Assignment 3– CodeQL Playground and Real-world Bug Finding

HSS
Fall 2025

There are two modules in this assignment. In the first module, you will work on writing CodeQL queries to search of certain potentially buggy patterns. In the second module, you will experience real-world bug finding by running CodeQL suites on open-source repositories.

1 Setup

Instructions to setup your development environment for writing CodeQL queries.

1. Setup `codeql cli` by following instructions at: `https://codeql.github.com/docs/codeql-cli/getting-started-with-the-codeql-cli/`.
2. Install VSCode and CodeQL extension by following instructions at: `https://codeql.github.com/docs/codeql-for-visual-studio-code/setting-up-codeql-in-visual-studio-code/#installing-the-extension`
3. Create a new workspace and add standard libraries from github repo into the new workspace by following instructions at: `https://codeql.github.com/docs/codeql-for-visual-studio-code/setting-up-codeql-in-visual-studio-code/#updating-an-existing-workspace-for-codeql`.

2 Module 1: Writing CodeQL queries

You are expected to write CodeQL queries for the following issues. In each of the following queries you should select the **File name**, **Line number**, and **Column Number** where the corresponding issues is present.

Part 1 - snprintf (10 points)

Although, `snprintf` prevents certain security issues. It should be used carefully to avoid certain security issues [1]. You goal is to write a CodeQL query that will find the following pattern.

```
var += snprintf(var, ..., ...);
```

Part 2 - sprintf, sscanf, and, fscanf (15 points)

Using %s in `sprintf`, `sscanf` or `fscanf` is always dangerous and most likely a sign of badly written code.

You goal is to write a CodeQL query that will find the following patterns.

```
sprintf(,..%s..., ...);
sscanf(,..%s..., ...);
fscanf(,..%s..., ...);
```

Specifically, the format string should contain %s.

Part 3 - Mishandled return value of realloc (25)

The allocator function `realloc` can return `NULL` when there is no enough space. So, if you do `p = realloc(p,...);` and there is no enough space you will end up overwriting `p` with `NULL` and loose the pointer to the previous data. So, you should always do `y = realloc(x,...); if (y != NULL) x = y;`.

Your goal is to find these risky `realloc` patterns. Blindly trying to find above patterns might result in false positives.

Instead, lets try to find the following patterns, we may miss some bugs but most of the patterns we find will be true issues.

```
x->f = realloc(x->f,....);
// or
y.f = realloc(y.f,....);
```

Note that, here, we want to make sure that the variable is a structure access.

Part 4 - Tainted multiplication used as in allocator (20)

One of the common problems with integer overflow is to use the overflowed values as a size argument. Consider the following example:

```
// Integer overflow of x*y the size of buffer
// pointed by p can be less than x.
p = malloc(x*y);
...
// Here there is a possibility of heap buffer overflow.
// This is because the size of memory pointed by p could be less
// than x.
memcpy(p,...,x);
```

You goal is to write a CodeQL query to find calls to memory allocators where the size is computed through multiplication of **tainted** values.

Part 5 - Double free (40)

Here, you will write CodeQL query to find potential double free patterns. Specifically, you need to find two calls to `free` with the *same* pointer argument, which satisfies the following two conditions:

- The first call to `free` reaches second `free`. For instance, following is not a double free bug:

  ```
  if(...) {
     free(p);
  } else {
     free(p);
  }
  ```

  Because, as you can see, although, there are two calls to `free`, the program execution that reaches the first `free` can *never* reach second `free`.

- There is *no* assignment to the pointer argument. For instance, following is not a double free bug:

  ```
  free(p->f);
  ...
  p->f = ....;
  ...
  free(p->f);
  ```

  Because, there is new assignment to the pointer and we are not `free`ing the same object.

2.1 Testing your queries on real-world code

Download the following sources and build CodeQL database for these programs and execute your queries on each of the databases:

- **gettext**:`https://www.gnu.org/software/gettext/`
- **cflow**:`https://www.gnu.org/software/cflow/`
- **autogen**:`https://www.gnu.org/software/autogen/`
- **libextractor**:`https://www.gnu.org/software/libextractor/`
- **a2ps**:`https://www.gnu.org/software/a2ps/`

Save the result of running the query into a `csv` file in CodeQL. Save the file somewhere because you will be submitting the file as part of your submission (Section 4).

3 Module 2: Real-world Bug Finding (100)

In this module, your goal is to run an existing CodeQL suite (i.e., `cpp-security-extended`) on **five** (20 points per project) Debian packages, analyze the results and report your findings to the corresponding package owners.

3.1 Project Selection

1. **You should pick five projects (one from each category)** from here: `https://docs.google.com/spreadsheets/d/10jUfcpiApnrf0EAkSYB0ykM_mhog_pjvBWUmYB0aN28/edit?usp=sharing`. The rows in green are available.
2. **Only one person can pick a project.**
3. Please add your name to the sheet after you make your selection.
4. It is important to select early, or else you will be left with what others have not picked.

3.2 Running CodeQL

You need to run the CodeQL suite and save the results into a `sarif` file. The CodeQL suite is available here: `https://github.com/github/codeql/blob/main/cpp/ql/src/codeql-suites/cpp-security-extended.qls`. You need to clone the CodeQL repository (i.e., `git@github.com:github/codeql.git`) and then use the local path while running the suite.

```
# Creating CodeQL Database
$ cd <extracted_folder>
$ codeql database create <new_folder> --overwrite --language=cpp
# Running CodeQL
$ codeql database analyze <db_folder> --format=sarif-latest --output=codeqlresults.sarif
↪  <local_path_to_cpp_security_extended.qls>
```

3.3 Results Analysis

For each analyzed project you need to record the following:

- Total errors reported by each query.
- Number of true and false positives by each query.
- Number of true positives acknowledged by project owners.

Look into the sheet `https://docs.google.com/spreadsheets/d/1uelxS3HwTjctc5NwOPGaOijMEVuRaKZN3mcx` `edit?usp=sharing` (note two sheets) for the things that you to submit for each project. You need to save two sheets as `.csv` files which you should submit.

4 Submission

Create a `tar.gz` with all the queries and results of running them on each of the programs in Section 2.1. The extracted folder should have the following structure:

```
extracted_folder:
 -Module 1
   -part1
     -query.ql <- Your CodeQL query
     -gettext.csv <- resulting of running the query on gettext.
     -cflow.csv <- Same as the above.
     -autogen.csv <- Same as the above.
     -libextractor.csv <- Same as the above.
     -a2ps.csv <- Same as the above.
   -part2
     Same as above
    ...
   -part5
 -Module 2
   -project1
     -codeqlresults.sarif
     -summary.csv
     -truepositives.csv
   -project2
     -codeqlresults.sarif
     -summary.csv
     -truepositives.csv
   -project3
     ...
   -project5
    Same as above
```

Submit the `tar.gz` to brightspace.

References

[1] https://access.redhat.com/blogs/766093/posts/1976193.