# Applied Cryptography in Cybersecurity

Sachin Kumar

Spring 2024

**Abstract**

Cryptography is essential in security. Learn how it's used to preserve integrity and confidentiality of sensitive information. The internet is used by virtually everyone today for very trivial purposes such as playing games to more important tasks such as online banking. Users have started using the internet to access and utilise important services. A large part of ensuring that this communication is secure is done using Cryptographic primitives. This module will explore core cryptographic topics that are used to secure communication over the internet and in machines/servers.

## Contents

**1 Cryptography** **2**
1.1 Hashing - Crypto 101 . . . 2
1.1.1 What's a hash function? . . . 2
1.1.2 Uses for hashing . . . 3
1.1.3 Recognising password hashes . . . 5
1.1.4 Password Cracking . . . 6
1.1.5 Hashing for integrity checking . . . 7
1.2 John The Ripper . . . 8
1.2.1 John who? . . . 8
1.2.2 Setting up John the Ripper . . . 9
1.2.3 Wordlists . . . 10
1.2.4 Cracking Basic Hashes . . . 11
1.2.5 Cracking Windows Authentication Hashes . . . 12
1.2.6 Cracking /etc/shadow Hashes . . . 13
1.2.7 Single Crack Mode . . . 13
1.2.8 Custom Rules . . . 14
1.2.9 Cracking Password Protected Zip Files . . . 17
1.2.10 Cracking Password Protected RAR Archives . . . 17
1.2.11 Cracking SSH Keys with John . . . 18
1.3 Encryption - Crypto 101 . . . 19
1.3.1 Why is Encryption important? . . . 20
1.3.2 Crucial Crypto Maths . . . 20
1.3.3 Types of Encryption . . . 21

1.3.4 RSA - Rivest Shamir Adleman . . . . . . . . . . . . . . . 21
1.3.5 Establishing Keys Using Asymmetric Cryptography . . . 22
1.3.6 Digital signatures and Certificates . . . . . . . . . . . . . 22
1.3.7 SSH Authentication . . . . . . . . . . . . . . . . . . . . . 23
1.3.8 Explaining Diffie Hellman Key Exchange . . . . . . . . . 24
1.3.9 PGP, GPG and AES . . . . . . . . . . . . . . . . . . . . . 25
1.3.10 The Future - Quantum Computers and Encryption . . . . 25

# 1 Cryptography

## 1.1 Hashing - Crypto 101

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.

1. *Plaintext* - Data before encryption or hashing, often text but not always as it could be a photograph or other file instead.

2. *Encoding* - This is NOT a form of encryption, just a form of data representation like base64 or hexadecimal. Immediately reversible.

3. *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.

4. *Brute force* - Attacking cryptography by trying every different password or every different key

5. *Cryptanalysis* - Attacking cryptography by finding a weakness in the underlying maths.

*Question.* Is base64 encryption or encoding? Encoding

### 1.1.1 What's 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 cybersecurity. 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.

*Question.*

1. What is the output size in bytes of the MD5 hash function? 16
2. Can you avoid hash collisions? Nay
3. If you have an 8-bit hash output, how many possible hashes are there? 256

#### 1.1.2 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.

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.

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.

Here's a quick example of Hash functions 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 |

*Questions.*

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? Nay

### 1.1.3 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.

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=`

| 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) |

`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.

*Questions.*

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

### 1.1.4 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.

*Quesions.*

1. Crack this hash:

   $2a$06$7yoU3Ng8dHTXphAg913cyO6Bjs3K5lBnwq5FJyA6d01pMSrddr1ZG

   Solution. 85208520

2. Crack this hash:

   9eb7ee7f551d2f0ac684981bd1f1e2fa4a37590199636753efe614d4db30e8e1

   Solution. halloween

3. Crack this hash:

   $6$GQXVvW4EuM$ehD6jWiMsfNorxy5SINsgdlxmAEl3.yif0/
   c3NqzGLa0P.S7KRDYjycw5bnYkF5ZtB8wQy8KnskuWQS3Yr1wQ0

   Solution. spaceman

4. Bored of this yet? Crack this hash:

   b6b0d451bbf6fed658659a9e7e5598fe

   Solution. funforyou

### 1.1.5 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.

*Questions.*

1. What's the SHA1 sum for the amd64 Kali 2019.4 ISO? `http://old.kali.org/kali-images/kali-2019.4/` 186c5227e24ceb60deb711f1bdc34ad9f4718ff9
2. What's the hashcat mode number for HMAC-SHA512 (key = $pass)? 1750

## 1.2 John The Ripper

### 1.2.1 John who?

Welcome John the Ripper is one of the most well known, well-loved and versatile hash cracking tools out there. It combines a fast cracking speed, with an extraordinary range of compatible hash types. This room will assume no previous knowledge, so we must first cover some basic terms and concepts before we move into practical hash cracking.

