

Assignment 2b

Due: Friday, October 22, 2010, 11:59PM EST

Assignment 2 has been divided into 2 parts, 2a and 2b. Part A will be worth 50% of the grade for the assignment; part B will be worth 50%. These instructions are for part B, using static analysis for security.

Submission will be via the Blackboard dropbox and must be received by **11:59 PM EST on Friday, October 22, 2010**. You may resubmit as many times as you like before the deadline; only the last submission will be graded. Please read and follow the instructions on the following pages carefully to ensure you receive maximum credit for your submission.

1 Assignment overview

In this assignment, you will use the Coverity tool to explore static analysis of open-source software. You will be provided a VM with the following software pre-installed:

- The Coverity Static Analysis tool (SA), for checking source code.
- The Coverity SDK, for writing and compiling new checker modules.
- OptiPNG, a PNG optimizer that recompresses image files to a smaller size.
- QKC, an ultra highspeed Kanji code converter.
- AALib, an ASCII art graphics library.
- Overdose, a Yahoo! chat client.

1.1 Using the VM

This is just like what we did for WebSecLab and HW2A.

Unzip the HW2b_VM.tgz file. Using VirtualBox, select File→Import Appliance. Navigate to HW2b.ovf and select it to import. Take all the default options.

When you run the VM, you can log in with username `user` and password `user`.

SA is found in `~/cov-sa-linux-5.2.1`. The SDK is found in `~/cov-sdk-linux-5.2.1`. OptiPNG is found in `~/optipng-0.6`. QKC is found in `~/qkcc`. AALib is found in `~/aalib-1.4.0`. Overdose is found in `~/overdose-0.1.2`.

2 Running Coverity

Running static analysis. Please note that helpful directions and references can be found in `~/cov-sa-linux-5.2.1/doc/`. In particular, “Getting started: C” (`getting_started_c.html`) provides a quick overview of running SA, and “Checker reference” (`checker_ref.html`) provides information about the various errors that SA catches and what they mean.

In general, using the tool will work as follows:

- Navigate to the directory of the software you want to check.

- Use `make clean` to remove any existing object code.
- Build the intermediate representation of the target software you want to check with `cov-build --dir analysis-dir <build cmd>`. (See Note 1 for details.)
- Do the analysis with `cov-analyze --all --dir analysis-dir`. (Using `--all` turns on all checkers, including the security checkers that are disabled by default.)
- The defect summary will be output to the screen. You can look at individual defects by examining the related XML files, found at `./analysis-dir/c/output/BUG_NAME.errors.xml`. There will be one of these files for every type of bug listed in the defect summary. (The defect summary is also archived at `./analysis-dir/c/output/summary.txt`.)

Note 1: For most of the software we are checking, the build command is simply `make` from the top-level directory of that software (for example, go to `~/qkcc` and type `make clean` to clean up and `make` to build. However, for OptiPNG you need to navigate to `~/optipng-0.6/src` and use `make -f scripts/gcc.mak`. You also use that as the input to `cov-build`. Use `make -f scripts/gcc.mak clean` to clean.

Note 2: Once you run SA, subsequent runs on the same application will not produce a new list of errors because all the previously found errors are still catalogued in `./analysis-dir/c/output/`. You can review the summary with `summary.txt` from that directory, or simply delete the entire analysis directory and run it again to get a fresh start.

Building your own checkers. For building your own checkers, there are a lot of helpful resources. First, `~/cov-sdk-linux-5.2.1/extend/doc/` contains the “Extend User Guide” (`extendug.html`), which explains how to design and build new checkers. The “Extend Reference” (`extend_ref.html`) contains a syntax reference for the state-machine language and commands you can use to build checkers.

Second, in `~/cov-sdk-linux-5.2.1/extend/samples/` there are many sample checkers demonstrating how the various features can be used. The User Guide explains many of these samples in detail, and also contains directions for how to build them.

It may also be helpful or necessary to review the C++ header files that define the SDK functionality, which can be found in `~/cov-sdk-linux-5.2.1/extend/headers/`. The `patterns` and `types` headers may be particularly helpful.

