss", $user, $pass);
$statement->execute();
```

Bind variables are typed

Using prepared statements

```
$statement = $db->prepare("select * from Users
       where(name=?       and password=?);");
$stmt->bind_param("ss", $user, $pass);
```



Binding is only applied to the leaves, so the structure of the tree is *fixed*

Takeaways: Verify before trust

- Improperly validated input causes **many** attacks
- Common to solutions: ***check*** or ***sanitize*** **all data**
  - **Whitelisting**: More secure than blacklisting
  - **Checking**: More secure than sanitization
    - Proper sanitization is *hard*
  - **All data**: Are you sure you found all inputs?
  - Don't roll your own: libraries, frameworks, etc.

Static Analysis

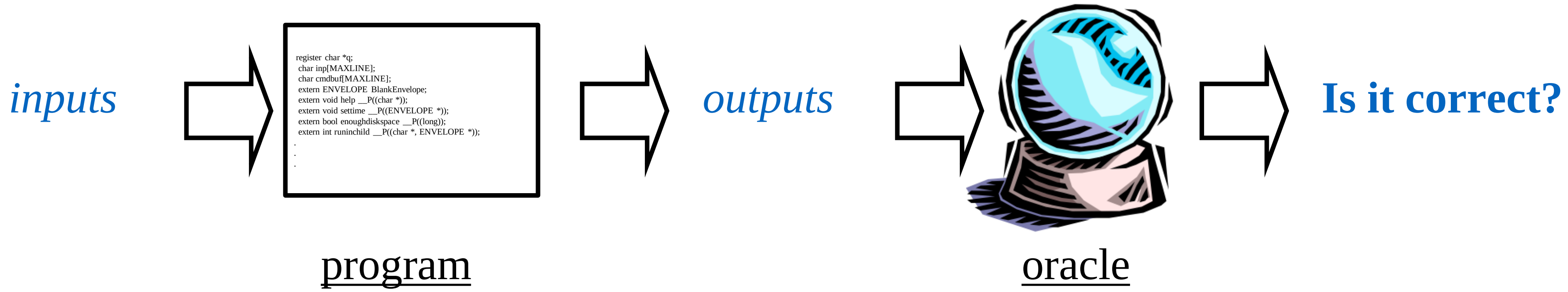
With material from Dave Levin, Mike Hicks, Dawson Engler, Lujo Bauer, Michelle Mazurek

http://philosophyofscienceportal.blogspot.com/2013/04/van-de-graaff-generator-redux.html

Static analysis

Current Practice

for Software Assurance



- **Testing:** Check correctness on set of inputs
- **Benefits**: Concrete failure proves issue, aids fix
- **Drawbacks**: Expensive, difficult, coverage?
  - No guarantees

Current Practice

(continued)

- **Code audit:** Convince someone your code is correct
- **Benefit:** Humans can generalize
- **Drawbacks**: Expensive, hard, no guarantees


```
  if (fileno(OutChannel) != fileno(stdout))
  {
    /* arrange for debugging output to go to remote host */
    (void) dup2(fileno(OutChannel), fileno(stdout));
  }
  settime(e);
  peerhostname = RealHostName;
  if (peerhostname == NULL)
    peerhostname = "localhost";
  CurHostName = peerhostname;
  CurSmtpClient = macvalue('_', e);
  if (CurSmtpClient == NULL)
    CurSmtpClient = CurHostName;

  setproctitle("server %s startup", CurSmtpClient);
#if DAEMON
  if (LogLevel > 11)
  {
    /* log connection information */
    sm_syslog(LOG_INFO, NOQID,
      "SMTP connect from %.100s (%.100s)",
      CurSmtpClient, anynet_ntoa(&RealHostAddr));
  }
#endif

  /* output the first line, inserting "ESMTP" as second word */
  expand(SmtpGreeting, inp, sizeof inp, e);
  p = strchr(inp, '\n');
  if (p != NULL)
    *p++ = '\0';
  id = strchr(inp, ' ');
  if (id == NULL)
    id = &inp[strlen(inp)];
  cmd = p == NULL ? "220 %.*s ESMTP%s" : "220-%.*s ESMTP%s";
  message(cmd, id - inp, inp, id);

