Ejercicios Prácticos : Unidad 4

Tomás F. Melli

October 2025

Índice

1 Ejercicio 1

En el contexto de desarrollo de una aplicación web el programador debe tomar ciertas decisiones de autenticación. En este ejercicio, se propone que el desarrollador implemente el mecanismo de autenticación guardando las contraseñas en una base de datos de las siguientes maneras :

1. **Texto claro** : el problema de dejar las contraseñas en claro en la base de datos es que cualquiera que tenga acceso a ella podrá leer las de todos los usuarios de la aplicación. El ataque potencial es que una persona no autorizada o autorizada pero malintencionada, acceda a la base de datos y robe las credenciales. Una consecuencia posible es que una vez robada la credencial, ese ataque se replique a otras plataformas en caso de que ese usuario reutilize la misma contraseña.

2. **Cifradas con AES y una clave fija** : en este contexto, sucede que si ciframos todas las contraseñas con la misma clave, el que esté en posesión de la clave puede decifrar todas las credenciales. Frente a un ataque de diccionario/fuerza bruta, el atacante con encontrar una clave le es suficiente para acceder a toda la db.

3. **Hasheadas con SHA-1** : como sabemos que SHA-1 es una función muy rápida, es posible que el atacante pruebe muchas contraseñas muy rápido, por tanto, usar SHA-1, además del tema de las colisiones, es inseguro frente a ataques de fuerza bruta y rainbow tables.

Tal vez la mejor solución es utilizar algoritmos de cifrado robustos como **scrypt**

- **Cuando el usuario crea la contraseña:**
  - Generar un *salt* aleatorio único por usuario.
  - Elegir parámetros de scrypt (N, r, p) que determinen la memoria y el tiempo de cálculo.
  - Aplicar scrypt a la contraseña usando ese *salt* y los parámetros; obtener un *hash* derivado (clave).
  - Guardar en la base de datos una entrada estructurada que incluya:
    - Identificador del algoritmo (scrypt)
    - Los parámetros (N, r, p)
    - El *salt*
    - El *hash* resultante (por ejemplo: `scrypt$N$r$p$salt$hash`)
- **Cuando el usuario se autentica:**
  - Recuperar el *salt* y los parámetros almacenados para ese usuario.
  - Recalcular scrypt con la contraseña ingresada y los mismos parámetros y *salt*.
  - Comparar el *hash* recalculado con el *hash* almacenado (comparación en tiempo constante).
  - Si coinciden $\rightarrow$ autenticación correcta; si no $\rightarrow$ falla.

De esta manera, el almacenamiento de las credenciales se vuelve ultra seguro, pero es vulnerable contra otros ataques como phishing por ejemplo.

2 Ejercicio 2

En el contexto de una aplicación que permite *remote OS Auth*, sucede que:

- Un usuario inicia sesión en Windows con su cuenta de dominio CORP/juan.
- Luego abre una aplicación que se conecta a una base de datos.
- El sistema de base de datos reconoce la identidad CORP/juan que proviene del entorno autenticado por el sistema operativo, y le permite el acceso sin pedir usuario y contraseña otra vez.

Con esto podemos deducir que si se falsifica la identidad, ya se accede directamente a la db. En este escenario hablamos de una base de datos Oracle que tiene la particularidad de permitir el parámetro de inicialización **REMOTE_OS_AUTHENT = TRUE** para simplificar el acceso desde todos los equipos de la red interna. Un atacante en la misma red puede falsificar un cliente Oracle (spoofear el nombre del host o del usuario del sistema operativo) y autenticarse sin necesidad de contraseña. Para resolver este problema, se puede hacer uso de Kerberos (Enterprise User Security), los pasos son :

1. Inicio de sesión

- El usuario inicia sesión en su sistema (por ejemplo, Windows) y se autentica ante el KDC (Key Distribution Center), el servidor central de Kerberos.
- El KDC emite un Ticket Granting Ticket (TGT) cifrado, que el cliente guardará temporalmente.

