Project 1B

Computer and Network Security — Released: 07Sept2025
COMP-5370/-6370 — Due: 17Sept2025 at 6pm CT

Part B of this project is due **Wednesday, 17Sept2025 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.

Overview

In Project 1-A, you practiced using the defensive-side of the Security Mindset to build an implementation capable of parsing the `nosj` data format. You not only engineered a safe, correct, and maintainable implementation but also thought-ahead and tried to account for what an actively-misbehaving auto-grader might throw at it. In the first portion of Project 1-B, you will get to practice using the attacker-side of the Security Mindset to generate Proof-of-Concept test-cases against a similarly well-implemented solution. The `nosj` specification from Project 1-A **remains unchanged**. In the second portion, you will build a proof-of-concept demonstration of how simple, brute-force attacks are possible given a deliberate approach to their implementation.

1 Break-It

This portion of the project must be completed with a single partner.

For this portion, you should select a partner within the course, exchange implementations, and use your in-depth knowledge of `nosj` from Project 1A to find corner-cases which demonstrate incorrect behavior in your partner's implementation. This may be by error-ing when given valid input, not error-ing when given invalid input, or by incorrectly handling valid input (i.e. the output is wrong). Your goal is to find three (3) different incorrect behaviors with *different root-causes*. It is important to note that three different examples with the same root-cause (i.e. triggering it with just different inputs) will be counted as only one (1). You **may not** submit any of the testcases from the specification as your testcases for incorrect behavior.

In your submission, you should provide input/output files for each erroneous behavior as well as a short, *less than 100 words*, ascii-only description/explanation identifying A) the root-cause and B) a sufficient explanation of how your partner's implementation could be patched to mitigate the issue. You **do not** have to supply a patch. If you are unable to find incorrect behavior in your partner's implementation, it is recommended that you remember that security is an *open-world* problem. Anything and everything is in-scope and creativity is often the path to success. If you still are unable to find incorrect behavior, you should discuss with the TA and/or instructor by **Monday, 15Sept2025**.

You may only have **one (1)** partner and therefore your Project 1A implementation should only be used by **one (1)** other person in the course. If you are unable to find a partner, you should contact the TA by Friday, 12Sept2025 and you will be issued a partner or an implementation to test against. It should be noted that you can begin working on possible testcases *prior* to having found a partner.

2 Brute-Force Attacks

This portion of the project must be completed individually.

Though brute-force attacks are rarely the best or most-efficient attack, they are always *an attack* that is possible and guaranteed to be successful given sufficient resources. In this portion of the project, you will demonstrate this by generating a partial collision against an otherwise-strong cryptographic hash function (SHA256). You **are not** required to generate a complete collision.

You should be cognizant of the fact that there is often a “point of diminishing returns” where it is more efficient to intentionally perform a non-optimized/inefficient attack rather than continue to optimize the attack/implementation/etc. If it takes 3 hours to optimize code that will reduce the run-time by 3 seconds, you have passed that point (hint… **HINT**).

⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆

WARNING

⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆ ⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆⋆

Depending on your implementation, **it may take many hours** for your attack to be successful. A non-optimized, single-thread, well-implemented implementation can probabilistically finish using standard, commodity hardware in a short amount of time (order hours) even if using Python. It is *highly recommended* that you validate your implementation’s logic with a further-reduced set of restrictions* prior to attempting to generate the partial collision you will submit. This will ensure that your implementation operates as you expect and you do not encounter errors such as:

- Your implementation runs but crashes before finding a solution.
- Your implementation does not find a solution even though guaranteed to be possible.
- Your implementation continues searching after finding a solution.
- Your implementation does not output the found solution.

2.1 Brute-Force Attack for *Specific* Partial SHA256 Collision

For this portion, you will implement a brute-force attack to find a *partial* SHA256 collision. The requirements for this partial collision are:

1. Both inputs to the SHA256 hash function must begin with your *root* Auburn email address (format: three letters || 4 numbers || `@auburn.edu`).
2. The trailing four bytes of the digests must all be the same value (e.g., `0x11111111`, `0x54545454`, `0xF8F8F8F8`, etc.).

The below requirements for implementation apply to all submission and submissions will be chosen at random from both the undergrad and grad-level sections to be tested. Any submission not meeting them will be penalized. It is in your best interest to be *highly confident* that your implementation will meet them if it is randomly chosen for execution. For the first and second requirements only, any submission which does not meet them be give a single re-try to do so to account for probabilistic uncertainty. These requirements are:

1. Your implementation **must be non-deterministic** such that every time it is executed, the colliding partial digest and both inputs are unique when compared to all other previous successful executions of your implementation. (*single re-try applicable*)

*Example: Only requiring the first two bytes to be identical as opposed to the first four bytes as is required for submission.

2. Your program **must** find an acceptable partial collision in less than 12 hours of run-time on commodity hardware with probabilistic bounds and given its non-determinism. (*single re-try applicable*)

3. Once the collision is found, you **must** print the input-data for *each input* to `stdout` as a BASE64 encoded string

4. Each of the above BASE64-encoded inputs **must** be on its own line using Linux line-endings. The first line should be prefixed with the string "`INPUT 1 -- `" and the second line with "`INPUT 2 -- `".

