Social Authentication for Mobile Phones

by

Bijan Soleymani

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering

Electrical and Computer Engineering

McGill University
Montreal, Quebec, Canada

This Thesis is dedicated to my parents.

Acknowledgement

First of all I would like to express my most sincere gratitude to my supervisor Professor Muthucumaru Maheswaran for his support, guidance and advice throughout my graduate research. I would like to thank each and every one of the student in the McGill Advanced NEtworking Research Lab (ANRL) for their opinions and ideas. I would also like to thank my parents for their encouragement throughout my studies and being there when I needed them.

Abstract

In this thesis we present a scheme for automating authentication based on social factors using mobile phones. We test its feasibility by running simulations on an existing dataset. We implement two protocols one based on public key infrastructure and the other on hash chains. Then we consider possible threat scenarios.

Web applications such as online banking, online shopping carts, and so on, depend on the user authenticating himself securely. Traditionally this involves a username and password and if more security is required an electronic token is used in addition to this password. Other than these two "factors" there is also biometrics, such as fingerprints, retinal scans and voice recognition. Thus the traditional systems use some combination of these three factors: something you know (passwords), something you have (tokens) and something you are (biometrics).

Recently it has been suggested that a fourth factor: someone you know also be part of the authentication process . This technique has been applied to the problem of emergency authentication, as a replacement for challenge questions or calls to a help-desk. The idea is that the user uses a token and pin to authenticate himself. If the user forgets his token, he can ask a friend who has their token to grant him a temporary password. Thus fourth factor or social authentication is based on the process of vouching. In this method a user asks a friend to vouch for them, that is the friend must recognize the user and then issue some proof of this recognition, which the user then uses to log in to the service. In , this vouching was done explicitly, with the user contacting a friend and literally asking for a vouching code. In this thesis we will use users' cellphones to automate this process.

Whenever a user calls a friend, a token will be issued "vouching" for this contact.

These tokens if obtained in sufficient numbers can then be used to prove that a user is who he says he is. In addition to this fourth factor we will make use of other means of authentication. These include a PIN (personal identification number) that must be entered when validating the “vouching” tokens, possibly fingerprint recognition and outputs from other biometric sensors, such as a wrist watch with heart-rate monitor, or a shoe with built-in pedometer. In this case we may want two out of three or four of these to match before authenticating the user.

Résumé

Dans cette thèse nous présentons un système d'automatisation de l'authentification basée sur des facteurs sociaux, utilisant des téléphones mobiles. Nous vérifions sa faisabilité en exécutant des simulations sur un ensemble de données disponible. Nous mettons en œuvre deux protocoles l'une basée sur l'infrastructure à clé publiques et l'autre sur les chaînes de hachage. Ensuite, nous considérons les menaces possible.

Les applications Web telles que les services bancaires en ligne, les paniers d'achat en ligne, etc, dépendent de l'authentification de l'utilisateur en toute sécurité. Traditionnellement, ceci néssecite un nom d'utilisateur et un mot de passe et si plus de sécurité est requis un jeton de sécurité est utilisé en plus de ce mot de passe. Hormis ces deux facteurs il y a aussi la biométrie, comme les empreintes digitales, empreintes rétiniennes et la reconnaissance vocale. Donc les systèmes traditionnels utilisent une combinaison de ces trois facteurs: quelque chose vous savez (mots de passe), quelque chose que vous avez (jetons) et quelque chose que vous êtes (biométrie).

Récemment, il a été suggéré qu'une quatrième facteur: quelqu'un vous connaissez fases aussi partie du processus d'authentification. Cette technique a été appliquée au problème de l'authentification d'urgence, comme un remplacement pour les questions de défi ou les appels à un centre d'assistance. L'idée est que l'utilisateur utilise un jeton électronique et un NIP pour s'authentifier. Si l'utilisateur oublie son jeton, il peut demander à un ami qui a son jeton de lui accorder un mot de passe temporaire. Ainsi le quatrième facteur ou authentification sociale est fondée sur un processus d'attestation. Dans cette méthode, un utilisateur demande à un ami à se porter garant pour lui, cet ami doit reconnaître l'utilisateur et lui livrer une preuve de cette reconnaissance, que l'utilisateur

utilise ensuite pour se connecter au service. En ce cas l'attestation a été fait de manière explicite, l'utilisateur devant contacter un ami et demander code temporaire verbalement. Dans cette thèse, nous utiliserons des téléphones cellulaires afin d'automatiser ce processus.

Chaque fois qu'un utilisateur appelle un ami, un jeton sera publié comme attestation de ce contact. Ces jetons si ils sont obtenus en nombre suffisants peuvent alors être utilisés pour prouver que l'utilisateur est bien celui qu'il prétend être. En plus de ce quatrième facteur on fera appel à d'autres moyens d'authentification. Il s'agit notamment du code NIP (Numero d'Identification Personnelle) qui doit être entré lors de la validation avec les jetons d'attestation et possiblement la reconnaissance d'empreintes digitales et d'autres signaux en provenance de capteurs biométriques, comme une montre avec cardio-fréquencemètre, ou des chaussures avec podomètre intégré. Dans ce cas, nous voulons vérifier deux sur trois ou deux sur quatre de ces derniers avant l'authentification de l'utilisateur.

Contents

List of Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
FMR	False Matching Rate
GPS	Global Positioning System
HMAC	keyed-Hash Message Authentication Code
MAC	Media Access Control
MD5	Message-Digest algorithm 5
PGP	Pretty Good Privacy
PIN	Personal Identification Number
PKI	Public Key Infrastructure
SHA1	Secure Hash Algorithm 1
SIM	Subscriber Identity Module
SMS	Short Messaging Service
TPM	Trusted Platform Module
USB	Universal Serial Bus

List of Figures

1

Introduction

Web applications such as online banking, online shopping carts, and so on, depend on the user authenticating himself securely. Traditionally this involves a username and password and if more security is required an electronic token is used in addition to this password. Other than these two “factors” there is also biometrics, such as fingerprints, retinal scans and voice recognition. Thus the traditional systems use some combination of these three factors: something you know (passwords), something you have (tokens) and something you are (biometrics).

Each of these factors has its advantages and disadvantages. For example, memorized passwords can’t be stolen, but may end being weak due to the limitations on the password length and complexity that can be memorized.

1.1 Fourth Factor Authentication

Recently it has been suggested that a fourth factor: someone you know also be part of the authentication process [1]. J. Brainard et al. have applied this technique to the problem of emergency authentication, as a replacement for challenge questions or calls to a help-desk. The idea is that the user uses a token and pin to authenticate himself. If the user forgets his token, he can ask a friend who has their token to grant them a temporary password.

Thus fourth factor or social authentication is based on the process of vouching. In this method a user asks a friend to vouch for him, that is the friend must recognize the user and then issue some proof of this recognition, which the user then uses to log in to the service.

1.2 Motivation

Fourth factor authentication has several advantages over factors used for authentication. The main advantage is that attacks are detectable by the user. For example if a user receives many vouching requests from people they do not know or recognize they can report the incident. This security is achieved at the expense of minor but regular disturbance to the user.

In [1], the vouching was done explicitly, with the user contacting a friend and literally asking for a vouching code. In this thesis we will use users' cellphones to automate this process, thus reducing the burden on the user. Whenever a user calls a friend, a token will be issued "vouching" for this contact. These tokens if obtained in sufficient numbers can then be used to prove that a user is who he says he is.

While this process is automated, in order to increase security in our implementation, the user is prompted to confirm Bluetooth sightings, and given the option of not issuing a token after a phone conversation if he doesn't recognize the other party.

In addition to this fourth factor we will make use of other means of authentication. These include a PIN (personal identification number) that must be entered when validating the "vouching" tokens, possibly fingerprint recognition and outputs from other biometric sensors, such as a wrist watch with heart-rate monitor, or a shoe with built-in pedometer. In this case we may want two out of three or four of these to match before authenticating the user.

1.3 Contribution of the Thesis

The contribution of this Thesis is the development of a social authentication system on mobile phones based on the users' phone conversations and Bluetooth sightings. A protocol is proposed and tested using simulations on cellphone logs. A software system implementing this protocol is written in pys60 (Python for Symbian system 60) and is tested on Bluetooth enabled cellphones.

Encrypted messages, “tokens”, are generated and used to prove that the phone conversations and Bluetooth sightings took place at specified times. Two interchangeable methods are proposed for ensuring the security and authenticity of these messages. One method uses public key infrastructure (PKI) and the other employs hash chains.

Conversation duration is analyzed to determine whether the user has actually talked to an acquaintance (ruling out wrong numbers and imposters). Similarly, Bluetooth distance is determined based on an indirect measurement of signal strength, and is used to eliminate out-of-sight Bluetooth devices.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows. Chapter 2 will present the required background on the traditional authentication factors. Chapter 3 will review some of the related work. In Chapter 4, we will outline the details of our proposed scheme. In Chapter 5, we will test its feasibility by running simulations on data from the Reality Mining project from MIT [2]. The implementation of our authentication technique will be explained in detail in Chapter 6. In Chapter 7 we will present the results of the implementation. In Chapter 8 we will examine the possible threat scenarios. The conclusion will be presented in Chapter 9.

2

Background

As previously stated the three traditional factors used for authentication are: Passwords, Security Tokens and Biometrics. They correspond to the categories: something you know, something you have and something you are respectively. More recently a new factor (someone you know) has been proposed as a fourth authentication factor. The following is a survey of these techniques, followed by a comparison of their strengths and weaknesses. Social or Fourth factor authentication is considered more fully in Section 3.2 of the next chapter.

Although we consider these factors in the context of automated authentication to computer systems, the use of these factors for authentication predates the information age.

Passwords (watchwords) were employed in the Roman military. The watchword for the night was distributed from the commander to a soldier who gave it to the leader of the first unit, who then gave it to the leader of the second unit, and so on until it got to the last unit, which would transmit it back to the tribunes, who could thus ensure that the leader of each unit had the correct watchword.

Keys a mechanical equivalent of Security Tokens, were used in the form of wooden keys as early as 4000 years ago in Egypt.

Biometrics in the form of face recognition has existed since prehistoric times. Fingerprints have been used for identification as early as 1900 BC in Babylon, to identify parties in a contract. The parties would impress their fingerprints on the clay tablets on which the contract had been written.

While social authentication is a new concept in computer systems, human authentication based on mutual acquaintances is not new. An example of this is the requirement for two references when applying for a passport.

2.1 Passwords

A password is a secret string that is used for authentication. They are the most common authentication factor in computer systems, due to their low cost and ease of use. They require no special hardware, and the only burden placed on the user is that of remembering and typing a relatively short string.

2.1.1 Password Randomness

The security of a password system depends on creating a password that an attacker can't predict or guess. The predictability of a password is inversely proportional to its randomness. Randomness implies unpredictability, uniqueness and even distribution. In 1948 Shannon introduced the notion of entropy as a measure of unpredictability or uncertainty[3]. The entropy is defined as:

Where p(xi) is probability of xi. It can be shown that the maximum entropy of an event (source) with n outcomes (letters) is logb(n) and is achieved when the source letters are equiprobable. "Equivalently, the Shannon entropy is a measure of the average information content one is missing when one does not know the value of the random variable."[4] Shannon showed that in the limit, the average length of the shortest representation of the message in a given alphabet is its entropy divided by the logarithm of the number of symbols in that alphabet.

A closely related concept is that of Kolmogorov complexity. The Kolmogorov complexity of a string is defined as the length of the smallest program that can generate that string. The Kolmogorov complexity K is approximately equal to the Shannon entropy H if the sequence is drawn at random from a distribution that has entropy H.

This notion of complexity corresponds to descriptive complexity or program length. There is also a notion of computational complexity or time complexity. The first relates to the length of the program required to generate the string, while the second relates to the amount of time or the number of computations required to generate that string. For example given “n” the size of the program required to generate a string of n repeated zeroes is constant and does not depend.on n. Similarly the size of the program required to generate the n first digits of pi is also constant. However the computational complexity of generating the digits of pi is much greater than that of generating the string of zeroes.[5]

In order to ensure that a password is secure against prediction we want to ensure that it has a high entropy or equivalently a high complexity. This can be done through three main means: using a random source of information (or a source with the random properties), increasing the number of characters in the password and increasing the size of the alphabet used.

2.1.2 Password Length

Logically the longer a password, the harder it is to guess. This is because there are more possible combinations of characters to go through. It also seems reasonable to assume that the longer a password the harder it will be for the user to remember. However this is not exactly true. It is true that a longer password will be harder to remember if it is as random as the shorter password (that is to say it has the same per character entropy). That need not be the case. The password could have less entropy per character but the increase in the number of characters could still result in a similar overall entropy. For example the password r$T56? Is 6 characters long with characters drawn randomly from a set of 94 characters (26 lowercase +26 uppercase +10 digits + 32 punctuation and symbols) results in an entropy of log2(94)*6=39.3 bits. The password: “trouble ejections person” consists of 24 letters drawn from words of the English language. The English language is said to have an entropy of 1.5 bits per letter, which would give an estimated entropy of 24*1.5 = 36 bits. However this is for text that forms part of a meaningful sentence, where the correlation between the words would be higher. Thus we would expect the entropy of this combination of these three random words to be slightly higher. Alternatively using the fact that there are 100000 words in a common dictionary one would arrive at an entropy of log2(100000)*3 = 34.54 bits if the attacker knows that exactly three words are used. Entropy can be increased significantly if random letters are shifted to uppercase, omitted or substituted for one another. In any case this password is probably much easier to remember, while its entropy is similar to short random one. Thus it is almost as hard to predict, guess or hack.

