Algoritmo HOTP (An HMAC-Based One- Time Password Algorithm)

RFC 4226

Introducción

Publicado en el 2005.

Los password de un solo uso (OTP) son una de las formas más simples y populares de autenticación doble factor para asegurar el acceso a las redes.

HOTP se trata de un algoritmo simple para generar passwords de un solo uso, basado en HMAC (Hashed Message Authentication Code). Fue desarrollado por miembros del OATH (Open AuTHentication Initiative). Puede ser implementado por cualquier fabricante de Hardware o desarrollador de software para crear dispositivos de autenticación interoperables y agentes de software.

Requerimientos del Algoritmo

Hay algunos principios de diseño que el algoritmo debe seguir, principalmente en la usabilidad del usuario final y la capacidad para que el algoritmo sea implementado en Hardware de bajo costo.

Requerimiento 1 - el algoritmo debe ser secuencial o basado en contador. Uno de los objetivos es poder embeber el algoritmo HOTP in dispositivos como Java Smart cards, USB dongles, y GSM Sim cards.

Requerimiento 2: La implementación del algoritmo en hardware debería ser económica, minimizando requisitos de batería, cantidad de botones, poder computacional, y tamaño de la pantalla LCD.

Requerimiento 3: el algoritmo debe poder trabajar con tokens que no soportan ninguna entrada numérica, pero podría también ser usado por dispositivos más sofisticados como PIN-pads seguros.

Requerimiento 4: el valor mostrado en el token debe ser fácil de leer y fácil de ingresar por el usuario. Esto requiere que el valor HOTP tengo un tamaño razonable (ni muy pequeño ni muy grande). El valor HOTP debería tener al menos 6 dígitos. También es deseable que este valor sea sólo numérico, así sería más fácil de ingresar en dispositivos como celulares.

Requerimiento 5: deben existir mecanismos fáciles de usar para resincronizar el contador.

Requerimiento 6: El algoritmo debe usar una clave secreta fuerte., cuyo tamaño debe ser al menos 128 bits. La RFC recomienda que la longitud de la clave secreta sea de 160 bits.

Notaciones y Símbolos

Algunos símbolos utilizados por el algoritmo son:

C: es el factor de movimiento, es decir, el valor del contador (8 bytes). Este contador debe estar sincronizado entre el generador de HOTP (o sea, el cliente) y el validador de HOTP (o sea, el servidor).

K: es la clave secreta entre el cliente y el servidor. Cada generador de HOTP posee un K único, diferente al resto de los clientes.

T: parámetro de regulación: el servidor rechazará conexiones de un usuario después de “T” intentos de autenticación incorrectos.

s: parámetro de resincronización: el servidor intentará verificar un valor HOTP recibido dentro de “s” consecutivos valores del contador.

Digit: es el número de digitos de un valor HOTP, se trata de un parámetro del sistema.

Descripción General

El algoritmo HOTP utiliza dos cosas, por un lado, un contador cuyo valor se va incrementando y por el otro una clave simétrica estática conocida sólo por el token y el servicio de validación. Para crear un valor HOTP, se utiliza en la mayoría de los casos el algoritmo HMAC-SHA-1 (definido en la RFC 2104).

La salida de HMAC-SHA-1 es un valor de 160 bits, el cual debe ser truncado en un valor fácil de ingresar por el usuario final. La fórmula para obtener un valor HOTP es la siguiente:

HOTP(K,C) = Truncate( HMAC-SHA-1(K,C) )

Donde Truncate representa la función que se encarga de convertir un valor HMAC-SHA-1 en un valor HOTP.

La clave (K) y el contador (C) son hasheados en un high-order-byte-first.

Los valores HOTP obtenidos por el generador HOTP son tratados como big endian.

Generando valores HOTP

Se puede describir esta operación en 3 pasos:

Paso 1: generar un valor HMAC-SHA-1.

HS = HMAC-SHA-1( K,C )

Donde HS es un String de 20 bytes.

Paso 2: Generar un String de 4 bytes (a través de Truncación Dinámica – Dynamic Truncation - DT).

Sbits = DT (HS)

Donde DT retorna un String de 31 bits.

Paso 3: computar un valor HOTP.

Snum = StringToNum(Sbits)

Esto convierte a Sbits en un número cuyo rango está en 0…2^{31}-1

Return D = Snum mod 10^Digit

D es un número en el rango de 0…10^{Digit}-1

La función de truncamiento realiza el Paso 2 y el Paso 3, es decir, el truncamiento dinámico y la operación módulo para reducir Snum. El propósito de la técnica de truncamiento de offset (desplazamiento) dinámico es extraer un código binario dinámico de 4 bytes del resultado devuelto por HMAC-SHA-1 (el cual posee un tamaño de 160 bits, o sea 20 bytes).

DT(String) // String = String[0]...String[19]

Let OffsetBits be the low-order 4 bits of String[19]

Offset = StToNum(OffsetBits) // 0 <= OffSet <= 15

Let P = String[OffSet]...String[OffSet+3]

Return the Last 31 bits of P

La razón para enmascarar el bit más significativo de P es para evitar confusiones sobre módulos de computación con signo y sin signo. Diferentes procesadores realizan estas operaciones de forma diferente, y al enmascarar el bit con signo se elimina toda ambigüedad.

La implementación debe extraer por lo menos un código de 6 dígitos, y si es posible, 7 o 8 dígitos. Esto depende del nivel de seguridad que necesitemos.

<<<ejemplo sigue>>>>

Consideraciones de Seguridad

El ataque más posible que se podría realizar contra el algoritmo HOTP es el ataque de fuerza bruta.

Asumiendo que un adversario es capaz de observar numerosos intercambios de protocolo y recoger secuencias de valores de autenticación exitosos. Este adversario, si trata de construir una función F para generar valores HOTP basados en sus observaciones, non tendrá una ventaja significativa sobre un random guess.

La conclusión lógica es simplemente que la mejor estrategia es realizar un ataque de fuerza bruta para enumerar y probar todos los valores posibles.

La seguridad del algoritmo HOTP puede definirse en base a la siguiente fórmula:

Sec = sv/10^Digit

Donde:

* Sec es la probabilidad de éxito del atacante.
* s es el tamaño de ventana de sincronización look-ahead.
* v es la cantidad de intentos de verificación.
* Digit es el número de dígitos de los valores HOTP.

Podemos jugar con s, T (el parámetro de regulación que limita el número de intentos por un atacante) y Digit hasta lograr cierto nivel de seguridad, preservando aún la usabilidad del sistema.

Requerimientos de Seguridad

Cualquier algoritmo OTP es solamente tan seguro como la aplicación y los protocolos de autenticación que la implementan. Sin embargo, se puede analizar aisladamente la seguridad de estos algoritmos. Los parámetros T y s tienen un significativo impacto en la seguridad. Es importante remarcar que el algoritmo HOTP no es un sustituto de la encriptación y no provee privacidad de transmisión de datos. Se deben utilizar otros mecanismos para defenderse de ataques destinados a corromper la confidencialidad y la privacidad de las transacciones.