  /* output remaining lines */
  while ((id = p) != NULL && (p = strchr(id, '\n')) != NULL)
  {
    *p++ = '\0';
    if (isascii(*id) && isspace(*id))
```

```
  if (!strcasecmp(c->cmdname, cmdbuf))
    break;
}

/* reset errors */
errno = 0;

/*
**  Process command.
**
**  If we are running as a null server, return 550
**  to everything.
*/

if (nullserver)
{
  switch (c->cmdcode)
  {
    case CMDQUIT:
    case CMDHELO:
    case CMDEHLO:
    case CMDNOOP:
    /* process normally */
    break;

    default:
    if (++badcommands > MAXBADCOMMANDS)
      sleep(1);
    usrerr("550 Access denied");
    continue;
  }
}

/* non-null server */
switch (c->cmdcode)
{
  case CMDMAIL:
  case CMDEXPN:
  case CMDVRFY:
```

```
  {
    *p++ = '\0';
    vp = p;

    /* skip to the end of the value */
    while (*p != '\0' && *p != ' ' &&
        !(isascii(*p) && iscntrl(*p)) &&
        *p != '=')
      p++;
  }

  if (*p != '\0')
    *p++ = '\0';

  if (tTd(19, 1))
    printf("RCPT: got arg %s=\"%s\"\n", kp,
      vp == NULL ? "<null>" : vp);

  rcpt_esmtp_args(a, kp, vp, e);
  if (Errors > 0)
    break;
}
if (Errors > 0)
  break;

/* save in recipient list after ESMTP mods */
a = recipient(a, &e->e_sendqueue, 0, e);
if (Errors > 0)
  break;

