Bluetooth Proximity Authentication as 2nd Factor

Tobias Zuber

BACHELORARBEIT

Nr. S1310237035-B

eingereicht am
Fachhochschul-Bachelorstudiengang

Mobile Computing

in Hagenberg

im Dezember 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Sicherheit in mobilen Systemen

im

Wintersemester 2015/2016

Betreuer:

Dipl. Ing. Dr. techn. Erik Sonnleitner

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, December 5, 2016

Tobias Zuber

Contents

Abstract

The goal of this bachelor thesis is to write a daemon, which detects the proximity of a Bluetooth device in order to verify that a certain action (e.g. an attempted login) is valid and performed by an authenticated user.
The authentication of the user is verified in two steps:

1. The user needs to provide the login credentials ("Something you know")
2. The user needs to have an authentication device - in this case a Bluetooth device ("Something you have")

This authentication process is called Two-factor authentication (2FA).

With this bachelor thesis comes a background application (daemon) and three functional client implementations of a two-factor authentication with Bluetooth proximity sensing. One of the client implementations is an internet browser extension for Mozilla Firefox, which will add an additional security factor to the included password manager. The other implementation is a Linux PAM module, which adds a security component to the login scheme. The third client is a Windows Desktop client, which automatically locks the screen if the Bluetooth device isn't nearby.

The thesis will cover besides the implementation of mentioned clients and daemon, a coverage and explanation of current already existing solutions of Two-factor authentication. Furthermore, it will present the usual operating places and limitations of Two-factor authentication.

Note: The expression "second-factor authentication" is equal to "two factor authentication" and can be used interchangeably.

Chapter 1

Introduction

1.1 Motivation

The number of credentials a typical user nowadays has to deal with - including the web, smartphones, encrypted storage, et cetera - is constantly increasing. Additionally, to the vast number of credentials rises the significance of mentioned credentials in consideration of our daily use of important systems and services. While the common login authentication procedure with credentials only was acceptable, today it is not advisable anymore. Despite the rise of significance, was a change to an enhanced authentication system not observed.

The thesis aims to give an overview of the functionality and advantages of an improved authentication through "second-factor authentication". A demonstration is made using a daemon with an open interface for Bluetooth proximity detection and sample clients utilizing the interface to extend an already existing login scheme.

1.2 Goals

The goal of this thesis is to describe the theoretical aspects of second-factor authentication, as well as its advantages and the increase of security by its usage. The theoretical background will be revisited and applied in the practical part, which is an implementation of a daemon that offers Bluetooth proximity detection. The daemon provides an open interface for 3rd-party applications to extend an authentication scheme to transform it into a two-factor authentication. Besides the daemon implementation, there are three clients provided. One client should be a Mozilla Firefox extension, which uses the daemon proximity detection in combination with the credential storage master password. The second client is a Linux PAM module, which increases the authentication security of the operation system's login scheme. The third client is a Windows Desktop application that automatically locks the screen when the proximity detection failed.

The final result of the thesis should cover the aspects of Two-factor authentication, as well as provide a daemon and the three mentioned sample clients, which are using the interface of the daemon in order to establish a second-factor authentication.

1.3 Structure

The next chapter *Related Work* presents the most common authentication solutions as a second factor such as the verification with physical devices, hardware and software token as well as the proximity authentication. It discusses the general and specific approach of the different verification techniques and demonstrates their typical operating place.
The *Implementation* chapter introduces the technical details such as the general approach and implementation architecture, subsequently followed by an in-depth overview of the components by the daemon and each implemented client.
Following this, the *Results* chapter analyses the overall results, as well as for the daemon and each implementation specifically the security enhancements using Bluetooth proximity.
A *discussion* will be presented in chapter 5, reflecting different views and aspects of the thesis while comparing it with other novel or existing solutions.
The *Conclusion* chapter gives a summary and future outlook of Bluetooth Proximity as a second-factor authentication, followed by a *Manual* which explains the usage of the accompanying and created tools of this thesis.

Chapter 2

Related Work

This chapter presents an overview of the functionality of second-factor authentication (2FA), subsequently followed the next section with an analysis, alongside examples, about existing 2FA solutions and their usual application scope. The last section addresses the limitations and drawbacks of 2FA.

2.1 What is and how does Multi- or Two-factor (2FA) authentication work?

Commonly an authentication of a login on system or service is done through the verification of login credentials. Such login credentials usually consist of a username or -id and a password. When a login attempt occurs, the system first checks the existence of the inserted username and only proceeds, if the mentioned user exists. Then the validation of the password takes places: Is the entered password incorrect for this user the authentication fails and the login dismisses. However, if the username and password combination succeeds, the user verifies itself and is authenticated to use the system or service.

While this authentication method was acceptable in the recent years, it is not advisable to use it today anymore. This is due to the rise of the significance of our login credentials in consideration of our modern daily use of important systems and services, especially online but also with stationary systems. Actually it has been proven, that the usage of login credentials only as verification method is quite risky and insecure, since "hackers have the option of using many techniques to steal passwords such as shoulder surfing, snooping, sniffing, guessing, etc."[1].

In order to regain a good security level, an additional security component should be added to the verification process: an authentication device, which the user needs to possess while performing a login attempt. By adding this security factor, the authentication would transform into a Multi- (or here:

Two-) factor authentication.
Multi- (or: Two-) factor authentication can be explained by the different factors[37]:

1. *Something you know* - Login credentials (username and password), which the user needs to *know*.
2. *Something you have* - An Authentication device, which the user needs to *have*.
3. *Something you are* - Physical verification (using biometrical methods, e.g. fingerprint or iris scan) of the user - to *be (are)* him-/herself.

