Computer and Information Science; Vol. 10, No. 3; 2017
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Using Improved d-HMAC for Password Storage

Mohannad Najjar[1]

[1] University of Tabuk, Tabuk, Saudi Arabia

Correspondence: Mohannad Najjar, University of Tabuk, Tabuk, Saudi Arabia. E-mail: najjar@ut.edu.sa

Received: May 3, 2017 Accepted: May 22, 2017 Online Published: July 10, 2017

doi:10.5539/cis.v10n3p1 URL: http://doi.org/10.5539/cis.v10n3p1

Abstract

Password storage is one of the most important cryptographic topics through the time. Different systems use distinct ways of password storage. In this paper, we developed a new algorithm of password storage using dynamic Key-Hashed Message Authentication Code function (d-HMAC). The developed improved algorithm is resistant to the dictionary attack and brute-force attack, as well as to the rainbow table attack. This objective is achieved by using dynamic values of dynamic inner padding *d-ipad*, dynamic outer padding *d-opad* and user's public key as a seed.

Keywords: d-HMAC, MAC, Password storage, HMAC

1. Introduction

Information systems in all kinds of organizations have to be aligned with the information security policy of these organizations. The most important components of such policy in security management is access control and password management. In this paper, we will focus on the password management component and mostly on the way of such passwords are stored in these information systems.

Password is the oldest and still the primary access control technique used in information systems. Therefore, we understand that to have an access to an information system or a part of this system the authorized user hast to authenticate himself to such system by using an ID (username) and a password

Hence, password storage is a vital component of the Password Access Control System. In this paper, we are proposing a new technique, which is superior to the existing known algorithms in the since that it yields good security for password storage and it is simple to use. In this algorithm, we are utilizing improved d-HMAC function, which is more resistance to RAINBOW attack than traditional HMAC.

We will present this new technique and we will compare it with the existing techniques that use HMAC algorithm to show the new possibility and potential in this new technique for further development for scientists.

Moreover, we will give a simple good introduction for the history of evolving the techniques used for password storage, which could be a good material and reference for the researchers and students interested in this subject.

2. Passwords and Passwords Storage History

Password is the most common access control tool used to authenticate authorized users. The basic definition of password is a secret series of characters remembered and known only by the user, used to login to any kind of information systems (Forouzan, 2008).

The security of password means that it must be resistance against dictionary attack and brute force attack and must not be guessable by the attacker. In order to have such strong password resistance against various attacks (Hitachi, 2016): 1) It must consist of a large alphabet with different combinations such as small letters, capital letters, numbers and special characters, 2) The length of password shall be as long as it can be memorized, and 3) the complexity of the password has to be very high.

Most of people think that having a strong none revealed guessable password will protect them, but they are wrong!! Because many of the know attack, where hackers were able to steal millions of accounts with their password was not a result of the password strength, it was because of how these passwords were stored (Hitachi, 2016). Password storage is an essential part of the whole Password access control system. Techniques used for this storage have to be designed to make it infeasible for the attacker to conclude the password even if the password storage system has been compromised.

In the following, we will present in which these password storage techniques evolved through the time.

Storage password systems involved through different stages:

First stage: Password (p) was saved as plaintext, which was vulnerable to many kinds of attacks (Forouzan, 2008). The attacker in an easy way could compromise the username and password.

$$c = (p)$$

Second stage: Programmers start to save passwords in a secret way by using symmetric encryption algorithms like Blowfish, DES, 3DES, etc (Stallings, 2011). Such solution was compromised in an easy way by phishing the encrypted passwords or the key itself.

$$c = \mathrm{E}_k(p)$$

Third stage: In this technique, one-way cryptographic hash functions like Message Digest MD4, Message Digest MD5, and Secure Hash Algorithm 1 SHA-1 are used to save the calculated hash code of the password (Forouzan, 2008). Therefore, if the attacker gained an access to saved passwords' hash codes, it will be infeasible to produce the input in an efficient polynomial time. This idea was invented to solve the weaknesses in the second stage.

