Agenzia per la
Cybersicurezza Nazionale

GPDP | GARANTE
PER LA PROTEZIONE
DEI DATI PERSONALI

LINEE GUIDA FUNZIONI CRITTOGRAFICHE

Conservazione delle Password

DICEMBRE 2023

Questo documento, che costituisce parte delle "Linee Guida Funzioni Crittografiche", elaborato dall'Agenzia per la Cybersicurezza Nazionale, d'intesa con il Garante per la Protezione dei Dati Personali, contiene le raccomandazioni in merito alla conservazione delle password.
Il documento tiene in considerazione le minacce presenti al giorno della sua pubblicazione. Data la diversa natura dei sistemi informativi di destinazione, non è possibile garantire che queste raccomandazioni possano essere utilizzate senza adattamenti specifici.
In qualsiasi caso, la pertinenza dell'attuazione delle soluzioni proposte deve essere sottoposta, preventivamente, a valutazione e validazione da parte dei responsabili della sicurezza dei sistemi informativi di destinazione.

Il documento è stato curato, in particolare, da Simone Dutto, Sergio Polese e Giordano Santilli, esperti crittografi in forza alla Divisione Scrutinio Tecnologico, Crittografia e Nuove Tecnologie del Servizio Certificazione e Vigilanza di ACN.

Si ringraziano per la collaborazione, in particolare, Dorotea Alessandra De Marco e Marco Coppotelli, funzionari dell'Autorità Garante per la Protezione dei Dati Personali.

Versione	Data di pubblicazione	Note
1.0	**07/12/2023**	**Prima pubblicazione**

Sommario

pag.

1. Introduzione **5**
2. Password hashing **6**
2.1. Il salt **7**
2.2. Il pepper **8**
2.3. Aggiornamento della funzione di password hashing **8**
3. Principali algoritmi **9**
3.1. PBKDF2 **9**
3.2. scrypt **11**
3.3. bcrypt **12**
3.4. Argon2 **13**
4. Conclusioni **16**
Bibliografia **18**

Indice delle figure

Figura 1 - Algoritmo di PBKDF2 **10**
Figura 2 - Algoritmo BlockMix interno a scrypt **11**
Figura 3 - Funzione Expand di bcrypt **13**
Figura 4 - Funzione di compressione di Argon2 **14**

Indice delle tabelle

Tabella 1 - Algoritmi di password hashing raccomandati con relativi parametri minimi **17**

Lista dei simboli matematici utilizzati

$\{0,1\}$	Campo binario dei valori assumibili da un singolo bit	$\oplus$	Operazione XOR, ovvero somma bit a bit tra stringhe binarie
$\{0,1\}^n$	Spazio vettoriale delle stringhe binarie di lunghezza n	$\mathrm{int}(i)$	Conversione dell'intero i in una stringa binaria di 32 bit
$\{0,1\}^*$	Insieme di stringhe binarie di lunghezza arbitraria	$\lfloor x \rfloor$	Parte intera inferiore di x, ovvero il più grande intero minore o uguale a x
$\|$	Concatenazione di stringhe	$M_{i,j}$	L'elemento della matrice M che si trova nella riga i e colonna j

1 Introduzione

Al giorno d'oggi, l'accesso alla maggior parte dei sistemi e servizi informatici prevede il superamento di procedure di autenticazione informatica a uno o più fattori che, spesso, comprendono l'utilizzo di una parola chiave (password).
La gestione delle password è un aspetto fondamentale nell'ambito della sicurezza informatica e della protezione dei dati personali. Pertanto, i gestori dei sistemi e servizi devono prevedere misure tecniche e organizzative efficaci per l'archiviazione, la conservazione e l'utilizzo delle password.
Gli archivi in cui sono conservate le password finiscono sempre più frequentemente nelle mani di soggetti esterni a seguito di attacchi informatici e vengono poi pubblicati online o utilizzati per condurre altri attacchi.

Per tali ragioni, è estremamente raccomandato l'utilizzo di robuste funzioni crittografiche di password hashing.
Questo documento ha l'obiettivo di fornire indicazioni e raccomandazioni sulle funzioni ritenute attualmente più sicure.
Il documento presenta la seguente struttura: nel capitolo 2 si introduce il concetto di password hashing, focalizzando l'attenzione sulle proprietà che le funzioni devono soddisfare e sui possibili attacchi a cui gli archivi di password possono essere soggetti; nel capitolo 3 si presentano nel dettaglio gli algoritmi più comuni utilizzati per il password hashing. Infine, nel capitolo 4 vengono fornite le indicazioni su quali sono gli algoritmi raccomandati e sui rispettivi parametri.

2 Password hashing

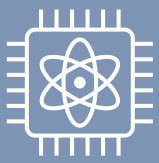
Una funzione di hash prende in input una stringa di bit di lunghezza arbitraria e restituisce una stringa di bit di lunghezza fissa, detta **digest**. I dettagli e le raccomandazioni per le funzioni di hash si possono trovare nel documento dedicato.
Una delle proprietà delle funzioni di hash crittografiche è quella di essere **one-way**, ovvero non invertibile: dato un digest, è computazionalmente difficile trovare un input che permetta di ottenerlo tramite la funzione di hash. Tale caratteristica rende le funzioni di hash adatte alla conservazione delle password: invece che memorizzare nell'archivio le password in chiaro, si salva il loro digest in modo che qualsiasi soggetto malintenzionato eventualmente venuto in loro possesso non possa disporre direttamente delle password, ma solamente del loro digest. In questo contesto si parla quindi di **password hashing** [1]. L'estrema complessità nell'invertire le funzioni di hash rende impossibile per l'attaccante il recupero delle password originali degli utenti. Al contrario, è semplice verificare la correttezza della password dell'utente al momento della richiesta di accesso, in quanto è sufficiente calcolarne il digest tramite la funzione di hash utilizzata in fase di salvataggio della password e confrontare il valore ottenuto con quello presente all'interno dell'archivio.
Nonostante il password hashing migliori la sicurezza della conservazione delle password, è importante prestare attenzione ad alcuni aspetti delicati, che possono lasciare spazio a vulnerabilità. Il principale scenario da tenere in considerazione è il **data breach**, ovvero l'incidente in cui trapela tutta o parte della lista contenente i digest di password salvati su un server. Una volta ottenuta la lista delle coppie utente-digest, l'attaccante può procedere per **forza bruta**, cioè calcolando l'hash di password casuali fino a trovare una corrispondenza, oppure può effettuare un **attacco al dizionario** [2]. Questo tipo di attacco sfrutta il fatto che, solitamente, gli utenti scelgono password banali o comunque parole di senso compiuto, per cui un malintenzionato potrebbe calcolare l'hash delle password più comuni fino a trovare una corrispondenza all'interno della lista trapelata.
È importante osservare come chiunque sia in possesso dell'archivio sia anche in grado di individuare tutti gli utenti che utilizzano la stessa password, poiché, in questo caso, anche i digest sono uguali.

