Project 3

Help

White Box and Black Box Fuzz Testing

Week 5 presents static analysis and symbolic execution as two techniques for automatically finding security bugs (and other bugs) in programs.

In this lab, we will play with a symbolic executor called KLEE, an open source symbolic execution engine built on top of the LLVM compiler framwork. We will compare KLEE's ability to find memory errors to that of black box fuzzing tool, called radamsa that we see a bit more during week 6.

We will see that the symbolic executor is able to systematically explore the different control paths of the program, including paths that a black box tool may have difficulty finding. As such, symbolic executors are sometimes called "white box fuzz testers."

Setup

We will be re-using the VM from Project 1. If you did not download that VM, or you have since erased it, directions for installation can be found on the Project 1 page.

We have installed KLEE as a self-contained package in the `klee-cde-package` sub-directory of the `seed` user's home directory. To use the KLEE binaries we wil set your PATH, below, to include the `bin` directory, which is a sub-directory of the root of the KLEE package. Note that because we've installed the cde version of the package, to run the KLEE executables you'll need to add `.cde` to the end of each executable name, e.g., `llvm-gcc.cde` and `klee.cde`. We installed `radamsa` in the usual place (`/usr/bin`).

We have created modified versions of the project 1 files, as well as some other scripts to help with this lab. Download the following zip file for Project 3. You should be able to use `wget` within the virtual machine to get it, i.e.,

```
$ wget https://d28rh4a8wq0iu5.cloudfront.net/softwaresec/project3/project3.zip
```

Unzip this file in the home directory of the virtual machine. It will make a `project3` directory. Enter this directory, and run the following commands.

```
$ make wisdom-alt
$ make wisdom-alt2
```

These will build the `wisdom-alt` and `wisdom-alt2` executables in this directory.

Run all the following examples from this directory. Now go through each of the steps below, noting your answers, and then take the project quiz.

Question 1: Fuzzing

Radamsa is a fuzzer that generates random program inputs by mutating some given input. We have provided a Python script, called `fuzz.py` that connects the output from radamsa to the input of the wisdom program. Radamsa must start with some data to mutate, this data is contained in the `fuzzinput` file.

Execute the fuzzer against wisdom-alt. Use the command line:

```
$ python fuzz.py ./wisdom-alt > out && tail out
```

(This runs the fuzzing program to completion, storing its output in the file `out`, and then presenting the last bit of that file. Hit Ctrl-C to get back to the command line.)

How many iterations does it take the fuzzer to find the bug (i.e., record a crash)? What is the string that it discovers crashes the program?

See what happens if you change `fuzzinput` to something else and then rerun. Or, try editing fuzz.py and change the seed that is given to radamsa to generate different inputs, to see how that might change things.

Question 2: Fuzzing alt2

The `wisdom-alt2.c` file is the same as `wisdom-alt.c` file except that we have added an additional guard restricting the values that can index into ptrs.

```
$ diff wisdom-alt.c wisdom-alt2.c
101,102c101,104
<         fptr tmp = ptrs[s];
<         tmp();
---
>         if(s == 1 || s == 2) {
>           fptr tmp = ptrs[s];
>           tmp();
>         }
```

This change fixes the bug you found in Project 1, by only permitting 1 or 2 to be legal inputs. The question is, How does this impact the effectiveness of the fuzzer?

```
$ python fuzz.py ./wisdom-alt2 > out && tail out
```

Does this find the bug, i.e. record a crash? If so, in how many iterations?

Once again, you can adjust the seed value in the script, or try more iterations, to see what happens. (But to answer this question for the quiz leave the seed and iteration count at what they were in the download.)

Question 3: Symbolically executing wisdom-alt2

Now let's try symbolic execution on the program instead. To do this, we have to modify the program to identify which variables KLEE should consider to be symbolic. We also have to make some other cosmetic changes to get things to work properly. The result is in the file `wisdom-alt-sym.c`.

