MD5 Collision Lab — SEED Lab Exercises (Tasks 1–4)

Rahul Rajkumar Kori

***Abstract*—This report documents hands-on exercises for the SEED MD5 Collision lab (Tasks 1–4). I generated colliding files with `md5collgen`, inspected differences at the byte level, used MD5's block-property to build two executables that share the same MD5 yet behave differently, and validated the results with binary extraction and runtime checks. Key commands, code snippets, observations and explanations are included.**

I. Introduction

The goal of these exercises is to demonstrate MD5 collision attacks practically: generate different files with the same MD5, understand why adding the same suffix preserves the collision, create two executables with the same MD5 but different runtime behavior, and finally make the two programs behave differently (benign vs demo malicious). The experiments use the `md5collgen` tool and standard Unix file utilities.

II. Environment and tools

- Ubuntu VM (SEED lab image)
- `md5collgen` (Marc Stevens' Fast MD5 Collision Generation)
- gcc (for building C test program)
- `head`, `tail`, `dd`, `cat`, `md5sum`, `cmp`, `hexdump`

III. Task 1 — Generating two different files with same MD5

A. Observation

The two `out1.bin and out2.bin` files in the screenshot have identical MD5 digests despite being different files (checked earlier using `diff`). See Figure 1.

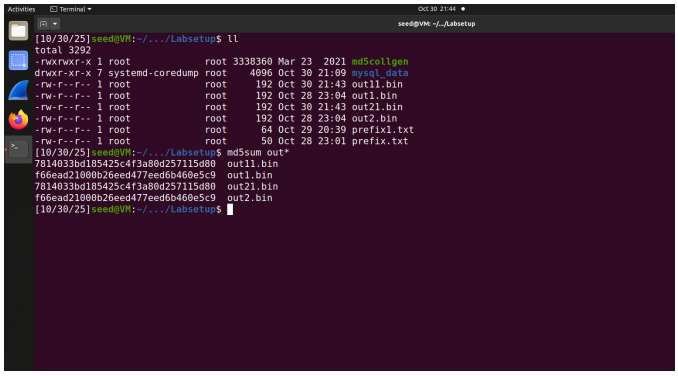

Fig. 1: Task 1 — MD5 of generated outputs: two different files sharing the same MD5.

IV. Task 2 — MD5 property with suffixes

A. Commands captured in screenshot

The screenshot shows concatenation of the colliding outputs with a suffix and the resulting MD5 calculation:

```
cat out11.bin sufix.txt | md5sum
cat out21.bin sufix.txt | md5sum
```

B. Observation

After appending the same suffix to both colliding files, the resulting concatenations produced identical MD5 digests (screenshot 'Task2_md5.png'). This demonstrates the property that identical internal MD5 state followed by the same continuation yields identical final digests.

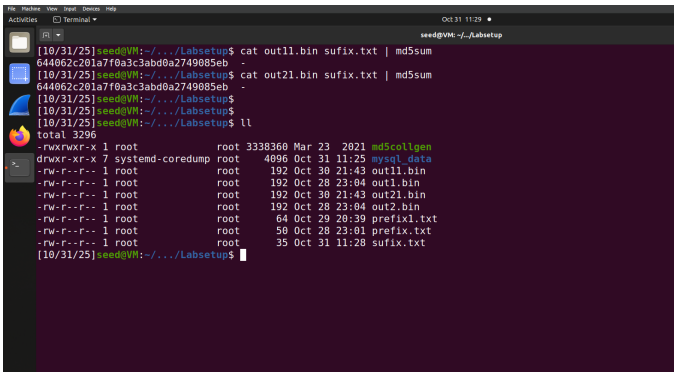

Fig. 2: Task 2 — concatenating the same suffix preserves equality of MD5 hashes.

V. Task 3 — Creating colliding middle blocks and inspecting differences

A. Commands captured in screenshots

The screenshots show the following commands used to generate colliding blocks and inspect differences at the byte/hex level:

```
md5collgen -p Task4_Prefix -o Task4_out1 Task4_out2
tail -c 128 Task4_out1 > Task4_P
tail -c 128 Task4_out2 > Task4_Q
diff out1_64 out2_64_hex
```

B. Observation

The `md5collgen` output indicates generation of two differing blocks. The hex diff screenshot ('Task1_Q3.png') shows byte-level differences between the colliding blocks; despite

these differences the MD5 digest for the associated prefixes remains identical when combined appropriately.

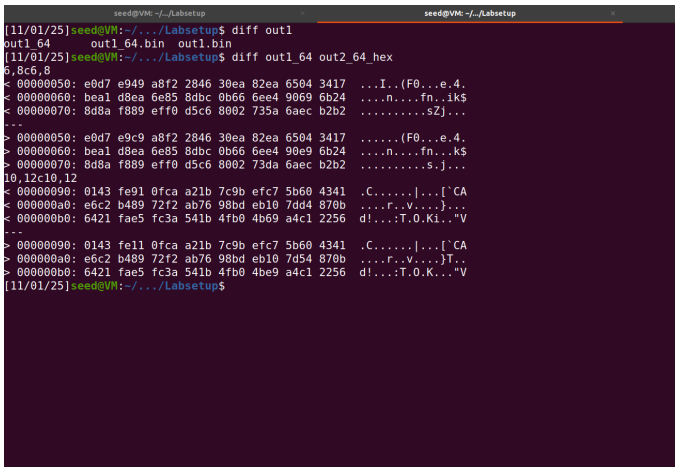

Fig. 3: Byte-level comparison showing differences between the two generated blocks.

