CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 4th, 2011

Model checking

Model checking

Model checking is a **static analysis** method. Simply put, your program is fed into an analyzer which outputs your program's *potential* bugs.

There are a couple of big **model checking projects**:

- MOPS
- SLAM, research project by Microsoft, led to the development of the **Static Driver Verifier**
- BLAST, by Berkeley, sort of a *lovingly* rip-off of SLAM

Identifying bugs using static analysis

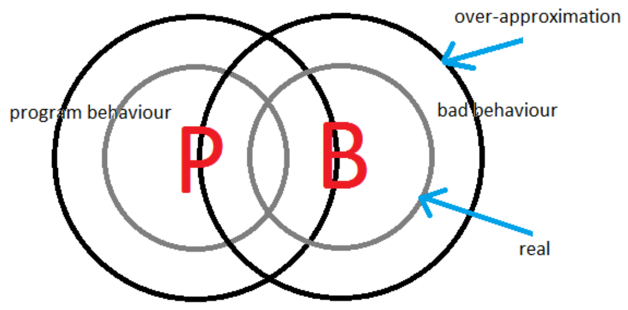
Example: Some bugs involve the order in which you do things.

example.c:
```
        setuid(0);

        if(error)
            longjmp(&jmpbuf); //    this can escape the setuid(ruid) call

        setuid(ruid);
```

Idea of model checking is to **analyze the behavior of the program**. First, we can determine the program's general behavior and then we can look at the set of all possible bad behaviors and intersect the two, obtaining the set of bad behaviors the program might exhibit:

$$BadBehaviours(prog) = GeneralBehaviour(prog) \cap AllBadBehaviours()$$

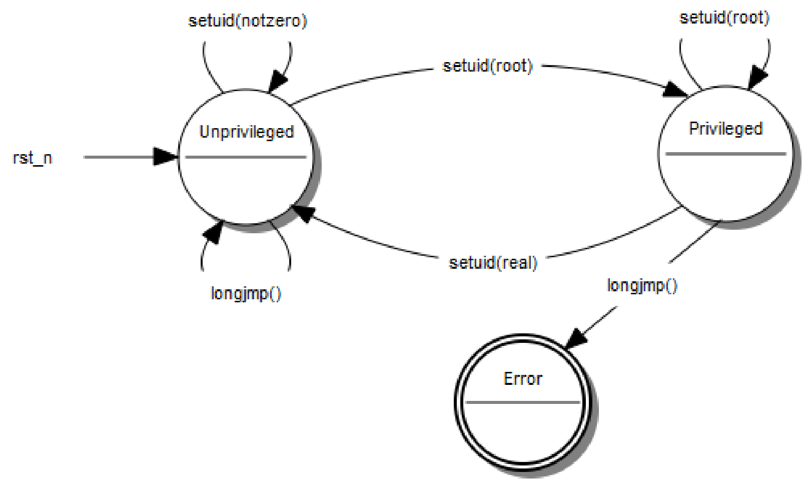
These behavior sets are often an **over-approximation**: If the larger sets have no intersection, then the real ones won't either.



Modeling a program's behavior

Hint: We can do it using state machines, since we're dealing with the **sequencing and ordering of events**, which are captured very well by state machines.

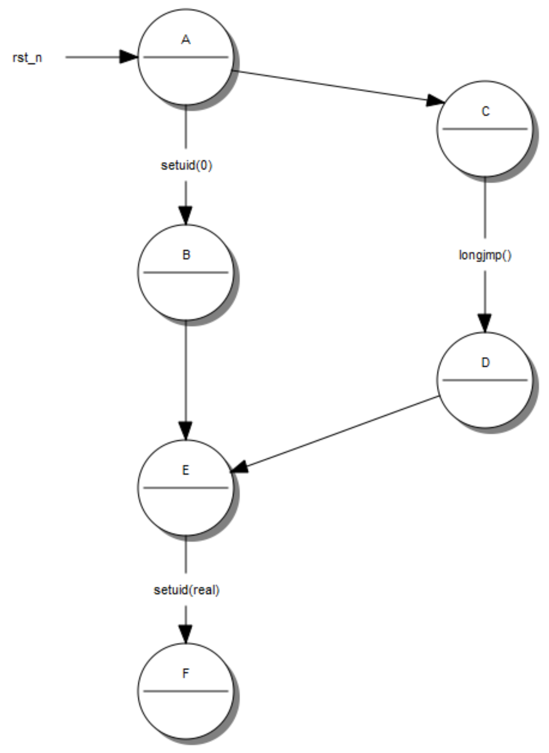
If you were to write a FSM that models a generic version of the bad behavior above, you would get this:



Issues:

