Almacenamiento seguro de contraseñas de usuarios

Recomendación
Abril/2019

José Ramón Revuelta
j.r.revuelta@computer.org

Este documento describe una recomendación para el manejo de contraseñas de usuarios en los sistemas, principalmente de servicios a clientes en instituciones financieras. En materia de su procesamiento para almacenar y verificar las contraseñas al interactuar con los usuarios del sistema, siguiendo las mejores prácticas y haciendo uso de los dispositivos de alta seguridad HSM, en cumplimiento con el marco normativo para la banca en México.

Contenido

Introducción

Se ha reportado un enorme número de casos en los medios acerca de sitios web y otros sistemas importantes de cómputo de los cuales ha sido extraída de sus bases de datos información de los usuarios incluyendo, por supuesto, las contraseñas. Esto es una vulnerabilidad de otros sistemas que podríamos observar con cierta tranquilidad pensando que es ajena, sin embargo, se trata de una vulnerabilidad propia, y veremos que esa vulnerabilidad en otros sistemas tiene un impacto directo en los nuestros.

Es muy común para las personas hacer uso de la misma contraseña a través de los diversos sistemas que usa. La complicación en las reglas que se exigen para hacer una buena contraseña hace que se haga cada vez más difícil para las personas crear contraseñas robustas, y más si tratan con una enorme cantidad de servicios en línea y redes sociales, por lo que terminan reutilizando la misma contraseña.

Sin embargo, es importante sensibilizar a los usuarios de un sistema bancario donde se encuentran depositados sus recursos patrimoniales que, para este caso particular, hagan uso de una contraseña diferente a todas las demás que usan en redes sociales y otros sistemas. Si otro sitio con menores exigencias de seguridad es comprometido, sus credenciales serían expuestas y podrían ser utilizadas para tener acceso a sus registros y hacer operaciones en el sistema bancario.

Aún así, es una responsabilidad muy importante de cualquier institución salvaguardar la confidencialidad de las contraseñas de sus usuarios, asegurando así que en caso de que la información custodiada se vea comprometida, ésta no pueda ser explotada de ninguna manera en perjuicio de nuestros usuarios y clientes.

Antecedentes

Es muy importante para un banco asegurar el resguardo de las contraseñas con la máxima seguridad disponible, así como una obligación impuesta por las autoridades y las leyes que regulan a instituciones financieras en México.

Las contraseñas deben ser protegidas en múltiples momentos en el ciclo del sistema y en varias dimensiones, esto es; la contraseña es introducida por el usuario en su propia computadora conectada por Internet, esto obliga a la aplicación web del banco a protegerla desde ese momento, después, en tránsito a los servidores del banco, estableciendo una línea de comunicaciones segura, finalmente, en el tratamiento durante su verificación en la aplicación y su resguardo al almacenarla en las bases de datos.

Para salvaguardar las contraseñas se han intentado muchas técnicas, incluyendo el cifrado de la contraseña utilizando algún algoritmo muy robusto. Sin embargo, esto tiene dos grandes inconvenientes; (a) toda la seguridad de la contraseña depende de la llave usada para su cifrado, y más importante (b) al guardar una contraseña cifrada, es posible *descifrar* la contraseña.

Esto es muy delicado; una contraseña solamente se necesita en el sistema para verificar al usuario del sistema, pero ninguna otra persona debe tener acceso a esa contraseña en absoluto. Al guardar una contraseña cifrada, se arriesga la posibilidad de que alguna otra persona (quizás con habilidades técnicas, o con privilegios suficientes en el sistema) pueda tener acceso al valor de la contraseña. En una institución bancaria, donde la contraseña es utilizada por un cliente para realizar movimientos con sus recursos patrimoniales, se debe prevenir a toda costa la posibilidad de que cualquier otra persona, sin importar si es de confianza en el banco, pueda tener la posibilidad de conocer contraseñas de los clientes. Las contraseñas NO DEBEN cifrarse. Ver: *Marco Regulatorio*.

Entonces, la alternativa utilizada en este caso debe ser un proceso criptográfico seguro; que garantice que la contraseña pueda ser verificada cuando el usuario está presente, y además se pueda conservar en el sistema de una forma no recuperable en su forma original. Esto se logra mediante algoritmos de *"digestión"* (conocidos en inglés como *hash criptográfico*). Estos algoritmos permiten procesar contenidos de texto claro, y transformarlos en una cadena única de bits, pero no puede reversarse y volver al valor original. Además, es muy remota la posibilidad de encontrar otro valor que pueda generar la misma cadena de bits (tanto así, que se considera imposible para efectos prácticos). Las contraseñas DEBEN ser sometidas a un proceso de *digestión criptográfica segura*.

Marco Regulatorio

En México, las instituciones financieras, bancos y otras empresas similares, están reguladas por el marco regulatorio establecido por la ley y por las disposiciones emitidas por los organismos regulatorios para el propósito de complementar lo dispuesto en la ley cuando ésta así lo indica.

Tal es el caso de la conocida CUB (Circular Única de Bancos) emitida por la CNBV (Comisión Nacional Bancaria y de Valores, Secretaría de Hacienda y Crédito Público, 2005-2018). La cual dedica el Capítulo X "Del uso del servicio de Banca Electrónica" a todas las consideraciones en materia de sistemas de cómputo e informática que la institución debe cuidar, particularmente, en materia de seguridad informática.

En referencia al punto discutido anteriormente respecto a la inconveniencia de mantener cifradas las contraseñas (en lugar de hacer una digestión criptográfica), las disposiciones emitidas establecen la siguiente restricción:

> ***Artículo 316 Bis 4.-*** *Para el manejo de Contraseñas y otros Factores de Autenticación, las Instituciones se sujetarán a lo siguiente:*
> *…*
> *II. Tendrán prohibido contar con mecanismos, algoritmos o procedimientos que les permitan conocer, recuperar o descifrar los valores de cualquier información relativa a la Autenticación de sus Usuarios.*

Este artículo prohíbe explícitamente almacenar las contraseñas de los usuarios en forma cifrada.

Ataques a contraseñas de usuarios

Hay dos modalidades generales para atacar a un sitio con el fin de extraer las contraseñas de sus usuarios; en línea, y fuera de línea. En el primer caso, el atacante intenta obtener contraseñas de los usuarios de un sitio mediante intentos repetidos de entrar en el sistema en línea probando con diferentes identificadores de usuario y con diferentes contraseñas hasta encontrar un acierto. En el segundo caso, el atacante obtiene la información almacenada en la base de datos de un sistema por cualquier mecanismo y una vez en posesión de estos datos, intenta descifrar las contraseñas almacenadas mediante varias formas de cripto-analizar esa información.

