Assignment 1– Static Analysis and Instrumentation with LLVM

HSS
Fall 2021

The goal of this assignment is to get your hands dirty using LLVM and gain experience developing static analysis techniques that can work on real-world code.
Logistics:

- **LLVM Primer:** Please make sure that you have skimmed the LLVM Primer presentation (access it from course webpage) to know the capabilities of LLVM.
- **Setup Repo:** I have created a `github` repo with all the necessary scripts to install LLVM and starter code to write a pass. You can access it at: `https://github.com/purs3lab/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.

1 Setup

Instructions to setup your development environment for writing LLVM passes.

1. Follow the instructions in the *Setup Repo* and install LLVM 12.
2. Install `CLion`.
3. Open the `llvm_passes` folder in `CLion`.
4. *Build* → *Build Project*. This should create `cmake-build-project`, which will contain the `.so` of each of the passes. For instance, the folder `cmake-build-project/StaticAnalysisPasses/GetLoopExitingBBs` contains `libGetLoopExitingBBs.so`.
5. Alternatively, you can build the passes on command line using `cmake`, instructions are in the repo.

2 Generating Bitcode file

You can generate a bitcode file for a given C file by following the below instructions:

```
clang -c -emit-llvm <your_c_file> -o <path_to_output_bitcode>
```

Example:

```
clang -c -emit-llvm simple_log.c -o simple_log.bc
```

The file `simple_log.bc` will be in binary format. You can get human readable bitcode file from the binary format using the following command:

```
llvm-dis <path_to_bitcode_file>
```

Example:

```
llvm-dis simple_log.bc
```

The above command will generate `simple_log.ll` which is a text file containing human readable LLVM IR.

> You can generate bitcode for your entire project (i.e., multiple C files) using `wllvm` [1].

3 Running LLVM passes

There are many ways to run an LLVM passes. We will use one of the most common ways i.e., `opt`, short form for optimizer. `opt` is a program that is available in your LLVM installation and you can use it to run individual LLVM passes.

Given a bitcode file and the names of the passes to run, `opt` runs each of the provided passes and if any of the requested pass is a transformation pass, it outputs the modified bitcode at the given path.

3.1 Analysis Pass

As the name suggests, an Analysis pass just *analyzes* the given bitcode file, i.e., it computes certain program properties on the given bitcode file. You can run an analysis pass as shown below:

```
opt -load <path_to_so_file_of_pass> -<pass_name> <path_to_bc_file>
```

Example:

```
opt -load ./libFunctionIdentifier.so -identfunc <input_bc_file>
```

3.2 Transformation Pass

A Transformation pass modifies the given bitcode file by adding/removing instructions/data/functions, etc. While running a **transformation** pass, you should provide the path to the output bitcode using `-o` option, where the modified bitcode file will be saved.

```
opt -load <path_to_so_file_of_pass> -<pass_name> <path_to_bc_file> -o <path_to_output_bc_fle>
```

Example:

```
opt -load ./libLogMemAccess.so -logm <input_bc_file> -o <path_to_output_bc_file>
```

4 Warm up

Before delving into the assignment, please play with passes in the repository and make sure you understand what each pass is trying to do and how it is doing it.

5 Debugging

Most often `print`, i.e., `outs<<` is your friend. However, you can use `CLion` to do interactive debugging by following the instructions as posted on our slack.

Part 1 - Invalid Pointer Cast Detector (70 points)

Invalid pointer casts [2] occur when we cast a pointer of one type to another incompatible type. Different alignments are possible for different types of objects. If a pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed. Consider the following example:

```
char c = 'x';
int *ip = (int *)&c; /* This can lose information */
char *cp = (char *)ip;

/* Will fail on some conforming implementations */
assert(cp == &c);
```

Here, we expect `cp == &c` to always hold, but it may not hold depending on the alignment of pointers.

Invalid casts primarily result in undefined behavior, however, in few cases they can result in security vulnerabilities. Consider the following example:

```
1 void foo(..) {
2    char *q = malloc(n);
3    ...
4    read_int(q);
5 }
6 void read_int(void *ptr) {
7    int *d = (int *)ptr;
8    *d = atoi(...);
9 }
```

Here, the pointer `q` pointing to heap memory is casted to `int *` at line 8 through the function call to `read_int` at line 4. If the value of `n` is less than `sizeof(int)`, then we will be accessing unallocated memory, which can lead to a security vulnerability.

