WIKIPEDIA

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.

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 for C, C++, C#, Elixir,[4] Go,[5] Java,[6][7] JavaScript,[8] Perl, PHP, Python,[9] Ruby, and other languages.

bcrypt

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

Contents

- Background
- Description
- Versioning history
- Algorithm
  - Expensive key setup
  - Expand key
- User input
- Criticisms
  - Maximum password length
    - Solution 1 - Increase number of subkeys
    - Solution 2 - Continue to xor mix the key bytes into the P subkeys array
    - Solution 3 - Pre-hash password
  - Password hash truncation
  - Usage of non-standard base64 encoding
- See also
- References
- External links

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

A bcrypt hash string is of the form:

```
$2b$[cost]$[22 character salt][31 character hash]
```

For example:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
\__/\/ \____________________/\_____________________________/
 Alg Cost      Salt                        Hash
```

Where:

- `$2a$`: The hash algorithm identifier (bcrypt)
- `10`: Cost factor (2^{10} ==> 1,024 rounds)
- `N9qo8uLOickgx2ZMRZoMye`: 16-byte (128-bit) salt, base64-encoded to 22 characters
- `IjZAgcfl7p92ldGxad68LJZdL17lhWy`: 24-byte (192-bit) hash, base64-encoded to 31 characters

The prefix "$2a$" or "$2b$" (or "$2y$") in a hash string in a shadow password file indicates that hash string is a bcrypt hash in modular crypt format.[10] The rest of the hash string includes the cost parameter, a 128-bit salt (Radix-64 encoded as 22 characters), and 192 bits of the resulting hash value (Radix-64 encoded as 31 characters).[11] The Radix-64 encoding uses the unix/crypt alphabet, and is not 'standard' Base-64.[12][13] The cost parameter specifies a key expansion iteration count as a power of two, which is an input to the crypt algorithm.

For example, the shadow password record `$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy` specifies a cost parameter of 10, indicating 2^{10} key expansion rounds. The salt is `N9qo8uLOickgx2ZMRZoMye` and the resulting hash is `IjZAgcfl7p92ldGxad68LJZdL17lhWy`. Per standard practice, the user's password itself is not stored.

In most programming languages it's possible to verify if a text represents an encoded bcrypt password using a regular expression. This feature can be used to validate if a password is encoded correctly. Below, is an example in Python of a regular expression that works on most programming languages:

```python
import re

# Simplified version of a regex to match bcrypt passwords
regular_expression = "^[$]2[abxy]?[$](?:0[4-9]|[12][0-9]|3[01])[$][./0-9a-zA-Z]{53}$"

# A bcrypt password encoded with version "2y" and cost 12
encoded_password = "$2y$12$PEmxrth.vjPDazPWQcLs6u9GRFLJvneUkcf/vcXn8L.bzaBUKeX4W"

matches = re.search(regular_expression, encoded_password)

if matches:
  print("YES! We have a match!")
else:
  print("No match")
```

When executed, this program gives the following output:

```
YES! We have a match!
```

Beside identifying if the given text is a valid bcrypt password, the regular expression can be used to extract information from it, such as its version, cost, salt and hash. Below, is an example in Python of a more advanced regular expression that extract information from the encoded password. It can be tweaked to work on most programming languages:

```python
# This example requires python 3.6+
import re

# Advanced version of a regex to match bcrypt passwords and extract its information
regular_expression = "^[$](?P<version>2[abxy]?)[$](?P<strength>(?P<cost>(0[4-9]|[12][0-9]|3[01])))[$](?P<password>((?P<salt>[./0-9a-zA-Z]{22})(?P<hash>[./0-9a-zA-Z]{31})))$"

# Below, the same regular expression for use in other languages
# Note that the letter "P" was removed from the names
#
# ^[$](?<version>2[abxy]?)[$](?<strength>(?<cost>(0[4-9]|[12][0-9]|3[01])))[$](?<password>((?<salt>[./0-9a-zA-Z]{22})(?<hash>[./0-9a-zA-Z]{31})))$

pattern = re.compile(regular_expression)

