# Static Analysis

Proactive Computer Security, 2011

Ken Friis Larsen
kflarsen@diku.dk

# Static Analysis

- Goal: Use computers to assist in auditing
- We assume we have either source code or a binary available

# Classic SQL Injection

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = ’"
              + username
              + "’and password = ’"
              + pwd + "’";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

# Tainted Data

- Idea: Keep tack of tainted (user-controlled) data
- Sources: functions that get tainted data
- Sinks: functions that needs to be protected from tainted data
- Objective: check that tainted data cannot get from a source to a sink

# Tainted SQL Query

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = ’"
               + username
               + "’and password = ’"
               + pwd + "’";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

# Tainted SQL Query

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = '"
             + username
             + "'and password = '"
             + pwd + "'";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

# Tainted SQL Query

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = ’"
               + username
               + "’and password = ’"
               + pwd + "’";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

# Tainted SQL Query

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = ’"
               + username
               + "’and password = ’"
               + pwd + "’";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

# Tainted SQL Query

```
public void authenticate(HttpServletRequest request){
  String username = request.getParameter("user");
  java.sql.Statement stmt = con.createStatement();

  String query = "select * from users where username = ’"
           + username
           + "’and password = ’"
           + pwd + "’";

  stmt.execute(query);
  ... // process the result of SELECT
}
```

