UNIVERSIDAD COMPLUTENSE MADRID

Autenticación en aplicaciones web: contraseñas

Enrique Martín - emartinm@ucm.es

Gestión de la Información en la Web (GIW)

Optativa de grados

Fac. Informática

Contenidos

Hypertext Transfer Protocol Secure (HTTPS)

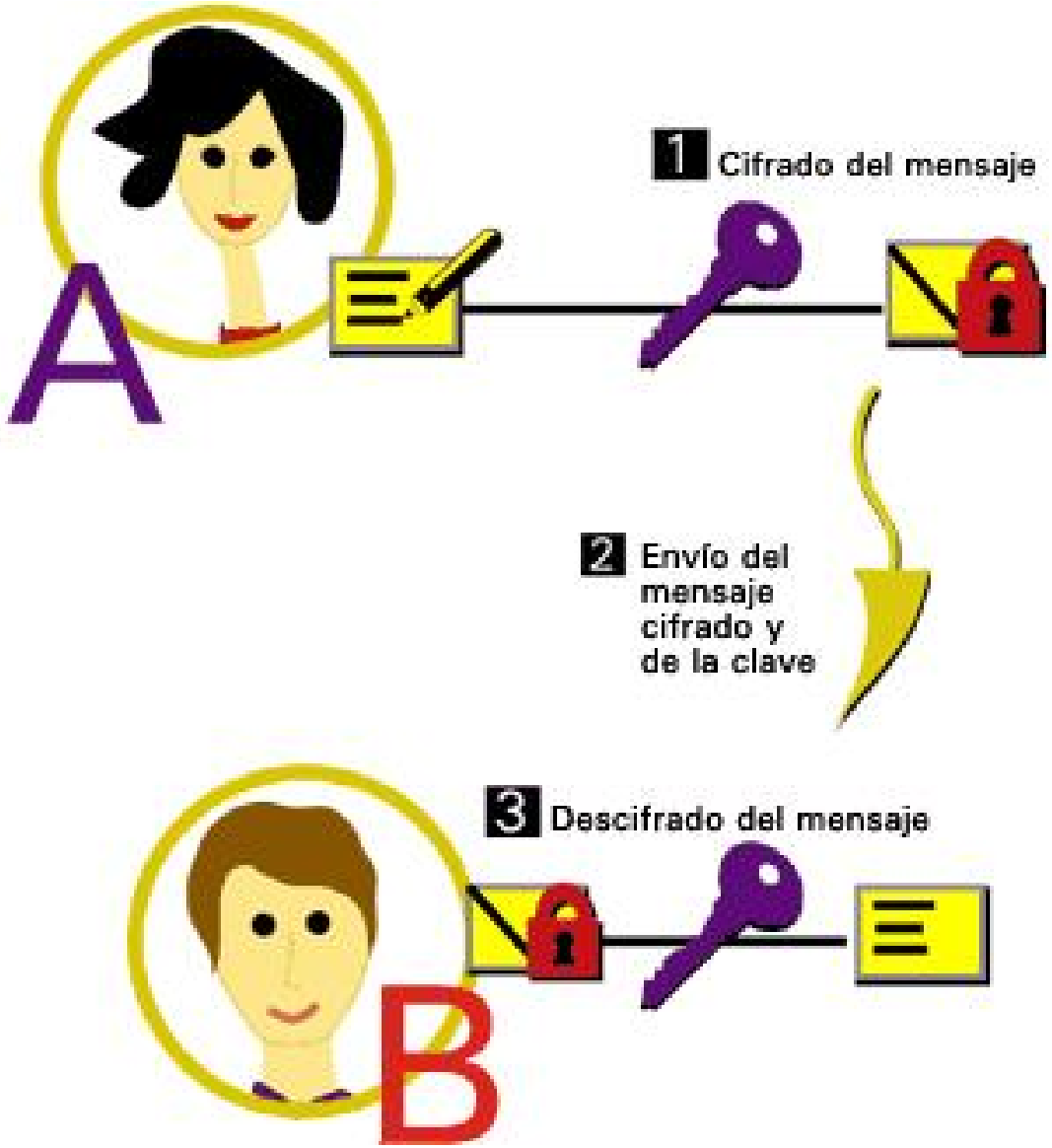
Criptografía

- La **criptografía** sirve para dotar de **seguridad a las comunicaciones**:
  - Confidencialidad
  - Integridad
  - No repudio
- Se distinguen dos **tipos** principales de criptografía:
  - **Simétrica** (o de clave secreta)
  - **asimétrica** (o de clave pública)

Criptografía simétrica

- La **misma clave** se utiliza para cifrar y descifrar el mensaje
- Por tanto, la clave debe mantenerse **secreta** y compartirse por canales seguros $\rightarrow$ *problema de compartición*
- Este tipo de cifrado es bastante **rápido** incluso para cifrar grandes cantidades de datos
- Se distinguen dos tipos principales:
  - De bloque
  - De flujo

