The University of Melbourne

SWEN90006: Software Security & Testing

Assignment 3 (Group Assignment)

Second Semester, 2020

Due Date: 18:00pm, Friday, 23 October, 2020

1 Introduction

This is a group-based assignment, to be completed in groups of 4 (four) of your own choosing. It is worth 25% of your grade for the subject.

In this group-based assignment, you will gain experience with security testing applied to a small but non-trivial program, including investigating the trade-offs between different fuzzer designs and contemporary off-the-shelf greybox fuzzers. You will also learn about memory-error vulnerabilities in C, and use LLVM's memory sanitisers.

You will implement a custom fuzzer to find memory-error vulnerabilities in a C program and insert your *own* vulnerabilities. Your fuzzer will then "play" competitively against the fuzzers of other teams to determine whose can find the most vulnerabilities. You will also compare your fuzzer against LLVM's general-purpose greybox fuzzer, libFuzzer, and evaluate it using standard coverage metrics, as well as other metrics that you should decide upon. You will produce a report detailing your vulnerabilities, your fuzzer's design and systematically evaluating its performance and the implications of your design decisions.

Marks are intentionally skewed towards the report (see the marking critera in Section 7), so make sure you apportion your efforts appropriately in accordance with the marking criteria.

The program under test will be a command-line *desktop calculator tool* implemented in C, and inspired by the UNIX `dc` utility.

2 Things To Do Early

To make sure you don't have last minute problems, **make sure you do each of the following very early on (i.e. once you have assembled your team.)**

1. Decide as a group on the responsibilities of each of your team members: who will do what, how you will divide up the tasks between team members, and how you will deal with non-performance. Complete the *Group Agreement* on the assignment page of the LMS and email a copy to Toby. See Section 5.1 for further details.

2. Fork the assignment repo and add team members (see instructions below) plus the admin user. **MAKE SURE YOUR FORKED REPO IS PRIVATE!**

3. Make sure everybody in your group can clone the forked repo and push to it

4. Make sure that the automated eligibility checks, run by gitlab when you push to your repository, are passing.
5. Make sure everybody in your group can build the code and run the various targets (see instructions below).
6. Ask questions on the discussion board if you are having difficulty with these tasks. **Problems with getting the code building or with git/gitlab will not be grounds for extensions. Make sure you work out how to use the tools early on.**

3 Gitlab and Assignment Repository

As with Assignment 1, your team will carry out their work in a git repository hosted on the `gitlab.eng.unimelb.edu.au` GitLab service.

3.1 Assignment Repository

The assignment repository is at:

`https://gitlab.eng.unimelb.edu.au/tobiasm1/swen90006-a2-2020`.

The top-level file `README.md` documents its structure.

3.2 Working as a Team

The user listed in the "Gitlab username1" column of the groups spreadsheet will fork the assignment repository, on behalf of your group. They should then give permission to the other group members to push to the repository; although it is up to each team to manage how updates should be made to their repository. As with Assignment 1, **your repositories MUST be private and you must add the admin user to your project.**

Having forked the repository, the "Gitlab username1" user must then add each of their team members as *Members* of their repository, plus the admin user (as in Assignment 1). From the repository's page in gitlab, choose *Settings* then *Members*, and then invite the other team members to the repository by specifying their gitlab username and giving them the the appropriate permission (e.g. the *Maintainer* permission). They can then `clone` your repository and `push` to it just as you would, allowing your team to work collaboratively.

Also, **remember to add the 'Administrator (@root)' user as in Assignment 1, giving them the *Developer* permission**.

4 The DC C Implementation

Since the focus of this assignment is security testing for *memory-error vulnerabilities*, you will be using a C implementation of a program similar to the UNIX `dc` desktop calculator utility.

Specifically, the program is a command-line utility for performing arithmetic computations. It takes input either from text files or from the terminal (i.e. standard input, aka stdin).

The calculator has an operand stack plus a store of local variables. Commands manipulate the stack, perform arithmetic operations, and modify variables.

4.1 Commands

Each line of input is a *command*. Commands conform to the following grammar:

<COMMAND> ::= “+”
“-”
“*”
“”
“push” <NAME>
“pop”
“load” <NAME>
“store” <NAME>
“remove” <NAME>
“save” <NAME>
“list”
“print”

Tokens in the grammar above are separated by one or more *whitespace* characters, namely space, tab, carriage-return and line-feed. Each <NAME> is a string of *non-whitespace* characters.

