Specification-Based Monitoring: Improving Model Precision

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
Wisconsin Safety Analyzer
Overview

• The need for specification-based monitoring
• Static binary analysis for intrusion detection
• Strengthening our analyses:
  - Interprocedural data flow analysis
  - General argument representation
  - Intelligent null call insertion
• Performance results
Worldview

- User desires to run program
- Running program makes operating system requests
- Attacker uses running program to generate malicious requests
Worldview

• **Attack goal:**
  - be creative...
  - Destruction
  - Information leaks
  - Service disruption

• **Attack technique:**
  - run arbitrary code in the user program
  - Buffer overrun
  - Virus or worm
  - Condor lurking jobs
Example: SQL Slammer

• Worm activated Saturday morning
  - Caused worldwide service disruption

• Propagation: exploited buffer overrun in Microsoft SQL Server to execute arbitrary code

• Detection: SQL Server makes unexpected system calls
  - Arbitrary code differs from SQL code
Example: The Condor Attack

- Users dispatch programs for remote execution
- Remote jobs send critical system calls back to local machine for execution
Example: The Condor Attack

- Attackers can manipulate remotely executing program
- Insert arbitrary code that takes control of link to user’s machine

![Diagram of a submitting host, operating system, shadow process, and user program.]
Example: The Condor Attack

• Detection: Remote user job makes unexpected remote system calls
  - Arbitrary code differs from job code
Our Objective

• Detect malicious activity before harm caused to local machine

• ... before operating system executes malicious system call
Our Objective

- Snort
  - Detection at service interface: limited to network-based attacks

- Our work
  - Detection at system call interface makes our work independent of intrusion technique
Specification-Based Monitoring

• Specify constraints upon program behavior
  - Static analysis of binary code
  - Construct automaton modeling all system call sequences the program can generate

• Ensure execution does not violate specification
  - Operate the automaton
  - If no valid states, then intrusion attempt occurred
Specification-Based Monitoring

User Program

Analyzer

Rewritten Binary

Runtime Monitor
Specification-Based Monitoring

User Program

Analyzer

Rewritten Binary

Runtime Monitor
Specification-Based Monitoring

User Program

Analyzer

Rewritten Binary

Runtime Monitor
Specification-Based Monitoring

- Rewritten Binary
- Runtime Monitor
- Event Interface
- User Program
- Operating System
Specification-Based Monitoring

- Our runtime monitor monitors program execution at the event interface layer.
- Ensures program events match specification.
Specification-Based Monitoring

- Our runtime monitor monitors program execution at the event interface layer.
- Ensures program events match specification.
- Runtime monitor must be part of trusted computing base.
Specification-Based Monitoring

- Event interface defines observable events
- Observed events may be superset of system calls
- Expand interface between program and monitor
  - Call-site renaming
  - Null calls
Specification-Based Monitoring

- **Expanded** set of observable events
  - More precise program modeling
  - More efficient model operation
- User program rewritten to use expanded interface
Observable Events

• Initial interface
  - System call names
  - Add system call arguments
  - Add system call return values

• Expanded interface
  - Add null call names
  - Add null call arguments
Model Construction

User Program

Analyzer

Rewritten Binary

Runtime Monitor

Binary Program

Control Flow Graphs

Local Automata

Global Automaton
Code Example

```c
int wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
```

```asm
wrapper:
    save %sp, -0x96, %sp
    cmp %i1, 0
    beq,a L1
    mov 1, %o1
    mov 3, %o1
    L1:
    call open
    mov %i0, %o0
    ret
    restore %o0, %g0, %o0
```
CFG Construction

- **Executable Editing Library (EEL)**
  - Parses SPARC binary programs
  - Identifies functions
  - Constructs control flow graphs (CFG)
  - Rewrites binary program
int
wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
Attack Restriction

• Only sequences of system calls in automata are accepted
  - Attacks that do not match these sequences will fail
**Argument Manipulation**

- **Shortcoming**: attacker can specify any arguments

- **Goal**: include system call arguments
  - Restricts opportunities for attacker to cause harm

- **Argument capture** is the analysis technique that statically identifies arguments
Argument Capture

```c
int
wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
```

**Problem:** Interprocedural data flow
int
wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
Improving Data Flow Analysis

• Need for data flow analysis
  - Argument capture
  - Identifying targets of indirect transfers

• EEL provides backward register slicing
Improving Data Flow Analysis

EEL

- Intraprocedural
- Tree based
- Recovers constants
Improving Data Flow Analysis

• New infrastructure component: data dependence graph
  - Subgraph of program dependence graph
  - Collection of expression graphs that set register values
  - Includes interprocedural data flows
Data Dependence Graph

- Nodes are individual instructions
- 3 edge types connect instructions $I \rightarrow J$:
  - Intraprocedural: $I$ writes a data value that $J$ reads
  - Call arguments: $I$ sets an actual argument to a call, $J$ reads the formal argument
  - Call return: $I$ sets a call return value, $J$ reads the return value
int
wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
Argument Capture

```c
int wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
```
```c
int wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}

void program (char *file)
{
    int fd1, fd2;
    char *name = "myfile";
    fd1 = wrapper(name, true);
    fd2 = wrapper("/etc/passwd", false);
    close(fd1);
    close(fd2);
}
```
int wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
void
program (char *file)
{
    int fd1, fd2;
    char *name = "myfile";

    fd1 = wrapper(name, true);
    fd2 = wrapper("/etc/passwd", false);

    close(fd1);
    close(fd2);
}
DDG Construction

name = "myfile"

file = name

file = "/etc/passwd"

file = {"myfile", "/etc/passwd"}

call wrapper

mode = O_RDONLY

mode = O_RDWR

mode = O_RDWR

call open

file = 

1 August 2003 WiSA - Jonathon Giffin
int
wrapper (char *file, bool wr)
{
    mode_t mode;
    if (wr)
        mode = O_RDWR;
    else
        mode = O_RDONLY;
    return open(file, mode);
}
Argument Capture

