Finding Application Errors and Security Flaws Using PQL: A Program Query Language

Michael Martin, Benjamin Livshits, Monica S. Lam
Stanford University
OOPSLA 2005
The Problem

- Lots of bug-finding research
  - Null dereferences
  - Buffer overruns
  - Data races
- Many bugs application-specific
  - Misuse of libraries
  - Violation of application logic
Solution: Division of Labor

- Programmer
  - Knows target program
  - Doesn’t know analysis

- Program Analysis Specialists
  - Knows analysis
  - Doesn’t know specific bugs

Give the programmer a usable analysis
Program Query Language

- Queries operate on *program traces*
  - Sequence of events representing a run
  - Refers to object *instances*, not variables
  - Events matched may be widely spaced
- Patterns resemble Java code
  - Like a small matching code snippet
  - No references to compiler internals
System Architecture

- Question
- PQL Query
- PQL Engine
  - instrumenter
  - static analyzer
- Instrumented Program
- Optimized Instrumented Program
- Static Results
Complementary Approaches

- **Dynamic analysis:** finds matches at run time
  - After a match:
    - Can execute user code
    - Can fix code by replacing instructions
- **Static analysis:** finds all possible matches
  - Conservative: can prove lack of match
  - Results can optimize dynamic analysis
Experimental Results

- Explored a wide range of PQL queries
  - Bad session stores (API violations)
  - SQL injections (security flaws)
  - Mismatched calls (API violations)
  - Lapsed listeners (memory leaks)
- Automatically repaired and prevented many bugs at runtime
  - Fixed memory leaks, prevented security flaws
- Runtime overhead is reasonable
  - Overhead in the 9-125% range
  - Static optimization removed 82-99% of instr. points
- Found 206 bugs in 6 real-life Java applications
  - Eclipse, popular web applications
  - 60,000 classes combined
System Architecture

Question

PQL Query

PQL Engine

Program

Instrumenter

Static analyzer

Instrumented Program

Optimized Instrumented Program

Static Results
Running Example: SQL Injection

- Unvalidated user input passed to a database back-end for execution
- If SQL is embedded in the input, attacker can take over database
  - Unauthorized data reads
  - Unauthorized data modifications
- One of the top web security flaws
**SQL Injection 1**

HttpServletRequest req = /* ... */;
java.sql.Connection conn = /* ... */;
String q = req.getParameter("QUERY");
conn.execute(q);

1. CALL o₁.getParameter(o₂)
2. RET o₂
3. CALL o₃.execute(o₂)
4. RET o₄
String read() {
    HttpServletRequest req = /* ... */;
    return req.getParameter("QUERY");
}

java.sql.Connection conn = /* ... */;
conn.execute(read());

1. CALL read()
2. CALL o₁.getParameter(o₂)
3. RET o₃
4. RET o₃
5. CALL o₄.execute(o₃)
6. RET o₅
The object returned by `getParameter` is then *argument 1 to `execute`*
Translates Directly to PQL

query main()
uses String x;
matches {
    x = HttpServletRequest.getParameter(_);
    Connection.execute(x);
}

- Query variables $\rightarrow$ heap objects
- Instructions need not be adjacent in trace
Alternation

```java
query main()
uses String x;
matches {
    x = HttpServletRequest.getParameter(_)
    | x = HttpServletRequest.getHeader(_);
    Connection.execute(x);
}
```
HttpServletRequest req = /* ... */;
n = getParameter("NAME");
p = getParameter("PASSWORD");
conn.execute(
    "SELECT * FROM logins WHERE name=" + n + 
    " AND passwd=" + p
);

- Compiler translates string concatenation into operations on String and StringBuffer objects
SQL Injection 3