- *What are Hashes?* A hash is a way of taking a piece of data of any length and representing it in another form that is a fixed length. This masks the original value of the data. This is done by running the original data through a hashing algorithm. There are many popular hashing algorithms, such as MD4,MD5, SHA1 and NTLM. Lets try and show this with an example:

  If we take "polo", a string of 4 characters- and run it through an MD5 hashing algorithm, we end up with an output of:

  b53759f3ce692de7aff1b5779d3964da

  a standard 32 character MD5 hash.

  Likewise, if we take "polomints", a string of 9 characters- and run it through the same MD5 hashing algorithm, we end up with an output of: 584b6e4f4586e136bc280f27f9c64f3b another standard 32 character MD5 hash.

- *What makes Hashes secure?* Hashing functions are designed as one-way functions. In other words, it is easy to calculate the hash value of a given input; however, it is a difficult problem to find the original input given the hash value. By "difficult", we mean that it is computationally infeasible. This has its roots in mathematics and P vs NP.

  In computer science, P and NP are two classes of problems that help us understand the efficiency of algorithms:

  - *P (Polynomial Time):* Class P covers the problems whose solution can be found in polynomial time. Consider sorting a list in increasing order. The longer the list, the longer it would take to sort; nonetheless, the increase in time is not exponential.

  - *NP (Non-deterministic Polynomial Time):* Problems in the class NP are those for which a given solution can be checked quickly, even though finding the solution itself might be hard. In fact, we don't know if there is a fast algorithm to find the solution in the first place.

  While this is an extremely interesting mathematical concept that proves fundamental to computing and cryptography, it is completely outside the scope of this room. But abstractly it means that the algorithm to hash the value will be "P" and can therefore be calculated reasonably. However an un-hashing algorithm would be "NP" and intractable to solve- meaning that it cannot be computed in a reasonable time using standard computers.

- *Where John Comes in...* Even though the algorithm itself is not feasibly reversible. That doesn't mean that cracking the hashes is impossible. If you have the hashed version of a password, for example- and you know the hashing algorithm- you can use that hashing algorithm to hash a large number of words, called a dictionary. You can then compare these hashes to the one you're trying to crack, to see if any of them match. If they do, you now know what word corresponds to that hash- you've cracked it!

  This process is called a *dictionary attack* and John the Ripper, or John as it's commonly shortened to, is a tool to allow you to conduct fast brute force attacks on a large array of different hash types.

### 1.2.2 Setting up John the Ripper

John the Ripper is supported on many different Operating Systems, not just Linux Distributions. As a note before we go through this, there are multiple versions of John, the standard "core" distribution, as well as multiple community editions- which extend the feature set of the original John distribution. The most popular of John the Ripper distributions is the "Jumbo John"- which we will be using specific features of later.

- *Parrot, Kali and AttackBox.* If you're using Parrot OS, Kali Linux or Try-HackMe's own AttackBox- you should already have Jumbo John installed. You can double check this by typing `john` into the terminal. You should be met with a usage guide for john, with the first line reading: "John the Ripper 1.9.0-jumbo-1" or similar with a different version number. If not, you can use `sudo apt install john` to install it.

- *Blackarch.* If you're using Blackarch, or the Blackarch repositories you may or may not have Jumbo John installed, to check if you do, use the command `pacman -Qe | grep "john"` You should be met with an output similar to "john 1.9.0.jumbo1-5" or similar with a different version number. If you do not have it installed, you can simply use `pacman -S john` to install it.

- *Building from Source for Linux.* If you wish to build the package from source to meet your system requirements, you can do this in five fairly straightforward steps. Further advice on the installation process and how to configure your build from source can be found here.

  1. Use `git clone https://github.com/openwall/john -b bleeding-jumbo john` to clone the jumbo john repository to your current working.
  2. Then `cd john/src/` to change your current directory to where the source code is.
  3. Once you're in this directory, use `./configure` to check the required dependencies and options that have been configured.
  4. If you're happy with this output, and have installed any required dependencies that are needed, use `make -s clean && make -sj4` to build a binary of john. This binary will be in the above run directory, which you can change to with `cd ../run`.
  5. You can test this binary using `./john --test`.

  To install Jumbo John the Ripper on Windows, you just need to download and install the zipped binary for either 64 bit systems here or for 32 bit systems here.

### 1.2.3 Wordlists

As we explained in the first task, in order to dictionary attack hashes, you need a list of words that you can hash and compare, unsurprisingly this is called a wordlist. There are many different wordlists out there, a good collection to use can be found in the SecLists repository. There are a few places you can look for wordlists on your attacking system of choice, we will quickly run through where you can find them.

- *Parrot, Kali and AttackBox.* On Parrot, Kali and TryHackMe's AttackBox- you can find a series of amazing wordlists in the `/usr/share/wordlists` directory.

- *RockYou.* For all of the tasks in this room, we will be using the infamous rockyou.txt wordlist- which is a very large common password wordlist, obtained from a data breach on a website called rockyou.com in 2009. If you are not using any of the above distributions, you can get the rockyou.txt wordlist from the SecLists repository under the `/Passwords/Leaked-Databases` subsection. You may need to extract it from .tar.gz format, using `tar xvzf rockyou.txt.tar.gz`.

