Specification-Based Analysis and Enforcement

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

• Intrusion detection and specification-based monitoring
• An unusual intrusion path
  - The Condor attack: How to easily do dangerous and malicious things to a running job
• How to detect attempted intrusions with pre-execution static analysis and runtime monitoring
• Precision & performance results for 3 programs
• Recent work
  - Null call insertion to improve precision & performance
  - Analysis of shared objects
Intrusion Detection

Goal: Discover attempts to maliciously gain access to a system
Goal: Discover attempts to maliciously gain access to a system

<table>
<thead>
<tr>
<th>Misuse Detection</th>
<th>Specification-Based Monitoring</th>
<th>Anomaly Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Specify patterns of attack or misuse</td>
<td>• Specify constraints upon program behavior</td>
<td>• Learn typical behavior of application</td>
</tr>
<tr>
<td>• Ensure misuse patterns do not arise at runtime</td>
<td>• Ensure execution does not violate specification</td>
<td>• Variations indicate potential intrusions</td>
</tr>
<tr>
<td>• Snort</td>
<td>• Our work; Ko, et. al.</td>
<td>• IDES</td>
</tr>
<tr>
<td>• Rigid: cannot adapt to novel attacks</td>
<td>• Specifications can be cumbersome to create</td>
<td>• High false alarm rate</td>
</tr>
</tbody>
</table>
Specification-Based Monitoring

• Two components:
  
  - **Specification**: Indicates constraints upon program behavior
  
  - **Enforcement**: How the specification is verified at runtime or from audit data
Analyst or Administrator

Training Sets

Static Source Code Analysis

Static Binary Code Analysis

Specification

Enforcement

Execution Obeys Static Ruleset

Execution Matches Model of Application
Representative Work by Ko, et. al.

• **Specification**: Programmers or administrators specify correct program behavior

```prolog
PROGRAM fingerd
    read(X) :- worldreadable(X);
    bind(79);
    write("/etc/log");
    exec("/usr/ucb/finger");
END
```

• **Enforcement**: At runtime, only allow actions that match the specified policy
Specification

Enforcement

Execution
Obeys Static
Ruleset

Execution
Matches
Model of
Application

Analyst or Administrator

Training Sets

Static Source Code Analysis

Static Binary Code Analysis
Our Approach

**Specification**: Static analysis of binary code

- Specifications are automatically generated
- Not reliant upon programmers to produce accurate specifications
- Analyzes all execution paths
- Source code may be unavailable

```assembly
function:
    save %sp, 0x96, %sp
    cmp %i0, 0
    bge L1
    mov 15, %o1
    call read
    mov 0, %o0
    call line
    nop
    b L2
    nop
L1:
    call read
    mov %i0, %o0
    call close
    mov %i0, %o0
L2:
    ret
    restore
```
Our Approach

**Enforcement**: Operate an automaton modeling correct system call sequences

- Dynamic ruleset
- More expressive than static ruleset of Ko, et. al.
Non-Deterministic Finite Automaton (NFA)

- **Structure**
  - States
  - Labeled edges between states

- **Edge labels are input symbols - call names**

- **Path to any accepting state defines valid sequence of calls**
Our Approach

**Enforcement:** Operate an automaton modeling correct system call sequences

- Dynamic ruleset
- More expressive than static ruleset of Ko, et. al.
Example: The Condor Attack

- Users dispatch programs for remote execution
- Remote jobs send critical system calls back to local machine for execution

![Diagram showing the Condor Attack process](image)
Example: The Condor Attack

- Attackers can manipulate remotely executing program to gain access to user’s machine

![Diagram showing the Condor Attack process with labels for Shadow Process, system calls, User Job, and Execution Host.](Image)
A New View

• Running programs are objects to be easily manipulated

• The vehicle: the **DynInst API**
DynInst: Dynamic Instrumentation

• Machine independent library for instrumentation of running processes
• Modify control flow of the process:
  - Load new code into the process
  - Remove, replace, or redirect function calls
  - Asynchronously call any function in the process
Condor Attack: Lurking Jobs

- **Shadow Process**
  
  - *giffin*

- **Submitting Host**

- **Execution Host**

  - **Malicious User Job**
    
    - *nobody*

  - **system calls**
Condor Attack: Lurking Jobs

Shadow Process: `giffin`

Malicious User Job: `nobody`

Lurker Process: `nobody`

Submitting Host

Execution Host

System calls

Fork
Condor Attack: Lurking Jobs

Lurker Process

nobody
Condor Attack: Lurking Jobs

Shadow Process: bart

Submitting Host

system calls

Innocent User Job: nobody

Execution Host

Lurker Process: nobody
Condor Attack: Lurking Jobs

Submitting Host

Shadow Process

\textit{bart}

system calls

Execution Host

Innocent User Job

\textit{nobody}

Lurker Process

\textit{nobody}

attach

attach
Condor Attack: Lurking Jobs

- Shadow Process: *bart*
- Innocent User Job: *nobody*
- Lurker Process: *nobody*

