Intrusion Detection via Static Analysis

By David Wagner & Drew Dean

IEEE Symposium on Security & Privacy, 2001

Acknowledgement

- Some of the following slides are borrowed or adapted from IEEE Symposium on Security & Privacy'01 talk by David Wagner
  - http://www.cs.berkeley.edu/~daw/papers/ids-oakland01-slides.ps

Intrusion Detection Basics

- Definition from wikipedia.org:
  - … is the act of detecting actions that attempt to compromise the confidentiality, integrity or availability of a resource
- Classifications
  - Manual detection vs. automatic detection
  - Host-based detection vs. network-based detection
  - Misuse detection vs. anomaly detection
- Evaluation
  - False negative
  - False positive
  - Unforeseen/mimicry attacks

Anomaly Detection

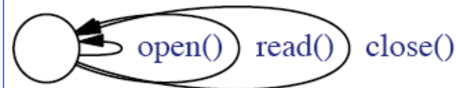
- Establish a model for **normal** program behavior
- Flag system states varying from the normal behavior model by statistically significant amounts as intrusion attempts
- Problem:
  - Detection effectiveness depends on the modeling of normal behavior and can be prone to a large number of false positives
- Proposal by Wagner and Dean
  - Use static analysis to derive a specification from the program source code and to reduce false alarms

Framework

- Key assumption
  - "A compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions may be readily monitored"
- Typically, the only way to interact with the OS is via system calls
- Approach
  - Derive the specification of expected system calls by statically analyzing the source code
    - Model the application as a transition system
  - Monitor system call trace at run-time
    - Check for compliance to the model

Trivial Model

```
g(){
   fd=open("foo",P_RDONLY);
   read(fd, …);
   close(fd);
}
```



- Construct the model S as the set of systems calls that the application can ever make
  - S={open, read, close}
- Enforcement: if the any system call not in S is observed, raise an alarm
- Properties
  - Simple, easy, efficient
  - Fail to detect many attacks

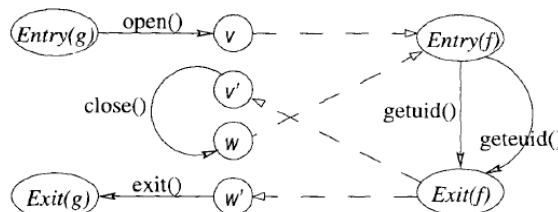
Callgraph Model

- Idea: retain some ordering information
- Model: represent the system call trace as an NDFA
  - Derived from the control-flow graph
  - Transitions: system calls + transfer of control
  - Non-deterministic: cannot statically predict which branch to take
- Enforcement: simulate the NDFA on the observed system call trace
  - Non-accepting state raises an alarm

Callgraph Model

- Example

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



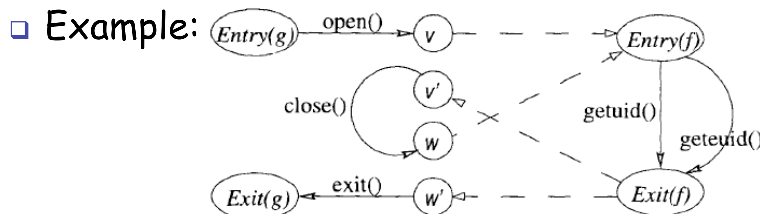
- Properties
  - More precise than the trivial model, no false positives
  - Still with imprecision: impossible paths are allowed
    - Eg. w➔Entry(f)➔ Exit(f)➔v'

Abstract Stack Model

- Idea: model the state of the call stack to eliminate impossible paths
- Model: a non-deterministic pushdown automaton with a stack
  - Equivalent to a context-free grammar
- Enforcement
  - Simulate the constructed NDPDA and compare the application call stack with a set of possible call stacks
    - Doesn't scale well for big applications
  - Efficient top-down parsing of the application system call trace

Abstract Stack Model

- Example:




```
while(true)
  case pop() of
  Entry(f)➔push(Exit(f);push(getuid());
  Entry(f)➔push(Exit(f);push(geteuid());
  Exit(f)➔no-op
  v➔push(v');push(Entry(f));
  v'➔push(w);push(close());
  …
  a∈∑➔read and consume a from the input
  otherwise➔error
```

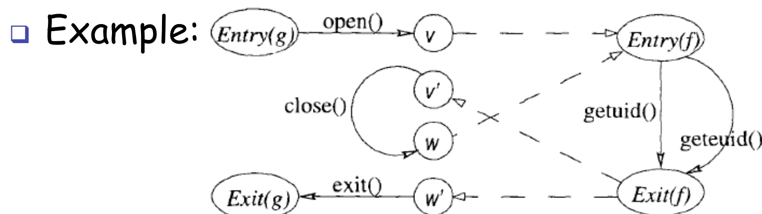
- Properties:
  - More precise modeling
  - Less efficient to monitor

Digraph Model

- Idea: check windows of consecutive system calls instead of the whole sequence
- Model: a list of possible k-sequences of consecutive system calls
  - For the context-free language L(M) and k-sequence s, s can occur in a trace iff $(\Sigma^* s \Sigma^*) \cap L(M) \neq \emptyset$
  - Expensive pre-computation and only k=2 is experimented
- Enforcement: keep a history of the last k-1 system calls, monitor the new system call and check the resulting k-sequence
  - Extremely efficient monitoring

Diagraph Model

- Example:



  - K=2, pre-compute the set $S \subseteq \Sigma^2$ such that $\langle x,y \rangle \in S$ iff $(\Sigma^* xy \Sigma^*) \cap L(M) \neq \emptyset$
  - S={<open,getuid>, <open, geteuid>, <getuid, clode>, …}
- Properties
  - Efficient to monitor
  - Less precise than the callgraph or abstract stack model