2. Solicitud de acceso a la base de datos
   - Cuando el cliente quiere conectarse a Oracle, presenta su TGT al KDC y solicita un Service Ticket específico para ese servicio (por ejemplo, "oracle/dbserver.corp.local").
3. Autenticación ante el servidor
   - El cliente envía al servidor el Service Ticket, que está cifrado con la clave secreta del servicio.
   - El servidor lo valida con su propia clave, confirmando la identidad del cliente.
   - No se transmite contraseña en ningún momento.
4. Conexión establecida
   - Si todo es válido, el usuario queda autenticado como una identidad Kerberos verificada.
   - Oracle mapea esa identidad a un usuario interno con privilegios específicos.

3 Ejercicio 3

El servicio Identd (TCP/113) permite identificar al usuario propietario de una conexión TCP saliente, con el objetivo de agregar una capa de trazabilidad o autenticación ligera entre sistemas conectados en red. La idea es que cuando un servidor remoto recibe una conexión TCP desde un cliente, puede abrir una conexión inversa al cliente en el puerto 113, y preguntarle qué usuario local tiene abierta esta conexión. El identd daemon en el cliente responde con el nombre de usuario local asociado al proceso que originó la conexión.
El flujo es el siguiente :

- Un usuario en la máquina cliente (IP 192.168.0.5) usa telnet para conectarse a servidor (IP 10.0.0.2, puerto 23).
- El servidor 10.0.0.2 recibe la conexión TCP desde el puerto local 23 hacia el puerto origen 54022.
- El servidor abre una conexión a 192.168.0.5:113 y pregunta: `54022 , 23`. La consulta que el servidor le hace al servicio Identd (en el cliente) tiene esta pinta `<puerto_origen_cliente> , <puerto_destino_servidor>`
- El identd en el cliente responde, por ejemplo: `54022 , 23 :  USERID : UNIX : tonius`
- El servidor puede registrar o verificar que el usuario remoto es "tonius".

Se mencionan los *comandos r*, estos son comandos clásicos de UNIX que se usaban antes de SSH para conectarse y ejecutar cosas en otros servidores de forma automática, cuando entre los hosts existía una relación de confianza (o sea que si alguien se conecta desde B, A asume que es "seguro" o "autenticado" sin pedir contraseña).

- **Remote login** (rlogin) : Si el servidor B confía en tu máquina A (porque A está en /etc/hosts.equiv o en tu /.rhosts), entonces no te pide contraseña. Resultado: obtenemos un shell remoto en B. `rlogin servidorB`
- **Remote shell** (rsh) : Sirve para ejecutar un comando en otra máquina sin abrir sesión interactiva. Si hay relación de confianza, el comando se ejecuta directamente sin contraseña. `rsh servidorB comando`
- **Remote copy** (rcpy): sirve para copiar archivos entre máquinas. `rcp archivo.txt servidorB:/tmp/`

En este ejercicio el escenario es: un servidor A y otro B, con el cual A tiene una relación de confianza . Entonces para que A sepa qué usuario desde B está intentando acceder, le pregunta a B (por el puerto 113) a través del servicio Identd. Si B responde "es el usuario juan", y A tiene en su configuración que juan@B es de confianza, entonces A le da acceso directo sin contraseña. Con esto en mente, el ejercicio plantea que un atacante con usuario no privilegiado se conecta a A por el puerto 113, y B responde algo, lo que el atacante logra falsificar.
Entonces, vayamos al punto, el SO ofrece distintos mecanismos de seguridad para evitar esta situación:

- En UNIX/Linux los puertos < 1024 (como 113) son privilegiados: sólo root o un proceso con la capacidad **CAP_NET_BIND_SE** puede hacer **bind() (asociar un puerto de red a su proceso)** en esos puertos. Un usuario normal intentando bind(113) recibirá un error (permiso denegado). No puede poner ahí su propio identd que atienda las consultas.

