PAKE protocols and Decoy passwords

Thursday 11th January, 2024 - 19:27

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 serves several purposes being to provide an example how one can implement such a protocol, the code can also be used as template and it can also be used to benchmark the protocol, to see if it is fast enough to operate in the real world. 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.

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

Username	Plain password	Hashed password using SHA256
Bob	bob123	8d059c3640b97180 dd2ee453e20d34ab 0cb0f2eccbe87d01 915a8e578a202b11
Alice	alice123	4e40e8ffe0ee32fa 53e139147ed55922 9a5930f89c220470 6fc174beb36210b3

TABLE 1. Hashed password without salt

User-name	Plain password	salt	Hashed password using SHA256
Bob	bob123	PqaH7b9P	d79fe7c073f3a081 e81b7e230c54124f 0b11484508cd961d 349436f9d4ef1e45
Alice	alice123	T2dYghL3	8988d499b23ee3d9 0f3240f10f1cb48e 0387701de5ef41d1 4100960ab007a203

TABLE 2. Hashed password with salt

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

$$c_i = (u_i, S_i)$$

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

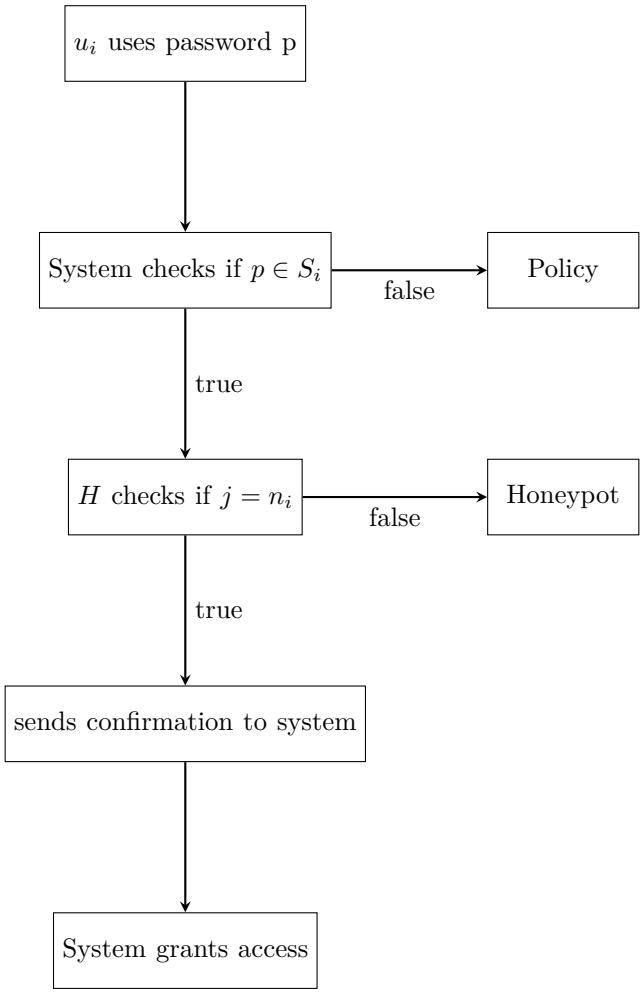

Figure 1. Honeychecker Login Flowchart

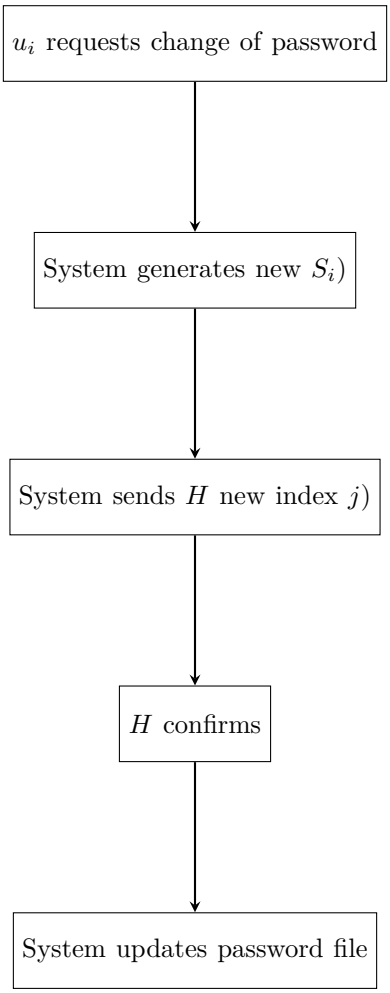

Figure 2. Honeychecker Change/Create Flowchart

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:

- 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

This section covers the technical part of this project. It consists of an implementation of a BeePAKE protocol [7].