5. Your implementation should **not** write anything to `stdout` other than the above two lines but you are welcome to use `stderr` for any status, debugging, or other messages you wish with the exception of the following requirement...

6. Your program **must not** output a language-default error or error message (e.g., `segfault` in C/C++, `RuntimeException` in Java, `OverflowError` in Python, etc.).

7. Your program **must** exit with the status code of zero (`0`).

8. Your program **must not** exit with a status code other than zero (`0`).

To be clear, you **are not expected to find a complete SHA256 collisions**, there are no requirements on the input-data other than the leading email address, and there are no requirements on the output digest outside of the trailing four bytes. Additionally, you would be well-served by reviewing the canonical Computer Science tradeoff of space vs. time when planning your implementation.

Implementation Restrictions

With the exception the recommended languages and dependency restrictions discussed below, all restrictions from Project 1-A remain in the analogous way. You **are not** required to complete this project in a specific language but Python, Java, and Golang are *highly recommended* by the instructor. C and C++ are also permitted (but not recommended) with the restriction that it must be possible to compile via `make build`. If you wish to use any language outside of these five, you **MUST** discuss with the TA/instructor **prior to Friday, 12Sept2025** to ensure that the grading environment is compatible.

The standard current Ubuntu 24.04 packages for `default-jdk, default-jre, and build-essential` as well as the Golang compiler will installed. All implementations must otherwise be entirely self-contained and **may not** rely on any other package such as those for the C++ "Boost" libraries.

2.2 SHA256/BASE64 Implementations

There are widely-documented and assumed-correct implementations of both the BASE64 and SHA256 algoritms built into Python, Golang, and Java which should be used where appropriate.

As standard implementations of BASE64 and SHA256 are not readily available as built-in functions/libraries in C/C++, you may use your choice of pre-existing, acceptably licensed, and publicly available implementation as long as:

1. You accept the risks of it being incorrect/buggy (your grade depends on its behavior)

2. Its complete source is included in your submission

3. Its origin is readily apparent in your submission

4. It is publicly available for review

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.

You will submit a tarball to Canvas (contents described below) in which **all files must be ascii-only**[†]. All source code files needed to compile your implementation **must** be included as well as a Makefile with both `make build` and a `make run` command.

Expectations:

1. You should submit a single uncompressed tarball. **You MAY NOT submit a zip file**.
2. When uploading to Canvas, tarballs **must** have the naming convention of your *root* email address followed by an underscore ("_") followed by "`project-1b`"

   EX: `hzs0084_project-1b.tar`
3. The TA and instructor must be able to run your testcases (Break-It) and compile/run your code (Brute-Force Attacks) as described below.
4. Per the syllabus, your program will be graded in an auto-grader style workflow and as such, it is **highly important** that your submitted tarball has the expected filesystem structure. That structure is:

   `partner.txt` — A single-line, ascii-only text file containing your partner's AU email address. This person's implementation of Project 1-A will be used to validate your provided testcases.

   `break-it/tc-1.input`

   `break-it/tc-1.output`

   `break-it/tc-1.description`

   `break-it/tc-2.input`

   `break-it/tc-2.output`

   `break-it/tc-2.description`

   `break-it/tc-3.input`

   `break-it/tc-3.output`

   `break-it/tc-3.description`

   `partial-collision/src/` — Contains your implementation's source code including the makefile if C/C++

   `partial-collision/Makefile` — Mechanism to build and run your solution (discussed below).

   `partial-collision/1-input.txt` — Contains BASE64 string for first input.

   `partial-collision/1-sha256-digest.txt` — Contains the complete, hex-encoded, SHA256 digest for first input

   `partial-collision/2-input.txt` — Contains BASE64 string for second input.

   `partial-collision/2-sha256-digest.txt` — Contains the complete, hex-encoded, SHA256 digest for second input.

[†]You can use the `file` bash command to quickly check whether a file contains non-ascii characters.

5. You must have a `Makefile` within the `partial-collision/` 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 execute your partial-collision implementation and output the newly generated partial collision inputs as two BASE64 encoded lines to `stdout` as described above. Your implementation **must not overwrite the `1-input.txt` or `2-input.txt` files submitted**.

6. If you are using a build-system (i.e. gradle, maven, CMake, etc.), you **MUST** discuss with the TA **prior to 15Sept2025** 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*.

7. 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.

8. By default, code will be ran on an up-to-date version of **Ubuntu 24.04** (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.

9. With the exception of files under the `partial-collision/src/` directory, your submission **MUST NOT** contain any file not listed above. Your implementation under the `partial-collision/src/` directory may be structured however you wish.

10. Your submission **MUST NOT** contain any pre-compiled binaries, object files, or byte-code. This includes under the `partial-collision/src/` directory.

Grading

Per the syllabus, your solution will be graded in an auto-grader style workflow in which your submission will be tested with a pre-determined set of commands (e.g., `cd partial-collision` →`make build` →`make run` →...) and any error encountered will result in it being given a zero and returned.

Weights

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

45% Break-It portion

- 15% — First test-case
- 15% — Second test-case
- 15% — Third test-case

55% Brute-Force Attacks portion

- 10% — Inputs are in the correct format
- 10% — Trailing bytes of input-1 are same value
- 10% — Trailing bytes of input-2 are same value
- 25% — Trailing bytes of input-1 and input-2 match

Errata

(None)