- No puede contestar la consulta TCP legítima. Cuando A abre una conexión a B:113, esa conexión TCP se dirige al socket que ya está en escucha (el identd legítimo controlado por root). Un usuario normal no puede interceptar ni "robar" esa conexión a menos que tenga control del kernel, o tenga privilegios para manipular sockets del sistema (cosa que no pasa sin escalada de privilegios).

La forma que tiene un atacante en este contexto para poder falsificar la respuesta es :

1. El atacante explota un bug para escalar privilegios.
2. El atacante está en la red y puede hacer MITM (entonces no necesita privilegios en B).

4 Ejercicio 4

Lo primero que vamos a hacer es bajar **John The Ripper**, nos dicen que tiene que ser la versión *Jumbo* ya que esta contiene más chiches. En este caso, como vamos a trabajar con un **.doc** (versión Microsoft Word 97-2003), hay un programa llamado **office2john.py** que se encuentra, en mi caso en `/snap/john-the-ripper/694/bin/` (para hallarlo se puede correr `sudo find / -type f -name 'office2john.py' 2>/dev/null`, yo lo hice parado en snap, porque lo bajé con snap). Una vez chequeado que lo tenemos al script, parados en la carpeta que tiene **parcia_1c2008** corremos :

$$python3 \;\; /snap/john-the-ripper/694/bin/office2john.py \;\; parcial\_1c2008.doc > parcial\_hash.txt$$

La idea es obtener el hash con esto. Chequeamos que esté bien creado :

```
cat parcial_hash.txt
 parcial_1c2008.doc:$oldoffice$1*366cfb1216a9c39e1e86215dea6cd26b*34a78c98857fc0c3935c
 65ddc7d5ae0d*8e8269fc85d736a8a5ec8b0dc2587e76:::::parcial_1c2008.doc
```

La idea ahora es, a partir de este hash, encontrar a qué contraseña corresponde. Vamos a hacer uso de la recomendación de la cátedra y vamos a clonar todo el repo (no es necesario)

```
git clone https://github.com/danielmiessler/SecLists.git ~/wordlists/SecLists
```

Con esto, pasamos a la parte jugosa, usar `John`

```
 --wordlist[=FILE] --stdin  Wordlist mode, read words from FILE or stdin --pipe like --stdin, but bulk
     reads, and allows rules
```

```
john --wordlist= (ruta)/wordlists/SecLists/Passwords/Common-Credentials/10k-most-common.txt
    parcial_hash.txt
```

Le pide a John the Ripper que pruebe (en modo diccionario) todas las palabras del archivo `10k-most-common.txt` contra el guardado en parcial_hash.txt hasta que encuentre la contraseña que coincide.
Obtenemos la siguiente respuesta :

```
Warning: detected hash type "oldoffice", but the string is also recognized as "oldoffice-opencl"
Use the "--format=oldoffice-opencl" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 1 password hash (oldoffice, MS Office <= 2003 [MD5/SHA1 RC4 32/64])
Cost 1 (hash type [0-1:MD5+RC4-40 3:SHA1+RC4-40 4:SHA1+RC4-128 5:SHA1+RC4-56]) is 1 for all loaded hashes
Will run 4 OpenMP threads
Note: Passwords longer than 41 [worst case UTF-8] to 64 [ASCII] rejected
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
0g 0:00:00:00 DONE (2025-10-13 20:50) 0g/s 500000p/s 500000c/s 500000C/s becky1..eyphed
Session completed.
```