The authentication device is usually a physical accessory, which the user can easily carry and obtain when the login queries for its existence at the system. The presence of the device at the system does not necessarily need to be physical. With modern technologies a wireless connection, for example via Bluetooth, is also imaginable. However, USB drives, access cards or keys are also common accessories of choice.

To ensure the exclusiveness and security of the verification only one authentication device must exist per user credential. Furthermore, each device itself has some particular unique characteristics (e.g. individual hardware serial number), which makes it different to other similar devices. The combination of both, login credentials and individual authentication devices for each user, enhances the security of the verification.

With a recent development, a trend has been established where the physical authentication devices are getting replaced with smartphones. This resolves from the case that smartphones are widely used today on a daily basis. This exchange is not a bad transition since the second factor still endures as an additional physical accessory, hence it is an improvement in the usability of using Two-factor authentication.[8]

An attempted Login with a Two-factor authentication would perform the following:

1. The user credentials are being entered. First, the authentication system is going to verify the existence of the user. When the user has been found and the provided password can be validated, the system proceeds to the second step of the authentication.
2. The system checks and queries for an authentication device. Once the system detects an authentication device, it checks the coherence with the supplied user credentials.

If both steps succeed without failure, the user can be verified and is logged into the system.

To increase the security of a Two-factor authentication, an additional security component could be added, which would transform it into a “Multi-factor authentication”. This third factor can be described as *something you are*. Through this factor, security is one step more increased, since aside of the verification of the login credentials (first factor) and the possession of the authentication device (second factor), the physical identification of the user (*something you are*) is going to be validated. The physical verification is done through biometric methods. Such biometric authentication methods are iris scans, fingerprint scans, the facial proportions check, voice and speech recognition, as well as handwriting and keystroke dynamics.[36]

While using biometrics as verification you need to guarantee that the person, which tries to log in and verify his/her identity, is alive while executing this process. So, for example, while performing iris or fingerprint scans, you need to be confident that person is conscious and that no plastic or other reconstructive tricks are used to cheat the verification process.

This assurance and guarantee of authenticity are not easily performed and comes with a certain complexity, which usually leads to high financial costs in production and installation of such authentication devices. Conclusively from those high expenses and the fact that in the least cases a high-security level (which would call for a Multi-factor authentication) is necessary, contributes to the use of 2FA, when a certain level of security is preferred.

“Multi-factor authentication” is not limited to a third factor *something you are* only, but can also be extended with more eligible factors, e.g. with *some**where** you are*. In particular applies somewhere you are a location factor to the authentication scheme, taking in consideration from which geographical location the authentication attempt occurs. This information is either actively used by limiting the authentication to certain location boundaries (e.g. within a building, campus or country) or by passively observing the location changes, enforcing on rule violation when locations switch unnaturally fast. [34] The latter requires and is used in conjunction with a *time* factor, which assures e.g. that the location shift happens in an appropriative time frame.

The location from where the attempt was sent is determined by either backtracking the IP address or by transmitting the detected position using Bluetooth or GPS. The downside of location as an authentication factor is the uncertainty of the location’s authenticity, due to the fact that it is possible to spoof the location in above-mentioned technologies. This usually results in doubt and disregarding location as a possible authentication factor.

However, the *time* factor can further be used solely as an authentication factor, allowing to restrict the authentication to a specific time schedule.[34]

2.2 Current 2FA Solutions and operating places

Today a lot of services offer Two-factor authentication (2FA), even though it is not commonly adopted by most users. Current 2FA solutions all follow the same approach, but they use different authentication methods, especially in the second factor, while the first factor, the verification of the login credentials, stays the same.

As mentioned before, the second factor describes the authentication with authentication devices. Originally these devices came with smart chips (e.g. integrated on USB drives, access cards or keys), which “store users' information such as passwords, certificates”[1] and need to be physically available during the verification. Later hardware tokens, which generate digit code sequences, were introduced. A more modern approach is wireless authentication via proximity sensing, where for an example a nearby Bluetooth accessory verifies a login. Another contemporary proposal are software tokens, which are successors of hardware tokens.

In this chapter, we are going to take a closer look at the mentioned verification solutions in the second factor and discuss their usual operating places.

2.2.1 Verification with physical devices

The verification with physical devices is the original proposal of second-factor authentication. The concept is, that while the verification of the login credentials (from the first factor) takes place, a physical device must be present as well. Through this device, the entered user credentials, hence the user's identity, are once more validated. This assurance comes through the characteristic of mentioned accessory. While every device is unique by itself, it also must be the only uniquely linked to one user credentials. Conclusively from this, comes that the device needs to have a data storage to store a certification of its user[10]. Furthermore, it is required, that the system is able to recognize the presence and read the content of the device in order to determine if it is the correct device for the entered login credentials. Typical devices are USB drives with an integrated smart chip, access cards or keys. So for example, if an access card is the chosen physical verification device, a card reader is required to read the content from the smart chip. The system proceeds and grants access only if the login credentials are correct and the plugged device is the appropriate one.

Example of a Verification with a physical device
Figure 2.1 shows a typical operating place of an authentication with a physical device: The withdrawal of the money at the ATM. The login of the bank account owner is verified by the presence of the ATM card, a physical device.

