Intrusion Detection Systems (..Contd..)

10/29/2010

Paper: Efficient Context-Sensitive Intrusion Detection

Last class:

- Introduction to IDS using system calls
- Model construction - statically from the source code or from the binaries
- The model set with n-gram models
- State machine and Control Flow Automaton for system calls in an application

Here we deal with Function calls. How does the IDS treat function calls?

Example:

```c
main( )       log( )
{        {
    setuid(1000);
    log( );
    system( );
    setuid(0)
    log( );
}
```

On the first look it seems like we cannot call system() when userid = 0

Control Flow Automaton for main( ) and log( )
There is a problem with the above graph. A cycle exists in it - main 4 → log 0 → log 1 → main 2 → main 3 → main 4

Therefore the call to system() after setuid(0) looks possible. The IDS validates the calls to system() when the uid=0 and does not raise any alarm. This can go on in a loop with as many calls to the system and still manage to evade detection by the IDS

Different methods to deal with this issue

- Maintain multiple copies of the Log() model, one for each time the function is called. This is not a good way since the number of calls could be huge and the size of the model could increase exponentially due to the duplication
- Allow a function to return only to its caller by using the PDA model

**PDA (Push Down Automaton)**

A PDA does 3 things on input

- Change its state
- Check and pop out the top of the stack
- Push new material onto the stack

An ε edge indicates that no Push or Pop is required

The PDA basically says that the edge can be taken only if β (system call) is the input, F is the TOS which has to be popped out and G is some material to be pushed onto the stack. An edge specifies how to update the program stack

We model the program’s stack of return addresses in PDA
In the case of ‘if’ conditions, both the True and False paths will have $\epsilon/\epsilon/\epsilon$ edges and hence either of the two paths can be taken, then backtracked if necessary.

The IDS maintains the state and the stack of the program as it runs. For the above code example, keeping the given PDA in mind, we can analyze as follows:

1. Initial State = main0
   Input: setuid(1000). There is only 1 edge and is taken (TOS = $\epsilon$)

2. State = main1
   $\epsilon$ edge taken as there is no input (syscall). TOS = $\epsilon$ and 2 is pushed

3. State = log0
   Input: write(). The only edge is taken. No pop or push
   If input: read(), then abort. No changes to stack

4. State = log1
   Two edges, both are $\epsilon$. Check the TOS match and then decide.
   Since we have 2 on top and it needs to be popped out in $\epsilon/2/\epsilon$,
   we take that edge to go to the next state and pop out the 2

5. The same continues for the rest of the program with all its inputs.
   We observe the input, the TOS and decide which way to go

6. In the case of the second log() call State = log1
   TOS – 5 and only the edge $\epsilon/5/\epsilon$ matches. Hence we take it
   and pop out the 5 from the stack

7. Continue till the end of inputs

Enhancements that could be done:

- Can model ‘if’s better
- Can model the syscalls arguments better
- Can model the function calls better

When we have functions that are often called (ex: malloc(), strcpy(), etc.), the inclusion of these in the model will make it quite complicated. Consider the below example:

```c
void foo( )
{
    if( … )
        func1( );
    else
        func2( );
}
```

Which way does the program go?

The state of the program is bifurcated. Copies of the state are made and pushed down the stack. When the next system call is encountered, we can discard one of the states.
(the wrong path). In case the same call is invoked, we continue to bifurcate and keep copies until we are able to eliminate all other paths and left with just one. But the pruning of unwanted states may not work in all cases.

We can improve the model by also feeding the IDS with more information on function calls. The model replaces \( \varepsilon \) transitions with the stack contents [null()] as shown:

```plaintext
null(2)/\varepsilon/2
null(-2)/2/\varepsilon
```

As the program feeds more info to the IDS, it helps the IDS in monitoring the program execution better.

We can completely remove the non-determinism in NDA if we write as shown below:

```plaintext
setuid(1000);
null(2);
log();
null(-2);
system();
setuid(0);
null(5);                    This mainly says that if there is a ‘5’ on the stack when log() is
log();                       entered, then the function should also return to a state where there
null(-5);                    is a ‘5’ on the top of the stack
```