$$c = \mathrm{Hash}\ (p)$$

When a user wants to login to the system the password is sent to the server by a secure channel. On the server, the password is hashed and the result is compared with the hash code already saved in the database if it is matched then authentication is granted else the request is rejected. However, an attacker can generate a big dictionary with hashed passwords; this attack is called the Rainbow attack (Oechslin, 2003). After this, the attacker will be able to crack the system by checking all hashed passwords saved in the system against the built dictionary.

Fourth stage: To prevent dictionary and Rainbow attacks new input parameter was added to the hashing process called "salt", the salt must be long unique random value, which saved with the hash code and used Id (Forouzan, 2008). Salt is just concatenated to the password before hashing process. Hence, each user password has a different hash codes even if they are similar. In this case, previously mentioned Rainbow table attack in the third stage will be worthless because the attacker must generate a separate dictionary for every user in the system to make a successful attack, which is so difficult to be done.

$$c = \mathrm{Hash}\ (p \parallel salt)$$

Hackers revert to brute-force attack utilizing fast computers, which calculate billions of hash codes per second (Percival & Josefsson, 2016). Hash functions are usually designed to be fast so they are vulnerable to such kind of attacks.

Fifth stage: Developers started using slow cryptographic algorithms such as Password-Based Key Derivation Function 2 PBKDF2, Blowfish-based BCrypt and SCrypt against the fast brute-force attacks (Kaliski, 2000), (Percival & Josefsson, 2016), (Provos & Mazières, 1999). These algorithms accept using work factor value along with the password and salt. The work factor defines the amount of time as an input parameter, which makes it possible to control the total time of computing the hash code for the password and the salt. Such technique is designed to safeguard against future hardware improvements as predicted in Moore's law (Moore, 1995).

$$c = \mathrm{Hash}\ (\text{“}work\ factor\text{”}, p \parallel salt)$$

3. Modern Algorithms Used for Password Storage

Nowadays, most common techniques for password storage uses storing of hash code of the password with a salt calculated by any kind of Message Authentication Code (MAC) tools (Stallings, 2011). These MAC tools like HMAC or Block cipher, where the password is used as MAC key and the salt as the message, are more secure than traditional calculating $\mathrm{H}(salt \| p)$. However, the main weakness for such solution is brute force attack, which still not resolved.

To overcome this issue, modern algorithms used for password storage depend on slowing the whole process to make it more resistant to brute force attack. Such algorithms like BCrypt, PBKDF2 and SCrypt use a technique called key stretching (Percival & Josefsson, 2016).

Most of the modern password storage algorithms take into consideration the control of the calculation time for generating hashed password to require more Central Processing Unit CPU power. Such solutions take into consideration Moore's Law. Hence, mean when computers get faster, such algorithms will be modified to request more CPU power. In this way, the attacker must invest more time to be able to calculate the password.

All of these techniques use the work factor parameter to slow the used hash function, as work factor will be different for each password, thus these algorithms are more resistant to all kinds of brute force attacks.

3.1. Algorithm BCrypt

BCrypt is one of the most known modern algorithms used in password storage. It was invented by Niels Provos and David Mazières (Provos & Mazieres, 1999). It is based on using Blowfish block cipher as a function of key derivation for passwords. BCrypt uses salt to protect against rainbow table attacks and it uses a function, which controls time of calculation. This function depends on controlling the iteration count to make the calculation of hashed password slower, in order to be more resistant to brute-force attacks.