Figure 2.1: Typical operating place for a verification with a physical device: At an ATM where the Bank account owner is verified through the presence of his/her bank card.[6]

In this case, the first and second factor are a bit modified, since the bank account holder only needs to enter the PIN (password), while the username (bank account number) does not need to be entered separately. However, the second-factor authentication, persists as intended: The presence of the ATM card is required and verifies the user who must enter the PIN in order to use the ATM (e.g. to withdraw money).[1]

While the authentication with physical devices is still used in some cases, it is not a popular solution nowadays, due to the widespread technology of secure online transactions.

2.2.2 Verification with hardware tokens

Whereas previously the user has been verified through the physical presence of an accessory on the system, the verification with a hardware token is completely different. The verification with a hardware token is achieved through a code sequence, which is generated on a physical device.

This physical device is an accessory, commonly a keychain or key fob, which the user must carry and retrieve when he/she would like to log into a system or service. This is necessary due to the fact that the mentioned code sequence is generated and displayed on the accessory. So in order to log into a system or service, besides the validation of the prior entered user credentials, the verification of the correct code sequence for the particular user is also necessary. The basic requirements for such an accessory are the same as earlier explained in verification with physical devices. However, the uniqueness of the accessory which possesses the ability to verify the user is even more important, since it creates a digit code sequence. This sequence

is the *token*, with which the login verified in the second factor.
The code sequence is generated with the user credentials, which are stored in a certificate on the accessory, and a randomly chosen so called “seed value”[8]. This “seed value”, the user credentials and sometimes other chosen values as well, are applied to a mathematical algorithm, which generates the code sequence. The mathematical algorithm is usually a cryptographic hash function, which uses the *seed value* and the other values to create the pseudo-random digit code sequence.

To increase the security of the verification a time factor is often introduced as well. This time factor has the result that each created *token* is only valid for a period of time. In order to implement this safety factor both devices, the accessory and the login system, must include a synchronized accurate clock. This assures the integrity and validity of each generated token. The generated tokens are called *one-time passwords* (OTP). One token is only valid for a period of time before it becomes invalid and a new token is being created. Each token is inimitable, non-reproducible and only valid for the login credentials of which it has been created.[8]

A system which uses an authentication with hardware tokens proceeds and grants access only if the login credentials are correct and the corresponding *token* for the user has been entered in the valid time period.

Example of a verification with a hardware token

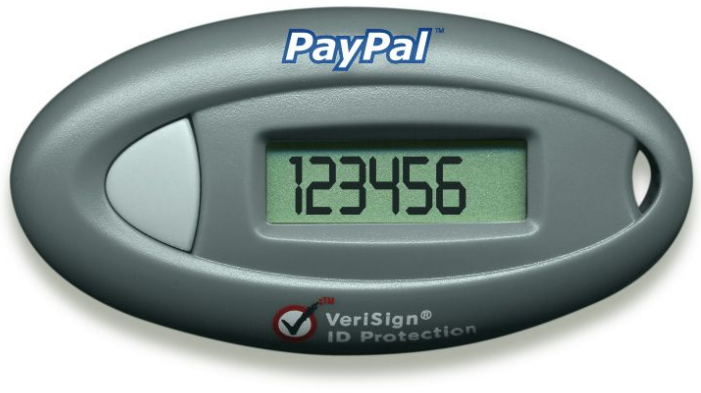


Figure 2.2: A hardware token as keychain by PayPal™[18]

Hardware tokens are used by different systems and services, but the most common operating place is the financial sector, where hardware tokens are used as transaction authentication number (TAN) generators. Figure 2.2

shows a PayPal™hardware token, that generates TAN numbers. An online transaction is only authenticated if the bank account owner was able to login into the online-banking system with his/her login credentials and if he/she is in the possession of his/her *hardware token* (second factor - *Something you have*) and enters the displayed token correctly.

2.2.3 Verification with software tokens

The verification with software tokens is similar to the verification with a hardware token when it comes to the verification process.[15] However, the generation of the *token* does slightly differ, especially concerning where the code sequences are generated. While the hardware token was generated on a physical device, the *software token*, as the name already says, is generated by software.[1] This software can be a desktop or mobile application, which generates the code sequence with the same requirements as explained in the chapter 2.2.2.
Since the token generation is done by software it is not necessary that the software runs on a local system (e.g. smartphone or desktop), but it could also be run on a secure and independent distributed system. This distributed system generates the token and sends it to the user. Common communication channels which are used to receive the software token are mail, SMS or phone calls. The type of transmission which is available depends on the system or service that offers the second-factor authentication.
The verification procedure with software tokens is the same as with hardware token: The system proceeds and grants access only if the user entered the correct login credentials and token.

Example of a verification with a software token

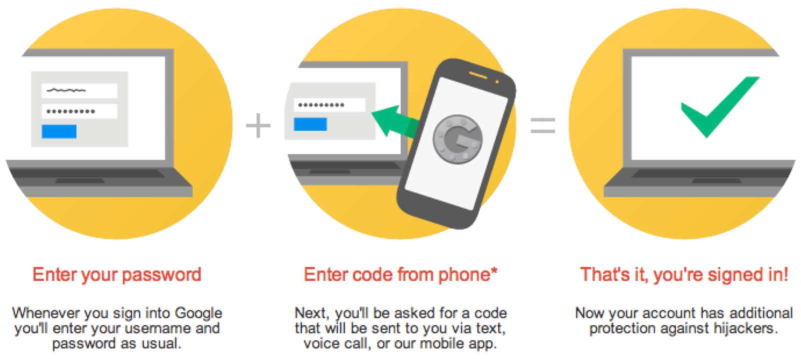

Enter your password
Whenever you sign into Google you'll enter your username and password as usual.
Enter code from phone*
Next, you'll be asked for a code that will be sent to you via text, voice call, or our mobile app.
That's it, you're signed in!
Now your account has additional protection against hijackers.