Another advantage of this type of password, which we could refer to as a passphrase, is that it is easier to type. It is much easier and quicker to type words than to type truly random characters. However this only applies to computer keyboards and not to other input methods such as the entry pad on a cellphone, where each additional character can result in up to 4 or even 5 key presses.

2.1.3 Character Diversity

Adding more letters to the alphabet used to generate a password also makes it harder to guess, since there are more possibilities for picking each character of the password. This is the motivation behind the policies in place in many password systems that require passwords that contain several different classes of characters e.g., lower case and upper case letters, numbers and symbols.

In practice most passwords in use (about 60%) contain only lowercase letters. This reduces the entropy of the password at most to log2(26*n), where n is the number of characters. Even worse nearly 10% of password contain only numbers giving an entropy of log2(10*n). The next most commonly used combination is lowercase letters and numbers (most often a word followed by digits). This account for about 25% of passwords. Almost all the remaining 5% of passwords use lowercase, uppercase and numbers. Finally a tiny fraction (0.1%) use symbols. [6][7]

A 4 character password drawn from all 94 printable characters is about as strong as an 8 digit numerical password, since 8*log2(10) = 26.6 bit while 4*log2(94)=26.2 bit.

Of course character diversity is problematic on a cellphone because the input system (keypad) is very limited.

2.1.4 Brute Force Attack

The simplest form of attack against password is a brute force attack. This attack tries every possible passwords combination. The number of attempts necessary to crack a password is on the average $c^n/2$ where c is the number of letters in the alphabet and n is the number of characters in the password. However the attacker is guaranteed to succeed after at most c^n attempts.

The fastest current computer CPUs can perform about 100 billion operations per second. Assuming that 1000 operations are necessary per password check this mean that a single computer could check:

$$100,000,000,000/1000 * 3600 * 24 * 365 = 3,153,600,000,000,000 = 2^5 1.5$$

passwords per year. Thus a 51 bit long password could be cracked in a year. It would take 6000 such computers to crack a 64bit password in a year. For comparison a password, if randomly chosen and using 94 the possible printing characters has an entropy of 52 when 8 characters long, 58 when 9 characters long and 65.5 when 10 characters long. A typical lowercase only password when 14 characters long has an entropy of 65.8 bits. Therefore to be fully secure we would require passwords with even higher entropy. This means either using characters beyond the printable character set or further increasing the length of the password. Both of these present difficulties to the user. Another solution would be to change passwords more often, therefore thwarting the hackers attempts. However even if users change their password every month this would give the hacker a 10% chance per month of cracking the password. Which would result in a $1-(1-0.1)^1 2 = 71\%$ chance of cracking the password in one year. Even continuously changing the password still results in a 1-1/e=63.2% chance of break in.

The above refers to randomly generated passwords, we have assumed that each of the 94 characters is equally likely to occur and each letter is chosen independently of the others, however in the case of human generated passwords this is not the case and the entropy is much less. People have a preference for choosing certain characters more often (lower case, numbers) and also to choose dictionary words and birthdates. According to one estimate a typical human generated password would have 4 bits of entropy for the first characters, 2 bits each for the next 7, 1.5 bits each for the next 10 and 1 bit each for any character beyond. Therefore this would require a 48 character long password to ensure 64 bits of entropy.[8] This is too long to comfortably type and probably too long to remember correctly. Of course for an attacker to be able to take advantage of the low entropy requires him or her to use an intelligent search rather than bruteforce, in order to take advantage of this statistical information.

Of course all this only applies if the attacker can perform password checks offline. If the attacker must attempt using the passwords through the system itself the rate or number of trials can be limited. In that case well-chosen passwords are relatively secure. For example many bank PINs are only 4 digits long. However due to the fact that the user has only 3 attempts before the card is “eaten” by the machine this doesn’t present much of a problem.

However in the case of offline password cracking, it seems that given the likely future increase in computing power, passwords will need to be so complex as to be unusable in their current form.

2.1.5 Password Cracking Beyond Bruteforce

Smarter methods of cracking passwords involve either guessing or dictionary attacks. In guessing the attacker uses personal information about the victim to construct passwords that the user is likely to use: birthdate, birthplace, license plate number, family member’s name, and so on. They may also try a list of common passwords (the most common 500 passwords account for over 10% the total).

Dictionary attacks involve using dictionary words and simple combinations of dictionary words and numbers. This works because a significant number of passwords involve a single dictionary words plus some trailing digits, in many cases simply the digit ’1’.

2.2 Security Tokens

A security token is a piece of hardware that authenticates the user, when the user tries to access a service and allows access to the system.

Generally there are three types of tokens: static passwords, synchronous dynamic passwords and asynchronous passwords also known as challenge response.

The static password token is simply an aid to the use of passwords. The token stores the user's password and allows for the use of passwords that are longer than what a user can memorize or comfortably type.

Dynamic synchronous tokens generate a temporary password based on the time. This time needs to be synchronized between the token and the server. These tokens contain a secret similar to that stored on password tokens. This secret is used along with the time to generate the temporary password. Since this secret is contained in the token, only someone who has access to the token can create the proper password.

Asynchronous tokens generate a password based on a challenge from the authentication server. The server sends a random string to the token. The token uses this string and a secret contained in the token, to generate a password. This avoids the need for time synchronization.

In order to be secure the token must contain a unique secret that is not accessible, so that it can't be replicated. In the case of static password tokens it is not possible to restrict access to the secret since it needs to be transmitted directly to the server as part of authentication. Also it is not possible to restrict access when implementing synchronous or asynchronous password tokens on a general purpose computer, as the secret will be stored on disk or in memory. The solution is to use a TPM (Trusted Platform Module), which is a chip dedicated to storing secret keys and carrying out cryptographic operations. Thus the secret can remain in the TPM which carries out the operations necessary to generate the temporary password.

Currents cellphones typically do not contain TPM modules. This means that using them as security tokens leaves the user vulnerable to having their key copied by anyone with access to their phone.

Tokens are extremely vulnerable to loss or theft. If authentication is based solely on the token, then anyone who acquires the token can authenticate to the system as the actual user. Therefore tokens are generally used in conjunction with passwords or biometrics, in order to reduce the chances of compromise. In fact many commercial tokens require a PIN to be entered before use.

The word token in the context of this thesis refers to a piece of data that is used for authentication. A temporary password (as generated by a security token) is a token in this sense.

2.3 Biometrics

Biometric authentication is based on using measures of one or more physical or behavioural traits to uniquely identify a user. This includes fingerprints, iris scans, voice recognition, signatures, etc. This type of authentication is based on pattern recognition. As any recognition problem there is the chance of false acceptance and false rejection.

Since the risks of false acceptance are generally greater than those of false rejection, usually systems are designed such that the probability of false acceptance is much lower than that of false rejection. For example several banks in Japan use palm vein or finger vein authentication. Palm vein authentication as developed by Fujitsu has a false acceptance rate of 0.01177% and a false rejection rate of 4.23%. Finger vein authentication as developed by Hitachi has a false acceptance rate of 0.0100% and a false rejection rate of 1.26%. [9]

The main problem with biometric authentication is that unlike passwords or tokens they are not cancelable or reissuable. Once a biometric trait is compromised there is no way to issue new biometric credentials. This is a serious problem because most biometrics are not secret. For example it is possible to retrieve a person's fingerprint without their knowledge.

Also biometrics cannot be used remotely unless the client hardware is secured. This is because if compromised the client hardware can record the biometric scans and replay them at a later time.

2.4 Relative Strengths and Weaknesses of Passwords, Security Tokens and Biometrics

Passwords' main advantage lies in their secrecy. This is an almost perfect defence against theft. However this is assuming memorization, and doesn't apply if the user writes down their password. The main drawback of passwords is that in order to be secure against search they need to be relatively long. This is particularly difficult for the user when they have multiple accounts with different passwords.

Besides this the main shortcomings of password are that they do not provide compromise detection nor defence against repudiation.

Compromise detection would mean that the user would know when their password is stolen. However unless the user notices odd activity on their account there is no indication that another individual has the password, because the password can be stolen without physically taking anything. For example even if a password is written on a piece of paper, the thief can simply copy the password and leave the piece of paper intact. Or someone can use a keylogger or even a device as simple as a camera to record the user's password as it is typed.

Non-repudiation is the ability of the system to prove that the person accessing the system or requesting a transaction is in fact the user herself, thus preventing the user from denying that they carried out a given action (repudiate). Password do not provide this guarantee, because anyone who has the password can carry out that action. It does not follow that the user willingly gave up their password, it may have been compromised without their knowledge and against their will.

Security tokens' main advantage lies in both strength against search attacks and excellent compromise detection.

They are secure against search attacks because their secrets can be arbitrarily long, since they need not be memorized by the user. Of course the search space is reduced in the case where the user has to manually type the dynamic password generated by the token. This due to the length of the temporary password being less than ideal, because of the cumbersomeness of typing such a long string. This is solved by having the token transmit the temporary password through USB or Bluetooth.

They offer excellent compromise detection since their loss will be detected by the user as soon as they try to log in to the system. Of course this assumes that the token's secret cannot be copied. If it can (as is the case with static password tokens), the token presents all the problems of a password written on a piece of paper.

However they are extremely vulnerable to theft. As anyone who acquires the will have full access to the user's account. For this reason, in practice tokens are almost never used without a second form of authentication.

Like password, security tokens do not provide non-repudiation as the user can claim that the token was stolen.

One final advantage of security tokens is their ability to prevent denial-of-service attacks. In order to prevent brute force searches, many systems limit the number of login attempts. If a user incorrectly enters their password more than a given number of times in a row, then they are blocked from accessing the system. A malicious user can simply make repeated incorrect login requests, until the legitimate user is blocked. What a security token can do to prevent this is to use its secret for data origin authentication (e.g. cryptographically sign the dynamic password). Thus the system can detect whether an incoming password is generated by the token or not.

Biometrics' main advantage is their stronger defence against repudiation. It is more difficult for an attacker to forge a biometric trait, though it is not impossible. Many biometrics are not secret and can be "stolen". The main difficulty for the attacker lies in taking this "stolen" biometric sign and interfacing it to the biometric reader. In the case of fingerprints the attacker may have the image of the fingerprint as obtained taken from a file or lifted off an object, but fingerprint readers are made to scan actual fingers. Although in one case a commercial security door was fooled with a printed version of a fingerprint after it had been licked.[9]

Biometrics are relatively weak against search attacks. This is due to the lack of accuracy of the comparison mechanism. While no two fingerprints have been shown to be identical. Incorrect matches are common in computer systems and have also occurred in cases involving human experts.[10] We can quantify the risk of such an attack as follows. The probability of an attacker's randomly guessed password being the correct one is: P(correct guess) = 1/kp where kp is the password's keyspace (the number of possible passwords). In biometrics a false match is analogous to a correct guess. Thus the "keyspace" of a biometric system is given by: kb = 1/FMR, since P(false match) = FMR = 1/kb. Applying this to the finger vein authentication above, this would give a keyspace of 1/0.01% = 10000.[10] Thus an attacker with a database of fingerprints could gain entry to any user's account after trying about 10000 different fingerprints on average.

Finally biometrics do not provide compromise detection.

As each factor has its strengths and weaknesses, it would make sense to use more than one type of authentication in a system. Passwords are secure against theft. Tokens provide compromise detection. Biometrics provide non-repudiation.

3

Mobile Social Authentication

Our work combines two techniques that have been widely used separately: using the user's mobile phone as an authentication device and using the user's social network as an authentication factor.

3.1 Mobile Phone as Authenticator

There are normally two ways in which to use a mobile phone for authentication, but both of them involve the user proving that they are in possession of the device. The first is to use use the mobile network itself for authentication, the second is to use the phone as a security token.

In addition to these traditional methods, authentication can also be based on location or biometrics .

Location can be obtained from GPS, cell-tower information or static beacons.

Biometrics that have been found suitable for mobile phones include: voice recognition, face recognition, eye (iris) recognition, keystroke patterns and acceleration or gait.

3.1.1 Authenticating through the Mobile Network

Authenticating the user through the mobile network involves contacting the user at authentication time. This can be achieved by either sending the user a one-time code by SMS [11], or by calling the user and requiring them to enter a PIN [12].

A problem with the first approach is that SMS traffic may be snooped. A solution is to encrypt and possibly sign the SMS messages. There are both PKI and symmetric key based methods.

The symmetric method is based on a shared password, used to generate a key. This has the advantage of protecting the user if the phone is stolen as the key is not stored unencrypted on the phone. It requires software to generate a key from a password input by the user and to encrypt/decrypt the data using this key. On the other hand this limits the strength of the key.[13]

The PKI based method requires a private key on the phone. It is equivalent to treating the phone as a token.

3.1.2 Mobile Phone as Token

Using the phone as a token usually involves making it carry a public/private key or a certificate. At authentication time the user is asked to prove that they have the private key, thus proving they are in possession of the phone. However as we will see in Chapter 7, without a Trusted Platform Module (TPM) that restricts access to the key, anyone with access to the phone will be able to read the key and possibly transfer it to another device.

In [14] the authors propose a scheme whereby a certificate is issued to each phone that binds a public key to the device's Bluetooth MAC address. Thus even if an attacker obtains the user's certificate it will only work on the phone with that particular MAC address. However if the attacker can modify his Bluetooth stack to report the user's MAC address then the same problem occurs. The solution, securing the Bluetooth stack, is similar in nature to using a TPM.