È importante non utilizzare direttamente le comuni funzioni di hash crittografiche, in quanto queste ultime risultano ottimizzate per effettuare calcoli in maniera molto rapida ed esistono metodi per velocizzare la generazione dei digest che renderebbero eventuali attacchi molto più veloci.

Risulta allora fondamentale che gli algoritmi di password hashing presentino i seguenti requisiti:

- una **complessità computazionale** tale per cui sia rapido calcolare un singolo digest, ma sia eccessivamente dispendioso calcolarne un numero elevato, così da scoraggiare gli attaccanti a cercare le password degli utenti procedendo per tentativi;
- una **capacità di memoria** richiesta tale da saturare la RAM quando molti digest vengono calcolati contemporaneamente.

Per tali ragioni, quando si parla di password hashing, servono algoritmi ad hoc che mirino a rallentare le capacità offensive dell'attaccante.
Una soluzione alternativa, che rispetta i requisiti sopra descritti, potrebbe prevedere la cifratura delle password. Tuttavia, questa opzione è sconsigliata, principalmente a causa della complessità della gestione delle chiavi crittografiche utilizzate in fase di cifratura delle password.



Quantum-safe

Come la maggior parte della crittografia simmetrica, le funzioni di password hashing risultano suscettibili agli attacchi perpetrati da un computer quantistico tramite l'**algoritmo di Grover** [3], che garantisce, tuttavia, solo un aumento quadratico della velocità degli attacchi di forza bruta. Quindi, un raddoppio delle dimensioni del digest permette di garantire lo stesso livello di sicurezza degli standard attuali, rendendo di fatto vani i miglioramenti quantistici.

2.1. Il salt

Come osservato precedentemente, in caso di acquisizione dell'archivio delle password da parte di soggetti malintenzionati, l'attacco al dizionario risulta una valida alternativa alla semplice forza bruta.
In aggiunta, per ridurre lo sforzo computazionale e, quindi, le tempistiche di attacco a discapito della memoria utilizzata, sono spesso utilizzate delle **rainbow tables** o "tabelle arcobaleno" [4], ovvero tabelle precompilate contenenti i digest di un numero elevato di password comuni. Questi digest possono essere direttamente confrontati con quelli nell'archivio, rendendo, di fatto, l'attacco estremamente rapido.
Per potersi difendere da questo attacco, si consiglia di utilizzare algoritmi di password hashing che prevedono l'aggiunta di un **salt** [5] ad ogni password. Il salt è una stringa di bit casuali che viene concatenata alla password prima del calcolo del digest e poi salvata in chiaro insieme al digest della password dell'utente. Nel caso in cui si desideri un maggiore livello di sicurezza, il salt può anche essere salvato in un archivio differente da quello contenente le password.
Il salt non ostacola la procedura di autenticazione informatica dell'utente: essendo salvato in chiaro, è sufficiente concatenarlo alla password inserita dall'utente in fase di accesso prima di applicare l'algoritmo di password hashing utilizzato. Nonostante non sia cifrato, il salt fornisce protezione su diversi fronti:

- in un attacco al dizionario, quando l'attaccante calcola il digest di una password, per ogni utente deve concatenare il salt corrispondente, che sarà valido solo per quel singolo utente. Quindi, il numero di applicazioni della funzione di hash che l'attaccante dovrà effettuare per tentare di individuare gli utenti che utilizzano una determinata password cresce all'aumentare della dimensione dell'archivio delle password degli utenti;

- due utenti che utilizzano la stessa password avranno diverso salt e quindi verranno loro associati due diversi digest. Un attaccante, dunque, non potrà individuare più utenti che utilizzano la medesima password con un'unica computazione;
- le rainbow table diventano inutilizzabili, in quanto anche le password più comuni vengono modificate dal salt e quindi è necessario ricalcolare la tabella aggiungendo tutti i possibili salt per ogni password, rendendo, chiaramente, il processo più oneroso.

Quindi, in generale, il salt non aumenta il livello di sicurezza della conservazione della password di un utente specifico, ma consente di rallentare le capacità offensive di un attaccante sull'intero archivio in modo direttamente proporzionale alla sua dimensione.
Per poter risultare efficace, il salt dovrebbe:

- essere generato casualmente per ogni password;
- avere una lunghezza adeguata, altrimenti un attaccante avrebbe comunque la capacità di precalcolare una particolare rainbow table, concatenando tutti i possibili salt alle password più comuni.

2.2. Il pepper

Per aggiungere un ulteriore livello di sicurezza, viene, a volte, sfruttato un altro strumento crittografico chiamato **pepper** [6]. Il pepper è una stringa di bit casuale che, al contrario del salt, può essere la stessa per tutte le password nell'archivio, ma deve essere tenuta segreta in quanto viene utilizzata come chiave di un HMAC o di un meccanismo di cifratura simmetrico applicato al digest della password.
Quindi, in caso di pepper, l'archivio conterrà gli HMAC o le cifrature dei digest delle password, che andranno confrontati con quelli ottenuti a partire dalle password inserite dagli utenti.