Figure 2.3: For example, Google is offering a second-factor authentication with soft-ware tokens through its mobile application **Google Authenticator**.[28]

With the mobile application *Google Authenticator* Google is offering a second-factor authentication, where signing into a Google account is secured and verified through software token.
The sample login procedure using *Google Authenticator* is shown in Figure 2.3. In order to activate the second-factor authentication, go to Google's website[1] and enter a valid mobile number during the activation process. The software token is now generated by Google and sent via SMS. To switch to the mobile application and turn on the generation of the software token on the mobile phone, an activation of this feature on the website is needed.
Whenever the user would like to use a service (e.g. Gmail, Drive, Google+, etc.) by Google, the known login credentials are required, as well as a valid generated software token from the connected mobile application.
Aside from the presented *Google Authenticator* other applications are also available. To get an overview of companies and services who are offering Two-factor authentication, take a look at https://twofactorauth.org/.

2.2.4 Verification with proximity authentication

The verification with proximity authentication is the most modern approach of Two-factor authentication and has similarities to the verification with physical devices. However, this similarity does disperse when it comes to the requirement of the physical presence by the authentication device described in chapter 2.2.1 since the user is now verified through the proximity of the device itself. The proximity is limited to the "Personal area network (PAN)", which has a range up to 10m[22]. Technologies which operate in the Personal area network would be for example NFC and Bluetooth.
The approach with the verification by proximity authentication is the usage of authentication devices, which have, for example, the Bluetooth technology included and making them visible or "discoverable" by the system. When a login attempt occurs, the system queries nearby devices and checks if an authorized token is "discoverable". Could the prior entered login credentials be validated and an authorized device is nearby as well, the user is authenticated to use the system.
In order to improve the security, it is furthermore recommended to introduce *one-time passwords* (OTP) as explained in chapter 2.2.2.

Example of a verification with proximity authentication
Figure 2.4 shows a YubiKey NEO with NFC functionality, which can store several OTP for different accounts. In order to access the OTP, the YubiKey mobile application [2] must be installed on the smartphone and the YubiKey NEO is necessary to be near the given device. The smartphone recognizes the proximity of the YubiKey using NFC and grants access to the OTP.

[1]https://www.google.com/landing/2step/
[2]https://play.google.com/store/apps/details?id=com.yubico.yubioath

Figure 2.4: YubiKey NEO is an NFC proximity authentication device [25]

2.3 Limitations and drawbacks of Multi- or Two-factor authentication

While two- or multi-factor authentication brings, in general, a security enhancement in user's authentication, it is certainly not flawless or without disadvantages. This starts with the costs of purchasing, issuing, managing authentication tokens and continues with the setup and ongoing maintenance of the whole infrastructure behind it.

From the user's perspective, two- or multi-factor authentication is also not satisfying, since it forces them to carry authentication token, in any form required, for (ideally) each account they have. The undesired responsibility of two-factor authentication is reflected in the mere adoption of this technology, shown with the example of a quantification in the study[20] by Petsas et al., where only 6.4% of 100.000 Google[3] accounts were secured using second-factor authentication.

Aside from the risk of losing or getting authentication tokens stolen, tokens are prone to different vulnerabilities and attacks. Physical tokens, for instance, are vulnerable to a combination of mechanical and electrical attacks, where the soldered board circuits can be manipulated and the EEPROM[4], storing the authentication key or similar, can be accessed by attaching a “device programmer to the device, while it is still soldered onto the circuit board, and read and write to it at will”[11]. This attack allows to extract all data on the token, including the key and other possible private information. Furthermore, it is possible to reset the authentication key to

[3]http://google.com/

[4]EEPROM - Electrically Erasable Programmable Read-Only Memory

its default, which prevents the legitimate user to authenticate itself in the future.

Neither the hardware or software token are free from vulnerabilities as well. In the case of software token, the systems that are generating the software token are prone to malicious software, e.g. keylogger that captures the token itself or the needed construction parameters, or to illicit remote access or the loss of the system[15][24]. The same applies for hardware tokens, where submitted tokens can be withheld and used for unauthorized access by an attacker using a replay attack[29]. Further, hardware token "[...] need to be replaced every few years when the battery supporting these devices is dead. Cost effectiveness is not the only challenge developing and using security tokens. Security token management is another key issue that thwarts it from popular market acceptance [sic]" and "Furthermore, once a token is lost, the time and cost to replace that token is frustrating [sic]"[15].

Proximity authentication using RFID devices has the drawback that it is exposed by skimming, eavesdropping and replay attacks, which allows the perpetrator to use these techniques to gain access by temporary creating a duplicate or replaying the transmitted data of the RFID device.[16][30]

Biometric authentication, which is the newest authentication and probably presumed the securest, is vulnerable to attacks as the authors Lafkih et al. in their study[13] show. In their study, they have tried to gain unauthorized access to a system secured by fingerprint and face recognition using an altered image of a fingerprint trace or in the case of face recognition, a photography. The result was that it has been sufficient enough to bypass the biometric systems as they state: "Our analysis shows that both systems are vulnerable to the proposed attack and the alteration level has serious impact on the security of biometric systems. The impostor can have a matching score greater than 90% for the tested fingerprint based biometric system and 190/194 associations are matched for the tested face based biometric authentication system [sic]".[13]

Chapter 3

Implementation

The *Implementation* chapter outlines the architectural details of this thesis' solution of a second-factor authentication utilizing Bluetooth proximity sensing. It begins with an insight into the required prerequisites and the general approach. Followed by an more in-depth analyis of the implemented daemon, the provided sample clients and Android authentication tool.

