

Workshop 3 – Cryptographic Hash Functions and Message Authentication Code

Task 1: Review Important Concepts

1. Explain the following concepts: 1) collision resistance, 2) second preimage resistance, and 3) preimage resistance.

Answer:

- 1) collision resistance

Collision resistance is a generalized case of second preimage resistance and preimage resistance, that is it should be computationally infeasible to find two random preimages that hash to the same image, or find two random messages that have the same hash value.

This is an important concept because hash values are fixed in size, so there bound to be duplicates of messages or preimages in set of infinite possibilities that hash to the same value. For this reason, a very good hashing algorithm is required. This is necessary to make passwords computationally secure against brute force attacks.

- 2) second preimage resistance

Second preimage resistance is a feature where two (different) preimages do not result in the same images being produced after being processed as an input argument to the hash function.

Having the same hash value being generated from two different preimages is not reliable to verify the data integrity of the original input. Second preimage resistance makes it difficult to locate the input of the second distinct (ie. Finding x_2 from y_1) that has the same output as the given input if:

$$\begin{aligned}y_2 &= H(x_2) \\ y_1 &= H(x_1)\end{aligned}$$

And

$$y_2 = y_1$$

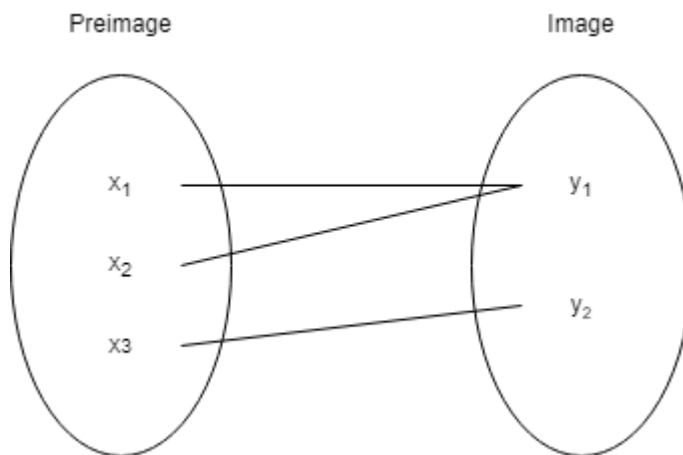
In conclusion Second Preimage is being able to workout another message, message 2, that hashes to hash 1 that was derived from message 1

3) preimage resistance

Preimage resistance is the non-inversibility of a cryptographic hash function that it is impossible to obtain any of the possible preimage data that corresponds to an image data.

$$y = H(x)$$

A preimage is the input x shown in the equation above where it can only have once corresponding image, being y . Therefore, an image can have several corresponding preimages.

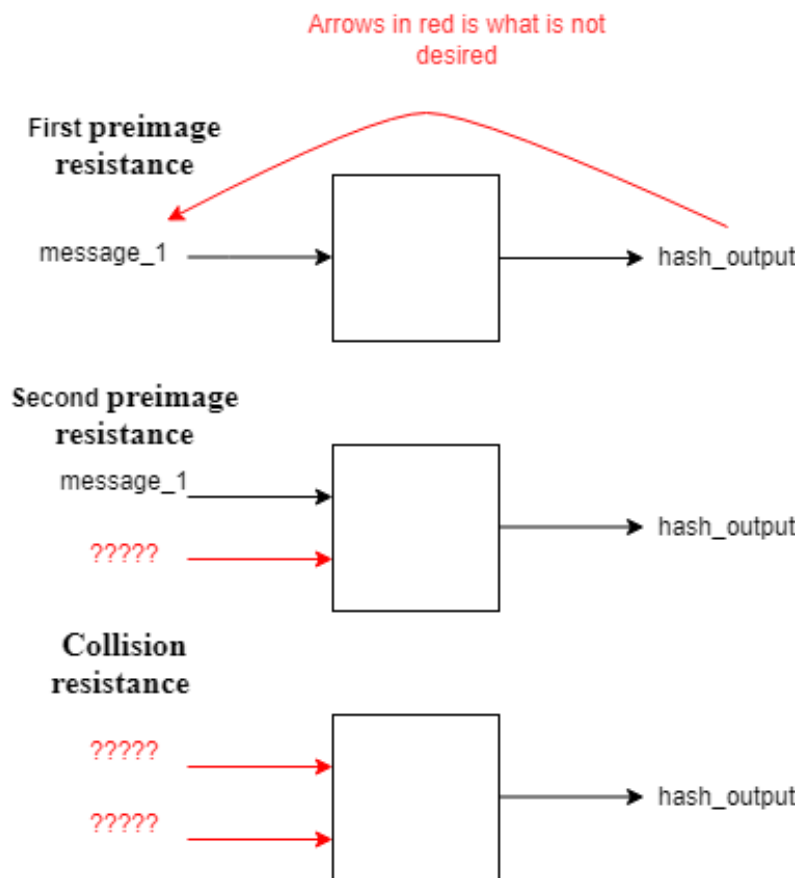


Preimage resistance is where the cryptographic hash function cannot be inverted like linear functions to retrieve any of the possible preimages.

In other words, if $f(x) = H(x)$ is the hash function where x is a preimage and y is one of the possible images. Preimage resistance ensures that $f(x)$ cannot be inverted like a linear equation (such as $f^{-1}(x)$), where an inverse function can be used to obtain a preimage.

In conclusion First Preimage is being able to workout the original message from the hash

Summary:



2. Explain what is unforgeability under a chosen-message attack.

Answer:

Unforgeability is a basic security property of the Message Authentication Code (MAC) algorithm.

$$T = \text{MAC}(K, M)$$

If one does not know the shared (symmetric) key, and manages to intercept a message, the attacker will not be able to compute the tag (T). Therefore, producing an input that collides with the original tag for any message (M) is not possible.

Unforgeability is a feature that makes it practically impossible to forge a tag (T) based on a message (acquired via unauthorized interception) without knowing the shared key(K). This is an essential authentication feature that prevents attackers from masquerading as valid senders.

In theory the attacker can get an idea of the Key after requesting numerous messages and tags from a server, unforgeability ensures that even if such passive attacks were conducted, the attacker should not be able to forge a valid tag regardless of the number of messages and tag pairs the attacker has analyzed.

Unforgeability is a (sender) authentication and (message) integrity design attribute, because if the attacker does not know the key, the attacker can never compute a valid tag in any way as it will conflict with the real tag that the valid user computes (in the case of a man in the middle attack).

3. Assume party A is communicating with party B. They have shared a secret key K for a MAC algorithm. Now, B receives a pair of a message and a tag (T, M). It checks that indeed $T = \text{MAC}(K, M)$. What does this check tell B?

Answer:

In this scenario A sends the client or peer the message and its associated tag {M,T}.