Criptografía simétrica



Fuente: `https://www.cert.fnmt.es/curso-de-criptografia/criptografia-de-clave-simetrica`

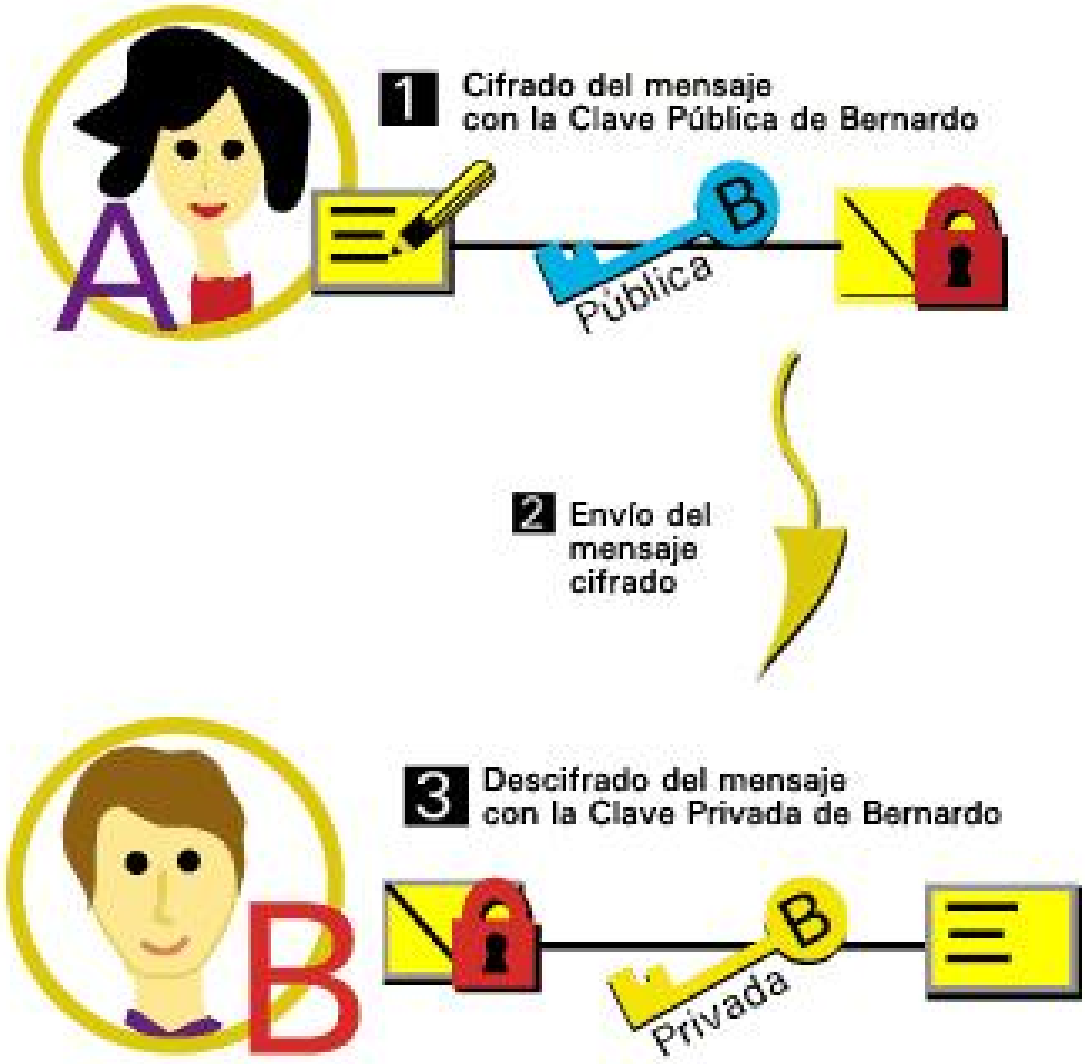
Criptografía asimétrica

- En lugar de una sola clave secreta, cada participante tiene un **par de claves**: una **privada** y una **pública**
- La clave **pública** se puede **distribuir libremente** a cualquier participante
- La clave **privada** solo debe conocerla el propio dueño $\rightarrow$ **secreta**
- El par de claves son **duales**: el mensaje que se cifra con una clave únicamente se puede descifrar con la otra

Criptografía asimétrica

- Además, es **impracticable** obtener la clave privada a partir de la pública
- Tiene varios usos:
  1. Si A firma un mensaje M con su clave privada, cualquiera que lo reciba podrá verificar que proviene de A y no ha sido manipulado → *integridad, no repudio*
  2. Si A cifra el mensaje M con la clave pública de B, solo B podrá leerlo → *confidencialidad*
- La criptografía de clave pública es más **lenta** que la de clave secreta
- **No presenta** *problema de compartición*

Criptografía asimétrica



Fuente: `http://www.cert.fnmt.es/curso-de-criptografia/criptografia-de-clave-asimetrica`