3.1 Prerequisites

3.1.1 OTP

A one-time password (OTP) is "an automatically generated numeric or alphanumeric string of characters that authenticates the user for a single transaction or session"[35]. One of the key advantages of OTP is the invulnerability for replay attacks[29] compared to normal passwords, which is a result of the automatic generation of a new OTP once it has been used as authentication password. In particular, OTP that has been already used, conceivably captured by an intruder can not be used for any later authentication. OTP was introduced by Neil Haller, Craig Metz, Philip J. Nesser II and Mike Straw and became an internet standard in February 1998[7]. Based on their system, two OTP generation algorithms have been introduced and published as RFC4426 and RFC6238, namely "HMAC-based One-time password (HOTP)"[14] and "Time-based One-time password (TOTP)"[17].

OTP are generated using a chained cryptographic hash function, starting with a given seed value. Each future OTP is based on the previous OTP. HOTP uses like its name suggests, a HMAC (see section 3.1.2) to generate the OTP. The necessary HMAC secret is either a random number or counter, which gets increased after each successful authentication. TOTP is the successor of HOTP and introduces a time factor, which sets a time validity to the OTP before it expires and a new one is required.

3.1.2 HMAC

“Hash-based message authentication code” or short HMAC, is a message authentication code which provides integrity and authenticity of a message. HMAC is based on a cryptographic hash function, like MD5 or SHA-1, and a secret key. It was defined and published in the RFC2104[12] by its inventors Mihir Bellare, Ran Canetti and Hugo Krawczyk. A HMAC computation goes as following: $HMAC(K, m) = H((K` \oplus opad) \| H((K` \oplus m) \| m))$

H denotes the chosen cryptographic hash function, K and K’ are secret keys, where K’ is derived from K by padding zeros until the matching block size of H to its right. ipad and opad are constants defined with the byte value 0x5c and 0x36. $\oplus$ describes the logical operation exclusive or.

3.1.3 RSA

RSA is an asymmetric cryptosystem developed by Ron Rivest, Adi Shamir and Leonard Adleman, which allows to exchange messages securely between two communication partners, assuring integrity, authenticity and confidentiality[21]. The cryptosystem’s foundation is the public-private keypair, where the encryption key is publicly accessible and the private key, used for decryption, is kept private.

Both keys are mathematically linked, based on the computationally infeasible problem of factoring the product of two large prime numbers, while the multiplication of these two primes (p, q) is easy.

The encryption of a message m using RSA is accomplished as following: $c \equiv m^e \mod n$. c is the resulting ciphertext, where n is the multiplication of the earlier mentioned large prime numbers p and q. e denotes the public key which was used to encrypt the message. The decryption of the cipher text c is similar achieved: $m \equiv c^d \mod n$. However, in this case, the corresponding private key d of the public key is necessary.

3.2 General Approach

A common and already existing approach for **Bluetooth proximity Authentication as 2nd Factor** is to use near Bluetooth devices to verify and authenticate a user and the following access to devices or services. An example of this concept would be the “Smart Lock” [27] in Google’s mobile operation system Android. This feature allows to unlock the device without user authentication when there is a “trusted” known Bluetooth device nearby. While this might be a good approach to facilitate an authentication, it certainly does not provide an additional authentication factor, since it rather replaces or adds a new method for authentication.
This thesis offers and explains a concept of how to add a second authentication factor to an already existing login scheme using nearby known Bluetooth

devices, which act as authentication tokens. Furthermore, the implementation of this thesis offers the feature, that is not limited to specific login schemes but can most likely be added to every login scheme.

In order to achieve this there are two required components as well as a third optional necessary, which needs to work seamlessly together:

The first component, a daemon, processes requests sent by the second component, a client, which accesses and temporary intervenes the login scheme on a device or service. The previously mentioned requests are triggering the daemon to query for nearby Bluetooth devices. The discovered devices will then be compared to a collection of known Bluetooth devices which have been verified by the daemon to function as authenticating tokens. The positive or negative result of this comparison will then be sent as a response to the client. During the query for Bluetooth devices on the daemon side, the login scheme is intercepted or disabled by the client. Upon receiving a positive response, the login scheme gets enabled and the familiar, formerly blocked login scheme, is needed in order to proceed with the login. If the response states a negative outcome for the daemons query, the login stays intercepted and disabled.

The third component is a tool (in the context of this thesis an Android application) that needs to be run on the authenticating Bluetooth Token. Its main function is to exchange and store OTP (see section 3.1.1) or HMAC (see section 3.1.2) on request by the daemon, which yields an enhancement in security. This security improvement can be enabled by specifying a "security level" in the query request for Bluetooth devices.