Los mecanismos más comunes para obtener la información de las bases de datos incluyen: Ataques en línea mediante inyección de SQL, troyanos y gusanos, y ataques de ingeniería social o hasta corrupción y extorsión para obtener la base de datos, incluyendo la obtención de bitácoras (*logs y journals*), extracciones (*dumps y queries*) y respaldos.

Cuando se logra obtener la información contenida en la base de datos, se puede montar un ataque fuera de línea con cualquier cantidad de recursos y tiempo disponible para su análisis y explotación. Por lo que, es importante considerar este escenario al guardar información como las contraseñas, ya que si se logra recuperar una contraseña de una base de datos robada sin que se sepa de su fuga, se podría hacer uso de las credenciales del usuario para hacer operaciones de forma normal en el sistema usurpando la identidad del cliente.

La información de las credenciales contenida en la base de datos del sistema debe ser guardada de una forma tal, que aún bajo el escenario del ataque fuera de línea, es decir, que el contenido de la base de datos se vea comprometido, no sea posible la extracción de información utilizable. El objetivo principal de esta recomendación va dirigido a proteger la información de contraseñas de usuarios guardadas en la base de datos del sistema en caso de ser comprometida, sobre todo para la modalidad de ataques fuera de línea.

Recomendaciones para prevenir ataques en línea

Los ataques en línea se llevan a cabo haciendo intentos de entrada al sistema directamente en el sistema mismo. Normalmente se hacen con programas (conocidos como "*bots*", o "robots") que de manera automática se comunican al sistema web simulando ser usuarios normales, pero con gran velocidad pueden repetir sus intentos de penetrar en el sistema.

Para prevenir los ataques con robots, lo que se hace es que el sistema previene la repetición de los intentos de varias formas; como utilizar los conocidos "*captchas*" (que evitan que un mecanismo automático realice los intentos) o aplicando válvulas que van reduciendo la velocidad de respuesta del sistema conforme se van haciendo intentos (permiten el siguiente intento después de un periodo creciente de tiempo), sin embargo, los bancos en México están obligados a realizar algo más drástico: después de 5 intentos fallidos, el sistema debe bloquear por completo al usuario hasta que sea revalidado por otro medio para poder desbloquear nuevamente su acceso.

Finalmente, una forma más robusta de prevenir estos ataques es mediante un segundo factor de autenticación como lo es un generador de contraseñas de uso en una sola ocasión (*OTP - one time password*). Con esto, cada intento de entrada al sistema debe incluir además de la contraseña del usuario, un OTP adicional generado por el dispositivo. De esta forma, si cualquiera de los elementos no es correcto, el intento de entrar al sistema falla, no dejando saber al atacante si la contraseña que intentó usar era correcta o no.

Modalidad de ataques fuera de línea

Cuando la información contenida en la base de datos es comprometida y un atacante obtiene los registros con la información de los usuarios y sus contraseñas, éste puede montar un ataque muy elaborado usando sus propios recursos de cómputo y el tiempo que tenga disponible.

Vamos a suponer que las contraseñas del sistema se encuentran almacenadas en la base de datos en una forma ***digerida criptográficamente fuerte***. Es decir que fue utilizada una función bien aceptada para hacer el *hashing* como lo es SHA-2 (puede ser SHA-512, que se trata de la modalidad en teoría más segura). Con esto sabemos que las digestiones guardadas no son reversibles por ningún mecanismo conocido, y que es prácticamente imposible encontrar una cadena de entrada que produzca la misma digestión de salida que conocemos.

La forma más simple e intuitiva de atacar es conocida como ***"ataque de fuerza bruta"*** y recibe este nombre porque es un procedimiento en el que sistemáticamente se intentan todas las posibilidades. En teoría, un ataque de fuerza bruta siempre encontrará la solución y no puede prevenirse (más adelante veremos que sí puede prevenirse), solamente es un asunto del tiempo que tomará encontrar la solución.

Cuanto mayor sea el espacio de probabilidades de solución, mayor será el tiempo que se requiere para encontrar la solución. Al grado que con los actuales algoritmos de digestión criptográfica con espacios de probabilidad muy altos (como SHA-256 o SHA-512) los tiempos necesarios para encontrar las cadenas de entrada a una salida propuesta, son virtualmente imposibles para cadenas de entrada relativamente grandes (cadenas mayores a 20 caracteres), tomaría trillones de trillones de años para encontrar por fuerza bruta usando todo el poder de cómputo disponible en el mundo.

Sin embargo, esta teoría es vencida fácilmente como veremos: Estos algoritmos de digestión (SHA-1, SHA-2, etc.) son muy eficientes y pueden ejecutarse a velocidades extremadamente altas. Una computadora apropiada para esta tarea puede ejecutar unos pocos millones de digestiones por segundo. Esta característica representa una debilidad en nuestro caso, ya que montar un ataque para ejecutar muchas digestiones es posible en tiempos relativamente cortos. Si consideramos que un lenguaje como el español tiene un vocabulario de entre 150 mil y 200 mil vocablos, podemos observar que un ***"ataque de diccionario"*** es relativamente fácil de montar. Una computadora dedicada con este propósito puede extraer en cuestión de horas o pocos días, todo tipo de combinaciones básicas de palabras, números, palabras con ciertos caracteres intercambiados (como usar el *cero* en lugar de la *letra "O"*), aunque estas combinaciones representen decenas o cientos de millones, se puede lograr en tiempos relativamente cortos.

Una variante de este ataque es el ***"ataque con tablas preparadas"***, en el que se tienen varios terabytes de información previamente procesada con los algoritmos más utilizados (SHA-1, SHA-2, etc.) y en estos casos, lo único que se tiene que hacer son búsquedas en las tablas de los valores en la información obtenida. Otro tipo de tablas son las ***"tablas rainbow"***, que ayudan al proceso de calcular palabras combinadas y variantes, con mayor facilidad.

En un ataque fuera de línea sobre bases de datos vulneradas suele reportarse que han sido extraídas entre el 50% y el 80% de las contraseñas en cuestión de días cuando se usan digestiones criptográficas aparentemente seguras. Solamente quedan exentas de estos ataques las contraseñas que son realmente robustas; como contraseñas muy largas, o combinaciones de caracteres que no son deducibles a partir de combinaciones usuales.

