Source: *Stable Diffusion AI*

LABORATORY 3. CRYPTOGRAPHY APPLICATIONS (I)

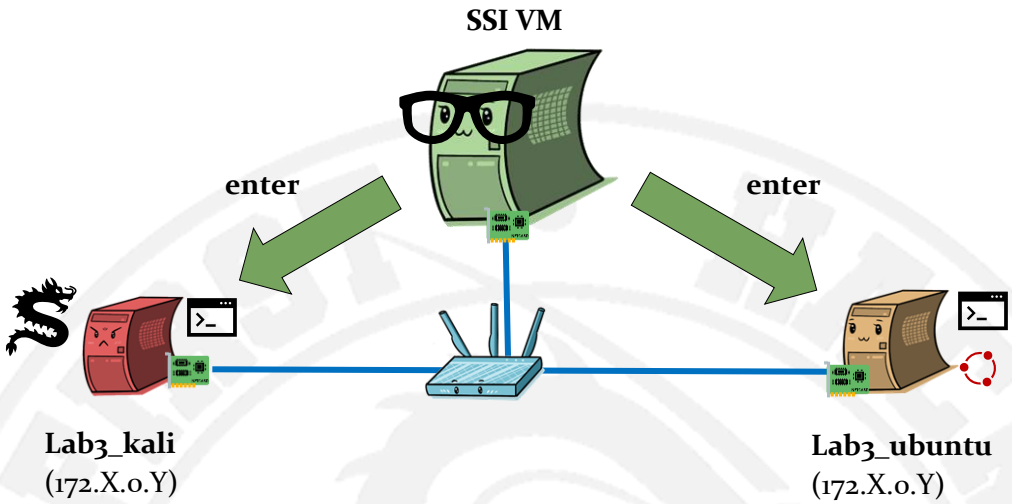
DEPARTMENT OF COMPUTER SCIENCE. UNIVERSITY OF OVIEDO

Computer Security | 2022 – 2023 (V2.7 *"S-81 Isaac Peral"*)

CONTENT

THE INFRASTRUCTURE OF THIS LABORATORY

First, ensure that you have the `ssi_labs` folder structure we provided to you inside your VM. This lab (`lab_03`) consists of an infrastructure composed by two containers communicated via a private LAN: A *Kali* and an *Ubuntu* machine. Both have the software for this lab installed, although **the *John the Ripper* part can only be completed from the *Kali* container** because of the `gpg2john` tool. Not all activities require using all of them. Please see the following diagram to fully understand the infrastructure layout:



BLOCK 1: SYMMETRIC CRYPTOGRAPHY APPLICATIONS

Cipher a file using a strong symmetric key algorithm

Practical application: *You need to cipher a file so nobody without a password can know its contents*

- Begin your work in the *Kali* container of the **Lab 3 infrastructure**. Create a new text file with the plain text message to be sent (for example, your UO and name)
- Then check the supported encryption/hashing algorithms supported by GPG in your Linux distribution with `gpg --version`.
- Choose a strong ciphering algorithm and key size among those that are supported, according to the theory concepts. Use the following cheatsheet to choose the correct options to perform a symmetric encryption of the file with the chosen ciphering algorithm.

gpg (GnuPG) 2.2.20 Cheatsheet (ingenieriainformatica.uniovi.es)
Multipurpose GPL cipher/hash/pubkey software
https://gnupg.org/

```
redondo@server1804:~$ gpg -o original.txt -d cryptandsign.enc
gpg: encrypted with 3072-bit RSA key, ID CA493341D9ED0E95, created 2019-11-18
      "redondo <redondo@mail.es>"
gpg: Signature made lun 18 nov 2019 10:12:34 UTC
gpg:                using RSA key 230C013753283601EDE49B6B4811A20BF1861B2A
gpg: Good signature from "operario <operario@mail.es>" [full]
```

```
operario@server1804:~$ gpg --list-keys
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2021-11-17
/home/operario/.gnupg/pubring.kbx
---------------------------------
pub   rsa3072 2019-11-18 [SC] [expires: 2021-11-17]
      230C013753283601EDE49B6B4811A20BF1861B2A
uid           [ultimate] operario <operario@mail.es>
sub   rsa3072 2019-11-18 [E] [expires: 2021-11-17]
```