The receiver B, computes a tag using the message and tag pair received, we will refer to as T'

$$T' = \text{MAC}(M_{\text{from_A}}, T_{\text{from_A}})$$

If the tag computed (T') by receiver B on the received message ($M_{\text{from_A}}$) is equal to the tag received from A ($T_{\text{from_A}}$). The receiver then knows the right message has been sent from the right sender.

The reason for this certainty of authentication and data integrity is due to the fact that if the data was altered T' would not be equal to $T_{\text{from_A}}$, also it is not possible receive the right tag from the sender ($T_{\text{from_A}}$) unless the sender has a valid key that both A and B share.

4. Explain which security property of cryptographic hash functions makes them a good choice of protecting passwords.

Answer:

Password is stored as hash rather than clear text, so this means even if the database was hacked by an attacker, they cannot obtain a plaintext password to enter the user's account.

A salted hash is much more secure than a hashing only algorithm. If an attacker obtained all the hashes that correspond to user passwords and performed an offline dictionary attack to generate hashes that match will the hashes obtained from the database (or stolen via black hat hacking). These passwords derived from the dictionary attack will not work on the victim's account as the attacker does not know the salt value used in hashing.

The fact that hash functions cannot be inverted to an inverse function to retrieve the password in plain text is what makes cryptographic hash functions as a good choice for a one way encryption. In cryptography this is referred to as one way encryption

5. What are the differences between passwords and (cryptographic) keys?

Answer:

Passwords are not as random as keys, they are smaller in size compared to keys. Passwords are more memorable to humans therefore almost always shorter.

The possible combinations of passwords are usually a lot more predictable than keys especially given context, for example 4-digit PIN, which is much weaker, 10,000 combinations compared to 2^{128} combination for a 128-bit key.

Both are used for different purposes, passwords are normally used for authentication while keys are used for encryptions or digital signatures.

In last weeks laboratory using random passwords that are long enough as seeds to create a key rather than using a key as an alternate option was also discussed.

Task 2: Unix/Linux user password protection

One of the most important security uses of cryptographic hash functions is password protection. In this part create a user in Ubuntu system and see how user information and passwords are stored.

1. We create a user using the following command (you can pick any username as you want). You can use the command `useradd` as you (user: seed) has a sudo privilege.

```
$ sudo useradd -c "My new user account" hashlab
```

Here the option `-c` allows you to give a description of your account (e.g., "My new user account"). Now, a new user account has been added to the system. This information is stored in `/etc/passwd`.

```
$ sudo cat /etc/passwd #display the content of /etc/passwd
```

or

```
$ sudo grep hashlab /etc/passwd #dispaly the entry for hashlab
```

The file name "passwd" may look confusing to you as you may think it should store password information, which *was* true. Originally, the file "passwd" in Unix systems was used to store password information. But now, the password information is stored in another file called shadow file.

Answer:

When a new account is created using useradd, initially there is no password:

```
yasin@yasin-Satellite-L50-A:~$ sudo useradd -c "My new user account" hashlab
[sudo] password for yasin:
yasin@yasin-Satellite-L50-A:~$ sudo cat /etc/passwd #display the content of /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106:/:/nonexistent:/usr/sbin/nologin
syslog:x:104:110:/:/home/syslog:/usr/sbin/nologin
_apt:x:105:65534:/:/nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:114:/:/run/uuidd:/usr/sbin/nologin
tcpdump:x:108:115:/:/nonexistent:/usr/sbin/nologin
avahi-autoipd:x:109:116:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
usbmux:x:110:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
rtkit:x:111:117:RealtimeKit,,,:/proc:/usr/sbin/nologin
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
cups-pk-helper:x:113:120:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
speech-dispatcher:x:114:29:Speech Dispatcher,,,:/run/speech-dispatcher:/bin/false
avahi:x:115:121:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/usr/sbin/nologin
saned:x:117:123:/:/var/lib/saned:/usr/sbin/nologin
nm-openvpn:x:118:124:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
hplip:x:119:7:HPLIP system user,,,:/run/hplip:/bin/false
whoopsie:x:120:125:/:/nonexistent:/bin/false
colord:x:121:126:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
geoclue:x:122:127:/:/var/lib/geoclue:/usr/sbin/nologin
pulse:x:123:128:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
gnome-initial-setup:x:124:65534:/:/run/gnome-initial-setup:/bin/false
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
sssd:x:126:131:SSSD system user,,,:/var/lib/sss:/usr/sbin/nologin
yasin:x:1000:1000:Yasin,,,:/home/yasin:/bin/bash
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
nvidia-persistenced:x:127:134:NVIDIA Persistence Daemon,,,:/nonexistent:/usr/sbin/nologin
hashlab:x:1001:1001:My new user account:/home/hashlab:/bin/sh
yasin@yasin-Satellite-L50-A:~$
```

2. To enable log in as the new user, a user password is needed. We can supply the password for the user hashlab via the following command:

```
$ sudo passwd hashlab
```

This passwd information is stored in the shadow file /etc/shadow.

```
$ sudo cat /etc/shadow or $ sudo grep hashlab /etc/shadow
```

You can see an entry like this under the user “hashlab”.

```
hashlab:$6$8MTIu8hIzg/pJm7Q$0XScsQClhavzhXb3aBxALdyELJFtlzn8PqLBg.W80Z2N5B7ICDn9bS3AnBQ  
rRgfs5Uozs6ng923nbCMNTWbj1:19055:0:99999:7:::
```

The values followed username “hashlab” are separated by \$ sign. The first part number 6 specifies that the hash function SHA-512 is used. The second part is the “salt”, a random string chosen by the system. The third part, which is 512-bit long, is the hash of the password. If we use “h” to denote the hash value, and “pass” to denote the password of the user, then we have $h = \text{SHA-512}(\text{salt}||\text{pass})$ where “||” is the string concatenation operation.

Now, we can clearly see that the user passwords are never stored in clear in the system. Even if the attacker captures the file /etc/shadow, she still needs to break the oneway property of the hash function, which is hard. Here dictionary attack is not possible anymore because of the random salt.

