

Feliam's Blog

Security stuff..

The Symbolic Maze!

October 7, 2010

i

17 Votes



<http://www.twitter.com/feliam>

In this post we'll exercise the symbolic execution engine KLEE over a funny ASCII Maze (yet another toy example)!



http://en.wikipedia.org/wiki/Low_Level_Virtual_Machine

VS.

Maze dimensions: 11x7
Player pos: 1x1 Iteration no. 0
Program the player moves with
a sequence of 'w', 's', 'a' or 'd'
Try to reach the prize(#!)

```
+--+---+---+
|X|      |#| |
| |  --+  | |
| |      | | |
| +- -  | | |
|      | | |
+-----+-----+
```

The match is between a tiny maze-like game coded in C versus the full-fledged LLVM based symbolic execution engine, [KLEE](http://klee.llvm.org/Documentation.html) (<http://klee.llvm.org/Documentation.html>).

How many solutions do you think it has?

The Maze

The thing is coded in C and the impatient can download it from [here](http://pastebin.com/6wG5stht) (<http://pastebin.com/6wG5stht>). This simple ASCII game asks you first to feed it with directions. You should enter them as a batch list of actions. As “usual”; a is Left, d is Right, w is Up and s is Down. It has this looks ...

Iteration no. 2. Action: s.

```

+--+---+--+
|X|      |#| |
|X|  --+  | |
|X|      | | |
|X+--  | | |
|      | | |
+-----+--+

```

It's really small I know! But the code hides a nasty trick, and at the end, you'll see, it has more than one way to solve it.

The KLEE

KLEE is a symbolic interpreter of LLVM bitcode. It runs code compiled/assembled into LLVM symbolically. That's running a program considering its input(or some other variables) to be symbols instead of concrete values like 100 or "cacho". In very few words, a symbolic execution runs through the code propagating symbols and conditions; forking execution at symbol dependant branches and asking the companion SMT solver for path feasibility or counter-examples. For more info on this check out [this \(http://klee.llvm.org/\)](http://klee.llvm.org/), [this \(http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf\)](http://llvm.org/pubs/2008-12-OSDI-KLEE.pdf) or even [this \(www.ece.cmu.edu/~ejschwar/papers/oakland10.pdf\)](http://www.ece.cmu.edu/~ejschwar/papers/oakland10.pdf).

Find it interesting? Keep reading!

The idea

Use KLEE to automatically solve our small puzzle.

Dissecting the code

Lets take a walk through the maze code. First it hardcodes the map as a static global rw variable.

```
#define H 7
#define W 11
char maze[H][W] = { "+-+---+---+",
    " | |   |#",
    " | |  --+ | |",
    " | |   | | |",
    " | +-+ | | |",
    " |   | | |",
    "+-----+---+" };
```

Sets up a convenient function to draw the maze state on the screen...

```
void draw ()
{
    int i, j;
    for (i = 0; i < H; i++)
    {
        for (j = 0; j < W; j++)
            printf ("%c", maze[i][j]);
        printf ("\n");
    }
    printf ("\n");
}
```

On the main function there are local variables to hold the position of the "player", the iteration counter, and a 28bytes array of the actions...

```
main (int argc, char *argv[])
{
    int x, y;      //Player position
    int ox, oy;    //Old player position
    int i = 0;     //Iteration number
    #define ITERS 28
    char program[ITERS];
```

The initial player position is set to (1,1), the first free cell in the map. And the player 'sprite' is the letter 'X' ...

```
x = 1;
y = 1;
maze[y][x]='X';
```

At this point we are ready to start! So it asks for directions. It reads all actions at once as an array of chars. It will execute up to ITERS iterations or commands.

```
read(0,program,ITERS);
```

Now it iterates over the array of actions in variable 'program'...

```
while(i < ITERS)
{
    ox = x;    //Save old player position
    oy = y;
```

Different actions change the position of the player in the different axis and directions. As "usual"; a is Left, d is Right, w is Up and s is Down.

```

switch (program[y])
{
    case 'w':
        y--;
        break;
    case 's':
        y++;
        break;
    case 'a':
        x--;
        break;
    case 'd':
        x++;
        break;
    default:
        printf("Wrong command!(only w,s,a,d accepted!)\n");
        printf("You lose!\n");
        exit(-1);
}