/* no errors during parsing, but might be a duplicate */
e->e_to = a->q_paddr;
if (!bitset(QBADADDR, a->q_flags))
{
  message("250 Recipient ok%s",
    bitset(QQUEUEUP, a->q_flags) ?
      " (will queue)" : "");
  nrcpts++;
}
else
{
  /* punt -- should keep message in ADDRESS.... */
```

- How can we do better?

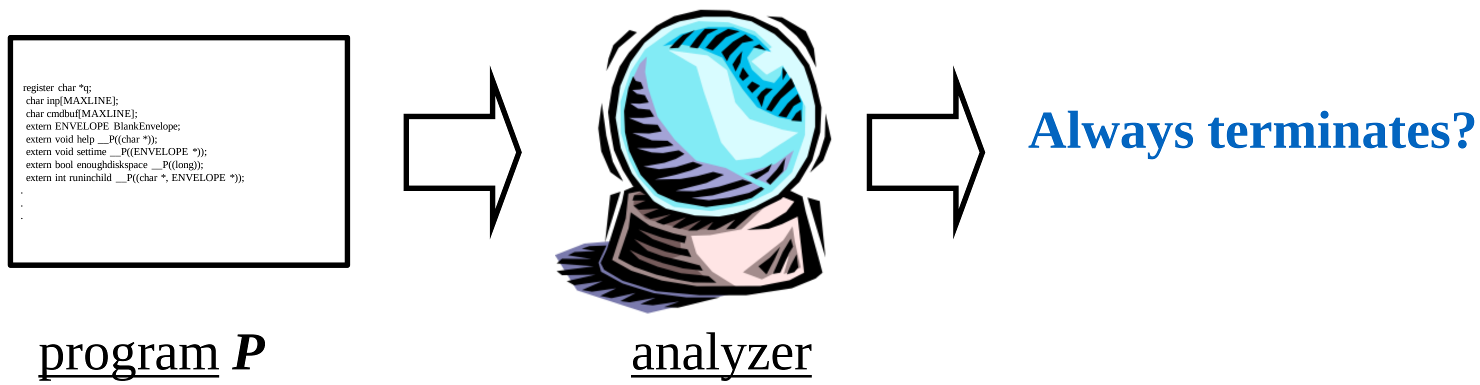
Static analysis

- Analyze program's code without running it
  - In a sense, ask a computer to do code review

- **Benefit:** (much) **higher coverage**
  - Reason about many possible runs of the program
    - Sometimes *all of them*, providing a **guarantee**
  - Reason about incomplete programs (e.g., libraries)

- **Drawbacks:**
  - Can only analyze limited properties
  - May miss some errors, or have false alarms
  - Can be time- and resource-consuming

The Halting Problem



- Can we write an analyzer that can prove, for any program *P* and inputs to it, *P* will terminate?
    - Doing so is called the **halting problem**
    - Unfortunately, this is **undecidable:** any analyzer will fail to produce an answer for at least some programs and/or inputs

Some material inspired by work of Matt Might: http://matt.might.net/articles/intro-static-analysis/

Check other properties instead?

- Perhaps security-related properties are feasible
  - E.g., that all accesses `a[i]` are in bounds
  - That a certain line of code is reachable
- *But* these **properties can be converted into the halting problem** by transforming the program
  - A perfect array bounds checker could solve the halting problem, which is impossible!
- Other undecidable properties (Rice's theorem)
  - Does this **SQL string** come from a **tainted source**?
  - Is this **pointer used after** its memory is **freed**?
  - Do any variables experience **data races**?

So is static analysis impossible?

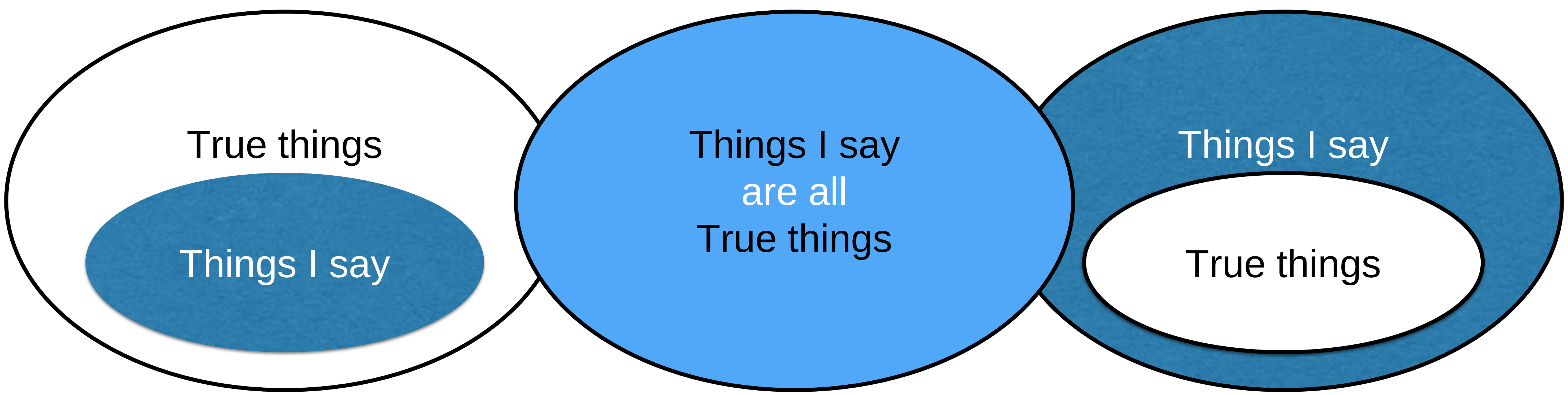
- **Perfect** static analysis is **not possible**
- **Useful** static analysis is **perfectly possible**, despite
  1. **Nontermination** - analyzer never terminates, or
  2. **False alarms** - claimed errors are not really errors, or
  3. **Missed errors** - no error reports ≠ error free
- Nonterminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors

Completeness

If analysis says that X is true, then X is true.

Soundness

If X is true, then analysis says X is true.



Trivially Complete: Say nothing

Trivially Sound: Say everything

Sound and **Complete**:
Say exactly the set of true things

Stepping back

- **Soundness**: No error found = no error exists
  - Alarms may be false errors
- **Completeness**: Any error found = real error
  - Silence does not guarantee no errors
- Basically any useful analysis
  - is neither **sound** nor **complete** (def. not **both**)
  - … usually *leans* one way or the other

Adding some depth: Taint (flow) analysis

Tainted Flow Analysis

- Cause of many attacks is **trusting unvalidated input**
  - Input from the user (network, file) 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 occasionally
  - 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
```