Answer

Following from part 1 a password is issued for the new account “hashlab”. The password issued was “abc123”

```
yasin@yasin-Satellite-L50-A: ~  
New password:  
Retype new password:  
passwd: password updated successfully  
yasin@yasin-Satellite-L50-A:~$ sudo cat /etc/shadow or $ sudo grep hashlab /etc/shadow  
[sudo] password for yasin:  
Sorry, try again.  
[sudo] password for yasin:  
root:!:19076:0:99999:7:::  
daemon*:19046:0:99999:7:::  
bin*:19046:0:99999:7:::  
sys*:19046:0:99999:7:::  
sync*:19046:0:99999:7:::  
games*:19046:0:99999:7:::  
man*:19046:0:99999:7:::  
lp*:19046:0:99999:7:::  
mail*:19046:0:99999:7:::  
news*:19046:0:99999:7:::  
uucp*:19046:0:99999:7:::  
proxy*:19046:0:99999:7:::  
www-data*:19046:0:99999:7:::  
backup*:19046:0:99999:7:::  
list*:19046:0:99999:7:::  
irc*:19046:0:99999:7:::  
gnats*:19046:0:99999:7:::  
nobody*:19046:0:99999:7:::  
systemd-network*:19046:0:99999:7:::  
systemd-resolve*:19046:0:99999:7:::  
systemd-timesync*:19046:0:99999:7:::  
messagebus*:19046:0:99999:7:::  
syslog*:19046:0:99999:7:::  
_apt*:19046:0:99999:7:::  
tss*:19046:0:99999:7:::  
uuidd*:19046:0:99999:7:::  
tcpdump*:19046:0:99999:7:::  
avahi-autoipd*:19046:0:99999:7:::  
usbmux*:19046:0:99999:7:::  
rtkit*:19046:0:99999:7:::  
dnsmasq*:19046:0:99999:7:::  
cups-pk-helper*:19046:0:99999:7:::  
speech-dispatcher:!:19046:0:99999:7:::  
avahi*:19046:0:99999:7:::  
kernoops*:19046:0:99999:7:::  
saned*:19046:0:99999:7:::  
nm-openvpn*:19046:0:99999:7:::  
hplip*:19046:0:99999:7:::  
whoopsie*:19046:0:99999:7:::  
colord*:19046:0:99999:7:::  
geoclue*:19046:0:99999:7:::  
pulse*:19046:0:99999:7:::  
gnome-initial-setup*:19046:0:99999:7:::  
gdm*:19046:0:99999:7:::  
sssd*:19046:0:99999:7:::  
yasin:$6$LyqeKwY/PrAJ23Q$CQ/LhA/Va6moGhwg5uZGzCxjYkBrKcr2WfK5YwR7A4FFtUcUEUBrj44k8jxWz5YSpZLxLY4wah6l/r.hKW1:19076:0:99999:7:::  
systemd-coredump:!:19077:7:::  
nvidia-persistenced*:19076:0:99999:7:::  
hashlab:$6$DREw03AVsFedvFsb$UhmGGU/kfbl10Uf0McVawLkgN1iNe54ay4NWaMacWUp1C.6VNM5Px/dgaaiPZSgrLM2fLGWJ/usTuFzvNRjL01:19093:0:99999:7:::  
cat: or: No such file or directory
```

- Of course, the attacker may still try to brute force users' passwords from the shadow file. In fact, this is a common way of privilege escalation after penetrating the system. There are several password cracking tools available and here we use the one called "john" or "john-the-ripper". We can supply the shadow file directly to john which will make john to try to crack passwords for all users. Here we can apply john to the new user.

```
$ sudo grep hashlab /etc/shadow > password.txt  
$ sudo john password.txt
```

The first command extracts the information of hash of hashlab's password and save it to a text file. The second command runs john to try to find the password.

In your experiment report submission, you should choose two passwords: a weak one (e.g., 12345 or what or something that is obviously not a good password) and a fairly strong password (e.g., a long password contains numbers, uppercase and lowercase letters, as well as several special characters). Then create two users with these passwords and try to break them using john and show your result. Note, john runs the basic brute-force attack in this case

(dictionary attack doesn't work). So, for strong passwords john could keep running forever. So, stop john when no results after at most 10 minutes.

Important: Do not attack other users' passwords unless you have permission to do so. The experiment is for education purpose.

Answer:

The password for username: hashlab was broken in 16 seconds

```
yasin@yasin-Satellite-L50-A: ~/Documents/3809ICT Lab3 Tasks/Task2/3
Get:1 http://au.archive.ubuntu.com/ubuntu focal/main amd64 john-data all 1.8.0-2build1 [4,276 kB]
Get:2 http://au.archive.ubuntu.com/ubuntu focal/main amd64 john amd64 1.8.0-2build1 [189 kB]
Fetched 4,466 kB in 36s (125 kB/s)
Selecting previously unselected package john-data.
(Reading database ... 282760 files and directories currently installed.)
Preparing to unpack .../john-data_1.8.0-2build1_all.deb ...
Unpacking john-data (1.8.0-2build1) ...
Selecting previously unselected package john.
Preparing to unpack .../john_1.8.0-2build1_amd64.deb ...
Unpacking john (1.8.0-2build1) ...
Setting up john-data (1.8.0-2build1) ...
Setting up john (1.8.0-2build1) ...
Processing triggers for man-db (2.9.1-1) ...
yasin@yasin-Satellite-L50-A:~$ cd '/home/yasin/Documents/3809ICT Lab3 Tasks/Task2/3'
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3$ sudo grep hashlab /etc/shadow > password.txt
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3$ sudo john password.txt
Created directory: /root/.john
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
abc123      (hashlab)
1g 0:00:00:16 100% 2/3 0.06045g/s 185.0p/s 185.0c/s 185.0C/s 123456..pepper
Use the "--show" option to display all of the cracked passwords reliably
Session completed
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3$
```

Two other users were created as follows:

Case 1:

Username: hashlab_4

Password: Az786_birIKIn9ne

hashlab_3:\$6\$v65LZzjDiCDbbNfK\$ILBV7QZZ9EEUKQgoivUSdjC5ouxnLE1AjaEv6SoT5
xmvw/1A8aCLw4ycb8KERzT/bj9W2anouj5iMk2M26loe/:19093:0:99999:7:::

Case 2:

Username: hashlab_2

Password: John07

hashlab_4:\$6\$aMarmAJJ.fVEL2.\$Cnv1Gfxg/83RjZHP9MsHPnLITmGcmh0ADgsa2Cw7ji
yiagSwuGICsN9iW3C7KHr1WGkVbMIZapbkeRj.Scgss/:19093:0:99999:7:::

The john the ripper application was aborted after a 40 minutes and 58 seconds for the first case.

```
yasin@yasin-Satellite-L50-A: ~/Documents/3809ICT Lab3 Tasks/Task2/3/hashlab_4
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3/hashlab_3$ cd '/home/yasin/Documents/3809ICT Lab3 Tasks/Task2/3/hashlab_4'
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3/hashlab_4$ sudo john password_4.txt
[sudo] password for yasin:
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:40:58 3/3 0g/s 308.1p/s 308.1c/s 308.1C/s 203124..203344
Session aborted
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3/hashlab_4$
```

As for case 2 the process was killed using the control C command after 7 minutes and 30 seconds into the process.

```
Session completed
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3$ cd '/home/yasin/Documents/3809ICT Lab3 Tasks/Task2/3/haslab_2/'
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3/haslab_2$ sudo john password_2.txt
[sudo] password for yasin:
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
q
0g 0:00:01:30 9% 2/3 0g/s 281.8p/s 281.8c/s 281.8C/s Lester1..Pancake1
Session aborted
yasin@yasin-Satellite-L50-A:~/Documents/3809ICT Lab3 Tasks/Task2/3/haslab_2$
```

Another account with a password “John_1997” was created however it took over 10 minutes without a result.

