Autenticación

Pimentar

Se puede utilizar una pimienta además del salado para proporcionar una capa adicional de protección. Es similar a la sal, pero tiene cuatro diferencias clave:

- La pimienta se **comparte entre todas las contraseñas almacenadas**, en lugar de ser *única* como la sal. Esto hace que la pimienta sea predecible y que los intentos de descifrar el hash de una contraseña *sean probabilísticos*. La naturaleza estática de una pimienta también "debilita" la resistencia a las colisiones de hash, mientras que la sal mejora la resistencia a las colisiones de hash al ampliar la longitud con caracteres únicos que aumentan la entropía de la entrada a la función de hash.
- La pimienta **no se almacena en la base de datos**, a diferencia de muchas implementaciones de una sal de contraseña (pero no siempre es cierto para una sal).
- La pimienta no es un mecanismo para hacer que el descifrado de contraseñas **sea demasiado difícil** para un atacante, como pretenden hacer muchas protecciones de almacenamiento de contraseñas (entre ellas la sal).
- Una sal evita que los atacantes compilen tablas de arco iris de contraseñas conocidas, sin embargo una pimienta no ofrece esta característica

El propósito de la pimienta es evitar que un atacante pueda descifrar cualquiera de los hashes si sólo tiene acceso a la base de datos, por ejemplo si ha explotado una vulnerabilidad de inyección SQL u obtenido una copia de seguridad de la base de datos.

La pimienta debe tener *al menos* 32 caracteres y debe generarse aleatoriamente utilizando un generador pseudoaleatorio seguro (CSPRNG). Debe almacenarse de forma segura en una "bóveda de secretos" (no en un archivo de configuración de la aplicación, independientemente de los permisos de los archivos que son susceptibles de SSRF) utilizando las API de acceso seguro, o para un almacenamiento seguro óptimo almacenar la pimienta en un Módulo de Seguridad de Hardware (HSM) si es posible.

La pimienta se utiliza a menudo de forma similar a la sal, concatenándola con la contraseña antes del hash, utilizando una construcción como `hash($pepper`

`. $password)`. Mientras que la concatenación se considera apropiada para una sal, sólo el prefijo se considera apropiado para una pimienta.

Nunca coloque una pimienta como sufijo, ya que esto puede conducir a vulnerabilidades tales como problemas relacionados con el truncamiento y los ataques de extensión de longitud. Prácticamente estas amenazas permiten que el componente de la contraseña de entrada se valide con éxito porque la contraseña única nunca se trunca, sólo la pimienta probabilística se truncaría.

Incrementar el número de iteraciones o work factor

Otra forma de aumentar la seguridad es repetir el número de iteraciones de hash. Aumentar el número de iteraciones significa que vamos a codificar la contraseña varias veces.

A la hora de elegir un factor de trabajo, hay que encontrar un equilibrio entre seguridad y rendimiento. Los factores de trabajo más altos harán que los hashes sean más difíciles de descifrar para un atacante, pero también harán que el proceso de verificación de un intento de inicio de sesión sea más lento. Si el factor de trabajo es demasiado alto, esto puede degradar el rendimiento de la aplicación, y también podría ser utilizado por un atacante para llevar a cabo un ataque de denegación de servicio haciendo un gran número de intentos de inicio de sesión para agotar la CPU del servidor.

No existe una regla de oro para el factor de trabajo ideal: dependerá del rendimiento del servidor y del número de usuarios de la aplicación. La determinación del factor de trabajo óptimo requerirá la experimentación en el servidor o servidores específicos utilizados por la aplicación. Como regla general, el cálculo de un hash debería tardar menos de un segundo, aunque en los sitios de mayor tráfico debería ser bastante menos que esto.

Otra forma de aumentar la seguridad es repetir el número de iteraciones de hash. Aumentar el número de iteraciones significa que vamos a codificar la contraseña varias veces. Por ejemplo, con sha512 tenemos el siguiente bucle:

```
Siempre que la iteración sea mayor que 0
hash = sha512 (hash)
Disminuir la iteración
```

Para un usuario que inicia sesión, el cálculo del hash será más largo (aún toma un milisegundo). Pero cuando un usuario pierde unos milisegundos para iniciar sesión, un atacante perderá mucho más tiempo, porque el atacante perderá varios milisegundos por intento, y dado que el atacante realiza millones de intentos, esto dará como resultado horas / días adicionales para recuperar las contraseñas.