GENERAL USAGE

`gpg [options] [files]`

NOTES

Sign, check, encrypt or decrypt

Default operation depends on the input data

Also supports the following compression algorithms: No compression, ZIP, ZLIB, BZIP2

OPTIONS

Symmetric encryption	Public/private key handling (II)
Supported cipher algorithms: IDEA, 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH, CAMELLIA128, CAMELLIA192, CAMELLIA256. Use ***--cipher-algo*** *option to choose one*	**--full-generate-key**: full featured key pair generation
-c, --symmetric: encryption only with symmetric cipher	**--generate-key**: generate a new key pair
Signatures	**--generate-revocation**: generate a revocation certificate
Supported hash algorithms: SHA1, RIPEMD160, SHA256, SHA384, SHA512, SHA224	**--import**: import/merge keys
--clear-sign: make a clear text signature	**--list-signatures**: list keys and signatures
--verify: verify a signature	**--lsign-key**: sign a key locally
-b, --detach-sign: make a detached signature	**--print-md**: print message digests
-s, --sign: make a signature	**--quick-add-uid**: quickly add a new user-id
Asymmetric encryption	**--quick-generate-key**: quickly generate a new key pair
Supported public key algorithms: RSA, ELG, DSA, ECDH, ECDSA, EDDSA	**--quick-lsign-key**: quickly sign a key locally
-d, --decrypt: decrypt data (default)	**--quick-revoke-uid**: quickly revoke a user-id
-e, --encrypt: encrypt data	**--quick-set-expire**: quickly set a new expiration date
Public/private key handling (I)	**--quick-sign-key**: quickly sign a key
--card-status: print the card status	**--receive-keys**: import keys from a keyserver
--change-passphrase: change a passphrase	**--refresh-keys**: update all keys from a keyserver
--change-pin: change a card's PIN	**--search-keys**: search for keys on a keyserver
--check-signatures: list and check key signatures	**--send-keys**: export keys to a keyserver
--delete-keys: remove keys from the public keyring	**--server**: run in server mode
--delete-secret-keys: remove keys from the secret keyring	**--sign-key**: sign a key
--edit-card: change data on a card	**--tofu-policy VALUE**: set the TOFU policy for a key
--edit-key: sign or edit a key	**--update-trustdb**: update the trust database
--export: export keys	**-k, --list-keys**: list keys
--fingerprint: list keys and fingerprints	**-K, --list-secret-keys**: list secret keys
EXAMPLES	**Miscellaneous options**
Sign and encrypt for user Bob: *gpg -se -r Bob [file]*	**--openpgp**: use strict OpenPGP behavior
Make a clear text signature: *gpg --clear-sign [file]*	**--textmode**: use canonical text mode
Make a detached signature: *gpg --detach-sign [file]*	**-a, --armor**: create ascii armored output
Show keys: gpg *--list-keys [names]*	**-i, --interactive**: prompt before overwriting
Show fingerprints: *gpg --fingerprint [names]*	**-n, --dry-run**: do not make any changes
Cipher a file using a strong symmetric key algorithm: *gpg --symmetric --cipher-algo AES256 –c message.txt*	**-o, --output FILE**: write output to FILE
Cipher files with a user-friendly Data Format: *gpg --armor --symmetric --cipher-algo AES256 –c message.txt*	**-r, --recipient USER-ID**: encrypt for USER-ID
Decipher a file that used symmetric encryption: *gpg --decrypt --output=dmessage.txt message.txt.gpg*	**-u, --local-user USER-ID**: use USER-ID to sign or decrypt
Clear signature for a file: *gpg --clearsign file.txt*	**-v, --verbose**: verbose
Assymetric encryption for a particular user (redondo): *gpg -o file_for_redondo.gpg -e -r redondo file.txt*	**-z N**: set compress level to N (0 disables)
Decryption of asymmetrically encrypted text: *gpg -o file.txt -d file.txt.gpg*	

- **When asked for an encryption key, give it the key** "`password`". The file `message.txt.gpg` should appear when the process finishes.

