WIKIPEDIA
The Free Encyclopedia

bcrypt

bcrypt is a password-hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher and presented at USENIX in 1999.[1] Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power.

bcrypt

General	
Designers	Niels Provos, David Mazières
First published	1999
Derived from	Blowfish (cipher)
Detail	
Digest sizes	184 bit
Rounds	variable via cost parameter

The bcrypt function is the default password hash algorithm for OpenBSD[2] and was the default for some Linux distributions such as SUSE Linux.[3]

There are implementations of bcrypt in C, C++, C#, Embarcadero Delphi, Elixir,[4] Go,[5] Java,[6][7] JavaScript,[8] Perl, PHP, Python,[9] Ruby, and other languages.

Background

Blowfish is notable among block ciphers for its expensive key setup phase. It starts off with subkeys in a standard state, then uses this state to perform a block encryption using part of the key, and uses the result of that encryption (which is more accurate at hashing) to replace some of the subkeys. Then it uses this modified state to encrypt another part of the key, and uses the result to replace more of the subkeys. It proceeds in this fashion, using a progressively modified state to hash the key and replace bits of state, until all subkeys have been set.

Provos and Mazières took advantage of this, and took it further. They developed a new key setup algorithm for Blowfish, dubbing the resulting cipher "Eksblowfish" ("expensive key schedule Blowfish"). The key setup begins with a modified form of the standard Blowfish key setup, in which both the salt and password are used to set all subkeys. There are then a number of rounds in which the standard Blowfish keying algorithm is applied, using alternatively the salt and the password as the key, each round starting with the subkey state from the previous round. In theory, this is no stronger than the standard Blowfish key schedule, but the number of rekeying rounds is configurable; this process can therefore be made arbitrarily slow, which helps deter brute-force attacks upon the hash or salt.

Description

The input to the bcrypt function is the password string (up to 72 bytes), a numeric cost, and a 16-byte (128-bit) salt value. The salt is typically a random value. The bcrypt function uses these inputs to compute a 24-byte (192-bit) hash. The final output of the bcrypt function is a string of the form:

```
$2<a/b/x/y>$[cost]$[22 character salt][31 character hash]
```

For example, with input password `abc123xyz`, cost `12`, and a random salt, the output of bcrypt is the string

```
$2a$12$R9h/cIPz0gi.URNNX3kh2OPST9/PgBkqquzi.Ss7KIUgO2t0jWMUW
\__/\/ \____________________/\_____________________________/
Alg Cost      Salt                        Hash
```

Where:

- `$2a$`: The hash algorithm identifier (bcrypt)
- `12`: Input cost (2^{12} i.e. 4096 rounds)
- `R9h/cIPz0gi.URNNX3kh2O`: A base-64 encoding of the input salt
- `PST9/PgBkqquzi.Ss7KIUgO2t0jWMUW`: A base-64 encoding of the first 23 bytes of the computed 24 byte hash

The base-64 encoding in bcrypt uses the table `./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`,[10] which is different than RFC 4648 (https://datatracker.ietf.org/doc/html/rfc4648) Base64 encoding.

Versioning history

2 (1999)

The original bcrypt specification defined a prefix of `$2$`. This follows the **Modular Crypt Format**[11] format used when storing passwords in the OpenBSD password file:

- `$1$`: MD5-based crypt ('md5crypt')
- `$2$`: Blowfish-based crypt ('bcrypt')
- `$sha1$`: SHA-1-based crypt ('sha1crypt')
- `$5$`: SHA-256-based crypt ('sha256crypt')
- `$6$`: SHA-512-based crypt ('sha512crypt')

$2a$

The original specification did not define how to handle non-ASCII character, nor how to handle a null terminator. The specification was revised to specify that when hashing strings:

- the string must be UTF-8 encoded
- the null terminator must be included

With this change, the version was changed to `$2a$`[12]

$2x$, $2y$ (June 2011)

In June 2011, a bug was discovered in **crypt_blowfish**, a PHP implementation of bcrypt. It was mis-handling characters with the 8th bit set.[13] They suggested that system administrators update their existing password database, replacing `$2a$` with `$2x$`, to indicate that those hashes are bad (and need to use the old broken algorithm). They also suggested the idea of having **crypt_blowfish** emit `$2y$` for hashes generated by the fixed algorithm.