5.1. Requirements

The application should be an implementation of BeePAKE [7] which has been discussed and analyzed in the previous section. The implementation should work on every operating system with Python version 2 or higher installed. The implementation will implement BeePAKE with the PAPKE-FO [6] protocol explained in previous sections. The project will include an example file that will show how the implementation can be used and tested. It will have two Parties and demonstrate how both share a key using the code provided.

5.1.1. Performance. Performance is important since the goal of BeePAKE is to provide people and enterprises with the ability to have a secure-by-design protocol without being limited by poor performance. The process of sharing the key should not take more than 25% time in seconds of k sugarwords. Let T be the acceptable time requirement.

$$T = 25\% \cdot k$$

As an example, a client and a server having 20 sugarwords should not take more than 5 seconds to share the session key.

5.1.2. Functional requirements. The protocol will be split into four parts each representing one step of the protocol. Each part has its own function:

- generate()
- encryption()
- decryption()
- retrieve_key()

The protocol also needs three functions of the encryption scheme of the PAPKE protocol:

- papke_generate()
- papke_encryption()
- papke_decryption()

The function generate() should use the following template:

- Description: First step of the BeePAKE protocol
- Parameter: No parameters
- Pre-condition: Protocol was started
- Post-condition: Generates a secret key, and a public key and returns the outbound message which will be sent
- Trigger: A party requests key-sharing with a server

5.2. Design

The implementation uses Python because it is easily readable and has a lot of great cryptographic libraries. As a basis for the project, modules of the GitHub repository python-spake2 created by user warner warner2016 were used. The modules used are groups.py, six.py, util.py and spake.py, it is an implementation of the SPAKE2 protocol.

The implementation splits the BeePAKE protocol into four steps and two parties A and B.

First step: The first party A generates a secret key and public key using its' password. It then stores the secret key and sends the first message to the other party B. The message includes the id of the party A sending it and the public key.

Second step: The second party B receives the messages, and parses them to get the public key and the id of the other party A. Then, he begins to generate the secrets, by encrypting a randomly generated session key for every sugarword using the encryption process of the PAPKE protocol. Party B, then sends his id and the generated secrets.

Third step: The first party A parses the received message and decrypts every key until a valid key is found. The resulting key is then stored and the party A sends the index of the valid key of the set of received keys including his id to party B.

The last step: Party B parses the received message, retrieves the index and asks the honeychecker to check if it is a honeyword. He then stores the associated session key.

5.2.1. File structure. The file structure is an important part of the project since modules refer to other files and use other modules within the project. The main folder has two main directories, one example Python file and one example password file.

- honeyword_generation (directory)
  - gen.py (python file)
  - c_pws (text file)
- sweet_pake (directory)
  - file_operation.py (python file)
  - groups.py (python file)
  - ___init___.py (python file)
  - six.py (python file)
  - sweet_pake.py (python file)
  - util.py (python file)
- client_sweetPake.py (python file)
- pw_file (text file)

The project is split into two main directories "honeyword_generation" and "sweet_pake". The "honeyword_generation" folder includes a text file "c_pws" having a fairly big list of the most common passwords and a Python file "gen.py" whose main purpose is to append or change a users' entry in a password file. The Python file has multiple honeyword generation algorithms implemented that were proposed by Rivest and Juels [1]. Some of these algorithms use the mentioned text file "c_pws". The directory "honeyword_generation" is meant to be used to create an example password file to test the implementation of BeePake.

The directory "sweet_pake" is the heart of the project, the implementation of BeePAKE. It consists of 6 files, beginning with the file six.py which is a library written by Benjamin Peterson [11]. The library makes it possible to make code of Python version 3, compatible with Python version 2 without having to declutter code.

The "___init___.py" file is used as a package initializer, which means that the current directory "sweet_pake" is treated as a Python package. Packages are treated like modules but the difference is that a module is one single file and packages can contain several modules.

The "util.py" file is a Python file which consists of various utility function which are needed for other modules in the directory.

