Source: Bing Chat AI

Laboratory 3. Applications of Cryptography (I)

COMPUTER SCIENCE DEPARTMENT. UNIVERSITY OF OVIEDO

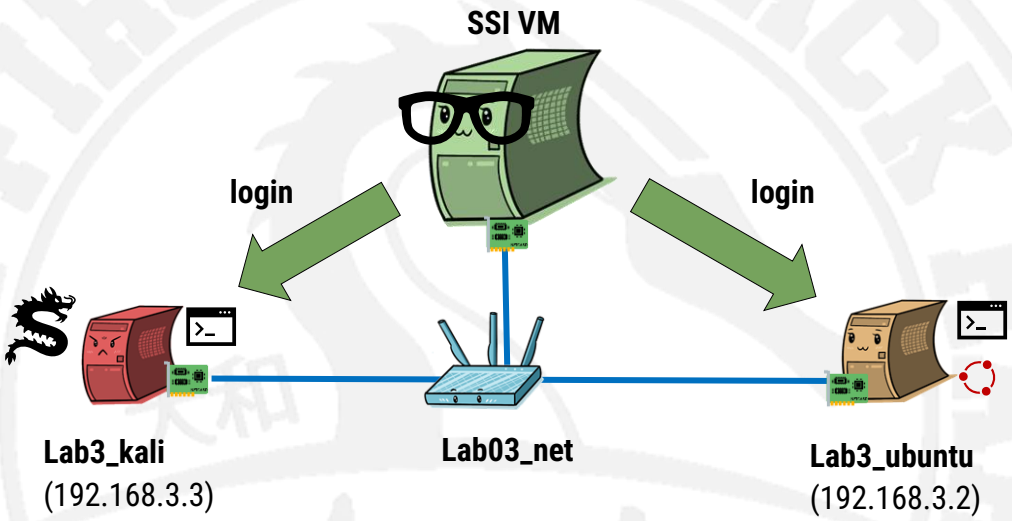
Computer Security | 2023 – 2024 (V4.0 *"S-81 Isaac Peral"*)

Content

Infrastructure of this laboratory

First, make sure you have the `ssi_labs` folder structure we provide for you within your virtual machine. This laboratory (`lab_03`) consists of an infrastructure composed of two containers communicated through a private LAN: a ***Kali* OS** and an ***Ubuntu OS***. Both have the *software* installed for this lab, though **the *John the Ripper part* can only be completed from the *Kali*** container due to the `openssl2john` tool. Not all activities require using all of them. Refer to the diagram below to understand the design of the infrastructure.

> 🔍 ***NOTE: The virtual machine has an offline copy of the CyberChef web tool available in the VM if you do not have Internet connectivity at any given time***



Block 1: Non-encryption techniques. Information concealment

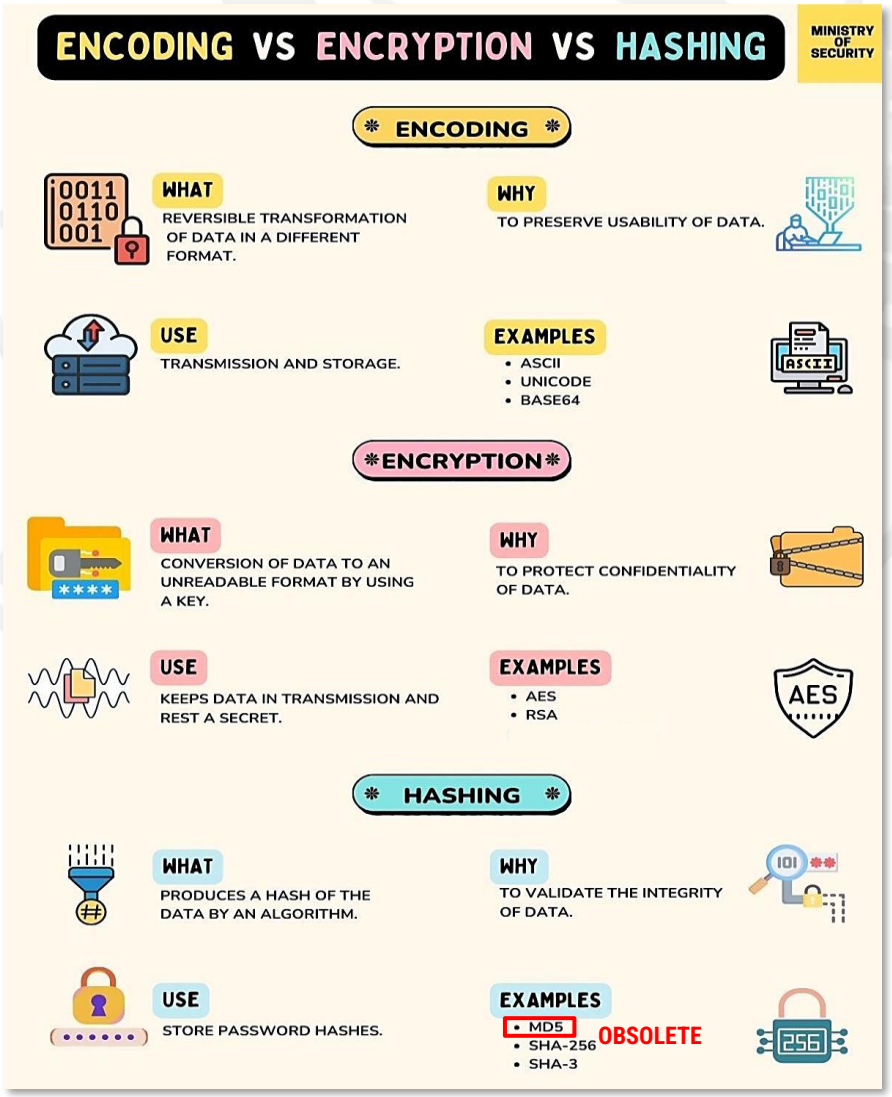
Exercise L3B1_ENCODE: Testing encoding mechanisms with *CyberChef*

Description of the activity / Practical application

It consists of **understanding what encoding is for** by trying to encode and decode your own information with the *CyberChef tool*: https://gchq.github.io/CyberChef/.

Expected results

You understand the basic handling of *CyberChef* by trying to encode and decode information. The goal of this activity is for you **to correctly distinguish the difference between encoding, encrypting, and creating a hash** (encrypting and creating a hash will be discussed later), modeling what appears in this image.



In addition, you can answer these questions:

- *Do you understand that encoding is not a security mechanism at all, but to obfuscate information because it does not have means to prevents anyone from decrypting it?*
- *Do you understand that the main use of encoding is to allow information to be transmitted when the transmission medium does not support certain characters of the original information?*
- *Have you understood how email allows you to transfer binary data when the protocol does not support it? Now that you know this, why do you think many email providers do not support attachments larger than 20Mb?*

Additional information required to do it

CyberChef (https://gchq.github.io/CyberChef/) is a reference website for performing all kinds of data transformations, from encoding to complete encryption/decryption algorithms, all via its web (online or offline) and with little effort.

🔍 ***Each transformation is a "Recipe", and multiple can be selected from the left panel (drag and drop) to transform ("Bake") the input you want. Familiarize yourself with Cyberchef usage, as it is one of the preferred tools used in CTF when you find encoded or encrypted data and have no other tools available to process the information***

This website has a LOT of recipes, so you usually find what you are looking for here, even if it is not related to encryption: for example, it **also allows you to view file metadata** (**Lab 4**). To practice with some examples, you can do the following.

