Dynamic taint analysis
(aka dynamic information-flow analysis)
Dynamic tainting applications

Attack detection / prevention

Information policy enforcement

Testing

Data lifetime / scope
Dynamic tainting applications

Attack detection / prevention
Detect / prevent attacks such as SQL injection, buffer overruns, stack smashing, cross site scripting

- Suh et al. 04,
- Newsome and Song 05,
- Halfond et al. 06,
- Kong et al. 06,
- Qin et al. 06

Testing

Data lifetime / scope
Dynamic tainting applications

Information policy enforcement

Attack detection / prevention

Information policy enforcement

ensure classified information does not leak outside the system

e.g., Vachharajani et al. 04, McCamant and Ernst 06

Testing

Data lifetime / scope
Dynamic tainting applications

- Attack detection / prevention

- Information policy enforcement

Testing

Coverage metrics, test data generation heuristic, ...

e.g., Masri et al 05, Leek et al. 07

Data lifetime / scope
Dynamic tainting applications

Attack detection / prevention

Information policy enforcement

Testing

Data lifetime / scope
track how long sensitive data, such as passwords or account numbers, remain in the application
e.g., Chow et al. 04
Motivation

- Flexible
- Easy to use
- Accurate

Configuration

Dytan Generic Framework

- Flexible
- Easy to use
- Accurate

Custom Dynamic Taint Analysis

Results
Outline

• Motivation & overview
  • Framework (Dytan)
    • flexibility
    • ease of use
    • accuracy
  • Empirical evaluation
  • Conclusions
Framework: flexibility

Configuration
Framework: flexibility

Taint sources

Propagation policy

Taint sinks
Framework: flexibility

Taint sources

Propagation policy

Taint sinks

Which data to tag, and how to tag it
Framework: flexibility

How tags should be propagated at runtime
Framework: flexibility

Taint sources

Propagation policy

Taint sinks

Where and how tags should be checked
## Taint sources

### What to tag

Identify what program data should be assigned tags

- Variables (local or global)
- Function parameters
- Function return values
- Data from an input stream network, filesystem, keyboard, ...
- Specific input stream 141.195.121.134:80, a.txt,...

### How to tag

Describe how tags should be assigned for identified data

- Single tag
- One tag per source
- Multiple tags per source
- ...

---

**Note:**

- Single tag
- One tag per source
- Multiple tags per source
- Specific input stream 141.195.121.134:80, a.txt,...
Taint sources

What to tag: a.txt
How to tag: single tag

a.txt

1 1 1 1 1 1
Taint sources

What to tag: a.txt

How to tag: multiple tags
Affecting data
Data that affects the outcome of a statement through
• Data dependencies
• Control dependencies

A policy can consider both or only data dependencies

Mapping function
Define how tags associated with affecting data should be combined
• Union
• Max
• ...

Propagation policy
if(X) {
  C = A + B;
}

Propagation policy

Affecting data:
- data dependence
- control dependence

Mapping function:
- union
- max
Propagation policy

if\( (X) \) {
  C = A + B;
}

Affecting data:
- data dependence
- control dependence

Mapping function:
- union
- max
if (X) {
  C = A + B;
}

Propagation policy

Affecting data:
- data dependence
- control dependence

Mapping function:
- union
- max
Taint Sinks

Where to check
Location in the program to perform a check
• Function entry / exit
• Statement type
• Specific program point

What to check
The data whose tags should be checked
• Variables
• Function parameters
• Function return value

How to check
Set of conditions to check and a set of actions to perform if the conditions are not met.
• validate presence of tags (exit or log)
• ensure absence of tags (exit or log)
• ...
Taint Sinks

cmd = read(file);
args = read(socket);
cmd = trim(cmd + args);
...
tok[] = parse(cmd);
exec(tok[0], tok[1]);
Taint Sinks

Where / what to check:
function: exec, param: 0

How to check:
validate presence of:
validate absence of:

Result:

```python
cmd = read(file);
args = read(socket);
cmd = trim(cmd + args);
...
tok[] = parse(cmd);
exec(tok[0], tok[1]);
```
Taint Sinks

<table>
<thead>
<tr>
<th>Where / what to check:</th>
</tr>
</thead>
<tbody>
<tr>
<td>function: exec, param: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How to check:</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate presence of:</td>
</tr>
<tr>
<td>validate absence of:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
</tr>
</tbody>
</table>
Framework: ease of use

Provide two ways to configure the framework

- **Basic**
  - Select sources, propagation policies, and sinks from a set of predefined options
  - XML based configuration

- **Advanced**
  - Suitable for more esoteric applications
  - Extend OO implementation
Framework: accuracy

- Dytan operates at the binary level
- consider the actual program semantics
- transparently handle libraries
- Dytan accounts for both data- and control-flow dependencies
Framework: accuracy

The most common source of inaccuracy is incorrectly identifying the information produced and consumed by a statement.

Two common examples:

- Implicit operands
  ```
  add %eax, %ebx  // A = A + B  
  produced: %eax, %eflags
  ```

- Address Generators
  ```
  add %eax, [%ebx] // A = A + *B  
  consumed: %eax, [%ebx], %ebx
  ```
Outline

- Motivation & overview
- Framework
  - flexibility
  - ease of use
  - accuracy
- Empirical evaluation
- Conclusions
Empirical evaluation

• RQ1: Can Dytan be used to (easily) implement existing dynamic taint analyses?

• RQ2: How do inaccurate propagation policies affect the analysis results?

• In addition: discussion on performance
RQ 1: flexibility

**Goal:** show that Dytan can be used to (easily) implement existing dynamic taint analyses

- Selected two techniques:
  - Overwrite attack detection [Qin et al. 04]
  - SQL injection detection [Halfond et al. 06]
- Used Dytan to re-implement both techniques
- Measure implementation time
- Validate against the original implementation
RQ1: results

- Implementation time:
  - Overwrite attack detection: < 1 hour
  - SQL injection detection: < 1 day

- Comparison with original implementations:
  - Successfully stopped same attacks as the original implementations
RQ2: accuracy impact

**Goal:** measure the effect of inaccurate propagation policies on analysis results

- Selected two subjects:
  - Gzip (75kb w/o libraries)
  - Firefox (850kb w/o libraries)

- Use Dytan to taint program inputs and measure the amount of heap data tainted at program exit

- Compare Dytan against inaccurate policies
  - no implicit operands (no IM)
  - no address generators (no AG)
  - no implicit operands, no address generators (no IM, no AG)
RQ2: results

Firefox (1 page)  Firefox (3 pages)  Gzip

- Dytan
- No IM
- No AG
- No IM, no IG
Performance

• Measured for `gzip`:
  \( \approx 30x \) for data flow
  \( \approx 50x \) for data and control flow

• High overhead, but...
  • In line with existing implementations
  • Designed for experimentation
  • Favors flexibility over performance
  • Implementation can be further optimized
Related work

- Existing dynamic tainting approaches
  [Suh et al. 04, Newsome and Song 05, Halfond et al. 06, Kong et al. 06, ...]
- Ad-hoc

- Other dynamic taint analysis frameworks
  [Xu et al. 06 and Lam and Chiueh 06]
- Focused on security applications
  - Single taint mark
  - No control-flow propagation
  - Operate at the source code level
Conclusions

• Dytan
  • a general framework for dynamic tainting
  • allows for instantiating and experimenting with different dynamic taint analysis approaches

• Initial evaluation
  • flexible
  • easy to use
  • accurate
Future directions

• Tool release (documentation, code cleanup)
  http://www.cc.gatech.edu/~clause/dytan/
  (pre-release on request)
• Optimization (general and specific)
• Applications
  • Memory protection
  • Debugging
Questions?

http://www.cc.gatech.edu/~clause/dytan/