PAKE protocols and Decoy passwords

Wednesday 10^{th} January, 2024 - 18:20

Steve Meireles Lopes
University of Luxembourg
Email: steve.meireles.001@student.uni.lu

This report has been produced under the supervision of:
Marjan Skrobot
University of Luxembourg
Email: marjan.skrobot@uni.lu

Abstract—

Index Terms—

1. Introduction

Since the invention of personal computers, a username and password combination has been one of the most used authentication methods until today. Even though it is known that it has various drawbacks and weaknesses; including, humans being bad at making up strong and unpredictable passwords which makes it vulnerable to brute force and dictionary attacks. A problem a system administrator also has to solve is how to store the usernames and passwords. This is usually done in a password file (Example Linux: /etc/shadow /etc/passwd). In the past enterprises struggled with data breaches being open without them realizing or realizing too late, which resulted in leakage of their password files. This can be devastating to them and their clients. To make a system more secure one can incorporate fake accounts in the password file, therefore if somebody tries to enter the system using a fake account, the system administrator will be alarmed and know that the password file is in possession of outsiders. This report will analyze several security mechanisms one of them being fake passwords also called honeywords proposed by Juels and Rivest [1].

Although weak passwords are not secure they have some benefits, one of them is being very user-friendly and easy to remember. Nowadays, several protocols make it possible to securely authenticate peers by agreeing on a session key using a weak password. Such protocols are called PAKE protocols [2] [3] [4] [5]. This report is going to analyze the PAPKE protocol of Bradley, Gamenisch, Jarecki, Lehmann, Neven, and Xu [6] in detail, which is a PAKE protocol using a public key. This protocol enables the ability to authenticate for example a browser without needing to trust a third party by using certificates, which is vulnerable to phishing attacks where attackers can pretend to be the authentication server.

The paper SweetPAKE of Arriaga, Ryan and Skrobot [7] shows several approaches to combine both principles Honeywords and PAKE. They highlight the secure approaches and implementation. This report revolves mainly around the SweetPAKE paper, it analyzes the different approaches and offers an implementation of one approach. The implementation uses parts of code of the SPAKE2 implementation of warner [8].

Analyzing Honeywords, PAKE protocols, and SweetPAKE, this report tries to answer the question: How to detect if a password file is in possession of intruders and at the same time prevent phishing attacks?

2. Honeywords

2.1. Password file

Juels and Rivest [1] assume in their paper that a system on which users have to log in with a password. Each user has an entry c_i in the password file. A good example is the password files of an unix-like system /etc/passwd [9] and /etc/shadow [10]. They store the username u_i, the hash of the password $H(p_i)$, and additional information about the user.

$$c_i = (u_i, H(p_i))$$

Honeywords as explained in the introduction section are decoy passwords, therefore when using this method, the password file will have up to k passwords attached to each entry. Let S_i be the set of the correct password and all decoy passwords of the user u_i, the set will be called sugarwords.

$$S_i = H(p_0), H(p_1), ..., H(p_k)$$

. Thus, an entry c_i in the password file will be defined as follows:

$$c_i = (u_i, S_i)$$

Username	Plain password	Hashed password using SHA256
Bob	bob123	8d059c3640b97180dd2ee453e20d34ab0cb0f2eccbe87d01915a8e578a202b11
Alice	alice123	4e40e8ffe0ee32fa53e139147ed559229a5930f89c2204706fc174beb36210b3

TABLE 1. Hashed password without salt

Username	Plain password	salt	Hashed password using SHA256
Bob	bob123	PqaH7b9P	d79fe7c073f3a081e81b7e230c54124f0b11484508cd961d349436f9d4ef1e45
Alice	alice123	T2dYghL3	8988d499b23ee3d90f3240f10f1cb48e0387701de5ef41d14100960ab007a203

TABLE 2. Hashed password with salt

The hashed passwords should be hashed with an additional salt. Salt is extra information which is usually a randomly generated string concatenated to the password, the resulting string is hashed. It is recommended to use one salt per user. If not salted passwords are easily brute forceable which makes the system more prone to attacks. After salting, the adversary has a larger set of strings to test to get the same hash. To make it more clear you can look at the examples provided in table 1 and 2.