1 CALL o₁.getParameter(o₂)
2 RET o₃
3 CALL o₁.getParameter(o₄)
4 RET o₅
5 CALL StringBuffer.<init>(o₆)
6 RET o₇
7 CALL o₇.append(o₈)
8 RET o₇
9 CALL o₇.append(o₃)
10 RET o₇
11 CALL o₇.append(o₉)
12 RET o₇
13 CALL o₇.append(o₅)
14 RET o₇
15 CALL o₇.toString()
16 RET o₁₀
17 CALL o₁₁.execute(o₁₀)
18 RET o₁₂
Old Pattern Doesn’t Work

1. \text{CALL} \quad o_1 \text{.getParameter} (o_2)
2. \text{RET} \quad o_3
3. \text{CALL} \quad o_1 \text{.getParameter} (o_4)
4. \text{RET} \quad o_5
17. \text{CALL} \quad o_{11} \text{.execute} (o_{10})

\textit{o_{10} is neither o_3 nor o_5, so no match}
Instance of Tainted Data Problem

- User-controlled input must be trapped and validated before being passed to database.
- Objects derived from an initial input must also be considered user controlled.
- Generalizes to many security problems: cross-site scripting, path traversal, response splitting, format string attacks...
Pattern Must Catch Derived Strings

1. CALL $o_1$.getParameter($o_2$)
2. RET $o_3$
3. CALL $o_1$.getParameter($o_4$)
4. RET $o_5$

9. CALL $o_7$.append($o_3$)
10. RET $o_7$
11. CALL $o_7$.append($o_9$)
12. RET $o_7$

15. CALL $o_7$.toString()
16. RET $o_{10}$
17. CALL $o_{11}$.execute($o_{10}$)
Pattern Must Catch Derived Strings

1. CALL $o_1$.getParameter($o_2$)
2. RET $o_3$
3. CALL $o_1$.getParameter($o_4$)
4. RET $o_5$

9. CALL $o_7$.append($o_3$)
10. RET $o_7$
11. CALL $o_7$.append($o_9$)
12. RET $o_7$

15. CALL $o_7$.toString()
16. RET $o_{10}$
17. CALL $o_{11}$.execute($o_{10}$)
Pattern Must Catch Derived Strings

1. CALL $o_1$.getParameter($o_2$)
2. RET $o_3$
3. CALL $o_1$.getParameter($o_4$)
4. RET $o_5$
9. CALL $o_7$.append($o_3$)
10. RET $o_7$
11. CALL $o_7$.append($o_9$)
12. RET $o_7$
15. CALL $o_7$.toString()
16. RET $o_{10}$
17. CALL $o_{11}$.execute($o_{10}$)
query derived (Object x)
    uses Object temp;
    returns Object d;
matches {
    { temp.append(x); d := derived(temp); } |
    { temp = x.toString(); d := derived(temp); } |
    { d := x; }
}
New Main Query

```java
query main()
uses String x, final;
matches {
    x = HttpServletRequest.getParameter(_)
    | x = HttpServletRequest.getHeader(_);
    final := derived(x);
    Connection.execute(final);
}
```
Defending Against Attacks

```java
query main()
uses String x, final;
matches {
    x = HttpServletRequest.getParameter(_)
    | x = HttpServletRequest.getHeader(_);
    final := derived(x);
}
replaces Connection.execute(final) with
    SQLUtil.safeExecute(x, final);
```

- Sanitizes user-derived input
- Dangerous data cannot reach the database
Other PQL Constructs

- Partial order
  - \{ x.a(), x.b(), x.c(); \}
  - Match calls to \texttt{a}, \texttt{b}, and \texttt{c} on \texttt{x} in \textit{any} order.

