Project 1A

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 (sometimes called marshalling/unmarshalling data). This is a well-known and widely-abused source of bugs and exploitable vulnerabilities. This is further complicated by overly complex data formats which may provide arbitrary functionality through such mechanisms as in-app JavaScript/macro execution[*].

For this project, you must play the role of both a *builder* (defensive-oriented) and a *breaker* (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 test *new and non-obvious* corner-cases which you personally identify.

You **are not** required to complete this project in a specific language but Python, Java, and Golang are the *recommended* languages. If you wish to use any language outside of these three, you MUST discuss details required for grading (e.g., compiler/interpreter versioning, dependencies, etc) with the instructor **prior to submitting your assignment**. The instructor will not forbid any specific language in its totality but may beg you not to use a memory unsafe language for reasons that will be covered later in the semester.

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 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-dashes).

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`”.

Catch-It: Handling errors

If you recognize input errors that makes it impossible to unmarshal from the nosj data format 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 “`ERROR`

`--` ”.

[*] *Instructor glares at PDFs and MS Office.*

Restrictions

Below is a list of 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. Examples:

   Python 2.7, 3.0, and 3.6 (among others) are dangerous and wholly unacceptable due to them being EOL'd. **Relying on the Python2 interpreter will result in an immediate grade of 0!**

   Python 3.8.10 (the current version shipped with Ubuntu) is not overly out-of-date nor dangerous even though it was released 03May2021 and is still no longer accepting non-security patches.

   Golang 1.18 (released 15Mar2022 but surpassed by 1.19 on 02Aug2022) is not overly out-dated or unsafe.

   Example: Golang 1.1 (released 13May2013) is overly out-dated.

2. If using any other language, it **MUST** be discussed with the instructor and grading details negotiated no later than 6pm CT on Wed 24Aug2022.

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.

   This includes debugging information and any other extraneous text.

6. Your programs **MUST NOT** crash regardless of the input.

   A hard-crash will result in failure to pass the test case and an additional penalty.

   "Cheap Tricks" to avoid hard-crashing[†] will result in a penalty. *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 will receive a penalty and 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 ahead of the deadline rather than making assumptions.

[†]Examples: overly broad `try-catch` handling in Python or deferring a call to `recover()` in golang

Expectations:

1. The TA and instructor must be able to compile and run your code as described.
2. You must include a “`BUILD.txt`” with sufficient details on how to compile/run your programs from a command line.

   If a built-system is not required (i.e. Python) or a built-in build-system (`go build` or `javac`), it is perfectly acceptable for this to be a single sentence.

   If you are using a built-system with is not obvious (i.e. gradle, maven, Makefiles, CMake, etc), you must provide sufficient information to compile your implementations.

   You are welcome to develop in an IDE but your code **MUST** be able to be compiled and ran via command line using your documentation.
3. By default, code will be ran on an up-to-date version of Ubuntu (amd64 or arm64-style ISA). If you believe your code must be compiled/ran on a different OS or ISA, this must be approved by the instructor prior to submission.
4. You should submit a single uncompressed tarball and **NOT** a zip file.
5. Your submission **MUST NOT** contain any pre-compiled binaries, object files, or byte-code.

Grading

Per the syllabus, your program 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.

50% Passing unreleased test cases.

10% Useful error messages.

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

Errata

- Minor typo corrections