The alternative when a TPM is not available is to add another layer of encryption by encrypting the private key with a password. While this adds some additional security it has the shortcoming of allowing an attacker to perform a bruteforce attack on this encryption if they can get the encrypted private key. Thus in that case the security of the system would be limited to the security of the password.

With a TPM the phone can be used to securely store the user's cryptographic credentials, such as private keys. Thus the phone can provide the functionality of a smartcard, saving the user the need to carry an extra piece of hardware. In addition the phone can provide an interface that helps the user update and manage their credentials [15].

A distinction can be made between the device's identity and that of the user [16]. In this case the user proves their identity through another means (e.g. a username and password) and the private key and certificate prove the identity of the device (e.g. the phone). Thus access can be restricted based on either the user, the device or both.

3.1.3 Network and Token Authentication

There is at least one proposed system that is a hybrid between these two approaches [17]. The idea is to generate a one time password using certain information unique to the user's phone and a PIN number. If this should fail then a one time password can be sent to the user's phone by SMS.

3.1.4 Location Based Authentication

As its name implies location based authentication involves determining the user's location and making access conditional to it. In this form it is only useful when the location is controlled and physically secure. This would apply to restricted military installations and server farms. In this case all that is necessary is to determine the user's proximity to the restricted area. This can be done with a trusted hardware sensor placed at the site.

In the general case where the user wants to log in from a location that is not secure, this method will not make sense. But we can still make use of location information in the authentication process. But instead of granting access based on location, we would deny access based on location. For example a user might only access his online bank account from her home or office, in that case we can deny access from any other location. This would be based on the location reported by their cellphone. So a thief in addition to having to steal her cellphone, would also need to be physically present at the user's home or office, which is more difficult, and more importantly risky, for the thief.

Sources of location information can be GPS, celltowers, beacons and proximity sensors.

GPS based location is mainly available in open areas as it requires line of sight access to the satellite signals, although some sensitive GPS devices have some reception indoors, especially near windows. Since GPS is a one way system (i.e. the user's device only receives satellite signals and computes its location itself), it can send fake location information. Preventing this requires a trusted or tamper-proof GPS device. Hacking such a trusted system would present a difficult challenge, since it requires generating fake GPS satellite signals from at least 3 sources. Difficult but not impossible for a resourceful attacker. This attack can be prevented by using the antispoofing information included in the GPS signal. This information is ignored by civilian users, and requires an encryption key that is only available to the defence establishment.[18]

Celltower information gives coarser location information but generally works indoors. While it does not have global coverage, thus excluding very rural or remote users, it usually covers places where most of the population lives. Since there is secure bidirectional communication, the celltower determine the user's position with high confidence. Faking location requires cloning the phone's SIM card.

Beacons provide an alternative to GPS in an indoor environment. Being unidirectional they present the same problems as GPS. However the signal from the beacons can be cryptographically signed preventing a "fake signal" attack.

The operation of proximity sensors is the reverse of the beacons. They receive a signal from the mobile device. Usually the system will issue a random challenge to the phone, which it will forward to the sensor to prove its identity. Alternatively the device can send a signed timestamp to the sensor.[19]

3.1.5 Mobile Biometrics

As in the general case, using biometrics on mobile phones basing authentication on measurements of one or more of the user's biological traits. Many different characteristic traits have been proposed for use with mobile phones: voice recognition[20], face recognition[21], iris recognition[22], keystroke patterns[23], arm swing acceleration [24] and gait[25]. The last two are uniquely applicable to the mobile environment.

An important problem that needs to be solved is the provision of reliable and tamper-proof biometric scanners on the mobile device. This would involve some combination of a trusted computing platform and trusted biometric reader. Both are necessary if we want to perform matching on the phone itself. Since in this case the phone tells the server that the biometric matched, the server needs to ensure that the verification code has been tampered with, nor is the biometric input been replayed. Performing matching on the server side would require only a tamperproof biometric reader. In this case the only thing we need to prevent is the replay of a previous reading by the device. This can be achieved by signing the reading with a timestamp.[26]

3.2 Social Network as Fourth Factor

It seems that making use of a user's social network to facilitate authentication hasn't been explored as fully. This may be because unlike the previous case this isn't a simple extension of existing techniques and technologies. Rather it is a completely new approach to computer authentication.

There are two different ways this can be used: one involves contacting the members of the user's social network in order to securely authenticate the user while the other involves making use of the user's account on a social networking site to securely contact members of the user's social network.

The first approach is the one proposed in the RSA paper [1]. As described in the introduction it involves the user contacting a friend when he has forgotten his token. The friend logs in with her own token, requests a "vouching" code for the user and relays this to the user, who can use this to log in temporarily. Our approach also falls under this category. The user obtains “vouching” tokens from their friends, whenever they have a phone conversation or a Bluetooth sighting. And these tokens are used to log in. The important feature of this approach is that the user’s friends are contacted directly (by phone or Bluetooth), and this contact is used to prove the user’s identity when logging in to a site or service.

The second approach contacts the user’s friends through the social networking site. Social network sites provide peer-discovery (finding friends) and secure messaging (instant messages). Therefore it is possible to set up a secure communication channel with a friend by sending them a key through the social network, and using that key to encrypt subsequent transmissions that will travel through the open Internet. This can be done manually by the user, or it can be implemented in the application that needs to send the encrypted data. A special API can be built to facilitate the interaction between applications and social networking sites [27].

Another technique that has been proposed is to test the user’s ability to recognize their friends in pictures that have been tagged on a social networking site. This approach faces two problems. The user’s close friends may also recognize most of these faces, and could login to the system in her place. The other issue is that facial recognition software could be used to automatically match the faces.[28]

3.3 Social Authentication on Mobile Phones

Finally we can consider the combination of mobile phone and social networking for authentication. One of the goals of the Reality Mining project is to measure users’ social networks using mobile phones. This can be done based on call patterns (which indicate who the user was talking to) and Bluetooth sightings (which indicate who the user was close to). To apply this to authentication one can take these measurements and compare them to typical values for the user. For example one can measure the devices (friends) in the user’s Bluetooth range and compare this to the value for a typical day [29]. Our scheme considers both call patterns and Bluetooth sightings, but does so in a slightly different way. We count the number of conversations and sightings in a given time period and base our authentication on whether this number exceeds a certain threshhold [30].

4

Proposed Scheme

In this chapter we will outline the details of our authentication scheme. First we will focus on social authentication based on telephone conversations and Bluetooth sightings. We will then consider additional factors that can be used in order to minimize the risk of intrusion.

4.1 Social Authentication

The goal of this scheme is to leverage the functionalities of a Bluetooth capable cellphone in order to automate the process of vouching. The user will obtain vouching tokens from friends and will use them together with a PIN to log in to a secure service.

4.1.1 Obtaining Vouching Tokens

The user starts by declaring a list of friends that will "vouch" for him. This list is stored on a central server. After a phone call with one of these friends the user will receive a token indicating that this communication took place. A token is only issued after a phone call that is longer than a minimum duration. This duration is determined by analyzing the distribution of the user's call durations. The idea is that it is unlikely that an intruder will be able to make a phone call long enough to receive a token, without alerting the other side that something is wrong.

While the use of vouching tokens from friends is a form of fourth factor authentication (i.e. someone you know), in this case it is also implicitly a biometric factor (i.e. something you are). This is due to the fact that in the process of obtaining the vouching token, the user's voice is recognized by her friend (human voice recognition). This has two advantages. Humans are better at recognizing voices than are machines, and secondly an attacker's failed attempt will be instantly detected by the user or her friends, and most probably reported on time.

Similarly after seeing a friend using Bluetooth a token will be received confirming this sighting. Bluetooth sightings are trickier because the long range (10m) doesn't imply that the users actually made contact. Thus the Bluetooth sightings are augmented in two ways. First a rough estimate of the distance of the other user is made by measuring the time it takes to establish a Bluetooth connection. Secondly the user is prompted to confirm the sighting of the other party. After both of these take place the vouching messages are exchanged.

In this case the Bluetooth tokens are also a form of location based authentication, with the added advantage that the location is confirmed and defined by the user's proximity to their friends.

One of the reasons to include this proximity based information is that in some locations the user is surrounded by many of their friends and is unlikely to have phone conversations with them, since they will simply talk to them directly. In this case detecting their presence will allow for authentication to take place, and avoid an unnecessary traffic load on the telephone system.

Figure 4.1 shows the process of obtaining tokens from both conversations and Bluetooth sightings.

4.1.2 Authentication Using Tokens

The user authenticates himself to the central server by sending the required number of fresh tokens, along with entering his PIN number. After repeated errors in the PIN number the tokens become invalid and new tokens must be obtained.

The central server (e.g. online banking website) needs to verify the authenticity of

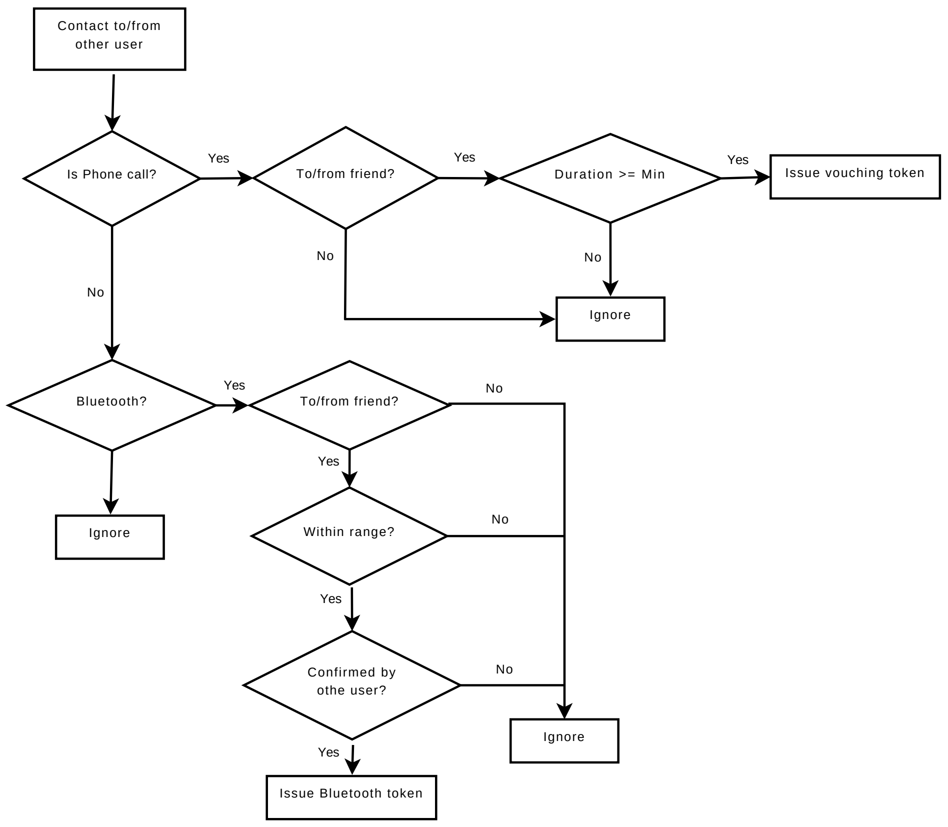

Figure 4.1 – Process of Obtaining Tokens

these tokens. There are two methods by which this can be accomplished. The server can verify authenticity from the tokens it receives from the user in question or it can contact the friends who are doing the vouching. The latter requires multiple session to be setup and more importantly requires all the friends to be online at the time of authentication. Therefore, we use the former method.

Even in this method there are two possibilities. The user can obtain the tokens and then send them to the server, or the friend can send the tokens to the server at some point before authentication. The disadvantage of this is that the user doesn't know whether or how many contacts have been registered with the server. This can be compared to the situation where a student applying to University gets signed and sealed letters of reference and mails them herself rather having each referee mail the letter of reference to the University.

The electronic equivalent of the referee physically signing and sealing a letter, is the vouching party digitally signing the message and then encrypting it. It is needless to say that this is safer than the physical method, since the server can reliably verify the signature, whereas a physical recipient of the letter would need to have a copy of all the signatures of the referees beforehand in order to be able to do the same, which is almost never the case in practice.

This digital signing and encryption requires the phones to perform signing and encryption operations. One can tradeoff security vs simplicity. Since the time of contact is not as important as the number of contacts, the server can generate a set of tokens and send it to each vouching device periodically. For example the server can send each device 100 tokens a week, and the rule can be to send the first token on the first contact, the second token on the second contact and so on. When the user logs in they send in the tokens and once the server verifies that they are in the current set of tokens, it knows how many contacts the user has had this week. This avoids the need for signing tokens, however this doesn't get rid of the encryption requirement, as we still need a way to get these tokens from the server to the device, without malicious users being able to eavesdrop.

This scheme requires a lot of data to be transferred periodically. We use hashing chains to alleviate this problem. Only one token is sent by the server to each phone. This token is then hashed repeatedly to generate the other 99 tokens that will be used. The server can perform this same hashing operation to verify that each token was generated from the original token.

4.2 Supplementary Factors

In addition to the “vouching” tokens which are “someone you know” (fourth factor authenticators), the PIN which is “something you know” and the private keys (see Chapter 6) which are “something you have” we can also make use of “something you are” (biometrics).

Just as it is common for a user to carry a cellphone with them, we would like to make use of other devices within the user's reach. For example it is likely that when using this system to authenticate to an online banking website, the user will be using her laptop, which may be equipped with a fingerprint scanner. A fingerprint scan can then be used in conjunction with the "vouching" tokens as an additional factor.