- Forbidden Events
  - Example: double-lock
    \texttt{x.lock(); \sim x.unlock(); x.lock();}
  - Single statements only
Expressiveness

- Concatenation + alternation = Loop-free regexp
- + Subqueries = CFG
- + Partial Order = CFG + Intersection
- Quantified over heap
  - Each subquery independent
  - Existentially quantified
System Architecture

Question

PQL Query

Program

PQL Engine

instrumenter

static analyzer

Instrumented Program

Optimized Instrumented Program

Static Results
Dynamic Matcher

- Subquery $\rightarrow$ state machine
- Call to subquery $\rightarrow$ new instance of machine
- States carry “bindings” with them
  - Query variables $\rightarrow$ heap objects
  - Bindings are acquired as the variables are referenced for the first time in a match
query main()
uses Object x, final;
matches {
    x = getParameter(_); | x = getHeader();
    f := derived (x); execute (f);
}

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); }
}
main() Query Machine

\[
x = \text{getParameter}(\_)
\]

\[
x = \text{getHeader}(\_)
\]

\[
f := \text{derived}(x)
\]

\[
\text{execute}(f)
\]
derived() Query Machine

y := x

t := x.toString()
y := derived(t)
t.append(x)
y := derived(t)
Example Program Trace

\[ o_1 = \text{getHeader}(o_2) \]
\[ o_3 \cdot \text{append}(o_1) \]
\[ o_3 \cdot \text{append}(o_4) \]
\[ o_5 = \text{execute}(o_3) \]
main(): Top Level Match

x = getParameter(_)  x = getHeader(_)

{ x = o₁ }  { x = o₁ }₁

f := derived(x)

execute(f)

o₁ = getHeader(o₂)
**derived(): call 1**

\[ o_1 = \text{getHeader}(o_2) \]
main(): Top Level Match

\[
x = \text{getParameter}(\_)
\]

\[
x = \text{getHeader}(\_)
\]

\[
o_1 = \text{getHeader}(o_2)
\]

\[
o_3.append(o_1)
\]

f := derived(x)

execute(f)
derived(): call 1

\[
\begin{align*}
& o_1 = \text{getHeader}(o_2) \\
& o_3.\text{append}(o_1)
\end{align*}
\]
derived(): call 2

\[ y := x \]
\[ t = x.\text{toString()} \]
\[ y := \text{derived}(t) \]

\[ o_1 = \text{getHeader}(o_2) \]
\[ o_3.\text{append}(o_1) \]
derived(): call 1

\[ t = x.toString() \]
\[ y := \text{derived}(t) \]
\[ \{ x = o_1 \} \]

\[ t = x.toString() \]
\[ y := \text{derived}(t) \]
\[ \{ x = y = o_1 \} \]

\[ t = x.toString() \]
\[ y := \text{derived}(t) \]
\[ \{ x = o_1, y = t = o_3 \} \]

\[ o_1 = \text{getHeader}(o_2) \]
\[ o_3.append(o_1) \]
main(): Top Level Match

$x = \text{getParameter}(_)\}$

$x = \text{getHeader}(_)\}$

$o_1 = \text{getHeader}(o_2)$

$o_3.\text{append}(o_1)$

$o_3.\text{append}(o_4)$

$o_5 = \text{execute}(o_3)$

$f := \text{derived}(x)$

execute($f$)

$\{x = o_1, f = o_3\}$
Find Relevance Fast

- Hash map for each transition
  - Every call-instance combined
- Key on known-used variables
- All used variables known-used → one lookup per transition
System Architecture

- Question
  - PQL Query
    - PQL Engine
      - instrumenter
      - static analyzer
        - Instrumented Program
        - Optimized Instrumented Program
        - Static Results
Static Analysis

- “Can this program match this query?”
- **Undecidable in general**
- **We give a conservative approximation**
- **No matches found = None possible**
Static Analysis

- PQL query automatically translated to query on pointer analysis results
- Pointer analysis is *sound* and *context-sensitive*
  - $10^{14}$ contexts in a good-sized application
  - Exponential space represented with BDDs
  - Analyses given in Datalog
- Whaley/Lam, PLDI 2004 (*bddbddb*) for details
Static Results

- Sets of objects and events that could represent a match

  OR

- Program points that could participate in a match

- No results = no match possible!
Optimizing the Dynamic Analysis

- Static results conservative
  - So, point not in result → point never in any match
  - So, no need to instrument
- Usually more than 90% reduction
Experiment Topics

