# Project #1-B

Computer and Network Security — Released: 30Aug2021
COMP-5370/-6370 — Due: 09Sept2021at 6pm CT

Project 1-B is due **09Sept2021at 1800 CT** and you may choose to use your "late days" as discussed in the syllabus. The final deadline is 11Sept2021at 1800CT (using both) and any submission after that time will be a zero (0). Submissions will be accepted through Canvas prior to the initial deadline or by emailing to the TA between the initial and final deadline.

**You may work in groups of up to 3 if you wish to do so.** It is entirely your choice but the responsibility of forming, communicating, coordinating, and collaborating is yours as well. The groups may be across any sections of the course and are not restricted by in-person or distance-learning aspects. Discord is an excellent way to find other people who wish to work in a group.

**Part 3** is, however, an individual assignment as it amounts to fixing your corresponding Project 1-A implementations.

As always, you are *highly encouraged* to **read the entire assignment** and think-through your approach prior to attempting to complete it. Any and all questions are welcome during Office Hours, on Discord, after-class, and through any other reasonable channel but your ability to reason about the scenario and make trade-offs on what *you* think are the pro's and con's is included in the grading scheme.

## Overview

In Project 1-A, you built a library capable of marshalling and unmarshalling the `nosj` data format while using the Security Mindset. Through practicing the "think like a defender" portion, you not only engineered a safe, correct, and maintainable implementation but also thought-ahead and tried to account for what the auto-grader might throw at it.

In Project 1-B, you will get to practice using the "think like an attacker" portion of the Security Mindset to not only analyze the scenario to identify likely errors, but also to generate Proof-of-Concept (PoC) test-cases which leverage that analysis. You will be using the same specification from Project 1-A and a test-harness (`test_adversarially.py`) has been provided for you to incorporate your test-cases into (both available on the course website).

Unlike Project 1-A, **6370 students have additional requirements** for Project 1-B. These are required for all students in the COMP-6370 sections of the course (distance learning and in-person) but COMP-5370 students may complete for bonus-points. Details are described below.

## Part 1: Adversarial Test-Cases

**⋆⋆⋆ REQUIRED FOR ALL STUDENTS ⋆⋆⋆**

For this part of the project, you will deeply analyze the scenario and devise five (5) independent test-cases which test for realistic *vulnerabilities* in others' implementations and would serve as a Proof-of-Concept exploit (PoC). To be clear, it is **NOT** sufficient to simply test for a *bug* in others' implementations (difference discussed in Lecture #01). For each test-case, you are required to:

1. Clearly mark which test-case in `test_adversarially.py` is being discussed.
   - See the provided test-harness for an example.
2. Describe how implementations which fail your test-case could be abused to the advantage of the actor supplying the input value.
   - This is your justification for reasonable belief that failing the test-case indicates that the weakness could be leveraged to the attacker's gain and victim's loss.
   - A simple failure to correctly process a valid input is not a Denial of Service vulnerability as the attacker is the only one being denied service.
3. Provide what the *root-cause* of this vulnerability is likely to be and justification for that assertion.
   - The root-cause should be an abstract action that is not specific to this project such as "Misinterpretation of the Specification", "Input Validation Failure", etc.

You **are required** to implement this part using Python 3 and your submission **must** automatically test an arbitrary implementation when `pytest` is called. This is identical to how the provided test-cases and any self-added test-cases were ran for Project 1-A.

### Expectations for Part 1

In addition to the expectations listed for all portions of the project (see below), it is expected that your test-cases will be ran via the `pytest` command. You may assume that the exact `exceptions.py` module and the `serialization.py` + `deserialization.py` implementation under-test are available in the current directory. Failures should be reported in a helpful manner consistent with the `pytest` conventions*.

Your `write-up.txt` file **must** be a text-file and not a Word doc, PDF, etc.

### Components Provided

**test_adversarially.py** This is a framed-out Python3 module for Part 1. Though not required, you are *highly encouraged* to build-off of this template (see submission details).

**write-up.txt** This is a framed-out text file for your write-up as expected by the auto-grader. Though not required, you are *highly encouraged* to build off of it to aid the auto-grader in parsing it correctly.