2.2. Honeychecker

The goal of using Honeywords is to alarm the system administrator that the password file is compromised. If the password file is compromised, one should assume that the system holding the file is also exposed to an adversary. This implies that the adversary has potential knowledge or even control over the alarm mechanism if it is programmed in this same system. To overcome this, Juels and Rivest suggest a distributed security system consisting of a separate hardened computer called a honeychecker. The honeychecker should detect abnormalities and raise an alarm if the password file is breached.

The honeychecker stores a single number n_i in its database for each user and it will never receive or store the password itself. The number n_i is in the range of $1tok$, k being the number of sugarwords of the user u_i. The honeychecker will have two functions, the first being:

$$Set(i, j)$$

The function takes as parameters the index of a user i and a new index of the correct password j. It sets $n_i = j$. The second function is:

$$Check(i, j)$$

The function takes as parameters also the index of the user i and an index of a password j. The honeychecker has to check whether n_i is equal to j.

An advantage of this distributed security system is, that if the honeychecker is compromised, the password has the same security level as if the honeychecker did not exist.

In figure 1 you see a flowchart depicting a login process. First, the user tries to log in with a password p. The system checks if $p \in S_i$; if not the system has a choice between several policies on what to do, in the case of a user entering the wrong password. If $p \in S_i$, the system sends i to the honeychecker H. H then checks, if the password is the correct one of the sugarwords. If it returns false, it raises an alarm, and the system administrator then again has several policies to choose from when such an alarm occurs. An example is a honeypot, where the user is directed to a decoy screen with restricted access. When the honeychecker returns true, it sends a confirmation to the system. After that, the user is granted access.

The change or create process is represented in 2. In both processes the user u_i starts with sending a request to the server. The server then generates $k - 1$ honeywords according to the given password p with help of a generating method $Gen(p)$. Generating methods are going to be looked at in detail in the following section. The function returns the resulting set of sugarwords S_i. After, that it sends the index of the correct password j to the honeychecker$H$. The honeychecker H then uses its' $Set(i, j)$ function explained above, then confirms the change/create of the index to the server. The server then updates the entry c_i with the new sugarwords.

2.3. Generation methods

Juels and Rivest propose several methods for generating decoy passwords, such that it is hard to guess the correct password in case the adversary knows all the sugarwords. The goal of a good method is to be flat, meaning that when the adversary knows all k sugarwords S_i of some user u_i. The probability that he chooses the correct one is

$$P = 1/k$$

.

There are two different categories legacy-UI password and modified-UI approach. With legacy-UI password approaches just have to tell the system their new password. With the help of this password, the honeywords are generated. The benefit of this approach is that the user does not need to know that honeywords are generated. The report looks at two methods and an additional hardening of such methods:

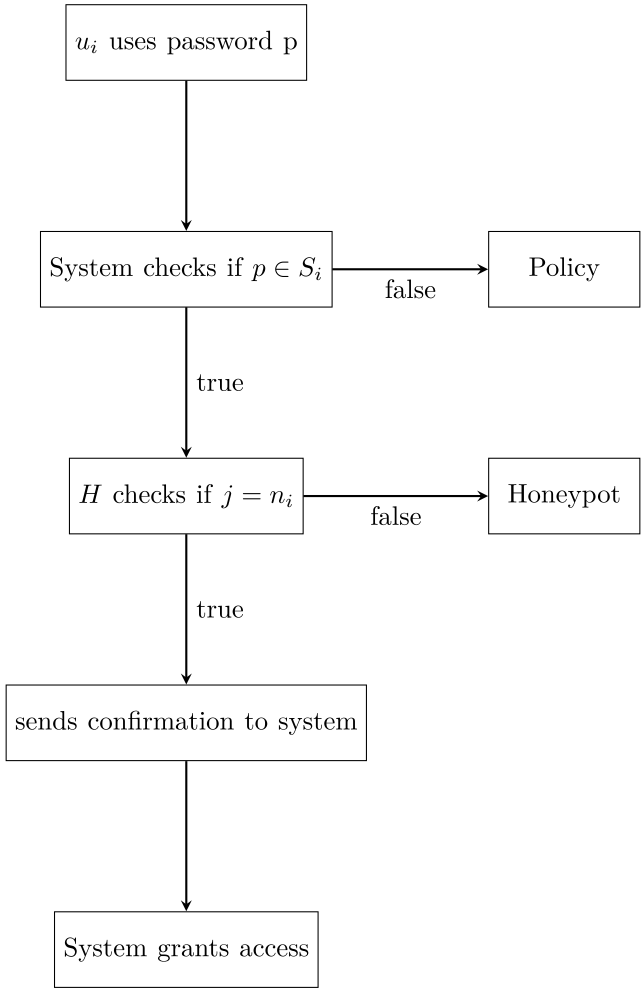
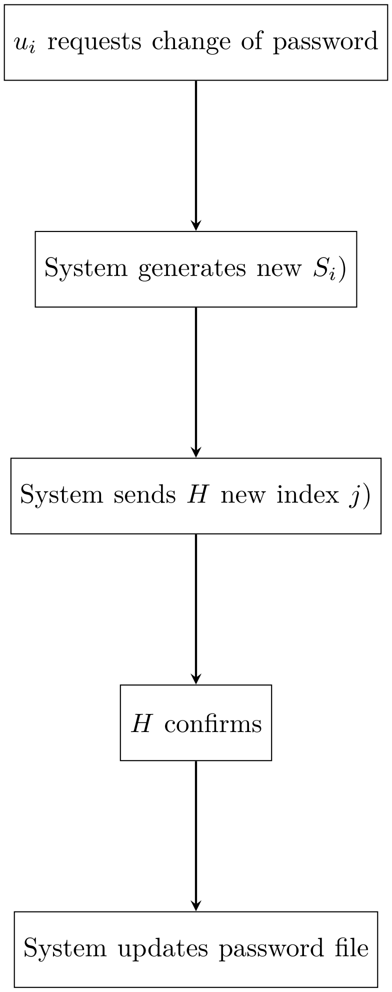


Figure 1. Honeychecker Login Flowchart



u_i requests change of password
System generates new S_i)
System sends H new index j)
H confirms
System updates password file

Figure 2. Honeychecker Change/Create Flowchart

- Chaffing by tweaking
- Chaffing with a password model
- Use of ”tough nuts”

On the other hand with modified-UI approaches the user is asked again for some extra intervention for example appending three random generated numbers, such methods make it easier to prove the flatness of a method. The report will look at the ”take-a-tail” method as an example.

2.3.1. Chaffing by tweaking. Chaffing by tweaking involves changing specific positions of strings, such that digits are replaced by digits, letters by letters, and special characters by special characters. The character is replaced by a random character. Examples of such methods are chaffing-by-tail-tweaking and chaffing-by-tweaking-digits. It is important to note that one should only change patterns of characters because it could become obvious to distinguish the real password if one changes specific characters of a word for example. The flatness of this method relies on the user, if the tweaked positions are also randomly chosen by the user implies perfect flatness.

2.3.2. Chaffing with a password model. This model generates honeywords using a large set of real passwords. Although using a public list might give the adversary to exploit the list to his advantage. A simple example approach using this method would be splitting the given password into separate words and replacing the words with the help of a large set of words. When replacing the words one would replace 4-letter words with 4-letter words and n-letter words with n-letter words.

2.3.3. Use of tough nuts. Tough nuts are honeywords that are very hard to impossible to crack. This can improve the security of chaffing algorithms. By incorporating several tough nuts, the adversary cannot know if the passwords are among the tough nuts or the rest. Thus, making it harder for him to guess the correct password.

2.3.4. Take-a-tail. This method is an example of a modified-UI change approach. When the user enters a new password, the system generates a random tail for example three numbers, and asks the user to remember and append them to their password. This ensures that the adversary can't distinguish between the honeywords and the password.