You can do `diff wisdom-alt2.c wisdom-alt-sym.c` to see the differences for yourself, but here's a summary. First, we added a function `sym_gets` that simulates the `gets` library call by filling a buffer with symbolic values, and replace the program's call to `gets` with a call to this function. Second, we removed the loop that repeatedly requests user input; this speeds up our testing because we only need one input to find the bug, and KLEE will repeatedly explore different (single) inputs. Third, we replaced the command to read the initial input, `read(infd, buf, sizeof(buf)-sizeof(char));` to instead ask KLEE to generate a symbolic value, using `klee_make_symbolic(buf, sizeof(buf), "buf");`

Now compile the `wisdom-alt-sym` program with the following command line (from within the `project3` directory):

```
$ export PATH=$HOME/klee-cde-package/bin/:$PATH
$ llvm-gcc.cde -I../klee-cde-package/cde-root/home/pgbovine/klee/include --emit-llvm -c -g wisdom-alt-sym.c
```

(Note that the first option is an uppercase I, not a lowercase L.) Now evaluate it with KLEE using the following command line:

```
$ klee.cde -exit-on-error wisdom-alt-sym.o
```

It should exit shortly and discover the error, printing a stack trace and some information about the current state. It will have created a directory `klee-last` in the current directory that contains further information about the symbolic execution. If you look in there, you will see that it generated some tests and some statistics.

The (binary) files ending in `.ktest` in this directory can be formatted intelligibly by using `ktest-tool`. Use the following command line to discover the symbolic state the error occurred in:

```
$ ktest-tool.cde klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['wisdom-alt-sym.o']
num objects: 2
object    0: name: 'AAAAAA'
object    0: size: NN
object    0: data: 'XXXXXXXX'
object    1: name: 'BBBBBBB'
object    1: size: JJ
object    1: data: 'XXXXXXXX'
```

We have replaced particular values in the above output with AAA, NN, etc. Which *symbolic variables* were involved (AAAAAA and BBBBBB in the above)? What was their contents (XXXXXXXX in the above)?

Question 4: Symbolically executing the maze

In a sense, a symbolic executor is exploring a maze defined by the program's execution space. We can make this analogy a reality by using KLEE to symbolically execute a program that asks its user to solve a maze. This program is taken from a blog post by Felipe Andres Manzano; I encourage you to check it out to go into more depth about what's going on.

The file `maze.c` defines the maze-solving program, and the file `maze-sym.c` is a slightly modified version of it, suitable for running with KLEE. Compile the maze programs both normally and symbolically:

```
$ make maze
$ llvm-gcc.cde -I ../klee-cde-package/cde-root/home/pgbovine/klee/include --emit-llvm -c -g maze-sym.c
```

Once again, the second version is different from the first in identifying data as symbolic.

```
$ diff maze.c maze-sym.c
10a11
> #include <klee/klee.h>
71c72,73
<     read(0,program,ITERS);
---
>     //read(0,program,ITERS);
>     klee_make_symbolic(program,ITERS,"program");
```

Here, the variable `program` is a buffer that contains the user's input; instead of reading it in from the user, the modified program treats the entire buffer as symbolic.

Run the (normal) maze program and see if you can find a solution to the maze. Then, run the symbolic maze program under KLEE:

```
$ klee.cde maze-sym.o
```

KLEE will work for a while and then end. The maze program will generate an assertion failure when a path through the maze has been identified, so the test that is a winning path through the maze is identified by an assert. Look for the path that solved the maze looking for a file that ends in `err`:

```
$ ls -1 klee-last | grep -A2 -B2 err
test0000AA.ktest
test0000BB.ktest
test0000NN.assert.err
test0000NN.ktest
test0000NN.pc
```

(Note the argument to `ls` is a 1 (one), not a lower-case l.) Here, AA, BB, and NN are numbers that may change depending on your VM setup (on mine, NN is sometimes 62 and sometimes 89). Use the `ktest-tool` on the `test0000NN.ktest` file (where NN is replaced with numbers from your VM) to see what path KLEE found through the maze.

```
$ ktest-tool.cde klee-last/test0000NN.ktest
ktest file : 'klee-last/test0000NN.ktest'
args       : ['maze-sym.o']
num objects: 1
object    0: name: 'program'
object    0: size: MM
object    0: data: 'XXXXXXXXXXXXXXXXXXXXXXXX\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

What was the data (the XXXXetc. part, not including any \x00\x00 parts, if any) for the program object?

It turns out there are multiple "solutions" to the maze; you can see them all by doing

```
$ klee.cde --emit-all-errors maze-sym.o
```

Then the `ls` command from above will show all the solutions. How many are there?

Question 5: Walking through walls

Are you surprised by the answer to question 4?

Something funny is going on: somehow the solution is allowed to walk through walls. Look through the code, and find the condition that allows this to happen. What line is it on? Comment it out and try again, to confirm you are getting just one solution.

Created Sat 15 Nov 2014 6:45 PM PST

Last Modified Mon 17 Nov 2014 12:23 PM PST