The "file_operation" module contains functions which are related to writing or reading file operations.

The "groups.py" file is the implementation of the math behind the protocol meaning the implementation of cyclic groups and various hashing algorithms. The major part of this module was created by user warner of Git Hub Warner2013.

Finally, the "sweet_pake.py" module consists of the actual BeePAKE protocol. It implements the various functions and logic of the protocol using the modules mentioned above including libraries os, hashlib and hkdf from the standard library.

5.2.2. Classes. The different data classes:

sweet_pake.py: The module is mainly split into two classes and various Exception classes. Beginning with by presenting the error classes.

- SweetPAKEError(Exception) is used as an abstract class every SweetPAKE exception will derive from this one
- OffSides(SweetPAKEERROR) is triggered if the message comes from the same side as received
- WongGroupError(SweetPAKEERROR) is triggered when a different or the invalid group is used
- ReflectionThwarted(SweetPAKEERROR) is triggered when someone tries to reflect the message back to the user
- OnlyCallStartOnce(SweetPAKEERROR) is thrown when someone sends the wrong confirmation code

The two main classes represent both parties of a connection. The first class named SweetPAKE_Client represents the client side and the second class name SweetPAKE_Server represents the server side of the connection.

- SweetPAKE_Client
- SweetPAKE_Server

group.py: The module consists of a class named IntegerGroup and a private class named _Element. The IntegerGroup class represents a cyclic group and the _Element class represents an element of the group. The module of a third class named _Params whose purpose is to create a know big secure cyclic group.

- IntegerGroup
- _Element
- _Params

5.2.3. data structures. Now, the different data structures are presented.

Beginning with the IntegerGroup class of the group.py module:

- q being the order of the subgroup generated by the generators stored as a whole number
- p being the order of the group stored as a whole number
- Zero being the identity Element of the group stored as an object of class _Element
- Base1 and Base2 are two generators of the group stored as an object of the class _Element

Now presenting the _Element class:

- _group is the group the element is in stored as an object of the IntegerGroup class
- _e is the value of the element can have multiple types but is usually stored as a whole integer number

_Params class

- group stored as an object of the IntegerGroup class

Outside of classes as global variables:

- I1024, I2048, I3072 are three objects of the IntegerGroup class
- Params1024, Params2048, Params3072 are three object of the _Params class created with the three IntegerGroup object showed above

Continuing with data structures of sweet_pake module. Global variables:

- ClientId and ServerId are the ids of both sides which are text stored as a byte object
- DefaultParams is the IntegerGroup used from both sides are stored as objects of the class _Params

The data structure which both classes SweetPAKE_Client and SweetPAKE_Server have in common:

- password is the shared password stored as a byte object
- ida and idb being the ids of the sides which are communicating with each other stored as byte object
- params is the integer group being used as the object of the _Params class

- entropy_f the entropy used and stored as byte object

The class SweetPAKE_Server has one additional data_structure:

- database which is an associative array with all passwords as values and the usernames as values

5.3. Implementation

The section will present the most important parts of the implementation. Beginning with the gen.py module from the honeyword generation module.

Line 5-42 is a honeyword generation method proposed by the authors of Honeywords [1]. The chaffing-by-digits method takes the password and creates a new password by replacing the digits with other random digits. The function takes the password p and the number of sugarwords k as parameters. First, an array named positions, an index i and an array of sugarwords are created. Then, it traverses every character of the string p, if a traversed character is a digit, its' position is appended to the array positions. Then the next for loop is traversed k times, at every traversal the password p is added and then every digit is replaced by a random digit. The complexity of this method is quadratic. $O(n^2)$.

```
1  def gen_chaff_digits(p, k):
2      positions = []
3      i = 0
4
5      sugarwords = []
6
7      for c in p:
8          if c.isdigit():
9              positions.append(i)
10         i += 1
11
12     sys_random = random.SystemRandom()
13     for n in range(k):
14         sugarwords.append(p)
15         for x in positions:
16             rand = sys_random.randint(0, 9)
17             sugarwords[n] = sugarwords[n][:x]
18            + str(rand)
19        + sugarwords[n][x+1:]
20
21     sugarwords.append(p)
22     return sugarwords
```

