Attacks and Defenses

Barton Miller
Jonathon Giffin, Somesh Jha
University of Wisconsin
{bart, giffin, jha}@cs.wisc.edu

WiSA - Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa
Overview

Attacks
- Server attack (conventional host-based IDS)
- Remote execution attack (remote IDS)

Model-based intrusion detection
- Constructing program models using static binary analysis
- Accuracy/performance tradeoff in prior models
- Our new Dyck model solves tradeoff
- Data-flow analysis to recover arguments

Milestones
Worldview

- Running processes make operating system requests
- Changes to trusted computing base done via these requests
- Attacker subverts process to generate malicious requests
Example: Server Attack

• Goal: Execute malicious code in the server
Example: Remote Execution Attack

Shadow Process

giffin

-- system calls --

User Process

nobody

Submitting Host

Remote Execution Host

Trusted computing base
Example: Remote Execution Attack

- **Shadow Process**: giffin
- **Submitting Host**: Trusted computing base
- **Malicious User Process**: nobody
- **Remote Execution Host**: fork

The process involves the shadow process on the submitting host initiating system calls to a malicious user process on the remote execution host, which results in the execution of a lurking process with nobody permissions.
Example: Remote Execution Attack

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

**Submitting Host**
- Trusted computing base

**Remote Execution Host**

**Innocent User Process**
- **nobody**

**Lurker Process**
- **nobody**

- `attach`
- `Control remote system calls`
- `system calls`

- `rm -rf *`
Our Objective

- Detect malicious activity before harm caused to local machine
- ... before operating system executes malicious system call
Model-Based Intrusion Detection

- Build model of correct program behavior
- Runtime monitor ensures execution does not violate model
- Runtime monitor must be part of trusted computing base

User Process

Operating System

Trusted computing base
Automated Model Construction

- **Dynamic analysis**
  - Under-approximates correct behavior
  - False alarms
  - Forrest, Sekar, Lee

- **Static analysis**
  - Over-approximates correct behavior
  - False negatives
  - Wagner&Dean, our work
  - Previous attempts at precise models problematic
Automated Model Construction

- Static analysis challenge
  - Design an efficient, context-sensitive model

- Answers
  - Dyck model
  - Argument dependency recovery
Our Approach

• **Build model of correct program behavior**
  - Static analysis of binary code
  - Construct an automaton modeling all system call sequences the program can generate

• **Ensure execution does not violate model**
  - Use automaton to monitor system calls.
  - If automaton reaches an invalid state, then an intrusion attempt occurred.
Model-Based Intrusion Detection

User Program

Analyzer

Rewritten Program
Model-Based Intrusion Detection

- Build model of correct program behavior

- Runtime monitor ensures execution does not violate model

- Runtime monitor must be part of trusted computing base
Model Construction

Rewritten Program

Program Model

Analyzer

User Program

Binary Program

Control Flow Graphs

Local Automata

Global Automaton
Code Example

```
void
link_wrap(char *f, char *t)
{
    char msg[BUFFSIZE];
    unlink(t);
    link(f, t);
    snprintf(msg, BUFFSIZE, "Linked %s to %s, f, t);
    log(msg);
}
```

```
link_wrap:
    save %sp, -596, %sp
    call unlink
    mov %i1, %o0
    mov %i1, %o1
    call link
    mov %i0, %o0
    add %sp, 56, %o0
    mov 50, %o1
    sethi %hi(str), %o2
    call snprintf
    or %o2, %lo(str), %o2
    call log
    add %sp, 56, %o0
    ret
    restore
```
Local Automaton

```
void
link_wrap(char *f, char *t)
{
    char msg[BUFFSIZE];
    unlink(t);
    link(f, t);
    snprintf(msg, BUFFSIZE, "Linked %s to %s, f, t;"
    log(msg);
}
```
unlink("/sbin/mailconf");
link("/bin/sh", "/sbin/mailconf");
write(-1, 0, 0);
exec("/sbin/mailconf");
Adding Context Sensitivity

- Model call & return behavior of function calls
- Use pushdown automaton (PDA) stack to model program’s call stack
- Model is sensitive to calling context of each system call
PDA Model

link_wrap

unlink(?)

link(?,?)

push X

log

write(?,,?)

push Y

exec_wrap

stat("/sbin/mailconf")

e

e

e

e

push X

ed

pop X

pop Y

exec("/sbin/mailconf")

ed

13 November 2003

WiSA - Barton P. Miller
PDA State Explosion

- e-edge identifiers maintained on a stack
  - Stack non-determinism is expensive
  - Unbounded stacks add complexity
  - Best-known algorithm: cubic in automaton size

- Unusable as program model
  - Orders of magnitude slowing of application
    - [Wagner et al. 01, Giffin et al. 02]
  - Conclusion: only weaker NFA models have reasonable performance
Dyck Model

• Efficiently tracks calling context

• As powerful as full PDA
• Efficiency approaches NFA model

• Implication: **accuracy & performance can coexist**
  - Invalidates previous conclusion
Dyck Model