Prevención de ataques fuera de línea

Lo único que previene los ataques mencionados anteriormente es el uso de ***"Sal"*** al guardar las contraseñas. Se le llama sal a una combinación de bytes que se agregan a cada contraseña con el fin de incrementar su espacio de entropía. Si dos usuarios seleccionan la misma contraseña por casualidad, una digestión simple resultará en la misma cadena guardada en ambos registros, haciendo más simple el esfuerzo de atacar la base. Pero si a cada registro se le agrega una sal diferente, los dos registros que guardan la misma contraseña tendrán cadenas diferentes en su contenido. Además, más importante aún, al agregar la sal, es imposible comparar esa base de datos con tablas previamente computadas o un ataque de diccionario, gracias al espacio de entropía introducido con la sal.

La sal que se agrega es una cadena de bytes aleatorios y diferentes para cada registro, siguiendo las siguientes recomendaciones:

1. Generar una sal única para cada registro de contraseña, no solo para cada usuario o para cada sistema
2. Obtener cadenas de bytes aleatorios robustos criptográficamente
3. Dependiendo de la función protectora subyacente, utilizar la longitud máxima (por ejemplo: 256 bits para SHA-256, y 512 bits para SHA-512)

Agregar sal a los registros agrega un espacio muy amplio de entropía al contenido, lo cual hace que encontrar contraseñas se hace inviable para ataques sobre la base de datos. Solamente se pueden limitar a ataques de diccionario sobre cuentas individuales. Lo cual en nuestro caso es también un alto riesgo si el atacante es capaz de seleccionar cuentas de clientes en las que puede hacer un daño mayor, o un beneficio económico que lo justifique. Para ello le agregaremos otra característica a la sal; será cifrada, lo que hace imposible el ataque sobre los registros.

La forma en la que se agrega la sal a la contraseña es también muy importante para lograr el beneficio mayor. Existen algoritmos específicamente diseñados para hacer esta "mezcla" conocidos como *HMAC (Hash Message Authentication Code – por sus siglas en inglés)* de acuerdo con la especificación (Krawczyk, y otros, 1997). Estos algoritmos se basan en un proceso de digestión subyacente, pero establecen la forma de combinar los valores del contenido a proteger y el contenido de una llave criptográfica de manera que no sea reversible.

También es necesario provocar un impacto considerable en la velocidad en que se pueden procesar las digestiones de manera que no sea posible hacerlas tan rápido. Hay algoritmos específicamente diseñados con estas características en mente conocido como ***algoritmos para derivar llaves basadas en contraseñas***. Uno de los algoritmos que estaremos analizando en este documento es *PBKDF2*, el cual reúne las condiciones que hemos mencionado:

1. Integra la incorporación de una sal robusta al contenido
2. Aplica un algoritmo basado en HMAC para asegurar la correcta incorporación de la sal
3. Se asegura que el tiempo de ejecución se vea retrasado consistentemente y no puede ser optimizado ni procesado en paralelo.

Finalmente, la integración de una llave criptográfica para cifrar el contenido hace que cualquiera de los ataques sea imposible. No solo lo hace inviable, sino que mediante la aplicación de una llave que no esté disponible para el sistema (usando un dispositivo de alta seguridad HSM), no es posible concretar los ataques ya que la sal resulta desconocida, y eso hace que, en el espacio tan amplio de posibilidades, sea posible que algún valor de la sal haga posible cualquier valor de la contraseña. Esto significa que no hay manera de que un atacante pueda sacar conclusiones del contenido de las contraseñas originales.

Algoritmos y Protocolos

Los algoritmos y protocolos descritos a continuación serán necesarios para la completa implementación de la solución recomendada. Esta sección desarrolla una especificación para estos elementos junto con el razonamiento y justificación para cada decisión tomada en la recomendación.

Algoritmo para obtener la Llave Derivada (DK)

Para el procesamiento de contraseñas propuesto, la recomendación es hacer una llave derivada a partir de la contraseña utilizando el algoritmo ***PBKDF2*** descrito en el capítulo 5.2 de la especificación (Moriarty, y otros, 2017). La idea es procesar la contraseña con este algoritmo para obtener una llave derivada de 512 bits o 64 bytes. Dicha llave derivada será almacenada en la base de datos y utilizada para verificaciones posteriores, junto con otras piezas de información necesarias, descritas más adelante.

La selección de algoritmo para obtener la *Llave derivada* es *PBKDF2* ya que se trata del algoritmo recomendado para aplicación formal en entornos empresariales y gubernamentales (Steven, y otros, 2018), por tratarse de un algoritmo que se originó de un proceso formal, inicialmente en RSA en su primera versión, como parte de la suite de algoritmos para la infra-estructura de llaves públicas, conocido por su definición original como PKCS#5 (Public Key Cryptography Standards #5) descrito en la especificación (RSA Laboratories, 2012).

Si bien hay otros algoritmos conocidos (como argon2, bcrypt, scrypt, etc.), éstos no cuentan con la formalidad de pruebas y maduración que se requiere para dar confianza a empresas y gobierno y además no ofrecen ninguna ventaja clara ni nivel de seguridad adicional.

El algoritmo propuesto es una versión de PBKDF2 con ajustes que lo simplifican sin alterarlo, para su aplicación particular en el uso descrito en este documento. Dichos ajustes de simplificación son:

1. **`dkLen = hLen`**: En la definición del algoritmo, es necesario especificar el *tamaño de salida de la llave derivada (dkLen)*. Esto es necesario ya que este mismo algoritmo puede ser aplicado para derivar una llave muy larga para su uso en cifrados de flujo continuo, y esto se logra concatenando la salida de múltiples ciclos de ejecución de la *función pseudo-random subyacente (PRF)*. Puesto que éste no es el caso para esta aplicación, la salida de un solo bloque de ejecución es suficiente para obtener una llave derivada. *hLen* representa el tamaño de salida de un solo bloque de ejecución de la función generadora subyacente, por lo que, en nuestro caso, estableceremos el tamaño de salida de la llave derivada igual al tamaño de salida de un solo bloque de la *PRF*.
2. **`PRF: HmacSHA512`**: La *función pseudo-random subyacente* elegida entre las recomendadas en la especificación será *HmacSHA512*. El objetivo de aplicar este algoritmo para proteger las contraseñas es proporcionar la mayor seguridad disponible, por lo que, al aplicar esta función estaremos seleccionando la que proporciona el mayor campo de probabilidades para la llave derivada. En el caso de un ataque de fuerza bruta, es la que proporciona la menor probabilidad de éxito. Como consecuencia de esta selección, el tamaño de salida de la llave derivada (*dkLen*) será siempre **512 bits o 64 bytes**.
3. (en el paso 2) **`l=1, r=0`**: Como consecuencia de lo anterior, el *número de bloques (l)* de tamaño *hLen* en la llave derivada de salida será siempre *uno*, y el *residuo en el último bloque (r)*, será siempre *cero*.
4. (en el paso 3) **una sola iteración, sin contador**: Como consecuencia de lo anterior, y cómo ya se describió, solamente será necesaria *una* iteración de la *PRF*. Como la única diferencia entre las iteraciones es el contador que se agrega al final (4 bytes), en este caso no será necesario agregar el contador a la entrada de la *PRF*.
5. (en el paso 4) **no concatenar cadenas**: Como consecuencia de lo anterior, al calcularse solamente una cadena de salida de la *PRF*, no será necesario hacer una concatenación de bloques, y la salida final será únicamente la salida íntegra de una sola ejecución de la *PRF*.

