CS 261: Systems Security

Taint Tracking

Oct 29, 2018

Prof. Raluca Ada Popa

Slides adapted from Univ of Michigan 583 Fall 12
Announcements

- Exam next Wednesday
  - Open book
  - All lectures except for this one
- Presenter: Pasin
- No writer, but we will post slides
Taint tracking

- A commonly used tool in systems security
- Helps track the flow of data through a program

In a nutshell:

- Data from sensitive sources (e.g., private or potentially malicious sources) is initially tainted
- Other data influenced by this data gets tainted too
Applications

Can be used in a number of ways:

- **Unknown Vulnerability Detection**
  - E.g. Taint Checking in Ruby and Perl
  - Any data input from an outside user is **tainted** (e.g., suspicious)
  - If it is used to set another variable, that gets tainted too
  - If a tainted variable gets used directly in a SQL query or a system call, flag as problematic

- **Malware Analysis**
  - What is the software doing with sensitive data?
  - Ex. TaintDroid
  - Any data from a **private** source (e.g., text messages) is **tainted**
  - Track where it is sent
Dynamic Taint Analysis

- Track information flow through a program at runtime

Set a taint tracking policy:

- Identify sources of taint – “TaintSeed”
  - What are you tracking?
    - Untrusted input
    - Sensitive data

- Taint Policy – “TaintTracker”
  - Propagation of taint

- Identify taint sinks – “TaintAssert”
  - Taint checking
    - Special calls
      - Jump statements
    - Outside network
Taint Analysis in Action
Example Policy

Taint seed example: Any input from an untrusted source is tainted

\[
\text{Input} \quad t = \text{IsUntrusted}(src) \\
\text{get\_input}(src) \downarrow t
\]

Taint tracker example: For a binary operation, the taint of the result is the OR of the taints of each operator input

\[
\text{BinOp} \quad t_1 = \tau[x_1], \quad t_2 = \tau[x_2] \\
x_1 + x_2 \downarrow t_1 \lor t_2
\]

Taint assert: Any goto statement can only go to a nontainted address

\[
P_{\text{goto}}(t_a) = \neg t_a
\]
(Must be true to execute)
x = get_input()
y = x + 42
...goto y

Input is tainted

\[ t = \text{IsUntrusted}(src) \]
\[ \text{get_input}(src) \downarrow t \]
\[ x = \text{get\_input}( ) \]

\[ y = x + 42 \]

\[ \text{goto } y \]

\[ \text{Data derived from user input is tainted} \]

\[ \begin{array}{c|c}
\text{Var} & \text{Val} \\
\hline
x & 7 \\
y & 49 \\
\end{array} \]

\[ \begin{array}{c|c}
\text{Var} & \text{Tainted?} \\
\hline
x & T \\
y & T \\
\end{array} \]
\( P_{\text{goto}}(t_a) = \neg t_a \)  
(Must be true to execute)
\[ x = \text{get\_input}() \]
\[ y = \ldots \]
\[ \ldots \]
\[ \text{goto } y \]

Helpful with buffer overflow:

\[
\ldots
\text{strcpy(buffer,argv[1])} ;
\ldots
\text{return ;}
\]
Pay attention to

- False Negatives
  - Use control flow to change value without gathering taint
    - Example: if (x == 0) y=0; else if (x == 1) y=1;
      - Equivalent to x=y;
  - Tainted index into a hardcoded table
    - Policy – value translation is not tainted
  - Hard to enumerating all sources of taint

- False Positives
  - Sanity Checks not removing taint
    - Requires fine-tuning
      - *Taint sanitization problem*
      - *Usually many and a lot of taint!*
How about loading from memory?
# Memory Load

## Variables

<table>
<thead>
<tr>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7</td>
</tr>
</tbody>
</table>

## Tainted?

<table>
<thead>
<tr>
<th>Var</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>T</td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>μ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>τμ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Problem: Memory Addresses

\[ x = \text{get\_input}() \]
\[ y = \text{load}(x) \]
\[ \ldots \]
\[ \text{goto } y \]

All values derived from user input are tainted??

<table>
<thead>
<tr>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
</tr>
</tbody>
</table>
Policy 1: Taint depends only on the memory cell

\[ v = \Delta[x] \quad \text{and} \quad t = \tau[\mu][v] \]

\[ \text{load}(x) \downarrow t \]

Undertainting
Failing to identify tainted values
- e.g., missing exploits

Taint Propagation

Var | Val
--- | ---
\( x \) | 7
\( 7 \) | 42

Addr | Val
--- | ---
\( 7 \) | F
Policy 2: If either the address or the memory cell is tainted, then the value is tainted.

x = get_input
y = load(jmp_table + x % 2)
…
goto y

Overtainting
Unaffected values are tainted
- e.g., flag exploits on safe inputs

Taint Propagation
Load
\[ v = \Delta[x], \ t = \tau_\mu[v], \ t_a = \tau[x] \]

\[ \text{load}(x) \downarrow t \ v \ t_a \]
General Challenge
State-of-the-Art is not perfect for all programs

Undertainting: Policy may miss taint

Overtainting: Policy may wrongly detect taint
Summary

- Taint tracking can be used to track flow of private data or suspicious inputs
- Further reading: *All You Ever Wanted to Know About Dynamic Taint Analysis*, Schwartz et al, Oakland 2010
- Next up: Pasin on TaintDroid