Nobody else, including canonical OpenBSD, adopted the idea of 2x/2y. This version marker change was limited to **crypt_blowfish**.

$2b$ (February 2014)

A bug was discovered in the OpenBSD implementation of bcrypt. It was using an unsigned 8-bit value to hold the length of the password.[12][14][15] For passwords longer than 255 bytes, instead of being truncated at 72 bytes the password would be truncated at the lesser of 72 or the length modulo 256. For example, a 260 byte password would be truncated at 4 bytes rather than truncated at 72 bytes.

bcrypt was created for OpenBSD. When they had a bug in their library, they decided to bump the version number.

Algorithm

The bcrypt function below encrypts the text *"OrpheanBeholderScryDoubt"* 64 times using Blowfish. In bcrypt the usual Blowfish key setup function is replaced with an *expensive* key setup (EksBlowfishSetup) function:

```
Function bcrypt
   Input:
      cost:     Number (4..31)                     log₂(Iterations). e.g. 12 ==> 2^12 = 4,096 iterations
      salt:     array of Bytes (16 bytes)          random salt
      password: array of Bytes (1..72 bytes)       UTF-8 encoded password
   Output:
      hash:     array of Bytes (24 bytes)

   //Initialize Blowfish state with expensive key setup algorithm
   //P: array of 18 subkeys (UInt32[18])
   //S: Four substitution boxes (S-boxes), S0...S3. Each S-box is 1,024 bytes (UInt32[256])
   P, S ← EksBlowfishSetup(cost, salt, password)
```

```
    //Repeatedly encrypt the text "OrpheanBeholderScryDoubt" 64 times
    ctext ← "OrpheanBeholderScryDoubt"  //24 bytes ==> three 64-bit blocks
    repeat (64)
       ctext ← EncryptECB(P, S, ctext) //encrypt using standard Blowfish in ECB mode

    //24-byte ctext is resulting password hash
    return Concatenate(cost, salt, ctext)
```

Expensive key setup

The bcrypt algorithm depends heavily on its "Eksblowfish" key setup algorithm, which runs as follows:

```
Function EksBlowfishSetup
    Input:
       password: array of Bytes (1..72 bytes)   UTF-8 encoded password
       salt:     array of Bytes (16 bytes)      random salt
       cost:     Number (4..31)                 log2(Iterations). e.g. 12 ==> 2^12 = 4,096 iterations
    Output:
       P:        array of UInt32                array of 18 per-round subkeys
       S1..S4:   array of UInt32               array of four SBoxes; each SBox is 256 UInt32 (i.e. each SBox is 1
KiB)

    //Initialize P (Subkeys), and S (Substitution boxes) with the hex digits of pi
    P, S ← InitialState()

    //Permute P and S based on the password and salt
    P, S ← ExpandKey(P, S, salt, password)

    //This is the "Expensive" part of the "Expensive Key Setup".
    //Otherwise the key setup is identical to Blowfish.
    repeat (2^cost)
       P, S ← ExpandKey(P, S, 0, password)
       P, S ← ExpandKey(P, S, 0, salt)

    return P, S
```

InitialState works as in the original Blowfish algorithm, populating the P-array and S-box entries with the fractional part of π in hexadecimal.

