On the Weaknesses of PBKDF2

Andrea Visconti(✉), Simone Bossi, Hany Ragab, and Alexandro Calò

Department of Computer Science, Università degli Studi di Milano, Milan, Italy
andrea.visconti@unimi.it,
{simone.bossi2,hany.ragab,alexandro.calo}@studenti.unimi.it

Abstract. Password-based key derivation functions are of particular interest in cryptography because they (a) input a password/passphrase (which usually is short and lacks enough entropy) and derive a cryptographic key; (b) slow down brute force and dictionary attacks as much as possible. In PKCS#5 [17], RSA Laboratories described a password based key derivation function called PBKDF2 that has been widely adopted in many security related applications [6,7,11]. In order to slow down brute force attacks, PBKDF2 introduce CPU-intensive operations based on an iterated pseudorandom function. Such a pseudorandom function is HMAC-SHA-1 by default. In this paper we show that, if HMAC-SHA-1 is computed in a standard mode without following the performance improvements described in the implementation note of RFC 2104 [13] and FIPS 198-1 [14], an attacker is able to avoid 50 % of PBKDF2's CPU intensive operations, by replacing them with precomputed values. We note that a number of well-known and widely-used crypto libraries are subject to this vulnerability.In addition to such a vulnerability, we describe some other minor optimizations that an attacker can exploit to reduce even more the key derivation time.

Keywords: Key derivation function · CPU-intensive operations · Passwords · PKCS#5 · Optimizations

1 Introduction

Passwords are widely used to protect secret data or to gain access to specific resources. For sake of security, they should be strong enough to prevent well-know attacks such as dictionary and brute force attacks. Unfortunately, user-chosen passwords are generally short and lack enough entropy [9,16,18]. For these reasons, they cannot be directly used as a key to implement secure cryptographic systems. A possible solution to this issue is to adopt a key derivation function (KDF), that is a function which takes a source of initial keying material and derives from it one or more pseudorandom keys. Such a key material can be the output of a pseudo-random number generator, a bit sequence obtained by a statistical sampler, a shared Diffie-Hellman value, a user-chosen password, or any bit sequence from a source of more or less entropy [12]. KDF that input user passwords are known as password-based KDF. Such functions are of particular interest in cryptography because they introduce CPU-intensive operations on the attacker side, increasing the cost of an exhaustive search. By applying a KDF to a user password, we allow legitimate users to spend a moderate amount of time on key derivation, while increase the time an attacker takes to test each possible password. The approach based on KDF not only slows down a brute force attack as much as possible but also allows to increase the size of a cryptographic key.

In PKCS#5 [17], RSA Laboratories provides a number of recommendations for the implementation of password-based cryptography. In particular, they described Password-Based Key Derivation Function version 2 (PBKDF2), a function widely used to derive keys and implemented in many security-related systems. For example, PBKDF2 is involved in Android's full disk encryption (since version 3.0 Honeycomb to 4.3 Jelly Bean)[1], in WPA/WPA2 encryption process [11], in LUKS [7,10], EncFS [2], FileVault Mac OS X [6,8], GRUB2 [3], Winrar [5], and many others.

In order to slow down the attackers, PBKDF2 uses a salt to prevent building universal dictionaries, and an iteration count which specifies the number of times the underlying pseudorandom function is called to generate a block of keying material. The number of iterations is a crucial point of the KDF. The choice of a reasonable value for the iteration count depends on the environment and can vary from an application to another. In SP 800-132 [15], Turan et al. suggests that it is a good practice to select the iteration count as large as possible, as long the time required to generate the key is acceptable for the user. Moreover, they specify that for very critical keys on very powerful system an iteration count of 10,000,000 may be appropriate, while a minimum of 1,000 iterations is recommended for general purpose.

PBKDF2 introduce CPU-intensive operations based on an iterated pseudorandom function. Such a pseudorandom function is HMAC-SHA-1 by default.

In this paper we show that, if HMAC-SHA-1 is computed in a standard mode without following the performance improvements described in the implementation note of RFC 2104 [13] and FIPS 198-1 [14], an attacker is able avoid 50 % of PBKDF2's CPU intensive operations, by replacing them with precomputed values. Readers note that a number of well-known and widely-used crypto libraries e.g., [1,4], are subject to this vulnerability, therefore an attacker is able to derive keys significantly faster than a regular user can do. Moreover, we present some other minor optimizations (based on the hash function used) that can be exploited by an attacker to reduce even more the key derivation time.