Use of a special-purpose wearable authentication device in the form of a wristband has been suggested [31]. This device would take fingerprint scans, but would also monitor the user's vital signs, including: heart rate, skin temperature, body capacitance and acceleration. Using these readings the person wearing the device is transparently authenticated once they are within radio range of the target computer.

Footstep characteristics have been used for personal identification [32]. Many users carry an iPod or palmtop device with them. These may be equipped with a pedometer that measures the user's footsteps. The characteristics of the footsteps can be analyzed and compared to the user's baseline measurements. In the event of a match the user is authenticated. The use of this feature further improves the reliability of the authentication.

In our system the primary means of authentication are the "vouching" tokens and the PIN. For additional security the user may opt to require one of the above biometric factors to authenticate herself. For example consider a user with a laptop and pedometer system. In addition to the social authentication if either the laptop recognizes the user's fingerprint or the pedometer's readings match the user's gait, then the user will be authenticated.

5

Simulation on Reality Mining Dataset

The Reality Mining project at MIT's Media Laboratory followed 100 cellphone users over the 2004-2005 academic year. These 100 students, faculty and staff were each given a Nokia 6600 smartphone with an installed application that would log their usage. Data collected includes: call logs, Bluetooth devices within range, cell tower information, application usage and phone status [2].

Among the data collected by the Reality Mining project, the readings of most interest to us are the call logs and the Bluetooth device sightings as these would lead to the issuing of "vouching" tokens.

Looking through the data, we see that there are many days over the course of the year, when there is no call data nor any Bluetooth sightings. So the first thing we did was determine the number of days that each phone logged any relevant data. This is shown in Figure 5.1 below. As can be seen the number of days with data varies between 0 and a little over 250 for the different users. Looking more closely we see that 8 users never logged any data.

In order to simulate the issuing of a "vouching" token based on phone conversations. We tried to establish a list of friends for each user. We used the ten most popular numbers that they talked to, which we assume to be their friends. This is the best case scenario, in actuality the user may call a certain number very often and yet this number may not correspond to a friend. So the actual results may be slightly worse than these simulation

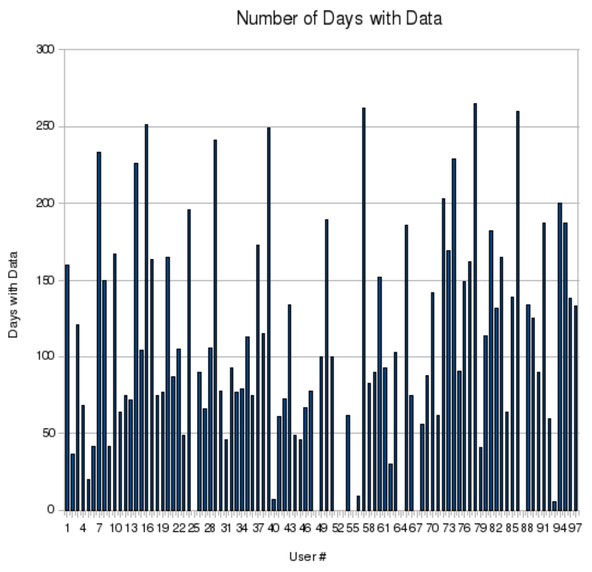

Figure 5.1 – Days with Data

would indicate.

We then established the minimum duration of a call before tokens are sent or received. When choosing the minimum duration there is a trade-off between security and convenience. If we choose it too small then an impostor can make random calls to the user's phonebook and then hang up after receiving the token. On the other hand if we put the threshold too high then very few tokens will be generated, and it will take a long time for a user to have enough tokens to be authenticated. In this case we tallied the durations of each user's calls and took the 25th percentile as this minimum duration. We do this to eliminate the effect of wrong numbers and such very short calls, and at the same time keep the probability of false rejection small.

With these two pieces of data in hand we then looked at each day and considered the two days immediately preceding it. If two friends or more friends were called in the current day or those two preceding days and each at least two of those calls were longer than the minimum duration, we assumed that two or more tokens were generated. In this case we only require two tokens to authenticate and thus the user can be authenticated on

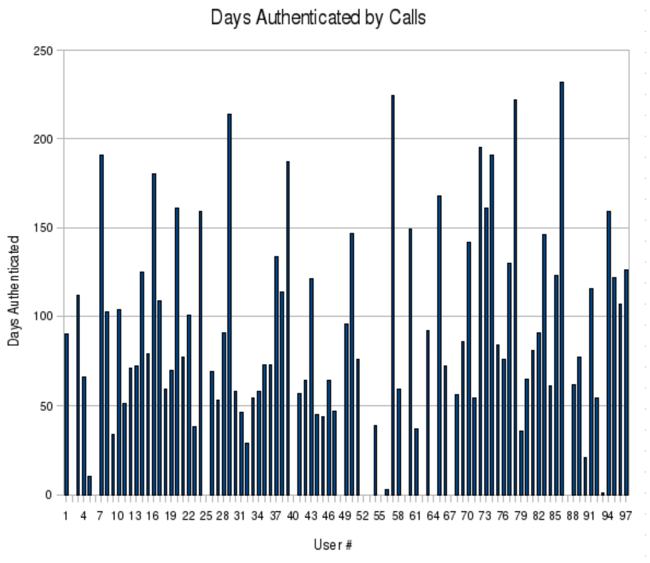

Figure 5.2 – Days Authenticated by Calls

that day. Figure 5.2 illustrates the number of days where the user could be authenticated based on phone conversations.

On the average the users are able to authenticate themselves on 74% of the days where they logged phone calls. We are assuming that the days where the users did not log any phone data, were days where their phone was off or the logging application was not running.

The next step was to investigate the effect of adding Bluetooth device sightings to the authentication process. Again for each one of the top ten devices that the user sees in the current day or the previous two days, the user receives a token. If the user has two tokens, either two from phone calls, or two from Bluetooth sightings or one of each, then the user is assumed to be authenticated for that day. Figure 5.3 displays the number of days where the user could be authenticated based on both factors.

We see that the numbers are much closer to those of Figure 5.1. In fact on the average users can authenticate on 95% of the days with bluetooth sightings or phone conversations. Again the assumption is that days without either of these communications were days where

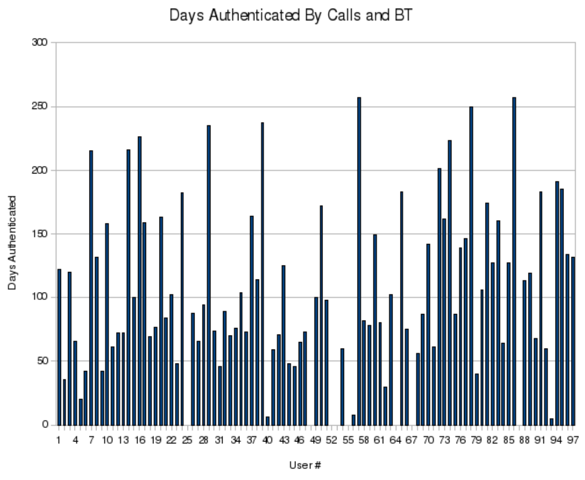

Figure 5.3 – Days Authenticated by Calls and BT

the user was not using their phone or the data was not recorded.

In the above we required two tokens for authentication. Obviously there is a trade-off between the number of tokens required and the probability of successful authentication. We varied the number of tokens required between 1 and 10, and calculated the probability of authentication. As Figure 5.4 shows, the probability of authentication varies between 98% and 50%. We chose 2 tokens above to have a high probability of authentication.

There is a similar trade-off between the number of days a token remains valid and the probability of successful authentication. We required two tokens and then varied the number of days between 1 day and seven days. As can be seen in Figure 5.5, the probability of authentication varies between 82% and 98%, and levels off after 3 days. We again chose 3 days (the current day plus the two preceding ones) to have a high probability of authentication.

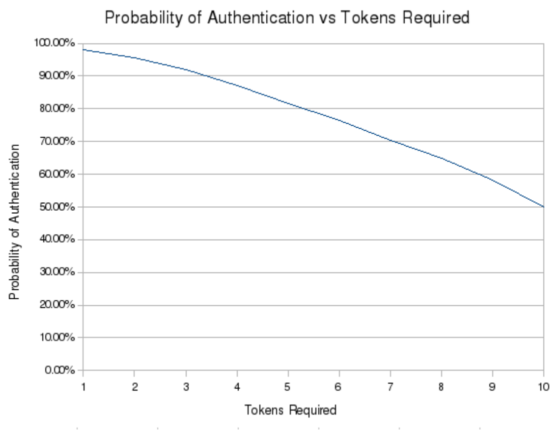

Figure 5.4 – Probability of Authentication vs Tokens Required

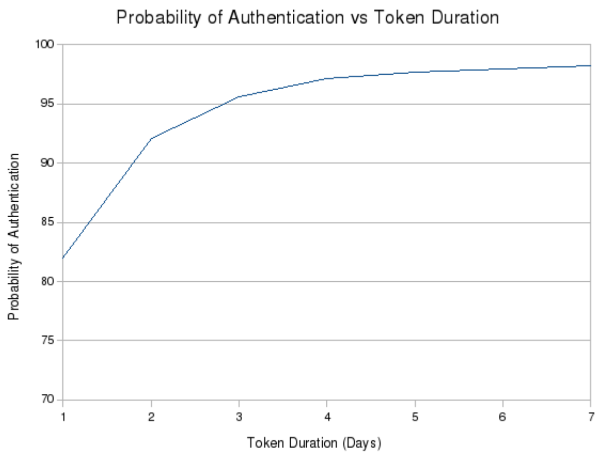

Figure 5.5 – Probability of Authentication vs Token Duration

6 Implementation

Our main implementation uses PKI (public key infrastructure) to provide confidentiality and integrity. In particular we use an implementation of the RSA encryption algorithm written in Python (Pys60, Python for Nokia S60 phones). We chose to code for Pys60 because it has modules that allow easy access to a number of phone features that we need, including: call logs, SMS inbox, SMS messaging and Bluetooth. Additional modules can be written in Symbian C++, in order to either speed up critical sections of the code or to provide access to low-level hardware not accessible from the standard modules. It is important to note that our system could be implemented using other programming languages such as C or Java for phones without a Python interpreter. At the end of this section we will present an alternate implementation using hash chains that avoids much of the public key computations.

6.1 Issuing Tokens

Each node has a public/private key pair. A copy of the public key resides on the server. When a user Bob (B) wants to give a token to user Alice (A), Bob signs (encrypts with his private key) A's name and the current time T_b: $K_{BS}(A, T_B)$, and then encrypts this with A's public key $K_{AP}(K_{BS}(A, T_B))$ and sends it to Alice. Only A can decrypt this to retrieve $K_{BS}(A, T_B)$.

Tokens are issued when a phone conversation with a friend occurs and it is over the minimum duration, or a friend is sighted over Bluetooth and is within a small distance.

Our software continuously scans the call logs and looks for calls that are to or from friends. It then checks the length and if the call is long enough a token is issued as above.

Similarly our software continuously scans for friends over Bluetooth. To determine whether the friend is close or not we time the duration of the Bluetooth obex discovery call. The Bluetooth obex discovery function, takes the target device's MAC address and returns a list of obex (Object Exchange) services available on that device. It turns out that the weaker the signal between the phone and the device it is trying to discover the longer this function takes. We call the function four times and time the last three. If all three calls took less than 0.06 seconds, then we ask the user if they would like to send a token to the corresponding friend. 0.06 seconds was chosen because it was determined experimentally to be a typical value when the device and the phone are within line-of-sight of each other.

6.2 Using Tokens to Authenticate

When A wants to use a token from B, she concatenates the current time T_a and signs with her private key K_{AS}: $K_{AS}(K_{BS}(A, T_b), T_a)$. And then encrypts with the server's public key K_{SP}: $K_{SP}(K_{AS}(K_{BS}(A, T_b), T_a))$. This is illustrated in Figure 6.1. The server can decrypt this to recover T_a, A and T_b. The server checks that A matches user A. That T_a is close to the current time. And T_b is within the allowed lifetime of a token.

Then the server issues a challenge for the PIN. If A responds correctly then she is authenticated.

If not the challenge is repeated. If A fails repeatedly then the tokens is invalidated and new tokens are required. This is done by requiring tokens with timestamp newer than T_b.

6.3 Hash Chain Implementation

In this case each user still has a public/private key pair, but it is only used to set up shared secrets between the user and the server. Periodically (e.g. ever 3 days) the user's phone

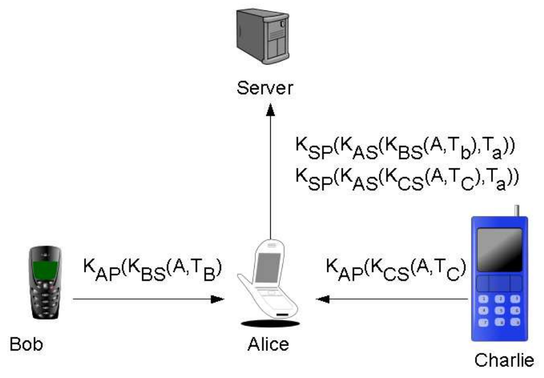
Server
$K_{SP}(K_{AS}(K_{BS}(A,T_b),T_a))$
$K_{SP}(K_{AS}(K_{CS}(A,T_C),T_a))$
$K_{AP}(K_{BS}(A,T_B)$
$K_{AP}(K_{CS}(A,T_C)$
Bob
Alice
Charlie

Figure 6.1 – Using Tokens to Authenticate

contacts the server and establishes a separate secret for each of her friends. To do this the server signs the secret S with its private key S_S, and then encrypts it with the user's public key U_P: $U_P(S_S(S))$. Only the user can decrypt it. Once decrypted the signature can be verified to make sure the message came from the server. The phone then takes each secret S and hashes it (using a hash function like MD5 or SHA1) to obtain $H_1(S)$. The phone then takes the hash of the hash to obtain $H_2(S)$ and repeats this n times, to obtain $H_3(S)$ through $H_n(S)$. These hashes will be used as tokens.