1. Requerimientos del protocolo de autenticación: a continuación, se nombran algunos requerimientos para un protocolo P que implementa HOTP como método de autenticación entre un proveedor y un verificador:
   1. RP1: P debe soportar la autenticación de dos factores, es decir, la comunicación y verificación de algo que usted conoce (código secreto, como un password, pin code, etc.) y algo que usted tiene (token). El código secreto es solamente conocido por el usuario y usualmente es ingresado con el valor OTP con el propósito de autenticarse.
   2. RP2: P no debería ser vulnerable a ataques de fuerza bruta. Esto implica que el servidor debe realizar ciertas validaciones y bloqueos.
   3. RP3: P debería ser implementado a través de un canal seguro para proteger la privacidad de los usuarios y evitar ataques de repetición (fuerza bruta).
2. Validación de los valores HOTP: el cliente HOTP (token de hardware o software) incrementa su contador y luego calcula el próximo valor HOTP. Si el valor recibido por el servidor de autenticación machea con el valor calculado por el cliente, el valor HOTP es validado. En este caso, el servidor incrementa el en uno el valor del contador.

Si el valor recibido por el servidor no machea con el valor calculado por el cliente, el servidor inicia el protocolo de resincronización (look-ahead window) antes de que el mismo solicite otro password.

Si la resincronización falla, el servidor solicita otro valor de autenticación, hasta que se alcanza la cantidad máxima de intentos autorizados.

Si la cantidad máxima de intentos autorizados se supera, el servidor debería bloquear la cuenta e iniciar un procedimiento para informarle al usuario.

1. Estrangulamiento en el servidor: truncar el valor HMAC-SHA-1 a un valor más corto puede dar pie a ataques de fuerza bruta. Sin embargo, el servidor de autenticación necesita detectar y parar estos ataques.

La RFC recomienda setear un parámetro de estrangulamiento T, el cual define el número máximo de posibles intentos para la validación de OTP. El servidor gestiona contadores individuales por cada dispositivo HOTP para tomar nota de cualquier intento fallido. La RFC recomienda que T no sea demasiado grande, particularmente si el método de resincronización usado en el servidor está basado en ventanas, y el tamaño de la venta es demasiado grande. T debería setearse tan bajo como sea posible, siempre garantizando que la usabilidad no se vea afectada.

Otra opción sería implementar un esquema de delay para evitar ataques de fuerza bruta. Luego de cada intento fallido, el servidor de autenticación esperaría por un tiempo determinado, con una distribución exponencial, es decir, si en el primer intento fallido esperé 5 segundos, en el segundo consecutivo esperaré 10.

1. Resincronización del contador: Aunque el valor del contador del servidor solo se incrementa después de una autenticación HOTP exitosa, el contador en el token se incrementa cada vez que el usuario solicita un nuevo HOTP. Debido a esto, los valores de contador en el servidor y en el token pueden estar fuera de sincronización.

La RFC recomienda setear un parámetro look-ahead “s” en el servidor, el cual define el tamaño de la ventana look-ahead. En pocas palabras, el servidor puede recalcular los valores del próximo valor HOTP y verificarlos contra el cliente HOTP recibido.

La sincronización de los contadores en este escenario simplemente requiere que el servidor calcule el proxímo valor HOTP y determine si hay un macheo. Opcionalmente, el sistema debe requerir que el usuario envíe una secuencia de valores HOTP (por ejemplo: 5 valores consecutivos) con el propósito de resincronizarse, ya que forjar una secuencia de valores de HOTP consecutivos es aún más difícil que adivinar un solo valor de HOTP.

El límite superior establecido por el parámetro s garantiza que el servidor no compruebe los valores de HOTP para siempre (lo que provoca un ataque de denegación de servicio) y también restringe el espacio de posibles soluciones para un atacante que intente fabricar valores de HOTP. s DEBE establecerse lo más bajo posible, al mismo tiempo que se asegura de que la usabilidad no se vea afectada.

1. Gestión de claves secretas: Las operaciones relacionadas con claves secretas utilizados para generar y verificar valores OTP deben realizarse de forma segura, a fin de mitigar los riesgos de fugas de información sensible.

Se pueden considerar dos diferentes caminos para generar y almacenar de forma segura claves secretas in los sistemas de validación:

* Generaciones determinísticas: las claves secretas se derivan de una semilla maestra, tanto en las etapas de aprovisionamiento como de verificación, y se generan sobre la marcha siempre que sea necesario.
* Generaciones aleatorias: las claves secretas son generados aleatoriamente en la etapa de aprovisionamiento y deben almacenarse inmediatamente y mantenerse seguros durante su ciclo de vida.

Generación Determinística

Una posible estrategia es derivar las claves secretas desde una clave maestra. La clave maestra debe almacenarse únicamente en el servidor. Se DEBE usar un dispositivo a prueba de manipulaciones para almacenar la clave maestra y obtener los secretos compartidos de la llave maestra y cierta información pública. El principal beneficio sería evitar la exposición de los secretos compartidos en cualquier momento y también evitar requisitos específicos de almacenamiento, ya que los secretos compartidos podrían generarse a pedido cuando sea necesario en el momento del aprovisionamiento y la validación.

Podemos distinguir dos casos diferentes:

* Una única claves maestra MK es usada para derivar las claves secretas; cada dispositivo HOTP posee una clave secreta diferente, K\_i = SHA-1 (MK, i), donde i representa información pública que identifica de forma única al dispositivo HOTP, como un número de serie, una identificación de token, etc. Obviamente, esto está en el contexto de una aplicación o servicio: diferentes proveedores de aplicaciones o servicios tendrán diferentes secretos y configuraciones.
* Varias claves maestras MK\_i son usadas y cada dispositivos HOTP almacena un conjunto de diferentes claves derivadas, {K\_i,j = SHA-1(MK\_i,j)}, donde j representa una porción de información pública que identifica a un dispositivo. La idea sería almacenar únicamente la calve maestra activa en el servidor de validación, en el Módulo de Seguridad de Hardware (HSM), y guardar en un lugar seguro, usando métodos de compartición de claves. En este caso, si una clave maestra MK\_i se vé comprometida, es posible cambiar a otra clave sin reemplazar todos los dispositivos.

El inconveniente en el caso determinista es que la exposición del secreto maestro obviamente permitiría a un atacante reconstruir cualquier secreto compartido basado en información pública correcta. Se requerirá la revocación de todos los secretos o el cambio a un nuevo conjunto de secretos en el caso de múltiples claves maestras.

Por otro lado, el dispositivo utilizado para almacenar las claves maestras y generar los secretos compartidos DEBE ser resistente a alteraciones. Además, el HSM no estará expuesto fuera del perímetro de seguridad del sistema de validación, lo que reduce el riesgo de fugas.

Generación Aleatoria

Las claves secretas son generadas aleatoriamente. Un generador aleatorio requiere una fuente de aleatoriedad natural. Prácticamente, hay dos posibles caminos a considerar para la generación de los secretos compartidos:

* Generadores basados en Hardware: explotan la aleatoriedad que ocurre en los fenómenos físicos. Una buena implementación puede basarse en osciladores e incorporarse de tal forma que los ataques activos sean más difíciles de realizar.}
* Generadores basados en Software: diseñar un buen generador aleatorio de software no es una tarea fácil. Una implementación simple, pero eficiente, debe basarse en varias fuentes y aplicar a la secuencia muestreada una función de un solo sentido, como SHA-1.

La RFC recomienda elegir productos probados.

También recomienda almacenar las claves compartidas de manera segura, encriptándolas y sólo exponiéndolas cuando sea necesario: por ejemplo, la clave secreta es desencriptada cuando necesita verificarse un valor HOTP, y vuelta a reencriptar inmediatamente para limitar la exposición en la RAM por un período corto de tiempo. Los datos almacenados que contienen la clave secreta debe estar en un área segura, para evitar tanto como sea posible los ataques directos en los sistemas de validación y las bases de datos que las contienen.

Particularmente, el acceso a las claves secretas se debería limitar a programas y procesos requeridos por el sistema de validación únicamente.