Now that we have our hash cracker and wordlists all set up, lets move onto some hash cracking!

*Questions.*

1. What website was the rockyou.txt wordlist created from a breach on? rockyou.com

### 1.2.4 Cracking Basic Hashes

There are multiple ways to use John the Ripper to crack simple hashes, we're going to walk through a few, before moving on to cracking some ourselves.

- *John Basic Syntax.* The basic syntax of John the Ripper commands is as follows. We will cover the specific options and modifiers used as we use them.
  - `john [options] [path to file]`
  - `john` - Invokes the John the Ripper program
  - `[path to file]` - The file containing the hash you're trying to crack, if it's in the same directory you won't need to name a path, just the file.
- *Automatic Cracking.* John has built-in features to detect what type of hash it's being given, and to select appropriate rules and formats to crack it for you, this isn't always the best idea as it can be unreliable- but if you can't identify what hash type you're working with and just want to try cracking it, it can be a good option! To do this we use the following syntax:
  - `john --wordlist=[path to wordlist] [path to file]`
  - `--wordlist=` - Specifies using wordlist mode, reading from the file that you supply in the following path...
  - `[path to wordlist]` - The path to the wordlist you're using, as described in the previous task.
- *Identifying Hashes.* Sometimes John won't play nicely with automatically recognising and loading hashes, that's okay! We're able to use other tools to identify the hash, and then set john to use a specific format. There are multiple ways to do this, such as using an online hash identifier like this one. I like to use a tool called hash-identifier, a Python tool that is super easy to use and will tell you what different types of hashes the one you enter is likely to be, giving you more options if the first one fails.

  To use hash-identifier, you can just pull the python file from gitlab using: `wget https://gitlab.com/kalilinux/packages/hash-identifier /-/raw/kali/master/hash-id.py`.

  Then simply launch it with `python3 hash-id.py` and then enter the hash you're trying to identify- and it will give you possible formats!
- *Format-Specific Cracking.* Once you have identified the hash that you're dealing with, you can tell john to use it while cracking the provided hash using the following syntax:

- `john --format=[format] --wordlist=[path to wordlist] [path to file]`
- `--format=` - This is the flag to tell John that you're giving it a hash of a specific format, and to use the following format to crack it.
- `[format]` - The format that the hash is in.

A Note on Formats: When you are telling john to use formats, if you're dealing with a standard hash type, e.g. md5 as in the example above, you have to prefix it with `raw-` to tell john you're just dealing with a standard hash type, though this doesn't always apply. To check if you need to add the prefix or not, you can list all of John's formats using `john --list=formats` and either check manually, or grep for your hash type using something like `john --list=formats | grep -iF "md5"`.

#### 1.2.5 Cracking Windows Authentication Hashes

Now that we understand the basic syntax and usage of John the Ripper- lets move on to cracking something a little bit more difficult, something that you may even want to attempt if you're on a real Penetration Test or Red Team engagement. Authentication hashes are the hashed versions of passwords that are stored by operating systems, it is sometimes possible to crack them using the brute-force methods that we're using. To get your hands on these hashes, you must often already be a privileged user- so we will explain some of the hashes that we plan on cracking as we attempt them.

- *NTHash / NTLM.* NThash is the hash format that modern Windows Operating System machines will store user and service passwords in. It's also commonly referred to as "NTLM" which references the previous version of Windows format for hashing passwords known as "LM", thus "NT/LM".

  A little bit of history, the NT designation for Windows products originally meant "New Technology", and was used- starting with Windows NT, to denote products that were not built up from the MS-DOS Operating System. Eventually, the "NT" line became the standard Operating System type to be released by Microsoft and the name was dropped, but it still lives on in the names of some Microsoft technologies.

  You can acquire NTHash/NTLM hashes by dumping the SAM database on a Windows machine, by using a tool like Mimikatz or from the Active Directory database: NTDS.dit. You may not have to crack the hash to continue privilege escalation- as you can often conduct a "pass the hash" attack instead, but sometimes hash cracking is a viable option if there is a weak password policy.

### 1.2.6 Cracking /etc/shadow Hashes

The /etc/shadow file is the file on Linux machines where password hashes are stored. It also stores other information, such as the date of last password change and password expiration information. It contains one entry per line for each user or user account of the system. This file is usually only accessible by the root user- so in order to get your hands on the hashes you must have sufficient privileges, but if you do- there is a chance that you will be able to crack some of the hashes.