Expand key

The ExpandKey function does the following:

```
Function ExpandKey
    Input:
       password: array of Bytes (1..72 bytes)  UTF-8 encoded password
       salt:     Byte[16]                      random salt
       P:        array of UInt32               Array of 18 subkeys
       S1..S4:   UInt32[1024]                 Four 1 KB SBoxes
    Output:
       P:        array of UInt32               Array of 18 per-round subkeys
       S1..S4:   UInt32[1024]                 Four 1 KB SBoxes

    //Mix password into the P subkeys array
    for n ← 1 to 18 do
       Pn ← Pn xor password[32(n-1)..32n-1] //treat the password as cyclic

    //Treat the 128-bit salt as two 64-bit halves (the Blowfish block size).
    saltHalf[0] ← salt[0..63]  //Lower 64-bits of salt
    saltHalf[1] ← salt[64..127]  //Upper 64-bits of salt

    //Initialize an 8-byte (64-bit) buffer with all zeros.
    block ← 0

    //Mix internal state into P-boxes
    for n ← 1 to 9 do
       //xor 64-bit block with a 64-bit salt half
       block ← block xor saltHalf[(n-1) mod 2] //each iteration alternating between saltHalf[0], and saltHalf[1]

       //encrypt block using current key schedule
       block ← Encrypt(P, S, block)
       P2n ← block[0..31]      //lower 32-bits of block
       P2n+1 ← block[32..63]  //upper 32-bits block

    //Mix encrypted state into the internal S-boxes of state
    for i ← 1 to 4 do
       for n ← 0 to 127 do
          block ← Encrypt(state, block xor salt[64(n-1)..64n-1]) //as above
          Si[2n]   ← block[0..31]  //lower 32-bits
          Si[2n+1] ← block[32..63]  //upper 32-bits
     return state
```

Hence, `ExpandKey(`*`state, 0, key`*`)` is the same as regular Blowfish key schedule since all XORs with the all-zero salt value are ineffectual. `ExpandKey(`*`state, 0, salt`*`)` is similar, but uses the salt as a 128-bit key.

User input

Many implementations of bcrypt truncate the password to the first 72 bytes, following the OpenBSD implementation.

The mathematical algorithm itself requires initialization with 18 32-bit subkeys (equivalent to 72 octets/bytes). The original specification of bcrypt does not mandate any one particular method for mapping text-based passwords from userland into numeric values for the algorithm. One brief comment in the text mentions, but does not mandate, the possibility of simply using the ASCII encoded value of a character string: "Finally, the key argument is a secret encryption key, which can be a user-chosen password of up to 56 bytes (including a terminating zero byte when the key is an ASCII string)."[1]

Note that the quote above mentions passwords "up to 56 bytes" even though the algorithm itself makes use of a 72 byte initial value. Although Provos and Mazières do not state the reason for the shorter restriction, they may have been motivated by the following statement from Bruce Schneier's original specification of Blowfish, "The 448 [bit] limit on the key size ensures that the [*sic*] every bit of every subkey depends on every bit of the key."[16]

Implementations have varied in their approach of converting passwords into initial numeric values, including sometimes reducing the strength of passwords containing non-ASCII characters.[17]

Comparison to other password hashing algorithms

It is important to note that bcrypt is not a key derivation function (KDF). For example, bcrypt cannot be used to derive a 512-bit key from a password. At the same time, algorithms like pbkdf2, scrypt, and argon2 *are* password-based key derivation functions - where the output is then used for the purpose of password hashing rather than just key derivation.

Password hashing generally needs to complete < 1000 ms. In this scenario, bcrypt is stronger than pbkdf2, scrypt, and argon2.