SSL y TLS

- *Secure Sockets Layer* (**SSL**) y su sucesor *Transport Layer Security* (**TLS**) son protocolos criptográficos para comunicación en **redes inseguras**
- Usan el cifrado de **clave asimétrica y simétrica**
- Son los protocolos que utiliza el HTTP seguro (**HTTPS**)

Creación de una conexión TLS

1. El usuario se conecta a un servidor, que le envía su **certificado**. Este certificado:
   - Contiene la clave pública del servidor
   - Está firmado por una autoridad de certificación
2. El navegador **verifica el certificado** del servidor. Para ello utiliza la clave pública de la autoridad de certificación
   - Este proceso puede requerir de varios pasos de verificación usando autoridades de certificación intermedias hasta llegar a una de confianza (FNMT, Verisign, Thawte...)
3. Usando la clave pública del servidor, el usuario envía de manera segura un valor aleatorio al servidor
4. A partir del valor aleatorio conocido únicamente por ambas partes, se genera una clave secreta. Esta clave se usará para cifrar de manera simétrica la comunicación $\rightarrow$ *rapidez*

Referencias sobre criptografía

- Intypedia (UPM)
  - Lista de reproducción completa: `https://youtube.com/playlist?list=PL8bSwVy8_IcMOdOouph8-mFagDEcrXe1w`
  - Introducción a SSL: `https://youtu.be/pOeWmStBOYY`
- *Libro electrónico de seguridad informática y criptografía*. Jorge Ramió Aguirre. `https://criptored.es/guiateoria/gt_m001a.htm`
- *Handbook of applied cryptography*. Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone. `https://cacr.uwaterloo.ca/hac/`

Autenticación

¿Qué es?

- La autenticación es la acción de comprobar que un usuario es quien dice ser
- Es una acción que se produce continuamente en aplicaciones web y de escritorio, apps, etc.
- Se pueden seguir 3 enfoques diferentes:
  1. Algo que sabes
  2. Algo que tienes
  3. Algo que eres

Autenticación basada en algo que sabes

- Es el método preferido, y el que más veces habréis utilizado: *preguntar al usuario algún secreto que únicamente él debería conocer*
- El ejemplo más conocido son las **contraseñas** o las preguntas de verificación (*¿dónde nació tu abuela Eustaquia?*)
- Aspecto **positivo**: **sencillez**
  - Sencillo de implementar e integrar en una aplicación
  - Fácil de entender y utilizar por parte de los usuarios. No hay que explicarles nada nuevo
- Aspectos **negativos**:
  - Los secretos escogidos por los usuarios pueden ser **débiles** (fáciles de adivinar o descubrir) o **inadecuados** para resistir ataques
  - El secreto se debe **reutilizar** en cada autenticación

Autenticación basada en algo que tienes

- Se basa en utilizar algo que únicamente el usuario tiene. Ejemplos:
  - Dispositivos OTP que devuelven la siguiente contraseña a utilizar (o apps en el móvil)
  - Tarjetas de coordenadas del banco
  - Tarjetas inteligentes con chip como el DNIe o las tarjetas de crédito actuales. Suelen necesitar PIN para acceder a su contenido.
- Proporcionan seguridad extra durante la autenticación aunque tienen algunos **aspectos negativos**:
  - El elemento debe estar disponible durante la autenticación
  - El método de generación de claves OTP debe estar bien diseñado y no poder predecirse
  - Se necesitan dispositivos adicionales para leer tarjetas inteligentes con chip

Autenticación basada en algo que eres

- Utiliza distintas técnicas **biométricas**:
  - Huella dactilar
  - Palma de la mano
  - Iris
  - Escáner de retina
  - Reconocimiento de voz
  - Reconocimiento de cara
  - Dinámica de la firma
- En teoría son los que proporcionan la **mayor seguridad**:
  - Los aspectos que miden son únicos, y nadie podrá suplantarte
  - Nunca dejas de ser como eres[1], no lo puedes dejar olvidado en casa

[1] Salvo accidente o evento muy traumático

Autenticación basada en algo que eres

- Sin embargo, este tipo de autenticación tiene **aspectos negativos**
  - Requieren dispositivos adicionales complejos para realizar la autenticación
  - Pueden dar lugar a falsos negativos/positivos
  - Según la técnica puede resultar intrusiva o molesta, y en algunas ocasiones no estar socialmente aceptada
  - Si un atacante obtiene una copia del aspecto medido (p.ej. de la huella digital) no es posible asignar una nueva al usuario legítimo

Autenticación en aplicaciones web

Autenticación en aplicaciones web

- En la inmensa mayoría de aplicaciones web, la autenticación se realiza mediante contraseñas (algo que el usuario sabe)
- En algunos servicios críticos, se puede **complementar** con algo que el usuario tiene, como el teléfono móvil para recibir un PIN temporal (SMS, llamada, app) o una tarjeta de coordenadas $\rightarrow$ **2º factor de autenticación**
- En cualquiera de los casos, el servidor debe **almacenar** las contraseñas del usuario.