**Submitting Host**

**Execution Host**

- System calls
- Control remote system calls
- Attach

4 August 2003
Condor Attack: Lurking Jobs

- **Shadow Process**
  - `bart`
  - `rm -rf *`

- **Innocent User Job**
  - `nobody`

- **Lurker Process**
  - `nobody`

- **Submitting Host**

- **Execution Host**
  - System calls
  - Control remote system calls
  - Attach

4 August 2003
WiSA - Jonathon Giffin
Can We Safely Execute Our Jobs Remotely?

The threats:
1. Cause the job to make improper remote system calls.
2. Cause the job to calculate an incorrect answer.
3. Steal data from the remote job.

Threat protection strategies:
- Monitor execution of remote job (threat #1)
- File or system call sand-boxing (#1)
- Obfuscate or encode remote job (#1, #3)
- Replicate remote job (#2)
Countering Remote Attacks

• **Goal:** Even if an intruder can see, examine, and fully control the remote job, no harm can come to the local machine.

• **Method:** Model all possible sequences of remote system calls. At runtime, update the model with each received call.

• **Key technology:** Static analysis of binary code.
Execution Monitoring

User Job

Analyzer

Checking Shadow

Modified User Job
Execution Monitoring

Submitting Host

Checking Shadow

Job Model

Modified User Job

Execution Host

system calls
Execution Monitoring

Submitting Host  
Checking Shadow  
Job Model

Modified User Job

Execution Host

system calls  
Call 3
Model Construction

- User Job
- Analyzer
- Checking Shadow
- Modified User Job

Binary Program ➔ Control Flow Graphs ➔ Local Automata ➔ Global Automaton
The Binary View (SPARC)

function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore

function (int a) {
  if (a < 0) {
    read(0, 15);
    line();
  } else {
    read(a, 15);
    close(a);
  }
}
Control Flow Graph Generation

function:
    save %sp, 0x96, %sp
    cmp %i0, 0
    bge L1
    mov 15, %o1
    call read
    mov 0, %o0
    call line
    nop
    b L2
    nop
L1:
    call read
    mov %i0, %o0
    call close
    mov %i0, %o0
L2:
    ret
    restore
Control Flow Graph Translation

CFG ENTRY

bge

call read

call close

call line

ret

CFG EXIT

read read

close line
Control Flow Graph Translation

CFG ENTRY

bge

call read

call read

call close

call line

ret

CFG EXIT

read read

close line

Interprocedural Model Generation
Interprocedural Model Generation
Interprocedural Model Generation

A

read

read

close

line

B

line

write

close
Interprocedural Model Generation

A

read
read

ε

B

line

line

write

close
close

e

e

4 August 2003

WiSA - Jonathon Giffin
Interprocedural Model
Generation

A

read

read

ε

close

ε

ε

ε

ε

write

ε

ε

close

B
Possible Paths
Possible Paths

A

read

read

close

4 August 2003

WiSA - Jonathon Giffin

40
Impossible Paths

A

read

read

ε

line

write

ε

ε

ε

B

close

close
Impossible Paths

A

read

read

close

line

write

close

B

ε

ε

ε

ε

ε

4 August 2003

WiSA - Jonathon Giffin
PDA State Explosion

• ε-edge identifiers maintained on a stack
  - Stack may grow to be unbounded

• Solution:
  - Bound the maximum size of the runtime stack
  - A regular language overapproximation of the context-free language of the PDA
Data Flow Analysis

Argument recovery

- Statically known arguments constrain remote calls
- Reduces opportunity given to attackers

function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
Rewriting User Job

Binary Program → Rewritten Binary

User Job → Analyzer → Checking Shadow → Modified User Job
Call Site Renaming

function:
    save %sp, 0x96, %sp
    cmp $i0, 0
    bge L1
    mov 15, %o1
    call read
    mov 0, %o0
    call line
    nop
    b L2
    nop
L1:
    call read
    mov %i0, %o0
    call close
    mov %i0, %o0
L2:
    ret
    restore

• Give each monitored call site a unique name
• Associates arguments with call sites
• Obfuscation
• Reduces nondeterminism
Call Site Renaming

function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism
Call Site Renaming

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call _83
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore
Call Site Renaming

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism

function:
  save %sp, 0x96, %sp
  cmp $i0, 0
  bge L1
  mov 15, %o1
  call _638
  mov 0, %o0
  line
  nop
  L2:
    b L2
  nop
L1:
  call _83
  mov %i0, %o0
  call _1920
  mov %i0, %o0
L2:
  ret
  restore
Call Site Renaming

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism
Call Site Renaming

- Give each monitored call site a unique name
- Associates arguments with call sites
- Obfuscation
- Reduces nondeterminism
Prototype Implementation

