Taint tracking

Suman Jana
Dynamic Taint Analysis

• Track information flow through a program at runtime
• 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, Format strings, etc.
    • Outside network
TaintCheck (Newsome et al.)

- Performed on x86 binary
  - No need for source
- Implemented using Valgrind skin
  - X86 -> Valgrind’s Ucode
  - Taint instrumentation added
  - Ucode -> x86
- Sources -> TaintSeed
- Taint Policy -> TaintTracker
- Sinks -> TaintAssert
- Add on “Exploit Analyzer”
TaintCheck (Newsome et al.)

- **TaintSeed**: Mark untrusted data as tainted
- **TaintTracker**: Track each instruction, propagate taint
- **TaintAssert**: Check is tainted data is used dangerously
TaintSeed

• Marks any data from untrusted sources as “tainted”
  – Each byte of memory has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted
  – Else store a NULL pointer
TaintTracker

• Tracks each instruction that manipulates data in order to determine whether the result is tainted.
  – When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand.
TaintAssert & Exploit Analyzer

- **TaintAssert**
  - Checks whether tainted data is used in ways that its policy defines as illegitimate

- **Exploit Analyzer**
  - Backtrace chain of taint structures: provides useful information about how the exploit happened, and what the exploit attempts to do
    - Useful to generate exploit fingerprints
    - Transfer control to sandbox for analysis
Automatic Signature Generation

- Find value used to override return address – typically fixed value in the exploit code
Taint Analysis in Action
\[ x = \text{get_input}(\text{src}) \]
\[ y = x + 42 \]
\[ \text{goto } y \]

Input is tainted

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

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

\[ \begin{array}{c|c}
\text{Var} & \text{Tainted} \\
\hline
x & T \\
\end{array} \]
\[
x = \text{get_input}(\text{Devil})
\]

\[
y = x + 42
\]

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

\[
\text{TaintTracker}
\]

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

\[
\begin{array}{c|c}
\hline
\text{Var} & \text{Tainted} \\
\hline
x & T \\
y & T \\
\hline
\end{array}
\]

\[
\text{BinOp: } t_1 = \tau[x_1], \ t_2 = \tau[x_2]
\]

\[
x_1 + x_2 \downarrow t_1 \lor t_2
\]
\[
\begin{align*}
\text{Policy Violation Detected} \\
\text{tainted} & \quad \text{untainted} \\
\text{x} = \text{get\_input} \quad \text{Devil} \\
y = x + 42 \\
\cdots \\
goto \ y
\end{align*}
\]

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

\[
\begin{array}{c|c}
\text{Var} & \text{Tainted} \\
\hline
x & T \\
y & T \\
\end{array}
\]

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

y = …
…

goto  y

Jumping to overwritten return address

... strcpy(buffer,argv[1]) ;
... return ;
Policy Considerations?
## Memory Load

### Variables

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

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

... goto y

All values derived from user input are tainted??

<table>
<thead>
<tr>
<th>( \Delta )</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \mu )</th>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \tau_\mu )</th>
<th>Addr</th>
<th>Tainted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Policy 1: Taint depends only on the memory cell

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

Taint Propagation

Load $\nu = \Delta[x]$, $t = \tau_\mu[v]$
load(x) ↓ t

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

<table>
<thead>
<tr>
<th>Addr</th>
<th>Tainted</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Policy 2: If either the address or the memory cell is tainted, then the value is tainted.

\[
x = \text{get\_input}()
\]
\[
y = \text{load}(\text{jmp\_table} + x \mod 2)
\]

\[
\vdots
\]

\[
\text{goto } y
\]

Overtainting

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

Taint Propagation

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

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

Memory

Address expression is tainted

- printa
- printb
General Challenge: State-of-the-Art is not perfect for all programs

Undertainting: Policy may miss taint

Overtainting: Policy may wrongly detect taint
TaintCheck Evaluation
Effectiveness of TaintCheck

- **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
    - Enumerating all sources of taint

- **False Positives**
  - Vulnerable code?
  - Sanity Checks not removing taint
    - Requires fine-tuning
    - Taint sanitization problem
Effectiveness of TaintCheck

• Does TaintCheck raise false alerts for existing code?
  – network programs: apache, ATPhttpd, bftpd, cfingerd, and named
  – client programs: ssh and firebird
  – non-network programs: gcc, ls, bzip2, make, latex, vim, emacs, and bash

• Networked programs: 158K+ DNS queries
  – No false +ves

• All client and non-network programs (tainted data is stdin):
  – Only vim and firebird caused false +ves (data from config files used as offset to jump address)
TaintCheck - Attack Detection

- Synthetic Exploits
  - Buffer overflow -> function pointer
  - Buffer overflow -> format string
  - Format string -> info leak

- Actual Exploits
  - 3 real world examples
TaintCheck Performance

Performance overhead for Apache