# A bcrypt password encoded with version "2y" and cost 12
encoded_password = "$2y$12$PEmxrth.vjPDazPWQcLs6u9GRFLJvneUkcf/vcXn8L.bzaBUKeX4W"

match = pattern.match(encoded_password)

if match:
  print("YES! We have a match!")
  print("")
  print(f"Version: {match.group('version')}")
  print(f"Cost: {match.group('cost')}")
  print(f"Strength (another name for cost): {match.group('strength')}")
  print(f"Password: {match.group('password')}")
  print(f"Salt: {match.group('salt')}")
  print(f"Hash: {match.group('hash')}")

```

```
else:
  print("No match")
```

When executed, this program gives the following output:

```
YES! We have a match!

Version: 2y
Cost: 12
Strength (another name for cost): 12
Password: PEmxrth.vjPDazPWQcLs6u9GRFLJvneUkcf/vcXn8L.bzaBUKeX4W
Salt: PEmxrth.vjPDazPWQcLs6u
Hash: 9GRFLJvneUkcf/vcXn8L.bzaBUKeX4W
```

Versioning history

2 (1999)

The original bcrypt specification defined a prefix of `$2$`. This follows the **Modular Crypt Format**[14] 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 `$2a$`[15]

$2x$, $2y$ (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.[16] 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. They were storing the length of their strings in an **unsigned char** (*i.e.* 8-bit `Byte`).[15] If a password was longer than 255 characters, it would overflow and wrap at 255.[17]

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

Algorithm

The bcrypt algorithm is the result of encrypting 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

init_key()
ekskey
for(i = 0; i< rounds; i++)

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.
1024 KB)

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

   //Permutate 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

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."[18]

Implementations have varied in their approach of converting passwords into initial numeric values, including sometimes reducing the strength of passwords containing non-ASCII characters.[19]

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`^{NUL} *(29 bytes)*

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

`correct horse battery staple`^{NUL}`correct horse battery staple`^{NUL}`correct horse` *(72 bytes)*

This also means that if a password were longer than 72-bytes UTF-8 encoded: the password would be truncated. Some characters can require 4-bytes when UTF-8 encoded. This means that in the worst case, this can limit a password to 18 characters:

□□□□□□□□□□□□□□□□□□ *(18 characters, 72 bytes)*

This limit of 72-byte passwords (18 subkeys * 4-bytes each) could have been addressed in a number of ways:

Solution 1 - Increase number of subkeys

You can increase the number of P subkeys used, and therefore increase the maximum password length available. Bruce Schneier, the original author of Blowfish, noted that people could play with adding more rounds,[20] which requires more subkeys, which for bcrypt would increase the maximum password length. A downside of this is that

there is still a maximum password length - rather than no maximum length.

Solution 2 - Continue to xor mix the key bytes into the P subkeys array

Password mixing only continues as long as P_1...P_{18}. Once you've reached $\mathbf{P_{18}}$ and you still have password bytes unprocessed, continue the xor process on $\mathbf{P_1}$. A downside of this approach is that the computation time leaks the password length (side-channel information leak)

Solution 3 - Pre-hash password

The long password can be pre-hashed with another cryptographic hashing function. This function always outputs a digest of a fixed size. As long as the resulting digest is less than 72 bytes: it can be fed into bcrypt without being cut off. This is the approach taken by Dropbox[21] and others. For example, consider the password:

> ਤੇਜ਼ ਭੂਰੇ ਲੂੰਬੜ ਨੇ ਆਲਸੀ ਕੁੱਤੇ ਉੱਤੇ ਛਾਲ ਮਾਰ ਦਿੱਤੀ *(32 characters, 126 bytes)*

If you hash that password using SHA-256, you get the 256-bit (32-byte) digest:

```
A7 B2 AA 86 CB 57 D3 08 EA 54 9F 34 E5 91 7E 8C 06 9B D0 0D 34 28 B1 CA 8D 71 B2 2E 84 DA C0 F8
```

Base64 encoding the digest gives a string that is consistently 44 characters:

> `p7KqhstX0wjqVJ805ZF+jAab0A00KLHKjXGyLoTawPg=` *(44 characters, 44 bytes)*

and will always fit in the 72-byte bcrypt limit.

This is similar to an approach used by **Pufferfish2**, an cache-hard evolution of bcrypt, that uses $\text{HMAC}_{\text{SHA-512}}$ to convert the password into a fixed length 64-byte (512-bit) key.

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)*

Which then should get radix-64 encoded to 32-characters:

> `hSCvnwM9s4wIX9JeLapehKK5YdLxKcmk` *(32-characters)*

But the canonical OpenBSD implementation truncates password hash 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` ~~`a4`~~ *(23-bytes)*

and the resulting base-64 encoded text, which normally would be:

```
hSCvnwM9s4wIX9JeLapehKK5YdLxKck=
```

has the trailing = removed, leaving a hash of:

```
hSCvnwM9s4wIX9JeLapehKK5YdLxKck
```

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

Usage of non-standard base64 encoding

The Base64 encoding used by the canonical OpenBSD implementation uses an encoding dictionary that is different from those available in nearly every language and platform. The encoding is compatible with crypt.[22] This means the encoding is not compatible with RFC 4648.

See also

- bcrypt is also the name of a cross-platform file encryption utility implementing Blowfish developed in 2002.[23][24][25][26]
- Argon2 (The algorithm selected by the Password Hashing Competition in 2015)
- Crypt (C)#Blowfish-based scheme crypt – password storage and verification scheme – Blowfish
- Key stretching
- PBKDF2 (Password-Based Key Derivation Function 2)
- scrypt
- PufferFish (https://github.com/epixoip/pufferfish) is a cache-hard password hashing function based on improved bcrypt design.

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. "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. "bcrypt: Modern password hashing for your software and your servers" (https://github.com/pyca/bcrypt/) – via PyPI.
10. passlib. "Modular Crypt Format" (https://passlib.readthedocs.io/en/stable/modular_crypt_format.html).
11. passlib. "bcrypt" (https://passlib.readthedocs.io/en/stable/lib/passlib.hash.bcrypt.html).
12. "Modern(-ish) password hashing for your software and your servers: pyca/bcrypt" (https://github.com/pyca/bcrypt). November 18, 2019 – via GitHub.
13. "GitHub - bcgit/bc-java: Bouncy Castle Java Distribution (Mirror)" (https://github.com/bcgit/bc-java). November 18, 2019 – via GitHub.
14. "Modular Crypt Format — Passlib v1.7.1 Documentation" (https://passlib.readthedocs.io/en/stable/modular_crypt_format.html). *passlib.readthedocs.io*.
15. "bcrypt password hash bugs fixed, version changes and consequences" (http://undeadly.org/cgi?action=article&sid=20140224132743). *undeadly.org*.
16. Designer, Solar. "oss-sec: CVE request: crypt_blowfish 8-bit character mishandling" (http://seclists.org/oss-sec/2011/q2/632). *seclists.org*.
17. " 'bcrypt version changes' - MARC" (https://marc.info/?l=openbsd-misc&m=139320023202696). *marc.info*.
18. Schneier, Bruce (1994). "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 (December 1993)*. Springer-Verlag: 191–204.
19. "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*.
20. https://www.schneier.com/academic/blowfish/
21. https://dropbox.tech/security/how-dropbox-securely-stores-your-passwords
22. https://medium.com/hackernoon/the-bcrypt-protocol-is-kind-of-a-mess-4aace5eb31bd
23. http://bcrypt.sourceforge.net bcrypt file encryption program homepage
24. "bcrypt APK for Android - free download on Droid Informer" (https://droidinformer.org/tools/bcrypt/). *droidinformer.org*.
25. "T2 package - trunk - bcrypt - A utility to encrypt files" (http://t2sde.org/packages/bcrypt.html). *t2sde.org*.
26. "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/)