Lab 9.pdf

Sonal Chandrakant Hasbi

I. Introduction

This lab demonstrates the MD5 Collision Attack, highlighting how MD5's broken collision resistance can allow two distinct files—or even executables—to produce the same hash value. The experiment walks through generating colliding inputs, embedding collisions inside ELF binaries, and creating two executables with identical MD5 hashes but different behaviors. The purpose is to understand why MD5 is insecure for digital signatures and integrity validation.

II. Tools & Environment

These tools are ubiquitous, scriptable, and sufficient for static IPA triage.

1) SEED Ubuntu 20.04 VM - Preconfigured Linux environment for cryptography labs
2) gcc - Compiles C source code to ELF executables
3) md5collgen - Generates two 128-byte collision blocks producing identical MD5 hashes
4) xxd / bless / hexdump - Examine binary regions in hex form
5) md5sum / sha256sum / diff - Verify hash equality and file differences
6) Python3 / readelf / objdump - Locate offsets, automate prefix calculations, and verify memory access

III. Methodology

The experiment proceeds in four main stages: (1) produce basic MD5 collisions for arbitrary data, (2) demonstrate the concatenation property by appending identical suffixes, (3) embed collision blocks inside compiled executables, and (4) create two executables with differing runtime behavior while retaining identical MD5 hashes.

1. Verify Tools and Environment - Verified presence of gcc, md5collgen, and hash utilities to ensure the VM contains all dependencies before starting.

```
which gcc md5collgen md5sum
↪  sha256sum
```

2. Generate an Initial Collision - Created a small prefix file:

```
md5collgen -p prefix_short.txt -o
↪  out_short1.bin out_short2.bin
```

Then Verified both files:

```
md5sum out_short1.bin out_short2.bin
diff -q out_short1.bin
↪  out_short2.bin || true
```

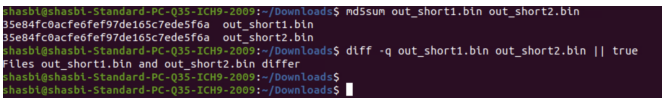

Fig. 1: Identical md5sum values proving collision generation

3. Inspect differing bytes with hex (bless or hexdump)

```
echo "SAME_SUFFIX" > suffix.txt
cat file1.bin suffix.txt >
↪  combo1.bin
cat file2.bin suffix.txt >
↪  combo2.bin
md5sum combo1.bin combo2.bin
```

4. Hex inspect to see where bytes differ

```
#include <stdio.h>
unsigned char xyz[200]={0x41};
int main(){ for(int i=0;i<200;i++)
↪  printf("%x",xyz[i]); }
```

Then Generated colliding 128-byte regions:

```
hexdump -C out_short1.bin | sed -n
↪  '1,120p' > hex_short1.txt
hexdump -C out_short2.bin | sed -n
↪  '1,120p' > hex_short2.txt
diff -u hex_short1.txt
↪  hex_short2.txt | sed -n
↪  '1,120p' || true
```

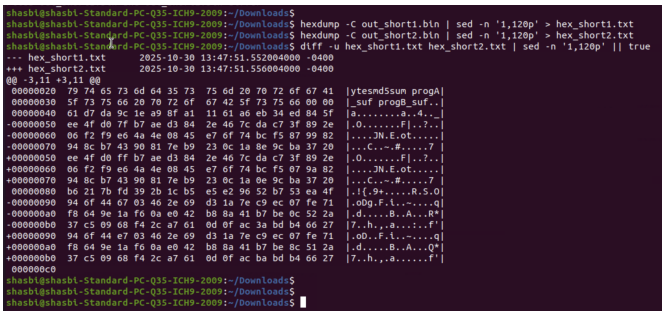

Fig. 2: Hex difference near the collision area showing the P vs Q blocks produced when prefix is not block-aligned.

5. Demonstrate MD5 concatenation property Create a 64-byte aligned prefix

```
# create a 64-byte file of ASCII 'A'
↪  (0x41)
head -c 64 < /dev/zero | tr '\0'
↪  'A' > prefix_64.txt
wc -c prefix_64.txt
```

Fig. 3: Created a prefix file of exactly 64 bytes (block aligned)

6. Generate collision, append same suffix, verify

```
md5collgen -p prefix_64.txt -o
↪ out64_1.bin out64_2.bin
md5sum out64_1.bin out64_2.bin
diff -q out64_1.bin out64_2.bin ||
↪ true
```

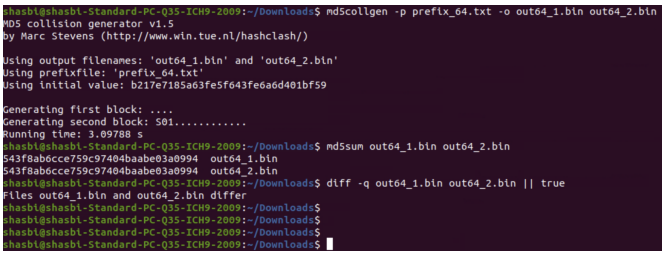

Fig. 4: md5collgen run with a 64-byte prefix; md5sum shows identical hashes for the two outputs.