- *Unshadowing.* John can be very particular about the formats it needs data in to be able to work with it, for this reason- in order to crack /etc/shadow passwords, you must combine it with the /etc/passwd file in order for John to understand the data it's being given. To do this, we use a tool built into the John suite of tools called unshadow. The basic syntax of unshadow is as follows:
  - `unshadow [path to passwd] [path to shadow]`
  - `unshadow` - Invokes the unshadow tool
  - `[path to passwd]` - The file that contains the copy of the /etc/passwd file you've taken from the target machine.
  - `[path to shadow]` - The file that contains the copy of the /etc/shadow file you've taken from the target machine.
- *Cracking.* We're then able to feed the output from unshadow, in our example use case called "unshadowed.txt" directly into John. We should not need to specify a mode here as we have made the input specifically for John, however in some cases you will need to specify the format as we have done previously using: `--format=sha512crypt`

  ```
  john --wordlist=/usr/share/wordlists/rockyou.txt
  --format=sha512crypt unshadowed.txt
  ```

### 1.2.7 Single Crack Mode

o far we've been using John's wordlist mode to deal with brute forcing simple., and not so simple hashes. But John also has another mode, called Single Crack mode. In this mode, John uses only the information provided in the username, to try and work out possible passwords heuristically, by slightly changing the letters and numbers contained within the username.

- *Word Mangling.* The best way to show what Single Crack mode is, and what word mangling is, is to actually go through an example:

  If we take the username: Markus

  Some possible passwords could be:

  - Markus1, Markus2, Markus3 (etc.)
  - MArkus, MARkus, MARKus (etc.)
  - Markus!, Markus$, Markus* (etc.)

  This technique is called word mangling. John is building it's own dictionary based on the information that it has been fed and uses a set of rules called "mangling rules" which define how it can mutate the word it started with to generate a wordlist based off of relevant factors for the target you're trying to crack. This is exploiting how poor passwords can be based off of information about the username, or the service they're logging into.

- *GECOS.* John's implementation of word mangling also features compatibility with the Gecos fields of the UNIX operating system, and other UNIX-like operating systems such as Linux. So what are Gecos? Remember in the last task where we were looking at the entries of both /etc/shadow and /etc/passwd? Well if you look closely You can see that each field is seperated by a colon ":". Each one of the fields that these records are split into are called Gecos fields. John can take information stored in those records, such as full name and home directory name to add in to the wordlist it generates when cracking /etc/shadow hashes with single crack mode.

- *Using Single Crack Mode.* To use single crack mode, we use roughly the same syntax that we've used to so far, for example if we wanted to crack the password of the user named "Mike", using single mode, we'd use:
  - `john --single --format=[format] [path to file]`
  - `--single` - This flag lets john know you want to use the single hash cracking mode.

  Example usage: john –single –format=raw-sha256 hashes.txt

  If you're cracking hashes in single crack mode, you need to change the file format that you're feeding john for it to understand what data to create a wordlist from. You do this by prepending the hash with the username that the hash belongs to, so according to the above example- we would change the file hashes.txt

  1efee03cdcb96d90ad48ccc7b8666033 $\mapsto$ mike:1efee03cdcb96d90ad48ccc7b8666033

### 1.2.8 Custom Rules

- *What are Custom Rules?* As we journeyed through our exploration of what John can do in Single Crack Mode- you may have some ideas about what some good mangling patterns would be, or what patterns your passwords often use- that could be replicated with a certain mangling pattern.

The good news is you can define your own sets of rules, which John will use to dynamically create passwords. This is especially useful when you know more information about the password structure of whatever your target is.

- *Common Custom Rules.* Many organisations will require a certain level of password complexity to try and combat dictionary attacks, meaning that if you create an account somewhere, go to create a password and enter: `polopassword`.

  You may receive a prompt telling you that passwords have to contain at least one of the following:

  - Capital letter
  - Number
  - Symbol

  This is good! However, we can exploit the fact that most users will be predictable in the location of these symbols. For the above criteria, many users will use something like the following: `Polopassword1!`.

  A password with the capital letter first, and a number followed by a symbol at the end. This pattern of the familiar password, appended and prepended by modifiers (such as the capital letter or symbols) is a memorable pattern that people will use, and reuse when they create passwords. This pattern can let us exploit password complexity predictability. Now this does meet the password complexity requirements, however as an attacker we can exploit the fact we know the likely position of these added elements to create dynamic passwords from our wordlists.

- *How to create Custom Rules?* Custom rules are defined in the `john.conf` file, usually located in `/etc/john/john.conf` if you have installed John using a package manager or built from source with `make` and in `/opt/john/john.conf` on the TryHackMe Attackbox.

  Let's go over the syntax of these custom rules, using the example above as our target pattern. Note that there is a massive level of granular control that you can define in these rules, I would suggest taking a look at the wiki here in order to get a full view of the types of modifier you can use, as well as more examples of rule implementation.

  The first line:

  - `[List.Rules:THMRules]` - Is used to define the name of your rule, this is what you will use to call your custom rule as a John argument.