- Simulates remote execution environment
- Measure model precision
- Measure runtime overheads
- Measure the effect of changing maximum stack depth on bounded PDA model
## Test Programs

<table>
<thead>
<tr>
<th>Program Size in Instructions</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Compress a 13 MB file</td>
</tr>
<tr>
<td>GNU finger</td>
<td>Finger 3 non-local users</td>
</tr>
<tr>
<td>procmail</td>
<td>Process 1 incoming email message</td>
</tr>
</tbody>
</table>
Precision Metric

• Average branching factor

• Lower values indicate greater precision
Optimizations Improve Precision

Program

<table>
<thead>
<tr>
<th>Program</th>
<th>None</th>
<th>Rename</th>
<th>Argument Capture</th>
<th>Rename+Capture</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNU finger</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>procmail</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average Branching Factor
PDA Precision Improves with Increased Stack Depth (procmail)
PDA Overhead (procmail)

Stack Bound

Overhead (seconds)

0 1 2 3 4 5 6 7 8 9 10

0 100 200 300 400 500 600 700 800

Rename+Capture
Recent Work

• Improving precision with null calls
  – **Surprise!** PDA performance improves

• Analysis of shared objects
Null Calls

- Observation: PDA is more precise than NFA because it provides context sensitivity

- Idea: Insert null calls into NFA model to add some context sensitivity without suffering runtime cost of PDA
Null Call Insertion
Null Call
Insertion
Null Call Experiments

• Inserted null calls at 3 rates
  - High: At entries of functions with fan-in of 2 or greater
  - Medium: At entries of functions with fan-in of 5 or greater
  - Low: At entries of functions with fan-in of 10 or greater
Precision Improves with Null Calls

![Bar chart showing average branching factor for different programs and insertion rates.]

- **gzip**: Low Insertion Rate - Rename+Capture
- **GNU finger**: Medium Insertion Rate
- **procmail**: High Insertion Rate - Rename+Capture
## Null Call Costs: Monitoring Overhead & Bandwidth

<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>

<table>
<thead>
<tr>
<th>Tool</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>4350.0 Kbps</td>
<td>5.6 Kbps</td>
<td>0.0 Kbps</td>
</tr>
<tr>
<td>GNU finger</td>
<td>14.1 Kbps</td>
<td>9.1 Kbps</td>
<td>0.9 Kbps</td>
</tr>
<tr>
<td>procmail</td>
<td>18.2 Kbps</td>
<td>13.1 Kbps</td>
<td>4.0 Kbps</td>
</tr>
</tbody>
</table>
PDA Precision Improves With Null Call Insertion & Increased Stack Depth (procmail)
Analyzing Shared Object Code

- Two new difficulties
  - Relocatable object code
  - Interprocedural data flows
Relocatable Object Code

- Data tables filled out dynamically at load time
- Data table recovery
  - Recover relocation tables
  - Simulate action of run-time linker to resolve table values
- Enables improved analysis
  - Trace global data accesses
  - Follow jumps through table values
Data Flow Analysis

Argument recovery technique

• Slice on each register of interest to build a data dependence graph for the value

• Simulate the execution of the instructions in the dependence graph to reach final value
Argument Recovery

function:
  save %sp, 0x96, %sp
  cmp %i0, 0
  bge L1
  mov 15, %o1
  call read
  mov 0, %o0
  call line
  nop
  b L2
  nop
L1:
  call read
  mov %i0, %o0
  call close
  mov %i0, %o0
L2:
  ret
  restore

What happens here?

Entry Point

mov %i0, %o0  mov 15, %o1
call read
Argument Recovery

Interprocedural Slicing

- Continue slice in calling functions

```asm
mov %i0, %o0  mov 15, %o1
```

Entry Point

call read
Argument Recovery

Call Site -> Entry Point

Entry Point

Call Site

mov %i0, %o0
mov 15, %o1
call read
Argument Recovery

... Call Site ... Call Site ...

Call Site ... Call Site ...

Entry Point

mov %i0, %o0
mov 15, %o1
call read
Argument Recovery

• Interprocedural slicing improves argument recovery
  - Imposes greater constraints upon attacker

• In shared objects, we can recover function pointers passed to library calls
  - Improves model precision
Analyzing Shared Object Code

Infrastructure Changes

• Both relocation table analysis & interprocedural slicing required modification of the analysis infrastructure

Status

• Recovering relocation tables is complete
• Interprocedural slicing is underway
Important Ideas

• Our work is specification-based monitoring with specifications generated automatically from binary code analysis.

• We enforce the specification by operating a finite state machine modeling correct execution.

• Null calls improve precision & PDA performance.

• Shared object analysis required addition of capabilities to the infrastructure.
Technical Agenda

• Integrating other specification sources
• Optimal null call insertion
• C++ vtable analysis
Specification-Based Analysis and Enforcement

Jonathon Giffin, Somesh Jha, Barton Miller
University of Wisconsin