CSE 509 Lecture 9

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

September 30, 2013

1 Review

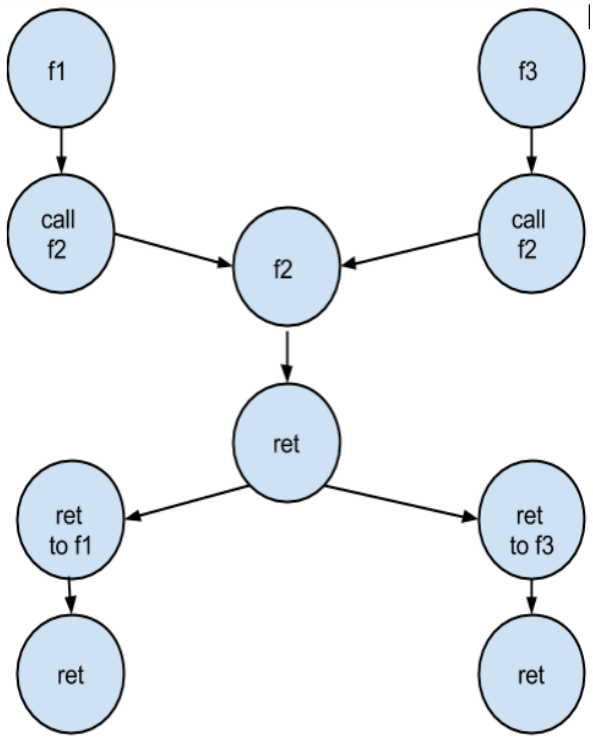
Control flow automaton is based on static code analysis. Consider the below scenario.

```
func1(){
    ...
    func2();
    ...
}
func2(){
    ...
    ...
}
func3(){
    ...
    func2();
    ...
}
```

func2 is a function that is often called by several functions in the application. printf is one that is often called. In func1, func2 is called. The call sequence is valid if the return of func2 is to the correct place in func1. However the invalid sequence of returning to func3 (instead of func1) from func2 is accepted by the model. The above model still does not prevent an attacker from forcing the control to return to a function different from the caller and carry out an attack.

2 Context Sensitive IDS

Context Sensitive IDS uses Context Free Grammars, represented by Push Down Automata (PDA) to model the control flow in an application. By pushing on to a stack an additional symbol (1 in case of func1 and 2 in case of func2). We can correctly define that a return to func1 is the only valid transistion by looking at the top symbol. This plugs the weaknesses in Finite State Machine model.



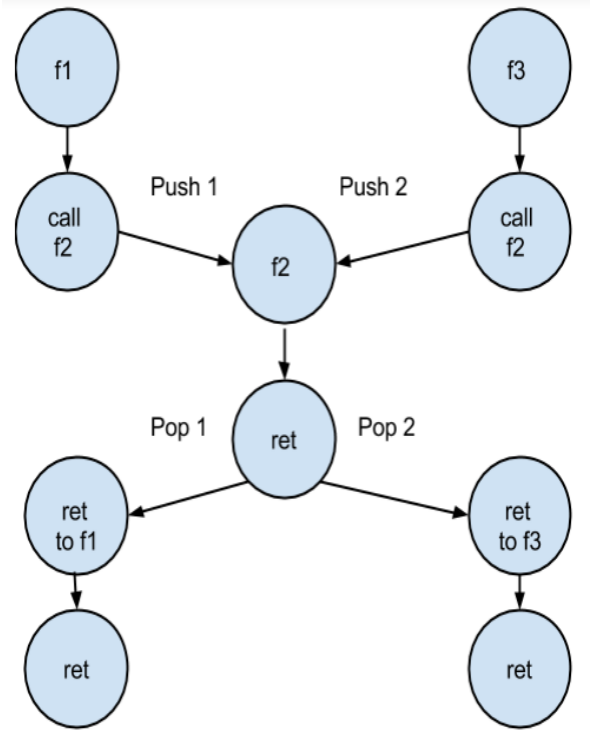
Control Flow Model

2.1 Verification

The model of the application stores the PDA along with a stack and a state to track the current state in a program. When a system call is made, it checks if it is allowed from the current state (as in Finite State Machine Model) and pushes a symbol to the stack. On return, it pops the top character and makes sure that the return is to the orginal caller. The weakness with this model is ambiguity and explosion in number of states due to non-determinism.

After an if condition, there are two states where control can go. If there is no sufficient information from the system call sequence to identify a particular branch, we are left with no option than to bifurcate and keep track of both possible stacks and states. At a later point, we will be able to spot the distinction and discard the incorrect states. But before such a point is reached there can be multiple branches thus exponentially increasing the amount of information to be tracked.