2.3 Aggiornamento della funzione di password hashing

Nel caso in cui si voglia incrementare il livello di sicurezza di un sistema di autenticazione che utilizza una vecchia funzione di password hashing h, adottando un algoritmo H più sicuro, è necessario provvedere a un aggiornamento dei digest delle password già salvati nell'archivio. Una possibile strategia è riportata di seguito:

1. si applica la nuova funzione H ai vecchi digest ottenuti tramite h, sostituendo nell'archivio i vecchi valori con quelli ottenuti, cioè, per ogni password P, si salva $H(h(P))$, invece di $h(P)$. Calcolare il digest di un digest, infatti, non comporta un rischio per la sicurezza se si utilizzano funzioni di password hashing adeguate;
2. in corrispondenza di ogni elemento contenuto nell'archivio, si inserisce un flag per tener traccia della necessità di aggiornare il digest;
3. al primo accesso di ogni utente, se è necessario aggiornare il digest, dopo aver controllato la corrispondenza tra $H(h(P))$ e il valore salvato nell'archivio, si calcola $H(P)$, che viene quindi salvato come nuovo valore del digest, e si disattiva il suddetto flag. In questo modo, in seguito, verrà utilizzato solo l'algoritmo H.

In certi contesti, altri metodi che comportano l'obbligo di aggiornamento delle password da parte degli utenti, salvando direttamente il digest ottenuto con il nuovo algoritmo di password hashing, potrebbero essere una soluzione più rapida.

3 Principali algoritmi

Come osservato precedentemente, è fortemente sconsigliato calcolare un digest per la conservazione delle password tramite la semplice applicazione singola di una funzione di hash crittografica come quelle indicate nel documento dedicato. In questo ambito, una pratica comune è adottare **funzioni di derivazione di chiave** (o key derivation function) [1], ideate per ottenere una o più chiavi segrete a partire da un valore iniziale segreto, come ad esempio una **master key**.
Nonostante il differente scopo di costruzione, tali funzioni soddisfano eccellentemente le proprietà richieste per il password hashing e quindi sono tra gli algoritmi più utilizzati. Le funzioni di derivazione di chiave sono spesso configurabili a seconda delle necessità rispetto a tutte o ad alcune tra le seguenti caratteristiche:

- il **costo computazionale**, solitamente rappresentato dal numero di iterazioni di una funzione dell'algoritmo che, se aumentato, ne rallenta l'esecuzione;
- la **memoria utilizzata**, ovvero la memoria massima necessaria per eseguire l'algoritmo. In questo contesto, è importante evidenziare che la memoria richiesta da tutte le operazioni non deve discostarsi molto dal valore massimo, altrimenti questa diversa distribuzione potrebbe fornire delle informazioni all'attaccante (attacchi **side-channel** [1]);
- la **possibilità di parallelizzazione**, in quanto l'algoritmo potrebbe essere suddiviso in più parti l'una indipendente dall'altra e quindi eseguibili simultaneamente. Nel caso questa possibilità esista, l'utilizzatore può solitamente scegliere il numero di processi simultanei.

Queste caratteristiche vengono definite da parametri personalizzabili. Se da un lato viene così resa possibile l'applicazione in diversi contesti, dall'altro questa libertà richiede una responsabilità nella scelta di parametri che non inficino la sicurezza dell'algoritmo.
Di seguito sono riportate alcune raccomandazioni sugli algoritmi per il password hashing, comprensive di indicazioni sui valori minimi di questi parametri.

3.1 PBKDF2

Password Based Key Derivation Function 2 (PBKDF2) è una funzione di derivazione di chiave basata su password progettata nel 1999 e pubblicata nel 2000 come parte della serie "Public Key Cryptography Standards" (PKCS) [7]. Queste pubblicazioni fanno parte degli standard ideati e pubblicati dagli RSA laboratories, una società specializzata in sicurezza informatica che possiede, tra gli altri, il brevetto del famoso sistema crittografico a chiave pubblica RSA. Il funzionamento di PBKDF2 è illustrato in Figura 1: la funzione prende in input una password P, un salt S e, tramite l'applicazione di un HMAC ripetuto per un numero preimpostato di iterazioni c, calcola una stringa DK di lunghezza len. In formule,

$$DK = \mathrm{PBKDF2}(P, S, c, HMAC, len).$$

Per informazioni più approfondite sugli algoritmi per generare HMAC si rimanda al documento dedicato ai codici di autenticazione del messaggio.
Il numero di blocchi ℓ che andranno a comporre DK è il minimo che permette di raggiungere la lunghezza len desiderata. Nel caso in cui $\ell > 2^{32}$, PBKDF2 restituisce un errore e non procede con la computazione. Questo evita la creazione di un digest eccessivamente lungo da memorizzare.
Lo stato di partenza per il blocco i-esimo è formato dalla concatenazione del salt con il contatore i convertito in binario da 32 bit (tipo integer), ovvero $S\|\text{int}(i)$.
Questo valore viene aggiornato effettuando per c volte consecutive l'algoritmo $HMAC$ usando sempre la password P come chiave. Il valore finale T_i del blocco è lo XOR ($\oplus$) tra tutti gli stati parziali ottenuti.
È importante osservare come le operazioni svolte sui singoli blocchi siano completamente indipendenti le une dalle altre e quindi i valori finali T_i possano essere ottenuti in parallelo.
Infine, il valore DK è la concatenazione di tutti gli stati T_i, ovvero

$$DK = T_1 \| \dots \| T_\ell,$$

in cui l'ultimo blocco viene troncato in modo da ottenere la lunghezza len desiderata.