El factor de trabajo es esencialmente el número de iteraciones del algoritmo de hashing que se realizan para cada contraseña (normalmente es realmente `2^work` iteraciones). El propósito del factor de trabajo es hacer que el cálculo

del hash sea más caro computacionalmente, lo que a su vez reduce la velocidad a la que un atacante puede intentar descifrar el hash de la contraseña. El factor de trabajo se suele almacenar en la salida del hash.

Fusionar los 3 métodos

Podemos fusionar los tres métodos (sal, pimienta y número de iteraciones) para tener un método para almacenar contraseñas de forma más segura que un simple hash.

```
Function calculation_hash(password,salt,pepper,iteration)
Inputs
password is the user's password in plain text
salt is the unique salt per user and is randomly generated
pepper is the common pepper for all users and is randomly generated.
iteration is the number of iterations

Output:
The password hash
Hash = sha512(salt+password+pepper)
As long as iteration is greater than 0
hash = sha512(hash)
Decrement iteration

return hash
```

Luego, para verificar las contraseñas al iniciar sesión, simplemente se llama a la misma función con la contraseña ingresada por el usuario y compárela con el hash en la base de datos. Si ambos son idénticos, entonces el inicio de sesión es exitoso.

Usar funciones específicas

Escribir código criptográfico personalizado, como un algoritmo hash, es **realmente difícil** y **nunca** debería **hacerse** fuera de un ejercicio académico. Cualquier beneficio potencial que puedas tener al usar un algoritmo desconocido o hecho a medida será ampliamente eclipsado por las debilidades que existen en él.

No lo hagas.

Algoritmos modernos

Hay una serie de algoritmos de hashing modernos que han sido diseñados específicamente para almacenar contraseñas de forma segura. Esto significa que deben ser lentos (a diferencia de algoritmos como MD5 y SHA-1, que fueron diseñados para ser rápidos), y su lentitud puede configurarse cambiando el factor de trabajo.

A continuación se enumeran los tres principales algoritmos que deberían considerarse:

Argon2id Argon2 es el ganador del Concurso de Hashing de Contraseñas de 2015. Hay tres versiones diferentes del algoritmo, y la variante Argon2id debería usarse cuando esté disponible, ya que proporciona un enfoque equilibrado para resistir tanto los ataques de canal lateral como los basados en la GPU.

En lugar de un simple factor de trabajo como otros algoritmos, Argon2 tiene tres parámetros diferentes que pueden configurarse, lo que significa que es más complicado ajustarlo correctamente para el entorno. La especificación contiene una guía para elegir los parámetros adecuados, sin embargo, si no estás en condiciones de ajustarlo correctamente, entonces un algoritmo más simple como Bcrypt puede ser una mejor opción.

PBKDF2 PBKDF2 está recomendado por el NIST y tiene implementaciones validadas por FIPS-140. Por lo tanto, debería ser el algoritmo preferido cuando se requiera. Además, es compatible con el marco de trabajo de .NET, por lo que se utiliza comúnmente en aplicaciones ASP.NET.

PBKDF2 puede utilizarse con HMACs basados en un número de algoritmos hash diferentes. El HMAC-SHA-256 está ampliamente soportado y es recomendado por el NIST.

El factor de trabajo para PBKDF2 se implementa a través del recuento de iteraciones, que debe ser de al menos 10.000 (aunque valores de hasta 100.000 pueden ser apropiados en entornos de mayor seguridad).

Bcrypt Bcrypt es el algoritmo más ampliamente soportado y debería ser la elección por defecto a menos que haya requisitos específicos para PBKDF2, o conocimientos apropiados para ajustar Argon2.

El factor de trabajo por defecto para Bcrypt es 10, y por lo general debe ser elevado a 12 a menos que se opere en sistemas más antiguos o de menor potencia.**Bcrypt**

bcrypt es una función hash creada por Niels Provos y David Mazières. Se basa en el algoritmo de cifrado Blowfish y se presentó en USENIX en 1999.

Entre estos puntos positivos además de los mencionados anteriormente encontramos implementaciones en muchos idiomas. Además, dado que este algoritmo se remonta a 1999, ha mostrado su robustez a lo largo del tiempo, donde algunos algoritmos como Argon2 (i) solo existen desde 2015.