Expected Results: This activity finishes when you can cipher a file using a strong symmetric key algorithm.

Cipher files in an "ASCII-armored" format

Practical application: *You need to cipher a file so nobody without a password can know its contents, but in text format so you can send the encrypted content easily*

Repeat the previous procedure adding the `--armor` option. Check the result and think about what possibilities does this format offer over the previous one.

Expected Results: This activity finishes when you can cipher a file using a strong symmetric key algorithm in armored format.

Decipher a file that use symmetric encryption

Practical application: *You need to know the contents of a previously ciphered file, knowing the proper password*

Using the previous cheatsheet, search the appropriate option to decipher both files generated previously (the armored and the non-armored one) into different output files. Check that the output of both operations is the same.

Send both ciphered files using `scp` to the *Ubuntu* container of the infrastructure and decipher them there (`scp <file> <user>@<IPOfTheUbuntuMachine>:.`). If the transfer fails, make sure that the `ssh` service is active on the destination container (`service --status-all` must show a `[ + ]` attached to the `ssh` service) and do `service ssh start` if not.

Expected Results: This activity finishes when you manage to decipher a previously ciphered file.

Exchange a key using the Diffie-Hellman algorithm

Practical application: *You can know the basics of the Diffie-Helman key exchange algorithm, used worldwide in many applications nowadays*

***(NOTE**: This exercise assumes you use a partner. However, it is possible to do this without one. Just send the file from the Kali to the Ubuntu container in the infrastructure and continue with the instructions).*

The *Diffie-Hellman* key exchange algorithm needs a pair of numbers to proceed to exchange the keys. We will use the following numbers, known by each communication parties.

- **p**: The prime number 997
- **a**: The p root primitive, 7

Now **each student/user must think into a private number X < p to generate the public key Y of each one**. We calculate Y using the following formula `Y = pow(a, X) mod p`. We can use the *Windows Calculator* for this. Once done, we store the result into a file. For example, if we have `user1` and `user2`:

- $User_1$ chooses the private number **X = 35**
- $User_2$ chooses the private number **X = 55**

Then we can calculate each Y value

- $\mathbf{Y_{user1}} = 7^{35} \bmod 997 = 389$ (**pkPubUser1**)
- $\mathbf{Y_{user2}} = 7^{55} \bmod 997 = 195$ (**pkPubUser2**)

Once Y is calculated, we can follow this procedure:

- Place your Y (public key) value into a file (for each user)

```
echo -e "The public key is <place here the calculated Y>" | tee -a pkPubUOxxxxx.txt
```

- Send the public key to your partner. Another option is to send it via **Secure Copy** (`scp`) to the other machine in the infrastructure, using the same procedure we saw.
- Our partner will do the same operation with us, so both of you can now calculate the private key using the public key that has been sent. To calculate the corresponding private key, we use this formula (again with the *Windows Calculator*) `pkPrv = pow(<pkPubOfThePartner>, X) mod p`. Example:
  - For `user1`: `pkPrvUser1 = pow(pkPubUser2, X) mod p -> (195^35) mod 997 -> 146`
  - For `user2`: `pkPrvUser2 = pow(pkPubUser1, X) mod p -> (389^55) mod 997 -> 146`