3. PAKE protocols

PAKE is an abbreviation for Password Authenticated Key Exchange, such protocols make it possible to have secure communication using a weak shared secret key. As already mentioned in the introduction there are several PAKE protocols. In this section, we are going first to look at some basic mandatory knowledge needed to understand these protocols by explaining Cyclic Groups and the Diffie-Hellman protocol. After that we are going to look at two protocols in detail: EKE and PAPKE. EKE is a rather simpler PAKE protocol and PAPKE is an important protocol because, in the following section, the SweetPAKE protocol is built upon exactly this protocol.

3.1. Cyclic Groups

This subsection covers the math base for the upcoming section.

3.1.1. Group. A group g is a structure that consists of a set of elements and an operation $\cdot$ that satisfies four properties:

Closure:

$$\forall g, h \in G, g \cdot h \in G$$

Identity element:

$$\exists i \in G, \forall g \in G, i \cdot g = g = g \cdot e$$

Associativity:

$$\forall g_1, g_2, g_3 \in G, (g_1 \cdot g_2) \cdot g_3 = g_1 \cdot (g_2 \cdot g_3)$$

Inverses:

$$\forall g \in G \exists h \in G, g \cdot h = e = h \cdot g$$

3.1.2. Generator. A generator g is an element of a cyclic group G, such that g when repeatedly using a group operation $\cdot$ on itself, it can generate all elements of G.

3.1.3. Definition. A group G is cyclic if there exists $g \in G$ such that g is a generator.

Important note: If G is a cyclic group and has generator g then

$$G = \{a^n | n \in \mathbb{Z}\}$$

3.1.4. Decisional Diffie-Hellman assumption. The assumption states that having g^a and g^b, it is hard to compute g^{ab}.

3.2. Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange method is named after Whitfield Diffie and Martin Hellman. The method allows two parties to establish a secret key without prior knowledge by using cyclic groups.

First, two parties Alice and Bob agree on a modulus p and a generator g. Both can be publicly known.

Second, Alice generates a random integer a, then sends Bob

$$A = g^a \% p$$

while Bob also generates a random integer b and sends

$$B = g^b \% p$$

Now, Alice computes

$$g^{ab} = B^a \% p$$

and Bob computes

$$g^{ab} = A^b \% p$$

Both never send their generated key a and b therefore it is very hard to compute gab for an eavesdropper according to Decisional Diffie-Hellman assumption.

3.3. EKE: Encrypted Key Exchange

The first PAKE protocol was introduced by Bellovin and Merrit called EKE [2]. In this section, we are going to analyze how the protocol works. It allows two parties to securely authenticate and communicate to each other over an insecure channel using a shared secret key namely a password.

Let Alice and Bob be two parties communicating with each other using the EKE protocol. Both share a common secret key, here s for secret. Often long before the protocols begin they agree on a modulus p and a generator g. Alice starts by generating a random integer a and computes $g^a \% p$ encrypted with the secret key s. She then sends her id id_a in plain, and her encrypted key.

$$id_a, s(g^a \% p)$$

Bob receives the message and also generates a random integer b and computes $g^b \% p$ and encrypts it with the secret key s. He then decrypts the message by Alice and computes the session key k.

$$g^{ab} \% p$$

Bob, then generates what is called a ”challenge” by Bellovin and Merrit, which serves to check the message is modified between the two parties. The challenge is encrypted with the computed session key k and sent together with $s(g^b\%b)$.

$$s(g^b\%p), k(challenge_B)$$

Next, Alice generates the session key k by decrypting $s(g^b\%p)$ and computing

$$g^{ab}\%p$$

She then generates her challenge, encrypts it together with Bobs' challenge and sends it to Bob.

$$k(challenge_A, challenge_B)$$

Bob, then decrypts the message and checks, if $challenge_B$ was changed during communication by an adversary. He sends $challenge_A$ back to Alice.

$$k(challenge_A)$$

Alice verifies if $challenge_A$ is still the same.

Now, having a basic understanding of the idea of PAKE protocol, more complex protocols using this idea can be explained.