- **Encode and decode any text** you want using ***Base64*** formatting
- Try **encoding in action in real life with the email** like this:
  - Find an image as small as you can (a few Kb) and send it to yourself via email.
  - Email protocol **does not support sending binary data**, so in order to send you the image it needs to **be encoded in text format** thanks to the MIME (https://es.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions) specifications
  - MIME **will encode the image in the *Base64*** format we saw, and the mail client automatically handles the encoding/decoding and viewing process, so you do not have to worry about that. But you can investigate what it does like this:
    - Open the message and look for the option that allows **you to download it** in `.eml` format:
      - In *Gmail*, go to the button with the three vertical dots and *"Download Message"*
      - In *Outlook*, go to the three horizontal dots button and "*Save As*"
    - Open the downloaded `.eml` file with *Notepad* or a similar text editor
      - Look for the `Content-Transfer-Encoding: base64` header. Just below *(in Outlook*) or two headers below (in *Gmail*) you will see **a large block of text** . This is the image you have sent encoded in *Base64*
      - Copy the entire block of text until the next blank line and paste it into *CyberChef* with the recipe that allows you to undo the *Base64 encoding*. Does it *output binary or text data?*
      - Download the *CyberChef* output and put the same extension on it as the original image. *Is what you have downloaded a copy of the image you sent?*

Exercise L3B1_STEGO: Using steganography with *steghide*

Description of the activity / Practical application

You can **hide secrets in images** without altering what the image shows

Expected results

This activity ends when **you manage to hide a message using steganography** in an image and extract it later. You can investigate what happens when the image with hidden information is uploaded to a social network.

It is known that when one uploads an image to a social network, email, messaging application, forum or similar, a lot of software manipulates that image to recode it, add information, compress it, or make transformations that, although it usually does not have visual effects, completely change its content. This can mean that if you have hidden a message with steganography on it, when you recode the image, that message is lost. The question is: Can you post an image on Twitter (or any other social network) so you can check if the steganographed content is lost or not?

In case a social network does not remove steganographed content: *Could you send hidden messages to someone in plain sight?*

Additional information required to do it

To do this exercise, we can choose any image from the Internet and embed any message with `steghide`, using a password that you can remember. Images must be in a format recognized by the application; Most `.jpg` should work. An example of a valid image is this: https://images.idgesg.net/images/article/2018/06/intel-8086-100760661-large.jpg

You have `steghide` installed in the *Kali container* of the **lab 3 infrastructure**. Use the appropriate `steghide` options to hide the message within the image (see *cheatsheet* below) and send the image with the hidden message to the *Ubuntu container*, which should extract the message.

To do this, remember that the `/shared` directory of each container can be used to share files between the virtual machine and each container, so it is very easy to share files between them. Use the appropriate options to extract the encrypted message.

steghide 0.5.1 Cheatsheet (ingenieriainformatica.uniovi.es)
Classic steganography tool

http://steghide.sourceforge.net/

GENERAL USAGE

`./steghide <first argument> <options>`

NOTES

The image you use must be of a compatible format (typically .jpg is)

OPTIONS

Possible First arguments (one is mandatory)	Embedding Options
embed, --embed: embed data	**-cf <filename>**: embed into the file <filename>
encinfo, --encinfo: display a list of supported encryption algorithms	**-cf, --coverfile**: select cover-file
extract, --extract: extract data	**-e <a>[<m>]|<m>[<a>]**: specify an encryption algorithm and/or mode
help, --help: display this usage information	**-e none**: do not encrypt data before embedding
info <filename>: display information about <filename>	**-e, --encryption**: select encryption parameters
info, --info: display information about a cover- or stego-file	**-ef <filename>**: embed the file <filename>
license, --license: display steghide's license	**-ef, --embedfile**: select file to be embedded
version, --version: display version information	**-f, --force**: overwrite existing files
Extracting Options	**-K, --nochecksum**: do not embed crc32 checksum of embedded data
-f, --force: overwrite existing files	**-N, --dontembedname**: do not embed the name of the original file
-p <passphrase>: use <passphrase> to extract data	**-p <passphrase>**: use <passphrase> to embed data
-p, --passphrase: specify passphrase	**-p, --passphrase**: specify passphrase
-q, --quiet: suppress information messages	**-q, --quiet**: suppress information messages
-sf <filename>: extract data from <filename>	**-sf <filename>**: write result to <filename> instead of cover-file
-sf, --stegofile: select stego file	**-sf, --stegofile**: select stego file
-v, --verbose: display detailed information	**-v, --verbose**: display detailed information
-xf <filename>: write the extracted data to <filename>	**-z <l>**: using level <l> (1 best speed...9 best compression)
-xf, --extractfile: select file name for extracted data	**-z, --compress**: compress data before embedding (default)
EXAMPLES	**-Z, --dontcompress**: do not compress data before embedding
To embed emb.txt in cvr.jpg: steghide embed -cf cvr.jpg -ef emb.txt *To extract embedded data from stg.jpg: steghide extract -sf stg.jpg*	**Options for the Info Command**
	-p, --passphrase: specify passphrase
	-p <passphrase>: use <passphrase> to get info about embedded data

Exercise L3B1_OBFUSCATE: Obfuscated code so it is hard to understand

Description of the activity / Practical application

You need **to obfuscate *JavaScript*** code to make it harder for someone to figure out how you have implemented certain functionality

Expected results

This activity ends when you have used all of the **obfuscation options mentioned** and you have verified that the output of its execution is still the same as the original one, even though the code itself looks very different.

Additional information required to do it

Code obfuscation is **NOT an encryption method**, and it transforms the code so that **even though it can still be executed, and its result is the same as the original**, it is more difficult to understand unless it is deobfuscated. In other words, both mechanisms have the same way of operating, although the result of the obfuscation is readable by anyone

🔍 *Readable does not mean understandable!* 😊

To quickly test how obfuscation works, we can use *JavaScript* as an example. Keep in mind that **most languages have their own obfuscation tools** tailored to their syntax, although the principles are the same. To see how obfuscation works, do the following:

- Go to this page and look at the *JavaScript* example it shows. You can save the output to a file for later verification: https://js.do/
- Copy the JavaScript *from the sample code* (**ONLY the JavaScript code block (not the `script` tags), and don't copy the HTML!**) and "pack it" (no options) with this online packager: http://dean.edwards.name/packer/
- Copy the packaged code into the `<script>` block of the first link page and verify that the result of the execution is the same.
- Repackage the code, **but enable the *Base62 encode* and *shrink variable options***
- Copy this heavily obfuscated code again and check that it works again on the first link web page.
- Go back over this *very obfuscated JavaScript and process it with the "embellishers" of* this website http://jsnice.org/ or this https://beautifier.io/. See what these tools do.
- Finally, process the heavily obfuscated code on this web page: https://obfuscator.io/, copy the results, and verify that the code still works again.
- *What if you try to "embellish" this last "terribly obfuscated" code? Can you get the original code back?*
- *Do you think the obfuscated code is smaller than the original and therefore takes up less space? If so, what do you think this can be used for?*

Block 2: Symmetric cryptography

Exercise L3B2_SYMCRYPT: Encrypt a file using a strong symmetric key algorithm

Description of the activity / Practical application

You need to encrypt a file so no one without its encryption password can read its contents using `openssl`, an encryption *software* that implements the same protocol that manages all the encrypted connections to a website that occur when browsing.