    We then use a regex style pattern match to define where in the word will be modified, again- we will only cover the basic and most common modifiers here:

  - `Az` - Takes the word and appends it with the characters you define
  - `A0` - Takes the word and prepends it with the characters you define
  - `c` - Capitalises the character positionally

These can be used in combination to define where and what in the word you want to modify. Lastly, we then need to define what characters should be appended, prepended or otherwise included, we do this by adding character sets in square brackets `[ ]` in the order they should be used. These directly follow the modifier patterns inside of double quotes `" "`. Here are some common examples:

  - `[0-9]` - Will include numbers 0-9
  - `[0]` - Will include only the number 0
  - `[A-z]` - Will include both upper and lowercase
  - `[A-Z]` - Will include only uppercase letters
  - `[a-z]` - Will include only lowercase letters
  - `[a]` - Will include only a
  - `[!£$%]` - Will include the symbols !£$%

Putting this all together, in order to generate a wordlist from the rules that would match the example password "Polopassword1!" (assuming the word polopassword was in our wordlist) we would create a rule entry that looks like this:

  - `[List.Rules:PoloPassword]`
  - `cAz"[0-9] [!£$%]"`

In order to:

  - Capitalise the first letter - `c`
  - Append to the end of the word - `Az`
  - A number in the range 0-9 - `[0-9]`
  - Followed by a symbol that is one of `[!£$%]`

- *Using Custom Rules.* We could then call this custom rule as a John argument using the `--rule=PoloPassword` flag. As a full command: `john --wordlist=[path to wordlist] --rule=PoloPassword [path to file]`.

  As a note I find it helpful to talk out the patterns if you're writing a rule- as shown above, the same applies to writing RegEx patterns too.

  Jumbo John already comes with a large list of custom rules, which contain modifiers for use almost all cases. If you get stuck, try looking at those rules [around line 678] if your syntax isn't working properly.

*Questions.*

1. What do custom rules allow us to exploit? Password complexity predictability
2. What rule would we use to add all capital letters to the end of the word? `Az"[A-Z]"`
3. What flag would we use to call a custom rule called "THMRules"? `--rule=THMRules`

#### 1.2.9 Cracking Password Protected Zip Files

We can use John to crack the password on password protected Zip files. Again, we're going to be using a separate part of the john suite of tools to convert the zip file into a format that John will understand, but for all intents and purposes, we're going to be using the syntax that you're already pretty familiar with by now.

- *Zip2John.* Similarly to the unshadow tool that we used previously, we're going to be using the zip2john tool to convert the zip file into a hash format that John is able to understand, and hopefully crack. The basic usage is like this:
  - `zip2john [options] [zip file] > [output file]`
  - `[options]` - Allows you to pass specific checksum options to zip2john, this shouldn't often be necessary
  - `[zip file]` - The path to the zip file you wish to get the hash of
  - `>` - This is the output director, we're using this to send the output from this file to the...
  - `[output file]` - This is the file that will store the output from

  Example usage: zip2john zipfile.zip > zip_hash.txt
- *Cracking.* We're then able to take the file we output from zip2john in our example use case called "zip_hash.txt" and, as we did with unshadow, feed it directly into John as we have made the input specifically for it.
  `john --wordlist=/usr/share/wordlists/rockyou.txt zip_hash.txt`

#### 1.2.10 Cracking Password Protected RAR Archives

We can use a similar process to the one we used in the last task to obtain the password for rar archives. If you aren't familiar, rar archives are compressed files created by the Winrar archive manager. Just like zip files they compress a wide variety of folders and files.

- *Rar2John.* Almost identical to the zip2john tool that we just used, we're going to use the rar2john tool to convert the rar file into a hash format that John is able to understand. The basic syntax is as follows:

  - `rar2john [rar file] > [output file]`
  - `rar2john` - Invokes the rar2john tool
  - `[rar file]` - The path to the rar file you wish to get the hash of
  - `>` - This is the output director, we're using this to send the output from this file to the...
  - `[output file]` - This is the file that will store the output from

  Example usage: rar2john rarfile.rar ¿ rar_hash.txt

- *Cracking.* Once again, we're then able to take the file we output from rar2john in our example use case called "rar_hash.txt" and, as we did with zip2john we can feed it directly into John..
  `john --wordlist=/usr/share/wordlists/rockyou.txt rar_hash.txt`.

#### 1.2.11 Cracking SSH Keys with John

Let's explore one more use of John that comes up semi-frequently in CTF challenges. Using John to crack the SSH private key password of id_rsa files. Unless configured otherwise, you authenticate your SSH login using a password. However, you can configure key-based authentication, which lets you use your private key, id_rsa, as an authentication key to login to a remote machine over SSH. However, doing so will often require a password- here we will be using John to crack this password to allow authentication over SSH using the key.

- *SSH2John.* As the name suggests ssh2john converts the id_rsa private key that you use to login to the SSH session into hash format that john can work with. Jokes aside, it's another beautiful example of John's versatility. The syntax is about what you'd expect. Note that if you don't have ssh2john installed, you can use ssh2john.py, which is located in the /opt/john/ssh2john.py. If you're doing this, replace the `ssh2john` command with `python3 /opt/ssh2john.py` or on Kali, `python /usr/share/john/ssh2john.py`.