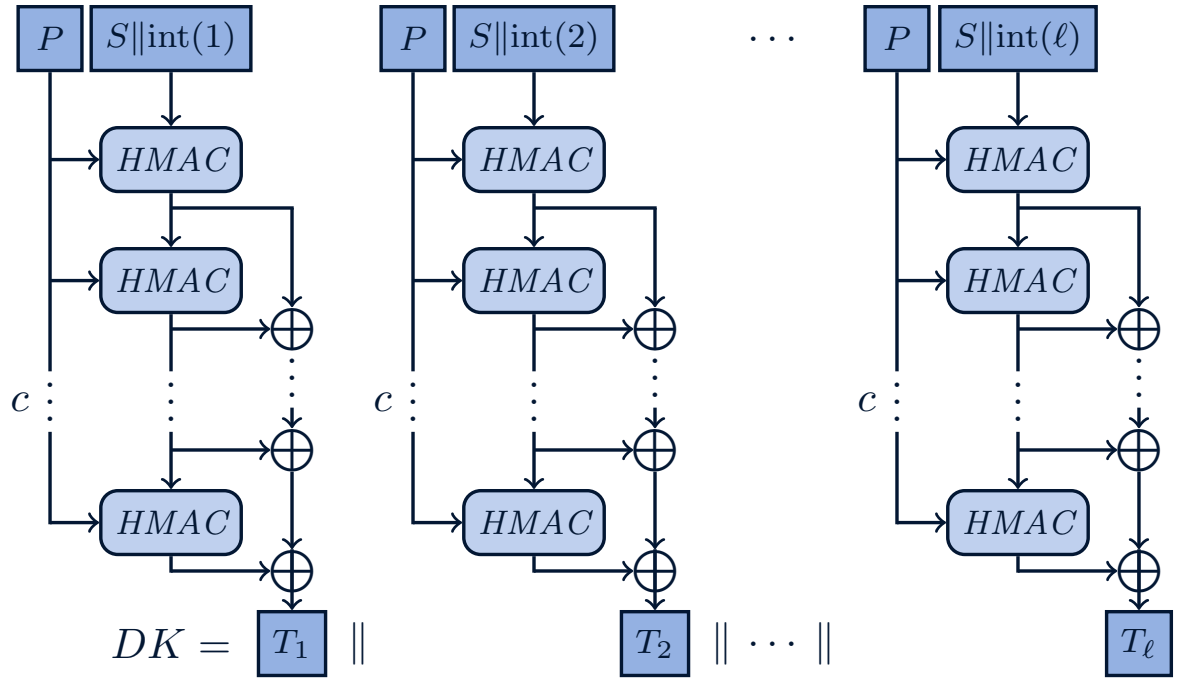


Figura 1 - Algoritmo di PBKDF2

Secondo le specifiche del NIST del 2010 [8] e il RFC del 2017 [9], il salt S deve essere lungo almeno 128 bit ottenuti tramite un generatore di numeri casuali approvato nelle raccomandazioni indicate in [10], [11], [12], il numero di iterazioni c deve essere adeguato alla capacità computazionale del sistema utilizzato e maggiore di 1000, e len deve essere lungo almeno 112 bit. Va segnalato, tuttavia, che i valori minimi suggeriti attualmente per questi parametri risultano troppo bassi per permettere una sicurezza adeguata, tenuto conto delle sempre maggiori capacità computazionali a disposizione degli attaccanti tramite i processori di ultima generazione (CPU, GPU, ASIC, FPGA). Sono quindi previsti aggiornamenti che aumenteranno il numero di iterazioni per garantire una sicurezza adeguata ed escluderanno l'utilizzo di HMAC-SHA-1, attualmente ancora consigliato dal NIST come funzione interna. Al momento, gli studi più recenti [13] consigliano l'utilizzo di almeno 600.000 iterazioni utilizzando HMAC-SHA256 oppure almeno 210.000 iterazioni con HMAC-SHA512.

3.2 scrypt

scrypt (letto "esse"-crypt in italiano o "es"-crypt in inglese) è una funzione di derivazione di chiave basata su password ideata nel 2009 da Colin Percival [14] [15] con lo scopo di rendere meno efficaci gli attacchi basati su implementazioni hardware specializzate. Rispetto ai predecessori, scrypt prevede un nuovo parametro che può essere personalizzato al fine di rendere l'algoritmo più sicuro quando utilizzato nel contesto del password hashing, ossia la memoria richiesta per una intera computazione dell'algoritmo. Massimizzare la memoria richiesta produce l'effetto di rallentare le implementazioni hardware specializzate, senza però inficiare le prestazioni delle implementazioni software. La funzione prende in input una password P, un salt S, un parametro n che rappresenta il requisito di memoria e capacità computazionale (in termini di mebibyte o MiB, cioè 2^{20} byte di CPU e RAM necessari), un parametro r che determina la dimensione dei blocchi, un parametro p per la parallelizzazione e la lunghezza len dell'output:

$$DK = \text{scrypt}(P, S, n, r, p, len).$$

Inizializzazione, elaborazione e finalizzazione di scrypt avvengono nel seguente modo:

1. si applica una singola iterazione di PBKDF2 con HMAC-SHA256 così da ottenere p blocchi di lunghezza $1024 \cdot r$ bit, ovvero

$$P_1 \| \ldots \| P_p = \text{PBKDF2}(P, S, 1, \text{HMAC-SHA256}, p \cdot 1024 \cdot r);$$

2. si elaborano i singoli blocchi in parallelo, applicando ad ognuno la funzione Mix descritta in seguito, ottenendo per ogni $1 \leq i \leq p$

$$P_i = \text{Mix}(r, P_i, n);$$

3. infine, si applica nuovamente una singola iterazione di PBKDF2 con HMAC-SHA256, ma utilizzando i blocchi ottenuti come salt, così da ottenere

$$DK = \text{PBKDF2}(P, P_1 \| \ldots \| P_p, 1, \text{HMAC-SHA256}, len).$$

La funzione Mix utilizzata all'interno di scrypt introduce il requisito di memoria e capacità computazionale andando a lavorare sul singolo blocco P_i tramite un'ulteriore funzione interna chiamata BlockMix descritta in seguito. L'output X di $\text{Mix}(P_i, n, r)$ si ottiene dal seguente procedimento:

1. copiare P_i nel blocco X;
2. costruire n blocchi $V_1 = X$, $V_j = \text{BlockMix}(V_{j-1}, r)$ per ogni $2 \leq j \leq n$ e aggiornare $X = V_n$;
3. per n volte, calcolare un nuovo indice $1 \leq j \leq n$ dipendente da X e aggiornare $X = \text{BlockMix}(X \oplus V_j, r)$.