Expected results

This activity ends when you can **encrypt a file using a strong symmetric-key algorithm** so that you can send the file anywhere and only those who know the encryption password can read its contents. You can answer these questions:

- *Are you aware that AES-128 is a strong encryption algorithm, that it has not yet been broken, and that, therefore, if the password were good (something we will see later 😉) no one would be able to read what you are sending without the key (i.e. that this encryption mode is truly secure)?*
- *Are you sure that even if your message can be broken with a quantum computer, if you change the algorithm to an AES-256 it would no longer be possible?*

Additional information required to do it

Start working on the *Kali container* in the **Lab 3 infrastructure.** Create a new text file with an unencrypted text message to be sent to the recipient (e.g., your UO and name). To encrypt the message, **use** `openssl`, a program that is **already installed in the *Kali* container**. This image contains the instructions you need for it (https://cheatography.com/albertx/cheat-sheets/openssl/). **Use** "`password`" **as a key** (it is a bad practice, but here we do it on purpose for later). Select the **AES-128-ECB** encryption algorithm.

SYMMETRIC ENCRYPTION
List all supported symmetric encryption ciphers **openssl enc -list**
Encrypt a file using an ASCII encoded password provided and AES-128-ECB algorithm **openssl enc -aes-128-ecb -in cleartext.file -out ciphertext.file -pass pass:thisisthepassword**
Decrypt a file using AES-256-CBC and a keyfile **openssl enc -d -aes-256-cbc -in ciphertext.file -out cleartext.file -pass file:./key.file**
Encrypt a file using a specific encryption key (K) provided as hex digits **openssl enc -aes-128-ecb -in cleartext.file -out ciphertext.file -K 1881807b2d1b3d22f14e9ec52563d981 -nosalt**
Encrypt a file using ARIA 256 in CBC block cipher mode using a specified encryption key (K:256 bits) and initialization vector (iv:128 bits) **openssl enc -aria-256-cbc -in cleartext.file -out ciphertext.file -K f92d2e986b7a2a01683b4c40d0cbcf6feaa669ef2bb5ec3a25ce85d9548291c1 -iv 470bc29762496046882b61ecee68e07c -nosalt**
Encrypt a file using Camellia 192 algorithm in COUNTER block cipher mode with key and iv provided **openssl enc -camellia-192-ctr -in cleartext.file -out ciphertext.file -K 6c7a1b3487d28d3bf444186d7c529b48d67dd6206c7a1b34 -iv 470bc29762496046882b61ecee68e07c**

Is the encrypted output binary or text? If it is binary, *how could you convert it to text so you can send it into the body of an email or paste it anywhere that only supports text?* Do this using this image as a guide:

ENCODING / DECODING
Encoding a file using Base64 **openssl base64 -in file.data**
Encoding some text using Base64 **echo -n "some text" | openssl base64**
Base64 decode a file with output to another file **openssl base64 -d -in encoded.data -out decoded.data**

Exercise L3B2_SYMDECRYPT: Decrypt a file that uses symmetric encryption

Description of the activity / Practical application

You need to **obtain the contents of a previously encrypted file** if you know the corresponding key

Expected results

This activity ends when **a previously encrypted file is decrypted**, either in binary form or converted to text in the previous activity. You can answer these questions:

- *Can you decipher the Base64-converted ciphered file directly or do you need to convert it to binary format first?*

- *Do you think you can use symmetric encryption to send sensitive information through non-secure transmission methods,, such as the email?*
- *Do you understand why it is important to separate the way you send the key from the way you send the encrypted content?*

Additional information required to do it

In the previous exercise, from the *cheatsheet* provided, you were able to encrypt a file with the specified key. To decrypt it, `openssl` uses the same command as you did but adding `-d` after the enc parameter.

Exercise L3B2_CIPHMODES: Testing the ECB and CBC encryption modes of the AES robust symmetric encryption algorithm

Description of the activity / Practical application

You can practice with the **different encryption modes of the AES algorithm** and understand how this impacts the output depending on the type of data that is used as input.

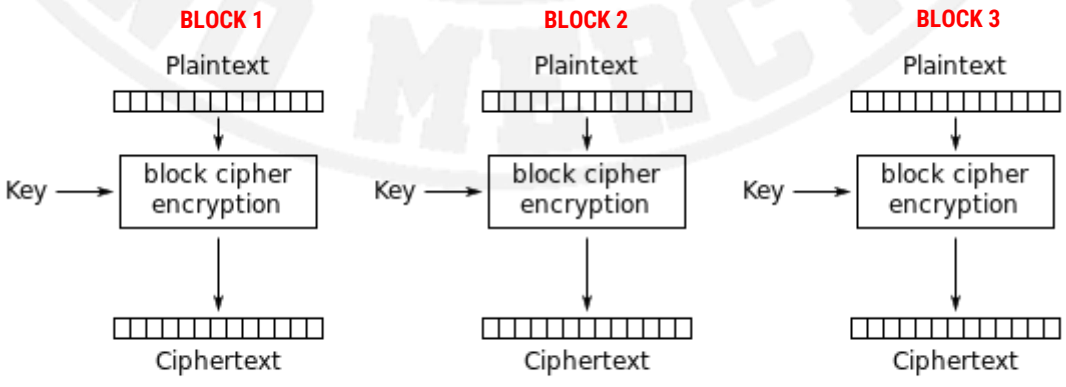
Expected results

You understand that **certain encryption modes are inappropriate for some types of information**, such as images, when blocks are individually encrypted, and therefore blocks encrypted with the same value (color) always have the same output. You are able to answer the questions that are asked during the course of the exercise.

Additional information required to do it

Practicing with ECB Mode

The ECB mode of the AES encryption algorithm is a dangerous encryption mode with some file types, such as images. The reason is that **it encrypts 128-bit blocks one by one independently,** with nothing to make how it encrypts a block depend on what was already encrypted previously, as happens in other modes. In other words, the blocks in the input are encrypted like this:



Electronic Codebook (ECB) mode encryption

🔍 ***Which, unfortunately, means that blocks that have the same value will return the same result when they finish being encrypted.***

If the input is a block that represents a color, the output will be a block that will represent a different color if the viewer of the encrypted file finds out or guesses that an image has been encrypted . Let's try it out by following these instructions:

- Download a **PNG image** that has a white background and a series of different colors to highlight the effect we are talking about. In this URL you have a few: https://commons.wikimedia.org/wiki/Category:PNG_files. This can be one: https://commons.wikimedia.org/wiki/File:Eslogan_Oficial.png
- **Convert the downloaded image to a** `.ppm` (*Portable Pixmap Format)* file *to* be able to easily separate the header and body of the image, and make the body a pixel map. This will allow us to encrypt only the body of the image, and then be able to see an image which only has the body encrypted. To convert an image we can install the following library: `sudo apt install graphicsmagick-imagemagick-compat`. Once the library is installed, this conversion can be done with `convert <image name>.png <image name>.ppm`
- **Now split the image into two**, header and body:

```
head -n 3 <image name>.ppm > header.txt
tail -n +4 <image name>.ppm > body.bin
```