3.4. PAPKE

PAPKE is an abbreviation of Password-Authenticated Public-Key Encryption. It is introduced by Bradley, Camenisch, Jarecki, Lehmann, Neven and Xu with a thorough study [6]. It focuses on generating long-term keys. It tries to replace to currently often used method where a third party using certificates has to be trusted to provide a secure end-to-end encryption. The study also points out that PAPKE implies PAKE but not the other way around, this proves that PAPKE is an improved primitive. They also suggest two schemes PAPKE named PAPKE-IC and PAPKE-FO. In this analysis, PAPKE-FO is a two-round PAKE protocol that is going to be looked at since it is the one that is implemented and used (see section Application).

The scheme splitted into three functions *Gen*, *Enc*, and *Dec*. Let Alice and Bob be two parties who want to generate long-term keys. Setup: Both parties agree on a group G of prime order p and two generators g_1, g_2. This protocol uses three hash functions, thus the setup consists of:

- password pwd
- Group G
- Generators g_1 and g_2
- $H_0 : \{0,1\}^* \to G$
- $H_1 : G^3 \times \{0,1\}^n \to \mathbb{Z}_p^2$
- $H_2 : G \to \{0,1\}^n$

3.4.1. *Gen*:.

Alice does the following computations: Generates random integers in the range of G (The $\leftarrow_R$ stands for random assignment):

$$x \leftarrow_R \mathbb{Z}_p$$

Generates two elements of G using both generators g_1, g_2 and x.

$$y_1 \leftarrow \ g_1^x$$

$$y_2 \leftarrow \ g_2^x$$

Then computes Y_2:

$$Y_2 \leftarrow \ y_2 \cdot H_0(pwd)$$

The tuple (y_1, Y_2) gives us the public key apk:

$$apk \leftarrow \ (y_1, Y_2)$$

Alice stores the secret key sk:

$$sk \leftarrow (x, y_1, y_2)$$

Alice sends her id and the public key apk to Bob.

The *Gen* function can be defined as follow:

$$Gen(pwd) = (sk, apk)$$

3.4.2. *Enc*.

Bob generates random long-term key k:

$$k \leftarrow_R \{0,1\}^*$$

After that Bob interprets the message $apk = (y_1, Y_2)$ and computes:

$$y_2 \leftarrow Y_2 \cdot H_0(pwd)^{-1}$$

Then he gets a random element of G and computes 2 elements r_1 and r_2 using H_1:

$$R \leftarrow_R G$$

$$(r_1, r_2) \leftarrow H_1(R, y_1, y_2, k)$$

In the next step, he starts computing secrets. The first secret c_1 using both generator and r_1, r_2. The second secret using $y_1^{r_1}$, $y_2^{r_2}$ and R. The third secret by XORing the hash of R and k

$$c_1 \leftarrow g_1^{r_1} g_2^{r_2}$$

$$c_2 \leftarrow y_1^{r_1} y_2^{r_2} \cdot R$$

$$c_3 \leftarrow H_2(R) \oplus k$$

$$c = (c_1, c_2, c_3)$$

Bob then sends his id and the computed secrets c.

The *Enc* function can be defined as:

$$Enc(pwd, apk) = c$$

3.4.3. *Dec.* Alice interprets the inbound message and computes R, the long term k by xoring c_3 with the hash of the computed R. She also computes r_1 and r_2 to later confirm that nobody tried to imitate Bob.

$$(r_1, r_2) \leftarrow H_1(R, y_1, y_2, k)$$

Alice then checks if an adversary tried to masquerade as Bob:

$$\text{checks if } c_1 = g_1^{r_1} g_2^{r_2}$$

The *Dec* function can be defined as:

$$Dec(pwd, c) = k$$

It is a rather more complex protocol but uses the same mathematical principles as Diffie-Hellman and EKE protocol with additional steps to ensure to share long-term key that can be used for encrypting multiple messages.