Este algoritmo define los siguientes parámetros, entradas y salidas:

```
DK = PBKDF2 ( P, S, c, dkLen )
    Opción:    PRF      función pseudo-random subyacente (HmacSHA512)
    Entradas:  P        password (contraseña)
               S        sal
               c        contador
               dkLen    logitud de la llave derivada (512 bits)
    Salida:    DK       llave derivada
```

Como ya se mencionó en la adaptación del algoritmo, la opción de *PRF* será *HmacSHA512*.

La primera de las entradas al algoritmo es la *contraseña (P)*. Está será una cadena de caracteres sin límite de longitud ni de tipo de datos contenidos. Será necesario seleccionar una forma de codificación estándar para la aplicación (como UTF-8), ya que es necesario separarla por bytes para su procesamiento interno. Únicamente es recomendable que se apliquen reglas para contraseñas robustas de acuerdo con las políticas del banco y a las recomendaciones mencionadas en: *Recomendaciones para prevenir ataques en línea*.

La segunda de las entradas es la *Sal (S)*. Este es el elemento descrito en el razonamiento para proteger las contraseñas en: *Prevención de ataques fuera de línea*. Se trata de un número aleatorio único para cada contraseña. Es importante que no se reutilice esta sal ni se repita entre contraseñas. En este documento se describe el proceso de cifrado de este componente para su protección, sin embargo, para usarlo en el algoritmo de derivación de llave, es necesario alimentarlo en su forma clara, es decir, *descifrado*. Es importante guardar este dato junto a cada contraseña en su forma protegida, es decir, *cifrado*, para ser utilizado cada vez que se hace la verificación de la contraseña. Por lo tanto, también será necesario guardar junto a este dato el *identificador de la llave de cifrado para el HSM*. Ver la discusión acerca de la sal en: *Algoritmos para generar y manipular la Sal (S)*.

El tercer elemento de entrada es el *Contador (c)* que definirá la cantidad de veces que se ejecutará la función PRF. Como se ha mencionado, este contador debe ser lo más grande posible sin comprometer el tiempo de respuesta del sistema al usuario. Es importante guardar este dato junto a cada contraseña para ser utilizado cada vez que se hace la verificación de la contraseña. Ver la discusión acerca del contador en: *Valor del Contador (c)*.

El cuarto elemento de entrada al algoritmo es el *tamaño de la Llave Derivada (dkLen)*. Sin embargo, como ya se discutió en las condiciones de simplificación, este será siempre un valor constante de 512 bits o 64 bytes, por lo que no será necesario proporcionarlo ni guardarlo en la base de datos.

La salida de aplicar este algoritmo es la *Llave Derivada (DK)* y será siempre una cadena de 512 bits o 64 bytes. Estos serán representados en un **`java.lang.String`** de 128 caracteres hexadecimales para su almacenamiento y verificación.

Más adelante en el documento se muestra un ejemplo de implementación del algoritmo aquí descrito, con las adaptaciones mencionadas. Ver: *Algoritmo adaptado PBKDF2.*

Valor del Contador (c)

En las pruebas realizadas para hacer esta recomendación se observó que, en una computadora personal simple (Procesador Intel Core i7 a 2GHz, RAM 8 GB a 1600 MHz DDR3), se pudo ejecutar el algoritmo completo con el contador establecido en 100,000 (cien mil ciclos) en un tiempo promedio cercano a 500 milisegundos. Es un tiempo muy recomendable, ya que permite un tiempo de respuesta al usuario apropiado, mientras que provoca un tiempo de procesamiento inviable para un ataque de fuerza bruta.

Es importante recordar que este parámetro debe establecerse lo mayor posible para hacer “lenta” la ejecución en caso de ataque (no tiene nada que ver con incrementar la seguridad de la llave derivada). Se deben hacer pruebas en el servidor que se utilizará finalmente en producción para establecer el parámetro, **considerando cargas de trabajo y concurrencia**. Este elemento será representado en un **`java.lang.Integer`** para su almacenamiento y posterior aplicación.

Algoritmos para generar y manipular la Sal (S)

El algoritmo descrito en el punto anterior establece la necesidad de un elemento de *Sal (S)*. En este contexto, la sal se refiere a un elemento que se incluye en el procedimiento que hace que la cadena original a procesar (la contraseña) sea alterada para darle características únicas. La sal recomendada será una cadena aleatoria de bytes del mismo tamaño que el bloque generado por la *función pseudo-random subyacente*, es decir, ***una cadena aleatoria de 512 bits o 64 bytes***. Cualquier longitud mayor a esta no agrega absolutamente nada de seguridad puesto que el mecanismo de digestión siempre reducirá el campo de probabilidades a éste mismo tamaño.

Esta *Sal* se requiere durante la ejecución del algoritmo de derivación de la llave siempre que se ejecute. Esto puede ser al generar por primera vez la *Llave Derivada* para ser almacenada, o bien cada vez que se tenga que calcular para su verificación. Toda manipulación de la sal ocurrirá siempre en el interior del *Módulo de Seguridad*, y cuando la sal debe salir del módulo, únicamente lo podrá hacer en su forma cifrada. Esto quiere decir que cuando el Módulo de Seguridad le entrega la sal a la aplicación para su almacenamiento, previamente realizará una actividad de cifrado. Y cuando la aplicación entrega la sal al módulo para ser utilizada, el módulo deberá descifrarla primero para poder utilizarla en los algoritmos.

Algoritmo de generación de la Sal

Esta sal, deberá ser generada bajo las condiciones de una **función generadora de números aleatorios robustos con fines criptográficos** consistente con la recomendación (Barker, y otros, 2015). Estas cadenas de bits tienen las siguientes características:

1. Cada bit tiene una probabilidad de 50% de ser adivinado.
2. Cada valor de la sal tiene la misma probabilidad de ser seleccionado dentro de la población total de posibilidades, en este sentido es impredecible y no es influenciable por ninguna entrada al algoritmo de generación.