- **Encrypt only the** `body.bin` file the same way as in the previous exercise
- When you have the encrypted body, **add the header you extracted earlier** to re-interpret the file as a `.ppm` image. Here is how it can be done:

```
cat header.txt <file with encrypted body> > <union name>.ppm
```

- **NOTE**: A good tool for opening/editing images is GIMP. If your host (Windows in the school labs) does not open the image once you place them in the shared folder of the VM, you can install it in our VM with `sudo apt install gimp`

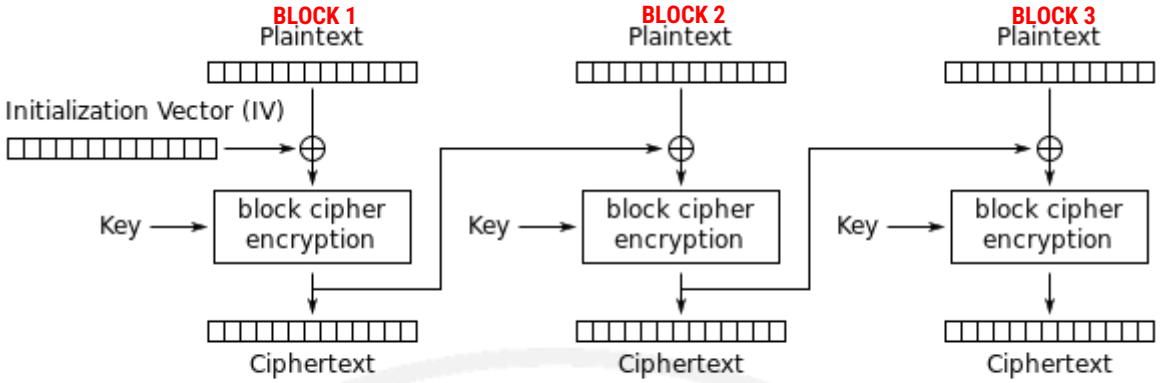
What happens now when you open this image? Is the data encrypted? Still, *can you see the color patterns? Do you think you can distinguish the image? Why do you think this is dangerous and can end up in an information leak?*

Practicing with CBC Mode

In theory we have seen other AES encryption modes, such as CBC mode. In that mode, an encrypted block **is used as input** when encrypting the next one. For the first of all blocks, an ***Initialization Vector*** (IV) is used,

🔍 ***An IV is a unique random number that is usually the same size as the block to be encrypted***

The scheme would look something like this:



Cipher Block Chaining (CBC) mode encryption

This ***Initialization Vector*** (IV) can be generated with the `rand` parameter of *OpenSSL*. Since we are using a 128-bit AES and each character is 8 bits, we will need a 16-character IV (16 x 8 = 128). Ex.: `openssl rand -hex 16`

Now encrypt the body of the image as before, but with the **AES-128-CBC** algorithm and adding the initialization vector you have generated as a parameter of the `-iv` option that you must now add to the call to the encryption command. Once this is done, rejoin the header and open the resulting file. *Are the color patterns now distinguishable or not? Which mode is better for encrypting images then?*

Exercise L3B2_DISKCIPH: Disk encryption

Description of the activity / Practical application

You need **to create a directory whose contents are automatically encrypted when copied into it**, so that no one other than your user can access them.

Expected results

This activity ends **when you can encrypt a folder** and verify that the contents are actually encrypted when you unmount it. You can answer these questions:

- *Do you understand why this type of encryption is often referred to as "data-at-rest encryption"?*
- *Do you think it would be a good idea to do what we've seen in this exercise on a USB stick that has sensitive data?*

Additional information required to do it

🔍 ***Encrypting file systems, directories, and disks is one of the most common applications of symmetric-key algorithms. When Ubuntu is installed it offers to encrypt our home directory as an option (this also encrypts the `/swap` partition). Even if we choose not to do so at that time, it is possible to create encrypted disks later: https://www.howtogeek.com/116032/how-to-encrypt-your-home-folder-after-installing-ubuntu/***

In this lab we are not going to do full-disk encryption, but a faster version of the same application of cryptography: encrypting only a folder (and its contents). To do this, follow this procedure **on the VM** (**this does not work inside the infrastructure containers!**):

- Install the disk encryption software: `sudo apt install ecryptfs-utils`
- Create a folder to encrypt in your home directory. If you are using an existing one, make sure it is empty (if not, back up its contents... just in case 😊)
- Encrypt the folder by mounting it under the `ecryptfs` file system: `mount -t ecryptfs <your folder path> <your folder path>` (yes, the mount point and the folder are the same. For example, `mount -t ecryptfs /home/ssiuser/secure /home/ssiuser/secure`). You will need `root` permissions
- Choose an appropriate symmetric encryption algorithm according to the theory.
- Choose `32` bytes for the key.
- Choose `No` in the *plaintext passthrough option*.
- Choose `No` to encrypt filenames (although this is a minor detail)
- Answer `Yes` to the following two questions (appear if this is the first use of disk encryption in the operating system, which it should)
- Verify with the `mount` command that the mount point is mounted as encrypted.
- Copy or create a text file in the encrypted folder. *Can you read it?*
- Unmount the mount point encrypted with `sudo umount /home/secure`. *Can you read their content now?* (Please exit the folder before disassembling it!)
- Mount it again it with the same parameters. *Can you re-read the content?*

Block 3. Introduction to offline password cracking

Exercise L3B3_SSLCRACK: *John the Ripper* to break OpenSSL

Description of the activity / Practical application

You need **to crack the password of an openssl-encrypted file** that uses a common password

Expected results

This activity ends **when the password hash of the encrypted message** with `openssl` in a previous exercise **is decrypted** (that is why we insist on using "`password`" as the password 😊). You can answer the following questions:

- *Do you understand that cracking a hash is just a repetitive process until you find the input that generates the correct hash?*
- *Have you understood that a hash cracker works with hashes and not another type of encrypted content?*
- *Do you understand that, luckily, there are many tools capable of extracting or calculating the hash of your password from encrypted files that have been generated with a large number of algorithms?*
- *Do you understand the difference between the process of hashing a password candidate and generating many password candidates, along with the different existing methods of generating possible passwords?*

Additional information required to do it

🔍 ***You've probably seen in movies the "art" of cracking passwords in a variety of ways: they are all lies 😊. The main reason is that a serious business does not save passwords as you enter them ("plaintext") but, as we saw in the theory, they are stored in the form of a hash (KDF).***

As we saw in the theory, **a hash is not reversible** (and even less "in chunks"), so everything you may have seen on TV about revealing a password character by character **is also a lie**.

🔍 ***Also, can you imagine trying to log into a Gmail account that you do not remember the password for and the service telling you "look, you have the two letters at the end wrong"?***

It is time to break myths and stereotypes and find out how it has actually done!

Have a good understanding of what "cracking a password" is and what you are cracking

John the Ripper is one of the most popular password hash cracking tools (along with `hashcat`, `hydra`, etc.) but it is just that: **a tool that cracks hashes**. *What does it mean?* That it is capable of generating **a huge number of combinations of characters per second**, hashing them according to the algorithm and other parameters that we specify (those used in the hashes file we are going to work on), and comparing the

hashing result with the hashes that the file has. If it matches, since `john` knows the original character combination used, it will have figured out the password corresponding to the hash. That is what is called "cracking a password," no more, no less 😊.