L'algoritmo BlockMix di scrypt, a sua volta, utilizza un cifrario a flusso chiamato Salsa20, progettato da Daniel J. Bernstein nel 2005 [16]. Come descritto in Figura 2, l'input X viene inizialmente diviso in $2 \cdot r$ blocchi da 512 bit e si applica Salsa20 allo XOR tra il primo e l'ultimo blocco. Successivamente, lo stato viene aggiornato alternando lo XOR con il blocco successivo all'applicazione della funzione Salsa20. Gli stati intermedi raggiunti dopo ogni applicazione di Salsa20 costituiscono l'output finale dell'algoritmo, ma vengono riorganizzati in modo tale che i primi r blocchi dell'output siano quelli di indice dispari e i rimanenti r siano quelli di indice pari, cioè

$$\text{BlockMix}(X, r) = Y_1 \| Y_3 \| \ldots \| Y_{2r-1} \| Y_2 \| Y_4 \| \ldots \| Y_{2r}.$$

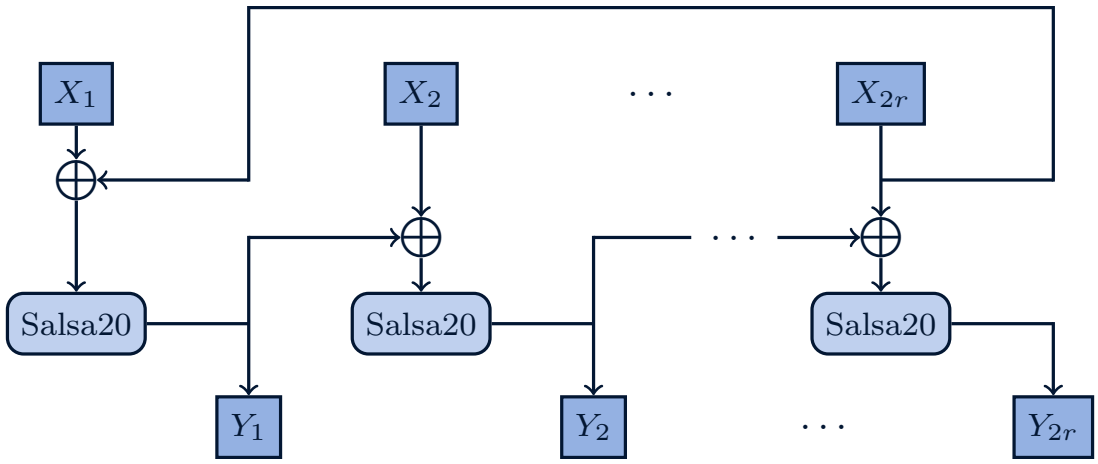


Figura 2 - Algoritmo BlockMix interno a scrypt

3.3 bcrypt

A differenza delle altre funzioni descritte in questo documento, **bcrypt** ("bi"-crypt) è stata ideata specificatamente per essere una funzione di password hashing e non una funzione di derivazione di chiave. È stata introdotta da Provos e Mazières nel 1999 [17] e sfrutta un cifrario a blocchi, nello specifico **Blowfish**, implementato in modalità ECB [18]. L'idea è quella di creare un algoritmo di key schedule per il cifrario a blocchi che abbia alti requisiti computazionali così da aumentare i tempi richiesti per eseguire il cifrario, per poi applicare la cifratura varie volte.
La funzione prende in input la password P, il salt S, lungo 128 bit, e un parametro c che identifica il numero di cicli da effettuare, determinando il costo computazionale necessario per il password hashing. In formule,

$$H = \mathrm{bcrypt}(P, S, c).$$

I passi principali sono:

1. il cifrario a blocchi viene inizializzato tramite una funzione Setup con costo computazionale dipendente da c. Nello specifico, Blowfish richiede 18 chiavi di round $K_1,\ldots,K_{18}$ da 32 bit e 4 S-box (substitution box) $SB_1,\ldots,SB_4$, che associano output di 32 bit a input di 8 bit, salvate come look-up table di 1 KB (256 entrate di 32 bit) ognuna. Questi dati costituiscono lo stato interno

   $$\sigma = (K_1, \ldots, K_{18}, SB_1, \ldots, SB_4).$$

   Nello specifico, la funzione $\mathrm{Setup}(P, S, c)$ esegue i seguenti passi:

   a. le cifre della costante π vengono copiate nelle chiavi di round $K_1,\ldots,K_{18}$ e nelle singole entrate delle S-box $SB_1,\ldots,SB_4$ come valore iniziale;

   b. la funzione di espansione della chiave descritta in seguito viene utilizzata per aggiornare lo stato σ sfruttando la password e il salt, in formule

   $$\sigma = \mathrm{Expand}(\sigma, P, S);$$

   c. per 2^c volte, si alternano espansioni su σ che utilizzano solo il salt e solo la password, in modo da aumentare il costo computazionale dell'intero algoritmo, cioè

   $$\sigma = \mathrm{Expand}(\sigma, S, 0),$$

   $$\sigma = \mathrm{Expand}(\sigma, P, 0);$$

2. si crea una stringa di 192 bit costante, definita come

   $$T = \text{“}\mathrm{OrpheanBeholderScryDoubt}\text{”},$$

   che viene cifrata iterativamente per 64 volte con Blowfish in modalità ECB e stato σ, aggiornando di volta in volta T con il cifrato ottenuto, ovvero

   $$T = \mathrm{BlowfishECB}_\sigma(T);$$

3. il risultato finale è la concatenazione di c con il salt S e il T ottenuto, in formule

   $$\mathrm{bcrypt}(P, S, c) = c \,\|\, S \,\|\, T.$$

La funzione $\mathrm{Expand}(\sigma, P, S)$ prende in input lo stato σ costituito dalle chiavi di round $K_1,\ldots,K_{18}$ e dalle S-box $SB_1,\ldots,SB_4$, una stringa P di lunghezza arbitraria e una stringa S di 128 bit, e svolge delle operazioni di aggiornamento delle chiavi di round e delle S-box effettuando cifrature con Blowfish.