Implementation Issues

- Non-standard control flow
  - Function pointers
  - Signals
    - Add nodes for signal handlers and monitor signal reception events
  - Primitive setjmp() and longjmp()
    - Maintain a list of possible call stacks
- Other modelling challenges
  - Libraries
  - Dynamic linking
    - No runtime update of models yet
  - Threads
    - No support yet

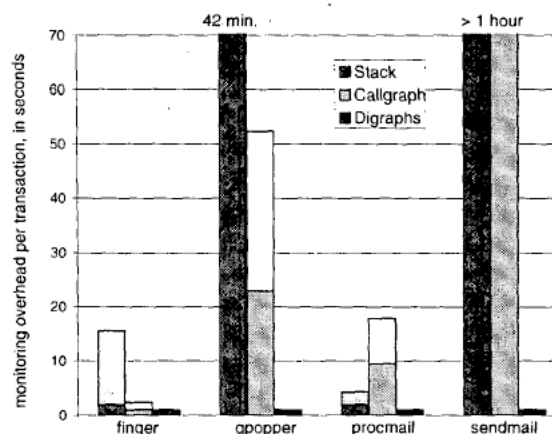
Optimizations

- Irrelevant system calls
  - Very common system calls provides little contextual information, e.g. brk()
  - Ignoring frequently executed harmless system calls can improve model precision
  - Useful on a per-application basis
- System call arguments
  - Quite a bit information gained by examining the arguments to systems calls
    - Lexically constant arguments can be analyzed statically
  - Improves both precision and performance

Evaluation

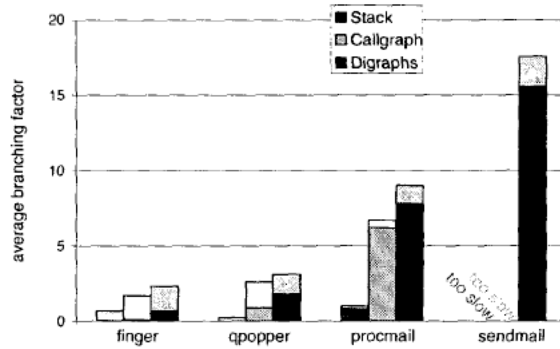
- Applications
  - finger, qpopper, procmail, sendmail
- Comparison
  - Three approaches(abstract stack, callgraph, digraph) × two variants(w/ arguments, w/o arguments)
- Metrics
  - Performance: runtime
  - Precision
    - Determines the robustness of detection against (unforeseen/mimicry) attacks
    - Branching factor = $|S|_{avg}$, given S as the set of system calls that would be allowed to come next without setting off any alarms

Overhead



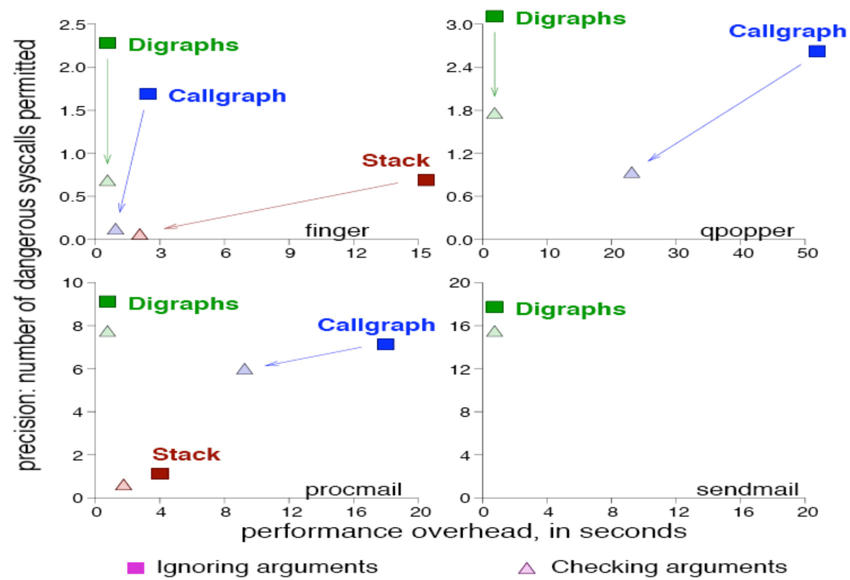
- Digraph models are consistently extremely fast
- Checking arguments improves performance by reducing the number of possible paths that we need to explore

Precision



- Abstract stack model is the most precise, digraphs the least
- Checking arguments provides substantial precision improvements

Overall Result



Summary

- Contributions
  - A novel approach by modelling possible system call traces via static analysis
  - Implementing and comparing the performance and precision of abstract stack model, callgraph model, and digraph model
    - Tradeoff between overhead and precision
    - Checking arguments should always be enabled
  - No false positives

Summary

- Unaddressed issues
  - Many optimization opportunities unexplored
  - No support for dynamic linking code
  - No support for multi-threaded code
  - What is the effect of considering only k=2 for digraph models
  - What would be a good metric to quantify the resistance of IDS to mimicry attacks
  - How to deal with applications with no source code

Questions? Comments?

Next Class

- Speaker: Kevin Dean and Lei Liu
- Topic: Java/Dynamic Optimization
- Reading assignment
  - R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode using the Soot Framework: Is it Feasible?", CC 2000.
- Other references
  - M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A Survey of Adaptive Optimization in Virtual Machines", Proceedings of the IEEE, 2005.