Thesis Dissertation

KBCRACKER: CRACKING KEYBASE-STYLE CREDENTIALS

Maria Stylianou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2019

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

KBCracker: Cracking Keybase-style Credentials

Maria Stylianou

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2019

Acknowledgments

I would like to take the opportunity to express my appreciation to my thesis advisor Dr. Elias Athanasopoulos, for his priceless advice and guidance during this project. His continuous help was critical for overcoming all the challenges and difficulties I have faced from the start of this thesis until the end. My sincere thanks for giving his guidance throughout the research of this thesis.

Summary

We have created a cracking tool, *KBCracker*, for Keybase's stored credentials. The Keybase's server does not hold the users' passwords in clear text or in encrypted form as most services do. The server authenticates users without having actually their passwords being revealed to it. Specifically, Keybase's server has the users' EdDSA public key and used it in order to authenticate them. The idea behind our tool is to compute the public key of leaked credentials (username,salt) using a set of common passwords. We use the Keybase's API and documentation for the creation of the public key, named *k-id*. Moreover, our tool is based on the *Dictionary attack* method.

In particular, Keybase uses Elliptic Curve Cryptography (ECC) in order to have a successful user authentication, as the login process is password-less. Keybase login system seems like a text-based authentication process but it is not. Keybase's users need to give their credentials (username & passphrase) as usual, but an encryption key is derived from their passphrase. The key that is derived is the user's *private key*. Moreover, using this *private key*, a *public key (k-id)* is generated. Those keys constitute a pair and cooperate in order to create a signature in terms of ECC. All these necessary steps are done at the client's side, as a result the server authenticates users without any knowledge of how their passwords look like.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Advances in cryptography have given rise to new applications, especially to the ones that utilize the network for exchanging securely information. Nonetheless, a core and vital process of communication, such as user authentication, has still received little benefit. Despite the fact that several protocols [6, 7, 11, 26] leverage modern cryptographic concepts for strengthening user authentication, in practice they are hardly used.

Today, the most established form of user authentication is based on text-based passwords. Services utilize cryptographic hash functions [4, 24] for storing the passwords in the form of a digest, and not in plain. This can delay an attacker that has leaked the password database [18] from figuring out the actual passwords, nevertheless, this very-well established scheme is associated with further problems. Key among them is the fact that the service *knows* the user password and needs to receive it *in plain* every time the user authenticates.

In short, this means that the user needs to *trust* the service not only for their in-between interaction, but also for their shared credentials. Since password re-use is common [13], users need to trust that services are not going to use the received credentials to impersonate their clients to other services.

1.1 Contributions

This thesis project makes the following contributions.

- We discuss and explore a first instance of user authentication based on public-key cryptography, which is used in production code and is offered by Keybase.
- We build a tool for cracking stored credentials based on public-key cryptography, and we compare it with similar tools that crack leaked cryptographic digests of passwords.

1.2 Organization

The rest of the thesis is organized in chapters as follows. In Chapter 2, we discuss the background of knowledge that is needed, in order to understand the necessity of authentication and security, including in detail how Public Key Cryptography, Elliptic Curve Cryptography and Keybase work. In Chapter 3, we discuss the Text-based Authentication and we stress the problem that service's servers know the user's password, and we emphasize the Keybase authentication. What is more, in Chapter 4, we explain in detail how our tool is implemented, and in Chapter 5, we evaluate the Keybase's login protocol and authentication methods in terms of security and performance. Finally, we discuss about Related work in Chapter 6 and we give some conclusions in Chapter 7.

Chapter 2

Background

Contents

2.1 Public Key Cryptography

Before Public Key Cryptography (PKC) all crypto systems were using symmetric algorithms for cryptographic operations [25]. In symmetric cryptography a symmetric key is used both for encryption and decryption. This special secret key is necessary to be exchanged between the communication parties via secure channels. The requirement of a secure channel was not very strong and shortly became unmanageable. Furthermore, the need for all communicators to have access to the special secret key is an essential disadvantage of the symmetric cryptography. Alternatively, PKC is an asymmetric encryption scheme that uses a pair of two keys, a public key and a private key. PKC scheme

was an innovation for cryptography as it was the first system that provided security in transformation of data without a shared special key but with key exchange.

Both the public key and the private key have their roles in the PKC scheme. The public key is used for encryption and the private key is used for decryption. Both keys are extremely big (1024-bits) due to that the public key is stored on digital certificates (DCs) and the private key can be stored in the hardware, the software, or even in the operating system. The reason that the keys are not stored together but they are stored in this way is that the public is widely known and the private key is known only by its owner. A PKC example is RSA algorithm. RSA's security is based on the difficulty of factoring. RSA uses very large prime numbers (p, q) for the generation of the public and the private key. It is impractical and computationally hard for an attacker to calculate and compromise the private or the public key without knowing p and q.

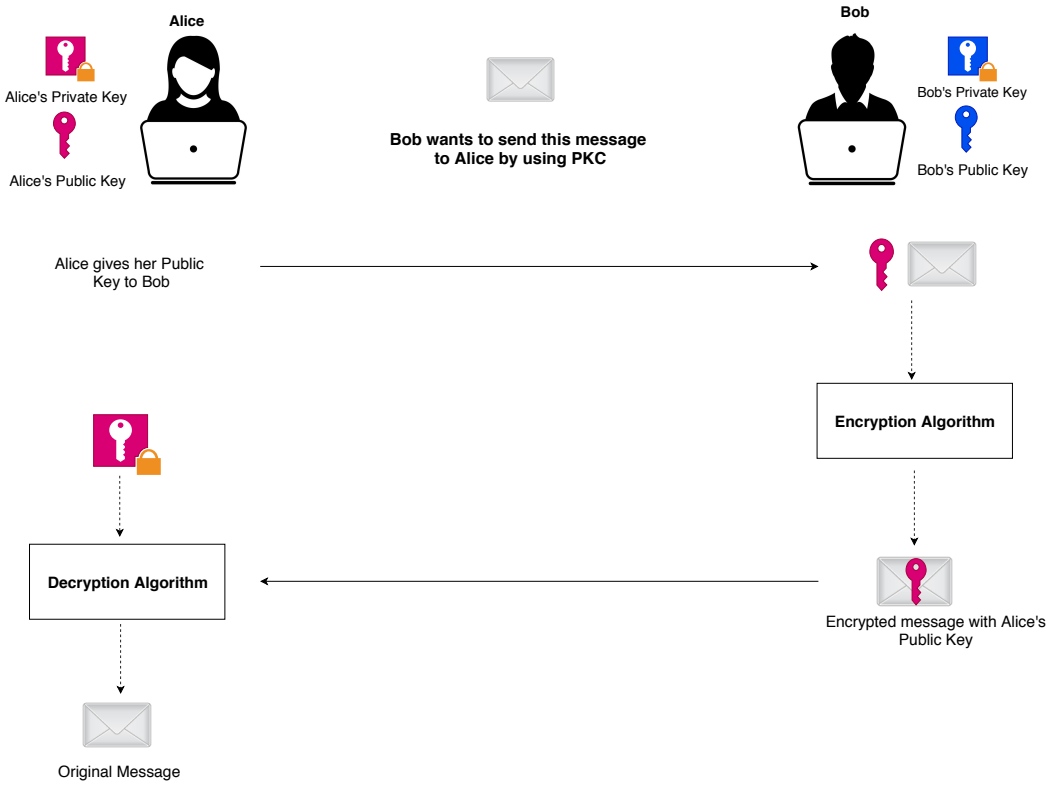

Figure 2.1: This figure shows the high idea of a communication that is based on PKC.

The main idea of a PKC scheme is that the transmitter has the public key of the receiver and uses it for the message's encryption. Therefore, the receiver gets the encrypted message and uses their own private key to decrypt it and get the original clear text message. The high idea relies on the fact that encryption is easy because anyone has the public key of the others but the decryption is hard. This is because no one can decrypt

the message, except of the owner of the public key that was used, who is also the owner of the private key that can decrypt the encrypted message. As a result, the one that the message is destined to is the only recipient that can decrypt it and get the original form of it, and no one else can get the clear text.

2.1.1 Key Generation

In Public Key Cryptography (PKC) the most expensive and important thing is the generation of the public and the private key. Although these two keys have a mathematical relationship, they are not identical twins. Therefore, it is not possible to compute the private key when the public key is known. Because of this, public keys can be shared without any risk but it is important that the private keys are kept secret and they are only known by the owner.

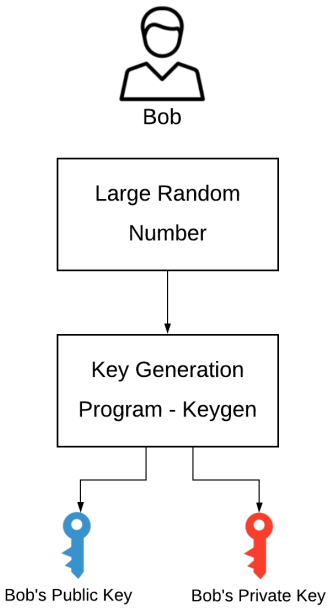

Figure 2.2: This figure shows the high idea of a the key generation based on RNG or PRNG.

For the generation of this pair large integer are used, preferably primes are being used. Moreover, these keys can be generated randomly using random number generators (RNG) or pseudorandom number generators (PRNG). PRNG is an algorithm that produces data that are not truly random. The result of a PRNG is determined by an input that is called *seed*. Furthermore, the result of the PRNG can be reproduced if the state of the PRNG is known. Otherwise, RNGs produce numbers that are truly random. Their production seems like the rolling of a dice, a coin flipping and the shuffling of cards. For that reason the production of RNG requires very expensive infrastructure, a lot of work and time.

2.1.2 Digital Signatures

Digital Signatures are used like physical signatures. The author of a document signs the document at the end to show that it is written by him and no one else. A digital signature has almost the same functionality and usage in the digital word. Basically, the author of a document generates a digital signature and appends it to the document to determine that the document is produced by him. Moreover, Digital Signatures offer authentication, non-repudiation and integrity. These aspects are very important in the digital word, both in communication and data transport.