Lines 27-48 implement the chaffing-by-model method explained in the honeywords section. It uses a large database of the most common pattern in passwords consisting of letters. The algorithm identifies the sequences of characters in the password that only consists of letters and then replaces them with a random word from the large database. To retrieve a random word from the large database it uses the get_rnd_pw_word() function. The algorithm below has a linear complexity $O(n)$.

```
1  def gen_by_model(p, k):
2      sugarwords = []
3
4      word_index = 0
5      words = []
6      for i in range(len(p)):
7          if p[i].isdigit():
8              if i != 0 and not p[i-1].isdigit():
9                  word_index += 1
10         else:
11             if i == 0 or p[i-1].isdigit():
12                 words.append(p[i])
13             else:
14                 words[word_index] += (p[i])
15
16     for n in range(k):
17         x = p
18         for word in words:
19             x = x.replace(word,
20         get_rnd_pw_word())
21         sugarwords.append(x)
22
23     return sugarwords
```

Lines 51-59, return a random word from the large database which is stored in c_pws.

```
1  def get_rnd_pw_word():
2      words = []
3      rnd = 0
4      sys_random = random.SystemRandom()
5      with open("c_pws") as f:
6          words = f.readlines()
7          rnd = sys_random.randint(0, len(words)-1)
8
9      return words[rnd][0:-1]
```

At lines 84 to 87, the function is given an array of sugarwords and it encrypts every element using a scrypt hash from the library hashlib. The sugarwords in the array are replaced with the encrypted words. The complexity of this function is linear $O(n)$.

```
1  def encrypt_pws(arr):
2      for i in range(len(arr)):
3          dk = hashlib.scrypt(bytes(arr[i], 'utf-8')
       , salt=b"NaCl", n=16384, r=8, p=16)
4          arr[i] = dk.hex()
```

In lines 90-93, the Fisher-Yates algorithm is implemented. It takes an array and shuffles it, such that each sugarword has a random position. The algorithm is useable for cryptographic usage and has linear complexity $O(n)$.

```
1  def fisher_yates(arr):
2      for i in range(1, len(arr)):
3          j = random.randint(0, i)
4          arr[j], arr[-i] = arr[-i], arr[j]
```

The function at lines 62-81 takes a username and a password as input. The function generates the first 3 honeywords using chaffing-by-model and then for every generated honeyword it uses chaffing-by-digit. The resulting array of sugarwords is then shuffled using the Fisher-Yates algorithm and then encrypted. The username together with the sugarwords are then written in the password file. The complexity is quadratic $O(n)$.

```
1  def append_user_to_file(name, pw):
2      k = 3
3      sugarwords = gen_by_model(pw, k)
4      for i in range(len(sugarwords)):
5          sugarwords +=
```

```
    gen_chaff_digits(sugarwords[i],
        k)

fisher_yates(sugarwords)
pw_index =
    random.randint(0, len(sugarwords)
- 1)
sugarwords[pw_index] = pw
encrypt_pws(sugarwords)

with open("pw_file", 'a') as file:
    file.write(name + ":")

    for words in sugarwords:
        file.write(word)
        if not word == sugarwords[-1]:
            file.write(":")

    file.write("\n")
```

Now, the sweet_pake package is presented, by first showing some functions of the group.py module.

As you can see in the constructor at lines 104-118. The IntegerGroup has two generators which is necessary for the BeePAKE protocol. In lines 117 and 118, it is double-checked that the two generators are elements of the group.

```
1   def __init__(self, p, q, g1, g2):
2       self.q = q
3       self.p = p
4       self.Zero = _Element(self, 1)
5       self.Base1 = _Element(self, g1)
6       self.Base2 = _Element(self, g2)
7
8       self.exponent_size_bytes =
9   size_bytes(self.q)
10      self.element_size_bits =
11  size_bits(self.p)
12      self.element_size_bytes =
13  size_bytes(self.p)
14
15      assert pow(g1, self.q, self.p)
16  == 1
17      assert pow(g2, self.q, self.p)
18  == 1
```

