Project 1A

Computer and Network Security — Released: 21Aug2024
COMP-5370/-6370 — Due: 04Sept2024 at 6pm CT

Part A of this project is due **Wednesday, 04Sept2024 at 6pm CT** and must be submitted through the Canvas assignment (if early/on-time) or by emailing the TA (if late). Late assignments will be penalized as described in the syllabus.

This project must be completed individually

Overview

In order to effectively evaluate security and privacy properties, it is important to not only analyze what *should* happen but also what *could* happen. An excellent microcosm of this concept is seen in software that serializes and deserializes data formats (also called marshalling/unmarshalling) and is a well-known and widely-abused source of bugs and exploitable vulnerabilities. This is further complicated by the numerous overly complex data formats common to computers and the Internet which can sometimes provide arbitrary functionality through such mechanisms as in-document JavaScript/macro execution*.

For this project, you will play the role of both a *builder* during 1A (defensive-oriented) and a *breaker* during 1B (offensive-oriented). The specification for your software is an entirely made-up data format named `nosj` and is available in the `specification.txt` file. This specification also contains a number of examples and though not required, it is **highly recommended** that you build a simple testing apparatus to validate your implementation. Additionally, it is **highly recommended** that you expand upon these examples to include *new and non-obvious* corner-cases which you identify during your analysis and implementation.

You **are not** required to complete this project in a specific language but Python, Java, and Golang are *highly recommended* by the instructor. If you wish to use any language outside of these three, you **MUST** discuss with the instructor **prior to 28Aug2024** to ensure that the auto-grader tooling can handle/be patched to handle your chosen language. Allowable compiler/interpreter versioning, dependencies, build system, etc. must be discussed and agreed upon but other than being readily available on the current Ubuntu distros without UI requirements, they are negotiable. The instructor will not forbid any specific language in its totality but will beg you not to use a memory unsafe language for reasons that you will come to understand during this course.

Build-It - Deserialization

You will implement stand-alone application/script which will read from a nosj formatted input file (passed as the first command line argument) and print a templetized description of that input to standard output (`stdout`). The output format **MUST** be lines of: "`key-name -- type -- value`" (note there is no leading/trailing whitespace other than the trailing newline ("\n"/ 0x0A) and the spaces on either-side of double-dash).

In the case that a map is encountered, your implementation **MUST** leave the value field of the above line empty and output a stand-alone line of "`begin-map`". When the end of a map is encountered, your implementation **MUST** output a stand-alone line of "`end-map`".

* *Instructor glares at PDFs and MS Office.*

Catch-It: Handling errors

If you recognize input errors that makes it impossible to safely unmarshal or is in conflict with the specification, you should print a simple, 1-line error message to standard error (`stderr`) and immediately exit with a status code of `66`. This error message **MUST** begin with the string "`ERROR -- `" (note the space on either side of the double-dash) and end with a single trailing newline (("\n"/ 0x0A).

Restrictions

Below is a non-exhaustive list of restrictions which your implementations are expected to comply with. Penalties of varying degrees will be applied if these expectations are not met.

1. If using a recommended language, you **MAY NOT** use an overly out-dated or unsafe version of any language. Any version of Python2 (notably Python 2.7) is dangerous and wholly unacceptable due to them being EOL'd. **Relying on the Python2 interpreter will result in an immediate grade of 0!**

2. If using any other language, it **MUST** be discussed with the instructor and grading details negotiated no later than 6pm CT on Wed 28Aug2024.

3. Your programs **MUST NOT** attempt attempt to interfere with the auto-grader workflow (see "Ethics, Law, and University Policies" section of the syllabus).

   This includes (but is not limited to) attempting to alter the environment, write/modify files, escalate privileges, make network requests, delete the OS, etc.

4. Your programs **MUST NOT** import/include/etc. any library that is not built-in to the language unless explicitly approved by the instructor in-writing. Examples:

   Using the built-in `re` or `urllib` in Python or analogous `java.util.regex.Pattern` or `java.net.URLDecoder/URLEncoder` in Java are acceptable.

   Using the `regex` Pip package or "Guava" libraries for Java are not acceptable.

5. Your programs **MUST NOT** print anything to `stdout` or `stderr` except for the information listed above as your program's output is character-matched against the known solution for correctness.

   This includes debugging information and any other extraneous text.

6. Your programs **MUST NOT** crash or exit on an exception regardless of the input.

   A segfault will result in failure to pass the test case and an additional penalty.

   "Cheap Tricks" to avoid crashing[†] are not acceptable. *The TA and instructor were once Computer Science students too and know many, if not most of the tricks…*

Submission Details

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 penalty and may receive a zero (0) for the entire project as discussed in the syllabus. If you have any questions, about submission format, details, contents, or anything discussed below, seek clarification ahead of the deadline rather than making assumptions.

[†] Examples: blanket `try-catch` around the main function in Python or deferring a call to `recover()` in Golang

Expectations:

1. You should submit a single uncompressed tarball. **You MAY NOT submit a zip file**.
2. The TA and instructor must be able to compile and run your code as described below.
3. You must have a `Makefile` in the root directory with two targets:

   `build` — Must compile your code from scratch and exit successfully. If a non-compiled language (e.g., Python), this target is not required to do anything (i.e. "`exit 0`").

   `run` — Must pass the `FILE` argument to `make` to the above compiled program and transparently pass `stdout` and `stderr`.
4. If you are using a build-system (i.e. gradle, maven, CMake, etc.), you **MUST** discuss with the TA **prior to 28Aug2024** to ensure that the autograder is able to run your "`make build`" target. Common/Widely-used build systems are acceptable but A) the TA/Instructor reserve the right to reject build-systems and B) it is *your sole responsibility to configure them*.
5. You are welcome to develop your solution in an IDE but your code **MUST** be able to be compiled and ran via a Linux shell as described above.
6. By default, code will be ran on an up-to-date version of Ubuntu (amd64) without GUI functionality. If you believe your code must be compiled/ran on a different OS or ISA for any reason, you must contact the instructor prior to submission and obtain such approval in-writing.
7. Your submission **MUST NOT** contain any pre-compiled binaries, object files, or byte-code.

Grading

Per the syllabus, your solution will be graded in an auto-grader style workflow in which many inputs will be passed to evaluate the handling of arbitrary input (both correct and incorrect). This will include both the released test cases and others which have not been released. Passing all released test cases only and only the released test cases **will not result in a passing grade**.

Weights

As discussed in the syllabus, penalties may be added as necessary but the baseline weighting will be:

30% Passing released test cases.

30% Passing unreleased correctly-formatted test cases.

20% Passing unreleased error test cases.

10% Useful error messages.

10% Code quality (i.e. readability, understandability, following language conventions, documentation of non-obvious functionality, etc.)

Errata

(none)