- `Warning: detected hash type "oldoffice", but the string is also recognized as "oldoffice-opencl"` : John detectó que el hash pertenece al esquema de Office antiguo (2003). Se sugiere la variante `-opencl` sólo si se desea usar GPU/OpenCL. No es un error — es una recomendación para aceleración por GPU.
- `Using default input encoding: UTF-8` : John procesará las palabras de las listas como UTF-8.
- `Loaded 1 password hash (oldoffice, MS Office` $\leq$ `2003 [MD5/SHA1 RC4 32/64])`: Se cargó 1 hash y se confirma el tipo: cifrado típico de Office 2003.
- `Cost 1 (... ) is 1 for all loaded hashes`: Información interna sobre el "cost" del hash; para estos formatos antiguos cada intento no es tan costoso como en Office moderno.
- `Will run 4 OpenMP threads` John usará 4 hilos de CPU para probar candidatos en paralelo (dependiendo de la CPU disponible).

- `Note: Passwords longer than 41 [worst case UTF-8] to 64 [ASCII] rejected` John descartará candidatos que superen ese largo máximo (límite impuesto por el formato).
- `Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status` Instrucciones interactivas mientras el proceso está corriendo.
- `Enabling duplicate candidate password suppressor` John evita probar la misma contraseña candidata más de una vez, lo que reduce trabajo redundante.
- `0g 0:00:00:00 DONE (2025-10-13 20:33) 0g/s 500000p/s 500000c/s 500000C/s becky1..eyphed` Esta es la línea clave y su interpretación:
  - `0g` : **0 guesses** encontradas (no se crackeó la contraseña).
  - `0:00:00:00` : duración de la corrida: 0 segundos.
  - `DONE` : terminó porque se agotó la wordlist + reglas (no quedaban candidatos por probar en esa ejecución).
  - `0g/s`:tasa de contraseñas resueltas por segundo (0 porque no se resolvió ninguna).
  - `500000p/s 500000c/s 500000C/s` : donde `p/s` son passwords tried per second (contraseñas probadas por segundo), `c/s` candidate passwords per second (candidatos generados/procesados por segundo) y `C/s` puede indicar candidatos totales en otro subproceso/algoritmo.
  - `becky1 .. eyphed` ejemplos del primer y último candidato probados en ese tramo (sólo ilustrativos).
- `Session completed.` : La sesión terminó normalmente.

Como no encontró nada, lo que vamos a hacer es probar con otro diccionario, pero esta vez le vamos a poner la opción –**rules** :

```
    --rules[=SECTION[,..]]  Enable word mangling rules (for wordlist or PRINCE modes), using default or
        named rules
```

Esta opción en John the Ripper sirve para activar las "reglas de mutación" que modifican las palabras de un diccionario para generar variantes más complejas de contraseñas, aumentando las chances de éxito en un ataque de fuerza bruta basado en diccionario. Es decir, aplicará transformaciones configuradas en el archivo john.conf (o john.ini), como:

- Mayúsculas/minúsculas (Password, PASSWORD)
- Agregar números (password1, password123)
- Revertir (drowssap)
- Sustituir letras por números (p@ssword, pa$$word)
- Combinar reglas (P@ssword123)

Probemos entonces con un diccionario en español que se obtiene de crack me if you can, lo descargamos y

```
    .../Downloads$ zcat spanish.dic.gz > spanishDic.txt
```

Lo vamos a mover a la carpeta wordlists, y entonces :

```
   ../SegInf$ john --wordlist=(ruta)/wordlists/spanishDic.txt --rules parcial_hash.txt
```

Y obtuvimos la siguiente salida :

```
Warning: detected hash type "oldoffice", but the string is also recognized as "oldoffice-opencl"
Use the "--format=oldoffice-opencl" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 1 password hash (oldoffice, MS Office <= 2003 [MD5/SHA1 RC4 32/64])
Cost 1 (hash type [0-1:MD5+RC4-40 3:SHA1+RC4-40 4:SHA1+RC4-128 5:SHA1+RC4-56]) is 1 for all loaded hashes
Will run 4 OpenMP threads
Note: Passwords longer than 41 [worst case UTF-8] to 64 [ASCII] rejected
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
Enabling duplicate candidate password suppressor
parcial2008      (parcial_1c2008.doc)
1g 0:00:00:16 DONE (2025-10-14 10:17) 0.06135g/s 1124Kp/s 1124Kc/s 1124KC/s onde2008..paso2008
Use the "--show --format=oldoffice" options to display all of the cracked passwords reliably
Session completed.
```