Assuming the scenario that Bob wants to send a document to Alice and they choose to make this transmission using Digital Signatures. Bob writes the document in clear text, calculates the document's digest using a Hash Algorithm (sha256) and then encrypts the digest with his private key. As a result, only Bob's public key can decrypt the encrypted digest. The encrypted digest is the Digital Signature of the document. Bob appends on the document the digital signature and he sends the *sign document* to Alice. However, the document is not encrypted, so Alice can read it without any decryption process. However, Alice wants to verify that the document is written by Bob and it has not been altered during the transmission by an unauthorized entity. For the verification, Alice uses the digital signature that is appended on the document. Alice has Bob's public key, so she can decrypt the digital signature using the same algorithm that Bob used but for decryption. As a result, Alice gets the document's digest. Finally, Alice calculates the document's hash value, without the digital signature, using the same hash algorithm that Bob used. If the decrypted digest is equal to the calculated digest, it means that the document is the right one and that it has not been modified during transmission.

In this scenario, Bob and Alice use Digital Signature instead of document encryption to make sure that the message has not been altered during transmission. Even though encryption can hide the contents of the document, it may be possible for an attacker to change the encrypted document without Bob and Alice understanding it. Digital signatures prevent this, for the reason that any change in the document after the sign of it invalidates the signature. Since there is no efficient way to modify the document and its signature and produce a new document with a valid signature. As a result, this scenario shows how digital signature scheme provides the aspect of integrity.

Additionally, digital signatures provide the aspect of non-repudiation. By this property, an entity, Bob, who has signed something, a document or a message, cannot at a later time deny the fact that he has signed it.

Finally, Alice can use Bob's digital certificate in order to verify that the public key really belongs to Bob. Digital certificate includes Bob's public key and his name and it is is digitally signed by the trusted Certificate Authority (CA). With this, Alice can verify

that there is not a man-in-the-middle that introduced themselves as Bob.

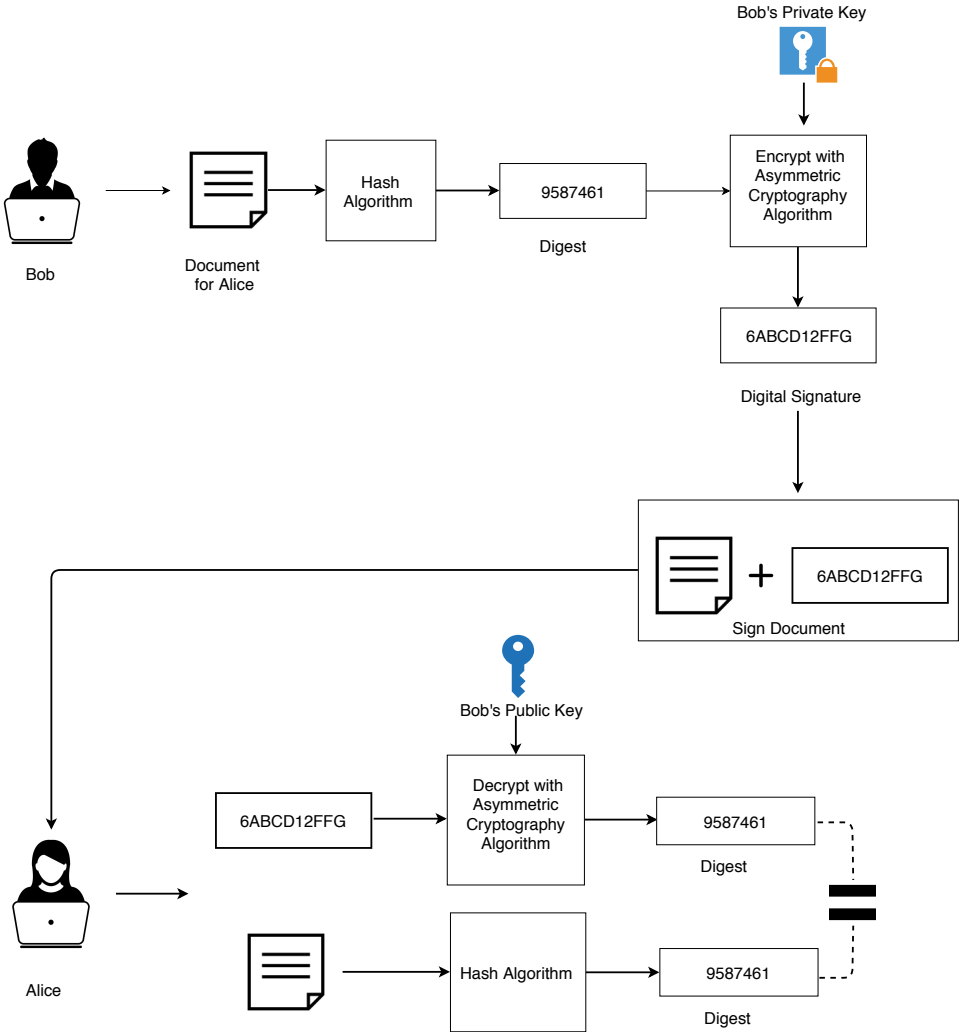

Figure 2.3: This figure shows the high idea of how Digital Signatures work in the digital word.

2.2 Elliptic Curve Cryptography

2.2.1 Overview

Elliptic Curves Cryptography (ECC) [16, 20] is based on the mathematical background and algebraic structure of elliptic curves over finite fields. The security of ECC is based on the inability to compute the multiplicand given and the product points. also, the important benefit that ECC provides is the ability to use a smaller key but keep a strong level of security. Using ECC is a way to create faster, smaller and efficient cryptographics keys.

2.2.2 Trapdoor Functions

All cryptosystems use a trapdoor function for their implementation and security. The trapdoor function is a function that is easy to be computed on one way but it is very hard, mathematically impractical, to compute its back direction. Trapdoor functions have a similar principle with one-way functions. However, in trapdoor functions there is a special key, which if its is known, it is possible and easy to reverse and calculate the back direction. On the other hand a one-way function is not reversible, because there is no back direction. In other words a trapdoor function, as the name indicates, make it is easy to fall down in a trap but it is very hard to jump out of it, unless you have the special knowledge of a key which is the *staircase*.

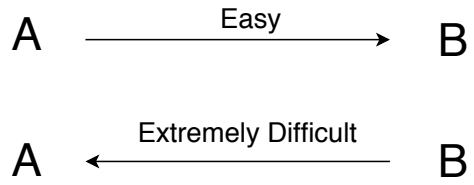

Figure 2.4: This figure shows the high idea of how trapdoor functions work.

For example, RSA cryptosystem uses a trapdoor function that is based on the concept of Prime Factorization. Especially, in RSA it is easy to compute n if you know p and q, $n = p * q$. Nevertheless, the inverse calculation is mathematically hard. The special knowledge for inverse calculation is p or q or both of them. It is extremely hard to calculate p and q from factoring n since they are large prime numbers (1024 bits).

Similarly, ECC uses a trapdoor function, but this function is based on Point Multiplication. The equation that is presented as the trapdoor function of the ECC is $A = p * B$, in which A is the end point that is produced by multiplying start point B by p. It is simple and easy to calculate A, the one direction of the equation, but it is difficult to find the n when both A and B are known. The special knowledge that is needed for the inverse

calculation is the n, $n \in Z$.

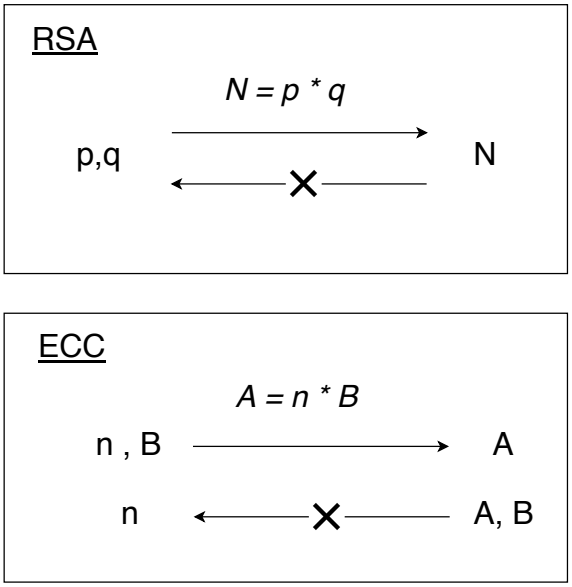

Figure 2.5: This figure shows the trapdoor functions of the RSA and ECC cryptosystems.

2.2.3 Point Addition, Point Doubling and Point Multiplication

EC cryptosystems is based on an elliptic curve equation. There are a lot equations, therefore each cryptosystem chooses one to use. However, the points that are valid for the equation are points on the curve. Generally, cryptosystems that are based on ECC use the form, $y^2 = x^3 + ax + b$.

Always, two points on an EC intersect another third point, except the points that are vertical. For instance, if we get a point P and a point Q on an EC then a third point R, is generated. Assuming that we have to add point P with point Q, then we reflect the third point R to get the point R'. The reflected point R' is the result of the addition of point P and point Q. The process that has been described is named Point Addition.

On the other hand, it is possible to add a point with itself. This method in ECs is named Point Doubling. In Point Doubling there is no second point so we can not generate the third intersected point. Instead, we *draw* the tangent of point P and then we get its intersect point to be R. Similarly with Point Addition, we reflect the R to get the point R' which is the result of the Point Doubling, $R' = P + P = 2P$.

As we can see in the figure 2.6, point R is vertical to point R', so we can not apply neither Point Addition or Point Doubling. Adding two vertical points is an undefined procedure. If we try to add two vertical points there will never be a third intersected point. Basically, the result of the addition of two vertical points is infinity, $R + R' = \infty$.

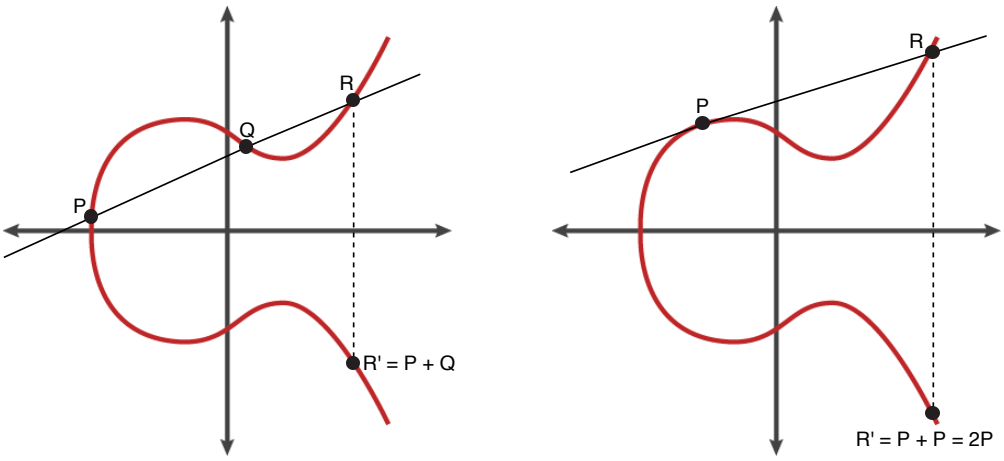

Figure 2.6: This figure shows how Point Addition and Point Doubling work.