Para lograr la generación de la sal con estas condiciones se recomienda hacer uso del servicio provisto por el dispositivo de alta seguridad HSM.

Algoritmo de cifrado de la Sal

Una vez que la sal ha sido generada, y al entregarla a la aplicación se procederá a su cifrado para su conservación en la base de datos. El *Módulo de Seguridad* se encargará de generar la sal y proporcionarla a la aplicación únicamente en su forma cifrada, de manera que este elemento en su forma natural nunca debe abandonar al *Módulo de Seguridad* como se ilustra en *Arquitectura de la Solución*. Esta salida será representada en un **`java.lang.String`** de 128 caracteres hexadecimales para su almacenamiento y posterior aplicación.

Para cifrar la sal, también es recomendable (y en conformidad con la regulación aplicable) que se realice el cifrado usando el servicio provisto por el dispositivo de alta seguridad HSM.

El algoritmo recomendado para hacer el cifrado de la sal es AES-256 (Advanced Encryption System) como está definido en la especificación (Federal Information Processing Standards Publication 197, 2001). El modo de operación sugerido es ECB (Electronic Code Book), y sin utilizar modo de relleno: Es decir, configurar el cifrado de la siguiente forma; **`AES-256/ECB/NoPadding`**.

Este algoritmo y modo de cifrado es recomendado en consecuencia del siguiente razonamiento:

1. El cifrado *AES* es considerado en este momento como el más adecuado para el cifrado de información sensible para el uso empresarial y de gobierno, es también reconocido por su alto nivel de seguridad y no hay reportes de debilidades ni vulnerabilidades encontradas hasta la fecha.
2. El tamaño de la llave es 256 bits. Se trata del tamaño de llave mayor que define el algoritmo y la de mayor seguridad teórica. No representa un costo de procesamiento adicional, por lo que es conveniente aprovechar esta medida al máximo.
3. El modo de operación es ECB que, si bien se trata del modo de operación más simple disponible, todos los demás modos de operación se recomiendan solamente para cifrado de contenidos de tamaño mayor o cuando existe una relación entre los bloques o repetición de bloques (Dworkin, 2001). En nuestro caso solamente estaremos cifrando 64 bytes de la sal que es el equivalente a 4 bloques de cifrado y no hay relación ni repetición alguna entre bloques por tratarse de cadenas aleatorias de bits como se especifico anteriormente. Además, éste es el único modo de operación que no causa una variación en el tamaño del criptograma de salida, lo cual es necesario para su manejo en la base de datos.
4. No es necesario hacer relleno (No-Padding), esto es consecuencia del tamaño de la sal, ya que por definición siempre será de 64 bytes, y por lo tanto un múltiplo exacto de 16, que es el tamaño de bloque del algoritmo

de cifrado. De esta manera, tanto la entrada como la salida del algoritmo siempre serán de este tamaño sin alteración por relleno innecesario.

Este proceso de cifrado y descifrado de la sal hace que la llave derivada y la sal almacenadas en la base de datos sean totalmente inutilizables para un ataque sobre los datos. Si se llegan a comprometer los datos almacenados, es **imposible** realizar un ataque para obtener las contraseñas (ni siquiera mediante fuerza bruta).

Llave de cifrado de la Sal

La medida de seguridad principal proporcionada por el dispositivo de alta seguridad HSM es la generación y conservación de la *Llave de cifrado de la Sal*. Como se discutió anteriormente, todo el mecanismo de protección de las contraseñas se hace imposible de descifrar (aun mediante fuerza bruta), solamente cuando el cifrado de la sal puede mantenerse confidencial. Esto se logra manteniendo la llave de cifrado de la sal en el interior del dispositivo de alta seguridad HSM, el cual, nos da las garantías de seguridad necesarias.

El dispositivo de alta seguridad proporciona el servicio de generar una llave de cifrado tal, que una vez generada, nunca será expuesta de ninguna manera, ni abandonará al dispositivo. Además, el dispositivo no puede ser violentado para llevar a cabo la extracción, aun sí el dispositivo fuera hurtado y se pudiera manipular con tiempo y recursos externos. Está diseñado para destruirse en caso de ser violado.

Cuando la llave de cifrado ha sido generada, solamente puede ser utilizada en el interior del dispositivo. Es decir, todas las actividades de cifrado y descifrado de la sal descritas anteriormente, se realizarán en el interior del dispositivo de alta seguridad HSM. Para hacerlo, es necesario proporcionar al dispositivo cada vez que se requiere, el contenido a cifrar o descifrar, acompañado de un identificador que se le asigna a la llave de cifrado para que el dispositivo lleve a cabo la tarea.

Al generar la llave de cifrado se le asigna un identificador único a la llave para ser referenciada posteriormente. Este identificador deberá ser guardado en el sistema junto a la *Sal (cifrada)* para que siempre que se vaya a utilizar la sal en el módulo, se pueda descifrar de forma correcta. Este elemento se representará en un **`java.lang.String`** de tamaño y formato definido por la interfaz de programación *(API)* del dispositivo de alta seguridad HSM.

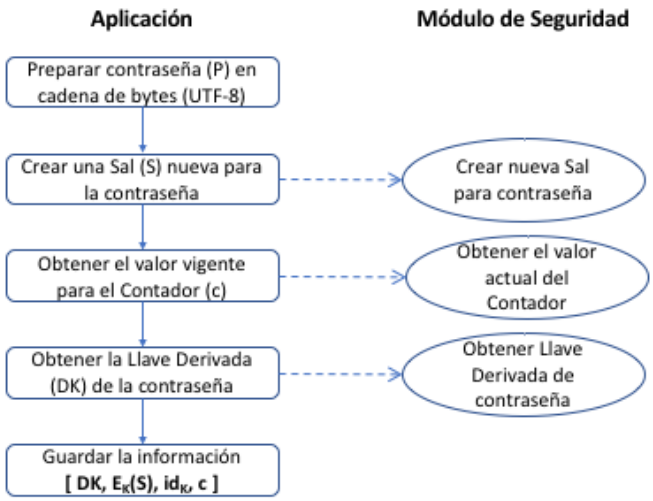
Es recomendable hacer una rotación de llaves de cifrado con cierta periodicidad (cada año o cada dos años, es una buena medida). Esto significa que después de un tiempo, se generará una nueva llave de cifrado que se agregará al sistema (sin eliminar las anteriores). Cuando esto ocurra, el Módulo de Seguridad debe saber de la existencia de una nueva llave y a partir de ese momento cifrar las nuevas instancias de Sal que se generen con la nueva llave. Lógicamente, el nuevo identificador será guardado junto a las mencionadas nuevas instancias de sal, de manera que, al procesar cada instancia de sal, siempre se sabe con cuál llave de cifrado fue protegida la instancia en cuestión.