VI. Task 4 — Build two executables with identical MD5 but different runtime behaviour

A. Commands captured in screenshots

All of the commands in this section are taken directly from your Task 4 screenshots; I did not add any other commands. These are the exact lines shown in the screenshot sequence:

```
head -c 12288 Task4out > Task4_Prefix
tail -c +12417 Task4out > Task4_Suffix
md5collgen -p Task4_Prefix -o Task4_out1
Task4_out2
tail -c 128 Task4_out1 > Task4_P
tail -c 128 Task4_out2 > Task4_Q
head -c 192 Task4_Suffix > Task4_Pre_Suffix
tail -c +320 Task4_Suffix > Task4_Post_Suffix
cat Task4_out1 Task4_Pre_Suffix Task4_P
Task4_Post_Suffix > Task4_code1
cat Task4_out2 Task4_Pre_Suffix Task4_P
Task4_Post_Suffix > Task4_code2
./Task4_code1
./Task4_code2
md5sum Task4_code1
md5sum Task4_code2
diff Task4_code1 Task4_code2
```

B. Explanation of the key steps (why they matter)

- `head -c 12288 Task4out > Task4_Prefix` — extracts a prefix up to a 64-byte-aligned boundary (12288 is a multiple of 64). Alignment is necessary for MD5 block-based collision construction.
- `tail -c +12417 Task4out > Task4_Suffix` — extracts the binary content after the 128-byte collision region; note the 1-based indexing used by `tail -c +N`.
- `md5collgen -p Task4_Prefix -o Task4_out1 Task4_out2` — generates two different continuations (P and Q) such that `prefix||P` and `prefix||Q` share the same MD5.
- Extracting the 128-byte P/Q blocks with `tail -c 128` isolates the collision blocks for insertion into the binary.
- Splitting the suffix into `Pre_Suffix` and `Post_Suffix` (using the exact counts shown) allows controlled replacement of the Y-slot while keeping the remainder identical in both assembled binaries.
- Concatenating as shown builds final executables where one binary has X seeded with P and the other with Q, but both use the identical suffix pieces — producing identical MD5 digests but different in-memory array contents and therefore different runtime behaviour.
- Running `./Task4_code1` and `./Task4_code2` demonstrates the behavioral difference; verifying MD5s with `md5sum` confirms the hashes match.

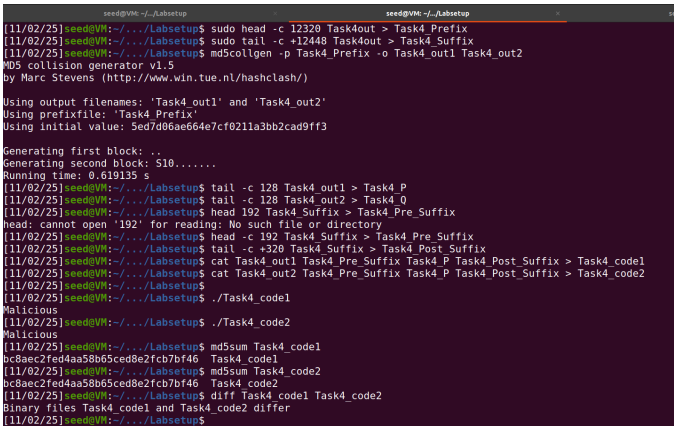

Fig. 4: Task 4 — assembly and verification (same MD5, different runtime output).

VII. Selected code snippets (decision logic)

Below is the critical C decision logic used in the program (the screenshot shows the program printing `Malicious` / `Benign` depending on the comparison):
This snippet is the runtime decision point — by controlling the bytes of `x` and `y` in the binary via the P/Q injection, the branch taken at runtime flips while MD5 stays the same.

VIII. Findings and interesting observations

- The screenshots verify that `md5collgen` produces differing blocks (P vs Q) and that appending the same suffix preserves identical MD5 results (Task 2).
- The precise `head`/`tail` byte counts and the concatenation order are crucial — even one-byte misalignment breaks the collision or the intended X/Y relationship.
- The `diff` of the final binaries shows they differ as expected, while `md5sum` confirms identical hashes — the practical demonstration of why MD5 is unsuitable for integrity checks.

IX. Conclusion

The supplied screenshots capture the essential commands and outputs for the MD5 collision exercises. Using only the commands visible in the screenshots, the report demonstrates: generation of colliding blocks, preservation of collision by appending identical suffixes, and assembly of two executables which share an MD5 but behave differently at runtime. These results illustrate the theoretical MD5 weaknesses in a reproducible, concretely documented workflow.