Si prestamos atención, esta línea nos dice que encontró la constraseña :

```
    parcial2008      (parcial_1c2008.doc)
```

El número de guesses es 1. Para verla corremos:

```
john --show parcial_hash.txt
```

Esta opción muestra qué hashes ya fueron resueltos (y opcionalmente cuáles quedaron sin resolver) consultando el fichero de resultados (el pot file).

```
parcial_1c2008.doc:parcial2008:::::parcial_1c2008.doc
1 password hash cracked, 0 left
```

Concluimos que la contraseña es : **parcial2008**

5 Ejercicio 5

En este ejercicio tenemos que acceder a un **.zip** con contraseña. Lo primero que vamos a hacer es tratar de mirar un poco los metadatos de `capitulo1.zip`. Para ello, usamos la herramienta **zipinfo** con la opción **-v** (multi-page format) :

```
Archive:  capitulo1.zip
There is no zipfile comment.

End-of-central-directory record:
-------------------------------

  Zip archive file size:                      85717 (0000000000014ED5h)
  Actual end-cent-dir record offset:          85695 (0000000000014EBFh)
  Expected end-cent-dir record offset:        85695 (0000000000014EBFh)
  (based on the length of the central directory and its expected offset)

  This zipfile constitutes the sole disk of a single-part archive; its
  central directory contains 1 entry.
  The central directory is 83 (0000000000000053h) bytes long,
  and its (expected) offset in bytes from the beginning of the zipfile
  is 85612 (0000000000014E6Ch).


Central directory entry #1:
---------------------------

  capitulo1.doc

  offset of local header from start of archive:   0
                                                  (0000000000000000h) bytes
  file system or operating system of origin:      Unix
  version of encoding software:                   3.0
  minimum file system compatibility required:     MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   2.0
  compression method:                             deflated
  compression sub-type (deflation):               normal
  file security status:                           encrypted
  extended local header:                          yes
  file last modified on (DOS date/time):          2020 May 17 19:13:44
  file last modified on (UT extra field modtime): 2020 May 17 19:13:43 local
  file last modified on (UT extra field modtime): 2020 May 17 22:13:43 UTC
  32-bit CRC value (hex):                         9293cb2e
  compressed size:                                85525 bytes
  uncompressed size:                              89600 bytes
  length of filename:                             13 characters
  length of extra field:                          24 bytes
  length of file comment:                         0 characters
  disk number on which file begins:               disk 1
  apparent file type:                             binary
  Unix file attributes (100644 octal):            -rw-r--r--
  MS-DOS file attributes (00 hex):                none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
    The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7875 (Unix UID/GID (any size)) and 11 data bytes:
    01 04 e8 03 00 00 04 e8 03 00 00.

  There is no file comment.
```

- **Encabezado general**
  - `Archive:  capitulo1.zip`
    Indica el nombre del archivo ZIP analizado.

- `There is no zipfile comment`
  El ZIP no contiene un comentario global opcional.

- **End-of-central-directory record**
  - `Zip archive file size`
    Tamaño total del ZIP en bytes.
  - `Actual / Expected end-cent-dir record offset`
    Posición donde termina el *central directory*. Coinciden en este caso, indicando integridad.
  - `Sole disk / single-part archive`
    El ZIP no está dividido en varios discos; es un archivo completo.
  - `Central directory length / offset`
    Tamaño y posición del *central directory*, que contiene los metadatos de los archivos dentro del ZIP.