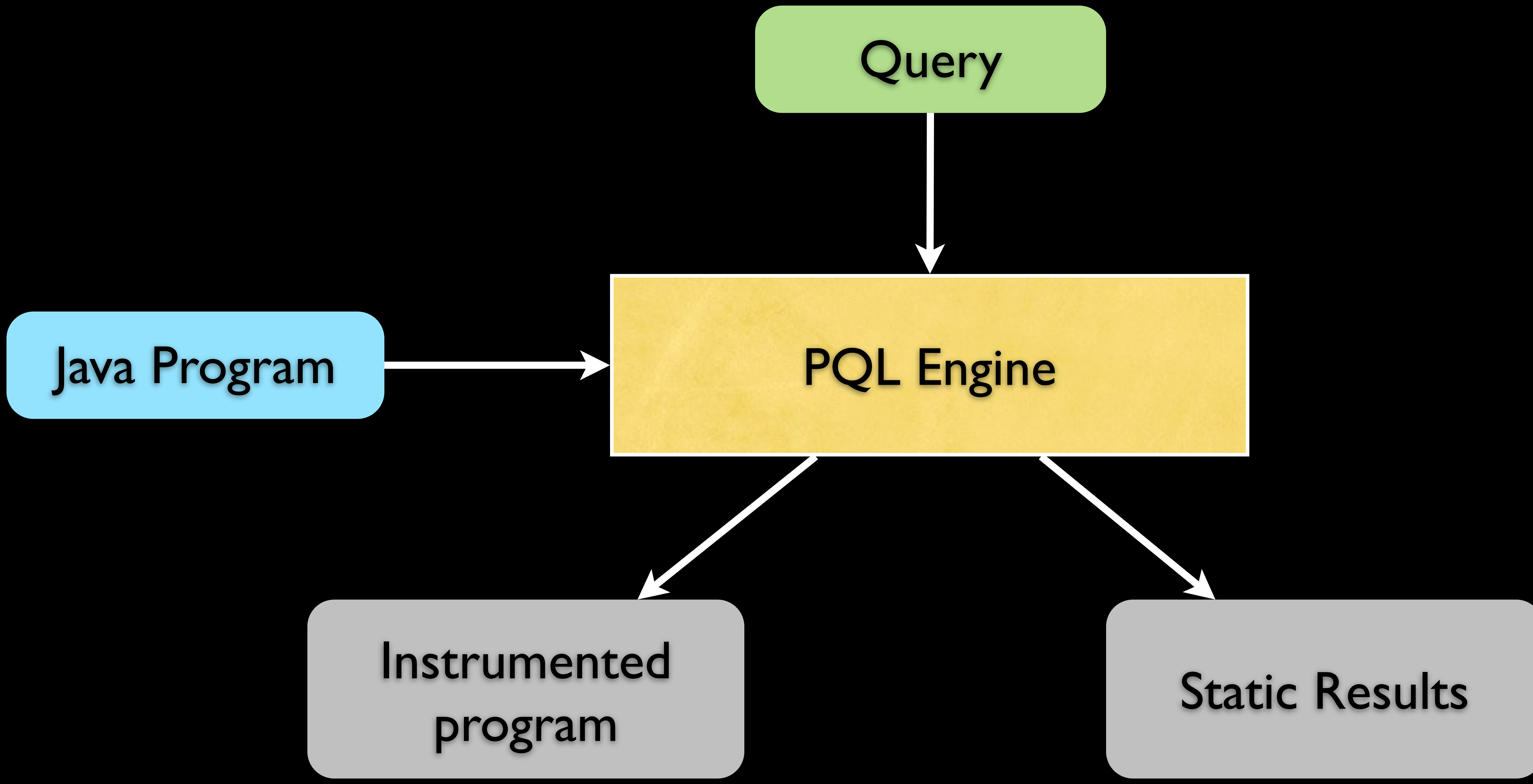
# Keeping Track of Tainted Data

- DIY using the type-system
  - make you own string type, wrap all sources
  - Perl has taint-mode
- Use flow analysis
  - except that program analysis can be hard to write

# Why a Program Query Language?

- Security specialist: knows which bugs to look for, but don't (necessarily) know how to automate audit for these bugs.
- Program analysis specialist: know how to write program analysis, but don't know which bugs to look for.

# PQL System Architecture



# When to Check a Query

- **Dynamic analysis:** finds matches at runtime
  Can fix code by replacing instructions
- **Static analysis:** finds all possible matches
  Conservative: can prove lack of match
  Results can be used to optimise dynamic analysis

# Query For SQL Injections

```
  query injection() uses Object param, final;
matches { param = getParameter(_) | param = getHeader();
          final := derived (param);
          java.sql.Connection.execute (final);
}

  query derived(Object x) uses Object t;
returns Object y;
matches { { y := x; }
        | { t = x.toString(); y := derived(t); }
        | { t.append(x); y := derived(t); }
}
```

# Combining Fuzzing and Static Analysis

- Advanced static analysis is sometimes not feasible for whole large programs. Too slow or too conservative
- Creating instrumented programs can also be problematic

# Whitebox Fuzzing

- Idea: mix fuzz testing with dynamic test generation

1) Symbolic execution
2) Collect constraints on inputs
3) Negate those, solve with constraint solver, generate new inputs using DART (directed automated random testing)

# SAGE

- SAGE is a whitebox fuzzer for *unmodified* x86 (windows) binaries
- Start with a well-formed input (not random)
- Combine with a generational search (not DFS): negate collected constraints on a path one-by-one

# Example Program

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

# Example Program

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

input: "good"

# Example Program

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

input: “good”

Path constrains:

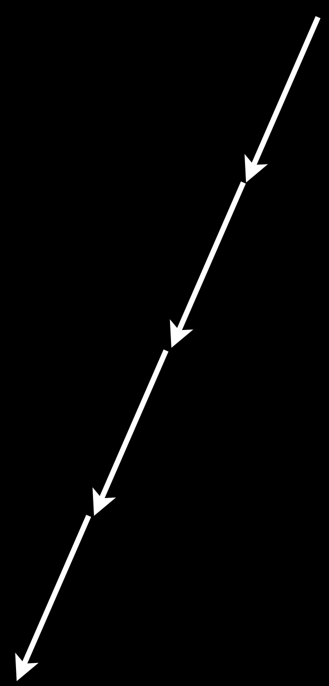
$I_0$ != ‘b’

$I_1$ != ‘a’

$I_2$ != ‘d’

$I_3$ != ‘!’