Claves Secretas Compuestas

Sería deseable incluir incluir factores de autenticación adicionales en la clave secreta K. Estos factores adicionales pueden consistir en cualquier dato conocido en el token pero no que no sea fácilmente obtenido por otros. Por ejemplo:

* PIN o Password obtenido por el usaurio en el token.
* Número de teléfono.
* Cualquier identificador único disponible programáticamente en el token.

En este escenario, la clave secreta compuesta K es construida durante el proceso de aprovisionamiento desde una semilla el valor de una semilla aleatorio combinada con otra o más factores de autenticación adicionales. El servidor podría generar a pedido o almacenar secretos compuestos, en cualquier caso, dependiendo de la elección de implementación, el token solo almacena el valor de inicialización. Cuando el token realiza el cálculo de HOTP, calcula K a partir del valor de inicialización y los valores de entrada o de entrada local de los otros factores de autenticación.

El uso de claves secretas compuestas pueden hacer más fuerte los sistemas de autenticación basados en HOTP a través de la inclusión factores de autenticación adicionales en el token.

En la medida en que el token es un dispositivo confiable, este enfoque tiene el beneficio adicional de no requerir la exposición de los factores de autenticación (como el PIN de entrada del usuario) a otros dispositivos.

Autenticación Bidireccional

el cliente HOTP también podría utilizarse para autenticar el servidor de validación, alegando que es una entidad genuina que conoce el secreto compartido.

Como el cliente HOTP y el servidor están sincronizados y comparten el mismo secreto (o un método para recalcularlo), se podría implementar un protocolo simple de 3 pasos:

1. El usuario final ingresa el TokenID y un primer valor OTP (OTP1);
2. El servidor chequea OTP1 y si es correcto, enviar devuelta OTP2;
3. El usuario final chequea OTP2 usando su dispositivo HOTP y si es correcto, utiliza el sitio web.

Obviamente, toda la comunicación OTP debe tener lugar sobre un canal seguro, por ejemplo: SSL/TLS, Ipsec connections.

Consideraciones de Seguridad

En nuestro escenario, el usuario y el servidor comparten una clave compartida K. Ambos mantienen un contador C, el cual se inicializa en cero, y el usuario se autentica enviando ALG( K,C ) al servidor, quien aceptará el valor si éste es correcto.

Para protegerse contra incrementos accidentales del contador del usuario, el servidor, una vez recibido un valor z, lo aceptará si está en un rango determinado, donde interviene el parámetro “s” que es el parámetro de resincronzación y el contador del servidor. Si acepta el valor recibido porque cae en el lugar i de la ventana, entonces incrementará el contador a i + 1. Si no, el valor del contador no cambiará.

Algunos posibles ataques podrían ser:

* Ataque de fuerza bruta.
* Ataque a SHA-1: una debilidad de este algoritmo son las colisiones. Una colisión en una función hash “h” significa que existe un par x, y de diferentes entradas tal que h(x) = h(y). Se podría realizar un birthday attack buscando una colisión en 2^{80} intentos. Sin embargo, los nuevos ataques sobre SHA-1 no tienen impacto en la seguridad de HMAC-SHA-1. Para empezar, HMAC no es una función hash. Se trata de un código de autenticación de mensajes (MAC) que usa una función hash internamente. Una MAC depende de una clave secreta, mientras que las funciones hash no. La preocupación en una MAC es la falsificación, y no las colisiones. HMAC fue diseñado para que las colisiones en la función hash (SHA-1) no produzcan falsificaciones para HMAC.
* Intercepción en la comunicación.

Extensiones

La RFC presenta varias mejoras al algoritmo HOTP. Pero éstas no son extensiones recomendadas ni parte del algoritmo estándar, sino que son simplemente variaciones que podrían usarse para implementaciones personalizadas.

1. Número de Dígitos: extraemos más dígitos del valor HMAC-SHA-1 calculado, por ejemplo, 8 dígitos. Así, el valor HOTP calculado reduciría la probabilidad de éxito del atacante al tratar de hacer fuerza bruta, desde sv/10^6 a sv/10^8.
2. Valores alfanuméricos: podríamos utilizar valores desde A-Z y 0-9. La consecuencia inmediata es que la seguridad aumenta.
3. Valores HOTP de secuencia: anteriormente se sugirió que para la resincronización es deseable ingresar una pequeña secuencia de valores HOTP. Este concepto se podría generalizar en el protocolo, y agregar un parámetro “L” que definiría la longitud de la secuencia de valores HOTP a ingresar.

Por defecto, el valor L debería estar seteada a 1, pero si la seguridad necesita ser incrementada, se puede pedir a los usuarios que ingresen L valores HOTP.

1. Un método de resincronización basado en contador (RCP): el cliente puede acceder u enviar no sólo el valor HOTP, sino también otra información, como el valor del contador.

El servidor aceptará el valor HOTP recibido solamente si las siguientes condiciones son verdades, donde C-cliente es el valor del contador del cliente y C-server el del servidor.

* + 1) C-client >= C-server
  + 2) C-client - C-server <= s
  + 3) Check that HOTP client is valid HOTP(K,C-Client)
  + 4) If true, the server sets C to C-client + 1 and client is authenticated

En este caos, n o hay necesidad de gestionar una ventana look-ahead. La probabilidad de éxito del atacante es v/10^6.

1. Campo de dato: otra opción interesante es la introducción de un campo de datos, que sería utilizado para generar los valores OTP: HOTP ( K,C,[Data] ), donde Data es un campo opcional que puede resultar de la concatenación de varias piezas de información pública, por ejemplo, Data = Dirección | PIN | Número de Teléfono.

También se podría utiliza un Timer, que podría ser el único factor de movimiento del algoritmo, o podría ser la combinación con el contador. En ese caso, Data = Timer, donde Timer podría ser Unix-time (1/1/2018) dividido por algún factor para dar un específico intervalo de tiempo.

Usar un campo de datos ofrece mayor flexibilidad en la implementación del algoritmo.

TOTP

Time-Based One-Time Password Algorithm

RFC 6238

Introducción

Publicado en el 2011.

El documento fue desarrollado por la IETF.

Se trata de una extensión del algoritmo HOTP, que soporta un factor de movimiento basado en el tiempo. El algoritmo HOTP especifica un algoritmo OTP basado en eventos, donde el factor de movimiento es un contador de eventos. TOTP basa el factor de movimiento en un valor de tiempo, el cual provee valores OTP de corta duración que son deseables para una mayor seguridad.

El algoritmo HOTP está basado en el algoritmo HMAC-SHA-1 y aplicado a un valor de contador creciente que representa el mensaje en el cálculo de la HMAC.

Básicamente, la salida del cálculo de HMAC-SHA-1 es truncada para obtener valores accesibles por el usuario:

HOTP( K,C ) = Truncate( HMAC-SHA-1( K,C ))

Donde Truncate representa la función que puede convertir un valor HMAC-SHA-1 en un valor HOTP. K y C representan la clave compartida y el valor del contador.

TOTP es la variante basada en el tiempo del algoritmo HOTP, donde el valor T, derivado de una referencia de tiempo y un time step, reemplaza el contador C en el cómputo de HOTP.

Las implementaciones TOTP pueden usar funciones HMAC-SHA-256 o HMAC-SHA-512, basadas en las funciones hash SHA-256 o SHA-512, en lugar de la función HMAC-SHA.1 que se ha especificado para el cómputo de HOTP.