El hash calculado por bcrypt tiene una forma predefinida:

```
$2y$11$SXAXZyioy60hbnymeoJ9.ulscXwUFMhbvLaTxAt729tGusw.5AG4C
```

- Algoritmo: Este puede tomar varias versiones dependiendo de la versión de bcrypt (2, 2a, 2x, 2y y 2b
- El costo: el número de iteraciones en potencia de 2. Por ejemplo, aquí, la iteración es 11, el algoritmo hará 211 iteraciones (2048 iteraciones).
- Sal: en lugar de almacenar la sal en una columna dedicada, se almacena directamente en el hash final.
- La contraseña hash

Dado que bcrypt almacena el número de iteraciones, esto la convierte en una función adaptativa, porque el número de iteraciones se puede aumentar y, por lo tanto, es cada vez más largo. Esto le permite, a pesar de su antigüedad y la evolución de la potencia informática, seguir siendo robusto contra los ataques de fuerza bruta. El siguiente punto de referencia muestra que el `hashcat` tarda 23 días en calcular la totalidad de los hashes de `rockyou`.

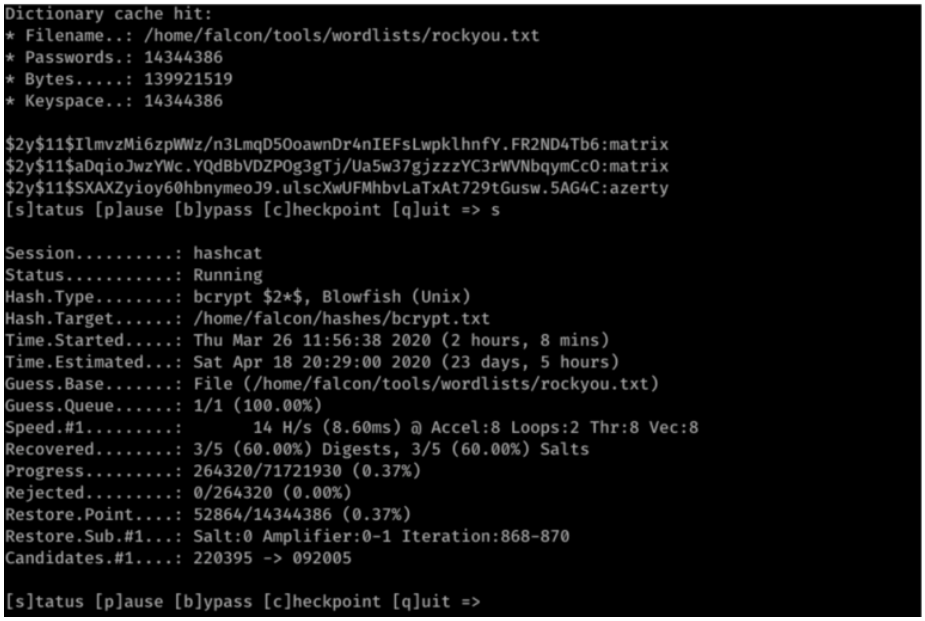

Figure 1: https://www.vaadata.com/blog/wp-content/uploads/2020/05/bcrypt-1-768x514.png

Conclusión

Hemos visto en este artículo la utilidad de una función hash robusta y la ventaja de utilizar una función ya existente. Además, el problema del almacenamiento de contraseñas tiene problemas legales además de problemas de seguridad.

Finalmente, es interesante notar que en todos los casos las contraseñas `azerty` y `matrix` se encontraron rápidamente, mientras que la contraseña `yep59f$4txwrr` nunca se encontró. De hecho, como éste no está presente en ninguna lista, la única forma de encontrarlo es realizar una fuerza bruta exhaustiva en 13 caracteres, lo que es una operación que requiere mucho tiempo (debido a la gran cantidad de posibilidades de contraseña). Esto también muestra lo importante que es para una aplicación web forzar una política de contraseña segura.

Basado en

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

https://stackoverflow.com/questions/674904/salting-your-password-best-practices

https://security.stackexchange.com/questions/3272/password-hashing-add-salt-pepper-or-is-salt-enough

https://www.vaadata.com/blog/how-to-securely-store-passwords-in-database/

https://es.wikipedia.org/wiki/Tabla_arco%C3%ADris

https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

https://md5hashing.net/hash/md5/21b72c0b7adc5c7b4a50ffcb90d92dd6

https://platzi.com/blog/introduccion-json-web-tokens/

https://www.bbva.com/es/que-es-fido-el-nuevo-estandar-de-autenticacion-online/