At lines 204-219 is the operation of the group implemented being a multiplication. It always makes sure that no other type is calculated except an element object type, if it does not raise an error in such cases, this would be a vulnerability that could be exploited.

```
1   def _exp(self, e1, i):
2       if not isinstance(e1, _Element):
3           raise
4     TypeError("E^i requires E be an element")
5       assert e1._group is self
6       if not isinstance(i, integer_types):
7           raise
8     TypeError("E^i requires i be an integer")
9       return _Element(self, pow(e1._e, i % self.
    q, self.p))
10
11  def _elementmult(self, e1, e2):
12      if not isinstance(e1, _Element):
13          raise
14    TypeError("E1*E2 requires E1 be an element")
15      assert e1._group is self
16      if not isinstance(e2, _Element):
17          raise
18    TypeError("E1*E2 requires E2 be an element")
19      assert e2._group is self
20      return _Element(self, (e1._e * e2._e) %
    self.p)
```

The function at lines 172-174 does a xoring between two byte sequences.

```
1   def xor(self, bytes1, bytes2):
2       xor = [a ^ b for a, b in zip(bytes1,
    bytes2)]
3       return bytes(xor)
```

The function at lines 38-50 takes as several parameters and hashes to compute two exponent which will be used for the protocol.

```
1   def key_to_2exponents(R, y1, y2, k,
    exponent_size_bytes, q):
2       ikm = R.to_bytes() + y1.to_bytes() + y2.
    to_bytes() + k
3       h = Hkdf(salt=b"", input_key_material=ikm,
    hash=hashlib.sha256)
4       info = b"SweetPAKE FO random exponents"
5       seed_rs = h.expand(info, 2 *
    exponent_size_bytes + 32)
6
7       (r1_bytes, r2_bytes) = splitInHalf(seed_rs)
8       r1_number = bytes_to_number(r1_bytes)
9       r2_number = bytes_to_number(r2_bytes)
10      r1_exponent = r1_number % q
11      r2_exponent = r2_number % q
12
13      return (r1_exponent, r2_exponent)
```

The function at lines 65-75 returns a vector of random session keys for each entry in the password file.

```
1   def secrets_to_array_of_keys(k, y1, Y2, n,
    length_bytes):
2       ikm = y1.to_bytes() + Y2.to_bytes() + k
3       h = Hkdf(salt=b"", input_key_material=ikm,
    hash=hashlib.sha256)
4       info = b"SweetPAKE vector of keys"
5       length_seed_rs = length_bytes
6       seed_rs = h.expand(info, n*length_seed_rs)
7
8       length_key = len(seed_rs) // n
9       keys = []
10      for index in range(n):
11          keys.append(seed_rs[index*n:index*n+
    length_key])
12
13      return keys
```

Now continuing with the sweet_pake module. Looking at the client side at lines 78-93, the function generates the secret key and the public key and stores the secret key. After that, it creates a variable named outbound_message which includes the id of the side, the size of the username, the username and the public key.

```
1   def gen(self):
2       #gen function
3       group = self.params.group
4       self.random_exponent = group.
    random_exponent(self.entropy_f)
5       self.y1_elem = group.Base1.exp(self.
    random_exponent)
```

```
        self.y2_elem = group.Base2.exp(self.random_exponent)
        Y2_elem = self.y2_elem.elementmult(group.password_to_hash(self.pw))

        #self.outbound_message = (self.y1+self.Y2) <-- apk
        y1_bytes = self.y1_elem.to_bytes()
        Y2_bytes = Y2_elem.to_bytes()
        self.outbound_message =  y1_bytes + Y2_bytes

        username_size = len(self.username).to_bytes()

        outbound_id_and_message = self.side + username_size + self.username + self.outbound_message

        return outbound_id_and_message
```

