Hashing - Crypto 101

An introduction to Hashing, as part of a series on crypto

[Task 1] Key Terms

Before we start, we need to get some jargon out of the way.
Read these, and take in as much as you can. We'll expand on some of them later in the room.

Plaintext - Data before encryption or hashing, often text but not always as it could be a photograph or other file instead.
Encoding - This is NOT a form of encryption, just a form of data representation like base64 or hexadecimal. Immediately reversible.
Hash - A hash is the output of a hash function. Hashing can also be used as a verb, "to hash", meaning to produce the hash value of some data.
Brute force - Attacking cryptography by trying every different password or every different key
Cryptanalysis - Attacking cryptography by finding a weakness in the underlying maths

This room will likely involve some research. Get good at using search engines, it's crucial to infosec.

#1
Read the words, and understand the meanings! Is base64 encryption or encoding?

encoding

[Task 2] What is a hash function?

What's a hash function?

Hash functions are quite different from encryption. There is no key, and it’s meant to be impossible (or very very difficult) to go from the output back to the input.
A hash function takes some input data of any size, and creates a summary or "digest" of that data. The output is a fixed size. It’s hard to predict what the output will be for any input and vice versa. Good hashing algorithms will be (relatively) fast to compute, and slow to reverse (Go from output and determine input). Any small change in the input data (even a single bit) should cause a large change in the output.
The output of a hash function is normally raw bytes, which are then encoded. Common encodings for this are base 64 or hexadecimal. Decoding these won’t give you anything useful.

Why should I care?

Hashing is used very often in cyber security. When you logged into TryHackMe, that used hashing to verify your password. When you logged into your computer, that also used hashing to verify your password. You interact indirectly with hashing more than you would think, mostly in the context of passwords.

What's a hash collision?

A hash collision is when 2 different inputs give the same output. Hash functions are designed to avoid this as best as they can, especially being able to engineer (create intentionally) a collision. Due to the pigeonhole effect, collisions are not avoidable. The pigeonhole effect is basically, there are a set number of different output values for the hash function, but you can give it any size input. As there are more inputs than outputs, some of the inputs must give the same output. If you have 128 pigeons and 96 pigeonholes, some of the pigeons are going to have to share.
MD5 and SHA1 have been attacked, and made technically insecure due to engineering hash collisions. However, no attack has yet given a collision in both algorithms at the same time so if you use the MD5 hash AND the SHA1 hash to compare, you will see they’re different. The MD5 collision example is available from https://www.mscs.dal.ca/-~selinger/md5collision/ and details of the SHA1 Collision are available from https://shattered.io/. Due to these, you shouldn't trust either algorithm for hashing passwords or data.

#1
What is the output size in bytes of the MD5 hash function?

16

#2
Can you avoid hash collisions? (Yea/Nay)

nay

#3
If you have an 8 bit hash output, how many possible hashes are there?

256

[Task 3] Uses for hashing

What can we do with hashing?

Hashing is used for 2 main purposes in Cyber Security. To verify integrity of data (More on that later), or for verifying passwords.

Hashing for password verification

Most webapps need to verify a user's password at some point. Storing these passwords in plain text would be bad. You've probably seen news stories about companies that have had their database leaked. Knowing some people, they use the same password for everything including their banking, so leaking these would be really really bad. Quite a few data breaches have leaked plaintext passwords. You're probably familiar with "rockyou.txt" on Kali as a password wordlist. This came from a company that made widgets for MySpace. They stored their passwords in plaintext and the company had a data breach. The txt file contains over 14 million passwords (although some are *unlikely* to have been user passwords. Sort by length if you want to see what I mean).