Tokens are issued as follows. After a successful phone conversation or Bluetooth sighting between Alice and Bob, they each exchange one of the hashes. Assuming this is their first token exchange Alice will send Bob $H_n(S)$, and so will Bob. At their next exchange they will send $H_{n-1}(S)$. And this will continue through $n-2$, $n-3$, $\ldots$,2,1. The reason they start from the end of the hash chain is that knowing S, one can generate all the hashes, but knowing $H_n(S)$ reveals nothing about the other hashes. So if Alice uses $H_n(S)$ to authenticate, and the token is invalidated (because she mistyped her PIN), the server can require a fresh token by requiring the next token to precede $H_n(S)$ in the hash chain. So Alice could then try to authenticate with $H_{n-1}(S)$ if she has it.

The user can use these hash tokens to authenticate because the server can verify that the tokens were issued by the correct friends, since each token could only have been generated from an original secret that was previously sent to each of the user's friends by the server in a secure manner.

To transmit these hash tokens to the server for authentication, the user could, as in the case of regular tokens, take the required number of tokens from different friends and sign them with her private key and then encrypt with the server's public key. This will ensure that the tokens can't be intercepted and that they are being used by the correct user. The server could decrypt this, then verify that the request is coming from the right user and that there are enough tokens.

However to further avoid public key computations, the user could establish an extra shared secret with the server, during the periodic secret exchange. The user can then use this secret to include a keyed-Hash Message Authentication Code or HMAC of the message to the server. This is a form of Message Authentication Code, which provides both integrity and authenticity of the message using a secret key, just as would have been provided by the signing and encryption using public/private keys.

6.4 Python Modules

The software consists of two programs a client running on the phones and a daemon on the authentication server.

The client usually runs one script "sendtoken.py" in the background that analyzes the phone logs, scan for Bluetooth devices and sends tokens accordingly.

Another script "authenticate.py" is run when the user wishes to authenticate to the server. It gathers the necessary tokens and PIN and uses them to authenticate.

The server runs "auth.py" as a daemon which accepts TCP connections from clients. It receives the tokens, verifies them, sends a challenge for the PIN, and authenticates the user if everything checks out.

In the case of the hash chain implementation, the client periodically runs "hashchain.py" to set up the shared secrets with the authentication server. Typically this is done when turning on the phone, and once in a while. For example after sending 100 tokens to a friend, if the hash chain length is 100, or equivalently after a certain period of time, say one week.

Once this is done “hashtoken.py” is substituted in place of “sendtoken.py”. The difference being that instead of generating (signing and encrypting) a new token, we simply select the next token in the chain and send that instead.

Finally “hashauthenticate.py” is run by the client to authenticate. It performs the same function as “authenticate.py” but using the hash tokens.

On the server side we have two daemons to implement the hash chain functionality: one to generate and transmit the shared secrets “hashsecrets.py” and another to verify the tokens and PIN “hashauth.py”.

In the following subsections a brief outline of the functionality of each of the above-mentioned Python modules will be given. This includes the details of the built-in modules that are invoked.

6.4.1 Sending Tokens: sendtoken.py

The sendtoken script makes use of the following modules: logs, messaging, rsa, socket, time, e32 and pickle.

“logs” allows access to the phone’s logs. We only use the incoming and outgoing call logs. In order to reduce processing, we only look at new calls, by taking the calls that have occurred between the previous run and the current time. We scan for calls from friends and then verify the duration of the call. Based on this we can decide to generate and send a token to the friend.

We use “rsa” to generate the token. First we take the current time and concatenate the friend’s number. Then we sign with our private key. Finally we encrypt with the friend’s public key.

One of the shortcomings of the “rsa” module is that it does not perform padding and so the same plaintext will always encrypt to the same cipherstring. So if the attacker knows the time they can try to generate a set of tokens with the plaintext (time number) where number is the phone number of one of the user’s contacts. Comparing these tokens with the actual token transmitted the attacker may be able to determine to whom the user has spoken. A simple solution is to append n random bits to the plaintext and have the recipient discard those extra bits after decryption. This would force the attacker to generate 2^n messages before being able to check for a match. In the general case this may not be sufficient, since a sophisticated attacker can take advantage of the predictable structure of a message.

After this the message is transmitted to the friend using “messaging”, which is a simple interface to the phone’s messaging services. In our case we use the function sms_send to send the token as an SMS. In order for tokens to be later identified by the receiving party, the token is prepended and appended with the string “token”. However before sending the token the user can be prompted to verify whether the person on the other end of the conversation was indeed the friend to whom the token is to be issued. An example of why this would be necessary is if Alice received a call from her friend Charlie from Bob’s phone. In this case Alice’s phone will ask her whether or not to send a token to Bob. This will alert Alice that Charlie is actually using Bob’s phone (either maliciously or not). Alice has to decide one way or another.

While SMS is becoming less and less expensive and many service providers allow users practically unlimited SMS traffic, mobile internet or wifi can be used in cases where the cost of SMS is still a concern. Theoretically an effective way to transmit this limited information would be to overload it onto the voice channel itself, by modifying the phone hardware to use part of the voice channel for data transmission. An added advantage would be that since the rate is low, transmitting the token would require a call with a specific minimum duration.

In the case of Bluetooth sighting we use the bt_obex_discover function, which is part of the socket module, to try to establish a connection to each of the user’s friends in turn. If any of these connections is successful we repeat the process three more times and measure the connection setup time (using “time”). If the times are below a certain threshold (0.06 seconds), which would be typical of line-of-sight between the two-phones, a token is sent. Again the user is prompted for confirmation before the token is actually sent to ensure that the user has indeed seen her friend.

The token is generated in exactly the same way as in the call log case. But is sent using bt_obex_send_file, which sends the token directly over Bluetooth saving the need for an extra SMS.

After analyzing the incoming and outgoing call logs, and scanning for Bluetooth devices the script sleeps for 10 seconds. Using the ao_sleep functions from the "e32" module. The e32 module includes Symbian specific functions. In Pys60 the standard sleep function locks the phone, while the e32 ao_sleep functions uses a timer that runs in the background and doesn't lock the phone. So the user can still use their phone when the script is not running.

Bluetooth scanning is not done after every wakeup. If a token has already been granted we wait 2 minutes (12 times), if a token was not granted because the user didn't respond to a sighting we wait 1 minute (6 times), and if the other device was near but out of range we scan again on next wakeup.

The "pickle" module is used in Python to serialize and deserialize objects. We use it to store our data structures (arrays or hashes of keys, numbers, Bluetooth MACs, etc.).

6.4.2 Client Authentication: authenticate.py

The authenticate script uses the same modules as sentoken, with the addition of "inbox".

"inbox" is used to access the user's SMS inbox and retrieve all the tokens that have been received. This is done by checking for messages that begin and end with the string "token".

After removing the delimiting string, each token is decrypted with the user's private key. The friend's number as well as the current time is concatenated with the decrypted token, and then the whole is encrypted with the server's public key.

Once all the tokens are in the proper format, a TCP connection is made to the server, and the tokens are sent. The server verifies that the tokens are signed by the user's friends and have valid timestamps. Then a challenge is sent for the PIN. The user responds to the challenge, and if successful the user is authenticated.

If the user doesn't have internet connectivity, SMS can be used for communication between the client and server instead. Many providers (e.g. Fido) offer unlimited SMS with many of their plans, but mobile internet still remains somewhat expensive, and wifi is only available on the newest phones.

6.4.3 Authentication Server: auth.py

The authentication server waits for connection from clients. It receives their tokens and verifies that they are from the user's friends and that their timestamps are valid. It then generates a random challenge. This is basically a random string encrypted with the PIN and then encrypted with the user's public key. The user must first decrypt the challenge with their private key and then decrypt the inner message with the PIN, perform some function on the string (possible add a constant, reverse the string or compute a hash) and then reencrypt this new data with the PIN, and then encrypt with the server's public key. The server can then carry out a similar process to verify that the user has both his private key and PIN. One this is done the user is authenticated and can access the system.

6.4.4 Establish Shared Secrets: hashchain.py

The hash chain implementation replaces the public key encryption and signing with tokens generated by hashing a shared secret. This requires the server to establish shared secrets with each user.

We do this through the use of public key encryption. The benefit of this is that public key encryption and signing is only necessary once a week or once every 100 tokens, cutting down on processing time and battery usage.

The client script "hashchain.py" receives three messages from the server. The first contains the expiry date or period of validity for the secrets (initially 48 hours). The second contains the secrets themselves (one per friend). The last one contains the hash counters for each friend (initially 100), this is the number of tokens that can still be issued based on this secret.

Each of these messages is decrypted using the user's private key and then the signature is verified using the server's public key.

Once the secrets are received the hashes are generated and stored in a file. These hashes will later be sent as tokens by “hashtoken.py”. The hashing function used must be the same as the one used to verify tokens by the server. There are different hashing functions. Standard ones include MD5 and SHA. For demonstration purposes we use MD5 as it is part of the standard Python distribution. Newer versions of Python also include SHA1 (and even SHA2), which should be more secure against attack. Switching hash functions is very straightforward and only involves changing a few lines of code.

If the user runs this script again, she will receive an updated version of the data. This will give an updated value for the remaining duration of validity and the number of tokens that can still be issued for each secret. If the period of validity has expired, the user will receive a new set of secrets with a validity of 48 hours and a counter value of 100.

6.4.5 Hash-based Tokens: hashtoken.py and hashauthenticate.py

When sending hash-based tokens to another user the script “hashtoken.py” simply loads the array of hashes and the current counter. It then sends the token at hash[counter], and decrements counter.

When counter reaches zero, the script runs “hashchain.py” to obtain a new set of secrets.

When using hash-based tokens to authenticate to the server the client script “hashauthenticate.py” encrypts the tokens with the server's public key and sends them to the server. As previously mentioned the public key encryption in this step could be replaced with a secret key encryption or an authenticated hash.

6.4.6 Hash-based Server: hashsecrets.py and hashauth.py

On the server side “hashsecrets.py” sends the most recent data and secrets to the client, and generates new secrets when necessary.

At authentication time “hashauth.py” receives the tokens from the client and verifies them by hashing the user's friend's secrets.

For example if the user Alice used token 95 from Bob and token 47 from Carolyn. The server hashes Bob's secret 95 times and compares this to the token from Bob and hashes Carolyn's secret 47 times and compares this to the token from Carolyn. If all the tokens match the hash computation the server grants access.

7

Implementation Results

Having implemented the system we can now consider its practicality. One major factor is the battery usage. Another is our ability to measure distances using Bluetooth. And most importantly we must consider the false acceptance and false rejection rates.

7.1 Battery Life

Both public-key cryptography and Bluetooth scanning are considered big battery drains. We will consider each in turn.

7.1.1 Public-key cryptography and battery life

We use public-key cryptography when generating tokens and when using tokens to authenticate. To test the effects of generating tokens on the phone's battery life we ran the code to generate a token in a continuous loop. We ran this on a phone with a full charge and let it run until the battery ran out. The result was that 7140 tokens were generated in 18710 seconds (or 5 hours 11 minutes and 50 seconds). This means that it takes 2.62 seconds to generate a token. This includes signing and then encrypting the message.

How this will affect battery life depends on how often the phone will be required to generate tokens. Even generating as much as 700 tokens per day would only drain 10% of the phone's battery.

7.1.2 Bluetooth scanning and battery life

Our program continuously searches for friends within Bluetooth range. Therefore to measure Bluetooth scanning battery usage we can simply run our program and see how long it takes to drain the battery. Scanning for Bluetooth connections every 10 to 20 seconds, drained the battery in about 32 hours. This is a little over 3% battery usage per hour.

Assuming the above 700 tokens per day, and 16 hours of daily use between charges the program as it stands would drain 60% of the phones battery in a typical day (50% Bluetooth scanning, 10% token generation). This leaves 40% of the battery for talk-time and other applications. Obviously the frequency of the scanning could be decreased to increase the battery life. But it seems that the program as it stands is still usable, if a little battery hungry.

The Bluetooth devices embedded in current mobile phones are usually class 2 devices that operate over a 10m range using 4 dBm (2.5 mW). However there are lower power modes of operation for Bluetooth. Class 1 devices, for example, operate over a 1m range using 0 dBm (1 mW), while new Wibree/Bluetooth low energy devices operate at a similar range with power as low as -6 dBm (0.25 mW). Using such low power devices would result in much longer battery life (up to 10 times as much).

For our application we would want a slightly larger range than 1m, probably as much as 2m or even 3m, to give some allowance for the users' mobility. In other words we want to make sure that Bluetooth sightings can be recorded without requiring both users to be within 1m of each other for 10 to 20 seconds.

Assuming we reduce the Bluetooth power consumption by a factor of 3 or 4, which would correspond to using Wibree/Bluetooth low energy devices operating at reduced distances, this would decrease the total daily battery consumption down to 20% to 25% down from 60%. In that case, Bluetooth would only account for half of the power consumption. It would then make sense to consider switching from the pubic key implementation to the hash chain one to reduce the CPU power consumption if further reductions in total energy use were to be made.