4. SweetPAKE

Honeywords assumes the connection runs over servers using TLS, which has several problems including that the trust of a third party is needed. It is also more prone to phishing if the attacker succeeds in disguise as the server that distributes certificates or by tricking the user. To solve this problem, Arriaga, Ryan, and Skrobot [7] propose the SweetPAKE protocol, which combines the strength of a PAKE protocol and honeywords. First, they propose a naive approach using the SPAKE protocol however it is not efficient. They then propose a secure SweetPAKE protocol using PAPKE, they call it BeePAKE.

Note: The authors of SweetPAKE highly recommend using MAC authentication during the protocol but this report does not include MAC because it is out of the scope of this project and is not implemented in the technical part.

4.1. Setup

Let two parties be Alice and Bob, both share a secret key, here password *pwd*. Let *Gen*, *Enc*, and *Dec*, be three functions of a secure PAPKE encryption scheme. Let PRF be a secure pseudo-random function which returns a vector of randomly generated long-term keys.

4.2. Process

4.2.1. . Alice generates (apk, sk) using the *Gen* function of PAPKE is explained in the section above.

$$(apk, sk) \leftarrow Gen(pwd)$$

She sends her *id* and *apk* to Bob.

4.2.2. . Bob generates a random key k that is used to generate a vector of long-term keys K.

$$k \leftarrow 0, 1^*$$

$$K = PRF(k, (id_a, apk))$$

Bob then creates a vector C by generating secrets c using *Enc* for every sugarword in the password file.

```
for $i = 1, ..., n$ do
    $C[i] \leftarrow Enc(pwd, apk, K[i])$
end for
```

After that he does a random permutation RP of vector C, meaning shuffles the vector, such that the position of the correct password remains unknown even for Alice. He then stores the mapping *pmap* of the permutation.

$$(C', pmap) = RP(C)$$

The vector C will then be sent to Alice together with Bobs' *id*.

4.2.3. . Alice uses the *Dec* PAPKE function for every element in vector C. If the function cannot decrypt a valid key the process is aborted. After a successful decryption, Bob stores the position i.

```
for $i = 1, ..., n$ do
    $k' \leftarrow Dec(sk, C'[i])$
end for
if $k' = undefined$ then abort
end if
```

Alice computes the long-term session key and sends the index i to Bob.

$$tr \leftarrow (id_a, id_b, apk, C, i)$$

$$key \leftarrow PRF(k', tr)$$

4.2.4. . Bob then computes with help of the stored mapping *pmap* the correct position of the password of the user and the key.

$$tr \leftarrow (id_c, id_b, apk, C, i)$$

$$key \leftarrow PRF(K[i], tr)$$

After the key computation, the server sends the honeychecker (explained in section Honeywords) the clients' user id and the position i. The honeychecker then checks if i is the position of the correct password, if it is not the honeychecker alarms the system administrator.

5. Application

5.1. Design

5.2. Implementation

6. Conclusion

7. Appendix

References

[1] A. Juels and R. L. Rivest, “Honeywords: Making password-cracking detectable,” in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, 2013, pp. 145–160.

[2] S. M. Bellovin and M. Merritt, “Encrypted key exchange: Password-based protocols secure against dictionary attacks,” 1992.

[3] M. Bellare, D. Pointcheval, and P. Rogaway, “Authenticated key exchange secure against dictionary attacks,” in International conference on the theory and applications of cryptographic techniques, Springer, 2000, pp. 139–155.

[4] V. Boyko, P. MacKenzie, and S. Patel, “Provably secure password-authenticated key exchange using diffie-hellman,” in Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19, Springer, 2000, pp. 156–171.

[5] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie, “Universally composable password-based key exchange,” in Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24, Springer, 2005, pp. 404–421.

[6] T. Bradley, J. Camenisch, S. Jarecki, A. Lehmann, G. Neven, and J. Xu, “Password-authenticated public-key encryption,” in Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17, Springer, 2019, pp. 442–462.

[7] M. Skrobot, “Sweetpake: Key exchange with decoy passwords,” in SweetPAKE, 2023.

[8] Warner, Python-spake2, https://github.com/warner/python-spake2, 2016.

[9] Passwd(5) linux user’s manual, Oct. 2023.

[10] Shadow(5) linux user’s manual, Oct. 2023.