(***NOTE**: Obviously in a real situation much bigger primes must be chosen to generate secure keys. Not every prime is equally valid, as there are primes that may weaken the algorithm. Currently, a minimum of 2048-bit key sizes are advised for Diffie-Hellman key exchanges, although the exchange procedure is exactly what we explained. Diffie-Hellman is mostly used by software, such as OpenSSH, instead of directly by an end-user like this. If you are curious, more information can be found here:*

- ***A public service to generate strong Diffie-Hellman parameters** (2048+ bit sizes, discarding inadequate primes): https://2ton.com.au/dhtool/*
- ***How to use these parameters to strengthen OpenSSH connections**: https://www.linode.com/docs/security/advanced-ssh-server-security/*

Expected Results: This activity is finished when you can calculate correctly the *pkPrv* keys for both users.

Disk encryption

Practical application: *You need to create a folder whose contents are automatically encrypted, so nobody but you can access to them*

Filesystem, directory, and disk encryption are one of the most common applications of symmetric key algorithms. When *Ubuntu* is being installed, it offers to cipher our home directory as an option (this also ciphers

the `/swap` partition). Even if we choose to not to do it, it is possible to do encrypt disks on a later time: https://www.howtogeek.com/116032/how-to-encrypt-your-home-folder-after-installing-ubuntu/

However, in this lab we are not going to do a full disk encryption, but a quicker version of the same cryptography application: encrypt just a folder (and its contents). To do that, follow this procedure **in the SSI VM** (**this does not work inside containers!** ☹️):

- Install the disk encryption software: `sudo apt install ecryptfs-utils`
- Create a folder to cipher on your home directory. Make sure it is empty (if not, backup their contents…just in case 😊).
- Encrypt the folder mounting it under the `ecryptfs` filesystem `mount -t ecryptfs <your folder path> <your folder path>` (yes, the mount point and the folder are the same. E. g. `mount -t ecryptfs /home/ssiuser/secure /home/ssiuser/secure`). You will need `root` permissions.
- Choose an appropriate symmetric ciphering algorithm according to the theory.
- Choose `32` bytes for the key.
- Choose `No` to plaintext passthrough.
- Choose `No` to encrypt file names (although this is a minor detail)
- Answer `yes` to the two following questions (they appear because this is the first usage of disk ciphering in the OS)
- Check with the `mount` command that the mount point is mounted as encrypted.
- Copy or create a text file to the encrypted folder. Can you read it?
- Unmount the ciphered mount point with `sudo umount /home/secure`. Can you read its contents now? (please get out of the folder before you do that! 😊)
- Mount it again with the same parameters. Can you read the contents again?

Expected Results: This activity is finished when you check that you can cipher a folder and test that the contents are indeed ciphered.

BLOCK 2. ENCODING, OBFUSCATION AND STEGANOGRAPHY

Cyberchef: encoding data

Practical application: *You need to know how to perform a wide range of data transformation operations with a reference tool*

The *Cyberchef* (https://gchq.github.io/CyberChef/) is a reference web page to perform all kinds of transformations to data, from encoding to full encryption/decryption algorithms, all online and with little effort. Every transformation is a "**recipe**" and multiple can be selected from the left pane (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 CTFs when you find encoded or encrypted data and have no other available tools to process the information (in the end, **this webpage has A LOT of recipes**, so you normally find what you want here, even if it is not encryption related: for example, it also allows to view file metadata). To make some examples, you can do the following.

- ROT13 substitution cipher over a text you choose.
- ROT13+ROT47 substitution ciphers over a text you choose (you see that you can make a cipher chain)
- Decipher the output of the previous one making the necessary transformation to the chain.
- *Blowfish* encrypt with a long enough Key and IV.
- *Blowfish* decrypt of the previous operation (save the encryption parameters! 😊)
- *Base64* encoding (the most common nowadays!)
- …

Expected Results: This activity is finished when you find yourself familiar enough with how to use *Cyberchef*.