The server side reacts to the outbound message with the function at lines 198-232. The function parses first the message extracts the username and the apk. He then generates a vector of session keys using the secrets_to_array function. Then encrypts every sugarword using the encryption method used in PAPKE FO [6] explained in the PAPKE section. The implementation of the method is at lines 234-252 it produces several secrets which are then included in an outbound message variable together with the servers' id and the byte size of one secret.

```
def enc(self, inbound_message):
        #parse inbound_messahe
        self.inbound_message = self._extract_message(inbound_message)

        #get username from message
        username_size = int.from_bytes(self.inbound_message[:1])
        self.working_with = self.inbound_message[1:username_size+1].decode('utf-8')

        self.inbound_message = self.inbound_message[username_size+1:]

        apk = self.parse_apk(self.inbound_message)
        group = self.params.group
        y1_elem = group.bytes_to_element(apk[0])
        Y2_elem = group.bytes_to_element(apk[1])
        client_pw_array = self.database[self.working_with]

        #PRF - get array of keys
        k = os.urandom(32)
        self.arr_K = group.secrets_to_array(k, y1_elem, Y2_elem, len(client_pw_array), 32)
        #self.arr_K = []
        self.ciphers = []

        #enc_function
        for i in range(len(client_pw_array)):
            gen_ciphers = self._papke_enc(group, y1_elem, Y2_elem, client_pw_array[i], self.arr_K[i])
            self.ciphers.append(gen_ciphers)

        #shuffle cipher array
        rp_ciphers, self.pmap = fisher_yates(self.ciphers)

        #message
        #self.outbound_message = c = (c1, c2, c3)
        self.outbound_message = b"".join(rp_ciphers)
        outbound_sid_and_message = self.side + len(rp_ciphers).to_bytes() + self.outbound_message
        return outbound_sid_and_message

def _papke_enc(self, group, y1_elem, Y2_elem, pw, session_key):
        #session_key = os.urandom(32)
        #self.arr_K.append(session_key)

        pw_to_hash = group.password_to_hash(bytes.fromhex(pw))
        y2_elem = Y2_elem.elementmult((pw_to_hash.exp(-1)))

        random_exponent = group.random_exponent(self.entropy_f)
        R_elem = group.Base1.exp(random_exponent)

        (r1, r2) = group.secrets_to_hash(R_elem, y1_elem, y2_elem, session_key)

        c1 = group.Base1.exp(r1).elementmult(group.Base2.exp(r2))
        c2 = y1_elem.exp(r1).elementmult(y2_elem.exp(r2)).elementmult(R_elem)
        c3 = group.xor(hashlib.sha256(R_elem.to_bytes()).digest(), session_key)

        C = c1.to_bytes() + c2.to_bytes() + c3

        return C
```

At lines 97-126 is the decrypt function it first parses the message and then tries to decrypt every secret using the decrypt function of the PAPKE protocol at lines 128-148. If the is decrypted successfully it stores the decrypted key and returns a message including the id and the index of the successfully decrypted key.

```
    def dec(self, inbound_message):
        #parse message
        group = self.params.group
        self.inbound_message = self._extract_message(inbound_message)

        len_ciphers = int.from_bytes(self.inbound_message[:1])
        if len_ciphers < 0:
            raise ValueError("Invalid size")

        self.inbound_message = self.inbound_message[1:]
        ciphers = self._parse_array(self.inbound_message, len_ciphers)
        index = -1

        #dec
        for i in range(len(ciphers)):
            session_key_computed = self._papke_dec(group, ciphers[i])

            #checks if decryption is successful
            if session_key_computed != -1:
```

```
                index = i
                break

        if index == -1:
            raise ValueError("Could not decrypt")

        self.session_key = session_key_computed

        self.second_outbound_message = i.to_bytes
    ()

        return self.side + self.
    second_outbound_message

    def _papke_dec(self, group, cipher_tuple):
        (c1, c2, c3) = self._parse_key(
    cipher_tuple)
        c1 = group.bytes_to_element(c1)
        c2 = group.bytes_to_element(c2)

        #computation
        R_elem = c2.elementmult(c1.exp(-self.
    random_exponent))
        session_key_computed = group.xor(c3,
    hashlib.sha256(R_elem.to_bytes()).digest())
        (r1, r2) = group.secrets_to_hash(R_elem,
    self.y1_elem, self.y2_elem,
    session_key_computed)

        if c1.to_bytes() != group.Base1.exp(r1).
    elementmult(group.Base2.exp(r2)).to_bytes():
            return -1

        return session_key_computed
```

After that, the server retrieves the key using the function at lines 258-264.

```
    def retrieve_key_ask_honeychecker(self,
    inbound_message):
        self.second_inbound_message = self.
    _extract_message(inbound_message)
        index = int.from_bytes(self.
    second_inbound_message)
        original_index = self.pmap[index]
        self.session_key = self.arr_K[
    original_index]

        # todo verify honeychecker

        return self.session_key
```

5.4. Improvements

The implementation is 90% finished but it still can be improved. List of improvements:

- Integration of honeychecker
- Improved benchmark system
- Unit tests
- Improved comments
- Refractor according to one style guide
- Include MAC authentication
- Improve Error Handling

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.

[11] B. Peterson, Utilities for writing code that runs on python 2 and 3, https://github.com/benjaminp/six, 2010.