Outline

- Introduction
- Motivation
- Models
  - Trivial model
  - Callgraph model
  - Abstract stack model
  - Digraph model
- Implementation
- Evaluation
Introduction

- IPS: Intrusion Prevention System
  - Find buffer overflows and remove them
  - Use firewall to filter out malicious network traffic

- IDS: Intrusion Detection System
  - Is what you do after prevention has failed
  - Detect attack in progress
    - Network traffic patterns, suspicious system calls, etc
Introduction

- **Host-based IDS**
  - Monitor activity on a single host
  - Advantage: better visibility into behavior of individual applications running on the host

- **Network-based IDS**
  - Often placed on a router or firewall
  - Monitor traffic, examine packet headers and payloads
  - Advantage: can protect many hosts
Problem

- Prevalent security problems
  - Abnormal behavior: Buffer Overflows

- Current Methodology
  - Define a model of the normal behavior of a program
  - Raise an alarm if the program behaves abnormally

- The Problem
  - False alarm rate is high!!!
Motivation

- System Call Interposition

Observation: all sensitive system resources are accessed via OS system call interface
  - Files, Network, etc.

Idea: Monitor all system calls and block those that violate security policy
Model Creation

- **Training-based:**
  - Use machine learning and data mining techniques
    - Log system activities for a while, then “train” IDS to recognize normal and abnormal patterns
  - Easy but may miss some of the behavior

- **Static analysis:**
  - Extracted the model from source or binary
  - NO false positives!!!
A Trivial Model

- Create a set of system calls that the application can ever make
- If a system call outside the set is executed, terminate the application
- Pros: easy to implement
- Cons: miss many attacks & too coarse-grained
Callgraph Model

- Build a control flow graph of the program by static analysis of its source or binary code

- Result: **non-deterministic finite-state automaton (NDFA)** over the set of system calls
  - Each vertex executes at most one system call
  - Edges are system calls or empty transitions
  - Implicit transition to special “Wrong” state for all system calls other than the ones in original code
  - All other states are accepting
Callgraph Example

```c
f(int x) {
    x ? getuid() : geteuid();
    x++;
}
g() {
    fd = open("foo", O_RDONLY);
    f(0); close(fd); f(1);
    exit(0);
}
```

Entry point

Function call site is split into two nodes

Epsilon edges
Imprecision in Callgraph

```cpp
f(int x) {
    x ? getuid() : geteuid();
    x++;
}
g() {
    fd = open("foo", O_RDONLY);
    f(0); close(fd); f(1);
    exit(0);
}
```

The return address in `f` can be overridden.

Impossible Path. Yet the model will not be able to detect it since all transitions are valid.

Valid Path
NDFA: Model Tradeoffs

- A good model should be...
  - **Accurate**: closely models expected execution
    - Need context sensitivity!
  - **Fast**: runtime verification is cheap

<table>
<thead>
<tr>
<th></th>
<th>Inaccurate</th>
<th>Accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fast</td>
<td>NDFA</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Stack Model

- NDFA is not precise, loses stack information
- Alternative: model application as a context-free language over the set of system calls
  - Build non-deterministic pushdown automaton (NDPDA)
  - Each symbol on the NDPDA stack corresponds to single stack frame in the actual call stack
  - All valid call sequences accepted by NDPDA; enter “Wrong” state when an impossible call is made
 NDPDA Example

```cpp
f(int x) {
    x ? getuid() : geteuid();
    x++;
}
g() {
    fd = open("foo", O_RDONLY);
    f(0); close(fd); f(1);
    exit(0);
}

Entry(f) ::= getuid() Exit(f)
| geteuid() Exit(f)
Exit(f) ::= ε
Entry(g) ::= open() v
  v ::= Entry(f) v'
  v' ::= close() w
  w ::= Entry(f) w'
  w' ::= exit() Exit(g)
Exit(g) ::= ε

while (true)
  case pop() of
    Entry(f) ⇒ push(Exit(f)); push(getuid())
    Entry(f) ⇒ push(Exit(f)); push(geteuid())
    Exit(f) ⇒ no-op
    Entry(g) ⇒ push(v); push(open())
    v ⇒ push(v'); push(Entry(f))
    v' ⇒ push(w); push(close())
    w ⇒ push(w'); push(Entry(f))
    w' ⇒ push(Exit(g)); push(exit())
    Exit(g) ⇒ no-op
    a ∈ Σ ⇒ read and consume a from the input
    otherwise ⇒ enter the error state. Wrong
```
Solve Impossible Path

- Consider the previous example of an impossible path.