• Bracketed context-free language
  - [Ginsberg & Harrison 67]

\[
\text{stat } [y \text{ write } ]_y \text{ exec }
\]
\[
\text{unlink link } [x \text{ write } ]_x
\]

• Matching brackets are alphabet symbols
  - Exposes stack operations to runtime monitor
  - Rewrite binary to generate bracket symbols
  - [Giffin et al. 04]
Binary Rewriting

User Program

Analyzer

Rewritten Program

Program Model

Binary Program

Rewritten Binary
Binary Rewriting

• Insert code to generate bracket symbols around function call sites

• Notify monitor of stack activity

```c
void link_wrap(char *f, char *t)
{
    char msg[BUFFSIZE];
    unlink(t);
    link(f, t);
    snprintf(msg, BUFFSIZE,
        "Linked %s to %s, f, t);
leftX();
log(msg);
rightX();
}
```
Data-Flow Analysis

• Can use knowledge of argument values to make model more precise.
• Use data-flow analysis of arguments:
  - Argument recovery
    • Sets of constant values
    • Sets of regular expression strings
  - Argument dependencies upon system call return values
  - System call return values that control branching
Argument Dependencies

\[ \text{fd1} = \text{open(“/home/foo”, O_RDWR);} \]
\[ \text{fd2} = \text{open(“/etc/passwd”, O_RDWR);} \]
\[ \text{read(fd2, buf, BUFSIZE);} \]
\[ \text{write(fd1, buf, BUFSIZE);} \]

open\(_1(“/home/foo”, \text{O\_RDWR})\)
open\(_2(“/etc/passwd”, \text{O\_RDWR})\)
read(=open\(_2, ? , \text{BUFSIZE}\))
write(=open\(_1, ?, \text{BUFSIZE}\))

open\(_1() = 3; \]
ope \(_2() = 4; \]
# Test Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>107,246</td>
</tr>
<tr>
<td>gzip</td>
<td>56,710</td>
</tr>
<tr>
<td>eject</td>
<td>70,177</td>
</tr>
<tr>
<td>fdformat</td>
<td>67,874</td>
</tr>
<tr>
<td>cat</td>
<td>54,028</td>
</tr>
</tbody>
</table>
## Runtime Overheads

Execution times in seconds

<table>
<thead>
<tr>
<th>Program</th>
<th>Base</th>
<th>NFA</th>
<th>Increase</th>
<th>Dyck</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>procmail</td>
<td>0.42</td>
<td>0.37</td>
<td>0%</td>
<td>0.40</td>
<td>0%</td>
</tr>
<tr>
<td>gzip</td>
<td>7.02</td>
<td>6.61</td>
<td>0%</td>
<td>7.16</td>
<td>2%</td>
</tr>
<tr>
<td>eject</td>
<td>5.14</td>
<td>5.17</td>
<td>1%</td>
<td>5.22</td>
<td>2%</td>
</tr>
<tr>
<td>fdformat</td>
<td>112.41</td>
<td>112.36</td>
<td>0%</td>
<td>112.38</td>
<td>0%</td>
</tr>
<tr>
<td>cat</td>
<td>54.65</td>
<td>56.32</td>
<td>3%</td>
<td>80.78</td>
<td>48%</td>
</tr>
</tbody>
</table>
Accuracy Metric

- Average branching factor
NFA and Dyck Model Accuracy

Average Branching Factor

- procmail
- gzip
- eject
- fdformat
- cat

- NFA
- Dyck
Important Ideas

• Model-based intrusion detection forces execution behavior to match model.

• Statically constructed program models historically compromise accuracy for efficiency.

• The Dyck model is the first efficient context-sensitive specification.
Milestones

Nov 01:
• NFA & PDA construction
• Call-site renaming

Jan 02:
• Bounded stack PDA
• Static argument recovery
• Binary code instrumentation

Feb 04:
• NDSS Paper

March 03:
• Invented Dyck model

Aug 02:
• USENIX Security paper
• Interprocedural data-flow analysis
• Argument dependency recovery

Oct 02:
• Data-flow-sensitive analysis

Jan 03:
• Smart binary code instrumentation

May 02:
• Began shared object analysis

July 03:
• Implemented kernel trap monitoring
Milestones

• Two conference papers


Milestones

Feb 04:
• Support dynamic linking

June 04:
• Move to BREW infrastructure

July 04:
• Recover arguments at additional program points

Ongoing 2004:
• Formal modeling of attacks & defenses
• Investigating tool to construct attacks for better IDS testing

Ongoing 2004:
• Investigating hybrid static/dynamic techniques
Collaboration with Wenke Lee

• Collaborated on static version of his dynamic analysis work
  - Compared with our Dyck model
  - Developed static model formalisms
  - Under submission: “Formalizing Sensitivity in Static Analysis for Intrusion Detection”

• Future: research hybrid techniques
  - New methods to recover calling context
  - Combine static & dynamic analysis
Attacks and Defenses

Barton Miller
Jonathon Giffin, Somesh Jha
University of Wisconsin
{bart,giffin,jha}@cs.wisc.edu

WiSA – Wisconsin Safety Analyzer
http://www.cs.wisc.edu/wisa