  - `ssh2john [id_rsa private key file] > [output file]`
  - ssh2john - Invokes the ssh2john tool
  - `[id_rsa private key file]` - The path to the id_rsa file you wish to get the hash of
  - `>` - This is the output director, we're using this to send the output from this file to the...
  - `[output file]` - This is the file that will store the output from

  Example usage: ssh2john id_rsa ¿ id_rsa_hash.txt

- For the final time, we're feeding the file we output from ssh2john, which in our example use case is called "id_rsa_hash.txt" and, as we did with rar2john we can use this seamlessly with John:
  ```
  john --wordlist=/usr/share/wordlists/rockyou.txt id_rsa_hash.txt
  ```

Click here, to learn more about John the ripper.

## 1.3 Encryption - Crypto 101

In this section, we will cover:

1. Why cryptography matters for security and CTFs
2. The two main classes of cryptography and their uses
3. RSA, and some of the uses of RSA
4. 2 methods of Key Exchange
5. Notes about the future of encryption with the rise of Quantum Computing

Some Key terms:

1. *Ciphertext* - The result of encrypting a plaintext, encrypted data
2. *Cipher* - A method of encrypting or decrypting data. Modern ciphers are cryptographic, but there are many non cryptographic ciphers like Caesar.
3. *Plaintext* - Data before encryption, often text but not always. Could be a photograph or other file
4. *Encryption* - Transforming data into ciphertext, using a cipher.
5. *Encoding* - NOT a form of encryption, just a form of data representation like base64. Immediately reversible.
6. *Key* - Some information that is needed to correctly decrypt the ciphertext and obtain the plaintext.
7. *Passphrase* - Separate to the key, a passphrase is similar to a password and used to protect a key.
8. *Asymmetric encryption* - Uses different keys to encrypt and decrypt.
9. *Symmetric encryption* - Uses the same key to encrypt and decrypt
10. *Brute force* - Attacking cryptography by trying every different password or every different key
11. *Cryptanalysis* - Attacking cryptography by finding a weakness in the underlying maths
12. *Alice and Bob* - Used to represent 2 people who generally want to communicate. They're named Alice and Bob because this gives them the initials A and B, these extend through the alphabet to represent many different people involved in communication.

### 1.3.1 Why is Encryption important?

Cryptography is used to protect confidentiality, ensure integrity, ensure authenticity. You use cryptography every day most likely, and you're almost certainly reading this now over an encrypted connection.

When logging into TryHackMe, your credentials were sent to the server. These were encrypted, otherwise someone would be able to capture them by snooping on your connection.

When you connect to SSH (Secure Shell), your client and the server establish an encrypted tunnel so that no one can snoop on your session.

When you connect to your bank, there's a certificate (webservers prove their identity using this) that uses cryptography to prove that it is actually your bank rather than a hacker.

When you download a file, how do you check if it downloaded right? You can use cryptography here to verify a checksum of the data.

You rarely have to interact directly with cryptography, but it silently protects almost everything you do digitally.

Whenever sensitive user data needs to be stored, it should be encrypted. Standards like PCI-DSS state that the data should be encrypted both at rest (in storage) AND while being transmitted. If you're handling payment card details, you need to comply with these PCI regulations. Medical data has similar standards. With legislation like GDPR and California's data protection, data breaches are extremely costly and dangerous to you as either a consumer or a business.

DO NOT encrypt passwords unless you're doing something like a password manager. Passwords should not be stored in plaintext, and you should use hashing to manage them safely.

### 1.3.2 Crucial Crypto Maths

There's a little bit of math(s) that comes up relatively often in cryptography. The Modulo operator. Pretty much every programming language implements this operator, or has it available through a library. When you need to work with large numbers, use a programming language. Python is good for this as integers are unlimited in size, and you can easily get an interpreter.

When learning division for the first time, you were probably taught to use remainders in your answer. `X % Y` is the remainder when $X$ is divided by $Y$.

*Examples:*

1. $25\%5 = 0$ ($5*5 = 25$ so it divides exactly with no remainder)

2. $23\%6 = 5$ (23 does not divide evenly by 6, there would be a remainder of 5)

3. An important thing to remember about modulo is that it's not reversible. If I gave you an equation: $x\%5 = 4$, there are infinite values of $x$ that will be valid.

### 1.3.3 Types of Encryption

The two main categories of Encryption are symmetric and asymmetric.

- *Symmetric encryption* uses the same key to encrypt and decrypt the data. Examples of Symmetric encryption are DES (Broken) and AES. These algorithms tend to be faster than asymmetric cryptography, and use smaller keys (128 or 256 bit keys are common for AES, DES keys are 56 bits long).

- *Asymmetric encryption* uses a pair of keys, one to encrypt and the other in the pair to decrypt. Examples are RSA and Elliptic Curve Cryptography. Normally these keys are referred to as a public key and a private key. Data encrypted with the private key can be decrypted with the public key, and vice versa. Your private key needs to be kept private, hence the name. Asymmetric encryption tends to be slower and uses larger keys, for example RSA typically uses 2048 to 4096 bit keys.