The remainder of the paper is organized as follows. In Sect. 2, we present Password Based Key Derivation Function version 2 (PBKDF2). In Sect. 3 we briefly describe HMAC, that is the pseudorandom function adopted in PBKDF2. In Sect. 4 we present the weaknesses of PBKDF2. Finally, discussion and conclusions are drawn in Sect. 5.

[1] At the time of writing this represents 58 % of the Android devices market share (see developer.android.com).

2 PBKDF 2

Password Based Key Derivation Function version 2, PBKDF2 for short, is a key derivation function published by RSA Laboratories in PKCS #5 [17]. In order to face brute force attacks based on weak user passwords, PBKDF2 introduce CPU-intensive operations. Such operations are based on an iterated pseudorandom function (PRF) — e.g. a hash function, cipher, or HMAC — which maps input values to a derived key. One of the most important properties to assure is that PBKDF2 is cycle free. If this is not so, a malicious user can avoid the CPU-intensive operations and get the derived key by executing a set of equivalent, but less onerous, instructions. Unlike its predecessor (PBKDF version 1) in which the length of the derived key is bounded by the length of the underlying PRF output, PBKDF2 can derive keys of arbitrary length. More precisely, PBKDF2 generates as many blocks T_i as needed to cover the desired key length. Each block T_i is computed iterating the PRF many times as specified by an iteration count. The length of such blocks is bounded by $hLen$, which is the length of the underlying PRF output. In the sequel by PRF we will refer to HMAC with the SHA-1 hash function, that is the default as per [17]. Note that HMAC can be used with any other iterated hash functions such as RIPEMD, SHA-256 or SHA-512.

PBKDF2 inputs a user password/passphrase p, a random salt s, an iteration counter c, and derived key length $dkLen$. It outputs a derived key DK.

$$DK = PBKDF2(p, s, c, dkLen) \tag{1}$$

The derived key is defined as the concatenation of $\lceil dkLen/hLen \rceil$-blocks:

$$DK = T_1 || T_2 || \dots || T_{\lceil dkLen/hLen \rceil} \tag{2}$$

where

$$T_1 = Function(p, s, c, 1)$$
$$T_2 = Function(p, s, c, 2)$$
$$\dots$$
$$T_{\lceil dkLen/hLen \rceil} = Function(p, s, c, \lceil dkLen/hLen \rceil).$$

Each single block T_i is computed as

$$T_i = U_1 \oplus U_2 \oplus ... \oplus U_c \tag{3}$$

where

$$U_1 = PRF(p, s||i)$$
$$U_2 = PRF(p, U_1)$$
$$\dots$$
$$U_c = PRF(p, U_{c-1})$$

3 HMAC

Hash-based Message Authentication Code (HMAC) is an algorithm for computing a message authentication code based on a cryptographic hash function. The definition of HMAC [13] requires (a) H : any cryptographic hash function, (b) K : the secret key, and (c) $text$: the message to be authenticated. As described in RFC 2104 [13], HMAC can be defined as follows:

$$HMAC = H(K \oplus opad, H(K \oplus ipad, text)) \tag{4}$$

where H is the chosen hash function, K is the secret key, and $ipad$, $opad$ are the constant values (respectively, the byte 0x36 and 0x5C repeated 64 times) XORed with the password. Equation 4 can be graphically represented as in Fig. 1.

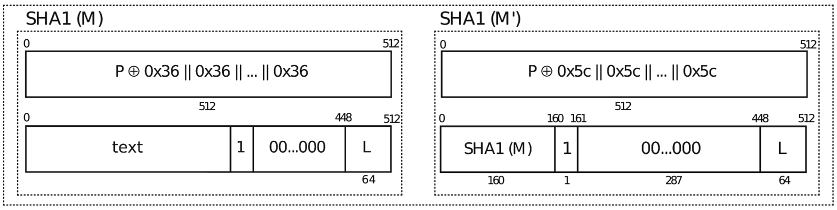

Fig. 1. HMAC-SHA-1

4 Weaknesses

In this section we present some weaknesses of PBKDF2. The major one concerns the precomputation of specific values that can be reused during the key derivation process. The others aim to avoid useless operations during the computation of the hash function. We will describe these weaknesses using as an example the parameters defined by LUKS format [9,10] — i.e., a salt length of 256 bits, and HMAC-SHA-1 as PRF.