> 🔍 ***Think that this makes total sense; since if a hash is not reversible, what you do is repeat the hashing process with different inputs until you find the "good" one. If you are thinking that this sounds a lot like a toddler "frying" you with questions until you say yes to something, you are not too far off the mark*** 😊

The consequence of understanding how this is done in real life leads to another conclusion: `john` **cannot work with directly encrypted files**. That is, you cannot open an encrypted file and directly test *passwords* against it (again like in the movies!). Luckily, this can be easily resolved. There are **tools to extract the password hash from an encrypted file** if you know the program and the algorithm that was used for it.

> 🔍 ***In our case we know it directly, but in "real life" you can try several that seem closer to you, search for or extract information from someone via social engineering or use an encryption algorithm detector (https://www.dcode.fr/cipher-identifier), but this is already material for a cryptography-only course.***

In the `john` installation you have in the *Kali* container, you can see how many john tools there are by typing: `locate *2john*`. The full list is also here: https://miloserdov.org/?p=5008&PageSpeed=noscript. Here you can see that the one we need in our case is `openssl2john`:

openssl2john.py

This utility helps in cracking files encrypted using "**openssl enc**" command.

```
openssl aes-256-cbc -in secret.txt -out secret.txt.enc
openssl aes-256-cbc -a -in secret.txt -out secret.txt.enc
openssl enc -aes-256-cbc -in secret.txt -out secret.txt.enc
```

Usage:

```
./openssl2john.py [-c cipher] [-m md] [-p plaintext] [-a ascii_pct] <OpenSSL encrypted files>
```

cipher can be: 0 => aes-256-cbc, 1 => aes-128-cbc

md can be: 0 => md5, 1 => sha1, 2 => sha256

ascii_pct: minimum ascii percent (1-100) on decrypted output (ignored if plaintext present)

OpenSSL 1.1.0e uses aes-256-cbc with sha256.

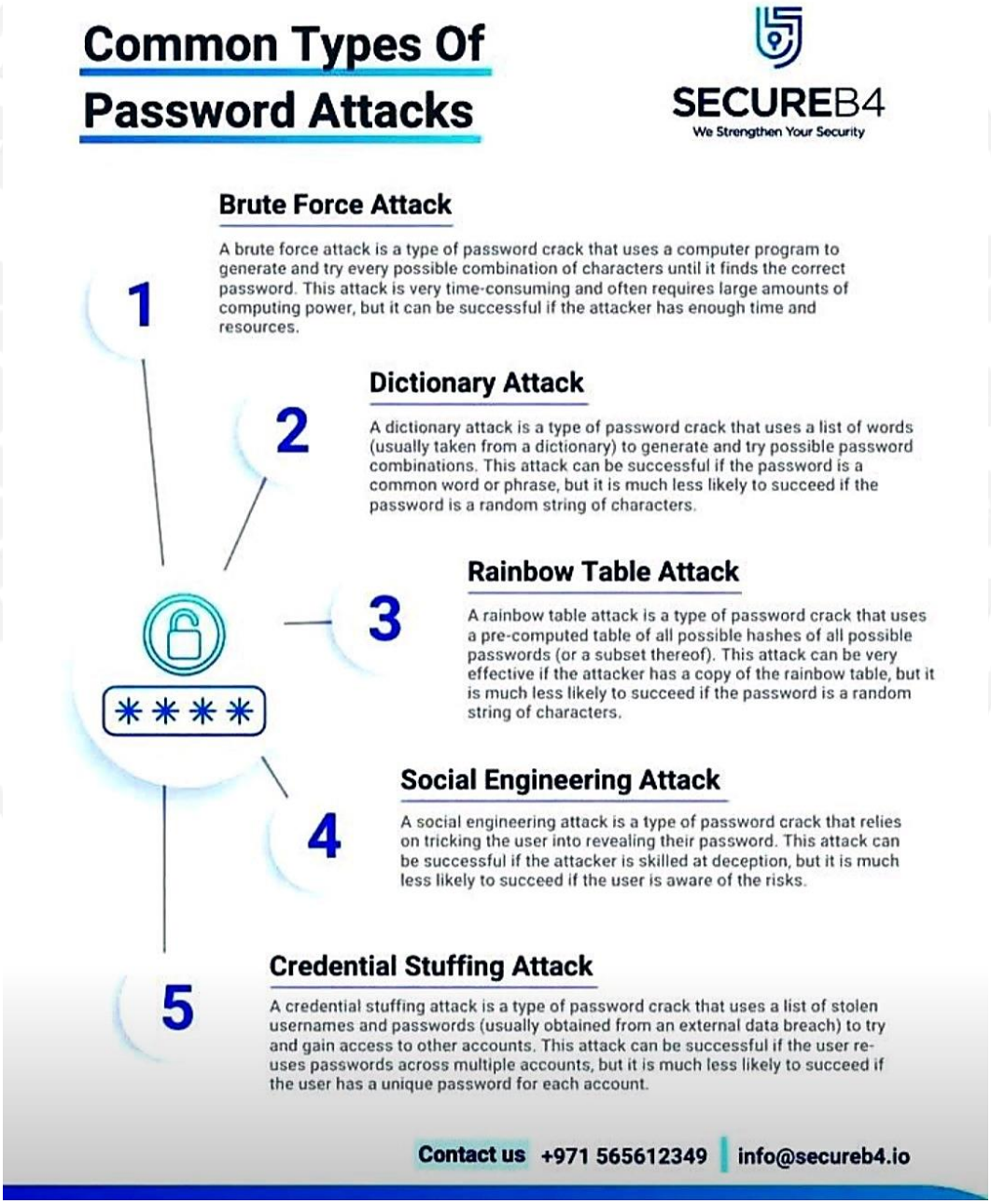
You already know how to hash your possible passwords, but…how do you generate a huge number of passwords to test them?

Now you know how to convert your candidate passwords **into a format that can be compared to the hashes you have** and want to crack. But one last problem remains to be solved: *How do I generate a huge number of possible passwords for* `john` *to test?* There are three basic techniques:

- **Brute force**: which tries to guess the password **by iterating sequentially** through every possible combination of letters, numbers, and special characters in a vocabulary of characters that we set in advance (or any possible combination if we do not put one!). This is a tremendously slow, but effective process... if you have enough time to wait for it to finish 😊. If the password is too strong, it may end around the time the universe is expected to collapse on itself :P

- **Dictionary** (also known as ***word lists***)*:* This attack uses a file containing lists of common passwords (usually extracted from various leaks of real password files) to guess a given password. It can be useful in CTFs, but it could be difficult to get real-world success if users follow a strong password policy.
- ***Rainbow tables***: These are a series of **pre-calculated hashes**. The idea is that these tables include all hashes for a given algorithm. So, instead of cracking the hash/password/etc., a lookup of the hash is performed on the table. This requires considerable processing power to achieve, but it could be more effective. We won't see them in this course, but if you're interested, you can check out more information about them here: https://en.wikipedia.org/wiki/Rainbow_table .

Actually, you can add two more techniques, but they are applied more against people than against files, as you can see in the following image 😊. These two are more related to the **Group Work** of the course.