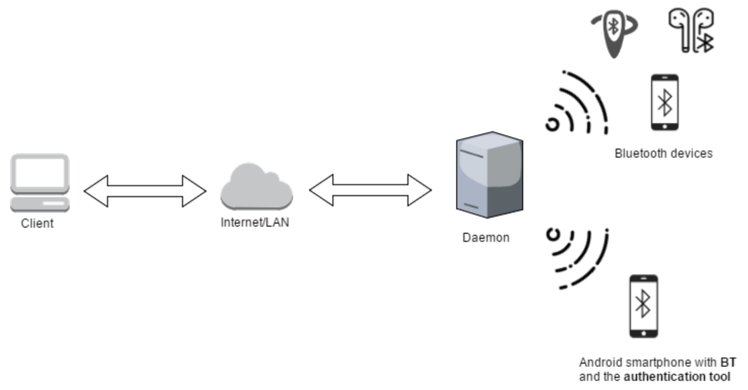

Client
Internet/LAN
Daemon
Bluetooth devices
Android smartphone with **BT** and the **authentication tool**

Figure 3.1: Architectural overview of the different components

The diagram 3.1 shows the different components and their connections. The user interacts with the client (left in the diagram) that is connect to the daemon (middle of the diagram) via a local network or the internet. The daemon discovers the nearby authorized Bluetooth device by querying for discoverable Bluetooth devices.

The available security levels are the following:

1. Proximity only
2. Proximity + OTP
3. Proximity + OTP + HMAC

The first and weakest security level checks only for the proximity of an authenticated Bluetooth device, whereas in the second level an OTP is additionally requested by the nearby device. The third and the most secure level requests a HMAC based on the OTP and a user-specified character sequence. The nearby and known Bluetooth device used to be sufficient enough to send a positive response, whereas now a matching OTP or HMAC, sent by a discovered and authenticated Bluetooth device, is required as well.

By querying for nearby, known Bluetooth devices (and requesting additional information) prior to the conventional verification through the login scheme, a second authentication factor has been applied.

3.3 Implementation Architecture

The daemon, the core of the prospective and new authentication factor, must fulfill certain requirements. One key requirement is the accessibility to the mandatory Bluetooth interface of the system running the daemon. Furthermore, the daemon must provide an interface for connecting clients. Those clients inquire for nearby Bluetooth device. Such an interface is accomplished by offering a REST API, where mentioned clients receive the positive or negative result of the inquiry. In order to communicate with the REST API, it is required to either offer it locally, within the same network, or publicly available, which in both cases requires having a network interface. Since the aim of adding an additional authentication factor is to increase security, the REST API must reflect this aspect as well by only allowing secure HTTP connections. The HTTP connection is secured by using the SSL/TLS protocol instead, as well as validating the established connection using a certificate. In the case of this thesis and for general development purposes a self-signed certificate is sufficient. However, under real-life conditions, a valid certificate, signed by an endorsed certificate authority, is highly recommended.

The second component, the client, has to be individually implemented, depending on the login scheme that it should get extended. However, there are general requirements for such a client implementation: As mentioned before the daemons interface is only reachable using HTTPS, which requires the clients for the ability to establish such a connection. Furthermore, to ensure that the client is not interfacing with a deceptive instance a certificate verification, validating the current HTTPS connection, should be implemented. This verification should get invoked with every sent request to the daemon, aborting immediately the outgoing request when the certificate of the interfacing instance does not match. The request invocation must take place before the login scheme verification process is set off. If that is not assured, the client needs to temporary disable the login scheme, before enabling it again upon a positive response by the daemon.

The third, optional component has to be implemented to offer the previously, in section 3.2, explained security level 2 and 3. To enhance the authentication security further, the component needs to run on the same Bluetooth device, which functions as the authenticating token. Aside from receiving, the device itself also needs to be able to store sent character sequences persistently. These character sequences are valuable and sensitive information, since they reflect the OTP that have been transmitted by the first component, the daemon. In order to persistently store the OTP and later exchange them asymmetrically encrypted on request, are fundamental encryption and cryptography capabilities are necessary. The practice of using asymmetric encryption minimizes the vulnerability of exchanging valuable information with deceived devices.

The diagram 3.2 shows the functional interaction and connection of the three different components at a glance. Once the user dispatches a login event to a system or service and a similiar client, as it explained in sections 3.5 to 3.7, is in place it forwards an authentication request with the configured security level to the daemon. At the daemon side, the Bluetooth discovery will be initialized, trying to discover an authorized Bluetooth device, which acts an authenticating token. Depending on the security level, the daemon tries to establish a connection to mentioned Bluetooth device in order to request the OTP (or HMAC).