Also since the model checking code runs in kernel, every time a function call is made, a system call may be required to indicate a need to change the state or stack which is a lot more costlier than actually making a function call in user space. This is a practical difficulty with the system.



Pushing symbols to stack: PDA

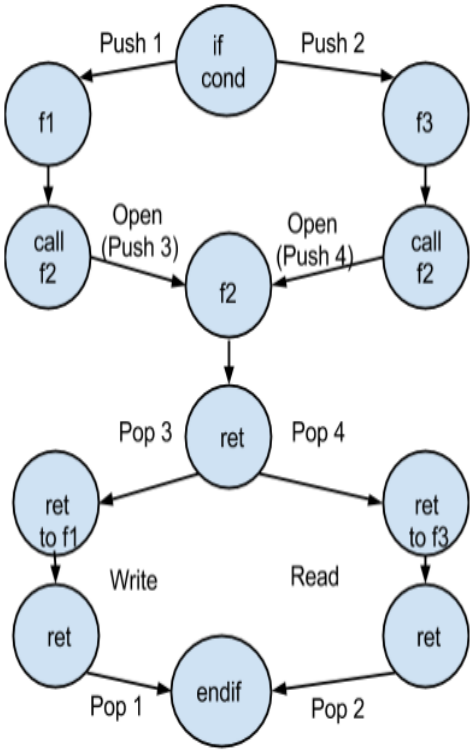
2.2 How is this overcome?

The binary is modified by adding a few instruction, so that model identifies a branch that is taken after a conditional statement. The stack is represented as a log, in the process address space. The controller keeps track of the last read position in the call. The special instructions enable the process to push special symbols that identify a branch. When a system call is made, the model checker pops the marked position in the stack and from its current state, verifies if the system can reach a state where the requested call can be allowed. This implementation does away with the need to maintain a large number of states and stacks and requires a system call (for the model checker to verify) only during when an actual system call is made. Another unique feature is that the code is changed so as to assist the PDA to predict a branch. Log is in application memory and the extra instructions write to this log.

2.3 Exact model building

Can we build a model that can perfectly predict the correct execution of a program, so that it will accept only all possible sequences of system calls? If we can do that, the model effectively replaces the application and we can run the model instead of the application.

Even if there were a model of computation that represented an application, we can not decide by executing a model checker if its exactly equivalent to the current application as the Equivalence problem of Turing Machines is undecidable. It's unrealistic to expect a fool proof model that allows only the intended system calls that an application can make.



Non-determinsm

3 Memory Safe Compilation of C

Ensures a program does not suffer from buffer overflow, even if the source code or an input makes it possible. The main challenge is to ensure that the pointer operations in a program are safe.

Memory Safe Compilation

3.1 Jones and Kelly Method

Our goal: No pointer ever goes out of bounds.
Possible basic pointer operations:

- Deference *p
- Arithmetic q = p + n
- Assignment q = p
- Assignment, Typecast q = (char *)p
- Malloc p = malloc(...);
- Free free(p);
- Assignment p = &x;

Other complex operations can be written in terms of basic operations. Assume p[5] can be rewritten as t = p+5; *t; in terms of the above operations.

If p is a safe pointer then q (on assignment) is also safe. Argument passing involves assigning actual parameters to local parameters, with similar operations. So it can be treated as a special case, without making an extra check.

- Track bounds for all objects.
- Check all potentially dangerous pointer operations.

A data structure will be added to your program to track the bounds. This supports,

- add(p, n): Allocation of a new object of size n at location p.
- (low,hi) = lookup(p);

If there is no object in the queried range we can indicate that with a special values like $(0, 0)$.
subsection Example Code

```
char * duplicate(char *p, int n){
    char * q = (char *) malloc(n);
    while(n--){
        *q++ = *p++;
    }
    return q-n;
}
```

3.2 Compiled Code

```
char * duplicate(char *p, int n){
    char * q = (char *) malloc(n);
    add(q,n);
```

```
    while(n--){
         (low, hi) = lookup(p);
         assert(low <= p && p < hi);
         (low, hi) = lookup(q);
         assert(low <= q && q < hi);
         *q = *p;
         (low, hi) = lookup(p);
         assert(low <= p+1 && p+1 < hi);
         (low, hi) = lookup(q);
         assert(low <= q+1 && q+1 < hi);
         p++;
         q++;
    }
    (low, hi) = lookup(q);
    assert(low <= q-n && q-n < hi);
    return q-n;
}
```

The caller can pass incorrect parameters that may cause a buffer overflow. The binary is compiled in such a way that any occurrence of buffer overflow leads to a termination of the program.