Expensive Key schedule Blowfish (Eksblowfish) was developed to calculate keys in Blowfish algorithm (Provos & Mazieres, 1999). This technique for key generation starts by using the standard Blowfish key setup with some modifications, where password and salt are used to generate sub-keys. A number of rounds depending on work factor parameters in which the standard Blowfish keying algorithm is applied by alternating the salt and the password as the key, each round starting with the sub-key state from the previous round. Such solution makes the number of sub-key calculation rounds modifiable. Using work factor can make the whole process slow, which makes it difficult for attackers to use brute-force attack in order to conclude the real password especially that the hash code for the same password is different depending on the work factor parameter.

Bcrypt depends on two main functions Eksblowfish and ExpandKey, and the main Bcrypt algorithm that uses these functions to calculate hashed passwords.

The algorithm includes the following parameters:

- S – Salt, octet string randomly generated
- Pass – Password phrase
- W – Work factor
- Key – Key

Bcrypt (WorkFactor, salt, input):

```
state ←  EksBlowfishSetup(W, S, pass)
ctext ←  "OrpheanBeholderScryDoubt" //three 64-bit blocks
repeat (64)
ctext ←  EncryptECB(state, ctext)//encrypt using Blowfish in ECB mode
return Concatenate(W, S, ctext)
```

EksBlowfishSetup (W, S, K)

```
state ←  InitState()
state ←  ExpandKey(state, S, K)
    repeat ($2^W$)
        state ←  ExpandKey(state, 0, K)
        state ←  ExpandKey(state, 0, S)
return state
```

ExpandKey(state, S, K)

```
    for(n = 1..18)
        $P_n$ ←  K[32(n-1)..32n-1] xor $P_n$ //treat the key as cyclic
  ctext ←  Encrypt(S[0..63])
  $P_1$ ←  ctext[0..31]
  $P_2$ ←  ctext[32..63]
  for(n = 2..9)
      ctext ←  Encrypt(ctext xor salt[64(n-1)..64n-1]) //encrypt using the
                                          //current key schedule and treat salt as cyclic
        $P_{2n-1)}$ ←  ctext[0..31]
        $P_{2n}$ ←  ctext[32..63]
  for(i = 1..4)
```

```
        for($n$ = 0..127)
            $ctext \leftarrow$ Encrypt($ctext$ xor $S$[64(n-1)..64n-1]) //as above
              $S_i[2n] \leftarrow ctext[0..31]$
              $S_i[2n+1] \leftarrow ctext[32..63]$
    return state
```

3.2. Algorithm PBKDF2

PBKDF2 is newer version of PBKDF1, which was presented in RFC 2898 (Kaliski, 2000). It uses hash function technique to create pseudorandom keys. PBKDF2 algorithm is not used for direct password storage; it is used to generate keys that are used with hash function to create hashed password with a salt, and then derives a key by repeating the process as many times (work factor) as specified.

Function of PBKDF2 (Pass, S, c, dkLen)

The algorithm includes the following parameters:

- S – Salt, octet string randomly generated
- Pass – Password phrase
- c – Iteration counts
- hLen – Length of the pseudorandom function output in octets
- dkLen – Output length of key in octets dkLen $\leq (2^{32} - 1) * \text{hLen}$

```
If dkLen > $(2^{32} - 1) * hLen$ then go to exit;
l $\leftarrow$ CEIL (dkLen / hLen) //CEIL(x) is smallest integer $\leq$ x
r $\leftarrow$ dkLen - (l - 1) * hLen
for j = 1 to l do
    Y $\leftarrow$ F (Pass,S,c,1)
    $Z_i \leftarrow Y$
end for
Output $\leftarrow (Z_1 \| Z_2 \| ... \| Z_l)$
```

Function F (Pass, S, c, i):

```
F(Pass,S,c,i) $\leftarrow U_1$ xor $U_2$ xor …  xor $U_c$
$Z_1 \leftarrow$ PRF(Pass,S || INT(i)) //PRF is a pseudrandom function,
                                      //INT(i)is 4-octet encoding of i,
for j = 2 to c do
    Y $\leftarrow$ PRF(P,$Z_{i-1}$)
    $Z_i \leftarrow Y$
end for
Output $\leftarrow (Z_1$ xor $Z_2$ xor ... xor $Z_c)$
```

3.3 Algorithm SCrypt

Scrypt was created by Colin Percival (Percival & Josefsson, 2012) and it depends on function with password-based key derivation. Scrypt needs a lot of memory to be implemented, which make it so difficult for attackers to use parallel attacks.