RSA and Elliptic Curve cryptography are based around different mathematically difficult (intractable) problems, which give them their strength. More about RSA later.

### 1.3.4 RSA - Rivest Shamir Adleman

- *The math(s) side.* RSA is based on the mathematically difficult problem of working out the factors of a large number. It's very quick to multiply two prime numbers together, say 17*23 = 391, but it's quite difficult to work out what two prime numbers multiply together to make 14351 (113x127 for reference).

- *The attacking side.* The maths behind RSA seems to come up relatively often in CTFs, normally requiring you to calculate variables or break some encryption based on them. The wikipedia page for RSA seems complicated at first, but will give you almost all of the information you need in order to complete challenges.

  There are some excellent tools for defeating RSA challenges in CTFs, and my personal favorite is `https://github.com/Ganapati/RsaCtfTool` which has worked very well for me. I've also had some success with `https://github.com/ius/rsatool`.

  The key variables that you need to know about for RSA in CTFs are $p$, $q$, $m$, $n$, $e$, $d$, and $c$. $p$ and $q$ are large prime numbers, $n$ is the product of $p$ and $q$. The public key is $n$ and $e$, the private key is $n$ and $d$. $m$ is used to represent the message (in plaintext) and $c$ represents the ciphertext (encrypted text).

- *CTFs involving RSA.* Crypto CTF challenges often present you with a set of these values, and you need to break the encryption and decrypt a message to retrieve the flag.

There's a lot more maths to RSA, and it gets quite complicated fairly quickly. If you want to learn the maths behind it, I recommend reading MuirlandOracle's blog post here: `https://muirlandoracle.co.uk/2020/01/29/rsa-encryption/`.

### 1.3.5 Establishing Keys Using Asymmetric Cryptography

A very common use of asymmetric cryptography is exchanging keys for symmetric encryption. Asymmetric encryption tends to be slower, so for things like HTTPS symmetric encryption is better. But the question is, how do you agree a key with the server without transmitting the key for people snooping to see?

- *Metaphor time.* Imagine you have a secret code, and instructions for how to use the secret code. If you want to send your friend the instructions without anyone else being able to read it, what you could do is ask your friend for a lock. Only they have the key for this lock, and we'll assume you have an indestructible box that you can lock with it. If you send the instructions in a locked box to your friend, they can unlock it once it reaches them and read the instructions. After that, you can communicate in the secret code without risk of people snooping. In this metaphor, the secret code represents a symmetric encryption key, the lock represents the server's public key, and the key represents the server's private key. You've only used asymmetric cryptography once, so it's fast, and you can now communicate privately with symmetric encryption.
- *The Real World.* In reality, you need a little more cryptography to verify the person you're talking to is who they say they are, which is done using digital signatures and certificates. You can find a lot more detail on how HTTPS (one example where you need to exchange keys) really works from this excellent blog post. `https://robertheaton.com/2014/03/27/how-does-https-actually-work/`

### 1.3.6 Digital signatures and Certificates

- *What's a Digital Signature?* Digital signatures are a way to prove the authenticity of files, to prove who created or modified them. Using asymmetric cryptography, you produce a signature with your private key and it can be verified using your public key. As only you should have access to your private key, this proves you signed the file. Digital signatures and physical signatures have the same value in the UK, legally.

  The simplest form of digital signature would be encrypting the document with your private key, and then if someone wanted to verify this signature they would decrypt it with your public key and check if the files match.
- *Certificates - Prove who you are!* Certificates are also a key use of public key cryptography, linked to digital signatures. A common place where

they're used is for HTTPS. How does your web browser know that the server you're talking to is the real website?

The answer is certificates. The web server has a certificate that says it is the real website. The certificates have a chain of trust, starting with a root CA (certificate authority). Root CAs are automatically trusted by your device, OS, or browser from install. Certs below that are trusted because the Root CAs say they trust that organisation. Certificates below that are trusted because the organisation is trusted by the Root CA and so on. There are long chains of trust. Again, this blog post explains this much better than I can. `https://robertheaton.com/2014/03/27/how-does-https-actually-work/`

You can get your own HTTPS certificates for domains you own using Let's Encrypt for free. If you run a website, it's worth setting it up.

### 1.3.7 SSH Authentication

- *Encryption and SSH authentication.* By default, SSH is authenticated using usernames and passwords in the same way that you would log in to the physical machine.

  At some point, you're almost certain to hit a machine that has SSH configured with key authentication instead. This uses public and private keys to prove that the client is a valid and authorised user on the server. By default, SSH keys are RSA keys. You can choose which algorithm to generate, and/or add a passphrase to encrypt the SSH key. `ssh-keygen` is the program used to generate pairs of keys most of the time.