Cómo almacenar contraseñas en el servidor

- Existen distintas maneras de almacenar contraseñas de usuarios en nuestros servidores, pero no todas ellas son seguras:
  1. Almacenar contraseña en texto plano
  2. Almacenar hash criptográfico
  3. Almacenar hash criptográfico con sal
  4. **Almacenar hash criptográfico con sal y ralentizado**

Siempre hay que almacenar las contraseñas de los usuarios de la manera más segura posible

Contraseñas en texto plano

- La opción más sencilla es almacenar los secretos directamente en la base de datos de la aplicación web. Ej:

username	**password**	**creation**
pepe	12345678	2020-05-12
ana	password90	2014-06-01
⋮	⋮	⋮

- **Autenticar** a un usuario es comprobar que la **contraseña proporcionada es la misma** que la almacenada en la base de datos
- La seguridad de las contraseñas será la misma que la de la base de datos

Contraseñas en texto plano

- Sin embargo, hay muchas situaciones en las que la seguridad de la base de datos no es suficiente:
  - Inyección de SQL
  - Vulnerabilidades del SGBD, ya sea Oracle, MySQL, PostgreSQL...
  - Vulnerabilidades en el servidor (en el sistema operativo o en alguna de sus aplicaciones)
  - Administrador malintencionado
- En todos estos casos, el atacante podría obtener directamente las contraseñas de los usuarios

Contraseñas en texto plano

- Esto provoca que el atacante pueda acceder a la aplicación web como cualquier usuario. Pero es aún peor...
  - ¿Cuántas cuentas habéis creado en aplicaciones web?
  - ¿Cuántas direcciones e-mail o nombres de usuario diferentes habéis utilizado?
  - ¿Y cuántas contraseñas diferentes?
- Lo más usual es repetir la misma pareja usuario/contraseña para muchas aplicaciones diferentes
- Si almacenamos las contraseñas en texto plano, un descuido o vulnerabilidad puede comprometer la seguridad **de muchas otras aplicaciones web**

Es mucho más seguro almacenar en el servidor un valor *obtenido a partir de la contraseña*, de tal manera que podamos utilizarlo para autenticar

Funciones hash criptográficas

- Para mejorar la seguridad a la hora de almacenar contraseñas, se utilizan **funciones hash criptográficas**
- Una función hash criptográfica recibe como **entrada** un bloque de bits de **tamaño arbitrario** (p.ej. una contraseña representada como una cadena de texto) y genera otro bloque de bits de **tamaño fijo** como **resultado** (llamado ***hash***)
- Las funciones hash criptográficas son **funciones de una sola dirección**:
  - A partir de la entrada, es muy rápido generar el resultado
  - A partir del resultado, es (computacionalmente) muy complejo descubrir la entrada que lo generó

Funciones hash criptográficas

- En una buena función hash criptográfica:
  - Debe ser **sencilla de calcular** (poco cómputo)
  - Debe ser **impracticable** generar una entrada que genera un *hash* dado
  - Debe ser **impracticable** modificar una entrada y que siga generando el mismo *hash*
  - Sebe ser impracticable encontrar dos entradas con el mismo *hash*
- Estas funciones se usan también para calcular *checksums* de ficheros

Funciones hash criptográficas

- Para cifrar las contraseñas en las aplicaciones web se suelen usar algoritmos como:
  - MD5
  - SHA-1
  - **SHA-2 (SHA-256, SHA-512)**
  - **SHA-3**
  - **RIPEMD (RIPEMD-160, RIPEMD-256, RIPEMD-320)**
  - **WHIRLPOOL**
  - **BLAKE** (BLAKE-2, **BLAKE-3**)
- Se recomienda no utilizar MD5 ni SHA-1, ya que se han encontrado formas de calcular colisiones (mensajes con el mismo *hash*)
- Ejemplos:
  - SHA-256('puerta')
    `5bdd32b21baa3ad54b6a23db5aef2960df25adbdb21158acf27e619935b0f5e9`
  - SHA-256('puerto')
    `6c0882925bede3b1b6d79f199be378b06cf45d088121bc3ffd4a9be127d73140`
  - SHA-256('esta es una contraseña muy muy larga')
    `823329e1652ad6f1e8aa0d6122a572335f5fff742f341eaa4e29c363e678e5d5`

Contraseñas almacenadas usando su hash

- Si la contraseña está transformada con una función hash criptográfica, un atacante no la podrá utilizar de manera directa aunque la obtenga

username	**hash**	**creation**
pepe	`ef797c8118f02dfb649607dd5d3f8c7623048c9c063d532cc95c5ed7a898a64f`	2020-05-12
ana	`a5536e54583b2a30f0a9048e0ac4be50d154e5c0b0b736ea23552154628f7142`	2014-06-01
⋮	⋮	⋮

- Sin embargo...
  - Las funciones hash son deterministas: misma entrada $\rightarrow$ misma salida
  - La mayoría de los usuarios utiliza palabras comunes (diccionarios) o contraseñas débiles