```

Checks if the prize has been hit! If affirmative... You win!

```

if (maze[y][x] == '#')
{
    printf ("You win!\n");
    printf ("Your solution \n",program);
    exit (1);
}

```

If something is wrong do not advance, backtrack to the saved state!

```

if (maze[y][x] != ' ' &&
    !((y == 2 && maze[y][x] == '|' && x > 0 && x < W)))
{
    x = ox;
    y = oy;
}

```

```

    if (ox==x && oy==y){
        printf("You lose\n");
        exit(-2);
    }

```

Ok, basically if we can move.. we move! Put the player in the correct position in the map. And draw the new state.

```

    maze[y][x]='X';
    draw ();          //draw it

```

Increment the iteration counter (used to select next action in the array), wait a second and loop.

```

    i++;
    sleep(1); //me wait to human
}

```

If you haven't won so far.. you lose.

```

printf("You lose\n");
}

```

Ok, that's all of it.

By hand...

Now considering you have it in maze.c. It should compile with a line like this

```
gcc maze.c -o maze
```

Run it! In a couple of tries you'll get to the priceless '#'. Maybe using this solution:

```
ssssdddddwwaawdddddssssddwww
```



<https://feliam.files.wordpress.com/2010/10/maze3.gif>

By KLEE

Let's see if KLEE is able to find the solution. First, for even start thinking about KLEE we need to get a copy of the LLVM toolchain, and compile our maze to LLVM bitcode. Here we have use LLVM 2.7 and llvm-gcc. You may want to take a tour to KLEE's official tutorials [here](http://klee.llvm.org/Tutorial-1.html) (<http://klee.llvm.org/Tutorial-1.html>). Once you have the LLVM thing in place, a compile and test cycle for the maze.c using LLVM will be like this...

```
llvm-gcc -c -emit-llvm maze.c -o maze.bc  
lli maze.bc
```

That will run the LLVM bitcode representation of our maze in the interpreter. But for testing it with KLEE we need to mark something in the code as symbolic. Let's mark all maze inputs as symbolic, that's the array of actions the maze code reads at the very beginning of the main function. KLEE will gain 'symbolic control' over the array of actions. In code, that's done by changing this line ...

```
read(0, program, ITERS);
```

... by ...

```
klee_make_symbolic(program, ITERS, "program");
```

Also you will need to add the klee header at the beginning of the code...

```
8 of 16  
#include <klee/klee.h>
```


Now KLEE will find every possible code/maze path reachable from any input. If some of those paths lead to a typical error condition like a memory failure or such, KLEE will signal it!

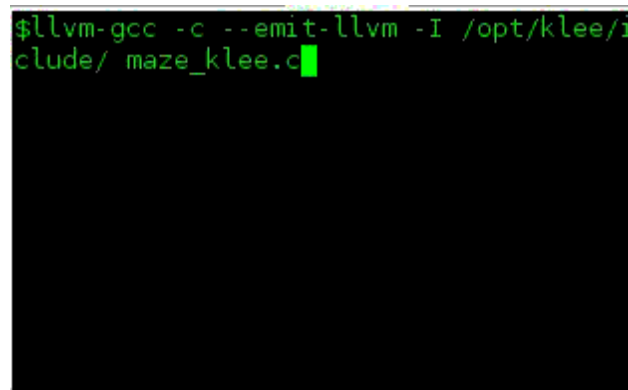
Symbolic execution, the chamigo way:

- Say.. every input is marked as a symbol.
 - Not the concrete value like 1 or “cachho”, but a symbolic variable representing every possible value.
 - Then the program evolves...adding restrictions to this symbols.
 - At some point it may face a branch that depends on such symbols.
 - On that case it checks feasibility of the different paths using a SMT solver.
 - If feasible, then it dives into each path repeating this basic algorithm
 - Of course if an error cond is reached, the SMT solver is asked for a way to reach that specific spot
- Hello, is mr. memory corruption here?! Let's give it a try...

```
llvm-gcc -c -Ipath/to/klee -emit-llvm maze_klee.c -o maze_klee.bc
```

```
klee maze.bc
```

Here there is the screen cast of the a run...