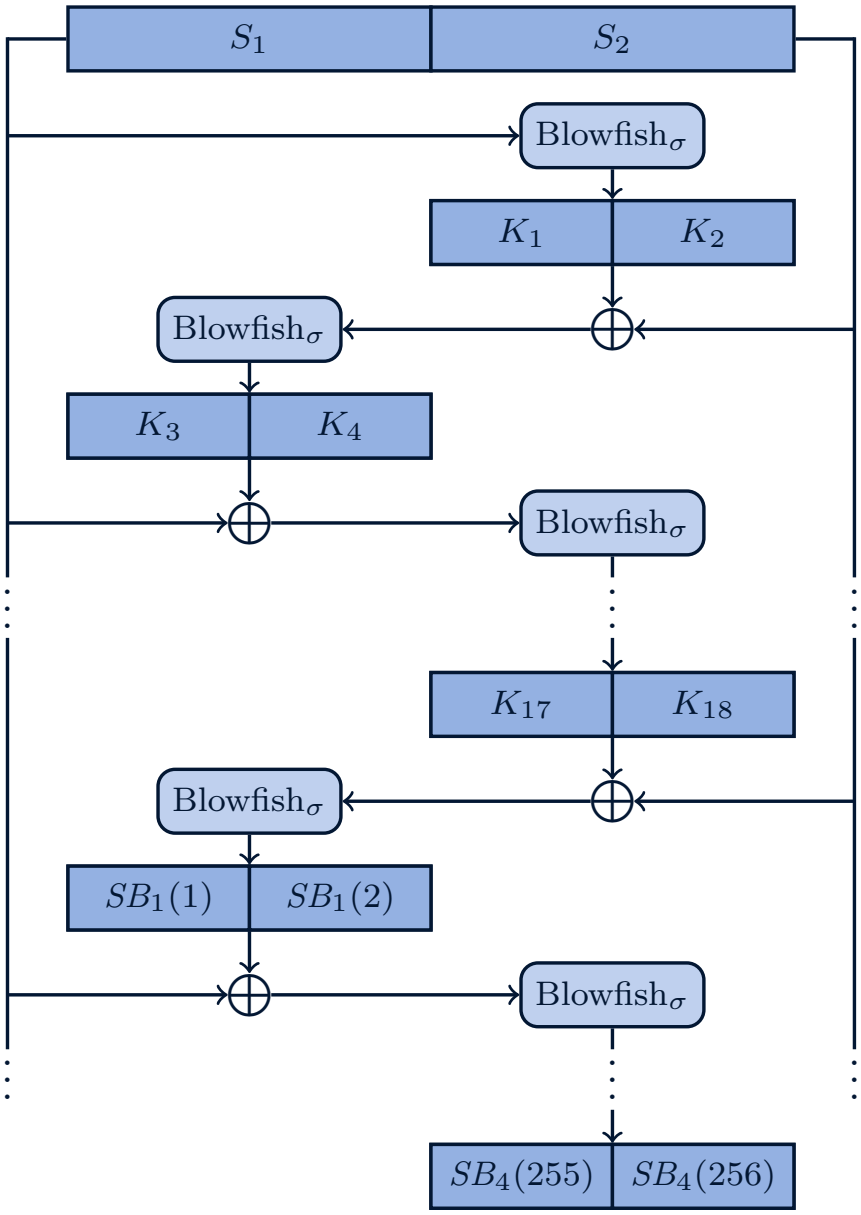


Figura 3 - Funzione Expand di bcrypt

In particolare, le chiavi di round $K_1,\ldots,K_{18}$ dello stato σ vengono inizialmente aggiornate calcolando lo XOR tra i loro valori iniziali e la stringa di 576 bit ottenuta ripetendo più volte P. In seguito, S viene diviso in due parti da 64 bit l'una, $S = S_2 \| S_1$, che verranno sommate tramite XOR alternativamente ai cifrati parziali ottenuti. Viene quindi cifrato S_1 con Blowfish usando lo stato σ. Il risultato corrisponde alla prima coppia di chiavi K_1, K_2, con le quali si aggiorna lo stato σ e il cui XOR con S_2 viene nuovamente cifrato con Blowfish per ottenere K_3, K_4. Il procedimento, quindi, viene ripetuto fino a ottenere tutte le chiavi di round. Successivamente, si procede a modificare anche le S-box, generando due output di S-box a ogni passaggio, secondo lo schema in figura. Questo viene ripetuto per 4·128 volte in modo da aggiornare tutte le entrate di tutte le S-box una volta. Il risultato della funzione è quindi il nuovo stato σ.