Using steganography via steghide

Practical application: *You can hide secrets in images without altering what it is displayed on it*

To finish this process, we can choose any image from Internet to embed the previously ciphered message into it via `steghide`, using as `steghide` password the one that has been exchanged in the previous step using ***Diffie-Hellman*** (`pkPrv`). Images must be in a `steghide` recognized format; most *.jpg* should be valid. An example working image is this: https://images.idgesg.net/images/article/2018/06/intel-8086-100760661-large.jpg

`steghide` is installed in the Kali container of the **lab 3 infrastructure**. Use the appropriate options of `steghide` to hide the ciphered message inside the image (use the provided cheatsheet for that) and send the image with the hidden message to the *Ubuntu* container, where you must extract the message. To do this, remember that the `/shared` directory of each container can be used to share files between the VM and the container, making sharing files between containers quite easy. Use the appropriate options to extract the ciphered 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

Expected Results: This activity is finished when you manage to hide a message into an image via steganography and extract it later.

Obfuscating code to make it difficult to understand

Practical application: *You need to obfuscate your JavaScript code so knowing how you implemented its functionality is much harder*

Code obfuscation is related to encryption and steganography because, although **it is NOT encryption**, it transforms the code in a way it is still runnable, but more difficult to understand unless de-obfuscated. In other words, both mechanisms have the same way of operating, although the result of obfuscation is readable by anyone (readable does not mean understandable! 😊).

To quickly evaluate how obfuscation works we can use *JavaScript* as an example. Please note that most languages have their own obfuscation tools adapted to their syntax that you may use, but the principles are the same. To check how obfuscation works, follow this procedure:

- Go to this page and check the *JavaScript* sample it provides. You can save the output in a file for checking it later: https://js.do/
- Copy the *JavaScript* in the sample code (**ONLY the *JavaScript* code block (not the script tags), leave the HTML alone!**) and pack it (with no options) with this online packer: http://dean.edwards.name/packer/
- Copy the packed code to the `<script>` block of the page of the first link, and check that the execution output is the same.

- Pack the code again, **but adding the *Base62 encode,* and the *shrink variable* options**
- Copy again this heavily obfuscated code and check it works again in the first link web page.
- Pick the heavily obfuscated *JavaScript* and process it with the "beautifiers" here http://jsnice.org/ or here https://beautifier.io/ . Think about what these tools do.
- Finally, process the heavily obfuscated code with this web page: https://obfuscator.io/, copy the results and check that the code still works again.
- What happens if you try to beautify this last "ultimately obfuscated" code?

Expected Results: This activity finishes when you have used all the obfuscation options mentioned and checked that the execution output is still the same one as the original web page.

BLOCK 3. INTRODUCTION TO OFFLINE PASSWORD CRACKING

John the ripper to break PGP

Practical application: *You need to crack a PGP ciphered message that uses a common password*

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

Passwords are stored in hash format, as we saw in the cryptography theory topic. Therefore, to break it we have only three strategies:

- **Brute force**: Which attempts to guess the password by sequentially iterating through every letter, number, and special character combination of a vocabulary of characters we stablished beforehand (or just any possible one!). This is a painfully slow process, but effective…if you have the time 😊.
- **Dictionary** (aka *wordlists*): This attack uses a file containing lists of common passwords (usually taken from a password breach of some kind) to guess a given password. Can be helpful in CTFs, but more difficult to apply in the real world (unless your dictionary has passwords that are used by actual users of the password file!).
- **Rainbow table**: Rainbow tables are a series of pre-computed hashes. The idea is that these rainbow tables include all hashes for a given algorithm. So instead of cracking the hash/password/etc. you perform a look up of the hash in the table. This takes considerable processing power to achieve but could be more effective.