(https://feliam.files.wordpress.com/2010/10/maze_klee.gif)

As you could check at the end of the demo, KLEE finds 321 different paths...

KLEE: done: total instructions = 112773

KLEE: done: completed paths = 321

KLEE: done: generated tests = 318

... and it throws the test cases to generate all them to the klee-last folder...

```
$ls klee-last/
```

```
assembly.ll test000078.ktest test000158.ktest
```

```
info test000079.ktest test000159.ktest
```

messages.txt test000080.ktest test000160.ktest
 run.istats test000081.ktest test000161.ktest
 run.stats test000082.ktest test000162.ktest
 test000001.ktest test000083.ktest test000163.ktest
 test000075.ktest test000155.ktest warnings.txt
 Each test case could be retrieved with the ktest-tool like this...

```
$ktest-tool klee-last/test000222.ktest
ktest file : 'klee-last/test000222.ktest'
args : ['maze_klee.o']
num objects: 1
object 0: name: 'program'
object 0: size: 29
object 0: data: 'ssssdddddwwaawdddddssssddwwwd\x00'
```

So in this case you may take that input to the original maze and check what it does.

Ok, so far so good but I'm not ktest-tooling every possible test case and check if it is a maze solution! We need a way for KLEE to help us tell the normal test cases apart from the ones that actually reaches the "You win!" state.

Note also that KLEE haven't found any error on the maze code. By design KLEE will issue a warning when any "well known" error condition (like a wrongly indexed memory access) is detected.

How to flag the portion of code we are interested in?

There is a `klee_assert()` function that pretty much do the same thing that a common C `assert`, it forces a condition to be true otherwise it aborts execution! You could check out the complete KLEE C interface [here \(https://llvm.org/svn/llvm-project/klee/trunk/include/klee/klee.h\)](https://llvm.org/svn/llvm-project/klee/trunk/include/klee/klee.h). But we already have what we need... a way to mark certain program part (with an `assert`) so KLEE will scream when it reach it.

In the code, that's done by replacing this line ...

```
printf ("You win!\n");
```

... by this two ...

```
printf ("You win!\n");
klee_assert(0); //Signal The solution!!
```

Now KLEE will assert a synthetic failure when it reaches the “You win state” (that means the player hit the #). OK, if you compile it to LLVM and run KLEE on the new version it flags one test case as being also an error...

```
$ls -l klee-last/ | grep -A2 -B2 err
```

```
test000096.ktest
```

```
test000097.ktest
```

```
test000098.assert.err
```

```
test000098.ktest
```

```
test000098.pc
```

Let's see what's the input that triggers this error/maze solution...

```
$ktest-tool klee-last/test000098.ktest
```

```
ktest file : 'klee-last/test000098.ktest'
```

```
args : ['maze_klee.o']
```

```
num objects: 1
```

```
object 0: name: 'program'
```

```
object 0: size: 29
```

```
object 0: data: 'sddwddddssssddwww\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

So it propose the solution...

sddwddddssssddwww

HEY! That's odd, it seems too short to even reach the other end of the maze! Lets try that input on the original maze...

Well in most cases (apparently) you need only one way to reach an error condition, so KLEE wont show you the other ways to reach the same error state. We desperately need to use one of the 10000 KLEE options (<http://pastebin.com/tDPGNn9D>). We need to run it like this..

```
$klee -emit-all-errors maze_klee.o
```

Check out the KLEE crazy run...

Now it gives 4 different “solutions”...