# Example Program

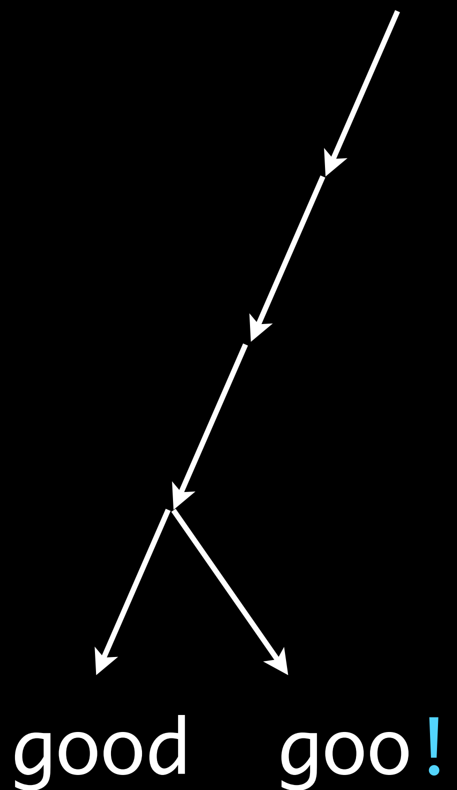


good

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;   I0 != 'b'
  if (input[1] == 'a') cnt++;   I1 != 'a'
  if (input[2] == 'd') cnt++;   I2 != 'd'
  if (input[3] == '!') cnt++;   I3 != '!'
  if (cnt >= 3) crash();
}
```

Negate the constraints one-by-one

# Example Program

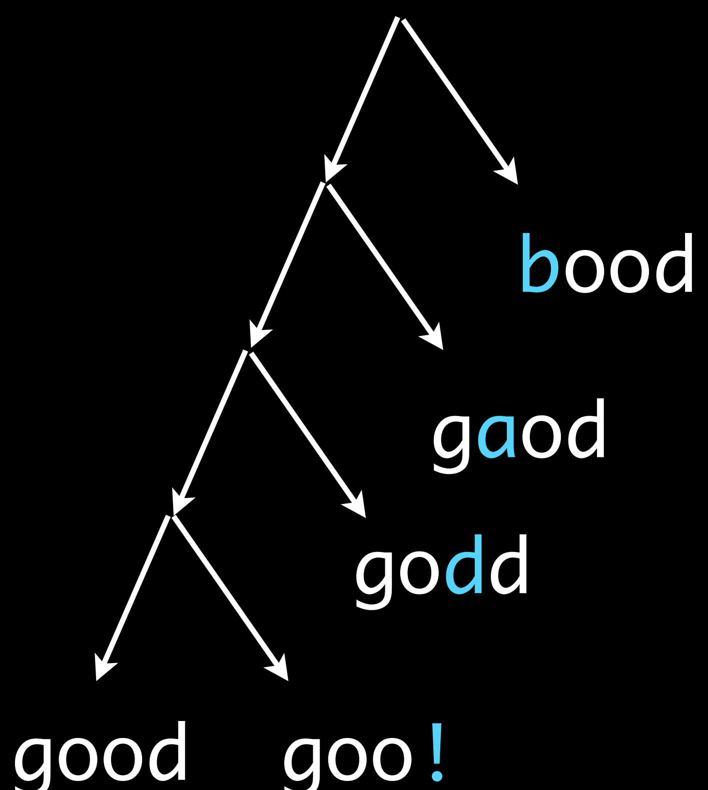



```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

$I_0$ != 'b'
$I_1$ != 'a'
$I_2$ != 'd'
$I_3$ == '!'

Negate the constraints one-by-one

# Example Program




```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) crash();
}
```

$I_0 == \text{'b'}$
$I_1 == \text{'a'}$
$I_2 == \text{'d'}$
$I_3 == \text{'!'}$

Negate the constraints one-by-one

# SAGE Results

- Huge success internally at Microsoft
- Example: 1/3 of all security (internally) bugs on Win 7 was found by SAGE
- Running 24/7 on 100s of machines

# Blackbox vs Whitebox Fuzzing

- Blackbox is lightweight, easy to implement, and fast. But may give poor coverage
- Whitebox gives better coverage, but is complex to implement and slower

# Summary

- Static analysis tools:
  - build-in set of queries;
  - or query language
- Queries can also be used for run-time monitoring
- Analysis technology can also be used for building smarter fuzzing tools

- Remember D-Day: June 8
- Please take time to fill-out the student evaluation

```
class TaintRoot extends Method {
  TaintRoot() {
    exists(string n  | n = this.getName() and
          (n = "getParameter" or n = "getQueryString" or n = "getHeader")) and
    exists(RefType t | t = this.getDeclaringType() and
            (  t.hasQualifiedName("javax.servlet.http","HttpServletRequest")
            or t.hasQualifiedName("javax.servlet","ServletRequest")))
  }
}

predicate isTainted(Expr e) {
      ((MethodAccess)e).getMethod() instanceof TaintRoot
   or exists(AssignExpr ae |
                ((VarAccess)ae.getDest()).getVariable()
                 =
                ((VarAccess)e).getVariable() and isTainted(ae.getSource()))
   or exists(LocalVariableDeclExpr lvde |
                ((VarAccess)e).getVariable() = lvde.getVariable() and
                isTainted(lvde.getInit()))
   or isTainted(((AddExpr)e).getAnOperand())
   or isTainted(((ParExpr)e).getExpr())
}

from MethodAccess ma, Method send
where ma.getMethod() = send and send.hasName("sendError")
  and send.getDeclaringType()
          .hasQualifiedName("javax.servlet.http", "HttpServletResponse") and
      isTainted(ma.getArgument(1))
select ma, "Cross-site scripting vulnerability"
```