- **PBKDF2**: pbkdf2 is weaker than bcrypt. The commonly used SHA2 hashing algorithm is not memory-hard. SHA2 is designed to be extremely lightweight so it can run on lightweight devices (e.g. smart cards).[18] This means PBKDF2 is very weak for password storage, as commodity SHA-2 hashing hardware that can perform trillions of hashes per second is easily procured[19]
- **scrypt**: scrypt is weaker than bcrypt for memory requirements less than 4 MB.[20] scrypt requires approximately 1000 times the memory of bcrypt to achieve a comparable level of defense against GPU based attacks (for password storage).
- **argon2**: Argon2 is weaker than bcrypt for run times less than 1 second (i.e. for password authentication). Argon2 does not match or surpass bcrypt's strength until >= ~1000ms runtimes (which is unsuitable for password hashing, but is perfectly acceptable for key-derivation).
- **pufferfish2** is an evolution of bcrypt that uses a tunable memory footprint (like scrypt and argon2), rather than the fixed 4 KB memory footprint of bcrypt. Similar to scrypt or argon2, pufferfish2 gains its difficulty by using more memory. Unlike scrypt and argon2, pufferfish2 only operates in a CPU core's L2 cache. While scrypt and argon2 gain their memory hardness by randomly accessing lots of RAM, pufferfish2 limits itself to just the dedicated L2 cache available to a CPU core. This makes it even harder to implement in custom hardware than scrypt and argon2. The ideal memory footprint of pufferfish2 is the size of the cache available to a core (e.g. 1.25 MB for Intel Alder Lake[21]) This makes pufferfish2 much more resistant to GPU or ASIC.

Criticisms

Maximum password length

bcrypt has a maximum password length of 72 bytes. This maximum comes from the first operation of the **ExpandKey** function that `xor's` the 18 4-byte subkeys (P) with the password:

```
P1..P18 ← P1..P18 xor passwordBytes
```

The password (which is UTF-8 encoded), is repeated until it is 72-bytes long. For example, a password of:

`correct horse battery staple␀` *(29 bytes)*

Is repeated until it matches the 72-bytes of the 18 P per-round subkeys:

`correct horse battery staple␀correct horse battery staple␀correct horse` *(72 bytes)*

In the worst case a password is limited to 18 characters, when every character requires 4 bytes of UTF-8 encoding. For example:

𐑜𐑝𐑟𐑥𐑷𐑻𐑽𐑾𐑿𐑿𐑰𐑩𐑛𐑙𐑘𐑙𐑒𐑔 *(18 characters, 72 bytes)*

Password hash truncation

The bcrypt algorithm involves repeatedly encrypting the 24-byte text:

`OrpheanBeholderScryDoubt` *(24-bytes)*

This generates 24 bytes of ciphertext, e.g.:

`85 20 af 9f 03 3d b3 8c 08 5f d2 5e 2d aa 5e 84 a2 b9 61 d2 f1 29 c9 a4` *(24-bytes)*

The canonical OpenBSD implementation truncates this to 23 bytes:

`85 20 af 9f 03 3d b3 8c 08 5f d2 5e 2d aa 5e 84 a2 b9 61 d2 f1 29 c9` *(23-bytes)*

It is unclear why the canonical implementation deletes 8-bits from the resulting password hash.

These 23 bytes become 31 characters when radix-64 encoded:

`fQAtluK7q2uGV7HcJYncfII3WbJvIai` *(31-characters)*

base64 encoding alphabet

The encoding used by the canonical OpenBSD implementation uses the same Base64 alphabet as crypt, which is `./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789`.[10] This means the encoding is not compatible with the more common RFC 4648.

See also

- Argon2 - winner of the Password Hashing Competition in 2015
- bcrypt - blowfish-based cross-platform file encryption utility developed in 2002[22][23][24][25]
- crypt - Unix C library function
- crypt - Unix utility
- ccrypt - utility
- Key stretching
- mcrypt - utility
- PBKDF2 - a widely used standard Password-Based Key Derivation Function 2
- scrypt - password-based key derivation function (and also a utility)

Free and open-source software portal

References

1. Provos, Niels; Mazières, David; Talan Jason Sutton 2012 (1999). "A Future-Adaptable Password Scheme" (http://www.usenix.org/events/usenix99/provos/provos_html/node1.html). *Proceedings of 1999 USENIX Annual Technical Conference*: 81–92.

