Intrusion Detection
Classes of Attacks

- **Probing: Reconnaissance before attack**
  - Port sweeps
  - OS/application finger printing

- **Denial of Service (DoS)**

- **Privilege escalation**
  - Remote to user
    - Attacker without any access to the victim machine gains access as a normal user, e.g., userid *nobody*
  - User to root
    - Attacker with access as normal user gains administrative privileges through an attack

- These two privilege escalation attacks may be chained
- Remote-to-user attacks typically exploit server applications (e.g., web server), while user-to-root attacks exploit other applications.
- They are rarely caused by OS errors or errors in network protocol implementations
Intrusion Detection

- Some attacks will get through in spite of every protection measure. Intrusion detection is targeted to detect such attacks.
- Detection is a solution of last resort

- Assumption: Behavior of a system changes when it is subjected to attack
- Approach: Detect these changes in behavior
Intrusion Detection Issues

- **Detection rate**
  - What fraction of attacks are detected

- **False alarm rate**
  - May be measured in multiple ways
    - how many false alarms per day
    - what fraction of normal behavior is flagged as attack
    - what fraction of behavior reported as attack is *not* an attack (false alarm ratio)
  - Considerable disagreement on which measure to use
    - but the third criteria is probably the best
    - But IDS vendors (and may be researchers) don’t like it
      - Will you buy a system will FA rate of 98%?
      - But you may not mind 10 false alarms a day!
Intrusion Detection Techniques

- **Anomaly detection**
  - Use machine learning techniques to develop a profile of normal behavior
  - Detect deviations from this behavior
  - Can detect unknown attacks, but have high FA rate

- **Misuse detection**
  - Codify patterns of misuse
  - Attack behaviors usually captured using signatures
  - Can provide lower false alarm rate, but ineffective for unknown attacks

- **Behavior (or policy) based detection**
  - Specify allowable behavior, detect deviations from specifications
  - Can detect new attacks with low FA, but policy selection is hard
Intrusion Detection Algorithms

- **Pattern-matching**
  - Most commonly used in misuse and behavior based techniques

- **Machine-learning**
  - Statistical
  - Algorithmic
  - Neural networks and other techniques
Intrusion Detection Behaviors

- Behaviors of
  - Users
  - Systems
    - processes, kernel modules, hosts, networks, …
Intrusion Detection Observation Points

- **Network-based (Network intrusion detection systems)**
  - **Benefits**
    - Unintrusive: plug a dedicated NIDS device on the network
    - Centralized monitoring
  - **Problems**
    - Encryption
    - Level of abstraction too low
    - Difference between data observed by NIDS and victim app.

- **Host-based**
  - Strengths/weaknesses complementary to NIDS
  - May be based on
    - system-call interception
    - audit logs and other log files
    - file system integrity (TripWire)
    - keystrokes, commands, etc.
Network Intrusion Detection

- Packet-based Vs Session-based
- Signature-based Vs Anomaly detection
- Example: SNORT (open source)
  - Uses pattern-matching on individual packets
- Some systems can block offending traffic
  - This is often dangerous, as systems usually have high false alarm rates
Host-based Intrusion detection

- System-call based characterizations most popular
- Behavior-based
  - System-call interposition plus wrappers
  - Domain/Type Enforcement
    - Certain application classes can access only certain files
    - Can prevent many privilege escalation attacks
    - Used in SELinux
- Anomaly detection
  - Sequences (finite-length strings) of system calls
  - FSA and PDA models of behavior
  - System call arguments
Automata Models for Learning Program Behaviors
Background

- Forrest et al showed that system call sequences provide an accurate and convenient way to capture security-relevant program behaviors
  - Subsequent research has further strengthened this result
- Key problem:
  - What is a good way to represent/learn information about system call sequences?
    - Issues: compactness, accuracy, performance, ...
Early Research

- Forrest et al [1999] compared several methods for learning system call sequences
  - Memorize subsequences of length N (N-grams)
  - Markov models
  - Data-mining (using RIPPER)

- N-grams found to be most appropriate
  - Markov models provided a slight increase in accuracy, but incurred much higher overheads
Illustration of N-gram Method

1. S0;
2. while (..) {
3.     S1;
4.     if (...) S2;
5.     else S3;
6.     if (S4) ...
7.     else S2;
8.     S5;
9. }
10. S3;
11. S4;