- **Central directory entry #1 (archivo contenido)**
  - `capitulo1.doc`
    Nombre del archivo dentro del ZIP.
  - `offset of local header`
    Posición del encabezado local dentro del ZIP.
  - `file system / OS of origin`
    Indica que el ZIP fue creado en Unix.
  - `version of encoding software`
    Versión del software que creó el ZIP.
  - `minimum file system compatibility / minimum software version`
    Versiones mínimas necesarias para extraer el archivo.
  - `compression method: deflated`
    El archivo está comprimido usando Deflate.
  - `file security status: encrypted`
    Archivo cifrado; requiere contraseña para extraer.
  - `extended local header: yes`
    Indica que hay un encabezado extendido después de los datos comprimidos.
  - `file last modified`
    Fecha y hora de modificación (DOS date/time y campos extra UTC/local).
  - `32-bit CRC value`
    Valor CRC32 de verificación de integridad.
  - `compressed size / uncompressed size`
    Tamaño dentro del ZIP y tamaño original del archivo.
  - `length of filename / extra field / file comment`
    Longitudes de los campos del ZIP.
  - `disk number on which file begins`
    Número de disco; 1 en este caso.
  - `apparent file type`
    Tipo aparente del archivo: binario.
  - `Unix / MS-DOS file attributes`
    Permisos de archivo y atributos del sistema de origen.

- **Central-directory extra field**
  - `0x5455 (universal time)`
    Contiene timestamps en UTC/GMT.
  - `0x7875 (Unix UID/GID)`
    Contiene UID y GID del creador del archivo.

- **File comment**
  - `There is no file comment`
    El archivo contenido no tiene comentario adicional.

Esto es realmente innecesario porque sabemos que está encryptado nomás, vamos al jugo, usamos un script de john que nos permite extraer el hash llamado **zip2john** :

```
zip2john capitulo1.zip > capitulo1hash.txt
```

Si esto no les funciona porque se bajaron john con snap y no está ese binario, entonces, borren john con `sudo snap remove john-the-ripper` y bajensé directo del repo con `git clone https://github.com/openwall/john.git` en algún lugar que les guste. Luego parensé en `john/src` y a continuación corran `./configure` y finalmente `make -s clean && make -sj$(nproc)`. Con esto, lo vamos a usar desde ahí con las rutas y fue :

```
(ruta)/john/run/zip2john capitulo1.zip > capitulo1hash.txt
```

Miramos si fue todo bien :

```
cat capitulo1hash.txt
capitulo1.zip/capitulo1.doc: pkzip$1*1*2*0*14e15*15e00*9293cb2e*0*47*8*14e15*99b6*47
    cce0c0529a776f5c7ee79418fbade ....
```

Tenemos el hash, ahora la metodología es correr jonh :

```
~/john/run/john  capitulo1hash.txt
```

Lo invocamos para que detecte automáticamente el formato y que intente los métodos por defecto. Obtenemos :

```
Using default input encoding: UTF-8
Loaded 1 password hash (PKZIP [32/64])
Will run 4 OpenMP threads
Note: Passwords longer than 21 [worst case UTF-8] to 63 [ASCII] rejected
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
0g 0:00:00:00 DONE 1/3 (2025-10-14 11:35) 0g/s 419354p/s 419354c/s 419354C/s Doccapitulo1.zip1900..
    Dzip1900
Proceeding with wordlist:/home/tonius/john/run/password.lst
Enabling duplicate candidate password suppressor using 256 MiB
jorgito          (capitulo1.zip/capitulo1.doc)
1g 0:00:00:00 DONE 2/3 (2025-10-14 11:35) 4.545g/s 321386p/s 321386c/s 321386C/s aparna..123qweas
Use the "--show" option to display all of the cracked passwords reliably
Session completed
```

Es decir, que la contraseña es **jorgito**.