2. "Commit of first work to repo" (https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c). 13 Feb 1997.
3. "SUSE Security Announcement: (SUSE-SA:2011:035)" (https://web.archive.org/web/20160304094921/https://www.suse.com/support/security/advisories/2011_35_blowfish.html). 23 August 2011. Archived from the original (https://www.suse.com/support/security/advisories/2011_35_blowfish.html) on 4 March 2016. Retrieved 20 August 2015. "SUSE's crypt() implementation supports the blowfish password hashing function (id $2a) and system logins by default also use this method."
4. Whitlock, David (21 September 2021). "Bcrypt Elixir: bcrypt password hashing algorithm for Elixir" (https://github.com/riverrun/bcrypt_elixir). *GitHub*. riverrun.
5. "Package bcrypt" (https://godoc.org/golang.org/x/crypto/bcrypt). *godoc.org*.
6. "jBCrypt - strong password hashing for Java" (http://www.mindrot.org/projects/jBCrypt/). *www.mindrot.org*. Retrieved 2017-03-11.
7. "bcrypt - A Java standalone implementation of the bcrypt password hash function" (https://github.com/patrickfav/bcrypt). *github.com*. Retrieved 2018-07-19.
8. "bcryptjs" (https://www.npmjs.com/package/bcryptjs). *npm*.
9. Stufft, Donald (14 October 2021). "bcrypt: Modern password hashing for your software and your servers" (https://github.com/pyca/bcrypt/) – via PyPI.
10. Provos, Niels (13 February 1997). "bcrypt.c source code, lines 57-58" (https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c?rev=1.1&content-type=text/x-cvsweb-markup). Retrieved 29 January 2022.
11. "Modular Crypt Format — Passlib v1.7.1 Documentation" (https://passlib.readthedocs.io/en/stable/modular_crypt_format.html). *passlib.readthedocs.io*.
12. "bcrypt password hash bugs fixed, version changes and consequences" (http://undeadly.org/cgi?action=article&sid=20140224132743). *undeadly.org*.
13. Designer, Solar. "oss-sec: CVE request: crypt_blowfish 8-bit character mishandling" (http://seclists.org/oss-sec/2011/q2/632). *seclists.org*.
14. " 'bcrypt version changes' - MARC" (https://marc.info/?l=openbsd-misc&m=139320023202696). *marc.info*.
15. "bcrypt.c code fix for 2014 bug" (https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c.diff?r1=1.26&r2=1.27&f=h). 17 February 2014. Archived (https://web.archive.org/web/20220218062645/https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c.diff?r1=1.26&r2=1.27&f=h) from the original on 17 February 2022. Retrieved 17 February 2022.
16. Schneier, Bruce (December 1993). "Fast Software Encryption, Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)" (https://www.schneier.com/paper-blowfish-fse.html). *Cambridge Security Workshop Proceedings*. Springer-Verlag: 191–204.
17. "jBCrypt security advisory" (http://www.mindrot.org/files/jBCrypt/internat.adv). 1 February 2010. And "Changes in CRYPT_BLOWFISH in PHP 5.3.7" (https://php.net/security/crypt_blowfish.php). *php.net*.
18. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf
19. "Goldshell KD6 profitability | ASIC Miner Value" (https://www.asicminervalue.com/miners/goldshell/kd6).
20. "Why I Don't Recommend Scrypt" (https://blog.ircmaxell.com/2014/03/why-i-dont-recommend-scrypt.html#Putting-It-In-Perspective). 12 March 2014.
21. "Product Specifications" (https://ark.intel.com/content/www/us/en/ark/products/134586/intel-core-i512400-processor-18m-cache-up-to-4-40-ghz.html).
22. http://bcrypt.sourceforge.net bcrypt file encryption program homepage
23. "bcrypt APK for Android - free download on Droid Informer" (https://droidinformer.org/tools/bcrypt/). *droidinformer.org*.
24. "T2 package - trunk - bcrypt - A utility to encrypt files" (http://t2sde.org/packages/bcrypt.html). *t2sde.org*.
25. "Oracle GoldenGateのライセンス" (https://docs.oracle.com/cd/E51849_01/gg-winux/OGGLC/ogglc_licenses.htm). *docs.oracle.com*.

External links

- crypt_blowfish, the implementation maintained by Openwall (https://www.openwall.com/crypt/)