We are going to use a dictionary-based attack to break the password of our previously encoded PGP message, following these rules:

- Use a popular **password-cracking tool**, like *John the Ripper* (provided in the **Lab 3 infrastructure** (*Kali* container), but installable with `sudo apt install john`)
- We need a **dictionary**, and we provided you the highly popular `rockyou.txt` dictionary of compromised passwords in the `/wordlist` directory of the Kali container. If you need to do something like this in the future, you can have bigger or specialized dictionaries here: https://ns2.elhacker.net/wordlists/
- **The message to crack**. GPG encrypted files are not directly processable with `john`, so you must process them previously with the `gpg2john` utility installed in the *Kali* container of the **Lab 3 infrastructure** (*Ubuntu* default `john` package does not include it). Store the output into a file.

Use the cheatsheet to use `john` with the provided wordlist against the previous file to crack the GPG message password.

Expected results: This activity is finished when you crack the previous PGP message password.

John the ripper + user passwords + crunch

Practical application: *You need to crack a user password with a custom word dictionary*

Let us use `john` abilities for a different purpose. `rockyou.txt` is a big dictionary of leaked passwords, but sometimes this is counterproductive because we know that the user passwords follow a pattern, a rule, or has certain shape. In that case, generating a custom word dictionary adapted to your guesses is the wisest approach, and this is precisely the goal of this exercise. However, instead of a file, we are going to teach you how leaked Linux users and passwords files are usually broken. Apart from `john`, for this you need:

- **A tool to combine the leaked `passwd` and `shadow` files**, like `unshadow` (already installed in the **Lab 3 infrastructure**). This tool takes the `passwd` and the `shadow` files you provide as the first and second parameters, joins them, and output them into a combined format, that you can save into a third file.
- **A wordlist generator**, like `crunch` (already installed in the **Lab 3 infrastructure**, but installing it is quite easy with `sudo apt install crunch`)
- **Before continuing**, first consider the cheatsheet and follow the instructions below.

`crunch` allows you to generate a wordlist following the word patterns you specify if you know or suspect part of the password of a user, the key shape (numbers + letters, only lowercases…), or any data that may save you to check impossible passwords. Going into the `crunch` pattern generation syntax is beyond the scope of this lab, but to ease the process you can use the cheatsheet and these tips:

- The % symbol means “any number”.
- The @ symbol means “any lowercase letter”.
- Fixed strings are treated as constants.

The password to guess has an easy structure we can generate. It begins with “`test`” and ends with “`.`”. We also know that the middle part **is either three numbers or three lowercase letters** (always 3, never mixing numbers and letters). Lengths of patterns for `crunch` are always number of characters + 1 (C strings always add ‘\0’ at the end, so the length must be also one character more than the actual length we need to use).

Expected results: This activity is considered finished when you crack the password of the user `ssiuser` in the provided sample `passwd` and `shadow` files (`/wordlist` directory of the Lab 3 infrastructure containers)

crunch 3.6 Cheatsheet (ingenieriainformatica.uniovi.es)

Tool that generates wordlists from a character set

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

```
ssiuser@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
```

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.

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 zzy9998. 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.

"Pure brute" John

Practical application: *You need to crack a user password using pure brute force, because you do not have clues about what could be the possible password of that user*

If we do not have any clue about the shape of the password, and no common password was useful, we can run the pure brute `john` against the previous file, with all defaults, and see what happens (we could be lucky…or we must wait for days/weeks/years/…). Please note that `john` applies default permutations to the words it generates and have multiple crack models. To evaluate this, follow these rules:

Remove the `john.pot` file you may find under the hidden `.john` directory that appears on the folder on which you are working. If you do not do that, and a password has been cracked in a previous execution, `john` will report that the password has been cracked and will not search anything, even if there are some uncracked passwords left! ☹ .

Run `john` in **incremental** mode, specifying an extra parameter `--max-length=<number>` to limit the amount of the character permutations `john` may use, and therefore make the process finish in a reasonable time. Use the previous cheatsheet to specify the adequate options to crack only the password of the user "`Android`", as we know that this user uses a truly short password (the password length is equal or less than 4 characters).