6 Ejercicio 6

En este caso, como son hashes, y puedo usar la GPU, me voy a descargar Hashcat que para usar la GPU es más fácil que John. La instalación es simple, nos tenemos que fijar que la detecte a la placa con (estoy en la máquina con Windows ahora):

```
hashcat.exe -I
```

Si la detecta, entonces, lo primero que vamos a hacer es separar por tipos de hash, los 1 son md5crypt y los 6 son sha512crypt. Esto es importante ya que Hashcat distingue el modo con **-m**. Dicho esto, primero corremos los de md5crypt con **-m 500**

```
hashcat.exe -m 500 -a 0 ...\hashesmd5crypt.txt spanish.dic -r .\rules\rockyou-30000.rule --session
    dict_rock30000 --status --status-timer=10 -w 3 -O
```

Desglose de cada parte

- `hashcat.exe`: Ejecutable de Hashcat, en la carpeta local.
- `-m 500`: Modo de hash. `500` corresponde a **md5crypt** (formato 1... / Unix MD5-based crypt). Indica a Hashcat cómo interpretar los hashes.
- `-a 0`: Modo de ataque. `0` = *dictionary attack* (ataque por diccionario).
- `hashesmd5crypt.txt`: Archivo de entrada con uno o varios hashes (una línea por hash).
- `spanish.dic`: el diccionario en español.
- `-r .\ rules \ rockyou-30000.rule`: Aplica un archivo de reglas que transforma cada entrada del diccionario en variantes (mayúsculas, sufijos, leet, etc.).
- `--session dict_...`: Nombre de la sesión; útil para pausar y reanudar con `--restore`.

- `--status`: Muestra información periódica del estado (velocidad, progreso, tiempo estimado, etc.).
- `--status-timer=10`: Intervalo en segundos para mostrar el estado (cada 10 s).
- `-w 3`: Perfil de carga (workload). `1`=bajo, `2`=por defecto, `3`=alto, `4`=extremo. `-w 3` aumenta utilización de GPU.
- `-O`: Usa kernels optimizados. Mejora velocidad y consumo de memoria por prueba, pero limita la longitud y algunos tipos de ataques.

Al correrlo vamos a ver que el estatus es algo como

```
Session..........: dict_spanish
Status...........: Running
Hash.Mode........: 500 (md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5))
Hash.Target......: ...\hashesmd5crypt.txt
Time.Started.....: Tue Oct 14 13:19:24 2025 (3 mins, 42 secs)
Time.Estimated...: Tue Oct 14 13:47:59 2025 (24 mins, 53 secs)
Kernel.Feature...: Optimized Kernel (password length 0-15 bytes)
Guess.Base.......: File (spanish.dic)
Guess.Mod........: Rules (.\rules\rockyou-30000.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#01........: 10436.9 kH/s (6.86ms) @ Accel:128 Loops:1000 Thr:256 Vec:1
Recovered........: 0/7 (0.00%) Digests (total), 0/7 (0.00%) Digests (new), 0/7 (0.00%) Salts
Progress.........: 2489401530/18071760000 (13.78%)
Rejected.........: 181860000/2489401530 (7.31%)
Restore.Point....: 0/86056 (0.00%)
Restore.Sub.#01..: Salt:0 Amplifier:27087-27088 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#01...: b -> buzoon
Hardware.Mon.#01.: Temp: 60c Fan: 77% Util: 94% Core:1875MHz Mem:6800MHz Bus:16
```