In this exercise we are going to use **a dictionary-based attack** to break the password of our previously encrypted message with `openssl` by following these steps:

- Use `john` as a **password cracking tool,** provided in the **Lab 3 infrastructure** (*Kali* container), but installable with `sudo apt install john`
- We need **a dictionary,** ***and we will use a very popular compromised password*** dictionary called `rockyou.txt` that is available in the `/wordlist` directory of the *Kali* container. If you need to do something like this in the future, you can use larger or specialized dictionaries, like the ones here: https://ns2.elhacker.net/wordlists/
- **Create a file to be decrypted** in the same way as you did in exercise **L3B2_SYMCRIPT** (with the password "`password`") but **using the AES-128-CBC format**. As we said, you must pre-process it with the `openssl2john` utility installed in the *Kali container* of the **Lab 3 infrastructure** (the default *Ubuntu* `john` package does not include it). Store the output in a file and remember well the format of the hash it extracts that you have specified in the `-m` parameter.

🔍 ***NOTE: It is possible to do this exercise using both the AES-128-CBC and the AES-256-CBC encryption algorithms. In both cases, the hash format for the output of*** `openssl2john` ***should be SHA256 (*** `-m 2`***). However, bear in mind that the*** `-c` ***parameter of*** `openssl2john` ***changes depending on the encryption algorithm you used. Please, DO NOT use AES-128-ECB or AES-256-ECB, as*** `openssl2john` ***does not support these operational modes of AES.***

Use this *cheatsheet* from `john` with the list of words provided against the above file to crack the password of the openssl-encrypted message. **Make sure you pass the hash format of the** file, which is the one you put in `openssl2john`: https://pentestmonkey.net/cheat-sheet/john-the-ripper-hash-formats

John the Ripper 1.9.0-jumbo Cheatsheet (ingenieriainformatica.uniovi.es)
Offline password cracking tool

https://github.com/openwall/john

GENERAL USAGE

john [OPTIONS] [PASSWORD-FILES]

NOTES

John cracking modes:	https://www.openwall.com/john/doc/MODES.shtml
John FAQ:	https://www.openwall.com/john/doc/FAQ.shtml
John wordlist rules:	https://www.openwall.com/john/doc/RULES.shtml

OPTIONS

Cracking modes

--external=MODE: external MODE or word filter
--incremental[=MODE]: "incremental" mode [using section MODE]
--markov[=OPTIONS]: "Markov" mode (see doc/MARKOV)
--single[=SECTION[,..]]: "single crack" mode, using default or named rules
--single=:rule[,..]: same, using "immediate" rule(s)
--wordlist[=FILE]: --stdin wordlist mode, read words from FILE or stdin

EXAMPLES

Please consult a full example set on:
https://www.openwall.com/john/doc/EXAMPLES.shtml

```
ssiuser@ubuntussi:~$ john -wordlist:myDict passwd.db
Created directory: /home/ssiuser/.john
Loaded 1 password hash (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
test123...       (ssiuser)
1g 0:00:00:00 100% 2.173g/s 417.3p/s 417.3c/s 417.3C/s test096.....test191...
Use the "--show" option to display all of the cracked passwords reliably
Session completed
ssiuser@ubuntussi:~$
```

Other options

--costs=[-]C[:M][,...]: load salts with[out] cost value Cn [to Mn]. For tunable cost parameters, see doc/OPTIONS
--dupe-suppression: suppress all dupes (duplicates) in wordlist (and force preload)
--encoding=NAME: input encoding (eg. UTF-8, ISO-8859-1). See also doc/ENCODINGS and --list=hidden-options.
--fork=N: fork N processes
--format=NAME: force hash of type NAME. the supported formats can be seen with --list=formats and --list=subformats
--groups=[-]GID[,...]: load users [not] of this (these) group(s) only
--list=WHAT: list capabilities, see --list=help or doc/OPTIONS
--loopback[=FILE]: like --wordlist, but extract words from a .pot file
--make-charset=FILE: make a charset file. It will be overwritten
--mask[=MASK]: mask mode using MASK (or default from john.conf)
--node=MIN[-MAX]/TOTAL: this node's number range out of TOTAL count
--pipe: like --stdin, but bulk reads, and allows rules
--pot=NAME: pot FILE to use
--prince[=FILE]: PRINCE mode, read words from FILE
--restore[=NAME]: restore an interrupted session [called NAME]
--rules[=SECTION[,..]]: enable word mangling rules (for wordlist or PRINCE modes), using default or named rules
--rules=:rule[;..]]: same, using "immediate" rule(s)
--rules-stack=:rule[;..]: same, using "immediate" rule(s)
--rules-stack=SECTION[,..]: stacked rules, applied after regular rules or to modes that otherwise don't support rules
--salts=[-]COUNT[:MAX]: load salts with[out] COUNT [to MAX] hashes
--save-memory=LEVEL: enable memory saving, at LEVEL 1..3
--session=NAME: give a new session the NAME
--shells=[-]SHELL[,..]: load users with[out] this (these) shell(s) only
--show[=left]: show cracked passwords [if =left, then uncracked]
--status[=NAME]: print status of a session [called NAME]
--stdout[=LENGTH]: just output candidate passwords [cut at LENGTH]
--subsets[=CHARSET]: "subsets" mode (see doc/SUBSETS)
--test[=TIME]: run tests and benchmarks for TIME seconds each
--users=[-]LOGIN|UID[,..]: [do not] load this (these) user(s) only

Exercise L3B3_PASS: John the Ripper + user passwords + crunch wordlist generator

Description of the activity / Practical application

You need **to crack the hash of a user's password** with a custom word dictionary

Expected results

This activity is considered complete when the password of the `ssiuser` user is decrypted in the provided sample `passwd` and `shadow` files (`/wordlist` directory of the **Lab 3 infrastructure** containers)

Additional information required to do it

We are going to use `john` again, but for a different purpose. `rockyou.txt` is a good dictionary of leaked passwords, but sometimes using it backfires, because somehow we know in advance that user passwords follow a pattern, a rule, or have a certain shape. In that case, generating a custom word dictionary, tailored to those assumptions, **is the smartest approach**, and this is precisely the goal of this exercise. However, instead of decrypting a file, we are going to show you how leaked *Linux* user files and passwords are routinely broken. Apart from `john`, to do this you need:

- **A tool to** combine the filtered `passwd` that has the usernames and the file with the hashes of their keys (`shadow`) as unshadow (already installed in the **Lab 3 infrastructure**). This tool reads the `passwd` and `shadow` files passed as the first and second parameters and joins them with a combined format usable by `john`, which can be saved in a third file.
- **A word list generator,** such as `crunch` (already installed in the **Lab 3 infrastructure**, but installing it is very easy with `sudo apt install crunch`)
- **Before proceeding**, first read the *cheatsheet* and follow the instructions below.

`crunch` allows you **to generate a list of words** following the word patterns you specify if you know or suspect part of a user's password, the shape of the key (numbers + letters, lowercase only...), or any data that can save you from trying impossible passwords. Getting into the `crunch` pattern generation syntax is beyond the scope of this lab, but to make the process easier you can use the *cheatsheet* and these tips:

- The `%` symbol stands for "any number"
- The `@` symbol stands for "any lowercase letter"
- If you type a string of characters, it will be treated as a constant (the generated keys will have that string in the position you indicate).