We provide this documentation as reference if you are curious, but it is not really needed to finish this activity.

- John cracking modes: https://www.openwall.com/john/doc/MODES.shtml
- John FAQ: https://www.openwall.com/john/doc/FAQ.shtml
- John examples: https://www.openwall.com/john/doc/EXAMPLES.shtml
- John wordlist rules: https://www.openwall.com/john/doc/RULES.shtml
- More information: https://manualdehacker.com/john-the-ripper-cracking-passwords/

Expected results: This activity is finished when you manage to crack the password of the user Android.

BADGES AND SELF-ASSESSMENT

NOTE: You have a version of this badge table in an editable format available in the *Virtual Campus*. You can use this file to easily create a document in the format you want and take extended notes of your activities to create a log of what you did. Remember that this material is key to keep track of your self-evaluation.

Badge Level	Unlocked when	Unlocked?
	Be able to cipher any file using a strong symmetric ciphering algorithm. Answer the following questions: *Are the contents of the ciphered file binary or plain text?*	
	Be able to cipher any file using a strong symmetric ciphering algorithm in armored format. Answer the following questions: *Are the contents of the ciphered file binary or plain text? Does it generate a different output file?*	
	Answer the following questions: *Can you send an armored file via email? Or via WhatsApp, etc. (excluding length limits)? How should you transmit the key to the receiver?*	
	Be able to decipher a file ciphered with a symmetric algorithm, no matter the format used (armored or not) into an output file you choose. Answer the following question: *Does the decipher process changes depending on the ciphered file format?*	
	You know how to use `scp`. You can answer this question: *which service uses* `scp` *to copy files securely?*	
	Use `crunch` to generate limited-length simple word dictionaries	
	Answer the following question: *How do you think you can guess the "shape" of a user password (that is, make a guess of part of its contents)?* (**TIP**: How can you know stuff about people you do not personally know?)	
	Understand how to cipher folders and their contents in the filesystem. Answer this question: *What kind of attack does this make more difficult? Is encryption transparent (it can be used without introducing passwords when working with encrypted files?*	
	You understand the process of hiding information into an image file. You can answer this question: *is the file with text introduced via steganography visually different from the original one?*	
	You know how to use *JavaScript* beautifiers and their limitations.	
	Use *John the ripper* to break a Linux password by brute force	
	Use *John the ripper* to break a PGP password with a dictionary attack	
	Use *John the ripper* to break a Linux password with a dictionary attack	
	Answer the following question: *if one of my keys have been compromised in have I been pwnd?, does that mean that it is immediately compromised, and the attacker knows my actual key outright?*	
	Answers the following questions: *Was the brute-force cracking process you made fast? What do you think it could happen if the password were not among the first in the dictionary? Do you understand what it is needed to enhance the speed of these processes?*	
	You know the advantages/disadvantages of using a large leak-based dictionary vs a custom one.	

	You can obfuscate and deobfuscate *JavaScript* code, and test that it works correctly even in its obfuscated mode. You can also answer this question: *what do you think is the main utility of code obfuscation regarding security?*	
	You clearly understand why EVERY service insist that you use long-enough and complex-enough passwords for their accounts	
	You understand how the *Diffie-Hellman* key exchange algorithm works.	
	Understand *Cyberchef* usage, modes of operation and chains of recipes.	
	You understand the advantages and disadvantages of brute-force attacks. You can answer this question: *which method is more prone to fail if the password is common, or a typical word? And if not?*	
	You can answer this question: *Can you imagine a procedure to send to someone a secret message hidden at plain sight and protected via password using "normal" things you typically use daily (email, social networks...)?*	
	**Far crypt**: Your knowledge of applied encryption enables you to use strong symmetric key encryption to send private messages to anyone, apply techniques to hide your data and your code, and break passwords from common sources.	