Lab 7: KLEE Symbolic Execution Engine

Fall Semester 2021

Due: 13 December, 8:00 a.m. Eastern Time

Corresponding Lecture: Lesson 11 (Dynamic Symbolic Execution)

Objective

The goal of this lab is to gain familiarity with a research based symbolic execution engine called KLEE. KLEE is a hybrid analysis engine built on the LLVM compiler infrastructure. This engine performs symbolic execution directly on LLVM bitcode, a proprietary C/C++ intermediate representation (IR).

KLEE leverages the power of a SMT solver to generate concrete inputs that correspond to specific paths through the given program, allowing the user to target specific paths in the given program that are of interest. KLEE can also alert the user to certain types of errors, like illegal array accesses and division by zero errors through analysis of the symbolic state of a program at a given point.

Resources

1. KLEE Site:
   `http://klee.github.io/`
2. KLEE Tutorials (very similar to our demo programs):
   `http://klee.github.io/tutorials/`
   `https://klee.github.io/tutorials/testing-regex/`
   `https://doar-e.github.io/blog/2015/08/18/keygenning-with-klee/`

3. KLEE Documentation:
   `http://klee.github.io/docs/`

Setup

In this lab, we will be running KLEE in an isolated Docker container. Neither Docker nor KLEE are pre-installed on the course VM. We have provided an installation script (`install_klee.sh`) that will perform the necessary updates to your VM.

First, download and extract the lab resources (`klee.zip`) provided on Canvas into the `~/` directory. It should create the following files / directories:

1. `~/install_klee.sh` - A shell script that will update your VM with the necessary software to complete this lab
2. `~/init_klee.sh` - A shell script that will initialize the KLEE container within Docker
3. `~/klee/RegexDemo/` - A program for demonstrating KLEE's bug finding capabilities

4. `~/klee/PasswordDemo/` - A program for demonstrating KLEE's constraint solving capabilities
5. `~/klee/Part1/` and `~/klee/Part2/` - Programs you will analyze with KLEE in the graded portion of this lab

Next, run the VM update script in the `~/` directory:

```
sudo ./install_klee.sh
```

You will see a significant amount of output to the console, but you should not see any error messages. Next, confirm that the script has successfully executed by initializing the KLEE Docker image with the following command:

```
sudo ./init_klee.sh
```

This command will create a new container from the KLEE default image, including mapping the `~/klee` directory as a shared folder between the host OS and the container. You see your command prompt change from the Ubuntu prompt to the KLEE prompt, indicating you are now issuing commands to the container and not the host OS.

Next, execute the `exit` command, which will return your prompt to the host OS. When you want to return to the KLEE container, execute this command in a terminal:

```
sudo docker start -ai klee_container
```

If you run into issues with the container, you can remove it with the command below (you must `exit` the container first). Make sure your work is in the shared folder so it isn't lost when you remove the container.

```
sudo docker rm klee_container
```

After removal, you can create a fresh container by re-invoking the `init_klee` script above.

Demonstration Part 1 - Finding a Bug with KLEE

In this demonstration, we will run KLEE on two simple programs to highlight the engine's strengths. The purpose is to demonstrate how to instrument a program with KLEE constructs in order to support symbolic execution and how to generate the LLVM bitcode necessary for running KLEE.

First, enter the KLEE container you created above and navigate to the `~/klee/RegexDemo` directory. In this directory, you will find a C program named `Regexp.c` containing a few Regex library functions and a KLEE test harness consisting of a simple `main` function and KLEE library calls to define which program inputs should be considered symbolic. In order to symbolically execute this code with KLEE, we must first generate LLVM bitcode from the program source. Do this by executing the following command:

```
clang -I ../../klee_src/include -emit-llvm -c -g Regexp.c
```

This will generate a file named `Regexp.bc`. Next, run KLEE on this program with the following command:

```
klee --only-output-states-covering-new Regexp.bc
```

The console output of the symbolic execution should report a memory error, the total number of instructions and program paths analyzed, and the number of concrete test cases generated. Additionally, KLEE will create a numbered subfolder containing data generated during execution. This includes test cases to exercise paths in the program, test cases to demonstrate errors in the program, run statistics, and other information about the run.

Next, let's view the test cases generated by KLEE, specifically the test case where an error was found. Navigate to the output subfolder, `klee-out-0`. Error log files (`*.err`) are plain text files and can be opened in a text editor. In order to view the `*.ktest` files, we need to use a command line utility provided by KLEE. Run the following command on the test case file with the same number as the error log file:

```
ktest-tool --write-ints [test case file]
```

The output of this command will display the concrete value for the symbolic object we defined - which you can use as an input to recreate the error. For this demo, we have not included an uninstrumented version of the program for you to compile and run with the sample input, but you will have an opportunity to do so later.