Furthermore, Point Multiplication operation is based on how Point Addition and Point Doubling work. In Point Multiplication, the operation of Point Addition is applied for k times, where k is the factor of the multiplication. For instance, when we have to multiply the point P three times, $3P = P + P + P = 2P + P$, we have to perform sequentially Point Doubling and Point Addition. The figure 2.7,below, shows the steps that are performed to calculate $3P$. First, the point that corresponds to $2P$ is calculated using Point Doubling, $2P = P + P$. As point P is known,by using the tangent we can find the point $2P$. For the second step, since point $2P$ and point P are known, Point Addition operation can be used to calculate the point that corresponds to $3P$, $3P = 2P + P$.

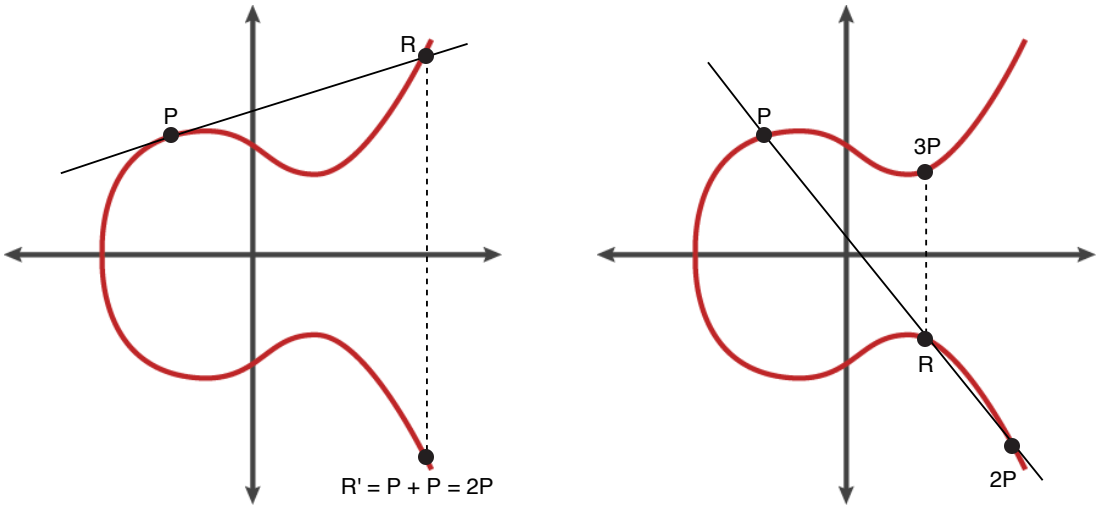

Figure 2.7: This figure shows how to calculate $3P = 2P + P = P + P + P$ using Point Multiplication operation in terms of Elliptic Curves.

Another example is the quadruple of the point P. The steps of Point Multiplication in terms of ECs for this example are shown in figure 2.8. The Point Multiplication for calculation $4P$ is a follow up of the previous example, in which the point that is calculated corresponds to $3P$. Since we have both points $3P$ and P, Point Addition operation can be used to calculate $4P$, $4P = 3P + P$. Furthermore, $4P$ can be calculated by using the Point Doubling operation, $4P = 2P + 2P$.

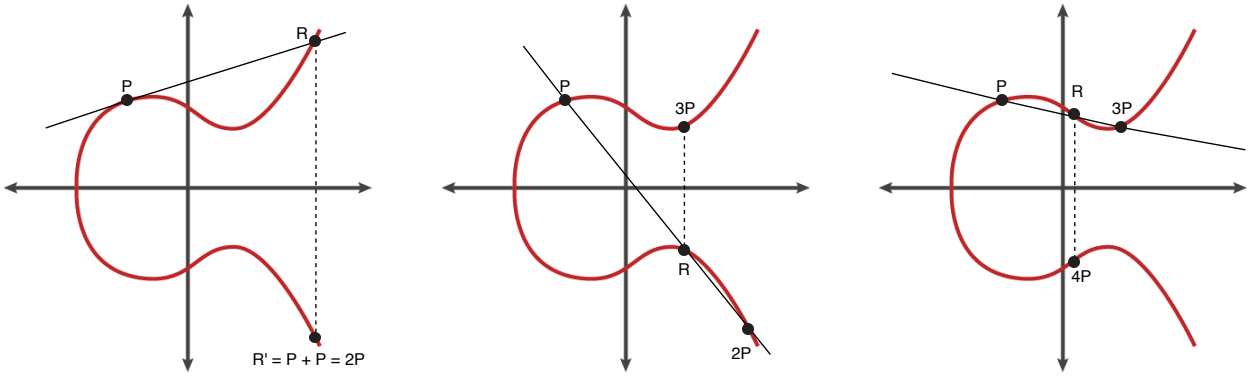

Figure 2.8: This figure shows how to calculate $4P = 3P + P = P + P + P + P$ in Point Multiplication operation.

To conclude, as its is explained and shown by the figures 2.7,2.8, there are a lot of *jumps* in the graph for the performance of some basic operations.Although in the examples above, small integer numbers were used, complicated steps are performed in order to calculate the final result. ECC gets its strength from the impracticability to divide these multiplications and find the specific point that has been multiplied in order to give the result of the multiplication, which is possible in linear algebra. For example, if we have point P and point $4P$ but we do not know that the number of multiplications of the latter is 4, we have to calculate all the possible multiplications until we get the $4P$. When the numbers are small this seems easy, but the numbers that are used in ECC are considerably large.

2.2.4 Key Exchange in Elliptic Curve Cryptography

Elliptic Curves are based on the Discrete Logarithm (DL) [22] problem. In cryptosystems that use EC, the DL problem helps in the key generation process. If we assume thet we have an equation, that presents an EC named E, a point P on the EC E and another point on E, T, the point T is the result of the performance of ECs Point Multiplication on point P, $T = dP = P + P + P... + P$. In ECs cryptosystems, T is the public key and d is the private key. The description above is named Elliptic Curved Discrete Logarithm Problem (ECDLP), but the d is found from DL problem.

Key exchange using elliptic curves is named Elliptic Curve Diffie-Hellman key exchange (ECDH). In ECDH, it is necessary to determine the Domain Parameters. The Domain Parameters are a prime number p, the elliptic curve E and a primitive element P which is a point on the curve E. The difficulty in determining the Domain Parameters is to find the elliptic curve that satisfies the cryptosystem's security properties.

For instance, Alice and Bob want to exchanged their keys using ECC. Suppose, they have decided the curve E and the primitive point P. Afterwards, Alice chooses her private key, a. Similarly, Bob chooses his private key b. Both private keys are two large numbers (256 bits). To compute their public keys both apply Point Multiplication on primitive point P. Alice's public key, A is computed by the form $A = aP$. Similarly, to compute Bob's public key, Point Multiplication is applied with factor his private key, $B = bP$. Both public keys, are points on the elliptic curve E.

Alice and Bob exchange their public keys. Both, Alice and Bob calculate their joint secret, W. For the calculation of the joint secret point, they performed Point Multiplication using their own private keys and the public key that they received. The joint secret point W that is computed is the same key. Alice calculates $W = aB$ and Bob $W = bA$, where $A = aP$ and $B = bP$. So, Alice computes $W = a(bP)$ and similarly Bob computes $W = b(aP)$, so both, Alice and Bob calculate the same *joint secret point*.

The details that are publicly known are the curve E, the prime p, the primitive point P and both public keys of Alice and Bob, A and B. If an attacker wants to compromise the ECDH protocol, he has to compute the joint secret that Alice and Bob compute, $T = aB = bA = abP$. The attacker has to solve one of this logarithm problems : $a = \log_p A$ or $b = log_p B$.

2.2.5 Elliptic Curve Digital Signature Algorithm

In this part we explain how Elliptic Curve Digital Signature Algorithm (ECDSA) [22] works. Specifically, the way that the contracting keys are generated, the algorithm of the generation of signature and the verification of signature are explained, in the scenario that Bob wants to send a sign message or document to Alice.

Firstly, the generation of keys is explained. Key generation in ECDSA is done by Bob, the transmitter. In key generation there are three steps. The first step is to find an elliptic curve E. The selection of this curve should be with care, as each curve has its own cryptography properties. For that reason it is an important step to select the appropriate curve for the scheme. By choosing an elliptic curve the parameters of the curve p, a, b, a point A and a q are selected, resulting in having $E : y^2 = x^3 + ax + b \mod q$. The point A is the generator point of a cyclic group of prime order q. The second step is to choose a random number d, $0 < d < q$, which is the private key. The final step of the key generation

is to calculate the point B, $B = dA$. Consequently, Bob has two keys, a public key and a private key. Afterwards, Bob sends to Alice the public key.

$$\textit{Private key}\text{: } k_{pr} = d$$
$$\textit{Public key}\text{: } k_{pub} = (p,a,b,q,A,B)$$

In the sign process, Bob has to compute the hash value of the document,$hash(m)$, and follows the above steps. The first step, is to choose a random key, k. This key is a integer short live key with $0 < k < q$. Second step, is the calculation of a point R , $R = kA$. From this step, Bob has the first member of the signature's pair "named" r, which is the abscissa of the point R, $r = x_R$. The last step, is the calculation of the second member of the signature's pair s, $(hash(m) + dr)k^{-1} \mod q$. Finally, Bob sends to Alice the message m and the pair *(r,s)*. The signature, that was mentioned above, in ECDSA, made of a pair of integers *(r,s)*.

The verification of the signature is done by Alice. Alice gets the document, m with the signature *(r,s)*. For the verification Alice has to compute the values w,u_1,u_2 using the parameters that are retrieved from Bob.

$$w = s^{-1} \mod q$$
$$u_1 = w \cdot hash(m) \mod q$$
$$u_2 = w \cdot r \mod q$$

Afterwards, Alice computes the point P using the values u_1, u_2 and the points A and B from public key. The computation of P is the following: $P = u_1A + u_2B$. Then, Alice has to check the validity of the signature, and if the x-coordinate of point P has same value with the $r \mod q$, then the signature is valid.