Scrypt needs a very fast hardware and a big amount of memory as opposed to Bcrypt and PBKDF2 algorithms (Percival & Josefsson, 2016), which use salting or iteration or both as they do not consume many resources. Any attacker with little resources can launch a parallel attack with large scale and sometimes he could have some results in reasonable time.

The algorithm includes the following parameters:

- S – Salt, octet string randomly generated

- Pass – Password phrase
- R – Block size in octets.
- N – CPU/memory cost parameter, where $1<N<2^{(128*R/8)}$
- P – The parallelization parameter, where $P \leq (2^{32}-1)*32/(128*R)$.
- hLen – Length of hash code $HMAC_{SHA256}()$ in octets.
- dkLen – Output length of key in octets $dkLen \leq (2^{32}-1) * hLen$.

Function Scrypt(Pass, Salt, N, P, dkLen):

```
$(M_0 \ldots M_{p-1}) \leftarrow PBKDF2_{HMAC\_SHA256}(Pass, Salt, 1, P * R)$
for i = 0 to P-1 do
        $M_i \leftarrow ScryptR0Mix(M_i, N)$
end for
$Output \leftarrow PBKDF2_{HMAC\_SHA256}(Pass, M_0 \| M_1 \ldots M_{p-1}, 1, dkLen)$
```

Function ScryptR0Mix(B,N):

```
$Y \leftarrow M$
for i = 0 to N − 1 do
        $V_i \leftarrow Y$
        $Y \leftarrow ScryptBlockMix(Y)$
end for
for i = 0 to N − 1 do
        $j \leftarrow Integerify(Y) \bmod N$
        $Y \leftarrow ScryptBlockMix(Y\ xor\ V_j)$
end for
$Output \leftarrow Y$
```

Function scryptBlockMix(M):

```
$(M_0, \ldots, M_{2r-1}) \leftarrow M$
$Y \leftarrow M_{2r-1}$
For i = 0 to 2r − 1 do
        $Y \leftarrow H(Y\ xor\ M_i)$
        $Z_i \leftarrow Y$
end for
$Output \leftarrow (Z_0, Z_2, \ldots, Z_{2r-2}, Z_1, Z_3, \ldots, Z_{2r-1})$
```

4. Improved d-HMAC Function

Improved d-HMAC (Najjar, 2015) uses dynamic values *d-ipad* and *d-opad* instead of using fixed values of *ipad* and *opad* in HMAC (Krawczyk & Bellare, 1997). Dynamic values in d-HMAC of *d-ipad* and *d-opad* depend on three parameters (Najjar, 2015):

- Message *m*.
- Public key used by the receive e_K.
- S-Box

In other words, in case we have different receivers, different message content, or both then *d-ipad* and *d-opad* values will be different.

HMAC and d-HMAC require a cryptographic secret key *K* and hash function *h*. We assume *h* to be a cryptographic hash function with hash code *h*(*m*) not less than 256 bits. Our recommendation is to use at least SHA-256 hash function with key length 256 bits or higher.

d-HMAC can be calculated as the following (Figure 1):

$$\text{d-HMAC}_K(m) = h(K^+ \oplus d\text{-}opad,\ h(K^+ \oplus d\text{-}ipad,\ m))$$