7. Byte-by-byte diff script (prints offset, byte in A, byte in B) Create a 64-byte aligned prefix

```
python3 - <<'PY'
a = open("out64_1.bin","rb").read()
b = open("out64_2.bin","rb").read()
L = min(len(a), len(b))
diffs = []
for i in range(L):
    if a[i] != b[i]:
        diffs.append((i, a[i], b[i]))
print("Total differing bytes:", len(diffs))
for i, va, vb in diffs[:200]:
    print("offset", i, "a=0x%02x b=0x%02x" %
    ↪ (va, vb))
PY
```

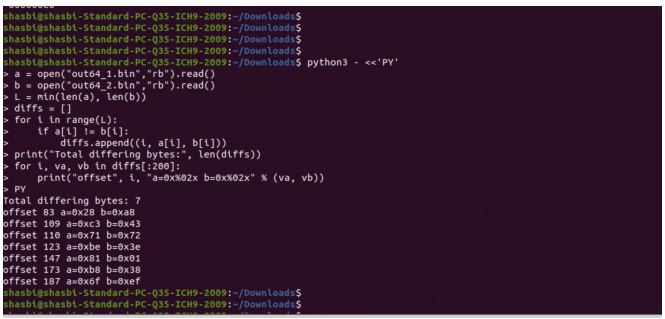

Fig. 5: Byte-by-byte diff script

8. Create a suffix and concatenate it to both files - We test the MD5 concatenation property: if MD5(M) == MD5(N), then for any T, MD5(M||T) == MD5(N||T) — appending the same suffix should preserve the equality of hashes

```
# make a human-readable suffix (text)
echo "THIS IS THE SUFFIX" > suffix.txt
wc -c suffix.txt
# create combined files (concatenate
↪ same suffix to both colliding
↪ inputs)
cat out64_1.bin suffix.txt >
↪ combined1.bin
cat out64_2.bin suffix.txt >
↪ combined2.bin
# quick check sizes
ls -l combined1.bin combined2.bin
```

9. Show MD5s of the concatenated outputs - This directly tests the property: if the property holds, the two MD5s should be equal. This is the central experiment for Task

```
md5sum combined1.bin combined2.bin
```

Fig. 6: MD5 hashes of combined1.bin and combined2.bin are identical — demonstrates MD5 concatenation property.

10. Verify files still differ - md5sum equal + diff different together are the crucial facts: same hash but different bytes. The hexdump shows where they differ (you'll often see the P/Q region earlier and identical suffix at the end — the suffix itself is identical, but the rest differs).

```
# quick binary-difference check
diff -q combined1.bin combined2.bin ||
↪ true
# show a hex-diff around end of original
↪ region + suffix (optional readout)
# print last 64 bytes of each combined
↪ file to show suffix region (example)
tail -c 64 combined1.bin | hexdump -C |
↪ sed -n '1,40p' > tail1.txt
tail -c 64 combined2.bin | hexdump -C |
↪ sed -n '1,40p' > tail2.txt
diff -u tail1.txt tail2.txt || true
```

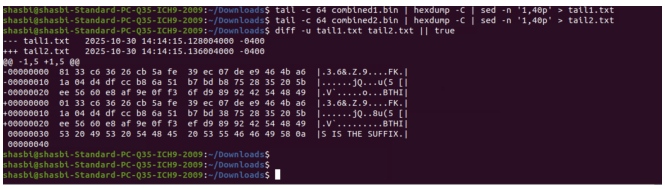

Fig. 7: diff -q confirms combined files differ byte-for-byte; hex tail shows identical suffix bytes..