```
$ktest-tool klee-last/test000097.ktest
```

```
ktest file : 'klee-last/test000097.ktest'
```

```
args : ['maze_klee.o']
```

```
num objects: 1
```

```
object 0: name: 'program'
```

```
object 0: size: 29
```

```
object 0: data: 'sddwddddsddw\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

```
$ktest-tool klee-last/test000136.ktest
```

```
ktest file : 'klee-last/test000136.ktest'
```

```
args : ['maze_klee.o']
```

```
num objects: 1
```

```
object 0: name: 'program'
```

```
object 0: size: 29
```

```
object 0: data: 'sddwddddsddssddwww\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

```
$ktest-tool klee-last/test000239.ktest
```

```
ktest file : 'klee-last/test000239.ktest'
args : ['maze_klee.o']
num objects: 1
object 0: name: 'program'
object 0: size: 29
object 0: data: 'ssssdddddwwaawwddddsddw\x00\x00\x00\x00\x00\x00\x00'
$ ktest-tool klee-last/test000268.ktest
ktest file : 'klee-last/test000268.ktest'
args : ['maze_klee.o']
num objects: 1
object 0: name: 'program'
object 0: size: 29
object 0: data: 'ssssdddddwwaawwdddddssssddwww\x00'
There are 4 possible solutions!!
```

1. **ssssdddddwwaawwdddddssssddwww**
2. **ssssdddddwwaawwddddsddw**
3. **sddwdddddssssddwww**
4. **sddwddddsddw**

Conclusion

Better to use symbolic execution than to do manual code exploration or even code an error prone ad-hoc solution searcher. Fuzzing for it may be unfeasible here even restricting the input to the interesting characters... but I'm not sure.

Comments and corrections are very welcome!!

f/



Posted by feliam

Filed in [security](#), [symbolic execution](#) · Tags: [ascii](#), [game](#), [klee](#), [llvm](#), [maze](#), [symbolic execution](#)

[7 Comments »](#)

7 Responses to “The Symbolic Maze!”

1. *Rex* said

[December 6, 2010 at 5:18 am](#)

Awesome! I am just trying to wriggle my way into klee and this is quite a nice job. I am wondering if there is another way to flag a desired code point besides asserting a false?

[Reply](#)

2. *Frank M. Eriksson* said

[March 30, 2011 at 1:52 am](#)

Fantastic!

I actually missed the “error” when I first skimmed through your maze code (well it is 03:42 at the night here, I probably should sleep someday soon), but I was able to figure out the `sddwddddsddww` solution when I read the dissected code above

Seems like KLEE is a better option than the strace / valgrind combination that I normally use, but I does only run them after I’ve become painfully aware of a bug in my code.

14 of 16

12/03/2014 08:41 PM

I guess that I maybe can use KLEE for a more automatic approach to finding bug – even before I’m painfully aware of them!

Reply

○ *Renjian Li* said

April 16, 2011 at 3:57 am

Huh, it seems that I encounter the same problem with you. Klee could find the solution, but its output may have some problems. My running covers only 309 completed paths, not 321.

Reply

○ *feliam* said

April 16, 2011 at 5:03 pm

That might have something to do with the `-emit-all-errors` parameter.

I think that by default klee wont show you more than one way to reach a given 'error' condition. Check what happens when you add that ...

[EDIT]

Ah no, you got less "completed paths" in total than the example from the post. That may have to do with some optimization or difference in the to-llvm compilation?? I've used the 2.6 release of llvm+klee...

`$ klee -version`

Low Level Virtual Machine (<http://llvm.org/>):

llvm version 2.6

Optimized build with assertions.

Built Mar 31 2010(20:46:55).

Registered Targets:

x86 - 32-bit X86: Pentium-Pro and above

x86-64 - 64-bit X86: EM64T and AMD64

`$ llvm-gcc -version`

llvm-gcc (GCC) 4.2.1 (Based on Apple Inc. build 5649) (LLVM build 2.6)

○ *Tomek* said

October 13, 2012 at 9:12 pm

Hi, I also had 309 paths but after I changed ITERs from 28 to 29 it's 321 now.

Great! This just gives me an intuition on how Klee can be used to find sorts of problems.

[Reply](#)

4. *Grzegorz Wierzowiecki* said

December 11, 2011 at 8:02 pm

Great tutorial ! Turning stuff into play and fun is my favourite way in computing. I am so happy to see such a nice and very educative example.

[Reply](#)

[The Ambiru Theme. Blog at WordPress.com.](#)

Follow

Follow “Feliam's Blog”

Build a website with WordPress.com