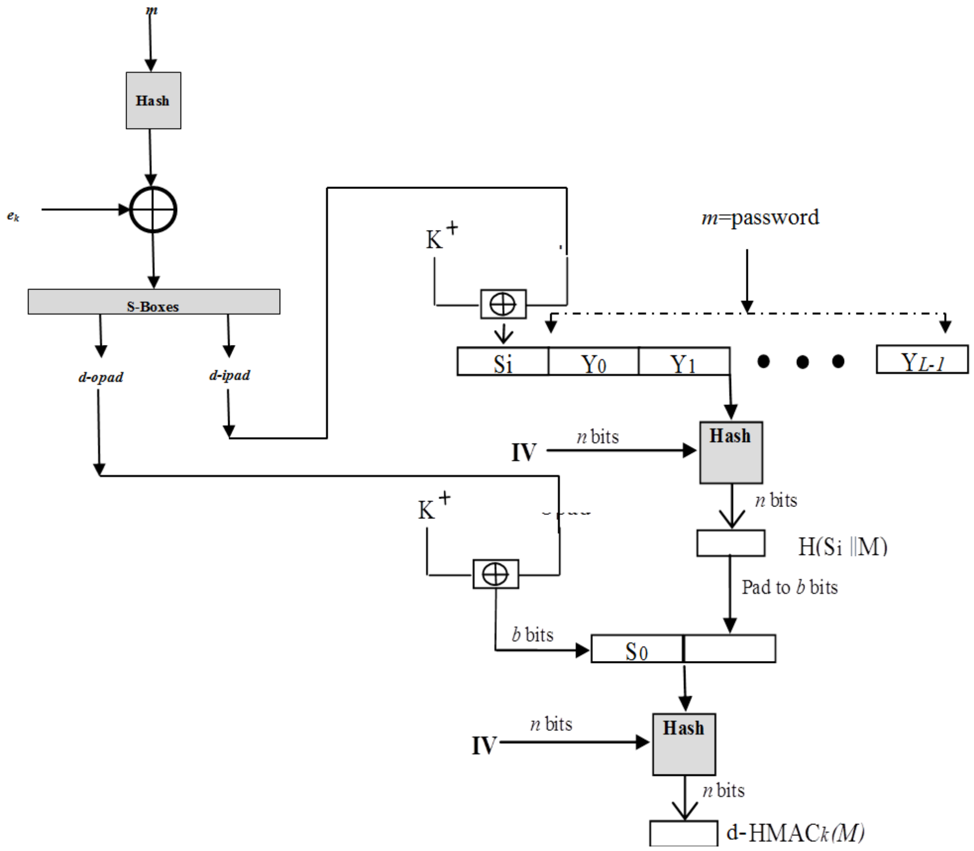

Figure 1. d-HMAC Algorithm (Najjar, 2015)

We are using d-HMAC, since it is more difficult to be comprised by brute force attack and birthday attacks than HMAC (Najjar, 2015). In d-HMAC hash code changes even if we send same message *m* and key *K* for different receivers, because these receivers have different public key e_R unlike HMAC, where we will have the same hash code for different receivers. On the other hand, d-HMAC uses dynamic *d-ipad* and *d-opad* values. This will make it difficult for the attacker to conclude *K* offline by collecting and observing band of hash codes generated by the same K, where in HMAC is easier because of using fixed *ipad* and *opad*.

In d-HMAC, the pepper already exists and the key represents it. Ordinary pepper is used with many hashed passwords with the same key, which makes it easier to the attacker to find *k* when the whole hashed passwords table with salt is compromised. In d-HMAC, such a threat does not exist because we always have different dynamic keys.

According to tests done for improved d-HMAC (Najjar, 2015), we proved in three different tests the strong cryptographic characteristics for d-HMAC, which make it a better tool used for password storage than HMAC.

These are the tests results:

Speed Test

As depicted in Table 1, it was proven that improved d-HMAC is slower than HMAC, which make it more attractive to be used in password storage.