Sample execution:
- S0 S1 S2 S4 S5
- S1 S3 S4 S2 S5 S3 S4
- S0 S3 S4

3-grams learnt:
- S0 S1 S2
- S1 S2 S4
- S2 S4 S5
- S4 S5 S1
- S5 S1 S3
- S5 S4 S4
- S1 S3 S4
- S3 S4 S2
- S4 S2 S5
- S2 S5 S3
- S5 S3 S4
Drawbacks of N-gram Method

- **Number of N-grams grows exponentially**
  - N must be small in practice (N=6 suggested)
  - Implication: difficult to capture long-term correlations
    - S0  S3  S4  S2 never produced by program, but all of the 3-grams in this sequence are

- **Remembers exact set of N-grams seen during training --- no generalization**
  - necessitates long training periods, or a high rate of false alarms
Models without Length Limitations

- **Finite-state automata**
  - Even an infinite number of sequences of unbounded length can be represented
  - Naturally capture program structures such as loops, if-then-else, etc.

- **Extended finite-state automata**
  - FSA + a finite number of state variables that can remember event arguments

- **Push-down automata**
  - By capturing call-return info:
    - PDAs are more accurate than FSM
    - Models are hierarchical and modular:
      - Hierarchical nature facilitates presentation
      - Smaller program models
      - Reuse of models for libraries
  - Extend PDAs to incorporate variables
Model extraction approaches

- **Static analysis [Wagner and Dean]**
  - Pros: conservative
  - Cons:
    - difficult to infer data values, e.g., file names
    - difficult to deal with libraries, dynamic linking, etc.
    - overly conservative
      - for intrusion detection, can detect only attacks that are outside of the semantic model used for analysis
      - specifically, buffer overflows, meta character attacks, etc.

- **Machine learning by runtime monitoring**
  - Pros:
    - can detect a much wider range of attacks
    - can deal with libraries, dynamic linking
    - inferring data values is easier
  - Cons:
    - False positives
Difficulty in Learning FSA from Strings

- Strings do not provide any information about internal states of an FSA
  - given S1 S2 S3 S2, which of the following FSA should we use?

    \[
    \begin{array}{c}
    S1 \rightarrow S2 \rightarrow S3 \rightarrow S2 \\
    S2 \rightarrow S1 \rightarrow S3
    \end{array}
    \]

  - what is the criteria for determining the “better” FSA?
  - even if we can answer this, the answer will depend on additional examples
    - e.g., sequences S1 S2 and S1 S2 S3 S2 S3 S2 will suggest that the second FSA is the right one

Learning FSA from sequences is computationally intractable
  [Kearns & Valiant 89, Pitt & Warmuth 89]
Learning FSA Models: Graybox Techniques

Key insight:

For learning program behaviors, additional information can be used to simplify the problem:

*exploit program counter value to obtain state information*
Learning FSA Models

A sample intercepted program behavior:
(S0,1) (S1,3) (S2,5) (S4,8) (S1,3) (S3,7) (S4,8) (S5,10)

1: s0;
2: while (...) {
3:   s1;
4:   if (...) 
5:     s2;
6:   else
7:     s3;
8:   s4;
9: }
10: s5;
Approach Details

- **Interception of system calls using ptrace (Linux)**
  - same mechanism used by Forrest and other researchers
- **Examine process stack to obtain program counter information**
- **Dynamic linking poses a problem**
  - same function may be loaded at different locations during different runs
  - Solution: use program counter value corresponding to the code calling the dynamically loaded library
  - Side benefit: ignoring library behavior makes FSA more compact
Approach Details (Continued)

- **Fork:** Parent and child monitored with same FSA, but process contexts maintained.
- **Exec:** typically, a new FSA for the execve’d program is used.
- **Detection time**
  - mismatch may occur in terms of either the system call or program location
  - use leaky bucket algorithm for aggregation
  - program counter helps resynchronize even after observing behavior not seen during training
Training Convergence

- FSA method converges faster than N-grams
  - roughly speaking, FSA method can do with roughly an order of magnitude less training period than N-gram method
False Positive Rate