- For a string *num* representing a decimal integer, e.g. “5”, the command “push *num*” pushes the value represented by *num* onto operand stack.
- The command “pop” pops the value from the top of the operand stack, discarding it.
- The commands “+”, “-”, “*” and “” each pop the top two values from the operand stack and compute the corresponding arithmetic operation on them (addition, subtraction, multiplication and division, respectively), and push the result onto the stack.
- For a string *var*, the command “load *var*” loads the value stored in variable *var* and pushes it onto the stack.
- For a string *var*, the command “store *var*” pops the value from the top of the stack and stores it into variable *var*, defining that variable if it is not already defined.
- The command “list” prints out all currently defined variables and their corresponding values.
- For a string *var*, the command “remove *var*” makes variable *var* undefined (i.e. it will not be printed by subsequent “list” commands).
- For a filename string *filename* the command “save *filename*” saves the value of each currently defined variable to the file *filename*.
- The command “print” prints out the contents of the operand stack.

4.2 Running the Application

Note, once you have built the program (see Section 4.3), the binary lives in the `bin/original/` directory, from which the following commands should be run.

The dc program, `dc`, when run with no command line arguments prints out simple usage information.

```
$ ./bin/original/dc
Usage: ./bin/original/dc file1 file2 ...
       use - to read from standard input
```

That is, `dc` takes as its command line arguments a series of filenames or the special name "-", which instructs it to read from standard input (i.e. user keyboard input). It simply processes each in turn, reading and executing commands line-by-line.

When `dc` saves the currently defined variables to a file (i.e. in response to a "save" command), it does so by writing to the file a series of commands that, when processed, cause the variables' state to be reconstituted.

For example, we might run the dc and use it to save two variables to the file `variables.txt`. Here we define two variables "five" and "ten" whose values are, respectively, 5 and 10:

```
$ ./dc -
push 5
push 10
store ten
store five
save variables.txt
```

We can exit the `dc` program by pressing CTRL-D, which indicates end-of-file/end-of-input, or by forcefully interrupting it via CTRL-C.

The contents of the `variables.txt` file is then:

```
push 10
store ten
push 5
store five
```

We can then re-run the dc on this file and then instruct it to take further input from the user by running "`./dc variables.txt -`" on the command line. The user can then list the variables via the `list` command, which in this case would cause the two saved variables to be printed:

```
VARIABLE: five, VALUE: 5
VARIABLE: ten, VALUE: 10
```

4.3 Building

There are two ways to build the software. The first is by using the Engineering IT machines:

- `dimefox.eng.unimelb.edu.au`
- `nutmeg.eng.unimelb.edu.au`
- `digitalis.eng.unimelb.edu.au`

The second is to use your own machine. We discuss each.

4.3.1 Engineering IT Machines

1. ssh to one of the machines, log in with your username and password.
2. Then, ensure that `/home/subjects/swen90006/llvm+clang-6.0.0/bin` is in your path:

```
export PATH=$PATH:/home/subjects/swen90006/llvm+clang-6.0.0/bin
```

You should add that line to the `.profile` file in your home directory, so that this directory is always in your path.

3. Test you can run `clang-6.0`, you should see the error message:
   "`clang-6.0:  error:  no input files`"

4. Then from the top-level directory of the subject repository, run

   ```
   source eng_unimelb_setup
   ```

   which will ensure that the environment variables `LDFLAGS` and `LD_LIBRARY_PATH` are set as required for building on these machines.

4.3.2 Building on your Own Machine

If you want to build on your own machine, you'll need to install the dependencies yourself.

Requirements (aka Dependencies)

Unix-like host: Linux or MacOS The build process and software has been tested on Ubuntu 16.04 LTS and MacOS 10.13.6 High Sierra. It might well work on other somewhat-recent Unix platforms too.

We *strongly* recommend using a Virtual Machine running e.g. Ubuntu 16.04 LTS if you want to build on your own machine. Installing Clang (see below) in this case is very straightforward.