Only the hashlab account was cracked almost instantly, these long times before killing the execution for the remaining cases can be for many reasons. One possibility of this process can be the fact that the operation was conducted on a local machine that is less powerful than professional enterprise equipment. Another factor to add long resolution times could be the fact that salting may have been used to encrypt user passwords.

In either case only the weakest and most predictable password (i.e. “abc123”) was broken (in less than thirty seconds).

If this experiment was to be repeated three computers will be used in parallel with each having a account with different password strengths.

This exercise shows the importance of having a strong seed regardless of the hash function being used.

Task 3: Finding collisions to MD5 Hash

MD5 is a cryptographic hash function that takes as input arbitrarily long inputs into 128-bit hash values (or digests). It was widely used for the practice of network security

and data integrity. Since 2004, a series of research work have demonstrated MD5 is not secure both theoretically and practically. And eventually, MD5's collision resistance property can be broken by normal PC almost instantly. In this task, we use the prefix attack to find two files that have the same MD5 hash value (i.e., the two files form a collision to MD5).

1. We start off by generating a prefix file.

```
$ echo "Hello World" > prefix.txt
```

You can check if the text file has the string as content (e.g., by cat or echo).

Answer:

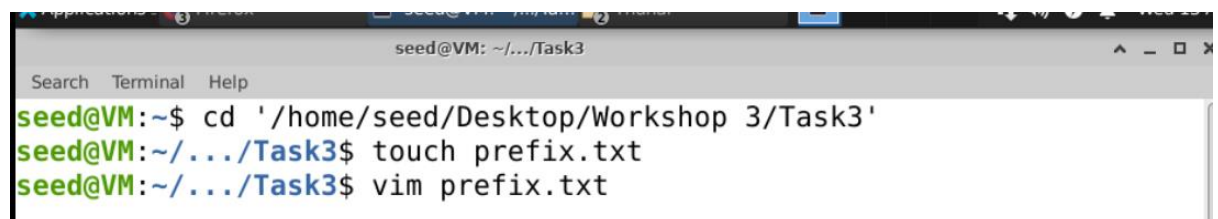
Ideally in general, there should not be a case where there are collisions in secure hash functions. This is necessary for data integrity.

Ideally no data is meant to have the same digest as network clients in client-server or peer-to-peer connections, as end users rely on this digest value as a tag to verify data integrity.

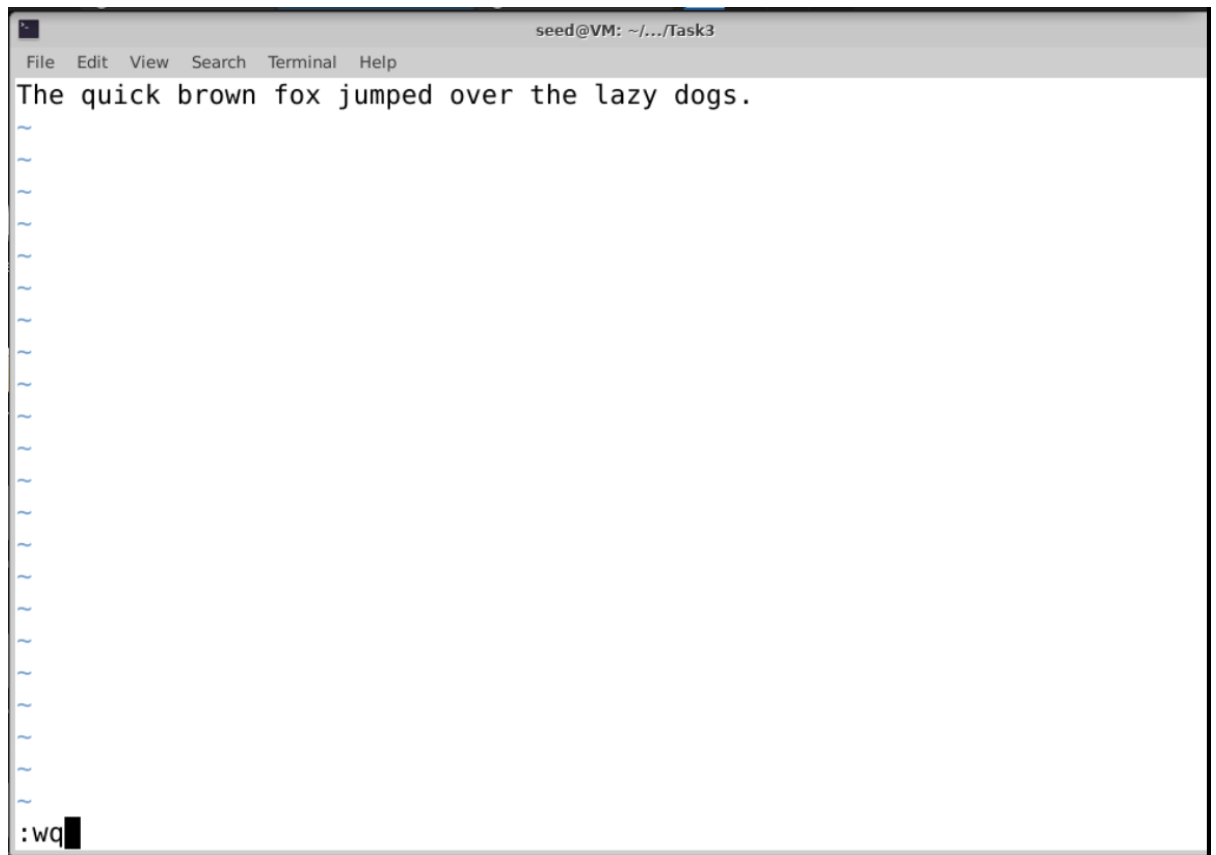
A collision is when a pair of different files that share the same hash value, that is two different preimages have the same hash digest image.

Collision if (x_1, x_2) ; and $H(x_1) = H(x_2)$

This can be seen with out1.bin and out2.bin which have different contents but share the same hash digest. This shows the MD5 hash algorithm is not collision resistant and thus not suitable for network transport applications, where security is an important design attribute.