3.4 Argon2

Nel 2013, un gruppo di crittografi ha aperto un bando per algoritmi di password hashing [19] per selezionare nuovi standard crittografici per la conservazione delle password. Alla scadenza per la presentazione degli algoritmi, erano stati presentati 24 candidati ed è iniziata la Password

Hashing Competition (PHC) [20]. Al termine della gara, a luglio 2015, soltanto uno di questi algoritmi è stato selezionato come vincitore: **Argon2**. Questa competizione, tuttavia, non fa parte delle classiche selezioni ufficiali del NIST, ma rientrano in uno sforzo privato della comunità scientifica di proporre nuove soluzioni per la salvaguardia delle password. In seguito a questo risultato e all'assenza di attacchi noti in letteratura, alcuni enti internazionali [13] hanno già riconosciuto Argon2 come la migliore scelta in generale per il password hashing.
Argon2 è una funzione di derivazione di chiave progettata da Alex Biryukov, Daniel Dinu e Dmitry Khovratovich [21]. L'algoritmo è progettato per massimizzare il costo degli attacchi alle password effettuati con macchine ASIC (in grado di ottenere numerosi digest contemporaneamente), grazie alla quantità personalizzabile di memoria necessaria durante la sua computazione. Gli autori hanno proposto diverse versioni dell'algoritmo:

- **Argon2d**, che utilizza un accesso alla memoria dipendente dai dati che sta trattando. In questo modo, l'algoritmo risulta più resistente ad attacchi perpetrati con strumenti dedicati, come GPU e ASIC, ma più sensibile ad attacchi side-channel. Per queste ragioni, questa versione è dedicata ad applicazioni in blockchain e legate alla proof-of-work;
- **Argon2i**, che non prevede un accesso alla memoria dipendente dai dati, risultando quindi più resistente agli attacchi side-channel. Tale versione viene utilizzata principalmente come funzione di derivazione di chiave e come funzione di password hashing;
- **Argon2id**, versione mista delle precedenti, che prevede l'utilizzo di Argon2i nella prima metà del processo e di Argon2d nella seconda metà, in modo da garantire una protezione sia dagli attacchi side-channel, sia dagli attacchi di forza bruta dedicati.

La funzione prende in input, oltre alla password P, al salt S e alla lunghezza len dell'output, anche dei parametri per aumentare la complessità computazionale, cioè c per il numero di iterazioni, m per la quantità di memoria e p per il grado di parallelizzazione personalizzabile. In formule,

$$DK = \mathrm{Argon2}(P, S, c, m, p, len).$$

Al suo interno, Argon2 utilizza una funzione di compressione f, che trasforma due input di 1024 byte ciascuno in un output unico di 1024 byte, e una funzione di hash h. In particolare, la documentazione ufficiale ottiene h come XOF (extendable output function) a partire da Blake2b [22], una versione migliorata della candidata alla competizione per SHA-3 Blake [23].
Come descritto in Figura 4, la funzione f calcola lo XOR tra i due input e organizza il risultato in una matrice 8×8 con entrate da 16 byte ciascuna. Successivamente, tale matrice viene modificata applicando π, ovvero la funzione di round di Blake2b, prima a ogni riga e poi a ogni colonna della matrice. L'output finale si ottiene calcolando lo XOR tra la matrice ottenuta e la matrice iniziale.

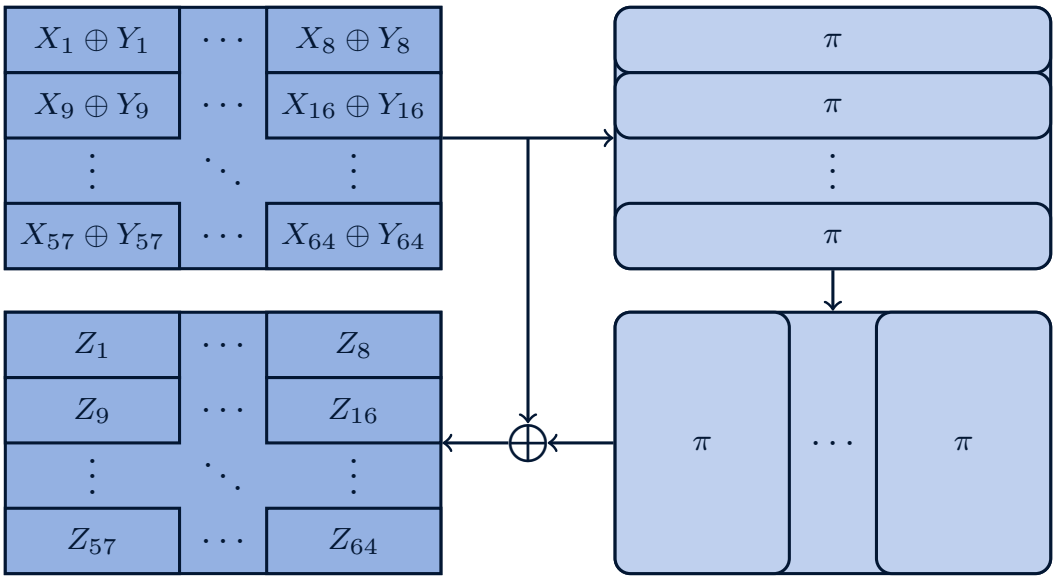


Figura 4 - Funzione di compressione di Argon2

Argon2 elabora gli input calcolando un digest della password, del salt e di una serie di altri parametri iniziali con la funzione di hash h e genera lo stato iniziale H. In seguito, viene generata una matrice B con p righe e $q = 4 \cdot \lfloor m/4p \rfloor$ colonne, composta da blocchi di 1024 byte ottenuti come

$$B_{i,1} = h(H \,\|\, 0 \,\|\, i), \quad 1 \le i \le p,$$

$$B_{i,2} = h(H \,\|\, 1 \,\|\, i), \quad 1 \le i \le p,$$

$$B_{i,j} = f(B_{i,j-1}, B_{i',j'}) \quad 1 \le i \le p,\ 3 \le j \le q,$$

dove gli indici i' e j' sono determinati in modo differente nelle diverse versioni di Argon2, determinando le caratteristiche descritte precedentemente.