2.2.6 Security

In this part we refer to some advantages of ECC over RSA [5,14] and vice versa, in terms of security. Also, we denote some attacks that these cryptosystems can suffer from.

RSA cryptosystem is the most known and widely used public key cryptosystem, especially, RSA is commonly used for digital signatures and key transport. On the contrary, Elliptic curves cryptosystem is under-research and not so widespread in use. Elliptic Curve Cryptography is used from new platforms and mainly from embedded platforms (like Keybase). The usage of ECC is mainly for key exchange, digital signatures and encryption. It is important to highlight that both cryptosystems base their security in mathmatical background. ECC on Elliptic Curvers algebra and RSA on factoring large prime numbers.

The attacks that RSA might suffer from is Protocol Attacks, Mathematical Attacks and Side-Channel Attack. In ECC the crucial step is the selection of the curve, if curve is chosen with attention and care the attacks that can affect ECC are ones using the generic Discrete Logarithm algorithms.

We are not going to emphasize and explain a lot about how the attacks work but we will be discussing characteristics of the performance of these cryptosystems and how these affect their security and usage.

Lets start with a small review of the advantages of each system. Firstly, if RSA uses small keys, the signature verification is faster than the ECC. However using small size keys in RSA it is weakened. As a result, RSA seems to have a small advantage over ECC in performance on the verification process but RSA needs to be using large keys. What is more, ECC uses shorter operands than RSA (ECC uses 160-256 bits, RSA uses 1024-3072). That makes the ECC to generate briefer signatures and cipher-texts. The importance of this advantage is that in ECC we can generate a key pair from anything. We can have the same level of security with RSA by using ECC with a small "seed". For instance, we can generate a strong key pair for crypto operations via characters, numbers or special symbols in ECC, and the EC's functions will transform them into a point on the curve. In contrast, RSA only uses large prime numbers for key generation.

In conclusion, ECC seems to be the next generation of cryptosystems against RSA cryptosystems. There are a lot of researches [3, 15] about the ECC and they promise to public key cryptography a new branch of evolution and development.

2.3 Keybase

Keybase is a relatively new application launched in February 2014. Users can use Keybase as command line tool, as website and as client. Keybase seems like other social media systems, but it is not. Keybase is based on public-key cryptography. Furthermore, Keybase offers the features of chat and storage. The interesting part of Keybase is not that is a messaging platform but that is an application which offers an end-to-end encrypted chat system and also, an end-to-end encrypted file-system.

Moreover, Keybase can be described as a system for users to use a public encryption key (generate or upload) to verify their online identity with a high degree of certainty. Specifically, users can verify their public key in Keybase through Twitter, Facebook, GitHub, Reddit, or Hacker News. Keybase urges users to "connect" their Keybase account with as many platforms as possible. As a result, if an attacker wants to impersonate someone else using a fake public key, they would come up against a wall. The idea behind this system is to map the users' social identities like a key directory. In this sense, Keybase is a database of these proofs that verify the public identity.

Figure 2.9: The Keybase interface.

We studied keybase for the reason that it is a new and promising platform in terms of security. Keybase uses techniques that are not very famous in usage. For example, keybase uses ECC signatures for sign and verification. Besides the way that Keybase maps users' identity, it is interesting to note the protocol that Keybase uses for authenticating users in login process. Keybase offers a user-friendly environment and the user has no idea how the whole process for their authentication is implemented. When a user needs to authenticate with Keybase, a passphrase is used to derive a cryptographic key that will carry out an EdDSA signing and verification process . The authentication process is implemented in the client's side (web browser, application) and not on the server side. This authentication process might resembles a typical password-base authentication process but it is much more than that.

Chapter 3

Text-based Password Authentication

Contents

3.1 Overview

It is widely known that, passwords should not be stored in clear text [1]. If an attacker steals the database of passwords, then the attacker will know everything and this means that they will not need any extra time or work to recognize the passwords. As a result, the attacker can use this database as they like. For example, the attacker could connect to a system to imitate someone else in order to offend him publicly. Similarly unsecured, it is storing the passwords based on encryption by using a special key, which might be available on the server. As a result, this key can be reached from the attacker and they can use it to decrypt the encrypted database.

In order to create a security storage for users' passwords, systems usually use one-way encryption. One-way encryption uses a hashing algorithm, which always produces the same result for a given password but not the same result for different passwords. Moreover it is impossible to find the password that has generated the result when you only have the result. Because of those statements, if an attacker has the database with the usernames and their corresponding encrypted passwords, it is computationally hard to compromise it and figure out the password. The process of how these systems authenticate the users' credentials is that when the users want to log in, they give their username and their password. Then the server gets the clear text of password and makes all the necessary procedures to encrypt it. Afterward, the server compares the stored encryption of the

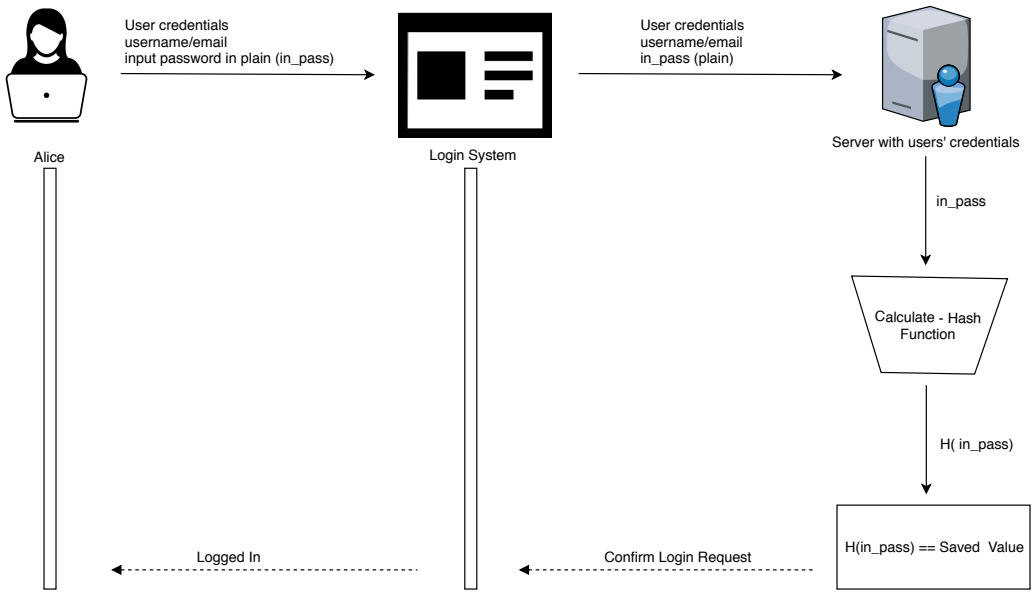

Figure 3.1: This figure shows the high idea behind the ordinary secure Password Storage and Authentication.

password associated with the user's login name with the calculated encrypted password. A match succeeds and the user is an authenticated user with access in the system. But if the mismatch fails and user is not able to use the system with the log in rights.

3.2 Research Problem

Why does a server need to know our passwords? When a user sends a login request for a system, then this request goes to the server. The server gets the password in plain text and then it makes all the calculations to turn the password into a long string of letters and numbers to keep it hidden. In addition, there are a lot of techniques that can make it difficult for the attacker to compromise a leaked database. One technique is to use a different and unique *salt* for each password, and even if the salts are stored on the same servers, it will be very hard to find those salted hashes in the *Rainbow Tables* [2, 12, 19]. This technique increases greatly the processing time for the attacker to find the passwords. However, the point that the server knows our passwords in salt technique still exists, so our passwords are not truly secret. We emphasize, at this point, that the server knows our passwords in order to stress that the attacker can be the service's provider. Even though an external attacker must have strong capabilities in order to compromise a server, a system already has our passwords. With this, the system that we *trusted* can use our passwords in order to match our digitally identity with other systems, and it might use our credit accounts for illegal purchases. Therefore, we are looking for a technique that is keeping

the plain password truly secret. Thus only the owner of a password knows their password and this will help to protect the users from having a different password for each service that they use.

3.3 Keybase Authentication

As explained above in the sections 3.1 and 3.2, in many systems and applications the password is sent to the server in plain text and then the server concludes the authentication process. Those approaches have a weak point that the server knows and stores the passwords in its side. In this type of methods, an opportunity for an attacker to steal and compromise the user's passwords is present. Users' credentials, and mainly users' passwords, are very sensitive data for the system and especially valuable for the user.

Keybase has a different way to authenticate the users in the log in process which starts from the sign up process. Keybase's sign up method starts with the generation of the salt which is used for stretching the user's passphrase. This stretching creates a passphrase stream of 256 bytes, which its slices create two parameters, a parameter pwh and an EdDSA private key. Both parameters are required for the sign up but the first parameter pwh is ignored at the server. However the EdDSA private key is important for the server and for the log in procedure. With this private key, a public key is created forming a pair. The public key is sent to the server as a *k-id*, with the Keybase's key ID format [17]. Up to this point, if a user wants to log into Keybase, then the below sequence of steps is going to be the following.

First, the user gives their username or email and their passphrase. Then the Keybase's client sends to the server the username in order to retrieve the salt that was generated in the sign up method. With the salt that is retrieved, the passphrase is going to be stretched and the *passphrase stream* will be produced. The passphrase stream is sliced to the last 32 bytes, which are the EdDSA private key. The next step of calculation is to generate the pair of the EdDSA private key, which is the public key that is stored to the server. To get this public key, Keybase uses Elliptic Curves Cryptography (ECC) functions. The EdDSA private key is the seed for the calculation of the public key. After those calculations, the Keybase's client has to prove its knowledge of the private key by making a signature that the server can verify who they are. The signature is the result of signing a *JSON blob*. The client sends the server the signature packaged as a *Keybase-style signature*. Then the server has to verify the signature with the public key (k-id) that was stored in server during the sign up process.