In this case, the error log file points us to the `matchhere` function in our program, which is missing a check for a null string passed as the `re` parameter. Without this check, the first statement looking for the '`*`' character at index 1 in the character array can access memory outside the bounds of the variable. KLEE has helped us find this error statically, allowing us to correct it instead of stumbling across it later during QA or when the code goes to production.

Demonstration Part 2 - Path Exploration with KLEE

Let's examine another sample program. In this scenario, we are trying to determine the behavior of a program, `Password.c` that has been obfuscated (made unreadable by a human programmer).

Our program is checking the user supplied password for a particular value, however instead of comparing this value to a string or a value read from a resource like a file or database, the password is implemented in obfuscated code. We are trying to determine a set of inputs that cause the program to reach an interesting program point, where the program prints `Successful Login!`.

We could manually inspect the code and reverse engineer the input; however it is much faster to run KLEE and use it to determine the input constraints necessary to reach this path in code. KLEE's SMT solver can then generate a concrete input that satisfies these constraints, which is the password we are looking for.

First, navigate to the `~/klee/PasswordDemo` directory in the KLEE container and take a look at the source code. It has been instrumented with the necessary KLEE functions to symbolically execute the code. Specifically, on line 52, we have added a KLEE assert statement that causes KLEE to treat this line as a failure. Generate the LLVM bitcode for this program in same manner as we did for the first demonstration.

Next, run KLEE on the generated bitcode with the command below. Since our symbolic input is a parameter passed when we execute the program and not a variable in code, we must specify the symbolic input when we run KLEE, as well as set a few other flags to tell KLEE how to handle standard library calls:

```
klee --only-output-states-covering-new --libc=uclibc --posix-runtime Password.bc --sym-arg 100
```

When KLEE runs you should see the output of the symbolic runs, including one error resulting from our assert statement. Using the `ktest-tool` as we did in the first part of this demonstration, open the test case file associated with run reporting the error. The test case file indicates that the null terminated string `Pandi_panda` is the password we were searching for.

Lab Instructions

Part 1 - Breaking a Weak Hash Function

In the first part of this lab, you will assume the role of a security researcher and use KLEE to break a weak hash function. You have access to the source code used to generate hashes of input strings, as well as a hash generated by this code. Your goal is to use KLEE to instrument the program in order to generate an input string to the program that results in your example hash being generated.

First, enter the KLEE container you created above and navigate to the `~/klee/Part1` directory. In this directory, you will find a C program named `SimpleHash.c`. This program is a test harness you have built that includes the hash generation code and a simple `main` function that you can use to generate hashes from input strings. You will find it helpful to build an uninstrumented program that you can use to familiarize yourself with how the program works, and later test the inputs KLEE generates to ensure they are correct. You can build the uninstrumented version of the hash program with the following command:

```
clang -I ../../klee_src/include -o SimpleHash SimpleHash.c
```

Your goal is to instrument the program with KLEE to compute an input string to `SimpleHash` that causes it to compute the hash code `80091566` value under the following input restrictions:

- Input string must be exactly 20 characters long, not including the null terminator
- Input string can only contain lowercase and uppercase letters and numerical digits ([a-z], [A-Z], [0-9])

You will need to add `klee_assume` and `klee_assert` instructions to the program to set restrictions on the expected input and output of the program. As shown in the examples, using `klee_assert` to trigger an error on a specific path in the program is helpful in identifying a test case of interest. You should use `klee_assume` commands to encode input restrictions for the symbolic execution engine to help KLEE narrow the input constraints and speed up its performance. KLEE should not run for an excessively long amount of time - a valid string should be generated in under a minute.

Once the file is instrumented, you need to generate the LLVM bitcode from the program using the following command:

```
clang -I ../../klee_src/include -emit-llvm -c -g SimpleHash.c
```

Next, symbolically execute your instrumented program using KLEE with the command below (all on one line):

```
klee --only-output-states-covering-new --libc=uclibc --allow-external-sym-
calls --posix-runtime SimpleHash.bc --sym-arg 20
```

Navigate to the output directory and use the `ktest-tool` command to view your generated tests and identify the input string that generates the specified hash code. You can verify that the input does indeed generate the specified hash by running your uninstrumented version of the program with the value generated by KLEE and verifying the generated hash is the value provided.