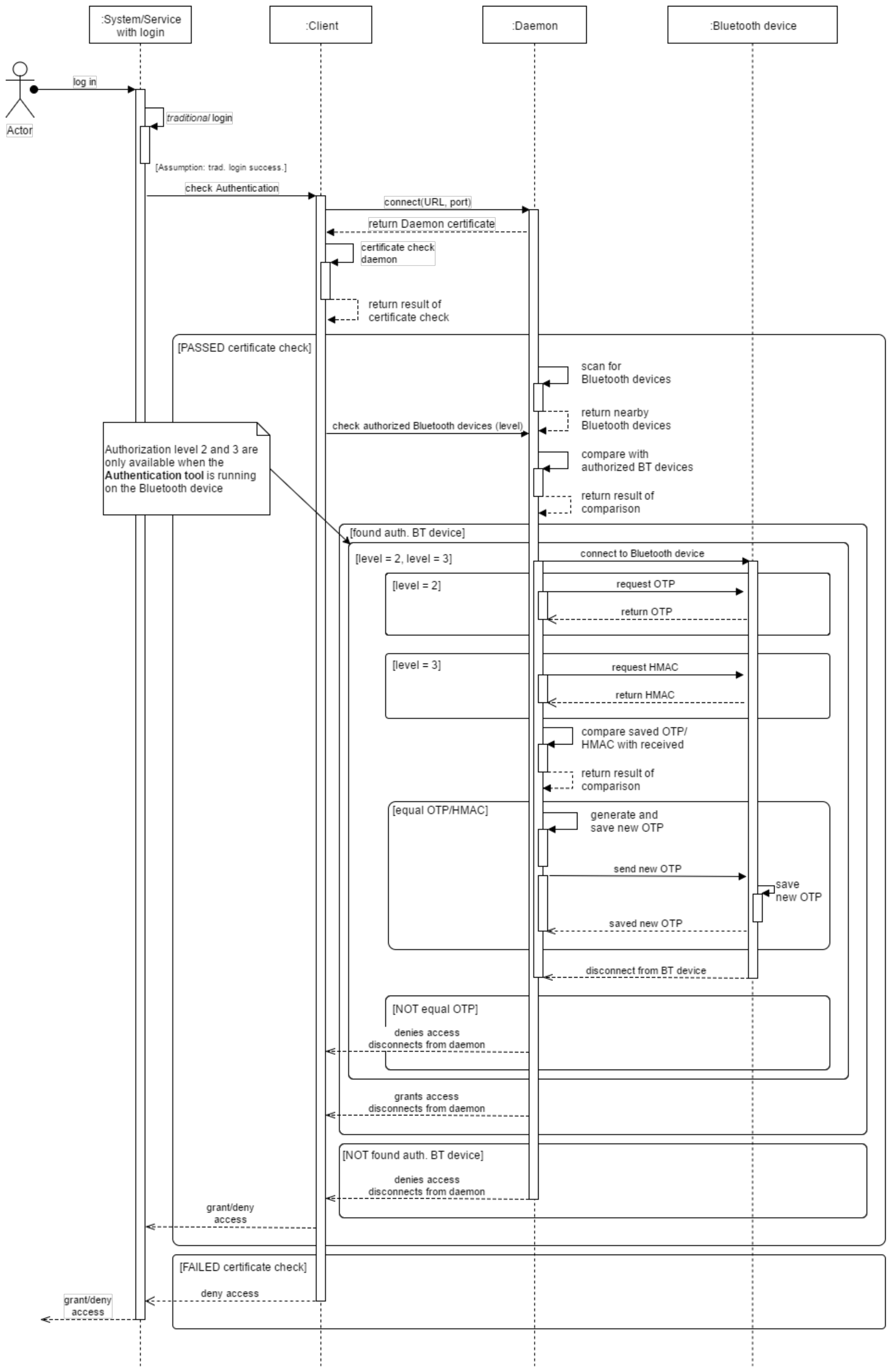

:System/Service with login
:Client
:Daemon
:Bluetooth device
Actor
log in
traditional login
[Assumption: trad. login success.]
check Authentication
connect(URL, port)
return Daemon certificate
certificate check daemon
return result of certificate check
[PASSED certificate check]
scan for Bluetooth devices
return nearby Bluetooth devices
check authorized Bluetooth devices (level)
compare with authorized BT devices
return result of comparison
Authorization level 2 and 3 are only available when the **Authentication tool** is running on the Bluetooth device
[found auth. BT device]
[level = 2, level = 3]
connect to Bluetooth device
[level = 2]
request OTP
return OTP
[level = 3]
request HMAC
return HMAC
compare saved OTP/ HMAC with received
return result of comparison
[equal OTP/HMAC]
generate and save new OTP
send new OTP
save new OTP
saved new OTP
disconnect from BT device
[NOT equal OTP]
denies access disconnects from daemon
grants access disconnects from daemon
[NOT found auth. BT device]
denies access disconnects from daemon
grant/deny access
[FAILED certificate check]
deny access
grant/deny access

Figure 3.2: Sequential procedure for an authorization request

3.4 Daemon

Aiming for a platform independent solution for the daemon Java has been chosen as the preferred programming language. Furthermore, it utilizes the daemon libraries to establish a Bluetooth connection, setup a RESTful API and apply cryptography.

3.4.1 API

The daemons interface, which allows the remote Bluetooth query, is offered by a RESTful API, developed using the “Spark Framework”[1].

The setup of the needed routes is straightforward and easily scalable:

```
[caption={Some Java code},
 label={lst:main-spark}]
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

Demonstrated above is the setup (using lambda expressions) of the ’GET’ route “/hello”. The HTTP request and response are accessible through the parameters “req” and “res”. Aside from the usual response code 200 ’OK’, the example returns “Hello World” in the HTML body. Other HTTP methods like ’POST’, ’DELETE’ or ’PUT’ are configured in a similar manner.

One of the key features of the Spark framework is the uncomplicated way to secure the HTTP connection, allowing only HTTPS connections:

```
public static void main(String[] args) {
  secure(keystorePath, keystorePwd, truststorePath, truststorePwd);