A screenshot of a terminal window titled 'seed@VM: ~/.../Task3'. The terminal shows three commands being executed: 'cd '/home/seed/Desktop/Workshop 3/Task3'', 'touch prefix.txt', and 'vim prefix.txt'. The prompt 'seed@VM:~\$' is visible at the start of each line.

```
seed@VM:~$ cd '/home/seed/Desktop/Workshop 3/Task3'
seed@VM:~/.../Task3$ touch prefix.txt
seed@VM:~/.../Task3$ vim prefix.txt
```



2. We apply the program MD5collgen to produce two files with the same MD5 hash.

```
$ md5collgen -p prefix.txt -o out1.bin out2.bin
```

Answer:

Using the prefix created in step one, two files are created that share the same hash value both files being out1.bin and out2.bin. Both files are different, yet they hash to the same MD5 digest.

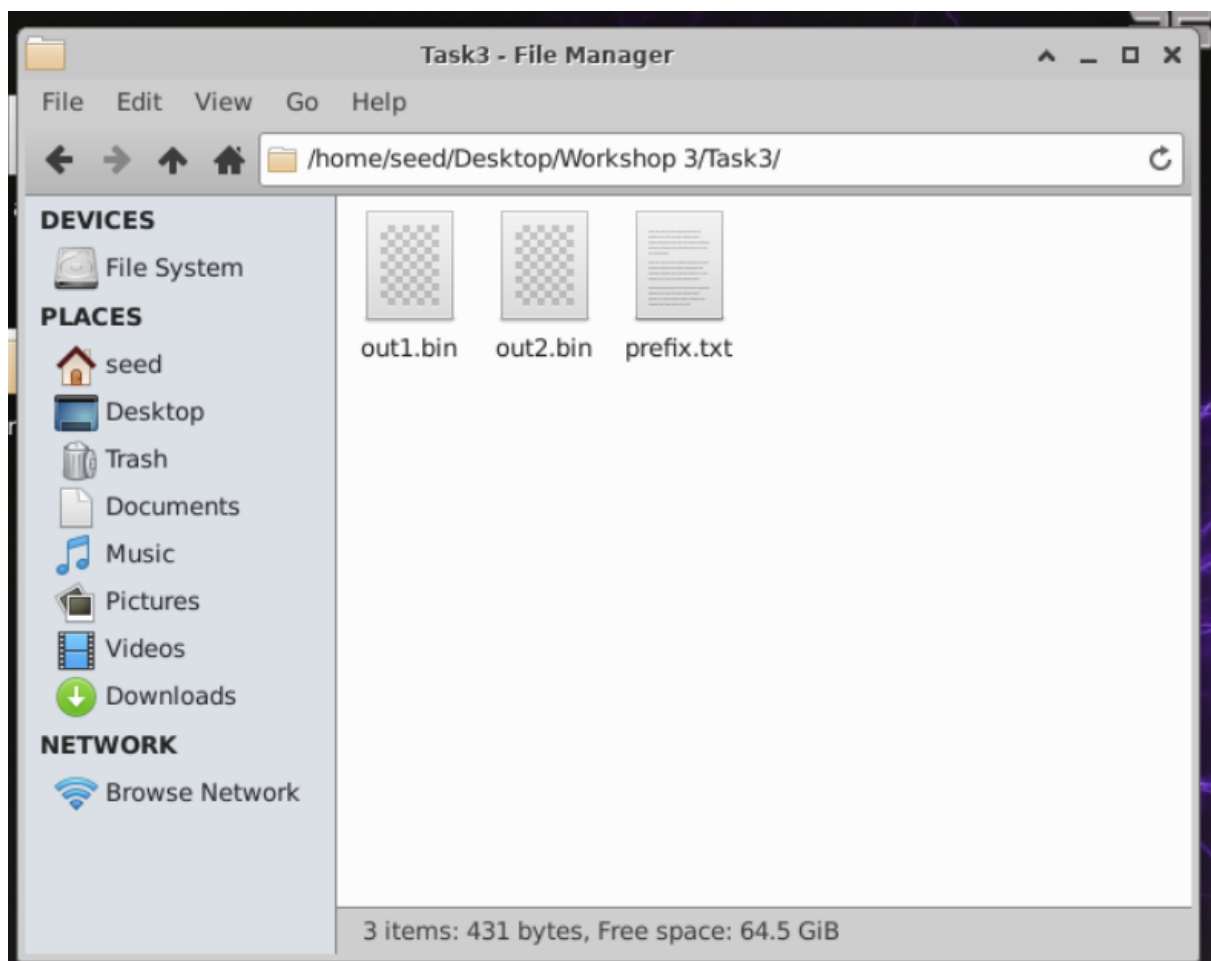
```
seed@VM: ~/.../Task3
File Edit View Search Terminal Help

Exception caught:
too many positional options have been specified on the command line
[04/13/22]seed@VM:~/.../Task3$ md5collgen -p prefix.txt -o out1.bin out2.bin
out3.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Error: exactly two output filenames should be specified.
[04/13/22]seed@VM:~/.../Task3$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: 53e85b2ace6a680917b403edfaa34d58

Generating first block: ..
Generating second block: S10.....
Running time: 4.81566 s
[04/13/22]seed@VM:~/.../Task3$
```



3. Check if the two files are different and have the same MD5 hash value.

```
$ diff out1.bin out2.bin
# Tell if the two files differ or not

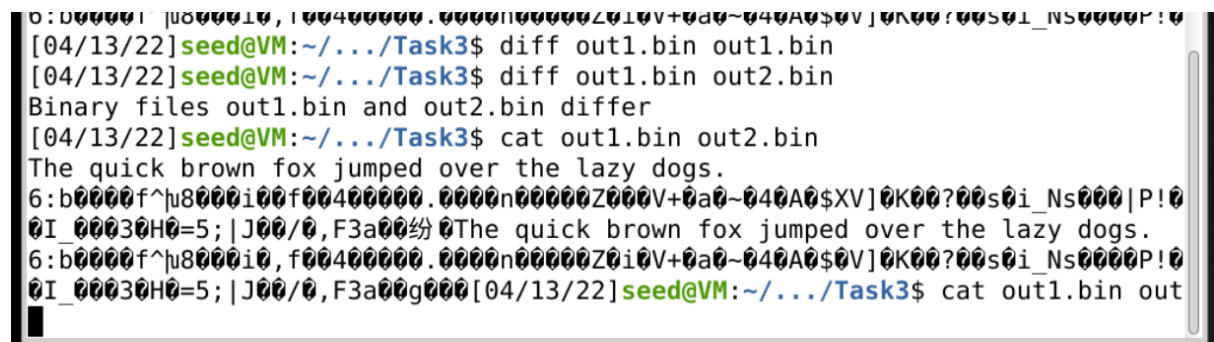
$ md5sum out1.bin out2.bin
```

Here we find the collision specifically for the hash function MD5. The two files we obtained do not form a collision for other cryptographic hash functions. For example:

```
$ sha256sum out1.bin out2.bin
```

Answer:

The “diff command shows a perfect example of where hash collision has occurred on two different files with different contents.