Invalid pointer casts in OS kernels, such as Linux kernel, are rampant mainly because of the complex interactions between various OS components. These bugs can have high security impact depending on the trigerrability from user space.

In this first part, you will develop a static analysis (i.e., analysis pass) technique to detect invalid pointer casts. There are two components of a invalid pointer casts detector: (1) Detecting pointer casts, (2) Figuring out which of these pointer casts are invalid.

You will focus on only the first component, i.e., *detecting pointer casts*. Here, you goal is to *detect casts for all pointers* that can be either local variables, global variables, or function parameters. Consider the following code:

```
1  void foo(..) {
2     char *q = malloc(n);
3     ...
4     read_int(q);
5  }
6  void bar(..) {
7    short b;
8    ...
9    baz((char *)&b);
10   ...
11 }
12 void baz(char *p) {
13   ...
14   read_int(p);
15   ...
```

```
16 }
17 void read_int(void *ptr) {
18    int *d = (int *)ptr, int *x;
19    *d = atoi(...);
20    ...
21 }
```

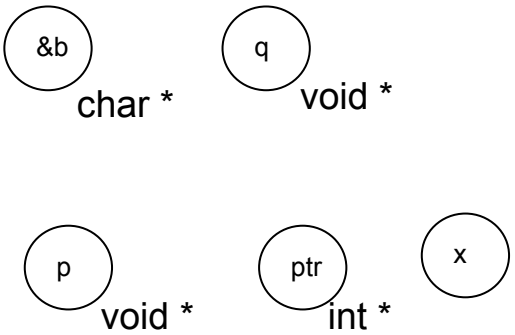
The expected output for the above code (i.e., bitcode file corresponding to above code) should be:

```
Invalid casts for local variable q (char *) of function foo at:
-Line 4 of main.c to [void *]
-Line 18 of main.c to [void *, int *]

Invalid casts for local variable b (short) of function bar at:
-Line 9 of main.c to [char *]
-Line 14 of main.c to [char *, void *]
-Line 18 of main.c to [char *, void *, int *]

Invalid casts for parameter p (char *) of function baz at:
-Line 14 of main.c to [void *]
-Line 18 of main.c to [void *, int *]

Invalid casts for parameter ptr (void *) of function read_int at:
-Line 18 of main.c to [int *]
```

If a pointer has no casts (e.g., `x` in `read_int`), then you should not output anything related to that pointer.

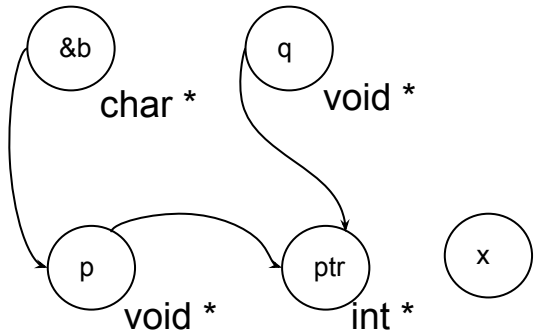
The high-level approach for this problem is to first find all pointer casts with in each of the function and then compute inter-procedural (i.e., information across functions) casts.

In the first phase, you will look into each function and record invalid casts that are visible within the function. One way to do this would be to maintain nodes for each pointer and keep track of casts for each pointer as shown below:



Implementation: You can start with a `cast` or `bitcast` instruction that is casting to a pointer type. Start backtracking from this cast instruction using `users()` function to find all the variables that reach the cast instruction. You should also backtrack `load` instructions to `store` instructions through pointer operand. Specifically, when you are back tracking a `load` instruction, you should find all `store` instructions to the same pointer that can reach the `load` instruction and then continue backtracking from the value operand in each of these `store` instructions.

In the second phase, you will connect function calls through parameters as shown below:



Given the above graph, you can start from a node and display the information at each node reachable from the given node. For instance, for `b`, you start with `[char *]` (information present at the node), then you go to `p` (i.e., by traversing the edge) and display `[char *, void *]` and finally you reach `ptr` and display `[char *, void *, int *]`. You can use LLVM's `DebugInfo` to know the source code line and file information.