In short, keybase authentication method seems like a Zero-Knowledge Proof protocol (ZKP) [11]. A ZKP authentication protocol provides a different level security due to the fact that the user has never sends their credentials to the server. Keybase's user

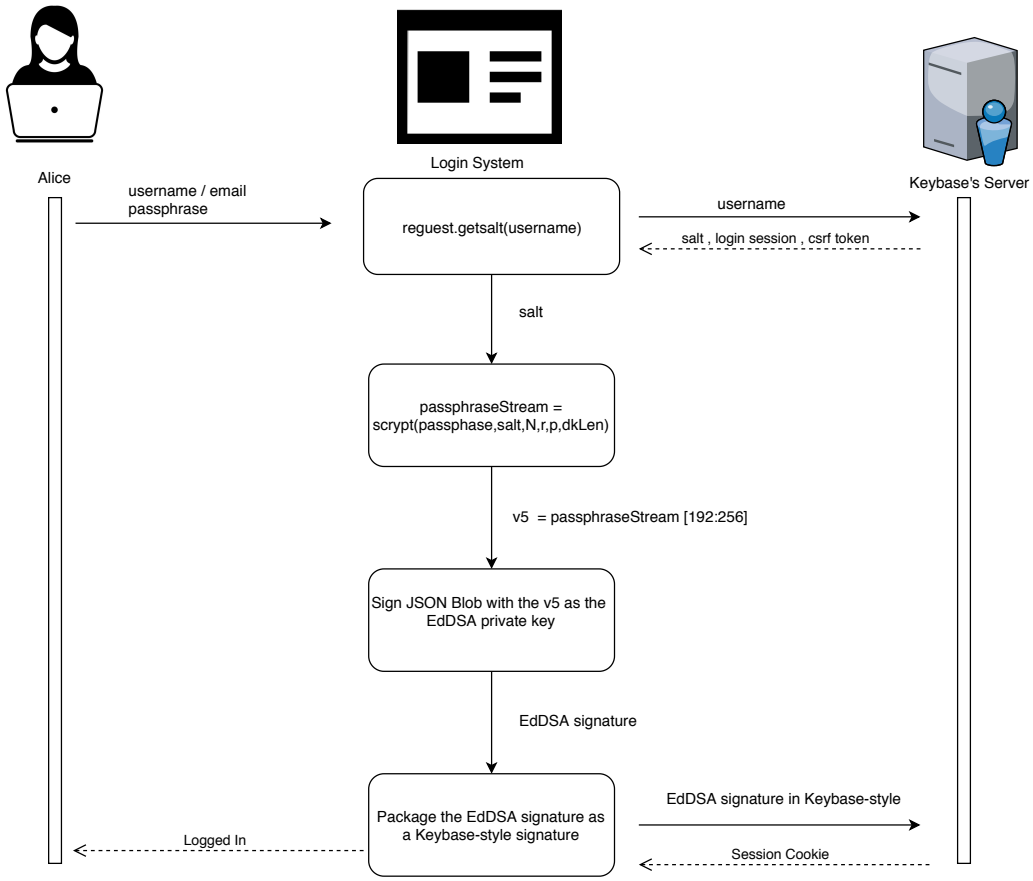

Figure 3.2: This figure shows the high idea behind the Keybase process of Authentication.

authentication is done without the user having to send their password to the server, while the latter can still authenticate them. Moreover, Keybase uses Public Key Cryptography for authenticating users. Specifically, Keybase uses ECC to generate the authentication signatures.

Chapter 4

Implementation

Contents

In this part we explain the idea of a popular cracking tool John the Ripper, how we built our tool and how the tool works. However, we explain how the Keybase's login protocol works until the *k-id* generation.

4.1 John the Ripper

John the Ripper (JtR) [21] is a powerful tool which is mainly used for cracking encrypted passwords. JtR supports a lot of cryptographic algorithms in order to crack given passwords. Also, it offers to the users several extra features. The attacks that JtR can apply to find the passwords are *Brute Force*, *Dictionary* and *Rainbow Table* attacks.

Brute Force attack is a method that is not efficient but it is effective. It is a slow process but it is sure that at some time it will have the right result. JtR tries to guess the password by generating words. It starts from a character and carries on until it finds the right word. In other words, JtR, in brute force attack, combines characters (letters, number or special characters like '?!@#') successively to find the password.

Dictionary attack in JtR tries to guess the passwords by using a file contained a list of words (wordlist file) which is determined by the user. Usually the words that are containing in the wordlist file are common passwords which are taken from a breach.

Rainbow tables include pre-computed hashes for a given algorithm. Specifically, *Rainbow Tables* offer a quick experience for the user. Firstly, JtR looks for the password in the *Rainbow Table* and then it starts its rules. Even though rainbow tables seems to be an efficient attack, they need a considerable processing power and memory. Also, if the hashes are salted then the rainbow attack is infeasible.

Finally, JtR supports some smart rules which can extend wordlists.These rules make efficient permutations and combinations on the given words, resulting in the expansion of the wordlist. The table 4.1 presents a result from using these rules.

blah	blahblah	Blah4	blahhalb	Blahs
Blah	halb	Blah6	blaH	blahed
blahs	1blah	Blah8	2blah	blahing
blah1	BLAH	Blah.	4blah	Blahed
BlahBlah	halB	blahhalb	Blah?	Blah0

Table 4.1: This table shows a small result of John the Ripper's rules. The actual result has 54 permutations of the word *blah*.

4.2 KBCracker

For the implementation of our tool, *KBCracker*, we have developed a python script that computes the public key for a given username using a file of common passwords. The python version that is required for this tool is 3.6. The tool does not have any graphical user interface (GUI). It can be run on terminal or any IDE that supports python (eg Pycharm, Visual Studio Code etc).

Basic Operation: The tool reads a leaked Database, which contains the usernames along with their *k-ids*. The purpose of the tool is to find which passphrase has created each *k-id*. We assume that users' salts have also been leaked along with the database. In case that the salts cannot be leaked, in our implementation, we send a request to Keybase's API to retrieve each salt. Nevertheless, in the actual attack, salts will be fetched once.

The deployment of our tool is based on dictionary attack. Users can define a wordlist that they believe it contains passphrases that are closer to the passphrases that they are searching for. If users do not define a wordlist file, then, by default, the tool uses the wordlist *rockyou.txt* [1]which contains $14,341,564$ unique common passwords. The tool has been developed to read a file with records of usernames and k-ids. For each record in this file, the tool tries to find the corresponding passphrase. When the tool finds the right

passphrase, it prints it to the console with its own username.

4.2.1 Tool's Arguments

The concept *arguments* is very important thing in every program and it is essential for their users. Our tool is running on terminal so it is necessary to have structure and organization on arguments. However, it is crucial to explain the arguments by giving help messages, when users pass incorrect arguments. For instance, the program has to terminate and show to the console an error message to help its user.

We used the module *argparse* for Arguments' Parsing in order to create a user-friendly command-line interface. This happened for the reason that *argaprse* is a helpful and simple module in use. The main purpose of this module is to parse the arguments out of *sys.argv*. We define in the program what arguments it requires, and then *argparse* knows how to parse them. Finally, *argparse* generates *automatically* help messages and issues errors when the program's users give invalid arguments.

The tool that we have developed has three options. These can be given as arguments by the users when the tool is initiated, in order to define its parameters. Furthermore, arguments in this program have short and long options. Short options are defined from the user by using the symbol *"-"* and one letter of the alphabet. Otherwise, if the user wants to use the long option of an argument they must use *"–"* and one word from the English dictionary. In addition, some options do not need parameters or they need one or more. Finally, there are options that are necessary for the tools' execution and options that are not. The table 4.2 presents the tool's possible arguments, the requirement of parameters and the necessity of each argument.

	Short Option "-"	**Long Option "–"**	**Parameters**	**Necessity**
1	h	help	No parameters	Optional
2	w	wordlist	Only one File	Optional
3	f	file	One or more file	Necessary

Table 4.2: This table presents the short options, long options, the required parameters and the necessity of each argument.

The first argument, *-h–help*, is an argument that displays a help menu with the available options with a short description, and then the program terminates it. This option does not requires any parameter.

[1]This wordlist was created by compromised passwords from RockYou [8] company

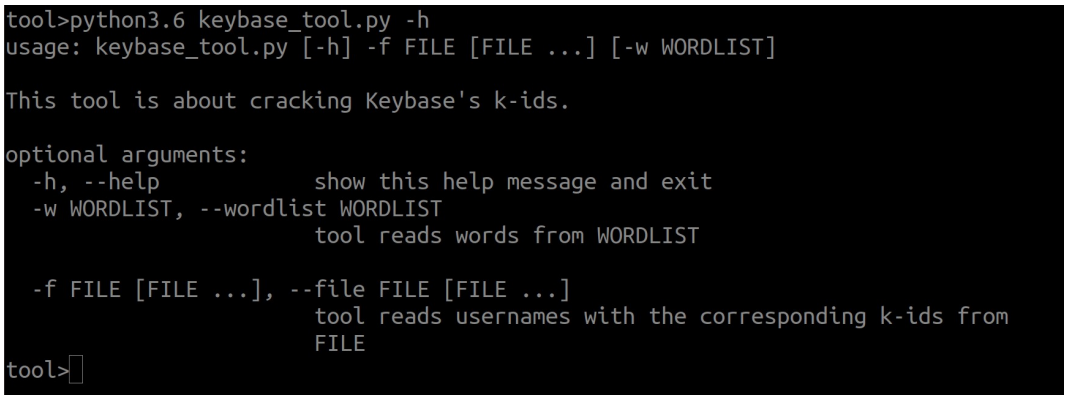

Figure 4.1: This figure shows the help menu that is presented, when the user choose to run the program with the option -h/–help.

The second argument, *-w –wordlist* [FILE], gives the option to the user to insert a file as a parameter. If the user chooses this option, it means that the tool's wordlist is the given file. Otherwise, the tool's wordlist is the *rockyou.txt* file. Furthermore, the user must determine only one file as *wordlist*, if they use this option. Wordlist in the tool is the *dictionary* that the program uses to find the right passphrase that generates the k-id. Moreover, by passing a file as a wordlist, the user must follow some rules regarding file's format. The file must be a set of words that are separated by a newline.

A small example of wordlist file:

```
1 abc123
2 computer
3 tiger
4 password
```

The final option is the *-f –file [FILE]*. The parameter *FILE* is a requirement for this option and user must define at least one file as parameter. The program reads from this file (or files) a set of usernames and their corresponding *k-ids*. Those *k-ids* are going to be cracked by the tool and tool will try to find which passphrases have generated them. When the tool finds the right passphrase, it prints it on the console. Otherwise, in the case where no passphrase in the wordlist matches with a *k-id*, the program does not print anything. The tool prints only the passphrases that it founds.