Protocolo para generar un registro nuevo de contraseña

Cada vez que se va a dar de alta un nuevo registro de usuario con su contraseña en el sistema (o cada vez que se realice un cambio de contraseña por el usuario), será necesario realizar los siguientes pasos:

1. Preparar la *contraseña (P)* proporcionada por el usuario como una cadena de caracteres en bytes individuales *(UTF-8)*
2. Crear una *Sal (S)* nueva para la contraseña, y recibir la *Sal (cifrada)* y el *id de la Llave de Cifrado de la Sal*
3. Obtener el valor vigente para el *Contador (c)*
4. Obtener la *Llave Derivada (DK)* de la *contraseña (P)*, con los datos anteriores: **`DK = PBKDF2(P, S, c)`**
5. Guardar en la base de datos la información obtenida: **`[ DK, `$E_K(S)$`, `id_K`, c ]`**

El siguiente diagrama muestra la secuencia de este protocolo:



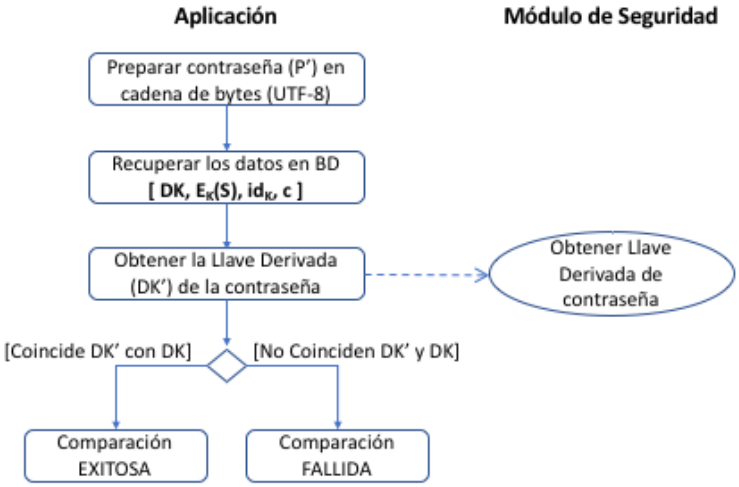
Secuencia para el Protocolo para generar un registro nuevo de contraseña

Protocolo para verificar una contraseña

Cada vez que sea necesario hacer una verificación de contraseña para validar la autenticidad de un usuario, éste debe proporcionar su contraseña. Con esta entrada y con los datos almacenados previamente en la base de datos, se realizarán los siguientes pasos:

1. Preparar la *contraseña (P')* recién proporcionada por el usuario como una cadena de caracteres en bytes individuales *(UTF-8)*
2. Recuperar los datos guardados en la base de datos: **[DK, E_K(S), id_K, c]**
3. Con los datos recuperados (excepto **DK**), y con la contraseña del usuario; Obtener la Llave Derivada (DK') de la contraseña recién proporcionada (P'): **DK' = PBKDF2(P', S, c)**
4. Comparar el valor obtenido **DK'**, con el valor previamente recuperado de la base de datos **DK**. Si los valores coinciden, la autenticación del usuario ha sido exitosa.

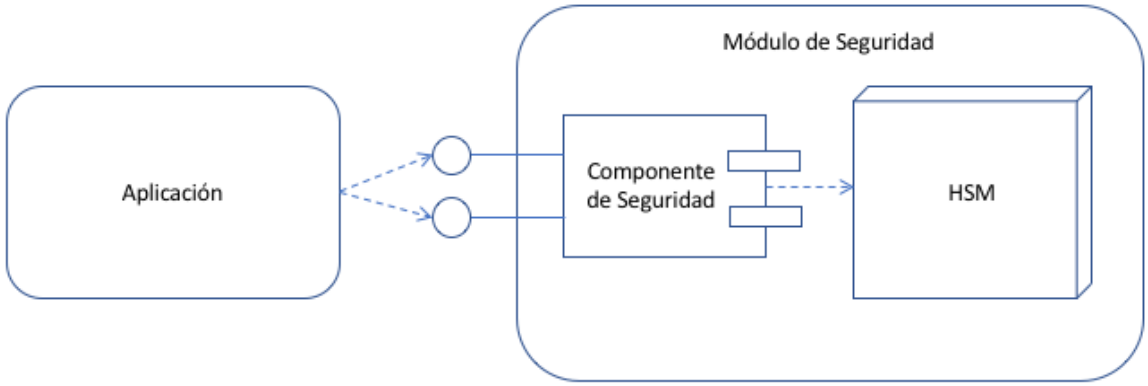
El siguiente diagrama muestra la secuencia de este protocolo:



Secuencia para el Protocolo para verificar una contraseña

Arquitectura de la Solución

Como lo ilustra el diagrama a continuación, podemos ver que la solución a implementar consiste en un *Módulo de Seguridad* que es la combinación de un *Componente de Seguridad* que proporcione la implementación de los algoritmos mencionados (a desarrollar), y un dispositivo de alta seguridad *HSM* el cual proporciona la protección de las llaves criptográficas mediante mecanismos de hardware y realiza la ejecución de las operaciones relacionadas con la criptografía necesaria.



Arquitectura general de alto nivel de la solución a implementar

Nota: Toda la actividad realizada en el Módulo de Seguridad deberá hacerse exclusivamente en memoria, y ser limpiada al terminar. El Módulo de Seguridad no debe persistir ningún dato que sea utilizado en el proceso.

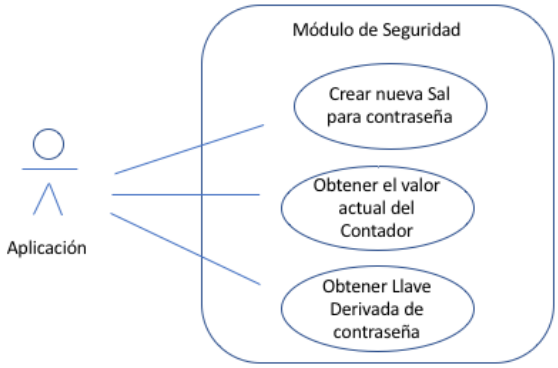
Funcionalidad del Módulo de Seguridad

El *Módulo de Seguridad* proporcionará servicios a la aplicación de manera que ésta se beneficie de la implementación descrita y elaborada en *Algoritmos y Protocolos*. Esto lo realizará mediante dos puntos de entrega (*endpoints*); el primero, para las operaciones cotidianas de generar nuevos registros y verificar contraseñas, y el segundo, de manera más eventual, para hacer ajustes a los parámetros de ejecución de los algoritmos.