The password to be guessed has a very easy way to generate. It starts with "`test`" and ends with "`...`". We also know that the middle part is **three numbers or three lowercase** letters (always 3, never mixing numbers and letters). Pattern lengths for `crunch` are always number of characters + 1 (C strings always add '\0' at the end, so the length should also be one character more than the actual length we need to use).

crunch 3.6 Cheatsheet (ingenieriainformatica.uniovi.es)

Tool that generates wordlists from a character set

https://tools.kali.org/password-attacks/crunch

GENERAL USAGE

```
crunch <min-len> <max-len> [<charset string>] [options]
```

NOTES

This is a reference tool to generate wordlists to bruteforce passwords and other similar requirements. Crunch can create a wordlist based on criteria you specify. The output from crunch can be sent to the screen, file, or to another program

OPTIONS

Required parameters

charset string: You may specify character sets for crunch to use on the command line or if you leave it blank crunch will use the default character sets. The order MUST BE lower case characters, upper case characters, numbers, and then symbols. If you don't follow this order you will not get the results you want. You MUST specify either values for the character type or a plus sign. NOTE: If you want to include the space character in your character set you must escape it using the \ character or enclose your character set in quotes i.e. "abc ". See the examples 3, 11, 12, and 13 for examples.

max-len: The maximum length string you want crunch to end at. This option is required even for parameters that won't use the value.

min-len: The minimum length string you want crunch to start at. This option is required even for parameters that won't use the value.

Options

-b number[type]: Specifies the size of the output file, only works if -o START is used, i.e.: 60MB The output files will be in the format of starting letter-ending letter for example: ./crunch 4 5 -b 20mib -o START will generate 4 files: aaaa-gvfed.txt, gvfee-ombqy.txt, ombqz-wcydt.txt, wcydu-zzzzz.txt valid values for type are kb, mb, gb, kib, mib, and gib. The first three types are based on 1000 while the last three types are based on 1024. NOTE There is no space between the number and type. For example 500mb is correct 500 mb is NOT correct.

-c number: Specifies the number of lines to write to output file, only works if -o START is used, i.e.: 60. The output files will be in the format of starting letter-ending letter for example: ./crunch 1 1 -f /pentest/password/crunch/charset.lst mixalpha-numeric-all-space -o START -c 60 will result in 2 files: a-7.txt and 8-\ .txt The reason for the slash in the second filename is the ending character is space and ls has to escape it to print it. Yes you will need to put in the \ when specifying the filename because the last character is a space.

-d numbersymbol: Limits the number of duplicate characters. -d 2@ limits the lower case alphabet to output like aab and aac. aaa would not be generated as that is 3 consecutive letters of a. The format is number then symbol where number is the maximum number of consecutive characters and symbol is the symbol of the the character set you want to limit i.e. @,%^ See examples 17-19.

-e string: Specifies when crunch should stop early

-f /path/to/charset.lst charset-name: Specifies a character set from the charset.lst

-i: Inverts the output so instead of aaa,aab,aac,aad, etc you get aaa,baa,caa,daa,aba,bba, etc

-l: When you use the -t option this option tells crunch which symbols should be treated as literals. This will allow you to use the placeholders as letters in the pattern. The -l option should be the same length as the -t option. See example 15.

-m: Merged with -p. Please use -p instead.

-o wordlist.txt: Specifies the file to write the output to, eg: wordlist.txt

-p charset OR -p word1 word2 …: Tells crunch to generate words that don't have repeating characters. By default crunch will generate a wordlist size of #of_chars_in_charset ^ max_length. This option will instead generate #of_chars_in_charset!. The ! stands for factorial. For example say the charset is abc and max length is 4.. Crunch will by default generate 3^4 = 81 words. This option will instead generate 3! = 3x2x1 = 6 words (abc, acb, bac, bca, cab, cba). THIS MUST BE THE LAST OPTION! This option CANNOT be used with -s and it ignores min and max length however you must still specify two numbers.

-q filename.txt: Tells crunch to read filename.txt and permute what is read. This is like the -p option except it gets the input from filename.txt.

-r: Tells crunch to resume generate words from where it left off. -r only works if you use -o. You must use the same command as the original command used to generate the words. The only exception to this is the -s option. If your original command used the -s option you MUST remove it before you resume the session. Just add -r to the end of the original command.

-s startblock: Specifies a starting string, eg: 03god22fs

-t @,%^: Specifies a pattern, eg: @@god@@@@ where the only the @'s, ,'s, %'s, and ^'s will change; @ will insert lower case characters; , will insert upper case characters; % will insert numbers; ^ will insert symbols

-u: The -u option disables the printpercentage thread. This should be the last option.

-z gzip, bzip2, lzma, and 7z: Compresses the output from the -o option. Valid parameters are gzip, bzip2, lzma, and 7z. gzip is the fastest but the compression is minimal. bzip2 is a little slower than gzip but has better compression. 7z is slowest but has the best compression.

```
ssluser@ubuntussi:~$ crunch 10 10 -t test%%%... -o myDict
Crunch will now generate the following amount of data: 11000 bytes
0 MB
0 GB
0 TB
0 PB
Crunch will now generate the following number of lines: 1000

crunch: 100% completed generating output
```

EXAMPLES

Example 1: crunch 1 8 - crunch will display a wordlist that starts at a and ends at zzzzzzzz

Example 2: crunch 1 6 abcdefg - crunch will display a wordlist using the character set abcdefg that starts at a and ends at gggggg

Example 3: crunch 1 6 abcdefg - there is a space at the end of the character string. In order for crunch to use the space you will need to escape it using the \ character. In this example you could also put quotes around the letters and not need the \, i.e. "abcdefg ". Crunch will display a wordlist using the character set abcdefg that starts at a and ends at (6 spaces)

Example 4: crunch 1 8 -f charset.lst mixalpha-numeric-all-space -o wordlist.txt - crunch will use the mixalpha-numeric-all-space character set from charset.lst and will write the wordlist to a file named wordlist.txt. The file will start with a and end with " "

Example 5: crunch 8 8 -f charset.lst mixalpha-numeric-all-space -o wordlist.txt -t @@dog@@@ -s cbdogaaa - crunch should generate a 8 character wordlist using the mixalpha-number-all-space character set from charset.lst and will write the wordlist to a file named wordlist.txt. The file will start at cbdogaaa and end at " dog "

Example 6: crunch 2 3 -f charset.lst ualpha -s BB - crunch with start generating a wordlist at BB and end with ZZZ. This is useful if you have to stop generating a wordlist in the middle. Just do a tail wordlist.txt and set the -s parameter to the next word in the sequence. Be sure to rename the original wordlist BEFORE you begin as crunch will overwrite the existing wordlist.

Example 7: crunch 4 5 -p abc - The numbers aren't processed but are needed. crunch will generate abc, acb, bac, bca, cab, cba.

Example 8: crunch 4 5 -p dog cat bird - The numbers aren't processed but are needed. crunch will generate birdcatdog, birddogcat, catbirddog, catdogbird, dogbirdcat, dogcatbird.

Example 9: crunch 1 5 -o START -c 6000 -z bzip2 - crunch will generate bzip2 compressed files with each file containing 6000 words. The filenames of the compressed files will be first_word-last_word.txt.bz2

Example 10: crunch 4 5 -b 20mib -o START - will generate 4 files: aaaa-gvfed.txt, gvfee-ombqy.txt, ombqz-wcydt.txt, wcydu-zzzzz.txt the first three files are 20MBs (real power of 2 MegaBytes) and the last file is 11MB.