Clang and LLVM The build process has been tested with Clang+LLVM 6.0

Ubuntu (16.04 LTS)

1. First, enable the `universe` package repository if you haven't enabled it already:

   ```
   sudo add-apt-repository universe
   ```

2. Next, just install Clang and LLVM 6.0 via `apt-get`.

   ```
   sudo apt-get install libfuzzer-6.0-dev llvm-6.0
   ```

MacOS

1. If you don't have them installed already, first install Xcode Command Line Tools for your OS version (e.g. "Xcode Command Line Tools (macOS 10.13) for Xcode 9.4.1"), from `https://developer.apple.com/download/more/`. Note that we won't be using the version of LLVM that ships with Xcode Command Line Tools. However we need the system include headers and so on that come with it.
2. Download Clang+LLVM 6.0.0 from `http://releases.llvm.org/6.0.0/clang+llvm-6.0.0-x86_64-apple-darwin.tar.xz`
3. Extract the resulting file:

   ```
   tar -xJf clang+llvm-6.0.0-x86_64-apple-darwin.tar.xz
   ```

This will put Clang+LLVM into the directory `http:releases.llvm.org6.0.0clang+llvm-6.0.0-x86_64-apple-darwin.tar.xz`

4. Try running clang:

   ```
   cd clang+llvm-6.0.0-x86_64-apple-darwin
   ./bin/clang-6.0
   ```

   You should see the message: "`clang-6.0: error: no input files`"

5. Put `clang-6.0` etc. into your path:

   ```
   export PATH=$PATH:$PWD/bin
   ```

   You should put this line into the file `.profile` in your home directory, which ensures that `clang-6.0` will be in your path each time you open a terminal.

6. Test you can run it from any location:

   ```
   clang-6.0
   ```

   You should see the same message as before. Note that this time we didn't need to specify *where* `clang-6.0` was located because it is now in the path.

4.3.3 Building the C Implementation

There is a `Makefile` for building the C implementation of the program, in the top-level directory. Assuming you are building with Clang and LLVM 6.0 and have successfully installed them, then all you should need to do for the initial build is to run '`make`' from the top-level directory:

```
$ make
```

This should give you at least the main binary, `dc` in the `bin/original` directory, as well as some other targets (see Section 4.3.6 below).

If you are building on `dimefox` etc. and you get an error message "clang-6.0: error: linker command failed with exit code 1 (use -v to see invocation)", it means you probably forgot to do '`source eng_unimelb_setup`' first.

4.3.4 Disabling Debug Output

The program outputs some (debugging) output by default. This can, and probably should, be turned off by re-building it with the compiler flags set to include '`-DNDEBUG`'.

First remove the old binaries by doing:

```
make clean
```

Then re-build, setting the `CFLAGS` variable to include '`-DNDEBUG`':

```
CFLAGS=-DNDEBUG make
```

4.3.5 Specifying the Compiler to Use

This is *not* recommended, as a recent version of Clang is required to build the more interesting versions of the program (see Section 4.3.6 below).

By default, the `Makefile` tries to build using the compiler `clang-6.0`. If you need to build using a different compiler (version), then you can do that by setting the `CLANG` variable, e.g.:

```
$ CLANG=clang make
```

This would cause building using the `clang` compiler (which is probably some default version of Clang that might already be present on your machine).

4.3.6 Other Build Targets

Besides the `dc` binary, the `Makefile` knows how to build the other following *targets*, which are different versions of the `dc` program.

`dc-san` the program built with AddressSanitiser enabled, for detecting memory errors. You should use this to test triggering of your security vulnerabilities (see Section 5.2).

`dc-fuzz` a libFuzzer target for the program. When run, this binary uses LLVM's in-built greybox fuzzer, libFuzzer, to fuzz the program. You should use this to compare your fuzzer against (see Section 5.4.2).

`dc-cov` the program built to gather coverage information. You should use this to help evaluate your fuzzer (see Section 5.4.3).

To facilitate automated fuzzing and evaluation, these other targets behave slightly differently than the `dc` program. Specifically, they do not save the variables to a file when executing the "save" command. This allows them to be run without modifying the filesystem, which is necessary to facilitate automated fuzzing and evaluation.

5 Your Tasks

5.1 Task 0: Documenting Team Roles and Responsibilities

Before you start on the assignment, the first thing you need to do is to decide *as a team* how you will carry out the various tasks of the assignment. **You need to document your decision making (see below).**

Each team member should read these PDF instructions carefully in their entirety, to understand what is required of your team. Then as a team you need to work out how you will divide the work between your team members. Naturally, you should divide the work fairly while ensuring that tasks are matched to the skills of each team member, as much as practicable.