- Domain of Java Web applications
  - Serialization errors
  - SQL injection
- Domain of Eclipse IDE plugins
  - API violations
  - Memory leaks
## Experiment Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Classes</th>
<th>Inst Pts</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>webgoat</td>
<td>1,021</td>
<td>69</td>
<td>2</td>
</tr>
<tr>
<td>personalblog</td>
<td>5,236</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>road2hibernate</td>
<td>7,062</td>
<td>779</td>
<td>1</td>
</tr>
<tr>
<td>snipsnap</td>
<td>10,851</td>
<td>543</td>
<td>8</td>
</tr>
<tr>
<td>roller</td>
<td>16,359</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Eclipse</td>
<td>19,439</td>
<td>18,152</td>
<td>192</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>59,968</strong></td>
<td><strong>19,579</strong></td>
<td><strong>206</strong></td>
</tr>
</tbody>
</table>
Session Serialization Errors

- Very common bug in Web applications
- Server tries to persist non-persistent objects
  - Only manifests under heavy load
  - Hard to find with testing
- One-line query in PQL
  - `HttpSession.setAttribute(_, !Serializable);`
- Solvable purely statically
  - Dynamic confirmation possible
SQL Injection

- Our running example
- Static optimizes greatly
  - 92%-99.8% reduction of points
  - 2-3x speedup
- 4 injections, 2 exploitable
  - Blocked both exploits
- Further applications and an improved static analysis in Usenix Security ’05
**Full derived() Query**

```java
query derived (Object x)
returns Object y;
uses Object temp;
matches {
    y := x
    | { temp = StringProp(x);
        y := derived(temp); }
}

query StringProp (Object * x)
returns Object y;
uses Object z;
matches {
    y.append(x, ...)
    | x.getChars(_, _, y, _)
    | y.insert(_, x)
    | y.replace(_, _, x)
    | y = x.substring(...)
    | y = new java.lang.String(x)
    | y = new java.lang.StringBuffer(x)
    | y = x.toString()
    | y = x.getBytes(...)
    | y = _.copyValueOf(x)
    | y = x.concat(_)
    | y = _.concat(x)
    | y = new java.util.StringTokenizer(x)
    | y = x.nextToken()
    | y = x.next()
    | y = new java.lang.Number(x)
    | y = x.trim()
    | { z = x.split(...); y = z[]; }
    | y = x.toLowerCase(...)
    | y = x.toUpperCase(...)
    | y = _.replaceAll(_, x)
    | y = _.replaceFirst(_, x);
```
Eclipse

- IDE for Java
- Very large (tens of MB of bytecode)
  - Too large for our static analysis
- Purely interactive
  - Unoptimized dynamic overhead acceptable
Queries on Eclipse

- Paired Methods
  - register/deregister
  - createWidget/destroyWidget
  - install/uninstall
  - startup/shutdown

- Lapsed Listeners
Eclipse Results

- All paired methods queries were run simultaneously
  - 56 mismatches detected
- Lapsed listener query was run alone
  - 136 lapsed listeners
  - Can be automatically fixed
Current Status

- Open source and hosted on SourceForge
- http://pql.sf.net – standalone dynamic implementation
Related work

- PQL is a *query language*
  - JQuery
- on *program traces*
  - Partiqle, Dalek, ...
- Observing *behavior and finding bugs*
  - Metal, Daikon, PREfix, Clouseau, ...
- and *automatically add code to fix them*
  - AspectJ
Conclusions

- **PQL – a Program Query Language**
  - Match histories of sets of objects on a program trace
  - Targeting application developers
- **Found many bugs**
  - 206 application bugs and security flaws
  - 6 large real-life applications
- **PQL provides a bridge to powerful analyses**
  - Dynamic matcher
    - Point-and-shoot even for unknown applications
    - Automatically repairs program on the fly
  - Static matcher
    - Proves absence of bugs
    - Can reduce runtime overhead to production-acceptable