Table 1. Test of speed HMAC and d-HMAC

Size of file (MB)	HMAC (s)	Improved d-HMAC (s)
1	0.062	0.107
5	0.189	0.443
20	0.901	1.886
50	1.844	3.634
100	3.210	6.169

Avalanche effect

As depicted in Table 2 and Table 3, for 10 000 random key samples it was proven that improved d-HMAC has a very good avalanche. This mean that the using of dynamic values *d-ipad* and *d-opad* in improved d-HMAC did not have a negative effect on Avalanche effect characteristic.

Table 2. Results for avalanche effect according to one bit change in message

Results for XOR	HMAC (bits)	Improved d-HMAC (bits)
MIN	97	97
MAX	157	159
Average	128.003	127.956

Table 3. Results for avalanche effect according to one bit key change

Results for XOR	HMAC (bits)	Improved d-HMAC (bits)
MIN	103	105
MAX	152	150
Average	127.775	128.295

Balancing

As depicted in table 4, it was proven that improved d-HMAC as HMAC has a very good balancing characteristic.

Table 4. Results for balancing according to message one bot change

Results for XOR	HMAC (bits)	Improved d-HMAC (bits)
MIN	98	98
MAX	162	157
Average	127.962	128.062

5. D-HMAC Resistance to Known Attacks for Password Storage

In d-HMAC, we can use any kind of known cryptographic hash functions. Our recommendation in this paper is to use a hash function resistance to exhausting attack like SHA-256 with hash code =256 bits length or more.

We can conclude d-HMAC is more secure than using normal salt, since salt may make the password less secure if the random value is not balanced. In our solution, we change the public key parameter for the receiver into a random string of characters not less than 20 bytes. In d-HMAC, even if the salt is not a good balanced random string of bits then our algorithm solves it and makes it balanced. As well, the salt is not appended to the password. It is used to modify the values of *d-ipad* and *d-opad*. This makes d-HMAC more resistance against RAINBOW attack.

d-HMAC is a slower function than HMAC because values of k_1= $K^+ \oplus$ *d-opad* and k_2= $K^+ \oplus$ *d-ipad* cannot be calculated once and stored like in HMAC, since we have dynamic values for *d-ipad* and *d-opad*.

In PBKDF2 (Kaliski, 2000), it is recommended to have at least 1000 iteration counts to be secure and to increase the cost of brute force attack search for passwords significantly. In d-HMAC without S-Box modification, you need 500 iteration counts, about half, as recommended in PBKDF2.

In d-HMAC the pepper is already exist and represented by the key. Ordinary pepper is used with many hashed passwords. Therefore, with the same key it is easier to find k when the whole hashed passwords table with salt is compromised. In d-HMAC, such threat does not exist because always we have different keys.

6. Conclusion and Perspectives

We showed that improved d-HMAC is a very good tool for password storage. Moreover, we showed the using of improved d-HMAC in password storage is resistant against Rainbow attack. In improved d-HMAC, for two different users having similar passwords, the hash code will not be equal since each of them has a different public key. Using such attack against improved d-HMAC will force the attacker to have a different table for each user, because they have different public keys. On the other hand, if we assume that the same user has different passwords in the same system for security issues such as different level of security, here also the rainbow attack will be infeasible because even for the same user there will be different input because of the dynamic character of *d-ipad* and *d-opad*.

In the future work, we will develop the S-boxes to be dynamic in such way to make it a one-way function and we will manipulate and expand it to control the speed of the whole algorithm calculation.

References

Bellare, M., Canettiy, R., & Krawczyk, H. (1996). Keying Hash Functions for Message Authentication. Crypto 96 Proceedings, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag.

Cazier, J. A., & Medlin, B. D. (2006). Password Security: An Empirical Investigation into E-Commerce Passwords and Their Crack Times. Information Systems Security, 15(6), 45–55.

Devillers, M. M. (2010). Analyzing Password Strength, Radboud University Nijmegen, Tech. Rep.

Devillers, M. M. (2010). Analyzing Password Strength, Radboud University Nijmegen, Tech. Rep.