Adobe had a notable data breach that was slightly different. The passwords were encrypted, rather than hashed and the encryption that was used was not secure. This meant that the plaintext could be relatively quickly retrieved. If you want to read more about this breach, this post from Sophos is excellent: https://-nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/

Linkedin also had a data breach. Linkedin used SHA1 for password verification, which is quite quick to compute using GPUs.
You can't encrypt the passwords, as the key has to be stored somewhere. If someone gets the key, they can just decrypt the passwords.
This is where hashing comes in. What if, instead of storing the password, you just stored the hash of the password? This means you never have to store the user's password, and if your database was leaked then an attacker would have to crack each password to find out what the password was. That sounds fairly useful.
There's just one problem with this. What if two users have the same password? As a hash function will always turn the same input into the same output, you will store the same password hash for each user. That means if someone cracks that hash, they get into more than one account. It also means that someone can create a "Rainbow table" to break the hashes.
A rainbow table is a lookup table of hashes to plaintexts, so you can quickly find out what password a user had just from the hash. A rainbow table trades time taken to crack a hash for hard disk space, but they do take time to create.
Here's a quick example so you can try and understand what they're like.

HASH	**PASSWORD**
02c75fb22c75b23dc963c7eb91a062cc	zxcvbnm
b0baee9d279d34fa1dfd71aadb908c3f	11111
c44a471bd78cc6c2fea32b9fe028d30a	asdfghjkl
d0199f51d2728db6011945145a1b607a	basketball
dcddb75469b4b4875094e14561e573d8	000000
e10adc3949ba59abbe56e057f20f883e	123456
e19d5cd5af0378da05f63f891c7467af	abcd1234
e99a18c428cb38d5f260853678922e03	abc123
fcea920f7412b5da7be0cf42b8c93759	1234567

Websites like Crackstation internally use HUGE rainbow tables to provide fast password cracking for hashes without salts. Doing a lookup in a sorted list of hashes is really quite fast, much much faster than trying to crack the hash.

Protecting against rainbow tables

To protect against rainbow tables, we add a salt to the passwords. The salt is randomly generated and stored in the database, unique to each user. In theory, you could use the same salt for all users but that means that duplicate passwords would still have the same hash, and a rainbow table could still be created specific passwords with that salt. The salt is added to either the start or the end of the password before it's hashed, and this means that every user will have a different password hash even if they have the same password. Hash functions like bcrypt and sha512crypt handle this automatically. Salts don't need to be kept private.

#1

Crack the hash

"d0199f51d2728db6011945145a1b607a" using the rainbow table manually.

basketball

#2

Crack the hash

"5b31f93c09ad1d065c0491b764d04933" using online tools

tryhackme

#3

Should you encrypt passwords? Yea/Nay

nay

[Task 4] Recognising password hashes

Automated hash recognition tools such as https://pypi.org/project/hashID/ exist, but they are unreliable for many formats. For hashes that have a prefix, the tools are reliable. Use a healthy combination of context and tools. If you found the hash in a web application database, it's more likely to be md5 than NTLM. Automated hash recognition tools often get these hash types mixed up, which highlights the importance of learning yourself.
Unix style password hashes are very easy to recognise, as they have a prefix. The prefix tells you the hashing algorithm used to generate the hash. The standard format is `$format$rounds$salt$hash`.
Windows passwords are hashed using NTLM, which is a variant of md4. They're visually identical to md4 and md5 hashes, so it's very important to use context to work out the hash type.
On Linux, password hashes are stored in /etc/shadow. This file is normally only readable by root. They used to be stored in /etc/passwd, and were readable by everyone.
On Windows, password hashes are stored in the SAM. Windows tries to prevent normal users from dumping them, but tools like mimikatz exist for this. Importantly, the hashes found there are split into NT hashes and LM hashes.
Here's a quick table of the most Unix style password prefixes that you'll see.

Prefix	**Algorithm**
$1$	md5crypt, used in Cisco stuff and older Linux/Unix systems
$2$, $2a$, $2b$, $2x$, $2y$	Bcrypt (Popular for web applications)
$6$	sha512crypt (Default for most Linux/Unix systems)

A great place to find more hash formats and password prefixes is the hashcat example page, available here: https://hashcat.net/wiki/doku.php?id=example_hashes.
For other hash types, you'll normally need to go by length, encoding or some research into the application that generated them. Never underestimate the power of research.