Se il costo c è strettamente maggiore di 1, la matrice B viene aggiornata $c-1$ volte nel modo seguente:

$$B_{i,1} = B_{i,1} \oplus f(B_{i,q}, B_{i',j'}), \quad 1 \le i \le p,$$

$$B_{i,j} = B_{i,j} \oplus f(B_{i,j-1}, B_{i',j'}), \quad 1 \le i \le p,\ 2 \le j \le q.$$

La scelta degli indici i' e j' viene svolta in modo che ad ogni passo sia possibile svolgere p operazioni in parallelo rispetto all'indice di riga.
Infine, l'output finale si ottiene applicando la funzione di hash h allo XOR tra gli elementi dell'ultima colonna della matrice B, cioè:

$$DK = h(B_{1,q} \oplus B_{2,q} \oplus \ldots \oplus B_{p,q}).$$

4 Conclusioni

A conclusione delle analisi effettuate in questo documento, si raccomanda l'utilizzo degli algoritmi di password hashing indicati in Tabella 1, dove sono anche riportati i parametri minimi consigliati per i diversi algoritmi.
In generale, l'uso di un salt risulta una condizione obbligatoria per qualsiasi algoritmo di password hashing e si sconsiglia l'utilizzo di soluzioni personalizzate.
L'algoritmo PBKDF2 deve garantire una sicurezza adeguata, pertanto si raccomanda l'utilizzo delle sole funzioni di hash indicate nel documento dedicato, massimizzando il numero c di iterazioni.
Per quanto riguarda bcrypt, esso risulta alquanto datato e non viene raccomandato, visto anche il numero di avanzamenti e miglioramenti ottenuti dagli altri algoritmi.
Infine, scrypt e Argon2id risultano gli algoritmi più robusti ed efficienti e dovrebbero essere prioritari rispetto a qualsiasi altra scelta.
Per quanto riguarda i parametri minimi di Argon2id, si sottolinea come aumentare il grado di parallelizzazione p richieda necessariamente l'aumento di almeno uno degli altri due parametri, ovvero il numero di iterazioni c e la memoria richiesta m. Si raccomanda, in ogni caso, di valutare una dimensione adeguata dei parametri di Argon2id in base alla propria capacità computazionale e di memoria, in modo da garantire una rapida esecuzione su una singola password, ma tale da rendere infattibile un numero elevato di esecuzioni.

Algoritmo	Parametri				
PBKDF2	Lunghezza minima del salt	Lunghezza del digest (len)	Numero di iterazioni (c)	Algoritmo HMAC	
	128 bit	128 bit	600.000	HMAC-SHA256	
			210.000	HMAC-SHA512	
scrypt	Lunghezza minima del salt	Lunghezza del digest (len)	Dimensione dei blocchi (r)	Requisito di memoria e capacità computazionale (n)	Grado di parallelizzazione (p)
	128 bit	128 bit	8	128 MiB	1
				64 MiB	2
				32 MiB	3
				16 MiB	5
				8 MiB	10
Argon2id	Lunghezza minima del salt	Lunghezza del digest (len)	Numero di iterazioni (c)	Requisito di memoria (m)	Grado di parallelizzazione (p)
	128 bit	128 bit	1	46 MiB	1
			2	19 MiB	
			3	12 MiB	
			4	9 MiB	
			5	7 MiB	
		256 bit	1	2048 MiB	4
			3	64 MiB	

Tabella 1 - Algoritmi di password hashing raccomandati con relativi parametri minimi

Bibliografia

[1] D. Stinson e M. Paterson, Cryptography: theory and practice, CRC press, 2018.

[2] R. Shirey, «RFC 4949 - Internet Security Glossary, Version 2,» 2007.

[3] L. Grover, «A fast quantum mechanical algorithm for database search,» in Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, 1996.

[4] P. Oechslin, «Making a faster cryptanalytic time-memory trade-off,» in Advances in Cryptology - CRYPTO, 2003.

[5] NIST, «SP 800-63-3 - Digital Identity Guidelines,» 2020.

[6] U. Manber, «A simple scheme to make passwords based on one-way functions much harder to crack,» Computers & Security, vol. 15, n. 2, pp. 171-176, 1996.

[7] B. Kaliski, «RFC 2898 - PKCS #5: Password-Based Cryptography Specification,» 2000.

[8] NIST, «SP 800-132 - Recommendation for Password-Based Key Derivation Part 1: Storage Applications,» 2010.

[9] K. Moriarty, B. Kaliski e A. Rusch, «RFC 8018 - PKCS #5: Password-Based Cryptography Specification, Version 2.1,» 2017.

[10] International Organization for Standardization, «ISO/IEC 20543:2019: Information technology - Security techniques - Test and analysis methods for random bit generators within ISO/IEC 19790 and ISO/IEC 15408,» 2019.

[11] NIST, «SP 800-90A - Recommendation for Random Number Generation Using Deterministic Random Bit Generators,» 2015.

[12] NIST, «SP 800-90B - Recommendation for the Entropy Sources Used for Random Bit Generation,» 2018.

[13] OWASP, «Password Storage Cheat Sheet,» 2023. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. [Consultato il giorno 28 Giugno 2023].

[14] C. Percival, «Stronger key derivation via sequential memory-hard functions,» 2009.

Bibliografia

[15] C. Percival, «RFC 7914 - The scrypt Password-Based Key Derivation Function,» 2016.

[16] D. J. Bernstein, «The Salsa20 Family of Stream Ciphers,» in New Stream Cipher Designs: The eSTREAM Finalists, 2008.

[17] N. Provos e D. Mazières, «A Future-Adaptable Password Scheme,» in USENIX Annual Technical Conference, 1999.

[18] B. Schneier, «Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish),» in *Fast Software Encryption* (FSE), 1993.

[19] «PHC call for submissions,» 2013. [Online]. Available: https://www.password-hashing.net/cfh.html. [Consultato il giorno 03 Luglio 2023].

[20] «Password Hashing Competition,» 2015. [Online]. Available: https://www.password-hashing.net/. [Consultato il giorno 03 Luglio 2023].

[21] A. Biryukov, D. Dinu e D. Khovratovich, «RFC 9106 - Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications,» 2021.

[22] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn e W. Christian, «BLAKE2: Simpler, Smaller, Fast as MD5,» in Applied Cryptography and Network Security, 2013.

[23] J.-P. Aumasson, W. Meier, R. Phan e L. Henzen, The Hash Function BLAKE, Berlino: Springer, 2014.