Requerimientos del Algoritmo

R1: El proveedor (Token) y el verificador (Servidor de autenticación o validación) debe conocer o ser capaz de el tiempo Unix actual (Por ejemplo, el número de segundos que pasaron desde la medianoche UTC del 1 de enero de 1970) para la generación de códigos OTP. La precisión del tiempo usado por el proveedor afecta a la cantidad de veces que la sincronización del reloj debe ser realizada.

R2: El proveedor y el verificador deben compartir el mismo secreto o el conocimiento de la transformación de un secreto para generar una clave compartida.

R3: El algoritmo DEBE usar HOTP como bloque de construcción para las claves.

R4: El proveedor y el verificador deben usar el mismo valor de time-step.

R5: Debe haber una única clave por cada proveedor.

R6: La clave debe ser generada aleatoriamente o derivada usando un algoritmo de derivación de claves.

R7: Las claves deben estar almacenadas en un dispositivo adecuado y deben ser protegidas contra accesos y usos no autorizados.

Descripción General

El algoritmo TOTP es una variante del algoritmo HOTP que especifica el cálculo de un password de un solo uso. Reemplaza el factor de movimiento basado en contador por uno basado en tiempo.

* Notaciones:
  + X representa el time step en segundos (Por defecto el valor de X = 30 segundos) y se trata de un parámetro del sistema.
  + T0 es el Unix Time, que se usa para comenzar el conteo de time steps (su valor por defecto es 0, es decir, Unix epoch) y también es un parámetro del sistema.
* Descripción: Básicamente, se define TOTP como

TOTP = HOTP ( K,T ),

donde T es un integer y representa el número de time steps entre el contador de tiempo inicial T0 y el tiempo actual de Unix.

Más en detalle,

T = (Current Unix Time – T0) / X

Por ejemplo, si T0 y Time Step X = 30, T = 1 si el Current Unix Time es 59 segundos, y T=2 si el Current Unix Time es 60 segundos.

La implementación de este algoritmo DEBE soportar un valor de tiempo T mayor que un entero de 32 bits cuando está más allá del año 2038. El valor de los parámetros de sistema X y T0 se preestablecen durante el proceso de provisión y se comunican entre un probador y un verificador como parte del paso de provisión.

Consideraciones de Seguridad

La seguridad y la robustez de este algoritmo depende de las propiedades de los cimientos de HOTP, el cual está construido sobre HMAC usando SHA-1 como la función hash.

Los estudios han demostrado que los ataques con más posibilidades de éxito contra HOTP son los ataques de fuerza bruta.

Las claves deben elegirse al azar o usando un generador criptográfico apropiado.

Las claves DEBERÍAN tener la longitud de la salida HMAC para facilitar la interoperabilidad.

Toda la comunicación debería tomar lugar en un canal seguro, por ejemplo, SSL/TSL o conexiones Ipsec.

También se recomienda almacenar las claves de forma segura en el sistema de validación, y encriptarlas debidamente, exponiéndolas sólo cuando sea necesario: por ejemplo, la clave es desencriptada cuando necesita verificar un valor OTP, y vuelta a encriptar inmediatamente para limitar el tiempo de exposición de la misma.

Validaciones y Tamaño del Time-Step

Un OTP generado sin el mismo time step sería lo mismo. Cuando un OTP es recibido por el sistema de validación, no conoce el timestamp exacto del cliente cuando un OTP fue generado. El sistema de validación debería usar el timestamp cuando un OTP para una comparación OTP. Debido a la latencia de red, el gap (mencionado como T, que es el número de time steps desde T0) entre el tiempo en el que el OTP fue generado y el tiempo en el que el OTP llega y es recibido por el sistema, es demasiado grande. El tiempo de recepción en el sistema de validación y la generación real de OTP pueden no estar dentro del mismo intervalo de tiempo (time-step window) que produjo la misma OTP. Cuando un OTP es generado al final del time-step window, el tiempo de recepción caería dentro del póximo time-step window. UN sistema de validación debería setear una política aceptable para la ventana de retardo de transmisión OTP para la validación. El sistema de validación debería comparar OTPs no solo con el timestamp recibido sino también con los timestamps anteriores que están dentro del retraso de transmisión. Un window delay demasiado grande tendría la consecuencia del aumento de posibles ataques. La RFC recomienda que como máximo se permita un solo time step, por si hay demoras en la red.

El tamaño del time-step tiene un impacto fuerte sobre la seguridad y la usabilidad. Un time-step largo significa un ventana de validación más grande para que un OTP sea aceptado. Las implicaciones de usar tamaños de time-step largos son:

* Segurida: Un time step largo expone una ventana de ataque mayor. Cuando un OTP es generado y expuesto a una tercera parte antes de ser consumido, la tercera parte puede consumir el OTP dentro de la ventana de time-step. Se recomienda un tamaño de time step por defecto de 30 segundos. Se lo considera como un balance entre seguridad y usabilidad.
* Usabilidad: El siguiente OTP debe ser generado dentro del próximo window time-step. Si nuestro time-step es muy grande, el usuario deberá esperar un tiempo muy grande hasta la generación del próximo OTP.

Resincronización

Debido a las posibles fluctuaciones de reloj entre un cliente y un servidor de validación, se recomienda que el validador sea seteado con un límite específico para la cantidad de time-steps que un proveedor puede estar "fuera de sincronización" antes de ser rechazado.

Este límite se puede establecer tanto hacia adelante como hacia atrás desde el time-step calculado al recibir el valor de OTP. Si el time-step (paso del tiempo) es de 30 segundos como se recomienda, y el validador está configurado para aceptar solo dos pasos de tiempo hacia atrás, la deriva máxima de tiempo transcurrido sería alrededor de 89 segundos, es decir, 29 segundos en el paso de tiempo calculado y 60 segundos para dos hacia atrás pasos de tiempo.

Esto significaría que el validador podría realizar una validación con respecto al tiempo actual y luego dos validaciones adicionales para cada paso hacia atrás (para un total de 3 validaciones). Tras la validación exitosa, el servidor de validación puede registrar la deriva del reloj detectada para el token en términos del número de pasos de tiempo (time-step). Cuando se recibe una nueva OTP después de este paso, el validador puede validar la OTP con la marca de tiempo actual (timestamp) ajustada con la cantidad registrada de desviaciones de reloj de paso de tiempo para el token.

También, es importante tener en cuenta que cuanto más tiempo un probador no ha enviado una OTP a un sistema de validación, mayor (potencialmente) va a ser la deriva acumulada del reloj entre el probador y el verificador. En tales casos, la resincronización automática descrita anteriormente puede no funcionar si la deriva excede el umbral permitido. Se deben usar medidas de autenticación adicionales para autenticar de manera segura el probador y volver a sincronizar explícitamente la deriva del reloj entre el comprobador y el validador.

<https://www.welivesecurity.com/la-es/2018/04/23/libros-seguridad-informatica/?utm_campaign=welivesecurity&utm_source=facebook&utm_medium=social>

Funciones Hash

Las funciones hash son algoritmos que al aplicarlos sobre archivos o textos, entregan un resumen de x bits. Un número que, a modo de huella digital, representa a dichos documentos de forma supuestamente única. A diferencia de la popular familia de aplicaciones tipo zip para la compresión de archivos, las funciones hash entregan siempre un resumen con número fijo de bits, independientemente del tamaño de ese archivo. Es menester indicar que, al carecer de claves, las funciones hash no son algoritmos de cifrado.