A terminal window screenshot showing a series of commands and their outputs. The user runs 'diff out1.bin out1.bin' which returns no output. Then they run 'diff out1.bin out2.bin' which returns 'Binary files out1.bin and out2.bin differ'. Next, they run 'cat out1.bin out2.bin' which outputs two identical lines of text: 'The quick brown fox jumped over the lazy dogs.' followed by a long, identical MD5 hash string. The terminal text is as follows:

```
[04/13/22] seed@VM:~/.../Task3$ diff out1.bin out1.bin
[04/13/22] seed@VM:~/.../Task3$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[04/13/22] seed@VM:~/.../Task3$ cat out1.bin out2.bin
The quick brown fox jumped over the lazy dogs.
6:b0000f^u8000i00f00400000.0000n00000Z000V+0a0~040A0$XV]0K00?00s0i_Ns000|P!0
0I_00030H0=5;|J00/0,F3a00纷0The quick brown fox jumped over the lazy dogs.
6:b0000f^u8000i0,f00400000.0000n00000Z0i0V+0a0~040A0$0V]0K00?00s0i_Ns0000P!0
0I_00030H0=5;|J00/0,F3a00g000[04/13/22] seed@VM:~/.../Task3$ cat out1.bin out
```

We can see from above the file contents are different, yet they share the same MD5 digest as shown below.

```

seed@VM: ~/.../Task3
6:b0000f^h8000i0,f00400000.0000n00000Z0i0V+0a0~040A0$0V]0K00?00s0i_Ns0000P!0
[04/13/22] seed@VM:~/.../Task3$ diff out1.bin out1.bin
[04/13/22] seed@VM:~/.../Task3$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[04/13/22] seed@VM:~/.../Task3$ cat out1.bin out2.bin
The quick brown fox jumped over the lazy dogs.
6:b0000f^h8000i00f00400000.0000n00000Z000V+0a0~040A0$XV]0K00?00s0i_Ns000|P!0
0I_00030H0=5;|J00/0,F3a00纷0The quick brown fox jumped over the lazy dogs.
6:b0000f^h8000i0,f00400000.0000n00000Z0i0V+0a0~040A0$0V]0K00?00s0i_Ns0000P!0
0I_00030H0=5;|J00/0,F3a00g000[04/13/22] seed@VM:~/.../Task3$ cat out1.bin out
[04/13/22] seed@VM:~/.../Task3$ md5sum out1.bin
b48f8e159a804f4961035bcad9a1a3c6 out1.bin
[04/13/22] seed@VM:~/.../Task3$ md5sum out2.bin
b48f8e159a804f4961035bcad9a1a3c6 out2.bin
[04/13/22] seed@VM:~/.../Task3$ md5sum out1.bin out2.bin
b48f8e159a804f4961035bcad9a1a3c6 out1.bin
b48f8e159a804f4961035bcad9a1a3c6 out2.bin
[04/13/22] seed@VM:~/.../Task3$ sha256sum out1.bin out2.bin
947681e264e78b1204bd9e8988e79e0977a409ce4d477d67543e76a45713adff out1.bin
c793da7e1769f804d344d7e160e034413090044129d3e5b03b7ee43461b45452 out2.bin
[04/13/22] seed@VM:~/.../Task3$

```

This collision clash vulnerability of the MD5 algorithm is shown by executing a sha256 hash function on the same files.

	openssl dgst -sha256 <file>	openssl dgst -md5 <file>
output1.b in	947681e264e78b1204bd9e8988e79e0977a409ce4d477d67543e76a45713adff	b48f8e159a804f4961035bcad9a1a3c6
output1.b in	c793da7e1769f804d344d7e160e034413090044129d3e5b03b7ee43461b45452	b48f8e159a804f4961035bcad9a1a3c6

- Recall that MD5 is constructed using the Merkle-Damgård structure. One property of this structure is that given two inputs X and Y with $X \neq Y$, $MD5(X) = MD5(Y)$, then for any other string Z, $MD5(X||Z) = MD5(Y||Z)$ where “||” denotes the operation of string concatenation. This means that once we have obtained a collision (X, Y), we can create many collisions.

```
$ echo "Mon, Jack did something bad. We should tell Santa to
put him into the "Naughty" list so he can't get presents on
Christmas." > suffix.txt
```

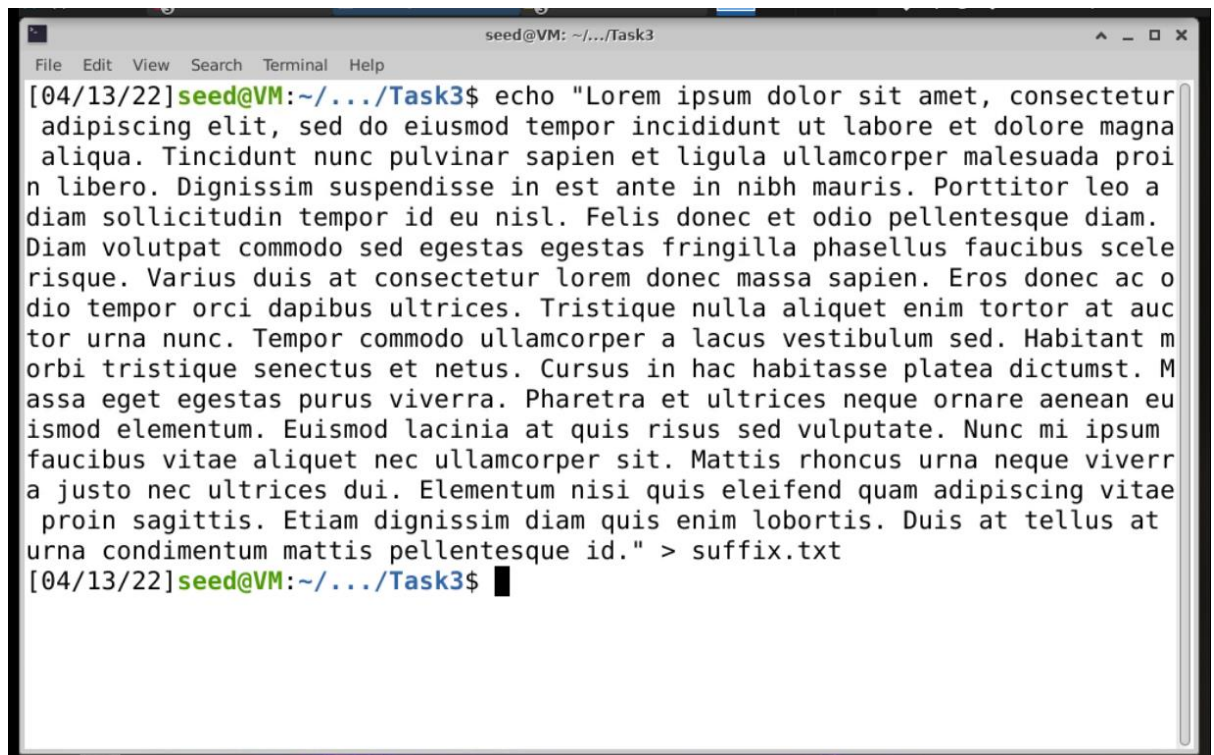
```
$ cat out1.bin suffix.txt > out1_long.bin
$ cat out2.bin suffix.txt > out2_long.bin
```

