EECS 583 – Class 21
Research Topic 3:
Dynamic Taint Analysis

University of Michigan

December 5, 2012
Announcements

❖ Exams returned
  » Answer Key can be found on the course website
  » One week to turn in written requests for exam regrades

❖ Sign up for project presentation slots
  » Friday on Scott’s office door, Monday in class

❖ Today’s class reading
  » "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,"
  » Optional background reading: "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask),"
EECS 583 Exam Statistics

Mean: 70.6  
StDev: 12.8  
High: 94  
Low: 44
The Problem

- **Unknown Vulnerability Detection**
  - Buffer Overflows
  - Format string vulnerabilities
  - The goal of TaintCheck

- **Malware Analysis**
  - What is the software doing with sensitive data?
  - Data propagation from triggers
  - Ex. TaintDroid

- **More Generally - Tracking the flow of data through a program**
Buffer Overflow

- Example Stack Buffer Overflow

```c
#include <string.h>

void foo (char *bar)
{
    char c[12];
    strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
    foo(argv[1]);
}
```
Buffer Overflow

- Unallocated Stack Space
  - Char c[12]
  - Char *bar
  - Saved Frame pointer
  - Return Address
  - Parent Routine’s Stack

- Address 0x80C03508

- Unallocated Stack Space
  - A
  - A
  - A
  - A

- Memory Addresses
  - Little Endian 0x80C03508
  - \x08 \x35 \xC0 \x80
Format String Vulnerability

- Attacker controls a format string

```c
printf("Number %d has no address, number %d has: %08x\n", i, a, &a);
```

From within the `printf` function the stack looks like:

```
stack top
...
<&a>
<a>
<i>
A
...
stack bottom
```

where:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>address of the format string</td>
</tr>
<tr>
<td>i</td>
<td>value of the variable i</td>
</tr>
<tr>
<td>a</td>
<td>value of the variable a</td>
</tr>
<tr>
<td>&amp;a</td>
<td>address of the variable i</td>
</tr>
</tbody>
</table>
Format String Vulnerability

- Attacker controls a format string
  » Overwrite arbitrary memory
    • Take control of program execution
    • %n commonly used in conjunction with other format specifiers
      - Writes number of bytes written thus far to the stack
  » Dump memory
    • ex. printf("%08x.%08x.%08x.%08x.%08x\n");
  » Crash the program
    • ex. printf("%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s\n");
    • Denial of Service
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
    - Outside network
TaintCheck

- 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”
Taint Analysis in Action
\[
\begin{align*}
\text{Input} & \quad t = \text{IsUntrusted}(src) \\
\text{get\_input}(src) & \quad \downarrow t
\end{align*}
\]

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

\[y = x + 42\]

... 

goto \ y

\(\Delta\)

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

\(\tau\)

<table>
<thead>
<tr>
<th>Var</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>T</td>
</tr>
</tbody>
</table>
x = get_input( )
y = x + 42

... goto y

Data derived from user input is tainted

TaintTracker

\[ t_1 = \tau[x_1] , t_2 = \tau[x_2] \]

\[ x_1 + x_2 \downarrow t_1 \lor t_2 \]

\[ \Delta \]

<table>
<thead>
<tr>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>7</td>
</tr>
<tr>
<td>y</td>
<td>49</td>
</tr>
</tbody>
</table>

\[ \tau \]

<table>
<thead>
<tr>
<th>Var</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>T</td>
</tr>
<tr>
<td>y</td>
<td>T</td>
</tr>
</tbody>
</table>
\[ x = \text{get\_input}( ) \]
\[ y = x + 42 \]

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

Policy Violation Detected

Tainted?

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

y = ...

goto y

Jumping to overwritten return address

... strcpy(buffer,argv[1]) ;
... return ;
TaintCheck - Exploit Analyzer

- TaintPolicy - Keep track of Taint propagation info
  - Backtrace chain of taint structures
  - Allow generation of attack signatures

- Transfer control to sandbox for analysis
TaintCheck - Evaluation

- **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
    - 0 false positives?

- **Thoughts?**
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>

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

### Memory

<table>
<thead>
<tr>
<th>μ</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>τ_μ</th>
<th>Addr</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Problem: Memory Addresses

\[
\begin{align*}
x &= \text{get_input}() \\
y &= \text{load}(x) \\
\ldots \\
goto y
\end{align*}
\]

All values derived from user input are tainted??

<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>
</tbody>
</table>

<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>Addr</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>
Policy 1: Taint depends only on the memory cell

\[ x = \text{get_input} \]

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

\[ \text{goto } y \]

**Undertainting**

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

**Taint Propagation**

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

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

\[ \tau_\mu \]

<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>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.

Taint Propagation:

\[ v = \Delta[x], \ t = \tau[\nu], \quad t_a = \tau[x] \]

Load: \[ \text{load}(x) \downarrow t' \nu \quad t_a \]

Overtainting:

Unaffected values are tainted - e.g., exploits on safe inputs.
General Challenge
State-of-the-Art is not perfect for all programs

Undertainting: Policy may miss taint

Overtainting: Policy may wrongly detect taint
TaintCheck - Attack Detection

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

- **Actual Exploits**  slightly more convincing
  - 3 real world examples
  - Random sample?
  - Prevalence of protected exploits?
Performance

- Implementation decisions
  - Use of Valgrind
  - Better on IO bound tasks
  - How much performance would you give up?

![Performance Overhead Factor Chart]

- CGI 6.63 ms, 1 KB 0.987 ms, 10 KB 2.05 ms, 100 KB 9.79 ms, 1 MB 86.4 ms, 10 MB 851 ms
TaintCheck Uses

- Individual Sites
  - Cost?
- Honeypots
- “TaintCheck plus OS Randomization”
  - Rely on OS randomization to cause crashes
  - Log and reproduce with tainting
- Sampling
  - Users rather than requests
    - Do I just need more infected computers?
  - Distributed Sampling
Signatures

- Automatic Semantic Analysis
  » Exploit Analyzer
- Generated signature validation
Other considerations

- Effectiveness in the wild
  - Side-channel attacks
    - Can the attacker determine when someone is using taint tracking?
      - Speed?
      - Perform the attack only on native systems
    - Similar to existing virtual machine and honeypot problems
  - Circumventing the taint system
    - Clean your data before exploit
      - Depends on policy
      - Example: Hex to ASCII translation
    - Cause false positives
      - Raises cost of running the system
      - Admins may turn it off
  - Difficult to evaluate without motivated attackers