Deben cumplir con 5 propiedades básicas:

1. La unidereccionalidad: conocido un resumen h(M), debe ser computacionalmente imposible encontrar el mensaje M a partir de dicho resumen.
2. La compresión: como lo normal es que el mensaje M tenga una longitud de bits mayor que la entrega el hash, la función hash actuará normalmente como un compresor. Si el mensaje tuviese menos bits que el hash, el resumen siempre tendrá ese valor fijo de bits.
3. La facilidad de cálculo: debe ser fácil y rápido calcular un h(M) a partir de un mensaje M.
4. La difusión de bits: el resumen h(M) debe ser una función compleja de todos los bits del mensaje M, por lo tanto, si se modifica tan sólo un bit del mensaje M, el nuevo resumen debería cambiar aproximadamente en la mitad de sus bits con respecto al anterior.
5. La resistencia débil a colisiones: esta propiedad se cumplirá si es computacionalmente imposible que conocido un mensaje M1, podamos encontrar otro mensaje M2, tal que h(M1) = h(M2).

Aplicaciones de las funciones hash

Muchas de las aplicaciones de las funciones *hash* son relativas al campo de la [criptografía](https://es.wikipedia.org/wiki/Criptograf%C3%ADa) ( [Cifradores](https://es.wikipedia.org/wiki/Cifrado_(criptograf%C3%ADa)), [acumuladores criptográficos](https://es.wikipedia.org/wiki/Acumulador_criptogr%C3%A1fico), [firma digital](https://es.wikipedia.org/wiki/Firma_digital), [protocolos criptográficos de autenticación](https://es.wikipedia.org/w/index.php?title=Protocolos_criptogr%C3%A1ficos_de_autenticaci%C3%B3n&action=edit&redlink=1), etc.).

SHA

El **SHA** (*Secure Hash Algorithm*, Algoritmo de *Hash* Seguro) es una familia de [funciones hash](https://es.wikipedia.org/wiki/Funci%C3%B3n_hash) de cifrado publicadas por el [Instituto Nacional de Normas y Tecnología](https://es.wikipedia.org/wiki/Instituto_Nacional_de_Normas_y_Tecnolog%C3%ADa) (NIST) de EE. UU. La primera versión del algoritmo se creó en [1993](https://es.wikipedia.org/wiki/1993) con el nombre de SHA, aunque en la actualidad se la conoce como **SHA-0** para evitar confusiones con las versiones posteriores. La segunda versión del sistema, publicada con el nombre de **SHA-1**, fue publicada dos años más tarde. Posteriormente se han publicado [SHA-2](https://es.wikipedia.org/wiki/SHA-2) en [2001](https://es.wikipedia.org/wiki/2001) (formada por diversas funciones: **SHA-224**, **SHA-256**, **SHA-384**, y **SHA-512**) y la más reciente, **SHA-3**. Esta última versión se caracteriza por ser la que más difiere de sus predecesoras.

SHA-1

SHA-1 producen una salida resumen de 160 bits (20 bytes) de un mensaje que puede tener un tamaño máximo de 264 bits, y se basa en principios similares a los usados por el profesor [Ronald L. Rivest](https://es.wikipedia.org/wiki/Ronald_L._Rivest) del [MIT](https://es.wikipedia.org/wiki/MIT) en el diseño de los algoritmos de resumen de mensaje [MD4](https://es.wikipedia.org/wiki/MD4) y [MD5](https://es.wikipedia.org/wiki/MD5). SHA-1 produce un hash de 160 bits, contra los 128 de MD5, y su fortaleza es también mayor

funcionamiento:

- SHA-1(x) = hash.

- El mensaje x puede tener un tamaño máximo de 264 bits. la función está preparada para aceptar caracteres

- la salida hash será siempre de 160 bits (160 bits/4 bits = 40 digitos hex)

Algunos ejemplos de hashes son los siguientes:

* SHA-1(“”) = DA39 A3EE 5E6B 4B0D 3255 BFEF 9560 1890 AFD8 0709
* SHA-1(“Hello World!”) = 2ef7 bde6 08ce 5404 e97d 5f04 2f95 f89f 1c23 2871
* SHA-1(“hello world!”) = 430c e34d 0207 24ed 75a1 96df c2ad 67c7 7772 d169

Un pequeño cambio en el mensaje provoca un hash completamente diferente.

<https://es.wikipedia.org/wiki/Secure_Hash_Algorithm>

<http://msmanuelsanchez.blogspot.com.ar/2015/10/tipos-de-cifrados-sha1-md5-rsa.html>

<http://cerezo.pntic.mec.es/fherna15/hash1010.htm>

<https://www.youtube.com/watch?v=pG8785ZEFuM>

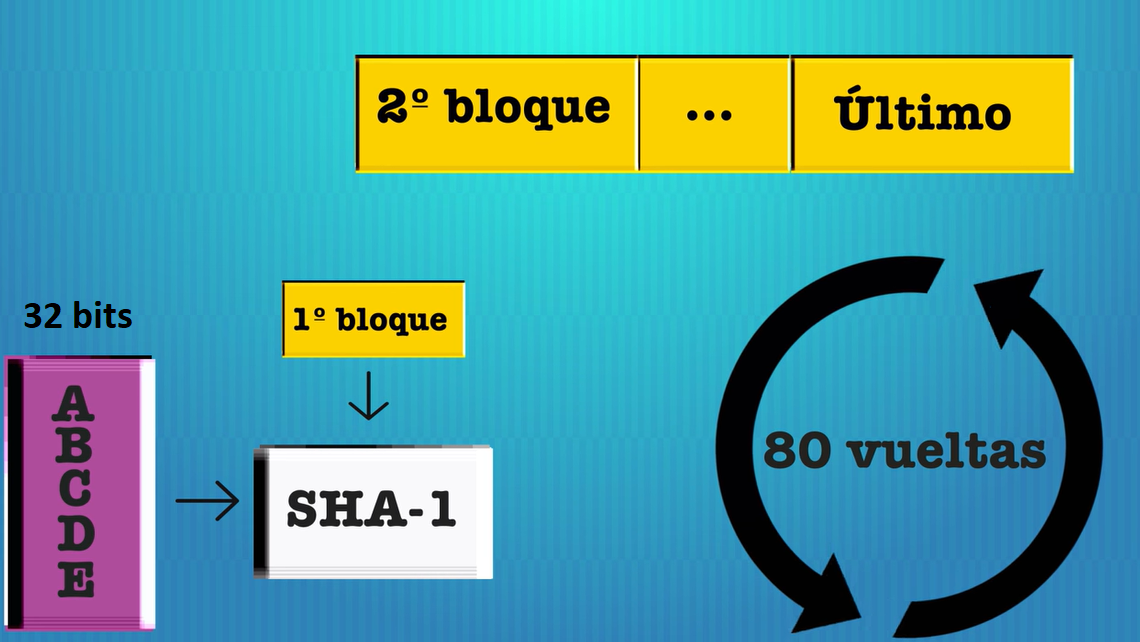
¿Cómo funciona el Hash SHA-1?

Al igual que en MD5, el mensaje se agrupa en bloques de 512 bits, con 16 palabras de 32 bits. Para completar el último bloque de 512 bits se aplicará un relleno de un 1 seguido de tantos ceros como haga falta, reservando los últimos 64 bits para indicar el tamaño del mensaje original.



Internamente, SHA-1 trata las palabras en representación Big-Endian, es decir, primero se leen los bytes más significativos y después los bytes menos significativos, que en el fondo es la forma natural de hacerlo. Por lo tanto, la palabra de 32 bits “Hola” que en Hexadecimal es “486f6c61” se leerá en el siguiente orden: 48 – 6f – 6c – 61.