#1
How many rounds does sha512crypt ($6$) use by default?

5000

#2
What's the hashcat example hash (from the website) for Citrix Netscaler hashes?

1765058016a22f1b4e076dccd1c3df4e8e5c0839ccded98ea

#3
How long is a Windows NTLM hash, in characters?

32

[Task 5] Password Cracking

We've already mentioned rainbow tables as a method to crack hashes that don't have a salt, but what if there's a salt involved?
You can't "decrypt" password hashes. They're not encrypted. You have to crack the hashes by hashing a large number of different inputs (often rockyou, these are the possible passwords), potentially adding the salt if there is one and comparing it to the target hash. Once it matches, you know what the password was. Tools like Hashcat and John the Ripper are normally used for this.

Why crack on GPUs?

Graphics cards have thousands of cores. Although they can't do the same sort of work that a CPU can, they are very good at some of the maths involved in hash functions. This means you can use a graphics card to crack most hash types much more quickly. Some hashing algorithms, notably bcrypt, are designed so that hashing on a GPU is about the same speed as hashing on a CPU which helps them resist cracking.

Cracking on VMs?

It's worth mentioning that virtual machines normally don't have access to the host's graphics card(s) (You can set this up, but it's a lot of work). If you want to run hashcat, it's best to run it on your host (Windows builds are available on the website, run it from powershell). You can get Hashcat working with OpenCL in a VM, but the speeds will likely be much worse than cracking on your host. John the ripper uses CPU by default and as such, works in a VM out of the box although you may get better speeds running it on the host OS as it will have more threads and no overhead from running in a VM.
NEVER (I repeat, NEVER!) use --force for hashcat. It can lead to false positives (wrong passwords being given to you) and false negatives (skips over the correct hash).
UPDATE: As of Kali 2020.2, hashcat 6.0 will run on the CPU without --force. I still recommend cracking on your host OS if you have a GPU, as it will be much much faster.

Time to get cracking!

I'll provide the hashes. Crack them. You can choose how. You'll need to use online tools, Hashcat, and/or John the Ripper. Remember the restrictions on online rainbow tables. Don't be afraid to use the hints. Rockyou or online tools should be enough to find all of these.

#1
Crack this hash: $2a$06$7yoU3Ng8dHTXphAg913cyO6Bjs3K5lBnwq5FJyA6d01pMSrddr1ZG

85208520

#2
Crack this hash: 9eb7ee7f551d2f0ac684981bd1f1e2fa4a37590199636753efe614d4db30e8e1

halloween

#3
Crack this hash: $6$GQXVvW4EuM$ehD6jWiMsfNorxy5SINsgdlxmAEl3.yif0/- c3NqzGLa0P.S7KRDYjycw5bnYkF5ZtB8wQy8KnskuWQS3Yr1wQ0

spaceman

#4
Bored of this yet? Crack this hash: b6b0d451bbf6fed658659a9e7e5598fe

funforyou

[Task 6] Hashing for integrity checking

Integrity Checking

Hashing can be used to check that files haven't been changed. If you put the same data in, you always get the same data out. If even a single bit changes, the hash will change a lot. This means you can use it to check that files haven't been modified or to make sure that they have downloaded correctly. You can also use hashing to find duplicate files, if two pictures have the same hash then they are the same picture.

HMACs

HMAC is a method of using a cryptographic hashing function to verify the authenticity and integrity of data. The TryHackMe VPN uses HMAC-SHA512 for message authentication, which you can see in the terminal output. A HMAC can be used to ensure that the person who created the HMAC is who they say they are (authenticity), and that the message hasn't been modified or corrupted (integrity). They use a secret key, and a hashing algorithm in order to produce a hash.

#1
What's the SHA1 sum for the amd64 Kali 2019.4 ISO? https://cdimage.kali.org/kali-images/kali-2019.4/

186c5227e24ceb60deb711f1bdc34ad9f4718ff9

#2
What's the hashcat mode number for HMAC-SHA512 (key = $pass)?

1750