Assignment 1– LLVM Playground (Part 4)

HSS
Fall 2023

In this part, you will build a "division-by-zero" dynamic analyzer for the C language using the LLVM framework.
Logistics:

- **LLVM Primer:** Please make sure that you have skimmed the LLVM Primer presentation (access it from the course webpage) to know the capabilities of LLVM.
- **Using Sanitizers:** Read the article on Hardening C/C++ Code with Clang Sanitizers [1] which surveys pre-existing sanitizers that target common kinds of programming errors. The dynamic analyzer you will build is a sanitizer that targets "division-by-zero" errors..
- **Setup Repo:** I have created a `github` repo with all the necessary scripts to install LLVM, Z3 and starter code to write a pass. You can access it at: `https://github.com/HolisticSoftwareSecurity/hssllvmsetup`. The repo has examples of analysis (i.e., the passes that do not modify the IR) and instrumentation (i.e., the passes that modify the IR) passes.
- **Development Environment:** I use CLion (`https://www.jetbrains.com/clion/`) while working with LLVM and strongly suggest you to use it. You can get unlimited access using your `@purdue.edu` email.

Setup

The skeleton code for this part is located under `part4_instrumentation` folder of the following repo. We will frequently refer to this top level directory as `part4` when describing file locations for the lab.

Repo

`https://github.com/HolisticSoftwareSecurity/LLVMPlayground`

Step 1

Clone the above repository to a folder:

```
$ cd
$ git clone https://github.com/HolisticSoftwareSecurity/LLVMPlayground.git
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 14 (delta 11), reused 11 (delta 8), pack-reused 0
Unpacking objects: 100% (14/14), done.
```

[1] `https://microblink.com/be-wise-sanitize-keeping-your-c-code-free-from-bugs/`

Step 2

Build the pass using the `CMakeLists.txt` as shown below:

```
$ cd ~/LLVMPlayground/part4_instrumentation
$ mkdir build && cd build
$ cmake ..
$ make
```

Among the files generated, you should now see `InstrumentPass.so` in the `build/DivZeroInstrument` directory, from code that we have provided in `DivZeroInstrument/src/Instrument.cpp` (which you will modify in this part), and an auxiliary runtime library, named `libruntime.so` that contains functionality to help you complete the lab.

Step 3

Before running the pass, the LLVM IR code must be generated:

```
$ cd ~/LLVMPlayground/part4_instrumentation/DivZeroInstrument/test
$ clang -emit-llvm -S -fno-discard-value-names -c -o simple0.ll simple0.c -g
```

The second line (clang) generates vanilla LLVM IR code from the input C program simple1.c.

Step 4

you will implement your analyzer as an LLVM pass, called `InstrumentPass`. Use the `opt` command to run this pass on the optimized LLVM IR program as follows:

```
DivZeroInstrument/test $ opt -load ../../build/DivZeroInstrument/InstrumentPass.so
-Instrument -S simple0.ll -o simple0.instrumented.ll
```

The produced program in simple0.instrumented.ll should be identical to simple0.ll but it will cease to be so once you implement the functionality of this lab:

```
DivZeroInstrument/test $ diff simple0.instrumented.ll simple0.ll
1c1
< ; ModuleID = 'simple0.ll'
---
> ; ModuleID = 'simple0.c'
```

Step 5

Next, compile the instrumented program and link it with the provided runtime library to produce a standalone executable named `simple0`:

```
DivZeroInstrument/test $ clang -o simple0 -L../../build -lruntime simple0.instrumented.ll
```

Step 6

Finally run the executable on the empty input; note that you may have to manually provide test input for programs that expect non-empty input:

```
DivZeroInstrument/test $ ./simple0
Floating point exception
```

Indeed, our sample program has a division-by-zero error. In this lab, you will complete the `Instrument` pass to catch this error at runtime, as well as report code coverage of the test run. In particular, your output on the above test program should be:

```
Divide-by-zero detected at line 4 and col 13
```

and code coverage information will be printed out in a file named `EXE.cov` where `EXE` is the name of the executable that is run (in the above case, look for `simple0.cov`). Our auxiliary functions will handle the creation of the file; your instrumented code should populate it with `line,col` information. If implemented correctly, you will see the following lines in `simple0.cov` that indicate the executed lines from the program:

```
2,7
2,7
3,7
3,11
3,7
4,7
4,11
4,15
```

You will see some duplicates in `EXE.cov`. The reason is that one line in the C source code maps to more than one line in the LLVM IR.

Lab Instructions

In this lab, you will build a dynamic analyzer to catch division-by-zero errors at runtime. A key aspect of dynamic analysis involves inspecting a running program for information about its state and behavior. We will develop an LLVM pass to insert runtime checking and monitoring code into a given program. Our instrumentation will perform division-by-zero error checking and record coverage information for a running program. In fuzzing lab, we will build upon this lab to develop an automated testing framework.

Instrumentation Primer. Consider the following code snippet where we have two potential divide-by-zero errors, one at Line A, the other at Line B.

```
int main() {
  int x1 = input();
  int y = 13 / x1;   // Line A
  int x2 = input();
  int z = 21 / x2;   // Line B
  return 0;
}
```

If we wanted to program a bit more defensively, we would manually insert checks before these divisions, and print out an error if the divisor is 0:

```
int main() {
  int x1 = input();
  if (x1 == 0) { printf("Detected divide-by-zero error!"); exit(1); }
  int y = 13 / x1;
  int x2 = input();
  if (x2 == 0) { printf("Detected divide-by-zero error!"); exit(1); }
  int z = 21 / x2;
  return 0;
}
```

Of course, there is nothing stopping us from encapsulating this repeated check into some function, call it `__sanitize__`, for reuse.

```
void __sanitize__(int divisor) {
  if (divisor == 0) {
```

```
    printf("Detected divide-by-zero error!");
    exit(1);
  }
}
int main() {
  int x1 = input();
  __sanitize__(x1);
  int y = 13 / x1;
  int x2 = input();
  __sanitize__(x2);
  int z = 21 / x2;
  return 0;
}
```

We have transformed our unsafe version of the program in the first example to a safe one by instrumenting all division instructions with some code that performs a divisor check. In this lab, you will automate this process at the LLVM IR level using an LLVM pass.