Finally, you need to *document* who will carry out the major tasks, the responsibilities and expectations of your group, and consequences for team members who do not deliver on their agreed responsibilities.

You should use the *Group Agreement* to document the outcomes of these discussions, which can be found on the assignment page of the LMS, and email a copy of this completed document to Toby: `toby.murray@unimelb.edu.au`

5.2 Task 1: Inserting Memory-Error Vulnerabilities

Your first task is to create 5 vulnerable versions of the software. In the subject repository are 5 identical copies of the program in the directories: `src/vuln-1/`, `src/vuln-2/`, . . . `src/vuln-5/`

Your goal is to construct *reasonable* vulnerabilities that are non-trivial to exploit. Part of the marks for your vulnerabilities will be determined by the number of other teams whose fuzzers can trigger them: the lower this number is, the better for your team *up to a point.*

Exploitability Your vulnerabilities must be *exploitable.* By this we mean that if we take your vulnerable version and compile it with LLVM's memory error sanitiser, AddressSanitiser, enabled, then there must exist at least one input that causes AddressSanitiser to report a memory error when run on your vulnerable version.

Proof of Concept You will *prove* that each of your vulnerabilities is exploitable by submitting with it a *proof of concept* (PoC). A PoC is an input that triggers your vulnerability under AddressSanitiser as described above.

Your PoCs will live in the `poc/` directory. The PoC for vulnerable version i you will place into the file: `vuln-`*i*`.poc` in that directory, which already exists as an empty file in the repository.

Confirming Exploitability of Your Vulnerabilities You can confirm if your vulnerabilities are exploitable as follows. Suppose we have a candidate PoC input for `vuln-1` in `poc/vuln-1.poc`.

Building with `make` produces binary for the `vuln-1` version in the `bin/vuln-1` directory. The `dc-san` binary in that directory is the `vuln-1` version compiled with AddressSanitiser enabled. You can then run

```
./bin/vuln-1/dc-san poc/vuln-1.poc
```

to observe if AddressSanitiser reports a memory error.

Reasonableness To receive full marks, your vulnerabilities must not only be exploitable but also be *reasonable.* By this we mean able to be triggered with non-negligible probability. An example of an unreasonable vulnerability is one that is triggered only when the user enters a particular number, or uses a particular variable name, or one that is triggered only if the variable name is a palindrome, or other situations that would be almost impossible to trigger via ordinary fuzzing.

The general rule to decide if a vulnerability is reasonable is to ask: "is it conceivable that an ordinary software developer might have made this mistake when implementing the program?". Another way to argue that a vulnerability is reasonable is, of course, to base it on an existing vulnerability that has been reported in similar software.

Naturally there is no hard and fast rule about reasonableness. We will judge by examining each of your vulnerable versions manually.

Ask on the Discussion Board if you want further clarification, *without revealing details of your vulnerabilities*, or email to ask privately.

Where to avoid inserting your vulnerabilities Recall from Section 4.3.6 that the other targets (including `dc-san`) have certain features disabled, namely saving to files. Therefore you should *not* add vulnerabilities to these parts of the code, since they will be impossible to trigger in `dc-san`. Specifically, do not insert your vulnerabilities in the parts of the code between "#ifdef DC_FUZZ ... #else" and do not modify or remove these "#ifdef ... #else ... #endif" directives.

5.3 Task 2: Design and Implement a Fuzzer

Your team needs to implement a fuzzer that generates input for the program, attempting to trigger other teams' (and your own team's) memory corruption vulnerabilities.

Your fuzzer should be implemented in Java, and lives in the `fuzzer/` directory of the repository. The top-level file for it is already provided as a stub in `Fuzzer.java`.

It should build with Java 8 and we should be able to build and run it by running the `run_fuzzer.sh` (unmodified) from the top-level directory of the repository.

Your fuzzer should place its output in the file `fuzz.txt`, overwriting that file if it exists already.

You are free to use whatever techniques you like in your fuzzer, including random fuzzing, mutation fuzzing, generation-based fuzzing, or some combination of these techniques.

When designing your fuzzer, you should consider what techniques are most appropriate to use and combine, given the function and design of the program under test. A good fuzzer will be one that is able to fuzz all of the various parts of the software.