Forouzan, B. (2008). Introduction Cryptography and Network Security, McGraw-Hill.

Hitachi ID Systems, (2016). Retrieved from http://hitachiid.com/passwordmanager/docs/passwordmanagementbest-practices.pdf

Kaliski, B. (2000). RFC 2898: PKCS #5: Password-Based Cryptography Specification Version 2.0, RSA Laboratories.

Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., Christin, N., Cranor, L. F., & Lopez, J. (2011). Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms, Carnegie Mellon University, Tech. Rep. CMU-CyLab-11-008

Krawczyk, H., Bellare, M., & Canetti, R. (1996) Pseudorandom Functions Revisited: The Cascade Construction and its Concrete Security Proceedings of the 37th Symposium on Foundations of Computer Science, IEEE

Krawczyk, H., Bellare, M., & Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication, RFC 2104.

Moore G. E. (1995, May). Lithography and the Future of Moore's Law, Proceedings of SPIE, Vol 2437.

Morris, R., & Thompson, K. (1979). Password Security: A Case History, Commun. ACM, 22(11), 594–597.

Najjar, M. (2003). Petra-r Cryptographic Hash Functions, International Conference on Security and Management SAM'03, Las Vegas, USA, Part I, 253-259.

Najjar, M. (2010, March). Generating of homogeneous Boolean functions with high nonlinearity using genetic algorithm, IJCSNS International Journal of Computer Science and Network Security, Vol.10, No.3, ISSN: 1738-7906, 1738-7906.

Najjar, M. (2015). d-HMAC — An improved HMAC algorithm, (IJCSIS) International Journal of Computer Science and Information Security, Vol. 13, No. 004

NIST FIPS PUB 198, (2002, March). The Keyed-Hash Message Authentication Code (HMAC), Federal Information Processing Standards Publication Issued.

Oechslin, P. (2003). Making a Faster Cryptanalytic Time-Memory Trade-Off. Advances in Cryptology-CRYPTO 2003. LNCS. 2729. p. 617. https://doi.org/10.1007/978-3-540-45146-4_36. ISBN 978-3-540-40674-7

Oechslin, P. (2003, August). Making a Faster Crypt-analytical Time-Memory Trade-Off. Lecture Notes in Computer Science. Santa Barbara, California, USA: Springer. ISBN 3-540-40674-3.

Oorschot, P., & Patrick, A. S. (2009). Passwords: If We're So Smart, Why Are We Still Using Them? FC '09: The 13th International Conference on Financial Cryptography and Data Security.

Percival, C., & Josefsson, S. (2012). The scrypt Password-Based Key Derivation Function. IETF. (Internet Engineering Task Force).

Percival, C., & Josefsson, S. (2016). RFC7914, The scrypt Password-Based Key Derivation Function.

Preneel B., (1998), Cryptographic primitives for information authentication – State of the art. Preneel B., Rijmen V. (Eds.), State of the Art in Applied Cryptography. LNCS 1528, Springer, Berlin, 1998, 49–104.

Provos N., & Mazières D., (1999). A Future-Adaptable Password Scheme. Proceedings of 1999 USENIX Annual

Technical Conference: 81–92**.**

Riddle, B. L., Miron, M. S., & Semo, J. A. (1989). Passwords in use in a university timesharing environment, Computers and Security, 8(7), 569–578.

Schaad, J., & Housley, R. (2003). Wrapping a Hashed Message Authentication Code (HMAC) key with a Triple-Data Encryption Standard (DES) Key or an Advanced Encryption Standard (AES) Key, RFC 3537

Stallings, W. (2011). Cryptography and Network Security Principles and Practices, Prentice Hall

Zhang Y., Monrose F., & Reiter M. K. (2010). The security of modern password expiration: an algorithmic framework and empirical analysis, in CCS ’10: Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM, pp. 176–186.

Copyrights