FILE format: Each username has one k-id separated with a comma ','.
Each record of username and k-id is separated by a newline.

A small example of file:

```
1 username,0120bfedbfda2cb903d526461b5127299025f8c100490427cb306665b4f7a75a6a230a
2 username1,0120cf564b2c954845f0c874de98c099fab6cd4db0962e24d5af6292be4a22b128530a
3 username2,01205153ac36df78eb1b8d5863d0ae36e1dcb7028ca6e269a5cbb73c68cf6774036c0a
```

Execution Examples :

$ python3.6 keybase_tool.py -h

$ python3.6 keybase_tool.py –help

$ python3.6 keybase_tool.py -f file1.txt file2.txt

$ python3.6 keybase_tool.py –wordlist wordlist.txt -f file.txt

$ python3.6 keybase_tool.py -w words.txt -f file.txt

A complete example of how the tool works is presented in *Appendix E*.

4.2.2 Keybase k-id Computation

The Keybase's login Protocol is divided into two rounds, with a two rounds protocol Keybase obviates replay attacks. We implemented this protocol until the *k-id's* creation.

For the implementation, we used Keybase's API for retrieving necessary parameters, we sent requests to the keybase's server and we got the analogous responses. It is important to understand what the responses of the server mean, by studying Keybase's documentation to learn how to manipulate the responses. In addition, all Keybase's API responses are in JSON format and include a status object and a csrf_token string. The status object is a number that presents success or failure. If the status number is "0" that means success. Other statuses mean error and they have extra parameters to describe the error.

First Round of Protocol

Each keybase's user has its own unique *salt* which is generated and stored at the server side at the sign up process. To get this *salt*, the username or email of the user is required to be sent to the server by a request. Thus, the server responses with the user's unique salt and a random challenge named *login_session*.

In our implementation, we assume that we have a leaked database with Keybase's users username and also salts. As a result, we do not need to check if the username exists. Nonetheless, we used the Keybase's API to retrieve the unique salt but in the actual attack the salt for each user will be fetched once. Also we can use this request for the case that the salt cannot be leaked along with usernames and *k-ids*.

Furthermore, if the user exists then we have a random challenge, *login_session*. The random challenge will always be different for every request. Also, it expires after a certain period of time. Moreover, it is not a truly random token but it is a token that is cryptographically tied to client's username with a timestamp. It is important for the keybase's server, as the *login_session* saves the server from keeping state.

Second Round of Protocol

In the second round of the protocol, the *k-id* is calculated. For every username and the *salt* that is retrieved, we get a word to be the passphrase from the wordlist file and compute the *k-id* that is produced. Until we find the passphrase that generates the username's k-id we try the words sequentially from the wordlist file.

The passphrase is going to be stretched and produced a *Passphrase Stream* (256 bytes). For the passphrase stretching, the key derivation function scrypt (from scrypt module) is used. The parameters for the scrypt function are the passphrase, the binary encoding of the user's salt, $N = 2^{15}$, $r = 8$, $p = 1$ and $buflen = 256$. The parameter N is a CPU/memory cost parameter which must be a power of two greater than 1, r and p must satisfy $r * p < 2^{30}$; r is the block size and p is the parallelization factor. Lastly, $buflen$ is the length of the derived key. The *Passphrase Stream* is sliced to the last 32 bytes, which are the *EdDSA private key*. By using the Ed25519 elliptic curve functions we can generate the public key. For the creation of the public key we need to pass the private key in *Ed25519.keygen* as a parameter. The functions's result is the user's *public key*. Afterwards, we add 0120 as prefix and 0a as suffix in the *public key* and thus we have the *k-id* that follows the Keybase's standards.

We check the *username's k-id* from the input file to see whether it is equal with the *calculated k-id*. If it is equal then the program prints to the console the passphrase with the corresponding username.

The tables below, 4.3-4.4, show in short where we can find each parameter and where its usage exists in the protocol. The parameters for the scrypt function *N,p,r* and *buflen* are standard so they are not presented on these tables.

Parameter	**Where**
Username/email	In the File that passed as parameter from the tool's user.
Salt	In the response from getsalt call or leaked along with usernames and k-ids.
Passphrase Stream	The result of the key derivation function scrypt.
EdDSA private key	The last 32 bytes of the Passphrase Stream.
EdDSA public key	Passed the Private Key in Ed25519.keygen as seed to get the EdDSA public key.
k-id	We add 0120 as a prefix and 0a as a suffix to EdDSA public key.

Table 4.3: This table shows where we can find the necessary parameters for the implementation of Keybase's k-id creation.

Parameter	**Usage**
Username/email	Sent with a request to the server to retrieve the user's salt.
Salt	We pass the unxhex salt as parameter in a Key derivation function named scrypt.
Passphrase Stream	Sliced to get the EdDSA private key.
EdDSA private key	Is the seed for the calculation of the public key.
EdDSA public key	Is the k-id without the Keybase's format that the server stored in sign up process.
k-id	Is the k-id with Keybase's format that the server stored in sign up process.

Table 4.4: This table shows where we can use the parameters in Keybase's k-id creation.

4.2.3 Necessary Modules

It is necessary to import the right modules for a successful implementation. In this section, we present the modules that are required and imported in the program. Specifically, the table 4.5 displays the modules that are necessary and the functions that are used from each module for the main Keybase's login protocol implementation. As a result, some standard modules are not presented on the table. Furthermore, for the Ed25519 calculations we use a class named curves, and from this class the module Ed25519 (Appendix C).

Modules	**Functions**
binascii	binascii.hexlify(data) binascii.unhexlify(hexstr)
Ed25519	Ed25519.keygen(seed)
scrypt	scrypt.hash(passphrase,N,p,r,buflen)
requests	requests.post(url, params={}, timeout=10)

Table 4.5: This table shows the necessary modules and the functions of each module that are used in python script for the k-id creation.

Chapter 5

Evaluation

Contents

5.1 Introduction

In this chapter, we evaluate the Keybase login protocol in terms of security and performance. The protocol's evaluation is based on the threat model, that the attacker has a leaked database. This database contains the Keybase's users credentials (usernames with its k-ids).

5.2 Performance

For the performance evaluation we used a password file that contains 3545 simple passwords. Each password was passed as input into several popular schemes. We measured the processing time for each scheme with the same simple password. In Keybase protocol, it is necessary to get the salt by sending a request to the server with the username or email. In order to avoid creating a big overhead in the server side, we use a specific username and we get the unique salt once. As a result, we passed the salt in the *scrypt* function with out sending 3545 requests to the Keybase's server.

Firstly, we timed each part of the Keybase's protocol. As we expected the higher percentage of the amount of time is consumed by the stretching part which is the function *scrypt*. The table 5.1 shows the average processing time for each part of the protocol. We timed the passphrase stretching and the ECC public key generation. As we can see

in the table below the time that is needed for the ECC public key generation is negligible in relation to the passphrase's stretching consumed time. Although, the key generation is not a slow function, its usage provides a strong level of security as we explain in *Section 2.2* .

Part	**Milliseconds**
Scrypt	117.419
ECC Public Key Generation	10.320
Overall k-id generation	127.739
Scrypt Percentage:	*91.91 %*

Table 5.1: This table shows the average processing time of each part of the keybase's protocol in millisecond and the percentage of time that the scrypt function consumed.

Therefore, we developed another python script that simulated some famous password hashing functions in order to contrast them with the Keybase's passphrase stretching (function *scrypt*). The table 5.2 shows the result of this measurement. Regarding the schemes that were tested, *bcrypt* (12 rounds) scheme is the slowest. However, *scrypt* causes an overhead that is on the same level as *bcrypt*. Moreover, the other schemes that are tested are out of competition based on their times. Specifically, Keybase's protocol requires the most processing time compared to the other schemes except from the *bcrypt* scheme.

	Average (milliseconds)	**Standard Deviation**
bcrypt (12 rounds)	272.820	70.412
Keybase	113.806	16.594
65,537 iterations of SHA1	38.011	6.691
8,192 iterations of MD5	4.701	0.885
SHA256	0.004	0.003
SHA512	0.001	0.001

Table 5.2: This table shows the average processing time that password hashing schemes needs for 3545 passwords.

Furthermore, we contrast the memory consumption between *scrypt* function and *bcrypt* function by monitor the memory usage for each function. For this measurement we used the *Memory Profiler* which is a python module [23]. To activate this module we run the script with the command *mprof run* <*executable*>. Thus, a file that contains the data

of simulation is generated. Finally, we run *mprof plot* which reads the latest file that is generated from *mprof run* and generates a graph using these data.

We have developed a python script that runs sequentially the *scrypt* function and then the *bcrypt* function. The graph 5.1 shows the memory that is needed in MiB and the simulation time. The result of the script that simulates the two functions is shown in figure 5.1.

As it seems, scrypt uses more memory than bcrypt. As a result a brute force attack at *scrypt* needs a strong and expensive hardware and makes the attack even harder.

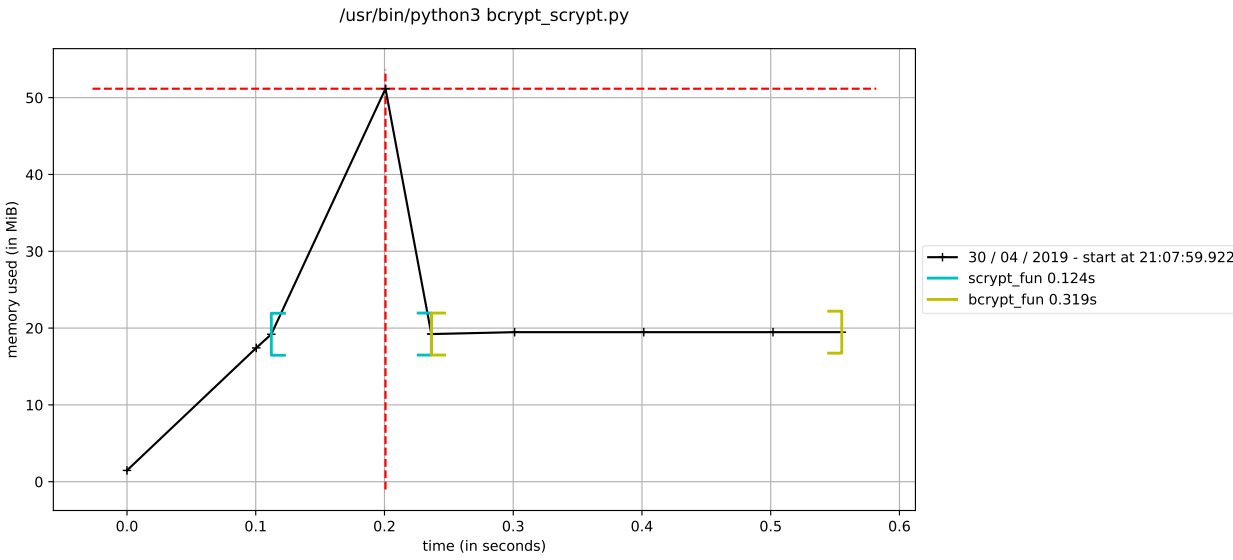

Figure 5.1: This graph displays the memory usage of the script that runs both bcrypt and scrypt functions.