Las funciones hash criptográficas no son suficientes

- Se pueden realizar listados precalculados para una función hash criptográfica y una lista de contraseñas concretas (conocidos como *rainbow tables*). P.ej. con SHA-256:

password	**sha-256**
hello	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hbllo	58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
⋮	⋮
waltz	c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542

- Un atacante puede buscar nuestros *hashes* en estos listados, que están disponibles en Internet:
  `http://project-rainbowcrack.com/table.htm`

Las funciones hash criptográficas no son suficientes

- Existen buscadores de los *hashes* de las palabras más comunes
- Podéis probar a calcular el *hash* de alguna contraseña en `https://www.fileformat.info/tool/hash.htm`
- Y luego comprobar si podéis obtener la contraseña original en `https://dehash.me`

Fortalecer los *hashes*

- Los *hashes* de las claves débiles aparecerán en los listados, así que el atacante podrá obtener las contraseñas originales.
- ¿Cómo evitarlo?
  - Forzando a los usuarios a utilizar **contraseñas fuertes**: largas, con variedad de caracteres $\rightarrow$ **mala idea**
  - **Añadir algo más a la clave antes de calcular su *hash***, de tal manera que el *hash* resultante de cada clave sea diferente en cada aplicación web incluso para usuarios con la misma clave

Fortalecer los *hashes*

- Para evitar el problema anterior, se añade **sal** a la contraseña **antes** de calcular el *hash*
- La sal es una cadena que se **concatena** a la contraseña como prefijo o sufijo. Se debe generar de manera **aleatoria** y almacenar en la BD junto con el resto de datos de cada usuario
- La sal conseguirá que a partir de claves iguales se obtengan *hashes* diferentes:
  - `hello` $\rightarrow$
    `2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824`
  - `hello + QxLUF1bgIAdeQX` $\rightarrow$
    `9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1`
  - `hello + bv5PehSMfV11Cd` $\rightarrow$
    `d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab`
  - `hello + YYLmfY6IehjZMQ` $\rightarrow$
    `a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007`

Consejos para el uso de la sal

- Para cada contraseña, hay que generar una **sal nueva y aleatoria**
- **No hay que reutilizar la sal**, ni entre usuarios ni dentro del mismo usuario. Si un usuario actualiza su contraseña hay que generar una nueva sal
- La **sal debe ser larga**. Como norma general tan larga como la salida de la función *hash* criptográfica
- La sal **no necesita ser secreta**, se puede almacenar tal cual en la BD junto con el valor *hash* obtenido

Contraseñas almacenadas con hash y sal

- La sal se almacenará en nuestra base de datos junto con el resultado de la función hash criptográfica:

username	salt	hash	creation
pepe	`QxLUF1bgIAdeQX`	`924ba4731e669b793919…635c`	2020-05-12
ana	`YYLmfY6IehjZMQ`	`6511b2429b4ceb3c8202…1105`	2014-06-01
⋮	⋮	⋮	⋮

- En ocasiones la sal se complementa con **pimienta**, un valor constante que está incrustado en el código del programa y **nunca almacenado**
- Antes de usar la función hash criptográfica, se concatenan tanto la sal como la pimienta a la contraseña del usuario (prefijo y sufijo, por ejemplo)
- Se pueden tener varios valores de pimienta en el sistema. Para autenticar se deberían probar todos

Cuando la sal no es suficiente

- La sal dificulta el uso de tablas precalculadas para obtener las contraseñas originales a partir de sus *hashes*. Esto evita que un atacante obtenga muchas contraseñas con poco esfuerzo
- Pero si un atacante roba tus *hashes* junto con la sal y tiene **mucho interés en conocer la clave de un usuario concreto**, puede realizar un ataque de fuerza bruta centrado en ese usuario
- Las funciones hash criptográficas son rápidas, lo que en esta situación ayuda al atacante. *¿Cómo evitarlo?*

Cuando la sal no es suficiente

- Para mitigar los ataques de fuerza bruta se pueden utilizar algoritmos de ralentizado como **PBKDF2**, o funciones de derivación de claves como **bcrypt**, **scrypt** o **Argon2**
- Básicamente aplican iterativamente funciones hash criptográficas para obtener un resultado derivado. Disponen de parámetros para configurar cuántos recursos consumen (tiempo y memoria)
- Es necesario encontrar un **equilibrio** entre **velocidad** de *hashing* y **seguridad**, o puedes acabar sobrecargando tu servidor a base de autenticaciones muy costosas

Contraseñas almacenadas con *hash*, sal y ralentizado

- Con el tiempo siempre aumenta la capacidad de cómputo general, así que será necesario aumentar el número de iteraciones
- Es común almacenar varios datos sobre la contraseña: algoritmo de ralentizado aplicado, parámetros de coste, sal, etc. P.ej. estos serían los datos almacenados usando **Argon2**:

username	**hash**	**creation**
pepe	`$argon2i$v=19$m=16,t=2,p=1$RldZaXBjTTdVbGpJblFqUA$EhDmbuN1P8gIfw`	2020-05-12
ana	`$argon2i$v=19$m=19,t=3,p=2$MkVuQjVZSWd4Zk5SWmNWMg$8J7gQ569altN`	2014-06-01
⋮	⋮	⋮