4.1 Precomputing a Message Block

Looking closely at Fig. 2, note the following: (a) first message block of a keyed hash function is repeated c times (the dark gray rectangles in Fig. 2), (b) first message block of a second keyed hash function is repeated c times (the light gray rectangles in Fig. 2), (c) all dark gray rectangles have the same content and they can assume only the values SHA1($P \oplus ipad$), and (d) all light gray rectangles have the same content and they can assume only the values SHA1($P \oplus opad$). Thus, it is possible to compute these two blocks in advance — i.e., SHA1($P \oplus ipad$) and SHA-1($P \oplus opad$)[2] — and then use such values for c times. In so doing, an

[2] Readers note that the weakness is independent of the hash functions used and remains valid with any others.

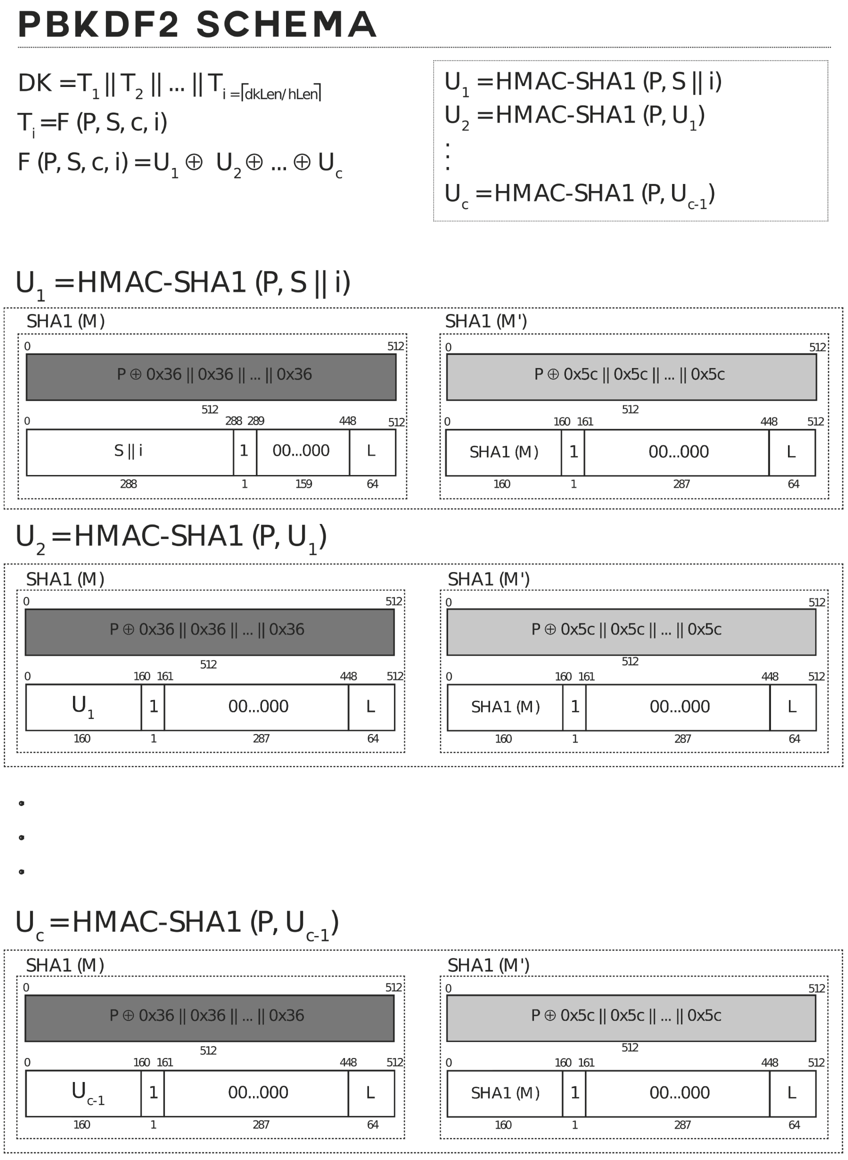

Fig. 2. PBKDF2 schema (HMAC computed in a standard mode)

attacker is able to avoid 50 % of the operations involved in the key derivation process (although the user can benefit from this optimization as well). Moreover, in real applications the counter c can be computed by benchmarking the user's system [7,9]. If this optimization is not implemented in crypto libraries, the benchmark fails to deliver the appropriate value for the counter, reducing the security level of the application.