- FP results are similar to convergence
  - for a given FP rate, FSA method requires an order of magnitude less training than N-gram method
Mimicry Attacks

- Attacks crafted with knowledge of IDS
- Execute only system call sequences that would be permitted by the model
- A mimicry attack can be developed from an attack sequence by inserting “junk” system calls that make it appear as if a legitimate sequence is generated
  - Junk system calls made possible by using bad system call arguments
- Graybox IDS complicate mimicry attacks due to the need to fake call site
  - Control does not return to attack code after a call!
- But can still be made to work
- Known mimicry attacks based on memory corruption+injected code
Learning System Call Arguments

- Earlier methods focus on control-flows
  - System call sequences (N-grams)
  - Automata models of behavior
    - FSA or PDA, with transitions labeled with system calls
  - System call arguments largely ignored

- Detects usual control-hijack attacks

- Don’t detect most attacks that modify resources access by a system call
  - Non-control data attacks
  - Race condition attacks
  - Mimicry attacks
  - ...


**Approach**

- **Incorporate dataflow info into control-flow models**
  - Exploit control-flow context to improve model precision
  - Go beyond treating system call arguments in isolation
    - “A program can only write a compressed version of its input file”
      - `open("/home/joe/model.ps", "r"); ...; open("/home/joe/model.ps.gz", "w"); ...`
    - “All sensitive files should be closed before execve operation”
      - `open("/etc/passwd", "r") = 6; ...; close(6); ...; execve("untrusted_prog", ...); ...`
  - Learning temporal relations => parameterized models
    - command ”find –exec /bin/ls...” resulting in execve("/bin/ls", ...)

- **Learning-based:**
  - learns dataflows that seem to be present in program
The dataflow properties need control-flow context

Example:

L1: fd1 = open(“/etc/passwd”, O_RDONLY);
L2: fd2 = open(“/tmp/out”, O_RDWR);

We cannot combine information about opened filenames, otherwise “/etc/passwd” at L2 will be accepted.

Control-flow context provided by previous FSA/PDA techniques could be used.

Context encoded by giving names for event arguments.
Motivating Example: `simpletar` program

```c
int main(int argc, char **argv) {
    source_dir = argv[1], target_file = argv[2];

    opendir("/opt/proj")
    opendir("/opt/proj/src")

    target_fd = open(target_file, WR);

    while ((dir_name = pop()) != NULL) {
        dir = opendir(dir_name);
        foreach (dir_entry) {
            if (isdirectory(dir_entry))
                push(dir_entry);
            else {
                source_fd = open(dir_entry, RD);
                read(source_fd, buf);
                write(target_fd, buf);
                close(source_fd);
            }
        }
    }
    close(target_fd);
}
```

start("/opt/proj", "/tmp/proj.tar")
open("/tmp/proj.tar", WR) = 3

```c
close(source_fd);
}
```
Possible Dataflow Relationships

- **Unary Relations**: properties of a single system call argument
  - Represented as $X R c$, where
    - $X$: an argument name,
    - $c$: a constant value,
    - $R$: a unary relation
  - Examples of unary relation $R$:
    - **equal** $=> X$ takes only a single value always equal to $c$
    - **elementOf** $=> X$ takes any value from the set $c$
    - **subsetOf** $=> X$ takes values all of which are drawn from set $c$
    - **range** $=> X$ takes values in the range $c$ (e.g., $c = (0, 2)$)
    - **isWithinDir** $=> X$ is a file name argument that is always contained within a specified directory $c$
      - Example: If $X$ takes values “/home/user/abc” and “/home/user/xyz”, we can say that $X$ **isWithinDir** “/home/user”
Possible Dataflow Relationships

- **Binary relations**: captures relationships between two event arguments
  - Represented as $X R Y$, where
    - $X, Y$: argument names
    - $R$: a binary relation
  - Examples of binary relation $R$:
    - **equal**: equality between $X$ and $Y$
    - **isWithinDir**: file/directory $X$ is within directory $Y$
    - **contains**: directory $X$ contains file/directory $Y$
    - **hasSameDirAs**: $X$ and $Y$ are within a common directory
    - **hasSameBaseAs**: $X$ and $Y$ have same base (eg: a.c, a.h)
    - **hasSameExtensionAs**: $X$ and $Y$ have same extension (eg: a.c, b.c)
Binary Relations

