**Intrusion Detection**

- Host-based Intrusion Detection
  - System call monitor, HBIDs
- Goal: Prevent damage from attack
- Key Idea: A program under attack behaves differently
- Key Idea: Attacker must make syscalls to do real damage

Two Questions:
1. What does model look like?
2. How do we build it?

Ideal:
- Model = set of system calls in application source code (set model) (static)
- Too course
- Alternative: run program and record syscalls it make (dynamic)
- Every application has a different model

<table>
<thead>
<tr>
<th>Dynamic Models</th>
<th>Static Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ easy to run the application</td>
<td>- harder to build</td>
</tr>
<tr>
<td>- false positive</td>
<td>+ no false positive</td>
</tr>
</tbody>
</table>

(Note: + means good, - means bad)
Another model: **n-gram models**

n-gram = sequence of n items, syscalls in our case

Ex./ Suppose an application performs \{open, read, read, close, open, write, close\}

2-gram model = \{ (open, read), (read, read), (read, close), (close, open),

(close, open), (open, write), (write, close) \}

Note:
- syscalls are ordered in this model
- higher gram degree = more info about sequencing = harder to attack
- most common use: 6-gram

How to implement n-gram model at run-time?

Monitor stores last n-1 syscall, \((S_1, \ldots, S_{n-1})\). When application makes syscalls, S, check whether \((S_1, \ldots, S_{n-1}, S) \in M\), store \((S_2, \ldots, S_{n-1}, S)\).

Another model: **FSA (Finite State Automaton) model**

Typically build statically
- just like model checking

Ex./

```
If ( (fd = open) <0)
    exit();
while(read(...))
    getTimeOfDay(...);
    close();
```

If (open, read, read, getTimeOfDay)
Compromised? YES, because \{read, read\} does not exist in this FSA.

Convert P (program) into Mp, Mp accepts L(Mp)

Fact: All legitimate execution of P should produce a sequence S of system calls such that \(S \in L(Mp)\)
Issue with this model: Function calls

Ex./

```c
void foo(void) {
    read(...);
}

void main() {
    open(...);
    foo(...);
    setuid(-1);//drop privilege
    foo();
    exec(...); //note: exec
    //occurs w/ no privilege
}
```

Problem with above FSA: two different syscalls, setuid and exec, can be called after foo(...).
Solution: To avoid such confusing, we will use a PDA to solve the problem.
What if a recursive function?

```c
void foo(void){
    If (…) foo();
    Read();
}

void main() {
    …
}
```

Problem w/ PDA:
- Takes too long time: very inefficient
- Don’t know how many calls in recursive

Solution: *Efficient Contest Sensitive Intrusion Detection*:
- Modify application to inform monitor of function calls
- Ex./
```c
void foo(void){
    notify_call();
    if (...)
    foo();
    read();
    notify_return();
}
```