El primer punto de entrega debe proporcionar la funcionalidad necesaria para:

1. El primer protocolo, cada vez que se va a crear un registro nuevo; es necesario que el *Módulo de Seguridad* proporcione a la aplicación una nueva *Sal (cifrada)*, el valor actual del *Contador (c)*, para que a continuación se obtenga la *Llave Derivada (DK)* a conservar en el registro.
2. El segundo protocolo es más simple, al verificar una contraseña; el sistema tomará los datos almacenados y la contraseña provista por el usuario para pedir al *Módulo de Seguridad* que obtenga la *Llave Derivada (DK')* y pueda ser comparada con la que se tiene almacenada.

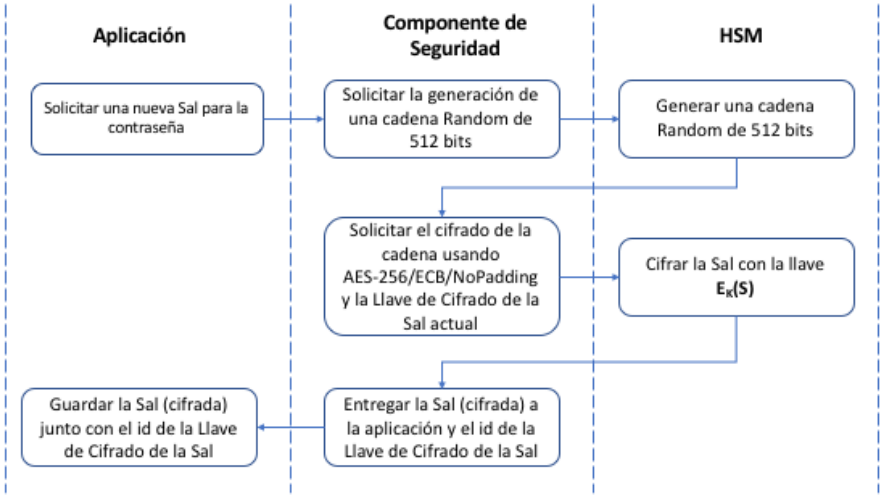
Esto se puede lograr si los servicios del Módulo de Seguridad permiten los casos de uso mostrados en el diagrama a continuación:



Casos de uso del Componente de Seguridad

Crear nueva Sal para contraseña

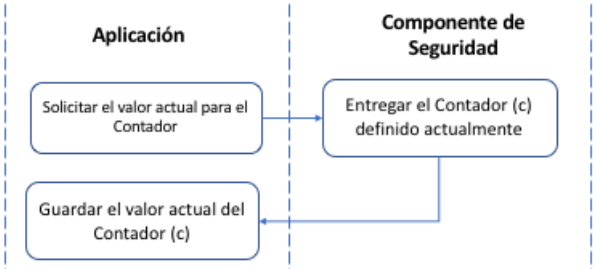
La creación de una nueva Sal para cada contraseña a registrar en el sistema; Como ya se ha descrito, asegurando que la entrega a la aplicación se haga únicamente en la forma cifrada de la Sal, como se muestra en el siguiente diagrama:



Secuencia para la Creación de una nueva Sal para la contraseña

Obtener el valor actual del Contador

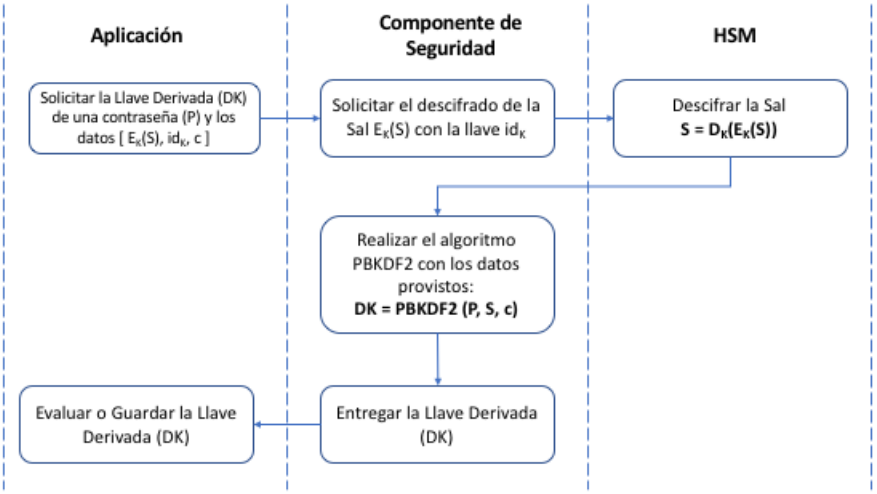
Cada registro creado debe guardar el valor del Contador usado para el cálculo. El Módulo de Seguridad se debe medir constantemente y actualizar para que este valor sea el más indicado. Ver discusión en *Valor del Contador (c)*.



Secuencia para Obtener el valor actual del Contador

Obtener Llave Derivada de contraseña

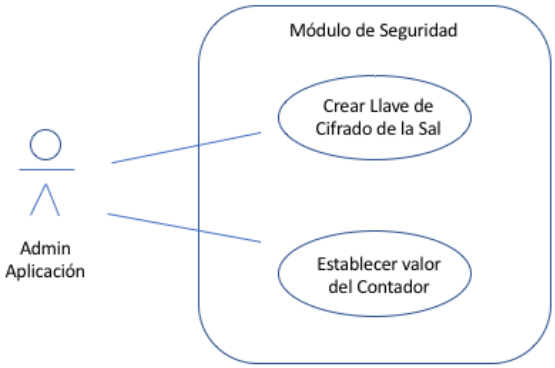
Es necesario obtener la Llave Derivada de la contraseña todas las veces que se va a realizar una verificación de usuario, y por supuesto, al crear el registro de la contraseña en el sistema. En esta operación se realiza el cálculo usando el algoritmo descrito en *Algoritmo para obtener la Llave Derivada (DK)*, como se muestra en el siguiente diagrama:



Secuencia para Obtener Llave Derivada de contraseña

El segundo de los puntos de entrega proporcionará la funcionalidad necesaria para hacer cambios a los parámetros de funcionamiento de los algoritmos, concretamente;

1. La creación de la Llave de Cifrado de la Sal, que se ejecutará de acuerdo con la definición de la institución para la rotación de llaves (la recomendación es hacerlo cada dos años). Ver: *Llave de cifrado de la Sal*.
2. Ajustes al Contador que provoca la "lentitud" en el algoritmo de derivación de la llave; recordando que debe ser el número máximo posible sin comprometer los tiempos de respuesta promedio del sistema para verificación de contraseña. El sistema cliente deberá obtener el valor del Contador establecido en el Módulo de Seguridad para usarlo en cada registro nuevo que se vaya a crear. Sin embargo, al realizar la verificación, siempre deberá hacerse con el valor del Contador que se estableció al crear el registro, de lo contrario la verificación no va a ser realizada de manera correcta.