Code Coverage Primer. Code coverage is a measure of the fraction of a program's code that is executed in a particular run. In this lab, you will implement the mechanism underlying modern code coverage tools, such as the LLVM's source-based code coverage tool [2] and gcov [3]. It instruments the program's LLVM IR instructions at compile-time to record the line and column number of the program's source-level instructions that are executed at run-time. This seemingly primitive information enables powerful software analysis use-cases. We will explore two such use-cases. In this part, you will use the information to improve your test suite by adding tests that cover more code and thereby uncover crashing bugs. In fuzzing lab, you will use the same information to guide an automated test input generator, thereby realizing the architecture of modern industrial-strength fuzzers.

Debug Location Primer. When you compile a C program with the `-g` option, LLVM will include debug information for LLVM IR instructions. Using the aforementioned instrumentation techniques, your LLVM pass can gather this debug information for an `Instruction`, and forward it to `__sanitize__` to report the location at which a divide-by-zero error occurs. We will discuss the specifics of this interface in the following sections.

Instrumentation Pass. We have provided a framework from which you can build your LLVM instrumentation pass. You will need to edit the `DivZeroInstrument/src/Instrument.cpp` file to implement your divide-by-zero sanitizer, as well as the code coverage mechanism. File `DivZeroInstrument/lib/runtime.c` contains functions that you will use in your lab:

```
void __sanitize__(int divisor, int line, int col)
```

Here, Output an error for `line,col` if divisor is 0.

```
void __coverage__(int line, int col)
```

Append coverage information for `line,col` in a file for the current executing process.

As you will create a runtime sanitizer, your pass should instrument the code with calls to these functions. In particular, you will modify the `runOnFunction` method in `Instrument.cpp` to perform this instrumentation for all LLVM instructions encountered inside a function.

Note that our `runOnFunction` method returns true. Since we are instrumenting the input code with additional functionality, we return true to indicate that the pass modifies, or transforms the source code it traverses over.

In short, the lab consists of the following tasks:

[2] `https://clang.llvm.org/docs/SourceBasedCodeCoverage.html`
[3] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

1. Implement the `instrumentSanitize` function to insert a `__sanitize__` check for a supplied Instruction.
2. Modify `runOnFunction` to instrument all division instructions with the sanitizer for a given block of code.
3. Implement the `instrumentCoverage` function to insert `__coverage__` checks for all debug locations.
4. Modify `runOnFunction` to instrument all instructions with the coverage check.

Inserting Instructions into LLVM code. By now you are familiar with the `BasicBlock` and `Instruction` classes and working with LLVM instructions in general. For this lab you will need to use the LLVM API to insert additional instructions into the code when traversing a `BasicBlock`. There are manys ways to do this in LLVM. One common pattern when working with LLVM is to create a new instruction and insert it directly after some previous instruction.

For example, in the following code snippet:

```
Instruction* Pi = ...;
auto *NewInst = new Instruction(..., Pi);
```

A new instruction (`NewInst`) will get created and implicitly inserted after `Pi`; you do not need to do anything further with `NewInst`. Subclasses of `Instruction` have similar methods for doing this. In particular, you will only need to create and insert new call instructions (`CallInst`), as discussed below.

Loading C functions into LLVM code. We have provided the definitions of the auxiliary functions `__sanitize__` and `__coverage__` for you, but you have to insert calls to them into the code as LLVM instructions. Keep in mind that both of these functions are only used for logging purposes. `__sanitize__` logs all the occurences of a divisor being equal to zero, and `__coverage__` logs any executed line of the code.

Before a function can be called within a Module, it has to be loaded into the Module using the appropriate API `Module::getOrInsertFunction` [4]. On way to do this is illustrated below:

```
Value* NewValue = M->getOrInsertFunction("function_name", return_type,
                          arg1_type, arg2_type, ..., argN_type);
Function* NewFunction = cast<Function>(NewValue);
```

Next, the function that you have created must be called. So you will have to create a call instruction at instruction `I` using `CallInst::Create` [5] as illustrated below:

```
CallInst *Call = CallInst::Create(NewFunction, Args, "", &I);
 Call->setCallingConv(CallingConv::C);
 Call->setTailCall(true);
```

You should populate `std::vector<Value *> Args` with appropriate values for arguments.

Debug Locations. As we alluded to in the primer, LLVM will store code location information of the original C program for LLVM instructions when compiled with `-g`. This is done through the `DebugLoc` [6] class:

```
Instruction* I1 = ...;
DebugLoc &Debug = I1->getDebugLoc();
printf("Line No: %d\n", Debug.getLine());
```

[4] https://llvm.org/doxygen/classllvm_1_1Module.html
[5] https://llvm.org/doxygen/classllvm_1_1CallInst.html#a850d8262cd900958b3153c4aa080b2bb
[6] https://llvm.org/doxygen/classllvm_1_1DebugLoc.html

You will need to gather and forward this information to the sanitizer functions. As a final hint, not every *single* LLVM instruction corresponds to a specific line in its source C code. You will have to check which instructions have debug information. Use this to help build the code coverage metric instrumentation.

Submission

Submit only your modified file `Instrument.cpp`.