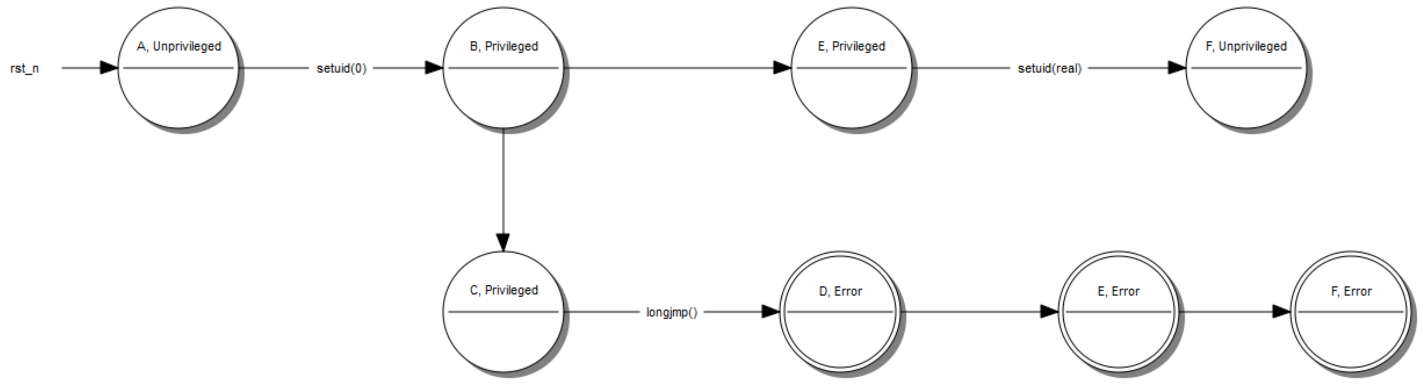
- **False positive rate:** reports an error when there's no error
  - If the FPR rate is high the programmer will turn the tool off
- **False negative rate:** doesn't catch an error in the code when there is an error
  - **Coverity**, a very popular commercial checking tool has a non-zero FNR, but it still catches a lot of bugs

We can also model a program's behavior as an FSM (we did this earlier in the semester):



By intersecting the FSM modeling the bad behavior and the program's behavior FSM, we'll get a **product FSM.** If there is a **path from the start state to the accept state** in the product FSM, then the model checking tool will report this as a possible error, and maybe also print the path to the programmer as an explanation.

CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 4th, 2011



Problem: When we modeled the program's behavior using an FSM for Intrusion Detection Systems, we had a problem with function calls. When one function was called by two different callers it would need to return to the caller, but the DFA would not "know" where to return to since it had no way of maintaining this information. We converted the DFA into a PDA, pushing the caller's identity on the stack and now the PDA would know when it reached the return state, depending on what was on the stack, where to return to, by taking the proper transition out.

Solution: We can model the program behavior as a PDA and the bad behavior as a DFA, intersect them, and get a PDA.

- In essence this is MOPS. If you take a program $P \rightarrow PDA_p$ and you take FSA_{bad}, it takes the product of these two and it determines if the language is equal to the empty set $\emptyset$.

$$PDA_p \times DFA_{bad} = PDA_{p \cap bad}$$

The big **difference between MOPS and SLAM** for instance, is that in SLAM, once an error path is found in the DFA/PDA, SLAM takes that path and it generates a predicate for each transition, feeding all the predicates to an automatic theorem prover, which tells it whether all those predicates can be true simultaneously which would lead to the program taking the erroneous path.

Conclusion: You can pick two out of three. Most tools pick two, and do their best with the third.

- No false positives,
- No false negatives,
- Your static analysis tool terminates.

Fuzzing and fault injection

These are both techniques for finding bugs that are much *lower tech* compared to the model checking method above. However, they can be just as useful.

Fuzzing is running the program with random input.

Fault injection is running the program in an environment where some things will fail and observing what happens.

Fuzzing

There was a paper called *Fuzz Revisited* where the authors built a tool called *Fuzz* which generated random input and fed it to a program, observing whether it crashed, went into an infinite look or completed successfully. The authors ran this on command line utilities of several versions of UNIX:

- NextStep (40% failure rate)

- Solaris, SunOS, HPUX, AIX (20 - 30% failure rate)
- Linux (9% failure rate)
- GNU (6% failure rate)

The **targets were command line utilities**. They also considered network server target but they never managed to crash it, most likely because they should have built a protocol specific fuzzer which would have generated valid and semi-valid SMTP (or whatever) commands.

Feeding completely random input to a program is not very effective, since chances are the input will be sanitized early in the execution of the program and you will not get good very good code coverage. Really what you want is **a spectrum between perfectly valid and totally random**. Also the commands have to be *semi-correct.*

Consider the **X network protocol** where there's **the X server** and **your application** connects via the network to your video's card driver and issues command to the server which draws stuff on the screen. The X server also sends events to the applications.

The *Fuzz revisited* authors built **a fuzzer for X**, which had three modes: perfectly correct, semi-correct and totally random. They had the fuzzer send commands to the X server and also send events to the applications. They crashed a lot of X applications, but they **never crashed the X server.**

Nowadays, there are lots of **open-source fuzzing tools** and frameworks available.

Fault injection

In your code, you call `new`, or you `malloc` memory, you `open` a file, or you do a `read` and the vast majority of the time these calls do not fail. Most **programs don't handle rejection well** (hehe) so failures are not always handled properly, which can lead to security bugs.

Fault injection is a way to test the behavior of the program under adverse conditions. You deliberately cause some actions to fail. This can be done either targeted or random.

- If it's a targeted failure, you can go into the code and modify the code to fail.
- If you want wider coverage, you can run a **fault injector** between your program and the OS.

With an **LD_PRELOAD**, you can write your own library, which gets loaded first by the loader before the program even starts, and it can intercept calls meant to go to another library:

```
void * malloc(size_t)
{       if(rand() % 2)
                return NULL;
        else
                return real_malloc(n);
}
```

LD_PRELOAD resources:

- http://developers.sun.com/solaris/articles/lib_interposers.html
- http://lca2009.linux.org.au/slides/172.pdf
- http://lca2009.linux.org.au/slides/172.pdf