We will run your fuzzer multiple times to collect multiple outputs from it, so it should be built to produce a different output each time it is run (although always in the same file `fuzz.txt`). If you want to design your fuzzer to save state between runs, you'll have to have it save its state in a custom file that it creates itself, separate to `fuzz.txt`.

For random numbers, you might wish to use the standard `java.util.Random` class.

5.4 Task 3: Evaluation

A non-trivial part of this assignment involves you critically *evaluating* your own fuzzer. We have provided some initial options available to you when evaluating your fuzzer. However, to get full marks here we want you to think creatively about how you can use each of them (and any others that you devise) in a carefully designed experiment from which you draw high quality insights (see below). Simply using each of these methods is not enough to guarantee a high mark for this part of the assignment: the options below are *starting points* to get you thinking. We want to see creativity and insight, especially in how you design the experimental evaluation.

5.4.1 Triggering Your Own Vulnerabilities

Naturally, your fuzzer's ability to trigger your own security vulnerabilities is relevant to its performance. If your fuzzer has created the output `fuzz.txt`, then you can determine whether e.g. it triggers `vuln-3` by running:

```
./bin/vuln-3/dc-san fuzz.txt
```

But of course your vulnerabilities are only a very small sample.

5.4.2 Comparison to Greybox Fuzzing

You should also compare your fuzzer against an off-the-shelf coverage-guided fuzzer like AFL or LLVM's libFuzzer.

The `Makefile` builds a libFuzzer-based fuzzer, with AddressSanitiser enabled, for each of your vulnerable versions. You can fuzz the ith vulnerable version using libFuzzer by running the binary '`dc-fuzz`' in the `bin/vuln-`i directory.

Unlike your fuzzer, this fuzzer generates output and runs it directly against the program under test, and does so repeatedly until it causes a crash (e.g. triggers AddressSanitiser to abort the program because a memory error occurred). Along the way it builds up a *corpus* of interesting inputs that then form a test suite for the program being fuzzed.

Each `dc-fuzz` binary accepts various command line arguments to control how the fuzzer behaves, where the corpus and crash outputs are placed in the filesystem, etc.

You can get a summary of these by running

```
./bin/original/dc-fuzz -help=1
```

which prints out usage information for libFuzzer.

You should read the libFuzzer documentation at `http://releases.llvm.org/6.0.1/docs/LibFuzzer.html` (for Clang 6.0) to understand what arguments you can pass and how you can control fuzzing is performed and how results are collected. Note that the `Makefile` already handles for you the steps in the "Fuzzer Usage" section of that documentation.

5.4.3 Coverage

You should also evaluate your fuzzer (including against libFuzzer) by gathering source coverage statistics.

The `Makefile` already generates a target `dc-cov` that automatically collects coverage statistics, and there is a script `get_coverage.sh` in the top-level directory for producing coverage reports from a test suite (corpus).

For example, suppose you have a test suite and you wish to measure the line coverage it achieves. Suppose your corpus is a set of files in the directory <`some-dir`>. You can run:

```
./get_coverage.sh <some-dir>/
```

Naturally, consult the LLVM documentation for further information about generating coverage reports and other coverage statistics. `get_coverage.sh` should be considered a starting point.

5.4.4 Designing Your Evaluation

You need to carefully design an evaluation (series of experiments and measurements) of your fuzzer, employing the above tools and any others that you deem worthwhile. We are particularly interested in seeing results that shed light on the quality and trade-offs made in your fuzzer's design.

5.5 Task 4: Report

Along with your repository, you also need to submit a written report.

Your report needs to include the following:

Team Members and Responsibilities A list of your team's members and what each of their responsibilities were during the project.

Your Vulnerabilities A description of each of your vulnerabilities, under what circumstances it is triggered, and what its potential consequences might be if present in a real implementation (e.g. in terms of integrity, availability, confidentiality, code execution, etc.).

Fuzzer Design A description of your fuzzer's design, including the techniques it employs, the trade-offs you made in its design, etc.

Evaluation Your evaluation. This should include a description of the experiments you performed, their results, and your conclusions from them. Remember we want to see you drawing insights into how your fuzzer's design affects its performance.

Your report should be submitted as a PDF document and be no longer than 10 A4 pages. Anything longer than 10 pages we make no promises to read when marking your report.