Casos de uso administrativos del Componente de Seguridad

Implementación

Con el fin de asegurar varios elementos de la presente recomendación, se desarrolló el código en *Java* de demostración de los elementos discutidos en el presente documento.

El código es totalmente funcional y comprobado, y puede usarse como modelo de entrada en el proyecto de protección de contraseñas. Sin embargo, hay que tener consideración por algunos elementos; por ejemplo, se indican algunas partes que deberían ejecutarse en el HSM en lugar del código provisto. En estos casos, la funcionalidad debe quedar exactamente igual, pero la implementación cambia para hacer uso de los servicios del dispositivo.

La autoridad para determinar la correcta implementación es el contenido del presente documento; principalmente en la sección de Algoritmos y Protocolos y de Arquitectura de la Solución. Este ejemplo solamente debe tomarse como un complemento de ayuda para su implementación.

Algoritmo adaptado PBKDF2

```
package org.jrrevuelta.passwords;

import java.math.BigInteger;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Date;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

public class PBKDF2 {

    private static final String password = “abc123”;
    private static final int count = 100000;
    private static final int hLen = 512;    // dkLen = hLen
    private static final String prf = “HmacSHA512”;
    private static byte[] salt;
    private static byte[] derivedKey;

    public static void main(String[] args) {

        Date t1 = new Date();
        try {
            System.out.println(“\n\tImplementación PBKDF2:”);
            System.out.println(“Contraseña: “ + password);
            System.out.println(“Contador: ” + count);

            // Generar Sal, secuencia random de 512 bits
            SecureRandom random = new SecureRandom();  // Hacer esto en el HSM
            salt = new byte[hLen / 8];
            random.nextBytes(salt);
            System.out.println(“Sal: ” + new BigInteger(1, salt).toString(16).toUpperCase();

            // Preparar PRF subyacente (función pseudo-random)
            // …sembrarla con ‘password’ como la llave para todas las iteraciones
            Mac hmac = Mac.getInstance(prf);
            Hmac.init(new SecretKeySpec(password.getBytes(), “RAW”));

            // Inicializar el vector ‘Derived Key’ y el primer bloque de ‘macText’ con ‘Salt’
            derivedKey = new byte[hLen / 8];
            for (int i=0; i<derivedKey.length; i++)
                derivedKey[i] = 0x00;
            byte[] macText = salt;   // No es necesario agregar el contador (i) porque l=0

            // Ejecutar PRF ‘count’ veces, comenzar con ‘macText = salt’
            for (int i=0; i<count; i++) {
                byte[] u = hmac.doFinal(macText);
                for (int j=0; j<derivedKey.length; j++)
                    derivedKey[j] ^= u[j];
                macText = u;
```

```
            }

            // Reportar la Llave derivada resultante
            System.out.println("Llave Derivada (DK): " +
                               new BigInteger(1, derivedKey).toString(16).toUpperCase());

        } catch (NoSuchAlgorithmException e) {
            System.err.println("Algoritmo no soportado: " + e.getMessage());
        } catch (InvalidKeyException e) {
            System.err.println("Password no puede ser usado en Hmac. " + e.getMessage());
        }

        // Reportar el tiempo de ejecución
        Date t2 = new Date();
        System.out.println("\nTiempo: " + (t2.getTime() – t1.getTime()) + " milisegundos.");
    }
}
```

Código fuente en Java de ejemplo con la implementación del algoritmo PBKDF2 simplificado propuesto

Referencias

Barker, Elaine and Kelsey, John. 2015. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* s.l. : National Institute of Standards and Technology, U.S. Department of Commerce, 2015. NIST SP 800-90A Rev. 1.

Comisión Nacional Bancaria y de Valores, Secretaría de Hacienda y Crédito Público. 2005-2018. *Disposiciones de Carácter General Aplicables a las Instituciones de Crédito.* México, CDMX : Diario Oficial de la Federación, 2 de diciembre de 2005, modificadas hasta 27 de Noviembre de 2018, 2005-2018.

Ducklin, Paul. 2013. Serious Security: How to store your users' passwords safely. *naked security by Sophos.* [Online] Nov 20, 2013. https://nakedsecurity.sophos.com/2013/11/20/serious-security-how-to-store-your-users-passwords-safely/.

Dworkin, Morris. 2001. *Recommendation for Block Cipher Modes of Operation.* s.l. : National Institute of Standards and Technology, U.S. Department of Commerce, 2001. NIST SP 800-38A.

Esslinger, Bernhard. 2018. *Learning and Experiencing Cryptography with CrypTool and SageMath.* s.l. : CrypTool Project (www.cryptool.org), 2018.

Federal Information Processing Standards Publication 197. 2001. *Specification for the ADVANCED ENCRYPTION STANDARD (AES).* s.l. : National Institute of Standards and Technology, U.S. Department of Commerce, 2001. FIPS PUB 197.

Hornby, Tailor. 2018. Salted Password Hashing - Doing it Right. *CrackStation.* [Online] Julio 30, 2018. https://crackstation.net/hashing-security.htm.

Krawczyk, Hugo, Bellare, Mihir and Canetti, Ran. 1997. *HMAC: Keyed-Hashing for Message Authentication.* s.l. : Internet Engineering Task Force (IETF), 1997. RFC 2104.

Moriarty, Kathleen M., Kaliski, Burt and Rusch, Andreas. 2017. *PKCS #5: Password-Based Cryptography Specification Version 2.1.* s.l. : Internet Engineering Task Force (IETF): RFC 8018, 2017. ISSN 2070-1721.

RSA Laboratories. 2012. *PKCS#5: Password-Based Encryption Standard Version 2.1.* 2012.

Schneier, Bruce. 2015. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* Indianapolis, IN : John Wiley & Sons, 2015. ISBN 1-119-09672-3.

Steven, John, Manico, Jim and Righetto, Dominique. 2018. Password Storage Cheat Sheet. *OWASP Foundation.* [Online] Julio 19, 2018. https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet.