Intrusion Detection Systems

10/22/2010

Paper: Efficient Context-Sensitive Intrusion Detection

We deal with Host-based Intrusion Detection Systems or IDS

- Applications have some “normal” “behavior”
- Applications under attack will behave abnormally

The behavior of an app is defined by the sequence of system calls it makes.

Why do we consider system calls?

- Attacker must use system calls to cause any damage
- It is a convenient point of mediation

What is “normal” for an application?

- The expected behavior
- The set of possible system calls it can make

Example:

```c
while(1)       A sequence of getpid() and
{        write() in any order is
    if (c == 0) accepted, as long as no other
        getpid(); call is made (eg. exec() )
    else
        write();
}
```

M = model of the normal behavior of App

When an application makes a system call, the kernel checks the model. If it is consistent \( \rightarrow \) allow, else \( \rightarrow \) abort/kill it (no notification required)

Evaluation criteria of Intrusion Detection Systems

- Performance
- No/ Low False Positives
- No/ Low False Negatives
**Model Construction:** A model can be constructed in the following ways

1) Automatic
   a) From source code – Leads to 0 False Positives
   b) From execution traces – Can customize the model in this method, but in the case of low code coverage it can lead to many False Positives
2) Manual – Works fine only for few syscalls and cannot be used in complex scenarios

It is good to have a single model for the system, covering all the possible calls. For the syscalls made by libraries we cannot build the model from the source code and hence use the binaries

![Diagram of model construction](image)

The app as well as the lib cannot be trusted completely. At runtime when the app is loaded, the kernel loads $M_{app}$ and $M_{lib}$. So the combined model gives rise to a lot of possibilities and may render the model useless. Eg: In case of ‘libc’ the model will allow almost all operations and the system becomes vulnerable. Therefore we trust the app to not be deliberately evil and that it tries to do the right thing, but also understand that it may have bugs

**Sets of System Calls:**

A better way to build a model is by taking into consideration the order of system calls. A simple way to do this is by building n-gram models

$M = \{ ( , , ,...) , ( , ,...) , ( , ,...) .... \}$

- Sequence of n syscalls that the app might perform

n = 1; Set Model

For n = 2 consider the below example

```c
int main()
{
    int fd = open();
    if ( fd >= 0 )
        read();
    else
        write();
    fork();
    open();
    ioctl();
    close();
}
```

What are all the pairs of consecutive syscalls that the app may perform?

- (open, read)
- (open, write)
- (read, fork)
- (write, fork)
- (fork, open)
- (open, ioctl)
- (ioctl, close)
We arrive at these pairs by running the app with different inputs and getting a different order of syscalls each time. Many executions build a good model. The model is built for a particular process and is inherited by all its children.

As \( n \) becomes larger, a redundancy in the set members is noticed. This model may have False Negatives.

The Operating System verifies the normal behavior of an application by storing the last \((n-1)\) system calls in an n-gram model. It reads the current system call and checks against the list to see if it is valid or not.

We can construct a **State Machine** for the same example

The State Machine shows all the allowed executions. The states represent the last system call made by the application. The state of the application is the execution point or the program counter.

We can also perform **Model Checking** and build a **Control Flow Automaton**

\[ \varepsilon \] - Transition that is allowed without a syscall execution

**This is a Non-deterministic Finite State Machine**

The automaton can be built from either the source code or the binaries. When the application is loaded into memory, the operating system remembers its state. When system calls are made, it checks if the transition matches the expected ones and keeps going till there are edges in the graph.

This model is an improved one over the others.