Since this is a simple analysis, you can create a custom node and graph structure for your analysis.

Usability Restriction:

We want to have a uniform interface to use your pass. Configure your `CMakeLists.txt` appropriately so that it produces `libInvalidCastDetector.so`.

I expect to run the pass using the following command, so please make sure that you configure your pass to run with `detectcasts` as shown below:

```
opt ./libInvalidCastDetector.so -detectcasts <input_bc_file>
....Your output...
```

Part 2 - Lightweight Control Flow Integrity (30 points)

In this part, we will develop a lightweight Control Flow Integrity [3] (CFI) mechanism. In short, CFI is a way to make sure that control flow of a program always follows expected order.

In this part, we will write an instrumentation pass, i.e., a pass that modifies the given bitcode file. Our goal here is to ensure that a function call through a function pointer will always jump to one of the expected functions. Consider the following example:

```
1  void foo(int a) {
2   ....
3  }
4  void bar() {
5   ....
6  }
7  void klm(int b) {
8   ...
9  }
10 int baz(float *a, int b) {
11  ....
12 }
13 int main(int argc, char **argv) {
14 ...
15 void (*fun_ptr)(int);
16 char buf[20];
17
18 //buffer overflow bug.
```

```
19 strcpy(buf, argv[0]);
20 // Calling.
21 (*fun_ptr)(10);
22 ...
23 }
```

Here, the buffer overflow bug at Line 19 can be exploited to modify the value of `fun_ptr`. So that later when the function pointer is used to call at Line 21, we can get arbitrary code execution. CFI will limit the exploitability of such control flow hijacking bugs.

To do this, CFI will check that when a function is called through a function pointer, the target is one the expected functions. First, for each function pointer, we need to find all the possible functions invoked through the function pointer.

In the above example, *What are the possible functions invoked at Line 21 through function pointer `fun_ptr`?*

Assuming well-written code, all functions that accepts a single `int` argument and returns `void` are the possible targets. Specifically, functions `foo` and `klm` are the possible targets of the function pointer `fun_ptr`.

Second, using the function targets information, the above code can be instrumented as follows:

```
1 + if (!(func_ptr == foo || func_ptr == klm)) {
2 +    printf("Invalid Control flow detected\n");
3 +    exit(-1);
4 + }
5 // Calling.
6 (*fun_ptr)(10);
7 ...
8 }
```

Now, if the buffer overflow overwrites the function pointer to arbitrary value, our instrumentation will prevent this.

Implementation: You can implement this pass in a smart way. First, define a C program (say `check_target`) that accepts a pointer (function pointer) and list of pointer values (targets for the corresponding function pointer). Second, instrument the given bitcode file such that before each function call through function pointer, insert a call to `check_target` function. For a concrete example, look into `InstrumentationPasses` of the *Setup Repo*.

Usability Restriction:

> Configure your `CMakeLists.txt` appropriately so that it produces `libLightweightCFI.so`.
>
> I expect to run the pass using the following command, so please make sure that you configure your pass to run with `lcfi` as shown below:
>
> ```
> opt ./libLightweightCFI.so -lcfi <input_bc_file> -o <output_bc_file>
> ....Your output...
> ```

6 Submission

Create a `tar.gz` with code and `CMakeLists.txt`, such that we should be able to extract it and run `cmake` to build the passes. For more details look into `llvm-passes` folder of the *Setup Repo*. Extracted folder should have the following structure:

```
extracted_folder:
 -InvalidCastDetector
 -LightweightCFI
 -CMakeLists.txt
```

We should be able to build passes by following the below instructions:

```
cd extracted_folder
mkdir build
cd build
cmake ..
```

Please make sure that passes confirms to the corresponding *Usability Restriction*. Submit the `tar.gz` to brightspace.

References

[1] `https://github.com/travitch/whole-program-llvm`.

[2] `https://wiki.sei.cmu.edu/confluence/display/c/EXP36-C.+Do+not+cast+pointers+into+more+strictly+aligned+pointer+types`.

[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009), 1–40.