Obviously, these two files are different. But they have the same MD5 value.

```
$ md5sum out1_long.bin out2_long.bin
```

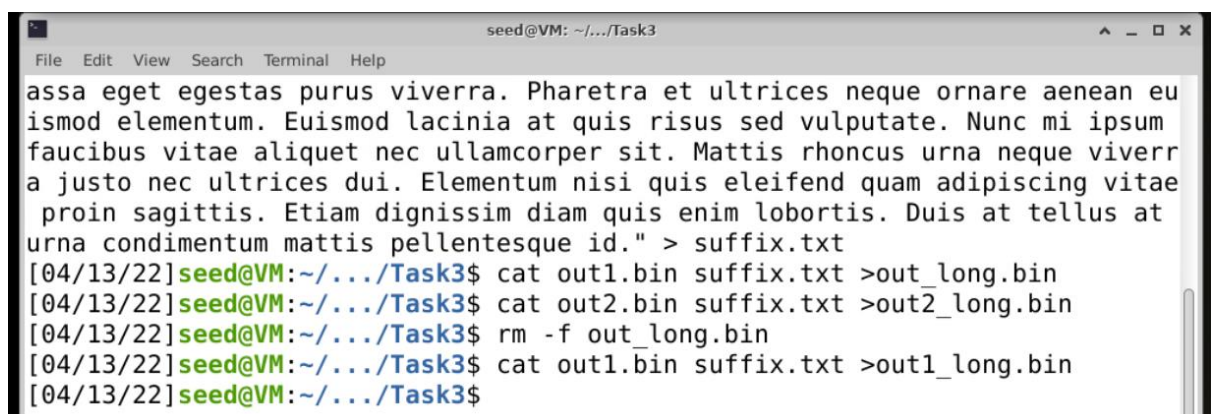

Answer:

An arbitrary suffix text file is created in order to test if the same hash is generated on colliding files even after more data is concatenated to the end of the file.

A terminal window titled 'seed@VM: ~/.../Task3' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is '[04/13/22] seed@VM:~/.../Task3\$'. The user enters a long Lorem Ipsum text string followed by '> suffix.txt'. The prompt changes to '[04/13/22] seed@VM:~/.../Task3\$' with a cursor.

```
[04/13/22] seed@VM:~/.../Task3$ echo "Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Tincidunt nunc pulvinar sapien et ligula ullamcorper malesuada proi
n libero. Dignissim suspendisse in est ante in nibh mauris. Porttitor leo a
diam sollicitudin tempor id eu nisl. Felis donec et odio pellentesque diam.
Diam volutpat commodo sed egestas egestas fringilla phasellus faucibus scele
risque. Varius duis at consectetur lorem donec massa sapien. Eros donec ac o
dio tempor orci dapibus ultrices. Tristique nulla aliquet enim tortor at auc
tor urna nunc. Tempor commodo ullamcorper a lacus vestibulum sed. Habitant m
orbi tristique senectus et netus. cursus in hac habitasse platea dictumst. M
assa eget egestas purus viverra. Pharetra et ultrices neque ornare aenean eu
ismod elementum. Euismod lacinia at quis risus sed vulputate. Nunc mi ipsum
faucibus vitae aliquet nec ullamcorper sit. Mattis rhoncus urna neque viverr
a justo nec ultrices dui. Elementum nisi quis eleifend quam adipiscing vitae
proin sagittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at
urna condimentum mattis pellentesque id." > suffix.txt
[04/13/22] seed@VM:~/.../Task3$
```

The suffix file is concatenated to the end of the bin files created in the previous sections.

A terminal window titled 'seed@VM: ~/.../Task3' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is '[04/13/22] seed@VM:~/.../Task3\$'. The user enters several commands: 'cat out1.bin suffix.txt >out_long.bin', 'cat out2.bin suffix.txt >out2_long.bin', 'rm -f out_long.bin', and 'cat out1.bin suffix.txt >out1_long.bin'. The prompt changes to '[04/13/22] seed@VM:~/.../Task3\$' after each command.

```
assa eget egestas purus viverra. Pharetra et ultrices neque ornare aenean eu
ismod elementum. Euismod lacinia at quis risus sed vulputate. Nunc mi ipsum
faucibus vitae aliquet nec ullamcorper sit. Mattis rhoncus urna neque viverr
a justo nec ultrices dui. Elementum nisi quis eleifend quam adipiscing vitae
proin sagittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at
urna condimentum mattis pellentesque id." > suffix.txt
[04/13/22] seed@VM:~/.../Task3$ cat out1.bin suffix.txt >out_long.bin
[04/13/22] seed@VM:~/.../Task3$ cat out2.bin suffix.txt >out2_long.bin
[04/13/22] seed@VM:~/.../Task3$ rm -f out_long.bin
[04/13/22] seed@VM:~/.../Task3$ cat out1.bin suffix.txt >out1_long.bin
[04/13/22] seed@VM:~/.../Task3$
```

