

Detecting Intrusions..

April 20, 2006

mnair@cs.sunysb.edu

Last Class

The structure given in figure 1 is very similar to Ostia/Sandboxing. The *policy* is a sequence of system calls which confirms Apache to the correct behavior. Here Apache is used just as an example. A similar design can be use for almost any network based service.

This design is based on the assumption that when a attacker injects errors, there will be anomalous system calls. “exec(“/bin/sh”)” the kind that cannot be expected during ordinary execution. The information of correct behavior, called *policy*, is checked by the monitor. We develop methods for extracting and storing(representing) these policies.

Model 1

1. Run the Apache on a safe controlled environment.
2. Learn the set of system calls that Apache uses (Learning Phase). This is stored in a set 'S'.
3. Allow the apache only to make the system calls in the set 'S'

This is a dynamic analysis method.

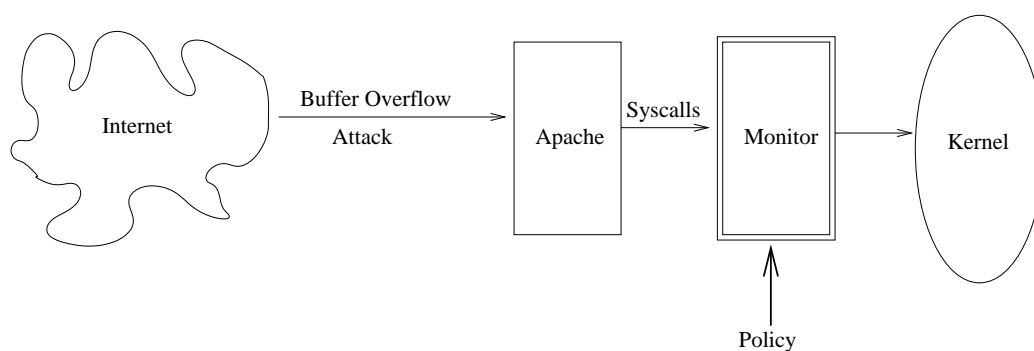


Figure 1: Schematic Model of Scheme

Model 2

1. grep the apache source for syscalls
 2. Collect the set S of syscalls apache uses
- This is a static analysis method.

Static vs Dynamic

The two common problems with Intrusion Detection Systems are:

- **False Positives** Events flagged as dangerous but are not really threats.
Analogy : "Boy who cries Wolf".
- **False Negatives** Threats which are missed by the detection systems.
Analogy : "Sleeping Security".

The limitation of static method (model 1) is that it is dumb. For example it will parse through comments also. Even if an intelligent parser is made which skips all the comments. There might be areas of the code that are not necessarily executed, or might be executed conditionally, which might always allowed by this scheme. Thus more often than not we end up getting a very large set. Thus leading to false negatives.

The dynamic analysis (model 2) firstly requires a safe environment for running. Code coverage is very important; we need to exercise all the possible execution paths. If we miss any of the path it could lead to a set S that is not complete – smaller than it should. This means that there will be high probability of many false positives. This could be very irritating to the user and be unusable. "Unusable security becomes Unused Security".

Model 3

```
1: read(...)
2: if(...)
3:     write(...)
4: else
5:     exec(...)
6: endif
7: read(...)
```

If either model 1 or 2 are used then the set 'S' would contain the following operation {read, write, exec }. Thus if the following sequence of operations

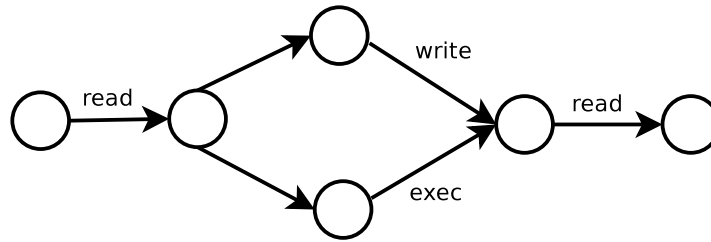


Figure 2: Finite State Automata for the code segment

is seen, read, write, exec, read is seen, it will be allowed. Clearly this is a bad sequence and should not be allowed. Unfortunately our model is not powerful enough to handle these cases. We use a more intelligent design to store our information. Instead of a set we use a finite state automata.