Analyzing taint flows

- **Goal**: For all possible inputs, prove 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 must figure it out)
- 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

Illegal Flow

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

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

untainted ≤ **tainted**

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

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

tainted ≰ **untainted**

Define allowed flow as a **lattice:**

untainted < **tainted**

At each program step, **test** whether inputs ≤ policy

Analysis Approach

- If no qualifier is present, we must **infer** it
- Steps:
  - **Create** a **name** for each missing qualifier (e.g., α, β)
  - For each program statement, **generate constraints**
    - Statement `x = y` generates constraint $q_y \le q_x$
  - **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(..);
```

1

```
α char *name = fgets(.., network_fd);
β char *x = name;
printf(x);
```

2

3

1 tainted ≤ α

2 α ≤ β

3 β ≤ untainted

Illegal flow!

No possible solution for α and β

First constraint requires α = **tainted**
To satisfy the second constraint implies β = **tainted**
But then the third constraint is illegal: **tainted ≤ untainted**

Taint Analysis: Adding ***Sensitivity***

But what about?

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

```
α char *name = fgets(.., network_fd);
β char *x;
x = name;
x = "hello!";
printf(x);
```

tainted $\leq \alpha$

$\alpha \leq \beta$

untainted $\leq \beta$

$\beta \leq$ **untainted**

No constraint solution. Bug?

False Alarm!

Flow Sensitivity

- Our analysis is **flow *in*sensitive**
  - Each variable has **one qualifier**
  - Conflates the taintedness of all values it ever contains
- **Flow-sensitive analysis** accounts for variables whose contents change
  - Allow each assigned use of a variable to have a different qualifier
    - E.g., α_1 is x's qualifier at line 1, but α_2 is the qualifier at line 2, where α_1 and α_2 can differ
  - Could implement this by transforming the program to assign to a variable at most once

Reworked Example

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

```
→ α char *name = fgets(.., network_fd);
char β *x1, γ *x2;
x1 = name;
x2 = "hello!";
printf(x2);
```

tainted ≤ α

α ≤ β

untainted ≤ γ

γ ≤ **untainted**

No Alarm

Good solution exists:

γ = **untainted**

α = β = **tainted**

Handling conditionals

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

```
α char *name = fgets(.., network_fd);
β char *x;
if (..)  x = name;
else     x = "hello!";
printf(x);
```

tainted ≤ α

α ≤ β

~~**untainted** ≤ β~~

β ≤ **untainted**

Constraints still unsolvable

Illegal flow

Multiple Conditionals

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

```
void f(int x) {
   α char *y;
   if (x) y = "hello!";
   else   y = fgets(.., network_fd);
   if (x) printf(y);
}
```

~~untainted ≤ α~~

tainted ≤ α

α ≤ untainted

No solution for α. Bug?

False Alarm!

(and flow sensitivity won't help)

Path Sensitivity

- Consider *path feasibility*. E.g., `f(x)` can execute path
  - **1-2-4-5-6** when $x \neq 0$, or
  - **1-3-4-6** when $x == 0$. But,
  - path **1-3-4-5-6** *infeasible*

```
void f(int x) {
  char *y;
  if (x) y = “hello!”;
  else y = fgets(…);
  if (x) printf(y);
}
```

- A **path sensitive analysis** checks feasibility, e.g., by qualifying each constraint with a **path condition**
  - $x \neq 0 \Longrightarrow$ **untainted** $\leq \alpha$ (segment 1-2)
  - $x = 0 \Longrightarrow$ **tainted** $\leq \alpha$ (segment 1-3)
  - $x \neq 0 \Longrightarrow \alpha \leq$ **untainted** (segment 4-5)

Static analysis in practice

coverity® A Synopsys Company

GRAMMATECH

hp Fortify

klocwork® a Rogue Wave Company

VERACODE

FindBugs™

LLVM

clang analyzer & KLEE

Joern

Caveat: appearance in the above list is not an implicit endorsement, and these are only a sample of available offerings