To experiment with the sample checkers, do the following. (This example assumes testing the `hello` sample; it works similarly for the others.)

```
% cd ~/cov-sdk-linux-5.2.1/extend/samples/
% make #This will build all the sample checkers at once.
% cd hello/test1
% cov-translate --dir analysis-dir gcc -c hello.test.c
% ../hello --dir analysis-dir --force --prevent-root $PREVENT_ROOT
```

In this case, `cov-translate` acts like `cov-build` from before and generates the intermediate representation of the target software we want to test. Then the new checker is run using the executable generated by the `make` command that builds all the sample checkers. (For convenience, we run new checkers directly, as standalone modules, rather than incorporating them into SA.)

To build your own checker, you can add a new subdirectory under `samples` and take advantage of the existing Makefile structure. Or, you can create a new directory and test it as follows (from any directory):

```
% mkdir mychecker
```

```

% cd mychecker
% touch mychecker.cpp      # write checker source code in this file
% build-checker mychecker  # don't include the file extension
% touch test.c             # write test source code in this file
% cov-translate --dir ad gcc -c test.c
% ./mychecker --dir analysis-dir --force --prevent-root $PREVENT_ROOT

```

And finally, to run your checker against real software (using qkcc as an example):

```

% cd ~/qkcc
% make clean
% cov-build --dir analysis-dir make
% ~/mychecker/mychecker --dir analysis-dir --force --prevent-root $PREVENT_ROOT

```

(Note that you can replace `~/mychecker/mychecker` with the actual path to your built checker executable.)

3 The assignment

The goals of the assignment are to use Coverity to identify bugs in real open-source software, and then to write your own Coverity checker modules to identify bugs.

1. **Analyze the provided open-source software.** (15 points) Run SA against all four provided open-source tools. Survey the bugs discovered during your analysis. Identify five bugs that you consider to be false positives, and then identify five bugs that you consider to be serious potential risks. Explain why the false positives are not actually bugs, as well as why the tool flagged them anyway. Explain why the serious bugs are real risks. Note: Among the 10 total bugs you discuss, you must include at least one from each of the four tools. You cannot use any category of bug more than twice in the 10 total. (Don't forget to run SA with `--all` to turn on security checking.) Your writeup accounts for all the points here.
2. **Write a “simple” checker.** (10 points) Use the SDK to write a “simple” (not stateful) checker that duplicates the functionality of the SECURE_CODING checker. (See the checker reference for details.) Your checker should find, at minimum, all the same defects that the SA SECURE_CODING checker finds. Include an explanation of how your checker works and what it does or does not report in your writeup. (8 points for successful defect finding, 2 points for writeup.)
3. **Write a checker for server2.** (25 points) In `~/server2/` you will find the code for server2 from HW1A. Run SA against it, and you will find that SA does not catch the off-by-one error. Write a new checker that can catch this defect. To get full credit, your checker should be generally applicable – that is, it should be capable of catching similar bugs in other software, not just the specific bug in server2. It also should alert only when there is a reasonable risk of a defect; just warning about the presence of a dangerous function without reasoning about how that function is used is insufficient. Describe how your checker works and what it does or does not report in your writeup. If there is anything you think you could do to improve your checker, describe that too. (20 points for defect finding, 5 points for writeup.)

4 Submission and grading

Turn in a gzip tarball containing exactly 3 files (replace XX with your group number):

- `secure_coding_gXX.cpp` – your checker for item 2 above.
- `s2_overflow_gXX.cpp` – your checker for item 3 above.
- `hw2B_gXX.pdf` – your writeup for all parts of HW2B.

The gzip tarball should be named **hw2b.gXX.tgz**. Put all your files in the directory `gXX`, then can create this tarball as follows:

```
tar -czvf hw2b_gXX.tgz gXX/
```

When the tarball is unzipped, it should result in a folder named `gXX` containing all your files.

Parts of the assignment will be tested using an automated grading script. Therefore, you **must conform to the naming conventions** if you wish to receive credit. The grading script will use the following commands to check each of your checkers:

```
% build-checker <checker name> # don't include the file extension
% ./<checker name> --dir analysis-dir --force --prevent-root $PREVENT_ROOT
```

(Where some source code to run the checker against has previously been built into intermediate format in `analysis-dir`).

Note that the name of your checker must match the name of the checker file (minus the file extension), so your checkers will be named `secure_coding_gXX` and `s2_overflow_gXX` respectively.

Late submissions will incur a penalty of 20% per day.