Interpretación

- **Session: dict_spanish** — nombre de la sesión que usaste. Bueno para pausar/reanudar.
- **Status: Running** — el ataque está en ejecución.
- **Hash.Mode: 500** — estás atacando md5crypt .
- **Hash.Target: C:...\hashesmd5crypt.txt** — el fichero de hashes que se está atacando.
- **Time.Started / Time.Estimated** — empezó hace 1:41 min y estima terminar en 26 min (predicción para el keyspace actual).
- **Kernel.Feature: Optimized Kernel (password length 0-15 bytes)** — estás usando -O (kernels optimizados), límite de longitud hasta 15 bytes. Ventaja: velocidad; limitación: no prueba candidatos ¿15 bytes.
- **Guess.Base: File (spanish.dic)** — base del ataque: el spanish.dic.
- **Guess.Mod: Rules (.\rules\rockyou-30000.rule)** — estás aplicando esa regla para generar variantes.
- **Speed.#01: 10605.2 kH/s** — 10.6 MH/s (millones de hashes/segundo). Para md5crypt esto es razonable (md5crypt es costoso por salt/iteraciones).
- **Recovered: 0/7 (0.00%)** — aún no encontró ninguna de las 7 contraseñas.
- **Progress: 1,230,804,470 / 18,071,760,000 (6.81%)** — va 6.8% del keyspace total generado por spanish.dic + reglas.
- **Rejected: 181,860,000 / 1,230,804,470 (14.78%)** — 15% de candidatos fueron descartados/"rechazados".
- **Restore.Point / Restore.Sub** — punto de restauración (útil para –restore).
- **Candidate.Engine: Device Generator** — candidato generado por la GPU.
- **Candidates.#01: a -¿ zuzoi2n** — ejemplos de candidatos recientes.
- **Hardware.Mon.#01: Temp/Fan/Util/Core/Mem** — GPU a 60°C, 76% fan, 94% util

Ahora vamos a ver qué onda con los otros hashes, los que están en **-m 1800**

```
hashcat.exe -m 1800 -a 0  ...\hashessha512crypt.txt spanish.dic -r .\rules\rockyou-30000.rule --
    session dict_spanish --status --status-timer=10 -w 3 -O
```

Que nos da este estatus :

```
Session..........: dict_spanish
Status...........: Running Hash.
Mode.........: 1800 (sha512crypt $6$, SHA512 (Unix)) Hash.
Target......: ...\hashessha512crypt.txt
Time.Started.....: Tue Oct 14 13:33:05 2025 (49 secs)
Time.Estimated...: Wed Oct 15 00:00:40 2025 (10 hours, 26 mins)
Kernel.Feature...: Optimized Kernel (password length 0-15 bytes)
Guess.Base.......: File (spanish.dic)
Guess.Mod........: Rules (.\rules\rockyou-30000.rule)
Guess.Queue......: 1/1 (100.00%)
Speed.#01........: 203.6 kH/s (40.55ms) @ Accel:11 Loops:500 Thr:512 Vec:1
Recovered........: 0/3 (0.00%) Digests (total), 0/3 (0.00%) Digests (new), 0/3 (0.00%) Salts
Progress.........: 87907230/7745040000 (1.14%)
Rejected.........: 77940000/87907230 (88.66%)
Restore.Point....: 0/86056 (0.00%)
Restore.Sub.#01..: Salt:0 Amplifier:117-118 Iteration:1000-1500
Candidate.Engine.: Device Generator
Candidates.#01...: a56 -> zuzon56
Hardware.Mon.#01.: Temp: 60c Fan: 77% Util: 99% Core:1725MHz Mem:6800MHz Bus:16
```

Conclusiones

Lo más interesante es comparar los distintos tiempos que estiman completar, Hashcat utiliza la siguiente fórmula :

$$Tiempo\_restante(s) = \frac{(Keyspace\_total - Progreso\_actual)}{Velocidad\_actual(intentosporsegundo)}$$

donde $Keyspace\_total$ es el número total de candidatos que el ataque generará (spanish.dic × reglas candidatos)
El tema de ambos algoritmos es el **Costo por intento** sha512crypt es mucho más caro por intento que md5crypt (más operaciones/iteraciones/salt). Por eso la velocidad en H/s es mucho menor para sha512crypt. Si miramos md5crypt $\approx$ 10.6 MH/s contra sha512crypt $\approx$ 0.2036 MH/s entonces md5crypt es $\approx$ 52 veces más rápido en intentos/segundo en mi GPU. Esto por sí solo explica la mayor ETA (Time.Estimated) de sha512crypt.