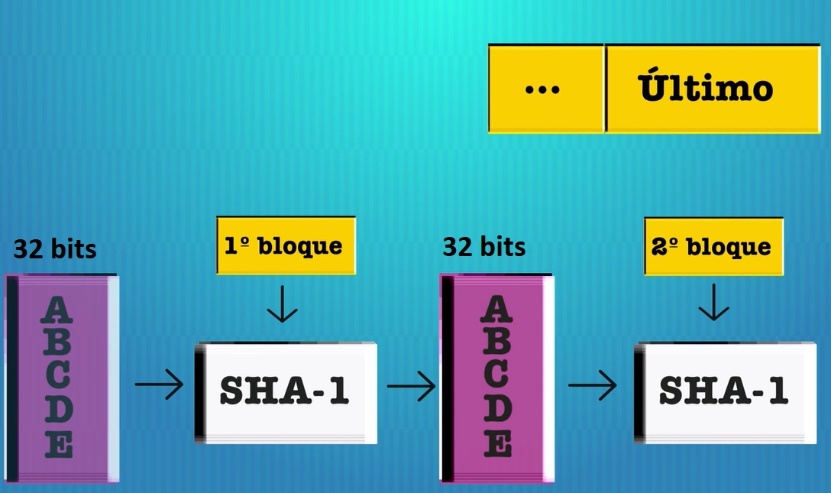
SHA-1 usa 5 vectores públicos de 32 bits conocidos como A, B, C, D y E respectivamente, que se mezclarán con las palabras del texto en claro también de 32 bits, realizando para cada bloque de entrada de 512 bits un total de 80 vueltas.



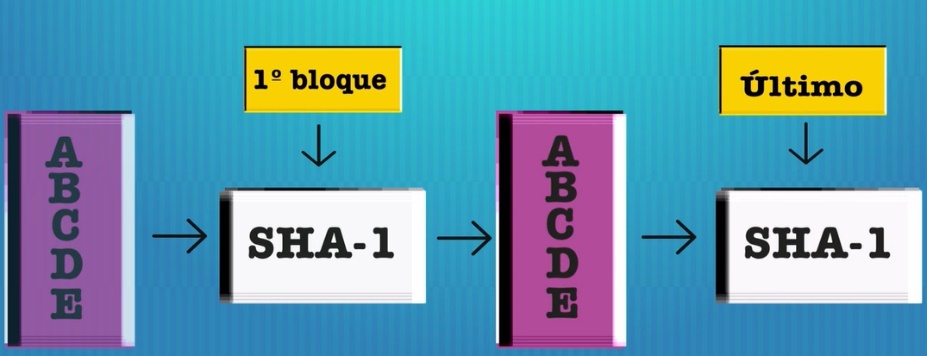
Cada uno de estos vectores tiene un tamaño de 32 bits, y los valores con los que se inicializan son los siguientes (valores hexadecimales):

* A=67452301
* B=EFCDAB89
* C=98BADCFE
* D=10325476
* E= C3D2E1F0

Terminadas las 80 vueltas, los vectores iniciales A, B, C, D y E, habrán cambiado varias veces de valor, y serán de nuevo los 5 vectores que se mezclarán con el segundo de 512 bits de entrada.



Esta acción se va encadenando con los siguientes bloques, hasta procesar el último.

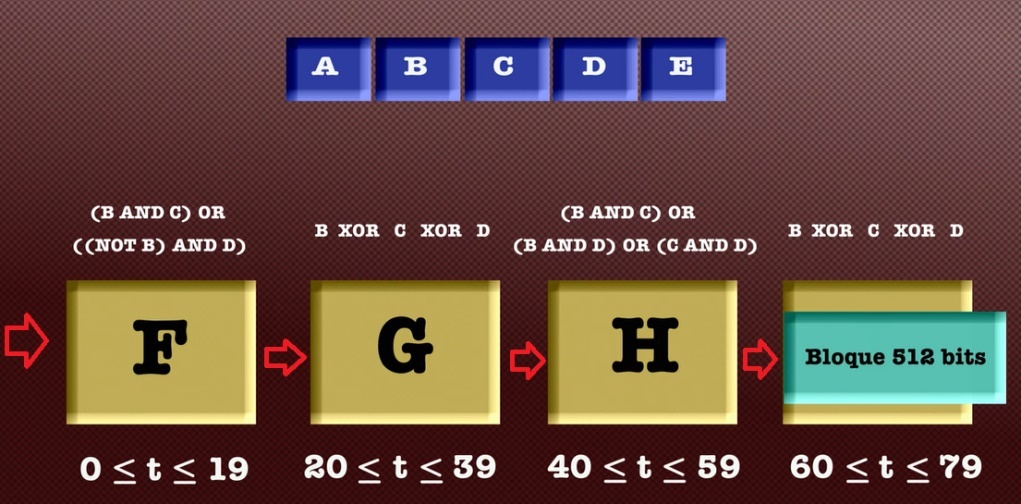


Finalmente, el último valor de los vectores A, B, C, D y E es el resumen de 160 bits de todo el mensaje.

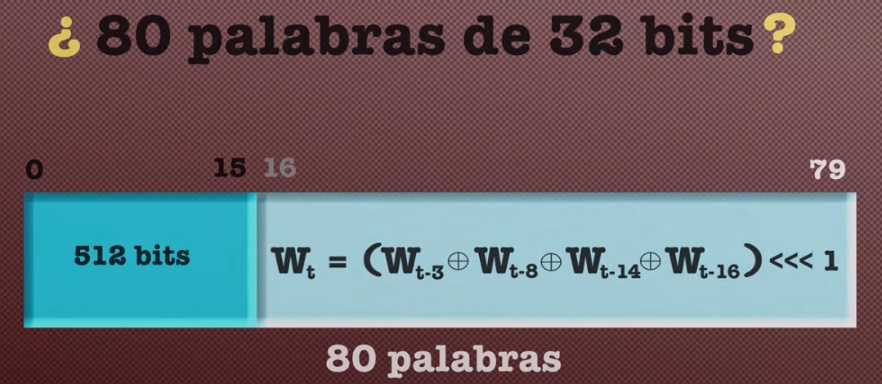


Las 80 vueltas de SHA-1:

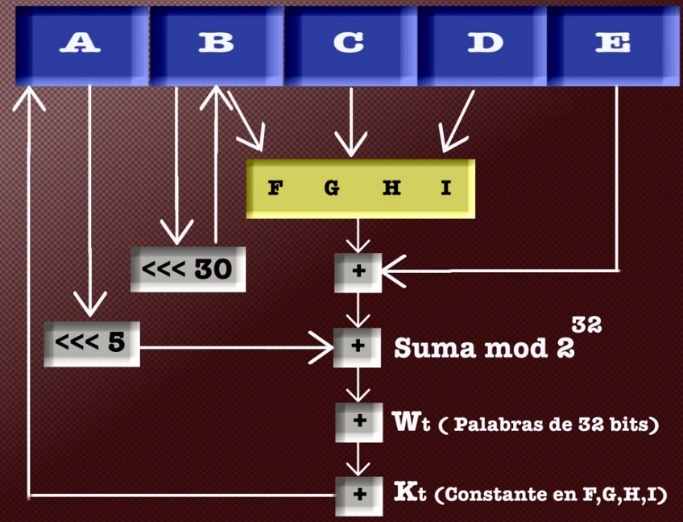
A cada bloque del texto de entrada de 512 bits se le aplicarán 20 vueltas con 4 funciones distintas (a las que llamaremos F, G, H, I). De forma tal que el número total de vueltas por bloque será igual a 20 x 4 = 80.