- Esto permite aumentar coste o actualizar a una función hash más segura durante una autenticación con éxito de un usuario **(único momento donde tenemos la clave real en texto plano)**

Transmitir la clave

- Siempre que un usuario introduzca una clave, debe ser en una página HTTPS:
  - Podrá confirmar en qué web está introduciendo la clave
  - La clave se enviará cifrada con la contraseña de sesión TLS
- Siempre aplicar la función hash en el servidor (*back-end*). Si quieres puedes aplicar hash también en el cliente (*front-end*), pero nunca sólo en el cliente

Referencias sobre autenticación en aplicaciones web

- OWASP Password Storage Cheat Sheet:
  `https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html`
- OWASP Authentication Cheat:
  `https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html`
- OWASP Transport Layer Protection Cheat Sheet:
  `https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html`
- Salted Password Hashing - Doing it Right:
  `https://crackstation.net/hashing-security.htm`

Time-based One-time Passwords (TOTP)

Debilidad de las contraseñas

- Autenticar usuarios únicamente por su contraseña no es muy seguro:
  - Los usuarios **repiten** mucho sus claves entre varias aplicaciones web
  - Las contraseñas suelen **débiles**: cortas y fáciles de adivinar. Ej: `password`, `1234`, `asdf1234`
  - Las contraseñas **no se cambian con frecuencia**
- Cualquier atacante que conozca la clave (no importa cómo: *man-in-the-middle*, ingeniería social, ataque a otra web) podrá acceder a la cuenta sin límites
- Para mitigar esta situación se utiliza una **autenticación** basada en **varios factores**. El caso más común son 2 factores:
  1. **Contraseña** escogida por el usuario (algo que sabe)
  2. ***One-Time password*** *(OTP)*: generada o transmitida a un dispositivo del usuario (algo que tiene). Ej: SMS al móvil, HOTP o TOTP en el móvil, dispositivos dedicados.

Segundo factor de autenticación

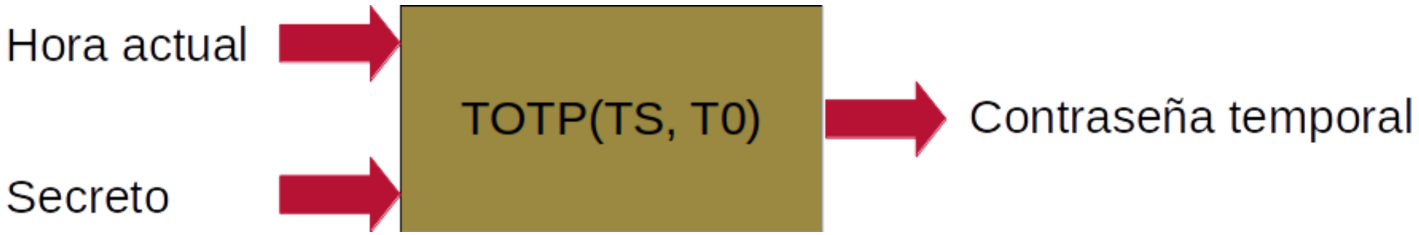
- Si la contraseña principal es comprometida pero el atacante no tiene acceso al dispositivo, no podrá acceder a la cuenta → carece de la OTP actual
- Idealmente, los sistemas deben avisar al usuario cada vez que un acceso es denegado por introducir una OTP incorrecta → posible acceso ilegítimo debido a una contraseña principal comprometida
- Existen dos tipos de OTP:
  - Basadas en secuencia: generan la OTP en base a un secreto y un contador que se incrementa con cada contraseña generada. Ej: ***HMAC-based OTP (HOTP)***
  - Basadas en tiempo: generan la OTP en base a un secreto y la hora actual. Tienen una corta duración (usualmente 30 segundos). Ej: **Time-based OTP (TOTP)**.

Segundo factor de autenticación

- Incluso aunque un atacante descubra nuestra contraseña y la última OTP utilizada, el sistema será seguro:
  - HOTP: La próxima OTP a usar siempre será diferente de la última utilizada, y no es posible calcularla a partir de esta última
  - TOTP: La siguiente OTP a usar será igual a la última utilizada únicamente si estamos en la misma ventana temporal $\rightarrow$ ese será el tiempo máximo de vulnerabilidad (usualmente 30 segundos)

TOTP

- Para calcular el TOTP actual hacen falta 2 valores:
  - la **hora actual** (se obtiene directamente del sistema)
  - el **secreto** (solo el servidor y el usuario lo conocen, se lo intercambian a través de un código QR)



- También existen 2 parámetros que es necesario establecer, aunque sus valores por defecto usualmente son adecuados:
  - (TS) Ventana temporal: 30 segundos
  - (T0) Fecha inicial: 1 de enero 1970 (tiempo UNIX)