Additional Notes:

- The resulting string generated by KLEE will actually be 21 characters long as it will include the terminating null character
- You will likely receive two errors for a successfully instrumented file. The first will mention a provably false `klee_assume` command. This can be ignored but ensure you reference the correct ktest file
- When using multiple constraints in a single `klee_assume`, it is safest to use bitwise operators `|` and `&` instead of `||` and `&&`
- You should not use `klee_make_symbolic` in your instrumented program. It will cause the autograder to incorrectly interpret your error tests and you will not receive credit
- You must use `klee_assert` to properly generate an assertion error. We will be using the `test{num}.ktest` with the same number as the `test{num}.assert.err` to validate that your instrumented code has generated a correct string
- Ensure your program completes its full run in less than a minute. Do not use flags such as `--exit-on-error` or `--exit-on-error-type` to force KLEE to exit early, as we will not be using these flags when running the grader on your programs

Grading

- We will be running your instrumented program using the KLEE command provided to you and ensuring that it completes running before the timeout limit is reached. We will give a small buffer when running on the grading computer to allow for differences in computing power
- We will be validating that your instrumented code generates a `test{num}.assert.err` file and that the .ktest file associated with that file contains proper input that will generate the correct hash and that meets the other criteria specified for Part 1
- We will be validating your submitted error test file against the uninstrumented version of the program to ensure that it contains input that will generate the correct hash and that meets the other criteria specified for Part 1

Part 2 - Identifying Obfuscated Program Behavior Code

In this part of the lab, you will assume the role of a security manager and use KLEE to identify obfuscated program behavior in the program source code. One of your users reports that a utility program they use to shortcut routine tasks exhibits some suspicious behavior. Specifically, the user reports that the utility intermittently writes a suspicious file to disk in the `/tmp/` directory. This behavior has been obfuscated in the source code, and the user cannot reproduce the behavior. Your goal is to symbolically execute the program with KLEE to identify the class of inputs that causes the file write behavior.

First, enter the KLEE container you created above and navigate to to the `~/klee/Part2` directory. In this directory, you will find a C program named `Utility.c`. As with Part 1, it will be helpful to build an uninstrumented version of the program to test inputs KLEE generates for you. You can build an uninstrumented version of the program with the following command:

```
clang -I ../../klee_src/include -o Utility Utility.c
```

You should now see a program named `Utility` in your working directory. You can use this program to familiarize yourself with its functions.

Next, analyze the source code and insert KLEE statements to identify a test case which causes the program to write the suspicious file. As shown in the demos, you will find it useful to use `klee_assert(0)` to cause KLEE to generate a test case labelled as an error when it explores a certain path through code.

Once you have inserted your KLEE statements, generate LLVM bitcode and run KLEE. Pay close attention to the number and type of command line arguments that can be used by the `Utility` program and ensure that your KLEE command properly generates the correct range of symbolic arguments for the program. Here is a sample KLEE command for this part of the lab:

```
      klee --only-output-states-covering-new --libc=uclibc --allow-external-
sym-calls --posix-runtime Utility.bc --sym-args <MIN> <MAX> <N>
```

NOTE: You will need to replace the `<MIN>`, `<MAX>`, and `<N>` values with integers. A suggested set of values is `0`, `3`, `40`, which we will use for grading. Further specification of the `--sym-args` flag is in the KLEE documentation. You should be able to understand why we provided these values to you to use although we will not be asking for a written response.

You can verify that the input generated by KLEE causes the suspicious file write by running the uninstrumented program with the input and checking the `/tmp/` directory **inside the Docker container** for the suspicious file. Don't be afraid to experiment with different instrumentations and run KLEE several times if your first attempt doesn't succeed. The 'suspicious file' will have a fairly obvious filename when you see it.

Additional Notes:

- The goal of this part of the lab is to determine how to constrain the input with your `klee_assume` statements. Taking the approach of blocking the suspicious file write, or assuming something about the output file or output path is an incorrect approach and will not receive full credit
- To be exceedingly clear, do not check for a file written in /tmp/ in your code. Your instrumentation should be completely agnostic as to where the file is being written. In other words, if the file were written in any other directory your instrumentation should still work. The only reason we tell you the file will be written in /tmp is so you know while performing your analysis
- You should not block the suspicious file being written as we will be grading your instrumented software in part by ensuring that it works as expected on specific input that we will test against. Note that this statement is referring to the instrumented version of the code, which includes the klee_assume and klee_assert statements. Make sure you are aware of what each does when run and how it affects the execution of your program
- If you attempt to solve this problem using only `klee_assert` statement(s), it is unlikely both that your lab will complete in under a minute and that you have properly constrained the input
- You should be looking to constrain to an entire class of inputs. Making an itemized list of inputs that will cause the suspicious file to be written is an incorrect approach. Additionally, a subset of the pattern is not correct. You should be looking to identify the pattern in the inputs which cause the suspicious file to be generated and attempt to instrument the code to identify that class of inputs. To further explain what is being said here with the use of an abstract example, assume that the pattern to identity is any animal. If you list out *cat* and *dog* that is an itemized list and will not properly identify the pattern

of any animal because it would reject ant. If you say any animal with a name of 3 letters, that is a pattern but still does not properly identify the full pattern of any animal because it would miss *alligators* and in fact would only correctly identify a tiny fraction of all animals

- Given the previous point, once you find a failing pattern with KLEE, you should spend time trying variations of that pattern on the uninstrumented version of the program until you are confident that you know the actual pattern and not a simplified pattern. The most frequent issue we see in submissions is students who have excluded possible failing inputs, likely because they did not spend enough time making sure they understand the pattern
- While your implementation should be able to generate any failing pattern, in practice, KLEE will generate the first pattern it finds that meets the criteria so you will usually see the simplest case in your KLEE output unless you specifically block that case
- You should not use `klee_make_symbolic` in your instrumented program. It will cause the grader to incorrectly interpret your error tests and you will not receive credit
- You must use `klee_assert` to properly generate an assertion error. We will be using the `test{num}.ktest` with the same number as the `test{num}.assert.err` to validate that your instrumented code has generated a correct string.
- Ensure KLEE can complete its full run in less than a minute using the suggested set of `--sym-args`. Do not use flags such as `--exit-on-error` or `--exit-on-error-type` to force KLEE to exit early, as we will not be using these flags when running the grader on your programs using a similar set of `--sym-args` arguments as we have provided to you

Grading

- We will be running your instrumented program using the KLEE command provided to you and ensuring that it completes running before the timeout limit is reached. We will give a small buffer when running on the grading computer to allow for differences in computing power
- We will be validating that your instrumented code generates a `test{num}.assert.err` file and that the .ktest file associated with that file contains proper input that will cause the suspicious file to be written
- We will also be testing your instrumented code against specific input parameters to ensure that the instrumented code handles the tests properly. We will not be disclosing the sets of test parameters prior to grading
- We will be validating your submitted error test file against the uninstrumented version of the program to ensure that it contains input that will cause the suspicious file to be written.

Items to Submit

Submit the following files in a single compressed file (`.zip` format) via gradescope. For full credit, there must not be any additional subfolders or extra files contained within your zip file. This zip file should contain exactly two folders and four total files as indicated below.

1. Contained within a folder named `Part1`:
   a. (35 points) Submit your version of `SimpleHash.c` that you instrumented for use with KLEE
   b. (15 points) Submit your `test{num}.ktest` file that contains an input to `SimpleHash` that results in the correct hash value being generated by the program. Please ensure you only submit a single error test file for this part. This file should correspond to a `test{num}.assert.err` file
2. Contained within a folder named `Part2`:
   a. (35 points) Submit your version of `Utility.c` that you instrumented for use with KLEE
   b. (15 points) Submit your `test{num}.ktest` file that contains an input to `Utility` that results in the suspicious file write. Please ensure you only submit a single error test file for this part. This file should correspond to a `test{num}.assert.err` file

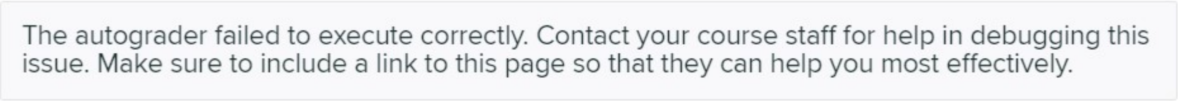
The full structure of your zip file should be:

```
Part1/
    SimpleHash.c
    test{num}.ktest
Part2/
    Utility.c
    test{num}.ktest
```

There should be no `lab7` or `Lab7` or any other parent folders.

Through Gradescope, you will have immediate feedback from the autograder as to whether the submission was structured correctly. The public test cases will confirm that your submissions compiled only. You will get your final score after the assignment due date.

We expect your submission to conform to the folder and files structure specified above. Gradescope will check folder structure and names match what is expected for this lab, but it won’t (and isn't intended to) catch everything. If you get the error below in Gradescope, please confirm folder structure prior to reaching out to instructors.



The autograder failed to execute correctly. Contact your course staff for help in debugging this issue. Make sure to include a link to this page so that they can help you most effectively.

Origin of the Name

KLEE is a reference to the artist Paul Klee: https://en.wikipedia.org/wiki/Paul_Klee

Similar Tools

Automating jUnit creation in Java using Symbolic Execution: https://github.com/osl/jcute

How Do We Use It?

Dynamic Symbolic Execution is a modern technique used in security audits especially of projects operating in hostile environments. For example, in the blockchain ecosystem, smart contracts are under constant attack by hackers looking for financial profit. Professional auditing companies have begun using symbolic execution and open sourcing their tools for automated bug detection. Most famously, trail of bits released an open-source symbolic execution tool called Manticore for auditing smart contracts on the Ethereum blockchain. Dynamic symbolic execution methods such as those in klee are becoming one of the standard tools for auditing smart contracts in this space.