## Grading

Submissions will be graded based on:

- Submission of five (5) independent test-cases and the accompanying text for each.
- Your discussion of the likely root-cause for each test-case.
  - Both its alignment to your test-case and its logic/rationale will influence
- Your justification of why each test-case is testing for a vulnerability and not a simple bug.
  - Both its alignment to your test-case and its logic/rationale will influence
- Each test-case's "believability" in finding errors in private implementations.

*You can report manually-created error messages via `assert(<arbitrary condition>), '<arbitrary message>'`.

  - Finding of an error in a given implementation is not proof-positive and the lack of finding an error is not proof dis-positive.

- Meeting the expectations of the submission.

# Part 2: Automatically Generating Adversarial Test-Cases

⋆⋆⋆ **REQUIRED FOR COMP-6370 STUDENTS** ⋆⋆⋆
⋆⋆⋆ Bonus for COMP-5370 students ⋆⋆⋆

Where as Part 1 generates adversarial test-cases via 100% manual effort, this part of the project will do so with zero per-test-case manual effort. "Fuzzing" is a common testing approach which relies on automatically generated values being used as input to the component under-test in order to test a pre-defined "invariant". This invariant should **ALWAYS** be evaluated to *True* regardless of the contents of any given input (i.e. a tautology). An extremely common invariant used in fuzzing is "the code under-test should not encounter a segmentation fault (segfault)". Due to the fact that it is difficult (but not impossible) to cause a segfault in Python's interpreter, this would be a poor invariant to select.

## Expectations for Part 2

In addition to the expectations listed for all portions of the project (see below), it is expect that your submission:

1. Will generate and test *n* inputs (passed as `--count` flag) and then exit with a exit-code of zero (0).
   - Handling of this value is not under-test and it is safe to assume that it will always be provided. See the provided template for handling.
   - Exiting with zero should occur implicitly if not triggered by an exception but can be explicitly exited with `os.exit(0)`.
2. Will generate deterministic but unpredictable test-cases based on the value passed to your fuzzer via the `--seed` flag.
   - Passing the same seed-value twice must result in the same inputs being generated.
   - Handling of this value is not under-test and it is safe to assume that it will always be provided. See the provided template for handling.
3. All inputs will be written as a line-separated, validly-formatted `nosj` map to the `<seed>.inputs` file in the current directory.
4. All inputs causing a failure will be similarly written to the `<seed>.failure` file in the current directory.
   - An empty file indicates no failures were found.

## Components Provided

**fuzzer.py** This is a framed-out implementation of a fuzzer described above. It **DOES NOT** implement good, safe, or useful fuzzer-logic and is only provided for an example of using the seed-value, command-line arguments, and various other python-specific attributes.

### Language Requirements for Part 2

You are *highly encouraged* to implement using Python3 but **are not** required to. Standard programming languages (C, C++, Java, Golang, etc.) will be considered with analogous constraints but any student who wishes to use an alternative language **must** contact the instructor by Thursday, 02Sept2021 at 1800 CT in order to A) obtain approval B) acknowledge language-specific constraints, and C) negotiate on how it will be incorporated into the auto-grader. Formal approval will only be provided via an email from the instructor discussing the above aspects and explicitly stating:

```
################################################################################
# YOU ARE APPROVED TO USE LANGUAGE XXXX FOR PROJECT 1-B WITH THE BELOW
# CONSTRAINTS AND ASSUMPTIONS.
################################################################################
```

## Grading of Part 2

Submissions will be graded based on:

- The "believability" of your fuzzer finding errors in private implementations.
  - Finding of an error in a given implementation is not proof-positive and the lack of finding an error is not proof dis-positive.
- Coverage over the major `nosj` data-types (map, string, int, etc.).
- Covers both the `marshal()` and `unmarshal()`.
- Meeting the expectations of the submission.

## Submission Details (Parts 1 and 2)

Only one member of each group should submit parts 1 and 2 for the group. Include a `group-members.txt` file containing the first and last name of each group member on a separate line. Those who choose to work individually on parts 1 and 2 must also include a `group-members.txt` file (with one line).