Figure 2 represents the finite state automata for the code segment shown in the beginning of the section. This automata is generated statically. One of the important things is that we consider only system calls. We are only trying to catch privilege abuse not logical errors. All other states are collapsed. Also by construction this is a non-deterministic automata. But by an important result from Automata theory we know that a non-deterministic Finite state Automata (FSA) is equivalent to a deterministic FSA. Thus we build a deterministic FSA from the generated non-deterministic FSA.

When this model is used, with each system call, the monitor traverses states in the automata. If from a state, a call occurs that does not lead to a valid state, then we know that an error is present. We can then take preventive action; such as kill the offending process.

Model 4

In this model too, a FSM is used to store the results of the allowed actions. But the difference is that the FSA is generated dynamically when the process is being run in a secure environment. This is similar to Model 1 but rather than a set an FSA is used.

Consider the code segment shown below:

```

while(...)
{
    if(flag)
    {

```

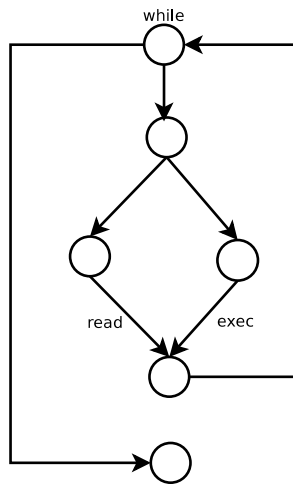


Figure 3: Problems with Models 3 & 4

```

    exec(...)
  }
  else
  {
    write(...)
  }
}

```

If we know that flag does not change within the while loop then we know that the regular expression that can occur with this code segment is $(read)^*|(exec)^*$. Rather if the automata is used along with either models 3 or 4 the following will be the regular expression of the automate $(read|exec)^*$. Figure 3 represents the automata of this (wrong) expression. This means that sequences such as read, exec, read are allowed by the automata but this is not what the code is intended to do.

The main problem is that automata does not store any state. We are unable to decipher where we came from. This issue is solved in the next model.

Model 5

This model is similar model 3. We generate the automata statically but rather than a FSA, we use a Push Down Automata(PDA). In a PDA, apart

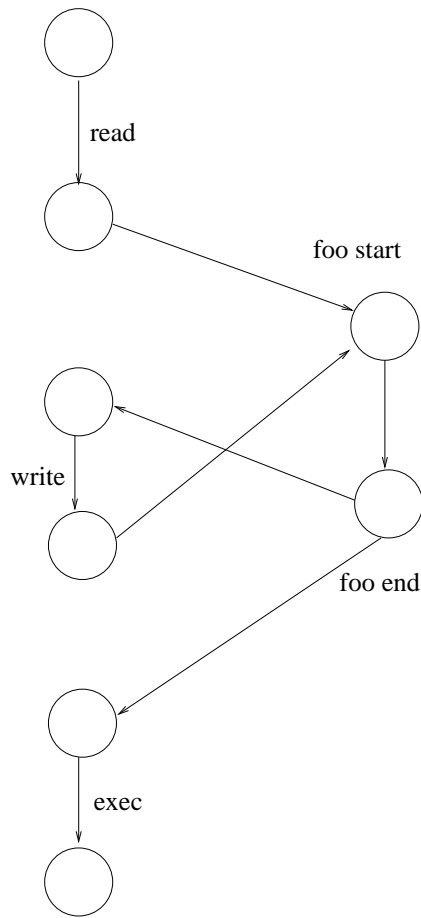


Figure 4: Finite State Automata for “foo” segment

from the states we also have a stack. The stack can be used to record from where we came from.

Let $foo(\dots)$ be a function from which no system calls are made. According to our model foo can be collapsed into a single node. Consider the following call sequence:

```

read(...)
foo(...)
write(...)
foo(...)
exec(...)
  