Pero, ¿cómo podemos generar 80 palabras de 32 bits a partir sólo de los 512 bits de entrada? Cada bloque de 16 palabras del mensaje se expandirá en 80 palabras, de forma que las palabras de 0 a 15 se corresponden a los 512 bits del bloque, y las 64 palabras restantes de la 16 hasta la 79 se obtienen mediante una operación lógica entre las palabras anteriores.

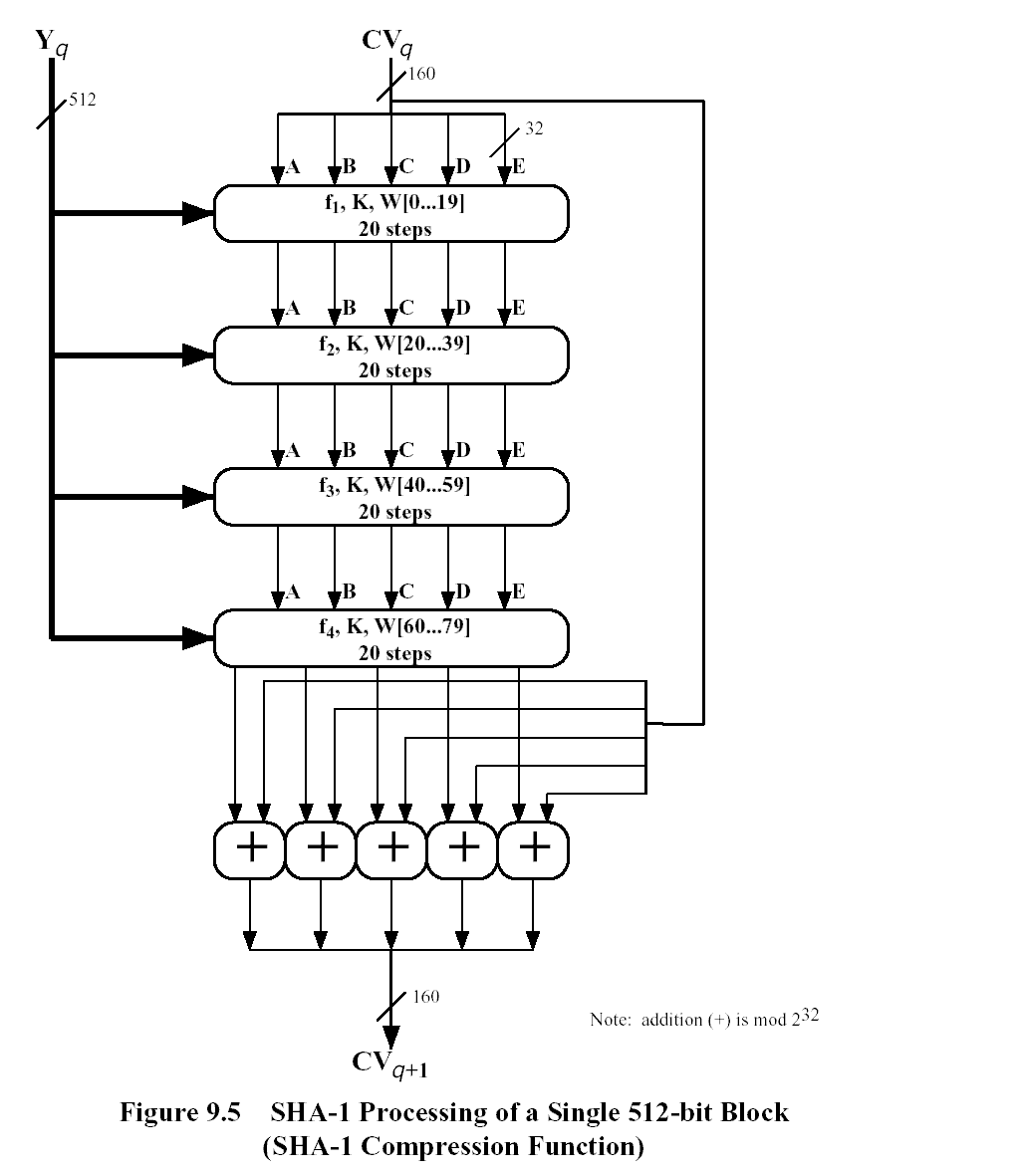


La siguiente figura muestra las operaciones que se realizan sobre cada bloque de entrada en SHA-1, así como el desplazamiento del registro de vectores.



La siguiente figura muestra un resumen del proceso: Se procesa el mensaje por bloques de 512 bits, cada uno pasa por un módulo que consta de 4 rondas de procesamiento de 20 pasos cada una. Las rondas tienen una estructura similar, con la excepción de que cada una ocupa una función lógica primitiva diferente (f1, f2, f3 y f4).

La entrada a cada ronda consta del bloque de 512 bits que se esté procesando (Yq) y los 160 bits de la memoria MD, nótese que cada bloque de 512 bits actualizará el valor de la memoria temporal. Cada ronda también hace uso de la constante aditiva Kt, donde 0<= t <= 79 indica uno de los 80 pasos a lo largo de las cuatro rondas.



Ataques contra SHA-1

**SHA-1** ha sido examinado muy de cerca por la comunidad criptográfica pública y no se ha encontrado ningún ataque eficaz. No obstante, en el año [2004](https://es.wikipedia.org/wiki/2004), se dio a conocer un número significativo de ataques contra funciones criptográficas de *hash* con una estructura similar a SHA-1, lo que plantea dudas sobre la seguridad a largo plazo de SHA-1.

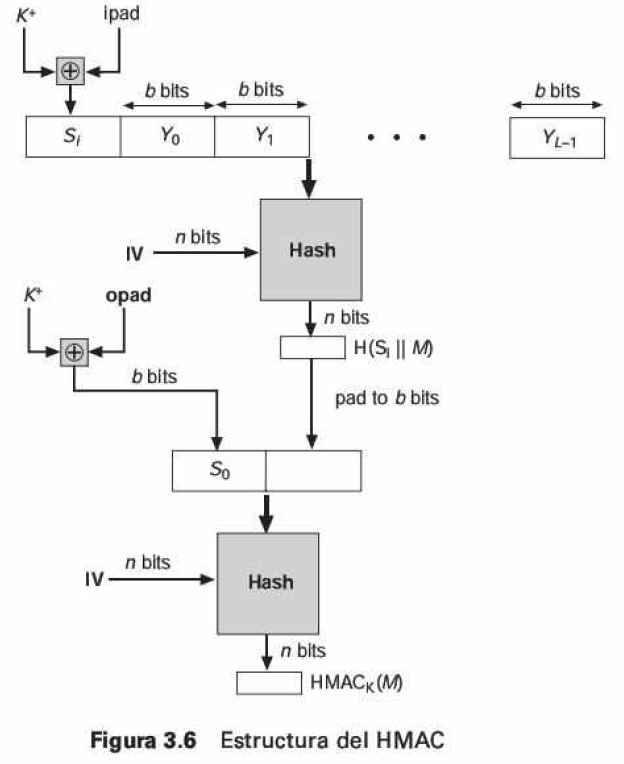
A lo largo de su historia, se han producido algunos ataques a la familia de algoritmos SHA:

* En [1998](https://es.wikipedia.org/wiki/1998) se encontró una vulnerabilidad en SHA-0, aunque esta no se podía hacer extensiva a **SHA-1**. En cualquier caso, la [NSA](https://es.wikipedia.org/wiki/Agencia_de_Seguridad_Nacional) aumentó en ese momento la seguridad del **SHA-1**.
* En [2004](https://es.wikipedia.org/wiki/2004) se encontró una debilidad matemática en SHA-1, que permitiría encontrar colisiones de hash más rápido. Sin embargo, este hallazgo resulta poco relevante, pues la complejidad de búsqueda de colisiones pasaría de 280 a 269, algo que aún es [computacionalmente inviable](https://es.wikipedia.org/wiki/Complejidad_computacional), requiriendo incluso más trabajo que [MD5](https://es.wikipedia.org/wiki/MD5) (264).
* El 23 de febrero de 2017, un equipo formado por [Google](https://es.wikipedia.org/wiki/Google) y CWI Amsterdam anunció la primera colisión de SHA-1, la cual ha sido nombrada como [SHAttered](https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html). Poniendo en duda el algoritmo.