- *SSH Private keys.* You should treat your private SSH keys like passwords. Don't share them, they're called private keys for a reason. If someone has your private key, they can use it to log in to servers that will accept it unless the key is encrypted.

  It's very important to mention that the passphrase to decrypt the key isn't used to identify you to the server at all, all it does is decrypt the SSH key. The passphrase is never transmitted, and never leaves your system.

  Using tools like John the Ripper, you can attack an encrypted SSH key to attempt to find the passphrase, which highlights the importance of using a secure passphrase and keeping your private key private.

  When generating an SSH key to log in to a remote machine, you should generate the keys on your machine and then copy the public key over as this means the private key never exists on the target machine. For temporary keys generated for access to CTF boxes, this doesn't matter as much.

- *How do I use these keys?* The ~/.ssh folder is the default place to store these keys for OpenSSH. The `authorized_keys` (note the US English spelling) file in this directory holds public keys that are allowed to access the server if key authentication is enabled. By default on many distros, key authentication is enabled as it is more secure than using a password to authenticate. Normally for the root user, only key authentication is enabled.

In order to use a private SSH key, the permissions must be set up correctly otherwise your SSH client will ignore the file with a warning. Only the owner should be able to read or write to the private key (600 or stricter). `ssh -i keyNameGoesHere user@host` is how you specify a key for the standard Linux OpenSSH client.

- *Using SSH keys to get a better shell.* SSH keys are an excellent way to "upgrade" a reverse shell, assuming the user has login enabled (www-data normally does not, but regular users and root will). Leaving an SSH key in authorized_keys on a box can be a useful backdoor, and you don't need to deal with any of the issues of unstabilised reverse shells like Control-C or lack of tab completion.

### 1.3.8 Explaining Diffie Hellman Key Exchange

- *What is Key Exchange?* Key exchange allows 2 people/parties to establish a set of common cryptographic keys without an observer being able to get these keys. Generally, to establish common symmetric keys.

- *How does Diffie Hellman Key Exchange work?* Alice and Bob want to talk securely. They want to establish a common key, so they can use symmetric cryptography, but they don't want to use key exchange with asymmetric cryptography. This is where DH Key Exchange comes in.

  Alice and Bob both have secrets that they generate, let's call these A and B. They also have some common material that's public, let's call this C.

  We need to make some assumptions. Firstly, whenever we combine secrets/material it's impossible or very very difficult to separate. Secondly, the order that they're combined in doesn't matter.

  Alice and Bob will combine their secrets with the common material, and form AC and BC. They will then send these to each other, and combine that with their secrets to form two identical keys, both ABC. Now they can use this key to communicate.

An excellent video if you want a visual explanation is available here. DH Key Exchange is often used alongside RSA public key cryptography, to prove the identity of the person you're talking to with digital signing. This prevents someone from attacking the connection with a man-in-the-middle attack by pretending to be Bob.

### 1.3.9 PGP, GPG and AES

- *What is PGP?* PGP stands for Pretty Good Privacy. It's a software that implements encryption for encrypting files, performing digital signing and more.

- *What is GPG?* GnuPG or GPG is an Open Source implementation of PGP from the GNU project. You may need to use GPG to decrypt files in CTFs. With PGP/GPG, private keys can be protected with passphrases in a similar way to SSH private keys. If the key is passphrase protected, you can attempt to crack this passphrase using John The Ripper and gpg2john. The key provided in this task is not protected with a passphrase. The man page for GPG can be found online here.

- *What about AES?* AES, sometimes called Rijndael after its creators, stands for Advanced Encryption Standard. It was a replacement for DES which had short keys and other cryptographic flaws. AES and DES both operate on blocks of data (a block is a fixed size series of bits). AES is complicated to explain, and doesn't seem to come up as often. If you'd like to learn how it works, here's an excellent video from Computerphile.

### 1.3.10 The Future - Quantum Computers and Encryption

Quantum computers will soon be a problem for many types of encryption.

- *Asymmetric and Quantum.* While it's unlikely we'll have sufficiently powerful quantum computers until around 2030, once these exist encryption that uses RSA or Elliptical Curve Cryptography will be very fast to break. This is because quantum computers can very efficiently solve the mathematical problems that these algorithms rely on for their strength.

- *AES/DES and Quantum.* AES with 128 bit keys is also likely to be broken by quantum computers in the near future, but 256 bit AES can't be broken as easily. Triple DES is also vulnerable to attacks from quantum computers.

- *Current Recommendations.* The NSA recommends using RSA-3072 or better for asymmetric encryption and AES-256 or better for symmetric encryption. There are several competitions currently running for quantum safe cryptographic algorithms, and it's likely that we will have a new encryption standard before quantum computers become a threat to RSA and AES.

If you'd like to learn more about this, NIST has resources that detail what the issues with current encryption is and the currently proposed solutions for these. `https://doi.org/10.6028/NIST.IR.8105` I also recommend the book "Cryptography Apocalypse" By Roger A. Grimes, as this was my introduction to quantum computing and quantum safe cryptography.