The two files have different contents that would produce a different hash digest or fingerprint in other hash algorithms


```
Applications: Firefox seed@VM: ~/.../Ta... Thunar seed@VM: ~/.../Task3
File Edit View Search Terminal Help
[04/13/22] seed@VM: ~/.../Task3$ cat out1_l
cat: out1_l: No such file or directory
[04/13/22] seed@VM: ~/.../Task3$ cat out1_long.bin
The quick brown fox jumped over the lazy dogs.
6:b0000f^u8000i00f00400000.0000n00000Z000V+0a0~040A0$XV]0K00?00s0i_Ns000|P!0
0I_00030H0=5;|J00/0,F3a00纷0Lorem ipsum dolor sit amet, consectetur adipisci
ng elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Tincidunt nunc pulvinar sapien et ligula ullamcorper malesuada proin libero.
Dignissim suspendisse in est ante in nibh mauris. Porttitor leo a diam soll
icitudin tempor id eu nisl. Felis donec et odio pellentesque diam. Diam volu
tpat commodo sed egestas egestas fringilla phasellus faucibus scelerisque. V
arius duis at consectetur lorem donec massa sapien. Eros donec ac odio tempo
r orci dapibus ultrices. Tristique nulla aliquet enim tortor at auctor urna
nunc. Tempor commodo ullamcorper a lacus vestibulum sed. Habitant morbi tris
tique senectus et netus. Cursus in hac habitasse platea dictumst. Massa eget
egestas purus viverra. Pharetra et ultrices neque ornare aenean euismod ele
mentum. Euismod lacinia at quis risus sed vulputate. Nunc mi ipsum faucibus
vitae aliquet nec ullamcorper sit. Mattis rhoncus urna neque viverra justo n
ec ultrices dui. Elementum nisi quis eleifend quam adipiscing vitae proin sa
gittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at urna cond
imentum mattis pellentesque id.
[04/13/22] seed@VM: ~/.../Task3$
```

```
seed@VM: ~/.../Task3
File Edit View Search Terminal Help
gittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at urna cond
imentum mattis pellentesque id.
[04/13/22] seed@VM: ~/.../Task3$ cat out2_long.bin
The quick brown fox jumped over the lazy dogs.
6:b0000f^u8000i0,f00400000.0000n00000Z0i0V+0a0~040A0$0V]0K00?00s0i_Ns0000P!0
0I_00030H0=5;|J00/0,F3a00g000Lorem ipsum dolor sit amet, consectetur adipisc
ing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Tincidunt nunc pulvinar sapien et ligula ullamcorper malesuada proin libero
. Dignissim suspendisse in est ante in nibh mauris. Porttitor leo a diam sol
licitudin tempor id eu nisl. Felis donec et odio pellentesque diam. Diam vol
utpat commodo sed egestas egestas fringilla phasellus faucibus scelerisque.
Varius duis at consectetur lorem donec massa sapien. Eros donec ac odio temp
or orci dapibus ultrices. Tristique nulla aliquet enim tortor at auctor urna
nunc. Tempor commodo ullamcorper a lacus vestibulum sed. Habitant morbi tri
stique senectus et netus. Cursus in hac habitasse platea dictumst. Massa ege
t egestas purus viverra. Pharetra et ultrices neque ornare aenean euismod el
ementum. Euismod lacinia at quis risus sed vulputate. Nunc mi ipsum faucibus
vitae aliquet nec ullamcorper sit. Mattis rhoncus urna neque viverra justo
nec ultrices dui. Elementum nisi quis eleifend quam adipiscing vitae proin s
agittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at urna con
dimentum mattis pellentesque id.
[04/13/22] seed@VM: ~/.../Task3$
```

However, even after having content appended, in other words, concatenated to the end of the file the hash digest of the new longer files have the same digest. This new digest ,for part 4, is different to the one in part 3 above, being a different file.

```
nec ultrices dui. Elementum nisi quis eleifend quam adipiscing vitae proin s  
agittis. Etiam dignissim diam quis enim lobortis. Duis at tellus at urna con  
dimentum mattis pellentesque id.  
[04/13/22] seed@VM:~/.../Task3$ md5sum out1_long.bin out2_long.bin  
fcc1317a5ead3dba83796326387dce98 out1_long.bin  
fcc1317a5ead3dba83796326387dce98 out2_long.bin  
[04/13/22] seed@VM:~/.../Task3$
```

The conclusion of this exercise reveals the hash collision weakness of the MD5 hash algorithm, where if two files that collide are appended with more data contents at the end of the existing file or data section of a packet. Both such files will have the same MD5 digest, this is not the case with more advanced hashing algorithms.

Task 4: Puzzle in Cryptocurrencies

When it comes to cryptocurrencies, one way to earn (new) is mining, which involves winning some puzzle-solving game amongst other participants. The puzzle requires 1) It is easy to verify a solution, and 2) It is hard to find solutions.

For Bitcoin, the puzzle is built using the cryptographic hash function SHA-256, which has collision resistance property (see lecture slides). SHA-256 takes as input messages of length between 0 bit to 2^{64} bits and outputs 256-bit strings. The puzzle asks to find a preimage of a SHA-256 hash value which starts with 64 zero bits, e.g.,

0x0000000000000000014dd3426129639082239efd583b5273b1bd75e8d78ff2e8d

(This is a hexadecimal number; the first 16 zeros represent the 64 zero bits). How are you going to solve the puzzle? Due to the pseudorandomness property of SHA-256, a hash value of SHA-256, it is infeasible to relate it to any input, **unless you see that input**. This means the **best** way to solve the puzzle is to randomly pick inputs and test if the first 64 bits are zeros. Three questions:

1. Why it is easy to verify a solution to the puzzle?

Answer:

Verifying the solution to the puzzle is a lot easier than finding solutions to the puzzle. The reason for this is that it is easy and straight forward to read the first 64 bits of a 256-bit digest (as is the case in this example).

Hash functions have a quality of being easily computable and irreversible as an inverse hash function, for this reason hashing the preimage to analyse the contents of the first 64 bits is the easiest part. Verification can be done immediately.

2. How many SHA-256 calculations are needed to solve the puzzle?

Answer:

The assumption of the SHA-256 algorithm is that it is pseudo-random such that it can not be distinguished from a truly random 256-bit value, for this reason, it is computationally infeasible to try to decode a SHA-256 hash digest.

A hash digest is theoretically expected to have randomly distributed outputs spanning over 256 bits.

This means that the first 64 bits will also be uniformly distributed. Mathematically this means that 2^{64} types of inputs are required to get all possible combinations for the first 64 bits. That is a one in 2^{64} chance of being correct for a one time guess, which is why the next question is very interesting.

3. Suppose a PC calculates one Mega hash per second, i.e., 1Mhash/s is approximately 2^{20} hashes/s. How much time does it take to use this PC to solve the puzzle?

Answer:

A case is presented where 2^{20} hashes can be computed per second this gives the following relationship to calculate how long 2^{64} hash combinations will take to calculate.

$$\begin{aligned} 2^{20} &\rightarrow 1 \text{ second} \\ 2^{64} &\rightarrow x \text{ seconds} \end{aligned}$$

There for the time it takes to create 2^{64} possible hashes is:

$$x = \frac{2^{64} \cdot 1}{2^{20}} = 1.7592186e^{13} \text{ seconds}$$

If we accept that there are 31,536,000 seconds in year the result in years is:

$$x_{\text{years}} = \frac{1.7592186e^{13}}{31,536,000} = 5578445.60008 \text{ years}$$

Therefore, all possible combinations for such a calculation would take about 55784 centuries to compute.

In reality, specialised SHA-256 hardware calculators are used in parallel for Bitcoin mining. Apart from constructing your mining devices, the electricity bill is another thing to worry about if you want to get rich by mining :)

Acknowledgement: This lab instruction is partially based on the SEED labs from the SEED project led by Professor Wenliang Du, Syracuse University.