Example 11: crunch 3 3 abc + 123 !@# -t @%^ - will generate a 3 character long word with a character as the first character, and number as the second character, and a symbol for the third character. The order in which you specify the characters you want is important. You must specify the order as lower case character, upper case character, number, and symbol. If you aren't going to use a particular character set you use a plus sign as a placeholder. As you can see I am not using the upper case character set so I am using the plus sign placeholder. The above will start at a1! and end at c3#

Example 12: crunch 3 3 abc + 123 !@# -t ^%@ - will generate 3 character words starting with !1a and ending with #3c

Example 13: crunch 4 4 + + 123 + -t %%@^ - the plus sign (+) is a place holder so you can specify a character set for the character type. crunch will use the default character set for the character type when crunch encounters a + (plus sign) on the command line. You must either specify values for each character type or use the plus sign. I.E. if you have two characters types you MUST either specify values for each type or use a plus sign. So in this example the character sets will be:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ123
!@#$%^&*()-_+=~`[]{}|\:;"'<>,.?/
```

there is a space at the end of the above string the output will start at 11a! and end at "33z ". The quotes show the space at the end of the string.

Example 14: crunch 5 5 -t ddd@@ -o j -p dog cat bird - any character other than one of the following: @,%^ is the placeholder for the words to permute. The @,%^ symbols have the same function as -t. If you want to use @,%^ in your output you can use the -l option to specify which character you want crunch to treat as a literal. So the results are

```
birdcatdogaa
birdcatdogab
birdcatdogac
<skipped>
dogcatbirdzy
dogcatbirdzz
```

Example 15: crunch 7 7 -t p@ss,%^ -l a@aaaaa - crunch will now treat the @ symbol as a literal character and not replace the character with a uppercase letter. this will generate

```
p@ssA0!
p@ssA0@
p@ssA0#
p@ssA0$
<skipped>
p@ssZ9
```

Example 16: crunch 5 5 -s @4#S2 -t @%^,2 -e @8 Q2 -l @dddd -b 10KB -o START - crunch will generate 5 character strings starting with @4#S2 and ending at @8 Q2. The output will be broken into 10KB sized files named for the files starting and ending strings.

Example 17: crunch 5 5 -d 2@ -t @@@%% - crunch will generate 5 character strings staring with aab00 and ending at zzy99. Notice that aaa and zzz are not present.

Example 18: crunch 10 10 -t @@@^%%%%^^ -d 2@ -d 3% -b 20mb -o START - crunch will generate 10 character strings starting with aab!0001!! and ending at zzy 9998. The output will be written to 20mb files.

Example 19: crunch 8 8 -d 2@ - crunch will generate 8 characters that limit the same number of lower case characters to 2. Crunch will start at aabaabaa and end at zzyzzyzz.

Example 20: crunch 4 4 -f unicode_test.lst japanese -t @@%% -l @xdd - crunch will load some Japanese characters from the unicode_test character set file. The output will start at @日00 and end at @語99.

Exercise L3B3_BRUTE: "Pure brute force" with *John the Ripper*

Description of the activity / Practical application

You need to crack users' keys **using pure brute force**, because you do not have any clue about what those users' keys might look like

Expected results

This activity ends when **the hash of the** `Android` user' s password **is cracked**.

Additional information required to do it

If we have no idea about what the password is like, and no common password has worked for us (e.g. from the dictionary in the previous exercise), we can try using pure brute force with `john` against the previous file, with all the defaults, and see what happens

Maybe we will get lucky... or maybe we should wait days/weeks/years/...

Note that `john` applies default permutations to the words he generates and has multiple crack models. To test this, follow these rules:

- Delete the `john.pot` file that is in the hidden `.john` directory of the folder you are working on. If you do not, and `john` has already cracked a password in a previous run, john will inform you that the password has already been cracked and will not compute anything, even if there are some passwords left uncracked! 😳 .
- Run `john` in incremental mode, specifying an additional parameter `--max-length=<number>` to limit the number of character permutations `john` can use and thus make the process complete in a reasonable amount of time . Use the *cheatsheet* above to specify the appropriate options to crack only the `Android` user's password, as we know that this user **uses a very short password** (the password length is equal to or less than 4 characters).

Finally, we give you this documentation as a reference if you are curious, but it is not really necessary to finish this activity.

- John's Cracking Modes: https://www.openwall.com/john/doc/MODES.shtml
- John's FAQs: https://www.openwall.com/john/doc/FAQ.shtml
- John's examples: https://www.openwall.com/john/doc/EXAMPLES.shtml
- John's Word List Rules: https://www.openwall.com/john/doc/RULES.shtml
- More information: https://manualdehacker.com/john-the-ripper-cracking-passwords/

✍ Badges & Self-Assessment

NOTE: You have a version of this badge table in editable format available on the *Virtual Campus*. You can use this file to create a document in any format you want and take extended notes of your activities to create a log of what you have done. Remember that the material you make can be taken for laboratory tests.

Badge Level	Unlocked when	Unlocked?
	You can encode any information in *Base64* format or similar thanks to *CyberChef*	
	You are able to encrypt any file using a strong symmetric encryption algorithm. Answer the following question: Is the *content of the encrypted file binary or plaintext?*	
	You are able to convert a *Base64* encoded file to an equivalent binary one	
	You can answer the following questions: *Can you send a Base64 encoded file via email? Or via WhatsApp, etc. (excluding length limits)?*	
	You are able to decrypt an encrypted file with a symmetric algorithm	
	You can use `crunch` to generate dictionaries of simple words of limited length	
	You can answer the following question: *Do you now understand why social engineering is so popular when it comes to getting someone's passwords considering how much a typical cracking process costs?*)	
	You understand how to encrypt folders and their contents in the file system. Answer these questions: *What type of attack does it hinder? Is the encryption transparent (can it be used without entering keys when working with encrypted files)?*	
	You understand the process of hiding information in an image file. You can answer this question: *does the text file introduced into the image using steganography make it visually different from the original?*	
	You know how to use *JavaScript* "beautifiers" and their possible limitations.	
	You can use *John the Ripper* to brute-force the hash of a *Linux* password	
	You can use *John the Ripper* to crack the password hash of an openssl-encrypted file with a dictionary attack	
	You can use *John the Ripper* to crack a *Linux* password with a dictionary attack	
	You can answer the following question in light of what you have learned in this lab: if one of my password appears in "*Have I been pwnd?*" Does that mean it is already compromised, and any attacker knows my password directly?	

	You can answer the following questions: *Was the brute force cracking process you did quick? What do you think might happen if the password was not at the top of the dictionary? Do you understand what it takes to improve the speed of these processes?*	
	You know the advantages/disadvantages of using a large dictionary based on information leaks versus a custom one.	
	You can obfuscate and deobfuscate *JavaScript* code, and test that it works properly even in its obfuscated mode. You can also answer this question: *what do you think is the main utility of code obfuscation with respect to security?*	
	You clearly understand why EVERY service insists that you use long and complex enough passwords for their accounts	
	You understand that even if the output information is encrypted, not all encryption modes are valid for certain types of information	
	You understand the use *of CyberChef*, its modes of operation, and recipe chains.	
	You understand the advantages and disadvantages of brute force attacks. You can answer this question: *which method is more likely to fail if the password is common, or a typical word?*	
	You can answer this question: *Can you imagine a procedure to send someone a secret message hidden in plain sight and password protected using "normal" things you use daily (email, social media...)?*	
	**King of the crypt**: Your knowledge of applied encryption allows you to use strong, symmetric-key encryption to send private messages to anyone, apply techniques to hide your data and code, and crack passwords from common sources.	