❖ **How to interpret a binary relation?**
  - A naïve interpretation of \( X \text{ equal} \ Y \): \( X \) and \( Y \) have only one possible value in all traces
    - not useful interpretation when \( X \) and \( Y \) occur multiple times

❖ **Our approach:** \( X \mathbin{R} Y \Rightarrow X \) is related to **closest preceding** \( Y \)
  - \( X \mathbin{R_T} Y \): For every occurrence of \( X \) in a trace \( T \), \( X \mathbin{R} Y \) holds
    - Example: For a labeled trace \( T \) of the form:
      \( Y = 1, Z = 2, X = 1, Y = 2, X = 2, \)
      we say that \( X \text{ equal}_T Y \), but not \( Y \text{ equal}_T X \)
  - \( X \mathbin{R_T} Y : \forall T \in \mathcal{T}, X \mathbin{R_T} Y \) holds
Learning Unary Relations

- With each argument, maintain a list of all values encountered in all traces
- If too many values are associated with an argument, use relations that approximates set
  - `range` (min-max pair) for port numbers, file descriptors,
  - `isWithinDir` for file names
  - `subsetOf` for flag arguments of open

- Runtime and storage requirements
  - A trace of size $N$ (in bytes) can be learnt in $O(N)$
  - Storage requirement: $O(S)$, where $S$ is size of control flow behavior model
Learning Binary Relations

- **Key algorithmic issue:** How to find pairs of arguments of relationship in near-linear time?
  - Trace lengths (N) can be $10^5$ to $10^7$
  - Quadratic algorithms too expensive

- **Approach:**
  - Equality relationships
    - Store most recent values of arguments in a hash table
    - Given the current value of an argument, simply look up in the hash table for the set of all candidate arguments
  - String relationships
    - Use *trie* data structure to achieve similar effect

- **Algorithm complexity is** $O(Nr)$ where $r$ is maximum number of relations involving any single argument
  - Much smaller than $N^2$ – typically $r < 10$
Effectiveness for Attack Detection

- Can detect data corruption attacks that don’t involve control-flow hijack
- Race condition attacks
- Allow verifying nontrivial security props
  - If the IDS doesn’t raise alarms then these properties will be preserved
- Examples of properties verified:
  - `find` executes only those programs that are specified using a “-exec” command-line option
  - All files read by `tar` would reside within the directory specified on the command-line
  - The only file written by `gzip` is obtained by adding “.gz” suffix to its argument
Effectiveness for Attack Detection

- **Fingerd symlink vulnerability**
  - Attack exploits absence of symbolic link check
    ```c
    if (lstat(tbuf, &sbuf1)) return 0;
    L1: fd = open(tbuf, O_RDONLY); ...
    fp = fopen(fd, "r"); ...
    ```
  - Attacker makes `.plan` as a symbolic link to `/etc/passwd`
  - **Attack detection**: violation of binary relationship between username command line argument and file opened at L1
## False Alarm Rates

<table>
<thead>
<tr>
<th>Program</th>
<th>Training Trace Length (#syscalls)</th>
<th>Detection</th>
<th>False alarm rates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Trace Length (#syscalls)</td>
<td>Base (X 10^{-5})</td>
</tr>
<tr>
<td>httpd</td>
<td>( X 10^6)</td>
<td>1.75</td>
<td>3.10</td>
</tr>
<tr>
<td>sshd</td>
<td>4.75</td>
<td>14.74</td>
<td>0.35</td>
</tr>
</tbody>
</table>

- Unary relations increase false alarms modestly
- Binary relations add high false alarms in httpd
  - 95% of these are due to accidental relationships learnt for a rarely occurring system call in training phase.
  - We can address them by adding a notion of confidence level with each relation.
Performance: Model Sizes

- Models are very small as compared to size of programs
  - Program sizes: 68KB(find) to 435KB(wu-ftpdp)
    - #Binary relations are linearly proportional to size of control flow model
Summary of Argument-Learning Approach

- Uses dataflow information to enhance precision of intrusion detection models
- Can be layered over other techniques for learning control flows
- Effective in detecting sophisticated attacks
- Extracts models that are compact and produce low false alarms
- Enables formal reasoning about security guarantees of models