• **Goal reached**
  - All arguments recovered

```c
open({"myfile", "/etc/passwd"}, {1,3})
```
Data Dependence Graph

• Foundation of all data flow analyses used by the binary analyzer
  - Identification of indirect transfer targets
    • Call graph construction
  - Argument capture
Improving Argument Capture

• Using EEL’s analysis, argument capture treated each value as an integer
• We can do more: general notion of argument types
  - Integer
  - Set of integers
  - Return values of previous system calls
  - Regular expressions for string arguments
  - Arguments passed to library functions
void

program (char *file)
{
    int fd1, fd2;
    char *name = "myfile";

    fd1 = wrapper(name, true);
    fd2 = wrapper("/etc/passwd",
                   false);

    close(fd1);
    close(fd2);
}
Regular Expressions

```c
void
program (char *file)
{
    int fd1, fd2;
    char *name = "myfile";

    fd1 = wrapper(name, true);
    fd2 = wrapper("/etc/passwd", false);

    close(fd1);
    close(fd2);
}
```
void program (char *file)
{
    int fd1, fd2;
    char *name =
        strcat(file, "\.txt");
    fd1 = wrapper(name, true);
    fd2 = wrapper("/etc/passwd",
                   false);
    close(fd1);
    close(fd2);
}
Improving Argument Capture

**EEL**
- Intraprocedural
- Tree based
- Recovers constants

**Data Dependence Graph**
- Interprocedural
- Graph based
- Recovers constants, sets, return values, regular expressions, and arguments
Measurements

• **Number of arguments recovered**
  - Intrprocedural, constants
  - Interprocedural, constants
  - Interprocedural, general representation
Test Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Program Size in Instructions</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>56,686</td>
<td>Compress a 13 MB file</td>
</tr>
<tr>
<td>GNU finger</td>
<td>95,534</td>
<td>Finger 3 non-local users</td>
</tr>
<tr>
<td>procmail</td>
<td>107,167</td>
<td>Process 1 incoming email message</td>
</tr>
</tbody>
</table>
## Results

- **Number of recovered arguments:**

<table>
<thead>
<tr>
<th>Technique</th>
<th>gzip</th>
<th>finger</th>
<th>procmail</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intraprocedural, constant</strong></td>
<td>30</td>
<td>149</td>
<td>206</td>
</tr>
<tr>
<td><strong>Interprocedural, constant</strong></td>
<td>50</td>
<td>157</td>
<td>212</td>
</tr>
<tr>
<td><strong>Interprocedural, general</strong></td>
<td>81</td>
<td>227</td>
<td>271</td>
</tr>
</tbody>
</table>
Improving Argument Capture

• Restrictions the attacker because the specification limits acceptable arguments
Smart Null Call Insertion

- **Null calls** are dummy system calls
  - Part of the **expanded interface**
  - Used by the monitor to update the model
  - Do not cross the interface to the operating system
Smart Null Call Insertion

• Previous experiments have shown:
  – Null calls improve precision
    • Restrict paths followed in automaton
  – Null calls improve performance of push-down automaton models

• Prior work used naïve null call placement
  – Based solely upon the fan-in of the function
  – Fan-in is a crude approximation of precision gain
Smart Null Call Insertion

- New algorithm places null calls based upon expected precision gain
- Expected effect:
  - Greater precision gains at less cost
Smart Null Call Insertion

• Precision metric: **average branching factor**

• Lower values indicate greater precision
Smart Null Call Insertion

• Statically compute branching factor at function entry points
• Instrument functions that contribute most to average branching factor

• **Effect**: See best precision improvement with fewest null call insertions
Smart Null Call Insertion

- Measurements: Comparison to naïve insertion
  - Monitoring overhead
  - Precision gain versus null call overhead
Null Calls Monitoring Overhead

• Naïve insertion:

<table>
<thead>
<tr>
<th>Insertion Rate</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>747.0 %</td>
<td>&lt; 0.1 %</td>
<td>&lt; 0.1 %</td>
</tr>
<tr>
<td>GNU finger</td>
<td>0.1 %</td>
<td>0.1 %</td>
<td>&lt; 0.1 %</td>
</tr>
<tr>
<td>procmail</td>
<td>0.8 %</td>
<td>1.1 %</td>
<td>0.7 %</td>
</tr>
</tbody>
</table>

• Intelligent insertion:
  - **Miniscule!** Lost in measuring noise.
Null Call Precision vs. Overhead: procmail

- Naive
- Smart
- No Instrumentation

Average Branching Factor vs. Number of Null Calls Executed
Null Call Precision vs. Overhead: finger

- Naive
- Smart
- No Instrumentation
Important Ideas

• We develop specification-based monitoring techniques using static binary analysis to detect intrusions.

• Interprocedural slicing improves argument capture by including data flow information.

• Intelligent null call insertion maximizes precision gain with minimal performance impact.
Technical Agenda

• Integrating other specification sources
• C++ vtable analysis
• Intelligent null call insertion
• Interprocedural data flow analysis
• General argument representation
Technical Agenda

• Integrating other specification sources
• C++ vtable analysis
• Construction of accurate models for dynamically linked applications
• Abstract stack representations for PDA models
Specification-Based Monitoring: Improving Model Precision

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin
Wisconsin Safety Analyzer