7.2 Estimating Bluetooth Distance

The main reason we want to estimate the distance of other Bluetooth devices is so that the user will not be prompted to send a token when a friend is within Bluetooth range, but outside of visual range.

The signal strength measured at the received depends on the distance and is inversely proportional to a power of the distance:

$$P_{r(d)} = \frac{k}{d^\alpha}$$

Where d is the distance between the transmitter and receiver, k depends on the transmitted power, gains of the antennas as well as, possibly, the wavelength of the signal and α is the loss exponent, which depends on the environment. This exponent usually ranges between 2 and 4 (where 2 is for free-space, and 4 is for lossy environments). In some buildings and indoor environments α can be as high as 6, while in a long corridor or tunnel it can be lower than 2. This is due to the tunnel acting as a waveguide.

Therefore, assuming a loss model, the distance can be based on a measurement of the power. Unfortunately the Bluetooth signal power is not accessible from applications running on most phones. Therefore, we use an indirect measurement of the signal power: the Bluetooth discovery time. As the signal strength decreases the probability of bit errors and thus the need for retransmission increases. This in turn increases the time for successful handshaking. In our application we attempt to discover the other Bluetooth device several times and measure the time taken for each attempt. If the maximum amongst several trials is below a certain threshold we accept the device as being within range.

We have performed extensive measurement of the average discovery time. Based on our measurements we found that when the phones are in very close proximity (less than 2m) this time almost never exceeds 0.06 seconds. Therefore, we use this as our threshold.

Another design parameter is the number of times to attempt discovery. We have used numbers between 3 and 10, and while using a larger number like 10 decreases the number of false acceptances, the difference is not very much. This is because once a device is successfully discovered subsequent discoveries usually take less time. This is probably due to some caching of the discovered unit's parameters by the phone's Bluetooth module.

The discovery time and therefore the false acceptance or rejection depends on the orientation on the antennas. When the two devices are pointed at each other the attenuation is much lower than when the devices are at 90 degrees. For example in one test we found that the acceptance rate for a 10m distance was 0 to 2% when the transmitting phone was pointing to a direction different from the line between the transmitter and receiver. At that same distance the acceptance rate jumped to 85% when the transmitter was pointed along the line. This is very close to the maximum acceptance rate of 90% when the phones are in close proximity.

Different obstacles like walls, doors and furniture can have an effect as well. However this depends on the component materials of the obstacles (metals shielding and wood and plastic having little effect). For example one wall had a 50-70% acceptance rate at a distance of 3m, while another had an acceptance rate of 3-10% for a similar distance.

In the case of real users, this means that depending on the orientation of their phone they may be very likely or unlikely to receive unwanted prompts. Two factors make this less of an issue. Firstly, in general it is rare for a user to stay in the same stance for a long period of time. Secondly, if the user has her phone in her pocket she will most probably not notice the prompt as it is not very audible and disappears after 10 seconds (but reappears again once a minute as long as the other user is in range).

The need for this estimation can be reduced by reducing the transmit power of the Bluetooth devices. This should reduce the range at which they can operate and thus limit the number of false positives. However since the range depends not only on the transmit power but also on the gain of the receiving device's antenna, therefore the need for distance estimation is reduced but not completely eliminated.

Finally, since distance information could useful for many other applications it might make sense to have the Bluetooth devices measure the distance themselves (based more directly on signal strength).

8

Threat Scenarios

Let's consider the case where an intruder Trudy wants to authenticate herself as legitimate user Alice. Let us consider the case with the vouching tokens and a simple PIN (no biometrics). There are many possible combinations of scenarios, depending on the intruder's possession of the phone, the pin, and the tokens.

8.1 Intruder does not have phone.

In this case as in all cases the intruder needs to obtain enough tokens and also obtain the PIN. Even if Trudy were to obtain the PIN, without access to Alice's phone Trudy can't get tokens directly from Alice's friends. She can't generate false tokens without the friends' private keys. But she can still snoop the SMS and Bluetooth traffic and grab the tokens as they are transmitted. However without the Alice's private key she can't decrypt the tokens to make use of them. Therefore the security of the tokens depend on the security of Alice's private key.

8.2 Intruder has phone

Now the problem is that without a TPM (Trusted Platform Module), Alice's private key is stored as a file or other accessible structure on her phone. Anyone with access to Alice's phone can copy the key, or send it to themselves over the network. Thus if Trudy can gain physical access to Alice's phone all bets are off. We can encrypt the key with a PIN, but that leads to a new problem. Either the user will constantly have to enter the PIN, since the private key is needed whenever a token is issued, or the user will enter the PIN once and then the private key will be stored in memory unencrypted, where Trudy can get to it if she has access to the phone.

Once Trudy has the phone there are three possibilities. If Trudy is lucky the phone already contains enough tokens, in that case she can try to authenticate using those tokens. If she also has the PIN it is game over for Alice. If there are not enough tokens on the phone, Trudy has two options: either return the phone to Alice and snoop tokens off the network, or keep the phone and try to obtain enough tokens directly.

8.2.1 Return phone and snoop

This approach is rather straightforward. The only problem is that Trudy has to return the phone before Alice notices it has disappeared and reports it as stolen. This is a problem if Trudy is a stranger, but if Trudy and Alice are friends then Trudy would simply have to borrow Alice's phone. Once the phone is returned Trudy waits, grabs enough tokens, and then all she needs is the PIN.

If the phone is equipped with a TPM then this approach is impossible, as there should be no way to extract the private key from the TPM. Thus Trudy will be forced to use the phone and get the tokens directly.

8.2.2 Keep phone and get tokens directly

There are several ways Trudy can go about trying to obtain tokens directly. For voice calls she can impersonate Alice to her friends, but this may backfire as the friends might find out and contact Alice or report suspicious activity on Alice's account. If Trudy and Alice are friends then she can call mutual friends and say that she is borrowing Alice's phone. Another tactic is for Trudy to call Alice's friends, say that she has found the phone and wants to return it to its owner. She may be able to drag out the conversation long enough for a token to be issued.

For Bluetooth sightings, she can trail Alice and receive tokens whenever Alice crosses a friend. Whenever they cross one of Alice's friends, Alice's phone will ask Trudy if she wants to send a token. Trudy will do so. The friend will then assume he is receiving a token from Alice and will send back a token in return. However this is dangerous, because Trudy has to be within line of sight of Alice for this to work, so only a particularly daring intruder would pull this off. However if Trudy and Alice are friends this may be a bit easier to do.

While a TPM will resolve the issue of the attacker's stealing the key and returning the phone to snoop traffic, this won't address the above problem of impersonation. An effective way to deal with this problem is to assign a weight to each friend, depending on the number of interactions they have with the user. This will favor the ones closer to the user and who will be most likely to report suspicious activity.

8.3 Stealing multiple phones

Of course we hope that the intruder is unable to obtain enough tokens either indirectly, because the phone has a TPM, or directly, because the intruder is unable to impersonate the user or otherwise fool her friends. However there is still a way for Trudy to get the required tokens: steal multiple phones, from the user's friends. This way Trudy can use these phones to generate tokens, by going through the call or Bluetooth token generation process.

Of course to succeed this requires the theft of N phones, where N is the number of tokens from different friends that are required to authenticate. The greater N the more secure the system is.

8.4 Once the intruder has enough tokens

Even if Trudy can obtain enough tokens, she can't authenticate without the PIN. Trudy can try randomly guessing the PIN. With a 4 digit PIN and 3 attempts, that gives Trudy a 3 in 10000 (0.03%) chance of success, and a 99.97% chance of failure, which will invalidate the tokens she obtained. Furthermore the system could be made to detect repeated failed attempts (say 100 failed attempts in a row), and then require a longer PIN or a manual reset.

On the other hand if Trudy can obtain the PIN, by for example observing Alice enter it while logging in, then she has all she needs to login. Thus if Trudy has only the tokens, the security of the system depends on the security provided by the PIN. While if Trudy has only the PIN, the security of the system depends on the security provided by the tokens.

9

Conclusion

In this Thesis we have developed a protocol for mobile social authentication based on phone conversations and Bluetooth sightings. We have substantiated our design based on simulation of our scheme using data from the Reality Mining dataset. We then implemented the scheme on actual phones using Python. In the process of doing so we developped a method of estimating the distance of a Bluetooth device indirectly through the measurement of the connection setup time. We tested the system for battery life and Bluetooth distance estimation accuracy. Finally we considered threats against the user available to an attacker.

With a standard security token, the intruder needs the token and the PIN to masquerade as the user. In our protocol the theft of a single phone would not necessarily result in a security breach, even if the PIN is known to the intruder. To generate enough tokens the intruder needs to have n phones. Where n is the number of tokens required for authentication. Having a large n, maintains high security. However it makes authentication more difficult.

This is a major improvement over the current state of public key authentication on mobile phones since nearly all phones lack a TPM. Without a TPM, access to a user's phone is all that is necessary to obtain their private key. If the passphrase protecting the private key is short and/or simple, which is almost a requirement given the limited input capabilities of mobile phones, the attacker can easily bruteforce the passphrase and compromise the system. In our system the user would need access to n of the user's friends' phones as well.

9.1 Directions for Future Research

In this Thesis we used public key encryption to securely encrypt the tokens. However a hybrid approach which uses symmetric key cryptography and includes the a public key encrypted version of the symmetric key along with the message would be preferable. Since this approach would provide all the advantages of public key encryption, but with the speed of symmetric encryption. This is the method employed in OpenPGP.

The only reason we didn't implement this feature is that there was that there was no suitable python implementation of a symmetric encryption algorithm that was faster than the RSA implementation. Besides writing such an encryption algortihm in Python, the system itself could be rewritten in Java ME. This should allow the program to run on more phones, as Java ME is currently supported on more mobile devices.

In chapter 8 we discussed the idea of using weights to the users' friends based on the number of interactions that they have with them. A simple scenario would be a case where at least one token from a "close" friend is required for authentication. Another would be the case where a token from a "close" friend is weighted the same as multiple tokens from more distant acquaintances. Determining the optimal weight assignment would be a worthwhile venture, since this threat to the system can't be solved through purely technical means.

We designed the authentication system, including client and server code for the general case of authenticating a user for an online service. Thus in its current version it could be used for logging into an online banking website, a social networking site, webmail, etc.

We can extend the work by removing the central server and making the protocol decentralized and peer to peer. In this case the main issue will be key distribution. Probably the easiest and most secure way of obtaining the necessary public keys is through Bluetooth exchange when the user is with the friend whose key they wish to obtain.

Once that is done the system could be adapted to a wider range of applications, most importantly mobile and social networking ones. For example a user might wish to share pictures they have taken on their phone with friends. Using social authentication they can issue tokens to friends and then require the tokens for access to the pictures. Or alternatively if the user wants to also grant access to friends of friends, she can verify tokens issued by their friends to these friends of friends. Similar social access controls can be built into social networking sites to improve security.

A

Source Code

A.1 sendtoken.py

```
import e32
import globalui
import logs
import messaging
import os
import pickle
import re
import rsa
import socket
import time

#Load data structures from files
mynumberfile = open("e:/mynumber")
mynumber = pickle.load(mynumberfile)
pubfile = open("e:/pubkey")
privfile = open("e:/privkey")
pub = pickle.load(pubfile)
priv = pickle.load(privfile)
```

```
keyringfile = open("e:/keyring")
keyring = pickle.load(keyringfile)
friendsfile = open("e:/friends")
friends = pickle.load(friendsfile)
btfile = open("e:/bt")
bt = pickle.load(btfile)
#Check for daylight savings time
if(time.daylight == 1):
  t = time.time() + time.altzone
else:
  t = time.time() + time.timezone
btwait = {}
for i in bt.keys():
  btwait[i] = 0;
def send_calls():
  global t
  #get new calls (new since last time we processed them)
  if(time.daylight == 1):
    cin = logs.log_data_by_time('call', t, time.time()
            + time.altzone, mode='in')
    cout = logs.log_data_by_time('call', t, time.time()
            + time.altzone, mode='out')
  else:
    cin = logs.log_data_by_time('call', t, time.time()
            + time.timezone, mode='in')
    cout = logs.log_data_by_time('call', t, time.time()
            + time.timezone, mode='out')
  for i in range(0,len(cin)):
    #send token if there is an incoming call from a friend
    for number in keyring.keys():
      s = number[2:]
      if(re.search(s, cin[i]['number'])):
        sign = rsa.sign(str(time.time()) + "␣"
            + str(number), priv)
        cipher = rsa.encrypt(sign, keyring[number])
        message = "token" + cipher + "token"
        messaging.sms_send(number, message)
        t = cin[i]['time'] + 1
  for i in range(0,len(cout)):
    for number in keyring.keys():
    #send token if there is an outgoing call to a friend
      s = number[2:]
      if(re.search(s, cout[i]['number'])):
        sign = rsa.sign(str(time.time()) + "␣"
            + str(number), priv)
        cipher = rsa.encrypt(sign, keyring[number])
        message = "token" + cipher + "token"
        messaging.sms_send(number, message)
        t = cout[i]['time'] + 1
def send_bluetooth():
  for i in bt.keys():
    #skip this iteration if we didn't see the peer recently
    if(btwait[i] != 0):
      btwait[i] = btwait[i] - 1;
    else:
      try:
        # measure handshake time repeatedly
        t1 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t2 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t3 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t4 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        t5 = time.clock()
        # if all times low assume peer is near
        # and send token
        if( ((t3-t2) < 0.06) and ((t4-t3) < 0.06)
          and ((t5-t4) < 0.06) ):
          globalui.global_note(u"Device␣in␣range.")
          q = globalui.global_query(u"Send␣token␣to␣"
              + friends[i] + "?", 15)
          if(q==1):
            print "Generating␣token."
            sign = rsa.sign(str(time.time()) + "␣" + str(i), priv)
            cipher = rsa.encrypt(sign, keyring[i])
            message = mynumber + "_" + str(time.time()) + "token"
              + cipher + "token"
            messagefile = open("e:\\message.txt", "w")
            messagefile.write(message)
            messagefile.close()
            print "Sending␣token."
            socket.bt_obex_send_file(b[0],b[1].values()[0],
              u"e:\\message.txt")
            del b
            print "Token␣sent."
            os.remove("e:\\message.txt")
            btwait[i] = 10
          elif(q==0):
            btwait[i] = 6
      except:
        print "Device␣out␣of␣range."
        btwait[i] = 0
while(1):
  send_calls()
  send_bluetooth()
  e32.ao_sleep(10)
```