5.3 Security

Regarding the results that we get from the performance evaluation, we deduce that if an attacker has the *k-ids*, it is computationally hard for the attacker to figures out the passphrases. We have limited time, so it is important, in terms of security to use functions that are designed on purpose for slowing down hashing operations in order to make the attack harder.

What is more, we monitored memory consumption of the *scrypt* function in contrast with the *bcrypt* function. We monitored the memory usage of these functions so as to evaluate the Keybase's protocol. While the protocol needs a quite large amount of memory that means that an attacker has to use an expensive and powerful hardware to compromise a database of *k-ids*.

In conclusion, both the Keybase's k-id creation and also the methods that Keybase uses in order to authenticates its users, provide a strong level of security. We stress that, if the attacker has the database, then they need strong capabilities of computational resources in order to figure out the passphrases. However, we are not stressed about the case that users give their passphrases or their personal computer with all connected accounts to another person, having as a result their passphrases now not to be secret.

Chapter 6

Related Work

Contents

The selection of a strong password is a headache for a user. Mainly, users prefer to select a password that is easy to remember and use it for all the online services that they use. Unfortunately, there are a lot of disadvantages selecting a simple and usual password if the system has simple authentication. We study about Keybase login protocol and authentication for the reason that it is not a simple protocol. Keybase protects the users' credentials by keeping them truly secret. In particular, Keybase does not rely on *passwords*, but on *passphrases*. Users need to remember a strong secret, but server has no knowledge how this secret looks.

In this part we are going to review some works that are related both for password authentication and Keybase. Keybase is a form of Password Authenticated Key Exchange (PAKE) and it is interesting to review how we can obtain strong level of security by using Keybase and its methods.

6.1 PAKE

Password Authenticated Key Exchange (PAKE) [6, 7, 26]
A method or protocol is referenced as PAKE when the password authentication is based on exchange without server knowing the password as it is. Keybase is a form of PAKE protocol. Furthermore, Keybase server does not have access on the actual user's password. The only knowledge that the server has about the passphrase is the *k-id* which is created in Keybase's login protocol.

On the contrary with the classic protocols, PAKE systems have the crucial point on the authentication process. Classic protocols let users to select a password and then the system has the password even if it is encrypted or not. PAKE systems do not know the password, so they have to solve the problem of authentication with alternative and efficient ways.

Keybase uses ECC signatures for the authentication. Keybase's users prove their identity with a not text-based authentication. Server has a key of the user's passphrase and user tries to verify that is the valid user by using ECC. These type of protocols are secure against active attacks. Furthermore, these protocols keep passwords truly secret, protecting users' credentials.

6.2 modssl-hmac

modssl-hmac [10] is an approach of password hardening by using a private key and HMAC. The propose of modssl-hmac is to protect passwords when they are leaked. It does not provide security in terms of not leakage but if the passwords are leaked, it provides a level of security that is not able to be compromised. The private key is stored in a module in Apache server, so if an attacker has access in the private key, then he can do stronger attack than to find the users' passwords. Keybase offers a different approach about users' password storage using ECC. Keybase stores the public key of the the ECC computations. If the database was leaked the attack that Keybase suffers is the brute force attack (dictionary). Brute force attack can compromised the *k-id* and figures out the actual passphrase. The attacker has to recreate and check if the public key is the same. We can use *modssl-hmac* and Keybase methods in some way, in future work, in order to increase the level of password hardening.

6.3 KAuth

KAuth [9] is based on Keybase platform. KAuth uses Keybase as reference and just accepts the features and security that Keybase offers. KAuth built a website that uses the Keybase's login protocol, username and passphrase that are needed for the KAuth authentication are the same credentials that a user use for Keybase's authentication. We were inspired to continue the approach of KAuth by studying about Keybase authentication and login protocol. KAuth references, as future work, a cracking tool for Keybase's login protocol. We work on Keybase's login protocol and we create this cracking tool to see how Keybase offers its security and to specify how Keybase creates the keys for user's authentication.

Chapter 7

Conclusion

In this thesis project, we studied about Keybase authentication, which relies on Elliptic Curve Cryptography and passphrases. Keybase, by combining these methods, can authenticate its users without any knowledge of their credentials. However, Keybase keeps users credentials truly secret.

We created a tool that follows the Keybase's API and documentation in order to create the key (k-id) that Keybase uses for its users' authentication. The purpose of our tool is to figure out which passphrase has generated each k-id. The idea behind that is based on the dictionary attack. Furthermore, using this tool, we evaluated Keybase's protocol in terms of performance and security. Finally, our tool relies on the leakage of the service's database with its user's credentials, thus it cannot attack Keybase, since its database is not leaked.

Bibliography

[1] Password Authentication and Password Cracking. https://www.wordfence.com/learn/how-passwords-work-and-cracking-passwords/. Last accessed in May 2019.

[2] Mary Cindy Ah Kioon, Zhao Shun Wang, and Shubra Deb Das. Security analysis of md5 algorithm in password storage. In *Applied Mechanics and Materials*, volume 347, pages 2706–2711. Trans Tech Publ, 2013.

[3] Sheikh Iqbal Ahamed, Farzana Rahman, and Endadul Hoque. Erap: Ecc based rfid authentication protocol. In *2008 12th IEEE International Workshop on Future Trends of Distributed Computing Systems*, pages 219–225. IEEE, 2008.

[4] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 33–62. Springer, 2017.

[5] M Aydos, T Yanık, and CK Koc. High-speed implementation of an ecc-based wireless authentication protocol on an arm microprocessor. *IEE Proceedings-Communications*, 148(5):273–279, 2001.

[6] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *International conference on the theory and applications of cryptographic techniques*, pages 139–155. Springer, 2000.

[7] Steven M Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.

[8] William J. Burns. Common Password List (rockyou.txt. https://www.kaggle.com/wjburns/common-password-list-rockyoutxt. Last accessed in May 2019.

[9] Panayiotis Charalambous, Marios Karapetris, and Elias Athanasopoulos. Kauth: A strong single sign-on service based on pki. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, 2018.

[10] Constantinos Diomedous and Elias Athanasopoulos. Practical Password Hardening based on TLS. June 2019. In Proceedings of the 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA). Gothenburg, Sweden.

[11] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[12] Praveen Gauravaram. Security analysis of salt|| password hashes. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, pages 25–30. IEEE, 2012.

[13] Shirley Gaw and Edward W. Felten. Password management strategies for online accounts. In *Proceedings of the Symposium on Usable Privacy and Security*, SOUPS, 2006.

[14] Kamlesh Gupta and Sanjay Silakari. Ecc over rsa for asymmetric encryption: A review. *International Journal of Computer Science Issues (IJCSI)*, 8(3):370, 2011.

[15] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *International workshop on cryptographic hardware and embedded systems*, pages 119–132. Springer, 2004.

[16] Darrel Hankerson and Alfred Menezes. *Elliptic curve cryptography*. Springer, 2011.

[17] Keybase. Keybase key ids (kids). `https://keybase.io/docs/api/1.0/kid`. Last accessed in May 2019.

[18] Georgios Kontaxis, Elias Athanasopoulos, Georgios Portokalidis, and Angelos D. Keromytis. Sauth: Protecting user accounts from password database leaks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 187–198, New York, NY, USA, 2013. ACM.

[19] Himanshu Kumar, Sudhanshu Kumar, Remya Joseph, Dhananjay Kumar, Sunil Kumar Shrinarayan Singh, and Praveen Kumar. Rainbow table to crack password using md5 hashing algorithm. In *2013 IEEE Conference on Information & Communication Technologies*, pages 433–439. IEEE, 2013.

[20] Julio Lopez and Ricardo Dahab. An overview of elliptic curve cryptography. 2000.

[21] Openwall. John the Ripper password cracker. `https://www.openwall.com/john/`. Last accessed in May 2019.

[22] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[23] Fabian Pedregosa. Memory profiler. `https://pypi.org/project/memory-profiler/`. Last accessed in May 2019.

[24] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[25] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.

[26] Thomas Wu. The secure remote password protocol. In *In Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, 1998.

Appendix A

In this part is presented the tool's README file.

```
This program is a tool,named KBCracker, that tries to figure out the Keybase's
users passphrases.The tool follows the Keybase's login protocol, in order to
create a k-id. The user should specify the username and the k-id (in a file).
From the username, the tool retrieves the user's unique salt. The salt is
passed as an argument in the scrypt function. The scrypt's result is the
Ed25519 private key from which we can find the public key and then create
the k-id, based on Keybase's standard. Finally, the tool checks if the k-id is
equal to the input k-id and prints the passphrase in the console.

Commands to run the tool:
        python3.6 keybase_tool.py -w <Wordlist> -f <File>
        python3.6 keybase_tool.py -f <File>

** -w <Wordlist> is optional , by default the wordlist is the file: "rockyou.txt"

Explain the options that user can use in the tool:

-h,  --help
        Display the help menu, available options and exit.

-w,  --worldlist [FILE]
        The file that passed as the wordlist that the tool will use.
        FILE format: Set of words separated by a newline.
                ex. 1234
                    abcde

-f,  --file [FILE]
        Read from file a set of usernames and corresponding k-ids.
        FILE format: Set of usernames and corresponding k-ids.
                Each username has one k-id seperated with a comma ','.
                Each record of userame and k-id is seperated by newline.
                ex. aaa,111111
                    bbb,222222
```