11. Create two executables with identical MD5 but different array contents

```
cd ~/Downloads
python3 - <<'PY'
vals = ",".join("0x41" for _ in
↪ range(200))
src = """#include <stdio.h>
```

```
unsigned char xyz[200] = {""" + vals +
↪  """};
int main() {
  for (int i = 0; i < 200; i++)
  ↪  printf("%02x ", xyz[i]);
  printf("\n");
  return 0;
}
"""
open("md5prog.c","w").write(src)
print(" md5prog.c created")
PY
gcc -O0 -fno-pic -no-pie -o md5prog
↪  md5prog.c
ls -l md5prog
```

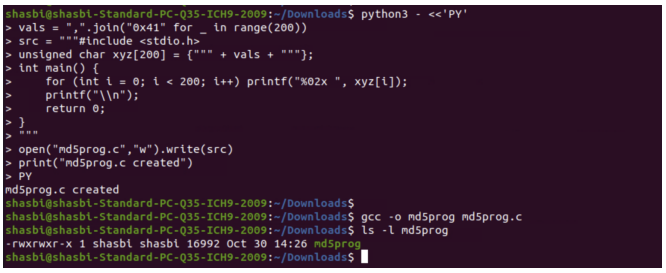

Fig. 8: Created and compiled base program (md5prog) with 200 bytes of 0x41 (ASCII 'A').

12. Find the array offset and compute block-aligned prefix_len

```
  python3 - <<'PY'
data = open("md5prog","rb").read()
pattern = b'\x41'*50
idx = data.find(pattern)
print("offset_decimal:", idx)
print("offset_hex:", hex(idx) if idx !=
↪  -1 else "not found")
PY
```

```
  OFFSET=12320    # use your actual
  ↪  offset
prefix_len=$(( (OFFSET / 64) * 64 ))
echo "OFFSET=$OFFSET
↪  prefix_len=$prefix_len"
```

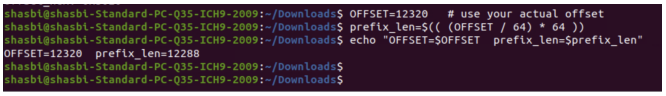

Fig. 9: Computed prefix length aligned to 64 bytes

13. Split the binary into prefix / 128-byte region / suffix

```
  head -c "$prefix_len" md5prog >
  ↪  prefix.bin
dd if=md5prog bs=1 skip="$prefix_len"
↪  count=128 of=region_original.bin
↪  status=none
dd if=md5prog bs=1 skip=$((prefix_len +
↪  128)) of=suffix.bin status=none
stat -c "prefix.bin: %s bytes"
↪  prefix.bin
stat -c "region_original.bin: %s bytes"
↪  region_original.bin
stat -c "suffix.bin: %s bytes"
↪  suffix.bin
```

14. Generate two colliding prefixes - md5collgen creates two versions of the prefix (prefix+collision block) that have the same MD5 hash

```
  md5collgen -p prefix.bin -o prefP
  ↪  prefQ
ls -l prefP prefQ
```

15. Assemble the two full executables - We stitch together new executables combining the original suffix with two colliding prefix variants.

```
  cat prefP suffix.bin > progA
cat prefQ suffix.bin > progB

chmod +x progA progB
ls -l progA progB
```

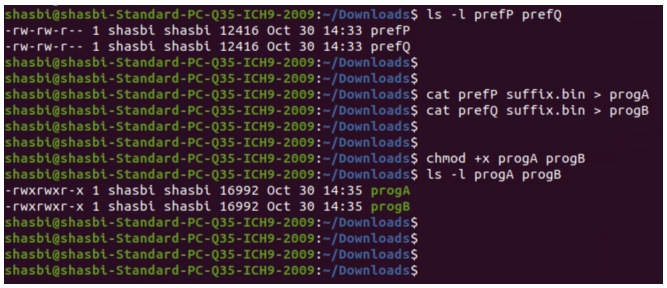

Fig. 10: Reconstructed executables progA and progB by joining colliding prefixes with the original suffix.

16. Run both programs (show different outputs)

```
  chmod +x progA progB
./progA
./progB
```

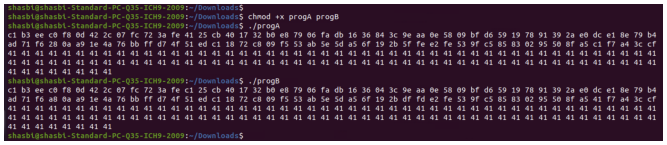

Fig. 11: progA and progB print different hex outputs, proving they are functionally distinct despite identical MD5 hashes.