A.2 authenticate.py

```
import globalui
import inbox
import os
import pickle
import re
import rsa
import socket
import time

#Load data structures from files
mynumberfile = open("e:/mynumber")
mynumber = pickle.load(mynumberfile)
privfile = open("e:/privkey")
```

```
priv = pickle.load(privfile)
keyringfile = open("e:/keyring")
keyring = pickle.load(keyringfile)
serverpubfile = open("e:/server_pubkey")
serverpub = pickle.load(serverpubfile)
friendsfile = open("e:/friends")
friends = pickle.load(friendsfile)
tokens=[]
authenticators=[]
i = inbox.Inbox()
m = i.sms_messages()
tb = "^token"
te = "token$"
total = 0;
#count the number of tokens in the sms inbox
for j in m:
  t = i.content(j)
  if(re.search(tb, t) and re.search(te, t)):
    total = total+1
count = 0
for j in m:
  t = i.content(j)
  #decrypt each token and then reencrypt for the server
  if(re.search(tb, t) and re.search(te, t)):
    count = count + 1
    t = re.sub(tb, "", t)
    t = re.sub(te, "", t)
    globalui.global_note(u"" + str(count) + "/" + str(total)
          + "␣Decrypting")
    d = rsa.decrypt(t, priv)
    a = i.address(j)
    for k in friends:
      if(re.search(a, friends[k])):
        n = k
    t2 = str(time.time()) + "␣" + str(n) + "␣" + str(d)
    globalui.global_note(u"" + str(count) + "/" + str(total)
         + "␣Encrypting")
    auth = rsa.encrypt(t2, serverpub)
    authenticators.append(auth)
HOST = 'server.crasseux.com'
PORT = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send("START")
#send encrypted tokens to server
for i in authenticators:
  l = len(i)
  sent = 0
  s.send("AUTHBEGIN")
  while(l>0):
    b = s.send(i[sent:])
    sent = sent+b
    l = l-b
  s.send("AUTHFINISH")
s.send("END")
status = s.recv(3)
#receive authentication or failure message
if(status=="YES"):
  globalui.global_note(u"Authenticated")
  c = ''
  while 1:
    data = s.recv(1024)
    c = c + data
    if(re.search("END$", c)):
      break
  c = re.sub("^START", "", c)
  c = re.sub("END$", "", c)
  print "c␣=␣" + c
  password = rsa.decrypt(c, priv)
  print "password␣=␣" + password
  globalui.global_note(u"Password:␣" + password)
else:
  globalui.global_note(u"Access␣Denied")
```

A.3 auth.py

```
#!/usr/bin/python
import pickle
import random
import re
import rsa
import socket

server_privkey_file = open("./keys/server_privkey")
server_privkey = pickle.load(server_privkey_file)
keyring_file = open("./keys/keyring")
keyring = pickle.load(keyring_file)
words_file = open("/usr/share/dict/words")
```

```
words = words_file.read()
list = words.rsplit("\n")

HOST = ''
PORT = 9000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
while 1:
  #accept connection
  conn, addr = s.accept()
  print 'Connected␣by', addr
  c = ''
  #receive all tokens
  while 1:
    data = conn.recv(1024)
    c = c + data
    if(re.search("END$", c)):
      break
  authenticators=[]
  c = re.sub("^START", "", c)
  c = re.sub("END$", "", c)
  finished=0
  while(finished==0):
    c = re.sub("^AUTHBEGIN", "", c)
    m = re.search("AUTHFINISH", c)
    authenticators.append(c[:m.start()])
    c = c[m.end():]
    if(not re.search("^AUTHBEGIN", c)):
      break
  for i in authenticators:
    #decrypt tokens, then verify signatures
    #if successful send authentication else send failure message
    m = rsa.decrypt(i, server_privkey)
    match = re.search("␣", m)
    t1 = m[:match.start()]
    m = m[match.end():]
    match = re.search("␣", m)
    n1 = m[:match.start()]
    m = m[match.end():]
    v = rsa.verify(m, keyring[n1])
    match = re.search("␣", v)
    t2 = v[:match.start()]
    n2 = v[match.end():]
  conn.send("YES")
  password = list[random.randint(1,len(list))]
  c = rsa.encrypt(password, keyring[n2])
  c = "START" + c + "END"
  conn.send(c)
  conn.close()
```

A.4 hashchain.py

```
import globalui
import md5
import pickle
import re
import rsa
import socket
```

```
#Load data structures from files
mynumberfile = open("e:/mynumber")
mynumber = pickle.load(mynumberfile)
privfile = open("e:/privkey")
priv = pickle.load(privfile)
serverpubfile = open("e:/server_pubkey")
serverpub = pickle.load(serverpubfile)

#connect to server and receive hash secrets
HOST = 'server.crasseux.com'
PORT = 9001
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send(mynumber)
c = ''
while 1:
  data = s.recv(1024)
  c = c + data
  if(re.search("END$", c)):
    break

c = re.sub("^START", "", c)
c = re.sub("^BEGIN_EXPIRY", "", c)
m = re.search("FINISH_EXPIRY", c)
se_ciph = c[:m.start()]
c = c[m.end():]

c = re.sub("^BEGIN_SECRETS", "", c)
```

```
m = re.search("FINISH_SECRETS", c)
s_ciph = c[:m.start()]
c = c[m.end():]

c = re.sub("^BEGIN_HASH_COUNTER", "", c)
m = re.search("FINISH_HASH_COUNTER", c)
hc_ciph = c[:m.start()]

globalui.global_note(u"1/6␣Decrypting␣Expiry␣Date")
se_sign = rsa.decrypt(se_ciph, priv)
globalui.global_note(u"2/6␣Verifying␣Expiry␣Date")
se_string = rsa.verify(se_sign, serverpub)
secrets_expiry = pickle.loads(se_string)

globalui.global_note(u"3/6␣Decrypting␣Secrets")
s_sign = rsa.decrypt(s_ciph, priv)
globalui.global_note(u"4/6␣Verifying␣Secrets")
s_string = rsa.verify(s_sign, serverpub)
secrets = pickle.loads(s_string)

globalui.global_note(u"5/6␣Decrypting␣Counters")
hc_sign = rsa.decrypt(hc_ciph, priv)
globalui.global_note(u"6/6␣Verifying␣Counters")
hc_string = rsa.verify(hc_sign, serverpub)
hash_counter = pickle.loads(hc_string)

se_file = open("e:/secrets_expiry", "w")
s_file = open("e:/secrets", "w")
hc_file = open("e:/hash_counter", "w")
```

```
pickle.dump(secrets_expiry, se_file)
pickle.dump(secrets, s_file)
pickle.dump(hash_counter, hc_file)

h_file = open("e:/hashes", "w")
hashes = {}
for i in secrets:
  h=[]
  cur = md5.new()
  cur.update(secrets[i])
  h.append(cur.hexdigest())
  for j in range(1,101):
    cur = md5.new()
    cur.update(h[j-1])
    h.append(cur.hexdigest())
  hashes[i] = h
pickle.dump(hashes, h_file)
```

A.5 hashtoken.py

```
import e32
import globalui
import logs
import messaging
import os
import pickle
import re
import rsa
import socket
import time
```

```
#Load data structures from files
mynumberfile = open("e:/mynumber")
mynumber = pickle.load(mynumberfile)
pubfile = open("e:/pubkey")
privfile = open("e:/privkey")
pub = pickle.load(pubfile)
priv = pickle.load(privfile)
keyringfile = open("e:/keyring")
keyring = pickle.load(keyringfile)
friendsfile = open("e:/friends")
friends = pickle.load(friendsfile)
btfile = open("e:/bt")
bt = pickle.load(btfile)
hc_file = open("e:/hash_counter")
hash_counter = pickle.load(hc_file)
hc_file.close()
se_file = open("e:/secrets_expiry")
secrets_expiry = pickle.load(se_file)
s_file = open("e:/secrets")
secrets = pickle.load(s_file)
h_file = open("e:/hashes")
hashes = pickle.load(h_file)

#Check for daylight savings time
if(time.daylight == 1):
  t = time.time() + time.altzone
else:
  t = time.time() + time.timezone
```

```
btwait = {}
for i in bt.keys():
  btwait[i] = 0;
def send_calls():
  global t
  #get new calls (new since last time we processed them)
  if(time.daylight == 1):
    cin = logs.log_data_by_time('call', t, time.time()
            + time.altzone, mode='in')
    cout = logs.log_data_by_time('call', t, time.time()
            + time.altzone, mode='out')
  else:
    cin = logs.log_data_by_time('call', t, time.time()
            + time.timezone, mode='in')
    cout = logs.log_data_by_time('call', t, time.time()
            + time.timezone, mode='out')
  for i in range(0,len(cin)):
    for number in keyring.keys():
    #send hash token if there is an incoming call from a friend
      s = number[2:]
      if(re.search(s, cin[i]['number'])):
        message = "hashtoken"
          + hashes[number][hash_counter[number]]
          + "␣" + hash_counter[number]
          + "hashtoken"
        hash_counter[number]=hash_counter[number]-1
        hc_file = open("e:/hash_counter", "w")
        pickle.dump(hash_counter, hc_file)
        hc_file.close()
        messaging.sms_send(number, message)
        t = cin[i]['time'] + 1
  for i in range(0,len(cout)):
    for number in keyring.keys():
    #send hash token if there is an outgoing call to a friend
      s = number[2:]
      if(re.search(s, cout[i]['number'])):
        message = "hashtoken"
          + hashes[number][hash_counter[number]]
          + "␣" + hash_counter[number]
          + "hashtoken"
        hash_counter[number]=hash_counter[number]-1
        hc_file = open("e:/hash_counter", "w")
        pickle.dump(hash_counter, hc_file)
        hc_file.close()
        messaging.sms_send(number, message)
        t = cout[i]['time'] + 1
def send_bluetooth():
  for i in bt.keys():
    #skip this iteration if we didn't see the peer recently
    if(btwait[i] != 0):
      btwait[i] = btwait[i] - 1;
    else:
      try:
        # measure handshake time repeatedly
        t1 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t2 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t3 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        del b
        t4 = time.clock()
        b = socket.bt_obex_discover(bt[i])
        t5 = time.clock()
        # if all times low assume peer is near
        # and send hash token
        if( ((t3-t2) < 0.06) and ((t4-t3) < 0.06)
          and ((t5-t4) < 0.06) ):
          globalui.global_note(u"Device␣in␣range.")
          q = globalui.global_query(u"Send␣token␣to␣"
            + friends[i] + "?", 15)
          if(q==1):
            print "Generating␣token."
            print "Here"
            number = str(i)
            print number
            message = "hashtoken"
            + hashes[number][hash_counter[number]]
            + "␣" + str(hash_counter[number])
            + "hashtoken"
            print "Token␣Generated"
            hash_counter[number]=hash_counter[number]-1
            hc_file = open("e:/hash_counter", "w")
            print "Counter␣Updated"
            pickle.dump(hash_counter, hc_file)
            hc_file.close()
            print "Writing␣temp␣file"
            messagefile = open("e:\\message.txt", "w")
            messagefile.write(message)
            messagefile.close()
            print "Sending␣token."
            socket.bt_obex_send_file(b[0],b[1].values()[0],
              u"e:\\message.txt")
            del b
            print "Token␣sent."
            os.remove("e:\\message.txt")
            btwait[i] = 10
          elif(q==0):
            btwait[i] = 6
      except:
        print "Device␣out␣of␣range."
        btwait[i] = 0
while(1):
  send_calls()
  send_bluetooth()
  e32.ao_sleep(10)
```