Appendix B

In this part is presented the python code of the tool.

```
import binascii
from curves import Ed25519
import sys
import os
import scrypt
import csv
import requests
import argparse

'''
This function is about to calculate the k-id that produces the
parameter password.
The calculation is based on keybase's login protocol and format.
@params password : The password which the k-id is going to be calculated
salt     : Each user has its unique salt. The salt is necessary for the key derivation
           function, scrypt.
@return k-id : In keybase's format.
'''
def findpassphrase(passphrase, salt):

    s = binascii.unhexlify(salt)

    passphrasestream = scrypt.hash(passphrase, s, N=2 ** 15, r=8, p=1, buflen=256)

    v5 = passphrasestream[224:256]

    private, pubkey = Ed25519.keygen(v5)

    public_key = binascii.hexlify(pubkey)
    pk = str(public_key, "utf-8")
    kid = '0120' + pk + '0a'

    return kid


'''
This function retrieves the salt for the username that is passed as parameter.
The salt is retrieved using the keybase's api. The server gets the username and
responses with the corresponding salt for this username.
@params username : Each username in keybase has a unique salt. Server gets the username
                  and response with the salt.
@return salt     : returns the user's salt.
'''
def findsalt(username):

    salt_url = 'https://keybase.io/_/api/1.0/getsalt.json'
    salt_resp = requests.post(salt_url, params={'email_or_username': username}, timeout
        =10)
    salt_resp.raise_for_status()
    salt_json = salt_resp.json()

    salt = salt_json['salt']

```

```
53      return salt
54
55
56  '''
57  This function finds the passphrase that generates the k-id. This function use the file
58  wordlist, that it is passed as parameter, and for each word in this file generates
59  a k-id. If the k-id equals with the k-id that it is passed as parameter, prints
60  out the username and its passphrase. For the generation of the k-id this function
61  calls other functions, findsalt(username), findpassphrase(passphrase, salt).
62  Basically, this function works like an intermediary to complete the process of
63  the passphrase's search.
64  @params
65  username : The username is needed for salt's retrive.
66  kid      : The searching k-id. This function uses words until to find the one
67             which generates this k-id.
68  wordlist : Each word is used for the generation of the k-id, until find the word
69             that matches with the searching k-id.
70  '''
71  def crack(username, kid, wordlist):
72
73      salt = findsalt(username)
74
75      with open(wordlist) as csv_file:
76          csv_reader = csv.reader(csv_file, delimiter=',')
77          for passw in csv_reader:
78              passphrase = passw[0]
79              result = findpassphrase(passphrase, salt)
80              if result == kid:
81                  print(username, passphrase)
82                  return 1
83
84
85  '''
86  This function reads the data from each file in order to figure out what
87  word generates each k-id.
88
89  @params
90  file       :The file that contains the usernames with the correspondign k-ids.
91  wordlist     :Is the file that contains words. Each word is used for the generation
92                of the k-id, until find the word that match with the searching k-id.
93  '''
94  def readfromfile(file, wordlist):
95      tab = [0,0]
96      if not os.path.isfile(file):
97          print("File", file, "not␣found.")
98          return
99
100     with open(file) as csv_file:
101         csv_reader = csv.reader(csv_file, delimiter=',')
102         for record in csv_reader:
103             username = record[0]
104             kid = record[1]
105             c = crack(username, kid, wordlist)
106             tab[0] = tab[0] + 1
107
108             if c == 1:
109                 tab[1] = tab[1] + 1
```

```
110
111         return tab
112
113
114
115 def main(argv):
116
117     wordlist = "rockyou.txt"
118     parser = argparse.ArgumentParser(description="This␣tool␣is␣about␣cracking␣Keybase's
        ␣k-ids.")
119     group = parser.add_argument_group()
120     group.add_argument("-f", "--file", nargs='+', help="tool␣reads␣usernames␣with␣the␣
        corresponding␣k-ids␣from␣FILE", required=True)
121     parser.add_argument("-w", "--wordlist", nargs=1, help="tool␣reads␣words␣from␣
        WORDLIST")
122
123     args = parser.parse_args()
124     if len(argv) == 0:
125         printhelp()
126         sys.exit()
127
128     for f in args.file:
129         if args.wordlist:
130           tab = readfromfile(f, args.wordlist[0])
131         else:
132           tab = readfromfile(f, wordlist)
133
134     print("Found:", tab[1],"out␣of",tab[0], "passphrases.")
135
136
137 if __name__ == "__main__":
138     main(sys.argv[1:])
```

Appendix C

In this part is presented the python code of the Ed25519 class.

```
#A point on Edwards25519
class Edwards25519Point(EdwardsPoint):
    #Create a new point on curve.
    base_field=Field(1,2**255-19)
    d=-base_field.make(121665)/base_field.make(121666)
    f0=base_field.make(0)
    f1=base_field.make(1)
    xb=base_field.make(hexi("216936D3CD6E53FEC0A4E231FDD6DC5C692CC76"+\
        "09525A7B2C9562D608F25D51A"))
    yb=base_field.make(hexi("666666666666666666666666666666666666666666"+\
        "6666666666666666666666658"))
    #The standard base point.
    @staticmethod
    def stdbase():
        return Edwards25519Point(Edwards25519Point.xb,\
            Edwards25519Point.yb)
    def __init__(self,x,y):
        #Check the point is actually on the curve.
        if y*y-x*x!=self.f1+self.d*x*x*y*y:
            raise ValueError("Invalid point")
        self.initpoint(x, y)
        self.t=x*y
    #Decode a point representation.
    def decode(self,s):
        x,y=self.decode_base(s,256);
        return Edwards25519Point(x, y) if x is not None else None
    #Encode a point representation
    def encode(self):
        return self.encode_base(256)
    #Construct neutral point on this curve.
    def zero_elem(self):
        return Edwards25519Point(self.f0,self.f1)
    #Solve for x^2.
    def solve_x2(self,y):
        return ((y*y-self.f1)/(self.d*y*y+self.f1))
    #Point addition.
    def __add__(self,y):
        #The formulas are from EFD.
        tmp=self.zero_elem()
        zcp=self.z*y.z
        A=(self.y-self.x)*(y.y-y.x)
        B=(self.y+self.x)*(y.y+y.x)
        C=(self.d+self.d)*self.t*y.t
        D=zcp+zcp
        E,H=B-A,B+A
        F,G=D-C,D+C
        tmp.x,tmp.y,tmp.z,tmp.t=E*F,G*H,F*G,E*H
        return tmp
    #Point doubling.
    def double(self):
        #The formulas are from EFD (with assumption a=-1 propagated).
        tmp=self.zero_elem()
        A=self.x*self.x
```

```
54          B=self.y*self.y
55          Ch=self.z*self.z
56          C=Ch+Ch
57          H=A+B
58          xys=self.x+self.y
59          E=H-xys*xys
60          G=A-B
61          F=C+G
62          tmp.x,tmp.y,tmp.z,tmp.t=E*F,G*H,F*G,E*H
63          return tmp
64      #Order of basepoint.
65      def l(self):
66          return hexi("1000000000000000000000000000000014def9dea2f79cd"+\
67              "65812631a5cf5d3ed")
68      #The logarithm of cofactor.
69      def c(self): return 3
70      #The highest set bit
71      def n(self): return 254
72      #The coding length
73      def b(self): return 256
74      #Validity check (for debugging)
75      def is_valid_point(self):
76          x,y,z,t=self.x,self.y,self.z,self.t
77          x2=x*x
78          y2=y*y
79          z2=z*z
80          lhs=(y2-x2)*z2
81          rhs=z2*z2+self.d*x2*y2
82          assert(lhs == rhs)
83          assert(t*z == x*y)
```

Appendix D

In this part is presented the python function which generates the Ed25519 public key.

```
#Generate a key. If privkey is None, random one is generated.
#In any case, privkey, pubkey pair is returned.
    def keygen(self,privkey):
        #If no private key data given, generate random.
        if privkey is None: privkey=os.urandom(self.b//8)
        #Expand key.
        khash=self.H(privkey,None,None)
        a=from_le(self.__clamp(khash[:self.b//8]))
        #Return the keypair (public key is A=Enc(aB).
        return privkey,(self.B*a).encode()
```

Appendix E

In this part is presented a complete example of how the tool works.

This is the input file.

```
username,0120bfedbfda2cb903d526461b5127299025f8c100490427cb306665b4f7a75a6a230a
username1,0120cf564b2c954845f0c874de98c099fab6cd4db0962e24d5af6292be4a22b128530a
username2,01205153ac36df78eb1b8d5863d0ae36e1dcb7028ca6e269a5cbb73c68cf6774036c0a
username3,0120047f6d2498c8e16a1e1f14ef609f3629c8add90914a326cf47aec26a133ba6060a
username4,01209369d1c5152026d92a9356386e7f2604f34142ad8393ba65efae40d77437b01b0a
username5,0120253dd59a305bf8185aaadae9d9c37bfd693eb2f60a6af1f30923286f9c55d9690a
username6,0120790fbfd36d03aef4b13dca97bf804dc51bcd32252ccd14121c7c35f7d33d96da0a
username7,012055909f0cbff46118436c2bac80f7532761f5c17269e69518c1063020d6b748f00a
username8,012032610e4832c59bc1d927317d53f08ccc0fec916bb55922f8f70be2d1a52a409d0a
username9,0120f91b11d5b85bf53382ce2ac58ff7c37e8af7fc775932f1982510936bc1943dfd0a
username10,0120b3dc08587168d9591951e5bdfa33115621cac74e60105401dd36a065fffe725a0a
username11,0120fb5eb5b7a6d72346b87aad96faee14b502d7368697875c6c63a99ef98aa147f80a
username12,01206aaaa263c77bf22d72db7dca4f5a8e57a0384e696e323022bd28900f853b99d30a
username13,01203a63c7714543391e4d0f71f08cf97a77301dd69025373040a556f28d5cc541e10a
username14,01200ff30c95a7dfdaa95c21a2759cd2f5bc31d88dd72b13ae527c3979b049a2cee20a
```

This is the wordlist file.

```
123456
12345
password
gandalf
magic
merlin
newyork
soccer
rainbow
bigmac
1234567890
computer
tigger
colorado
qwerty
money
carmen
mickey
secret
elpaco
green
helpme
linda
harrypotter
```

The result.

```
$ python3.6 keybase_tool.py -f input.txt -w wordlist.txt
username 123456
username1 12345
username2 password
username3 rainbow
username4 bigmac
username5 1234567890
username6 computer
username7 tigger
username8 colorado
username9 qwerty
username10 money
username11 carmen
username12 mickey
username13 secret
username14 harrypotter
Found: 15 out of 15 passphrases.
```