4.2 Useless XOR Operations

It is easy to observe that each SHA1 message block, performed on a 512-bits string and formatted as shown in Fig. 3, has a run of several consecutive zeros (light gray rectangles). More precisely, in the second SHA1 message block there are 287 zeros in the padding scheme and other 54 zeros in L (i.e., 64-bits message length). Readers note that each SHA1 message block is split in sixteen 32-bits words, called $W_0 \ldots W_{15}$, and then expanded into eighty words, called $W_0 \ldots W_{79}$, using the following equation:

$$W_t = ROTL((W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}), 1) \qquad t \in [16 \ldots 79] \qquad (5)$$

Because we have a run of several consecutive zeros, a number of W_t are set to zero. More precisely, in Eq. 5 are carried out 192 XOR, but 27 of them are involved in zero based operations. Following the idea suggested in [19], these W_t do not provide any contribution and can be easily omitted by an attacker.

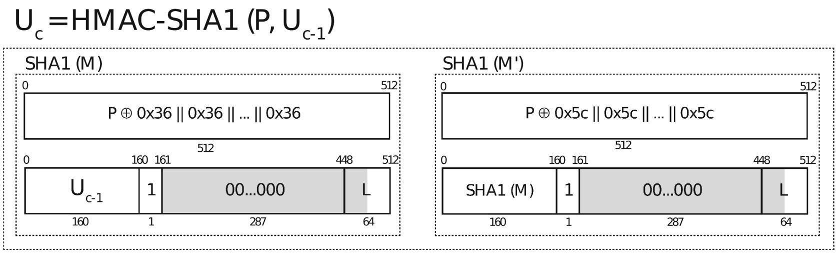
U_c =HMAC-SHA1 (P, U_c-1)
SHA1 (M)
SHA1 (M')
0
512
P ⊕ 0x36 || 0x36 || ... || 0x36
P ⊕ 0x5c || 0x5c || ... || 0x5c
160 161
448
U_c-1
SHA1 (M)
1
00...000
L
160
1
287
64

Fig. 3. Zero-padding scheme

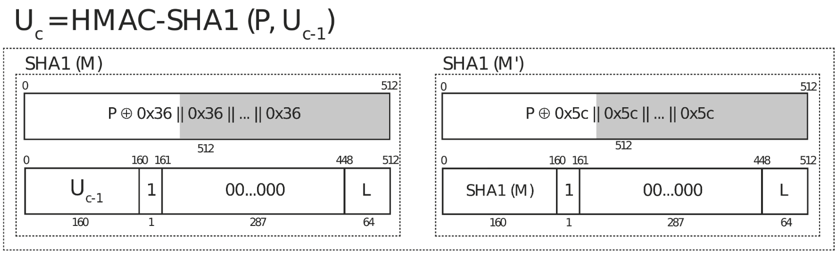
U_c =HMAC-SHA1 (P, U_c-1)
SHA1 (M)
SHA1 (M')
0
512
P ⊕ 0x36 || 0x36 || ... || 0x36
P ⊕ 0x5c || 0x5c || ... || 0x5c
160 161
448
U_c-1
SHA1 (M)
1
00...000
L
160
1
287
64

Fig. 4. Constant-padding scheme

In addition, an attacker is able to avoid some other useless operations in the word-expansion. As shown in Fig. 4, the constant 0x36 and 0x5C are used to pad the first message block up to the hash block size (dark and light gray rectangles). Since passwords are generally short, a number of W_t in Eq. 5 are set to the same value. If we XOR the same value twice, we get back the initial value. Again, these operations do not provide any additional contribution and can be omitted.

4.3 Precomputing a Word-Expansion

The last weakness described is a minor weakness. This provides the possibility to precompute the word-expansion part of the second message block of a keyed hash function (light gray rectangle in Fig. 5). Indeed, such a block is password-independent, and given a salt s (recall that s is a public information) an attacker is able to compute the expansion $W_0 \dots W_{79}$ in advance. A malicious user can reused the values precomputed with a dictionary of potential passwords to speed up a brute force attack.