TOTP

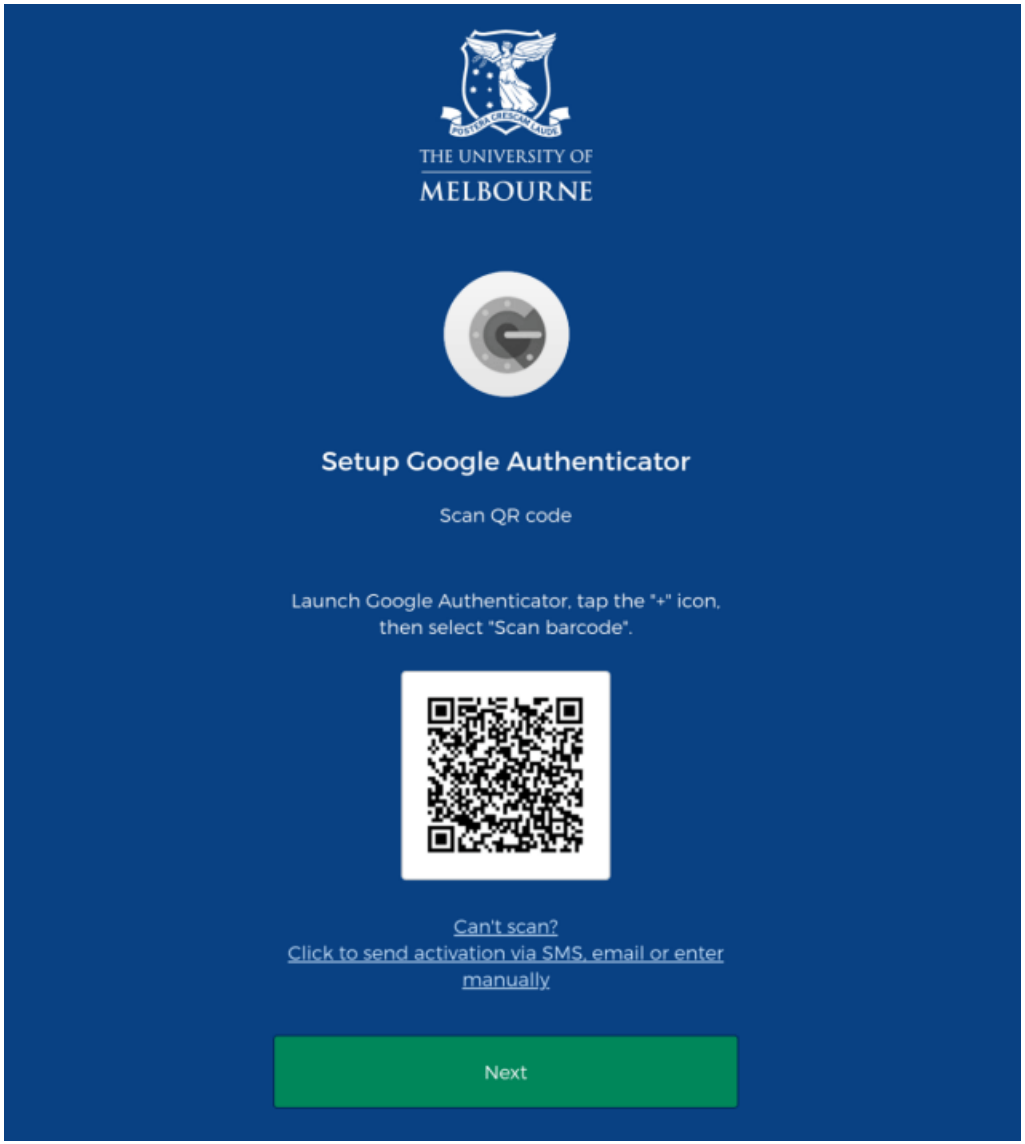
- El cálculo de la clave temporal se basa en funciones hash criptográficas, y es perfectamente conocido: `https://en.wikipedia.org/wiki/Time-based_One-time_Password_Algorithm`
- La **robustez** del sistema proviene de que es **impracticable** obtener el secreto a partir de una o varias de las claves temporales generadas

Autenticación con TOTP (I)

Para incorporar TOTP como segundo factor de autenticación en una aplicación web es necesario un primer paso de configuración:

1. La aplicación web genera un **secreto** para el usuario y lo almacena en su perfil
2. La aplicación muestra el secreto al usuario y este lo registra en su aplicación TOTP (FreeOTP Authenticator, Google Authenticator, Microsoft Authenticator, Latch, etc.). Este paso se debe agilizar mediante **códigos QR**