- The Abstract Stack model will detect the attack since it stores stack information. When returning from state \textit{Exit(f)}, the stack will have the return address \( v' \).
- State \( v' \) does not have a transition on system call \textit{exit()} hence the attack will be detected.
NDPDA: Model Tradeoffs

- Non-deterministic PDA has high cost
  - Forward reachability algorithm is cubic in automaton size
  - Unusable for online checking

<table>
<thead>
<tr>
<th></th>
<th>Inaccurate</th>
<th>Accurate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow</td>
<td></td>
<td>NDPDA</td>
</tr>
<tr>
<td>Fast</td>
<td>NDNFA</td>
<td></td>
</tr>
</tbody>
</table>
Digraph Model

- Combines some of the advantages of the callgraph model in a simpler formulation
- Model consists of a list of possible k-sequences of consecutive system calls (k=2 for simplicity)
- Monitor the application by checking the executed system calls vs. a precomputed list of the allowed k-sequences
- +: much more efficient than NDFA & NDPDA
- -: less precise than NDFA & NDPDA
Implementation Issues

- Non-standard control
  - Function pointers
  - Signals
    - Add extra edge to each handler + pre-/post-guard
  - Setjmp()
    - Modify stack, not suitable for NDPDA
    - Extend runtime monitor to handle

- Other modeling challenges
  - Libraries
  - Dynamic linking
  - Threads
Optimizations

- Irrelevant systems calls
  - Not monitoring harmless but frequently executed system calls such as brk()

- System call arguments
  - Monitoring the arguments at runtime improves both precision and performance
Evaluation: Performance

![Graph showing performance evaluations for different applications and monitoring overheads.](image-url)
Evaluation: Precision

Precision of each of the models, as characterized by the average branching factor. Small numbers represent better precision.
Unsolved Issues

- **Mimicry Attack**
  - Require high precision model to detect (poor performance)

- **Runtime Overhead**
  - Use more advanced static analysis to get more precise models
  - Later work such as VtPath, Dyck and VPStatic try to solve this problem
Backup
Push-down automata

- As in FSA, PDA have a set of states and a transition function.
- They differ from FSA by also having a stack. They accept context-free languages.
- At every transition, a symbol can be pushed or popped from the stack.
- They can accept either by state or by stack (if stack is empty), which are equivalent in terms of computational power.
- PDA is stronger than FSA. It can accept regular languages and also some irregular ones such as $0^n1^n$.

Once you see a 1, switch to the *End* state.
The stack contains as many 0 as seen in the input.
If the stack is empty at the end of the input, accept.
Dyck Model

- Idea: make stack updates (i.e., function calls) explicit symbols in the automaton alphabet
  - Result: stack-deterministic PDA

- At each moment, the monitor knows where the monitored application is in its call stack
  - Only one valid stack configuration at any given time

- How does monitor learn about function calls?
  - Use binary rewriting to instrument the code to issue special “null” system calls to notify the monitor
    - Potential high cost of introducing many new system calls
  - Can’t rely on instrumentation if application is corrupted

[Giffin et al.]
System Call Processing Complexity

<table>
<thead>
<tr>
<th>Model</th>
<th>Time &amp; Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>PDA</td>
<td>$O(nm^2)$</td>
</tr>
<tr>
<td>Dyck</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

$n$ is state count

$m$ is transition count
Reference

- cseweb.ucsd.edu/classes/sp02/cse231/eugene.ppt
- www.cs.utexas.edu/~shmat/courses/cs380s_fall09/08hostids.ppt
- Moss.csc.ncsu.edu/~mueller/seminar/spring05/sezer.ppt