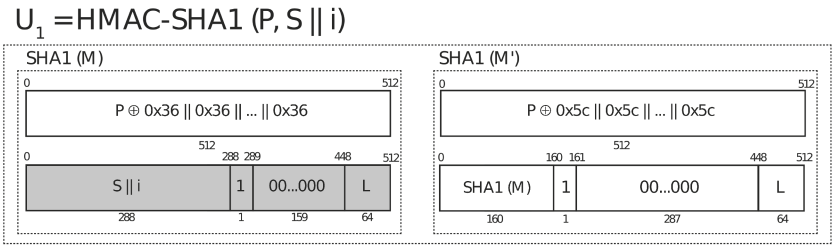

Fig. 5. Precomputing a specific word-expansion

5 Discussion and Conclusions

Passwords are generally short and lack enough entropy, therefore they cannot be directly used as a key to implement secure cryptographic systems. A possible solution to this issue is to adopt password-based key derivation functions.

This paper addressed the security of PBKDF2, one of the most commonly used function to derive cryptographic keys. We provided a detailed analysis of PBKDF2 schema and described some weaknesses that affect a number of well-known and widely-used crypto libraries such as [1,4].

The first one concerns the possibility to precompute the first message block of a keyed hash function and reuse such a value in all the subsequent HMAC invocations. This weakness allows an attacker to avoid 50 % of PBKDF2's CPU intensive operations, replacing them with constant values. Crypto libreries are subjected to this vulnerability if they do not implement the performance improvements described in RFC 2104 [13] and FIPS 198-1 [14].

The second one concerns the possibility to avoid useless XOR operations. Indeed, introducing zero-based operations and XORing the same value twice do not provide any additional contribution to the word expansion of SHA-1. Note that the same approach can be also applied to the word expansion of SHA-2 family hash functions.

The third one concerns the possibility to precompute the word-expansion of a specific 512-bits message block.

Readers note that the weaknesses of PBKDF2 described in this paper can be easily mitigated by selecting an iteration count c as large as possible and implementing the performance improvements that save the computation of several message blocks of a keyed hash function.

References

1. ARM mbed TLS, Version: 1.3.11. https://tls.mbed.org/
2. EncFS Encrypted Filesystem. https://sites.google.com/a/arg0.net/www/encfs
3. GNU GRUB Manual, Version: 2.00. http://www.gnu.org/software/grub/manual/grub.html
4. Libgcrypt, Version: 1.6.3. https://www.gnu.org/software/libgcrypt/
5. RAR Archive Format, Version: 5.0. http://www.rarlab.com/technote.htm
6. Apple Inc.: Best Practices for Deploying FileVault 2. Technical report (2012). http://training.apple.com/pdf/WP_FileVault2.pdf
7. Bossi, S., Visconti, A.: What users should know about full disk encryption based on LUKS. In: Proceedings of the 14th International Conference on Cryptology and Network Security (2015)
8. Choudary, O., Grobert, F., Metz, J.: Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption. Cryptology ePrint Archive, Report 2012/374 (2012). https://eprint.iacr.org/2012/374.pdf
9. Fruhwirth, C.: New methods in hard disk encryption (2005). http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf
10. Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.2.1 (2011). http://wiki.cryptsetup.googlecode.com/git/LUKS-standard/on-disk-format.pdf
11. IEEE 802.11 WG: Part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Std 802.11 i-2004 (2004)
12. Krawczyk, H.: Cryptographic Extraction and Key Derivation: The HKDF Scheme. Cryptology ePrint Archive, Report 2010/264 (2010)
13. Krawczyk, H., Bellare, M., Canetti, R.: RFC 2104: HMAC: Keyed-hashing for message authentication (1997)
14. NIST: FIPS PUB 198: The Keyed-Hash Message Authentication Code (HMAC) (2002)
15. NIST: SP 800–132: Recommendation for password-based key derivation (2010)
16. NIST: SP 800–63-2 Version 2: Electronic authentication guideline (2013)
17. RSA Laboratories: PKCS #5 V2.1: Password Based Cryptography Standard (2012)
18. Shannon, C.E.: Prediction and entropy of printed English. Bell Syst. Tech. J. **30**(1), 50–64 (1951)
19. Steube, J.: Optimizing computation of Hash-Algorithms as an attacker (2013). http://hashcat.net/events/p13/js-ocohaaaa.pdf