Dynamic taint analysis

Lujo Bauer
18-732
Spring 2015

[Slides from Michelle Mazeck]
What is dynamic taint analysis?

- Mark some data as tainted
- Track all future uses of that data
- Examples:
  - Untrusted input $\rightarrow$ control info (e.g., return addr)
  - Private data $\rightarrow$ outside parties (e.g., internet)
- Why dynamic?
  - You don’t always have source code
Example: Control flow

\[
x = \text{get\_input(keyboard)};
\]

\[
y = x + 180;
\]

\[
\text{goto } y;
\]
Introducing taint

\[ x = \text{get\_input(keyboard)}; \]

- Current taint status is boolean
- Updated with each dynamic step

\[ x \gets e \quad t = \text{isTainted(e)} \]

\[ \text{INPUT} \]

\[ \text{ASSIGN} \]
Propagating taint: Binary ops

\[ x = \text{get\_input}(\text{keyboard}); \]
\[ y = x + 180; \]

\[ t_1 = \text{isTainted}(e_1) \quad t_2 = \text{isTainted}(e_2) \]

\[ e_1 \circ e_2 \downarrow \quad t_1 \lor t_2 \]

- If either operand is tainted, result is tainted
Policy: Using taint

\[ x = \text{get}_\text{input}(\text{keyboard}); \]
\[ y = x + 180; \]
\[ \text{goto } y; \]

- On \text{goto}(e):
  \[ \text{assert(isTainted}(e) == \text{false}) \]
Challenge: Control-flow taint

- Example: SSN is tainted for privacy

```java
ssn = get_ssn();
//isTainted(ssn) = true;
email_to_me(ssn);
```

- What about:

```java
ssn = get_ssn();
if(ssn.starts_with(1)) {
    email_to_me(a);
} else (email_to_me(b));
```

Potential under-taint
Challenge: Sanitization

```c
x = get_input(keyboard);
x = x xor x;
goto(x + valid_pointer);
```

- XOR clears the value, but doesn’t remove taint
Challenge: Tainted addresses

- Consider:
  
  ```
  x = get_input(keyboard);
  y = mem[x];
  goto y;
  ```

- If `mem[x]` is clean, should `y` be tainted?
  - No! (But then the attacker is picking the address)
  - Yes! (But this could be user-selected options)
  - Neither answer is always right
Case Study: TaintDroid

- Enck et al., OSDI 2010
- Track privacy-sensitive information flow in Android apps
- At install time, user granted some access
  - How is that data used?
- **Must be lightweight**
  - Can’t disrupt user’s activities (use in real-time)
  - Phones are resource-constrained
How does it work?

- Use the VM to track \textit{variables} within apps
- Taint \textit{entire messages} between apps
- For library calls, propagate \textit{per method}
  - List mapping input taint status to output status
  - Default policy: union of argument taints
- \textbf{Keep one taint tag \textit{per file}}
  - Updated on file write
  - Data tainted on file read as appropriate
Taint sources and sinks

- **Low-bandwidth sensors (location, accel.)**
  - Taint at OS manager
- **High-bandwidth sensors (camera, mic)**
  - Taint at data buffer or file where captured
- **Databases (address book, SMS)**
  - Taint all such files
- **Device IDs**: Taint via API to access
- **Sink**: **Network interface** (native socket library)
Results

- Tested 30 popular apps that require location/camera/audio and internet
- Verified results by inspecting packet payload
  - Several sent phone IDs without EULA
  - 15 sent location to 3rd-party advertisers (no EULA)
  - No false positives from TaintDroid (despite very conservative approach)
Performance overhead

14% overhead microbenchmark
Summary: Dynamic taint analysis

- Trying to prevent information leaks *purely at run time*
- Hard to track information flow precisely
  - False positives, or false negatives, or both
- Efficient, prevents some leaks

... and is the topic of Assignment 5
Big Picture

- **Objective**
  - Design security into the language
  - Compiler rules out insecure programs
  - Compiler does not run the program (no testing)

- **What’s insecure?**
  - Buffer overflows
  - Information-flow leaks
  - Violations of access rights
  - ...

- **One approach is based on types**

Previously:
- Overview of types

Today:
- Non-interference (with types)
- *Dynamic* taint tracking

Later:
- Security-typed languages
- Typed assembly language
Acknowledgments

- Several slides adapted from David Brumley