5.6 Task 5: Individual Report (Optional)

Each team member also has the option to submit an individual report of no more than 0.5 page (2-3 paragraphs) in plain text. The purpose is to allow individual team members to report on

- What they contributed to the team
- What they perceived other teams members to have contributed

While the project is done as a team, we reserve the right to assess members of a team individually.

The individual report is optional, but you won't be given the opportunity to submit one after the deadline if your mark is affected by the individual report of one of your team members.

Anything longer than 0.5 pages will not be read.

6 Fuzzing Tournament

Part of your marks for this assignment will come from us automatically competitively evaluating your fuzzer and vulnerabilities against those of other teams, following a similar approach to that used in Assignment 1 for test cases and mutants. The goal of the tournament is to see whose fuzzer can trigger the most vulnerabilities and which teams's vulnerabilities are hardest to trigger.

The tournament will provide valuable feedback on your progress, and will operate on the `master` branch of your repository in the same manner as for Assignment 1. Therefore, it is in your interests to get started early and to keep your `master` branch in a working state.

7 Marking Criteria

7.1 Tournament (5 marks)

We will run your fuzzer some fixed number *NUMFUZZ* times to produce *NUMFUZZ* inputs to the program. (We will determine an appropriate value for *NUMFUZZ* during the course of the assignment, but it will likely be no smaller than 10.)

We will then run these against all vulnerable versions of every team to derive a score:

$$\mathit{fuzzScore} = \frac{k}{T}$$

where k is the number of vulnerable versions whose vulnerabilities were triggered by one of your *NUMFUZZ* inputs, and T is the highest number of vulnerabilities triggered by any other team's fuzzer (i.e. the number of vulnerabilities triggered by the team whose *NUMFUZZ* inputs triggered the most vulnerabilities).

The maximum possible score is 1, which will be scaled to 2.5.

For the vulnerabilities, they will be scored identically as to how mutants were scored in Assignment 1.

$$vulnScore = \frac{\Sigma_i^5 \Sigma_j^N a_{i,j}}{T \times N}$$

Here N is the total number of teams, and $a_{i,j} = 1$ if vulnerable version i is not triggered after being executed on the *NUMFUZZ* inputs generated by team j's fuzzer, and T is the highest number of vulnerable versions of any single team not triggered by any team in the class.

The maximum possible score is 1, which will be scaled to 2.5.

7.2 Report (20 marks)

Criterion	**Description**	**Marks**
Vulnerabilities	Reasonableness, clear and concise explanation	2.5
Fuzzer	Clear, well motivated design and trade-offs, considering the various aspects of the program under test	5
Evaluation Design	Logical, thorough experimental design, clearly explained	5
Evaluation Execution	Evaluation carried out according to its design	1
Results Presentation	Clear presentation of results using appropriate formats (graphs, tables etc.)	1.5
Discussion	Discussion of results shows insight, especially into the consequences of your fuzzer's design	5
Total		20

8 Submission Validation

We will be automatically running your fuzzer and vulnerable versions, and part of your marks for this assignment will depend on these running correctly on our setup.

As in Assignment 1, each time you push to your repository we will automatically run your fuzzer and your vulnerable versions against the output from other team's fuzzers. You should refer to the logs automatically generated by gitlab to check that your code is working as expected on our platform and to make sure that any eligibility issues are resolved well in advance of the submission deadline. Problems here at the time of the submission deadline will not be grounds for extension and will result in late penalties being applied.

9 Submission

Your team's submission includes the following parts:

1. Your group's repository, which we will clone at the time of the assignment deadline.
2. Your group's report, which should be submitted via the LMS as a PDF document.
3. Individual reports, which should be submitted via the LMS.

10 Late Submissions

As per [1], the penalty for late submissions is 10% of the assignment mark for each day late, up to seven days total. After seven days, submissions will no longer be accepted.

Extensions will only be granted for the situations outlined in the policy, meaning that extensions are not granted if other assignments are due around the same time. It is the responsibility of individuals to plan the use of their time effectively to meet assignment deadlines.

11 Academic Misconduct

The University misconduct policy[2] applies. Students are encouraged to discuss the assignment topic, but all submitted work must represent the individual's understanding of the topic.

The subject staff take academic misconduct very seriously. In this subject in the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

[1]See `http://www.policy.unimelb.edu.au/schedules/MPF1029-ScheduleA.pdf`

[2]See `https://academichonesty.unimelb.edu.au/`