A.6 hashauthenticate.py

```
import globalui
import inbox
import os
import pickle
import re
import rsa
import socket
import time

#Load data structures from files
mynumberfile = open("e:/mynumber")
mynumber = pickle.load(mynumberfile)
privfile = open("e:/privkey")
priv = pickle.load(privfile)
keyringfile = open("e:/keyring")
keyring = pickle.load(keyringfile)
serverpubfile = open("e:/server_pubkey")
serverpub = pickle.load(serverpubfile)
friendsfile = open("e:/friends")
friends = pickle.load(friendsfile)
tokens=[]
authenticators=[]
i = inbox.Inbox()
m = i.sms_messages()
tb = "^hashtoken"
te = "hashtoken$"
total = 0;
#count the number of hash tokens in the sms inbox
for j in m:
  t = i.content(j)
  if(re.search(tb, t) and re.search(te, t)):
    total = total+1
count = 0
for j in m:
  t = i.content(j)
  #encrypt hash tokens for the server
  if(re.search(tb, t) and re.search(te, t)):
    count = count + 1
    t = re.sub(tb, "", t)
    t = re.sub(te, "", t)
    a = i.address(j)
    for k in friends:
      if(re.search(a, friends[k])):
        n = k
    t2 = str(n) + "␣" + t
    globalui.global_note(u"" + str(count) + "/" + str(total)
          + "␣Encrypting")
    auth = rsa.encrypt(t2, serverpub)
    authenticators.append(auth)
HOST = 'server.crasseux.com'
PORT = 9002
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
#send encrypted tokens to server
s.send("START")
for i in authenticators:
  l = len(i)
  sent = 0
  s.send("AUTHBEGIN")
  while(l>0):
    b = s.send(i[sent:])
    sent = sent+b
    l = l-b
  s.send("AUTHFINISH")
```

```
s.send("END")
status = s.recv(3)
#receive authentication or failure message
if(status=="YES"):
  globalui.global_note(u"Authenticated")
  c = ''
  while 1:
    data = s.recv(1024)
    c = c + data
    if(re.search("END$", c)):
      break
  c = re.sub("^START", "", c)
  c = re.sub("END$", "", c)
  print "c␣=␣" + c
  password = rsa.decrypt(c, priv)
  print "password␣=␣" + password
  globalui.global_note(u"Password:␣" + password)
else:
  globalui.global_note(u"Access␣Denied")
```

A.7 hashsecrets.py

```
#!/usr/bin/python
import os
import pickle
import rsa
import socket
import time

#generate hash secrets
```

```
def setup_hashchain(number):
  def new_secrets():
    secrets_expiry = time.time() + 2*24*3600
    se_file=open("secrets_expiry", "w")
    pickle.dump(secrets_expiry, se_file)
    se_file.close()

    secrets = {}
    for i in friends:
      secrets[i] = rsa.urandom(32)
    s_file = open("secrets", "w")
    pickle.dump(secrets, s_file)
    s_file.close()

    hash_counter = {}
    for i in friends:
      hash_counter[i] = 100
    hc_file = open("hash_counter", "w")
    pickle.dump(hash_counter, hc_file)
    hc_file.close()
  os.chdir(number)
  ff = open("friends")
  friends = pickle.load(ff)
  if(os.access("secrets_expiry", os.F_OK)):
    se_file=open("secrets_expiry")
    secrets_expiry = pickle.load(se_file)
    s_file=open("secrets")
    secrets = pickle.load(s_file)
    hc_file=open("hash_counter")
    hash_counter = pickle.load(hc_file)
    if(secrets_expiry < time.time()):
      print "secrets␣expired"
  else:
    new_secrets()
  os.chdir("..")
  print "Done"
setup_hashchain("+15143866409")
setup_hashchain("+15149650900")

server_privkey_file = open("./keys/server_privkey")
server_privkey = pickle.load(server_privkey_file)
keyring_file = open("./keys/keyring")
keyring = pickle.load(keyring_file)

#listen for connections from clients
HOST = ''
PORT = 9001
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
#send hash secrets to client
while 1:
  conn, addr = s.accept()
  print 'Connected␣by', addr
  c = ''
  while 1:
    data = conn.recv(1024)
    c = c + data
    if(len(c)==12):
      break
  number = c
  print number
  os.chdir(number)
  ff = open("friends")
  friends = pickle.load(ff)
  se_file=open("secrets_expiry")
  secrets_expiry = pickle.load(se_file)
  s_file=open("secrets")
  secrets = pickle.load(s_file)
  hc_file=open("hash_counter")
  hash_counter = pickle.load(hc_file)
  se_string = pickle.dumps(secrets_expiry)
  s_string = pickle.dumps(secrets)
  hc_string = pickle.dumps(hash_counter)

  se_sign = rsa.sign(se_string, server_privkey)
  se_ciph = rsa.encrypt(se_sign, keyring[number])
  s_sign = rsa.sign(s_string, server_privkey)
  s_ciph = rsa.encrypt(s_sign, keyring[number])
  hc_sign = rsa.sign(hc_string, server_privkey)
  hc_ciph = rsa.encrypt(hc_sign, keyring[number])

  conn.send("START")

  conn.send("BEGIN_EXPIRY")
  l = len(se_ciph)
  sent = 0
  while(l>0):
    b = conn.send(se_ciph[sent:])
    sent = sent+b
    l = l-b
  conn.send("FINISH_EXPIRY")

  conn.send("BEGIN_SECRETS")
  l = len(s_ciph)
  sent = 0
  while(l>0):
    b = conn.send(s_ciph[sent:])
    sent = sent+b
    l = l-b
  conn.send("FINISH_SECRETS")

  conn.send("BEGIN_HASH_COUNTER")
  l = len(hc_ciph)
  sent = 0
  while(l>0):
    b = conn.send(hc_ciph[sent:])
    sent = sent+b
    l = l-b
  conn.send("FINISH_HASH_COUNTER")
  conn.send("END")

  os.chdir("..")
  conn.close()
```

A.8 hashauth.py

```
#!/usr/bin/python
import pickle
import random
import re
import rsa
import socket

#Load data structures from files
server_privkey_file = open("./keys/server_privkey")
server_privkey = pickle.load(server_privkey_file)
keyring_file = open("./keys/keyring")
keyring = pickle.load(keyring_file)
words_file = open("/usr/share/dict/words")
words = words_file.read()
list = words.rsplit("\n")

#listen for connections from clients
HOST = ''
PORT = 9002
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
#receive encryoted hash tokens from client
while 1:
  conn, addr = s.accept()
  print 'Connected␣by', addr
  c = ''
  while 1:
    data = conn.recv(1024)
    c = c + data
    if(re.search("END$", c)):
      break
  authenticators=[]
  c = re.sub("^START", "", c)
  c = re.sub("END$", "", c)
  finished=0
  while(finished==0):
    if(not re.search("^AUTHBEGIN", c)):
      break
    c = re.sub("^AUTHBEGIN", "", c)
    m = re.search("AUTHFINISH", c)
    authenticators.append(c[:m.start()])
    c = c[m.end():]
  #decrypt hash tokens and verify signatures
  for i in authenticators:
    m = rsa.decrypt(i, server_privkey)
    print m
    match = re.search("␣", m)
    t1 = m[:match.start()]
    m = m[match.end():]
    match = re.search("␣", m)
    n1 = m[:match.start()]
    m = m[match.end():]
    v = rsa.verify(m, keyring[n1])
    match = re.search("␣", v)
    t2 = v[:match.start()]
    n2 = v[match.end():]
  conn.send("YES")
```

```
password = list[random.randint(1,len(list))]
c = rsa.encrypt(password, keyring[n2])
c = "START" + c + "END"
conn.send(c)
conn.close()
```

Bibliography

[1] J. Brainard, A Juels, R. Rivest, M. Szydlo and M. Yung, “Fourth-Factor Authentication: Somebody you know”, CCS’06: Proceedings of the 13th ACM conference on Computer and communications security , pp. 168-178, Alexandria,Virginia, USA, October 30-November 3 2006,

[2] N. Eagle and A. Pentland, “Eigenbehaviors: Identifying Structure in Routine”, Behavioral Ecology and Sociobiology, Volume 63, Number 7, pp. 1057-1066, May 2009.

[3] C.E. Shannon, “A Mathematical Theory of Communication”, Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, July, October, 1948.

[4] “Entropy (information theory).” Wikipedia, The Free Encyclopedia. 11 Sep 2009, 15:13 UTC. 7 Oct 2009 `http://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)&oldid=313200698`

[5] C.M. Thomas. and J.A. Thomas, “Elements of Information Theory 2nd Edition”, Wiley Series in Telecommunications and Signal Processing, Wiley-Interscience, Chapter 14, Hoboken, New Jersey, July 2006.

[6] M. Burnett, “Password Trivia: Character Sets” `http://xato.com/passwords/password-trivia-character-sets`

[7] M. Burnett, “Perfect passwords: selection, protection, authentication”, Syngress, Rockland, Massachussetts, 2006.

[8] "Electronic Authentication Guideline" (PDF). NIST. `http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf` Retrieved October 7 2009.

[9] "Biometrics.", Wikipedia, The Free Encyclopedia. 8 Oct 2009, 18:40 UTC. 11 Oct 2009 `http://en.wikipedia.org/w/index.php?title=Biometrics&oldid=318706649`

[10] L. O'Gorman, "Comparing Passwords, Tokens, and Biometrics for User Authentication", The Proceedings of the IEEE, Vol. 91, No. 12, pp. 2019-2020, December 2003.

[11] "MobileKey (Mobile Authentication Server)", MobileKey `http://www.visualtron.com/products_mobilekey.htm`.

[12] PhoneFactor "Tokenless Two-Factor Authentication", PhoneFactor `http://www.phonefactor.com/how-it-works/overview/`.

[13] M. Hassinen, "SafeSMS - End-to-End encryption for SMS messages." Proceedings of the 8th International Conference on Telecommunications ConTEL 2005, pp. 359-365, Zagreb, Croatia, June 15-17 2005.

[14] R. Ghosh and M. Dekhil, "I, Me and My Phone: Identity and Personalization using Mobile Devices", HP Technical Reports, HPL-2007-184, 2007.

[15] M. Mont, B. Balacheff, J. Rouault and D. Drozdzewski, "On Identity-Aware Devices: Putting Users in Control across Federated Services", HP Technical Reports, HPL-2008-26, 2008.

[16] M. Mont and B. Balacheff, "On Device-based Identity Management in Enterprises", HP Technical Reports, HPL-2007-53, 2007.

[17] F. Aloul, S. Zahidi and W. El-Hajj, "Two Factor Authentication Using Mobile Phones", IEEE International Conference on Computer Systems and Applications (AICCSA), pp. 641-644, Rabat, Morocco, May 2009.

[18] "SAASM", Wikipedia, The Free Encyclopedia. 15 Sep 2009, 15:08 UTC. 14 Oct 2009 `http://en.wikipedia.org/w/index.php?title=SAASM&oldid=314120387`

[19] A. Durresi et al., "Secure Spatial Authentication using Cell Phones", Second International Conference on Availability, Reliability and Security (ARES'07), pp. 543-549, Vienna, Austria, April 10-13 2007.

[20] A. Das, O.K. Manyam, M. Tapaswi and V. Taranalli, "Multilingual Spoken-password Based User Authentication In Emerging Economies Using Cellular Phone Networks", SLT 2008: IEEE Spoken Language Technology Workshop 2008, pp. 5-8, Goa, India, December 15-19 2008.

[21] A. Hadid, J. Y Heikkild, 0. Silven and M. Pietikdinen, "Face And Eye Detection For Person Authentication In Mobile Phones", ICDSC '07: First ACM/IEEE International Conference on Distributed Smart Cameras 2007, pp. 101-108, Vienna, Austria, September 25-28 2007.

[22] D.H. Cho, K.R. Park and D.W. Rhee, " SoftwareReal-time iris localization for iris recognition in cellular phone", SNPD/SAWN 2005. Sixth International Conference on Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2005, and First ACIS International Workshop on Self-Assembling Wireless Networks, pp. 254-259, Towson University, Maryland, USA, May 23-25 2005.

[23] P. Campisi E. Maiorana M. Lo Bosco A. Neri, "User authentication using keystroke dynamics for cellular phones", IET Signal Processing, Volume 3, Issue 4, pp. 333-341, July 2009.

[24] F. Okumura, A. Kubota, Y. Hatori, K. Matsuo, M. Hashimoto, and A. Koike, "A Study on Biometric Authentication based on Arm Sweep Action with Acceleration Sensor", ISPACS '06: International Symposium on Intelligent Signal Processing and Communications 2006. pp. 219-222, Tottori, Japan, December 12-15 2006.

[25] D. Gafurov, E. Snekkenes. and P. Bours, ”Spoof Attacks on Gait Authentication System”, IEEE Transactions on Information Forensics and Security, Volume 2, Issue 3, Part 2, pp. 491-502, September 2007.

[26] Y. Zheng, D. He, W. Yu and X. Tang, “Trusted Computing-Based Security Architecture For 4G Mobile Networks”, PDCAT 2005: Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies 2005, pp. 251-255, Dalian, China, December 5-8 2005.

[27] A. Ramachandran and N. Feamster, “Authenticated out-of-band communication over social links”, WOSN’08: Proceedings of the first workshp on Online social networks, pp.61-66, Seattle, Washington, August 18 2008.

[28] S. Yardi, N. Feamster and A. Bruckman, ”Photo-Based Authentication Using Social Networks”, WOSN’08: Proceedings of the first workshop on Online social networks, Seattle, Washington, August 18 2008.

[29] A. Frankel and M. Maheswaran, “Feasibility of a Socially Aware Authentication Scheme”, CCNC 2009: Consumer Communications and Networking Conference 2009, pp. 1-6, Las Vegas, Nevada, 2009.

[30] B. Soleymani and M. Maheswaran, Social Authentication Protocol for Mobile Phones, SIN09: International Symposium on Social Intelligence and Networking, pp. 1-7, Vancouver, British Columbia, September 10-12 2009.

[31] S. Ojala, J. Keinanen and J. Skytta, “Wearable authentication device for transparent login in nomadic applications environment”, SCS 2008: 2nd International Conference on Signals, Circuits and Systems 2008, pp. 1-6, Sfax University, Tunisia, November 7-9 2008.

[32] A. Itai.and H. Yasukawa, “Footstep classification using wavelet decomposition”, ISCIT 2007: International Symposium on Communications and Information Technologies 2007, pp. 551-556, Darling Harbour, Sydney, Australia, October 17-19 2007.