Compartir secreto TOTP



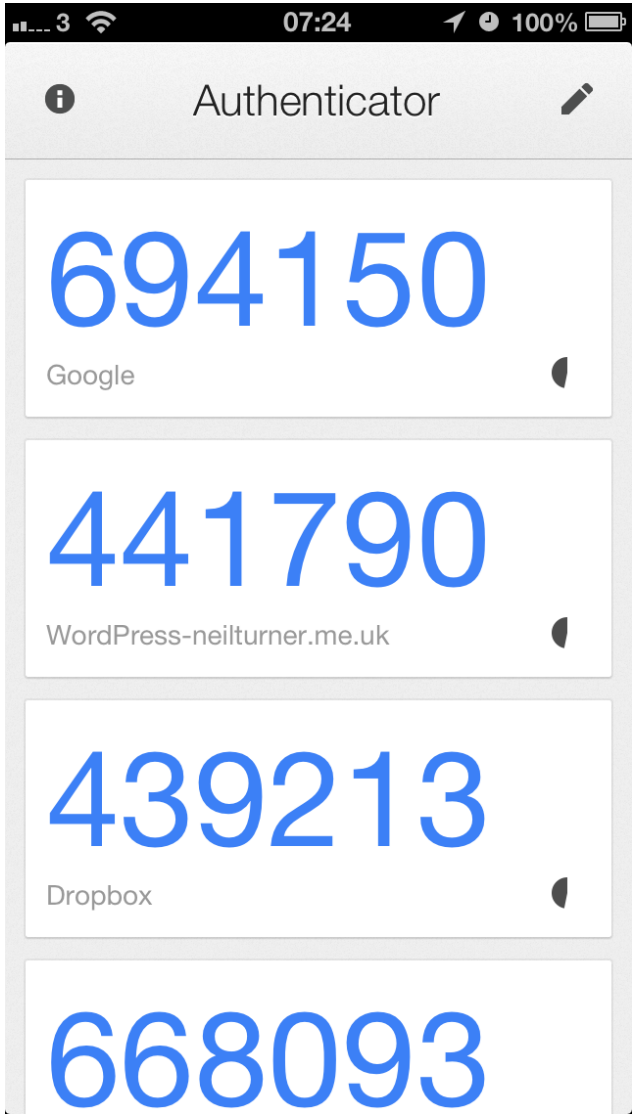
Fuente: `https://blogs.unimelb.edu.au/sciencecommunication/2021/09/30/totp/`

Autenticación con TOTP (II)

Cuando un usuario quiere acceder al sistema, la aplicación web debe realizar estos pasos:

1. Solicitar un nombre de usuario, contraseña y valor TOTP. Se pueden pedir los 3 valores a la vez o primero el nombre y la contraseña en una página y el valor TOTP en la siguiente página
2. A partir del secreto del usuario almacenado en la BD y la hora actual, generar un TOTP interno $\rightarrow$ comparar con el TOTP enviado por el usuario
3. Si las contraseñas coinciden y el TOTP también, la autenticación tiene éxito
4. (**Buenas prácticas**) En caso de fallo de TOTP, notificar al correo del usuario porque puede tratarse de un atacante

Autenticación con TOTP: vista del usuario



Fuente: `https://www.cyberriskopportunities.com/how-to-use-google-authenticator/`

Contraseñas almacenadas con *hash*, sal, ralentizado y TOTP

- Para soportar un segundo factor de autenticación con TOTP necesitamos **almacenar también en la BD el secreto TOTP para cada usuario**

username	hash	TOTP_secret	creation
pepe	`$argon2i$v=19$m=16,t=2,p=1$RldZaXBj...gIfw`	`ARLMT3FMCVENDNCY36R6Q2WTGKFZZVDB`	2020-05-12
ana	`$argon2i$v=19$m=19,t=3,p=2$MkVuQjVZ...altN`	`KIOK7IVUFL6Y5L3UHD435N7PR5H4VA3D`	2014-06-01
.	.	.	.
.	.	.	.
.	.	.	.

Aspectos de TOTP

- A diferencia de la contraseña, el secreto para TOTP solo se comparte una vez, al configurar la cuenta
- Sin embargo, cualquiera que conozca el secreto podrá generar el valor TOTP para cualquier ventana temporal
- Tanto la aplicación web como el dispositivo del usuario que genera TOTP deben estar sincronizados $\rightarrow$ *Network Time Protocol*
- Como la ventana temporal suele ser de 30 segundos, pequeñas diferencias no afectan a la usabilidad

TOTP en Python

- Es muy sencillo usar TOTP en Python con bibliotecas como `pyotp`:
  - Instalación en vuestro ordenador (en los laboratorios ya está instalado)

    ```
    $ pip install pyotp
    ```

  - Web del proyecto: `https://github.com/pyauth/pyotp`
- Generar un secreto aleatorio

  ```
  >>> pyotp.random_base32()
  'L7S2BXCKUS6LMXB7'
  ```

- Generar un TOTP con la hora actual y un secreto `secret`:

  ```
  >>> totp = pyotp.TOTP(secret)
  >>> totp.now()
  '905511'
  ```

- Verificar si `t` es un TOTP válido para la hora actual y el secreto:

  ```
  >>> totp.verify(t)
  True
  ```