Various expectations of your submission are listed below and **they are non-negotiable**. If you submission fails to meet these expectations, you may receive a zero (0) for the entire project. If you have any questions, about submission format, details, contents, or anything discussed below, seek clarification rather than making assumptions.

**Expectations:**

- It must contain **only** the files explicitly listed above for each part.
- Your implementation **must not** contain unrelated functionality with or without suspicious intent. The explicitly dis-allowed functionality includes, but is not limited to:
  - Writing directly to a file other than those specifically listed for Part 2.
  - Reading anything including files, variables, configurations, environment details, etc.
  - Making network requests or communicate with *any* other component.
  - Attempting to install packages, modules, or any other software.

- Your submission must be named "<*first-name*>-<*last-name*>-`project-1b.tar.gz`" using the first and last name of the submitter, and be a gzip'd tarball[†]
  - `tar -czf fname-lname-project-1b.tar.gz fuzzer.py test_adversarially.py group-members.txt write-up.txt`
  - Zipfiles **are not** acceptable
- The files must be in the root of the tarball and not nested in a directory

**Restrictions:**

1. Your submission **must** use a recent version of the language.
   - The current Ubuntu 20.04 LTS Desktop package for that language is an excellent "measuring-stick".
   - Python 2 is dangerous and wholly unacceptable as are early versions of C/C++ which do not provide reasonably modern feature via the standard language[‡].
2. Your implementation **must not** error, crash, or otherwise fail to run regardless of the reason.
3. Your implementation **must** gracefully handle any error thrown by the underlying `nosj` implementation under-test.
   - You **are** allowed to "catch" any Python 3 exception thrown via the built-in exception handling mechanism (`try-except-finally`).
4. Your implementation **may not** make use of resources not self-contained in the language itself with the exception of the `pytest` Python 3 module and other libraries explicitly allowed.

# Part 3: Fix-It

⋆⋆⋆ Bonus for all students ⋆⋆⋆

After beginning this project, you may realize that your implementation for Project 1-A incorrectly handles certain valid inputs. If you choose to, you may continue-on to the "fix-it" part of the Build-It/Break-it/Fix-It scheme being used in Project 1-A and 1-B.

Every effort will be made to complete grading for Project 1-A in order to give you ample time to address any deficiencies found by the grading test-cases. It is *highly recommended* that you review that feedback as soon as possible once it has been returned not only for your own awareness of failed grading test-cases but also for requesting re-grades.

# Expectations for Part 3

The expectations are identical to those laid-out in Project 1-A with the exception of:

- You **may not** work collaboratively on your fixes.
  - You may find mishandled test-cases collaboratively but may not re-use code across your implementations.

[†]The submission name may also contain a trailing `-n` if resubmitted on Canvas.

[‡]C89's lack of Unicode and C++89's lack of smart pointers are obvious examples.

- You **must** build off of your existing implementation and apply patches to your existing implementation.
  - You **may not** start-over and build from-scratch.
- You **must** submit a diff between your initial implementation and your fixed implementation.
  - `diff XXXX.py fixed_XXXX.py > XXXX.diff`
- You **must** submit your corrected implementation modules as `fixed_serialization.py` and `fixed_deserialization.py` within the tarball.

## Grading for Part 3

Grading will be performed identical to Project 1-A and bonus points will be added based on:

- Passing all provided test-cases
  - If any test-cases provided for the original assignment are not passed, no bonus points will be awarded.
- The average of your initial submission's grade and fixed submission's grade will be the new grade for Project 1-A
  - This means that each newly-passing test-case returns half the points deducted for it failing originally.
  - This also means that a newly-failing test-case will remove the same number of points.

## Submission Details (Part 3)

The submission details for Part 3 ("Fix-It") are similar to the Project 1-A assignment. There will be a separate Canvas assignment which each student will upload their fixed implementation to if they wish to try for bonus points. The only exception to this is the use of a different command to create the submission file due to the changes in filenames and addition of a diff files:

```
tar -czf fname-lname-project-1a-fixed.tar.gz fixed_serialization.py \
   fixed_deserialization.py serialization.diff deserialization.diff
```