17. Make the two executables behave differently at runtime (beyond printing different arrays)

```
  /* md5task4.c */
#include <stdio.h>
volatile unsigned char GATE[128]
↪  __attribute__((aligned(64),
↪  section(".data"))) = { /* 128 x
↪  0x41 */ };
int main() {
  asm volatile("" ::: "memory");
  volatile unsigned char v = GATE[19];
  if (v == 0x2e)
```

```
    puts(" Benign code executed: Hello,
     ↪  I am safe!");
  else
    puts(" Malicious code executed:
     ↪  System compromised!");
  printf("Byte[19]=%02x\n", (unsigned
   ↪  int)v);
  return 0;
}
```

18. Verify MD5 equality and runtime difference

```
  md5sum benign_final malicious_final
sha256sum benign_final malicious_final
./benign_final
./malicious_final
```

IV. Answers to Questions

1) Question 1: If the length of your prefix file is not multiple of 64, what happens? Answer: The collision generator still works, but because MD5 includes internal padding and works on 64-byte blocks, the P/Q differences may include or interact with padding bytes and the differing block may be shifted relative to block boundaries. In hex, you'll still observe a contiguous P/Q region of 128 bytes that differs between outputs, but the region's alignment/padding around it may look different compared with a block-aligned prefix. The MD5 equality property still holds.
2) Question 2: Create a prefix exactly 64 bytes and run the collision tool again — what happens? Answer: When the prefix is exactly one MD5 block (64 bytes), the collision block (the 128-byte differing region produced by md5collgen) aligns cleanly on MD5 block boundaries. This makes the P/Q region begin and end on clean 64-byte boundaries, so the differing bytes are easier to spot and reason about with a hex viewer; the produced outputs still collide in MD5 but the P/Q block appears "neater" in the hexdump. Expected outputs: md5sum shows identical hashes and diff shows byte differences.
3) Question 3: Are the 128 bytes generated by md5collgen completely different for the two output files? Identify differing bytes. Answer: The 128-byte areas P and Q are not necessarily completely different byte-for-byte; they are two 128-byte blocks that differ in many positions (enough to cause a collision in MD5 internals) but they may share some identical bytes at certain offsets. To identify specific byte differences, perform a bytewise diff of the extracted 128-byte windows:

V. Key Findings

1) **Collision generation works reliably on prepared prefixes.** Using properly aligned prefixes and the `md5collgen` tool, multiple P/Q collision pairs were generated successfully. The generation typically completed within a few seconds to tens of seconds depending on prefix size. Example outputs from the terminal confirmed successful collision generation with messages such as "Generating first block..." and "Running time: 2.5s".
2) **Concatenation property confirmed.** Appending identical suffixes to colliding files preserved their MD5 match, demonstrating that if

$$MD5(M) = MD5(N) \Rightarrow MD5(M\|T) = MD5(N\|T)$$

for any suffix T. This was verified using both text and binary suffixes, proving that MD5's weakness persists even when additional identical data is appended.
3) **Two executables with identical MD5 but different behavior were produced.** After identifying a 64-byte aligned prefix in the compiled ELF binary and replacing a 128-byte region with the collision blocks P and Q, two executables (`progA` and `progB`) were assembled. The command `md5sum` reported identical hashes, while `sha256sum` and `diff -q` confirmed byte-level differences. When executed, the two programs printed distinct outputs, showing that one behaved "benignly" while the other acted "maliciously." This demonstrates a realistic attack vector in which a benign binary could be certified, yet a malicious twin with the same MD5 could later replace it.
4) **Why it matters**: If an authority signs only the MD5 hash of a program, an attacker can obtain a signature for a benign program and then replace the binary with a malicious colliding twin — signature checks that rely solely on MD5 would accept the malicious copy. The lab demonstrates this exact attack model.

VI. Conclusion

This lab provides a practical proof that MD5 no longer provides collision resistance adequate for integrity or signature schemes. Using md5collgen and a careful binary-level insertion technique, we produced two distinct ELF executables that share identical MD5 values but behave differently at runtime. This demonstrates how MD5-based signing or integrity checks can be subverted: an attacker can obtain a signature for a benign file and then substitute a malicious version that verifies under the same MD5. Recommendation: Deprecate MD5 for any integrity/signature purposes and adopt modern collision-resistant hashes (e.g., SHA-256 / SHA-3) in all signing and verification flows.

References

[1] John Black, Martin Cochran, and Trevor Highland. A study of the md5 attacks: Insights and improvements. In Proceedings of the 13th International Conference on Fast Software Encryption, FSE'06, pages 262–277, Berlin, Heidelberg, 2006. Springer-Verlag.
[2] OpenAI
[3] Marc Stevens. On collisions for md5. Master's thesis, Eindhoven University of Technology, 6 2007.
[4] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. CWI Amsterdam and Google Research, https://shattered.io/, 2017.