```

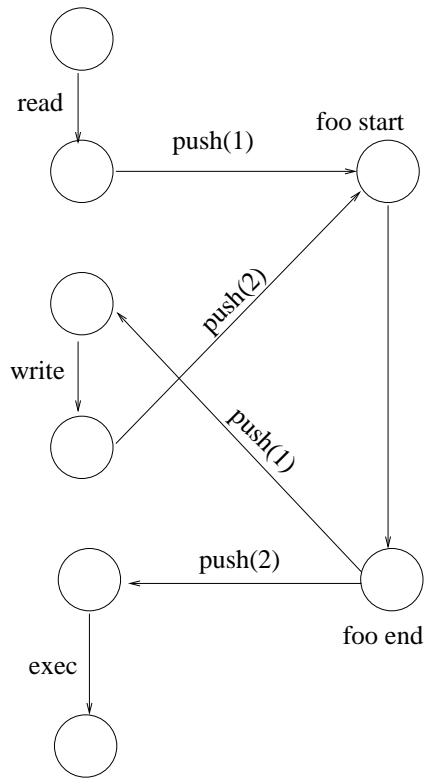


Figure 5: Push Down Automata for “foo” segment

Fig 4 shows the FSA that is generated from the above sequence. It is clear that such an automata would lead to the wrong result. It is also clear that it is an inherent problem with finite automata. We solve the problem by using push-down automata(PDA). Fig 5 show the PDA generated by the code segment. PDA does not suffer from the problem that FSA had.

Unfortunately PDA are not a very efficient. This is because unlike a FSA a deterministic PDA is not equivalent to a non-deterministic PDA. Thus if there is ambiguity the stack could grow infinitely.

Consider the non-deterministic PDA shown in figure 6

```
recursive_read(...)
{
    if(...)
    {
        read(...);
    }
}
```

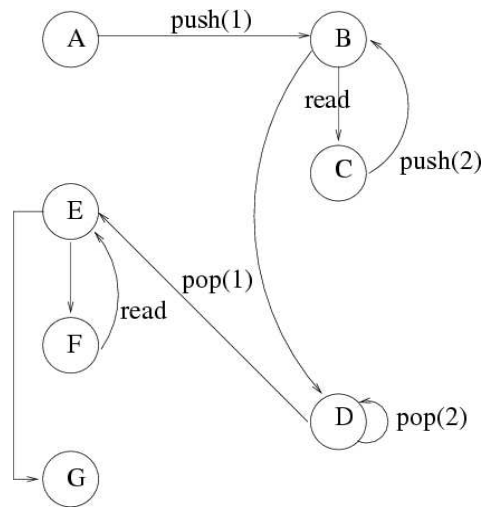


Figure 6: A non-deterministic Push Down Automata

```

    recursive_read(...);
  }
}

other_func(...)
{
  recursive_read();
  while(...)
  {
    read(...);
  }
}

```

For the above code snippet make the PDA shown in figure 6. Here the monitor is unable to decide whether the call is attributed to the read in `other_func` or whether the code is still in `recursive_read`. Fundamentally there is ambiguity or two poss

Performance Issues

Monitor needs to see what is the sequence of calls and returns. Only then can we ensure that the PDA is determinate. One of the ideas suggested to solve this is introducing two new system calls:

Approach	Performance
Set-Based	Very Efficient
Finite Automata	Efficient
Push-down Automata	Unusable (efficient with tricks)

Table 1: Performance Comparison

- `call_function(...)` - called just before a function call takes place.
- `return_function(...)` - called just after a function returns.

To both the system calls we pass the function pointer to the new calls. Unfortunately the downside with this method is the high overhead. This make each function call make a system call. While a function call involves only 2 instructions, a system calls involves atleast a 1000 instruction.

A better solution to the problem is to use macros to store the call and return pointers. These pointers are stored in fixed location in the memory. This call return sequence can be used to determine the calling function and resolve ambiguities.

Some Improvements

In all the models discussed, we only look at the system call and not its arguments. Sometimes it is useful to look at the arguments. For example the Apache may execute only some binaries but never `exec("/bin/sh")`. This could lead to a more intelligent monitors. Other improvements could be looking at the looking the relationship between the arguments. For example the monitor can guess that if a particular file is opened and a file descriptor is obtained, then read/write operations and finally close syscall should performed on the same descriptor; never on a descriptor which has not been obtained from open.