RFC de SHA-1: <https://www.ietf.org/rfc/rfc3174.txt>

HMAC

Funcionamiento:



Entonces el HMAC puede expresarse de la siguiente manera:



Estructura HMAC:

* IV (Initial Value) = valores iniciales de entrada para la función hash utilizada. Por ejemplo, si usamos SHA-1, las entradas serán los 5 vectores iniciales de dicha función.
* M = mensaje de entrada para HMAC (incluyendo el relleno especificado en la función hash empotrada).
* K = clave secreta. Se recomienda que su longitud sea mayor o igual a “n”. Importante: si longitud de la clave es mayor que “b”, la clave se introduce en la función hash utilizada para producir una clave de salida de “n” bits. En cambio, si la clave es menor a “b”, se le agregan ceros a la izquierda para que el resultado tenga “b” bits de longitud. A este valor lo llamamos (K+).
* ipad = constante cuyo valor es 00110110 (36 en hexadecimal) repetido b/8 veces.
* opad = constante cuyo valor es 01011100 (5C en hexadecimal) repetido b/8 veces.
* Hash = es la función hash embebida (MD5, SHA-1, RIPEMD-160).
* Yi = representa al bloque i-ésimo de M, donde 0 <= i <= (L - 1).
* (K+): es la clave K a la cual se le han agregado ceros a la izquierda, para que tenga una longitud igual a “b” bits.
* L = número de bloques que hay en el mensaje M.
* b = número de bits que hay en un bloque.
* n = longitud del código hash producido por la función H.

Algoritmo:

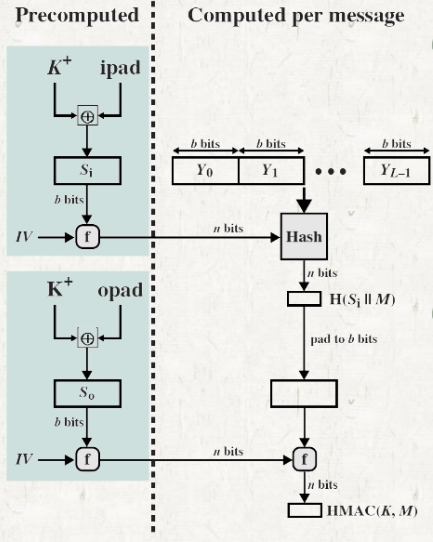
1. Primero agarro la clave secreta “K” y le agrego tantos ceros a la izquierda como sean necesarios para generar una clave (K+) cuyo tamaño es “b” bits.
   * Si “K” tiene un tamaño de 160 bits (20 bytes) y b = 512 bits (64 bytes), se le agregarán a “K” 44 bytes ceros 0x00, y este valor se llamará (K+).
   * El hecho de agregar ceros a la izquierda no me modifica el valor de K. Ejemplo: si K = 10 (2 en decimal), entonces si yo agrego 6 ceros a la izquierda, no me modifica nada, (K+) = 00000010.
2. Aplico la operación lógica XOR entre (K+) y ipad para producir un bloque de “b” bits al que llamaremos S(i).
   * ipad = 00110110
3. Añado “M” a S(i).
4. Aplico la función hash H al stream generado en el paso 3.
5. Aplico la operación lógica XOR entre (K+) y opad para producir un bloque de “b” bits al que llamaremos S(o).
   * opad = 01011100
6. Agrego el resultado de la función hash del paso 4 a S(o).
7. Aplico nuevamente la función hash H al stream generado en el paso 6 para obtener finalmente la salida que deseamos: HMAC(K,M).

Hay que tener en cuenta que en el XOR con el ipad da como resultado la inversión de la mitad de los bits de K. De forma parecida, el XOR con el opad da como resultado la inversión de la mitad de los bits de K, pero un grupo diferente de bits. En efecto, al pasar S(i) y S(o) por el algoritmo hash, hemos generado de forma pseudoaleatoria dos claves de K.

El HMAC debería ejecutarse en aproximadamente el mismo tiempo que la función hash empotrada para mensajes largos. El HMAC añade tres ejecuciones de la función hash básica (para S(i), S(o) y el bloque producido por el hash interno).

Algunas cosas para tener en cuenta:

* HMAC debería ejecutarse aproximadamente en el mismo tiempo que la función hash embebida, sin importar el tamaño del mensaje.
* HMAC incorpora 3 ejecuciones de la función hash. La primera de ellas la usamos cuando la longitud de K es mayor a “b”. La segunda la usamos en el paso 4, mientras que la tercera en el paso 7.
* Una implementación más eficiente de HMAC sería posible precomputando lo siguiente:
  + f( IV , ( K+ XOR ipad ) )
  + f( IV , ( K+ XOR opad ) )
  + Nota: la siguiente figura ilustra esto.



Estos valores solo necesitan ser computados inicialmente y cada vez que la clave cambia.

Los valores precalculados sustituyen al valor inicial.

Solo se agrega una instancia adicional de la función de compresión al procesamiento.

La Seguridad de HMAC

* La seguridad de cualquier función MAC basada en una función hash embebida depende de cierto modo en la fuerza criptográfica de dicha función hash.
* El atractivo de HMAC es que sus diseñadores han podido demostrar una relación exacta entre la fuerza de la función hash incorporada y la fuerza de HMAC.
* La seguridad de HMAC está expresada en términos de la probabilidad de falsificación exitosa con:
  + Una cantidad dada de tiempo gastado por el falsificador-
  + Un número dado de pares de mensajes MAC credos con la misma clave.
* Para un nivel de esfuerzo dado (tiempo, pareja de mensajes MAC) sobre mensajes generados por un usuario legítimo y vistos por el atacante, la probabilidad de un ataque exitoso sobre HMAC es equivalente a uno de los siguientes ataques.

La probabilidad

<https://tools.ietf.org/html/rfc2104>

<https://es.wikipedia.org/wiki/HMAC>

<https://www.youtube.com/watch?v=sxWxCtJ3og0>

<https://es.slideshare.net/freleoabad/encriptacion-hmac>

<https://books.google.com.ar/books?id=cjsHVSwbHwoC&pg=PA69&lpg=PA69&dq=hmac+funcionamiento&source=bl&ots=ZoxN--EaKB&sig=TBCxEKOVTA-klaEJ8VemTfsIMdQ&hl=es&sa=X&ved=0ahUKEwjgmv-YvoXbAhVLPJAKHVJsDmgQ6AEITzAC#v=onepage&q&f=false>

imagen: <https://en.wikipedia.org/wiki/File:Hmac.png#filelinks>

todo: <https://www.slideshare.net/JamesWong107/hash-mac-algorithms-61333526>