  // routing starts here...
  get("/hello", (req, res) -> "Hello World");
}
```

The **secure** method requires the path and password for both “Java KeyStore” and “Truststore” as parameters.

The Keystore is a special database that contains private keys and certificates with their corresponding public keys. This database must hold a (self-)signed certificate, which is used to validate the API connection. The Truststore database does not need to be set, but usually contains the trusted certificates of the other communication parties. In both stores the public keys

[1]Spark Framework - http://sparkjava.com/

or certificates are usually kept in the “*.PEM” format.

In order to expose no sensitive information to a connecting client, the daemon returns the least needed information possible. It does this by omitting information about the authorizing Bluetooth device and returns only the positive or negative result for the device query. To achieve this security measurement, the daemon informs the client on whether the access should be granted by setting the query result in a custom header field of the APIs HTTP response. By querying for this header field the client knows if the second authentication was successful.Respectively as the client uses the APIs response header the daemon utilizes the request headers to determine which authentication security level the connection client demands.

Querying a parameter of the request headers and a custom header is accomplished as following:

```
1   public static void main(String[] args) {
2     get("/checkForAuthToken", (req, res) -> {
3       String level = req.queryParams("level");
4
5       if(checkAuthentication(level)) {
6         res.header("foundBT", "true");
7         return "";
8       } else {
9         res.header("foundBT", "false");
10        return "";
11      }
12    });
13  }
```

You receive the value of the query parameter by calling the *queryParams*-method on the “req” object, from the function parameters, passing the query parameter. The custom header “foundBT” is added by calling the *header* method on the “res” object, with passing the header name into the first and the value of the second function parameter. The *true* or *false* value is set the header “foundBT” depending on the **checkAuthentication** functions result (see line 5).

3.4.2 Bluetooth

The **Bluetooth communication** of the daemon was established utilizing the *BlueCove*[2] Java library, which was chosen not only for its backward compatibility but also because of its ability to interface with various Bluetooth stacks. The library itself comes as *.jar-file, provided on the website from the developer and needs to be included in the daemons project.

[2]BlueCove - http://bluecove.org/

The Bluetooth discovery is accomplished using the BlueCove DiscoveryListener and the listener callback methods. As the line 17 of the following code snippet shows, the DiscoveryListener is forwarded to the DiscoveryAgent method, which starts the device inquiry. The discovery finishes when the *inquiryCompleted* callback gets triggered, notifying the *inquiryCompletedEvent*.

```
1   public void deviceDiscovery(String[] args) {
2     final Object inquiryCompletedEvent = new Object();
3
4     DiscoveryListener listener = new DiscoveryListener() {
5       public void deviceDiscovered(RemoteDevice btDevice,
6                                    DeviceClass cod) {
7         // do sth. with the found 'btDevice'
8       }
9       public void inquiryCompleted(int discType) {
10        // notify event
11      }
12    };
13
14    synchronized(inquiryCompletedEvent) {
15      boolean started = LocalDevice.getLocalDevice()
16              .getDiscoveryAgent()
17              .startInquiry(DiscoveryAgent.GIAC, listener);
18      // wait for device discovery end
19      inquiryCompletedEvent.wait();
20    }
21  }
```

A Bluetooth connection is established by utilizing the *open* method from the *Connector* instance, shown in line 2 of the following code snippet, to connect to a Bluetooth URL. A Bluetooth URL is returned by the inquiry of the supported service from the device. In order to exchange information with the Bluetooth device, you need to access the InputStream and OutputStream, derived by the StreamConnection cast, of the Connector instance.

```
1   public static void main(String[] args) {
2     StreamConnection streamConnection =
3           (StreamConnection) Connector.open(connectionURL);
4
5     OutputStream out = streamConnection.openOutputStream();
6     PrintWriter pWriter =
7               new PrintWriter(new OutputStreamWriter(out));
8
9     InputStream in = streamConnection.openInputStream();
10    BufferedReader bReader =
11              new BufferedReader(new InputStreamReader(in));
12
13    String payload = "payload"
14    pWriter.write(payload + "\r\n");
15    pWriter.flush();
16
17    String received = bReader.readLine();
```

```
  // Close In- and OutputStream and StreamConnection
}
```

3.4.3 Encryption

Providing an additional, but also *secure* authentication factor requires a comprehensive need of **Encryption** algorithms. As a result of this, the daemon uses a combination of symmetric and asymmetric encryption techniques to secure the communication, as well as the stored and exchanged information.

Symmetric encryption is used to secure valuable stored information about the authenticating token, as for example the hashed Bluetooth address. In this thesis the Advanced Encryption Standard (AES) with an 128bit key size in Cipher Block Chaining (CBC) mode and PKCS5 padding is applied.

Asymmetric encryption is used to secure the sent OTP by encrypting it with the public key of the authenticating token. However, before actually sending the encrypted OTP is signed by the daemon to assure its authenticity. Upon receiving an OTP or a HMAC, the daemon tries first to verify the originating party, before decrypting it. For the verification, the daemon uses the public key of the authenticating token, while for the decryption it uses its own private key. This has, aside from the benefit of an enforced encryption on both sides and therefore an enhancement of security, the advantage that the origin of the exchanged authentication token, the OTP, are valided.

The OTP encryption and decryption is performed utilizing a ECB (electronic code book) cipher with a PKCS1 Padding in conjunction with the 2048Bit RSA public-private key pair of the daemon.

```
1   public byte[] encryptRSA(String _plain, PublicKey _pubKey) {
2     final Cipher cipher;
3     try {
4       cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
5       cipher.init(Cipher.ENCRYPT_MODE, _pubKey);
6       return cipher.doFinal(_plain.getBytes());
7     } catch (Exception e) {
8       // Exception handling
9     }
10  }
```

The decryption method follows a similar pattern, except that the function parameters are a byte array and a private key. As for the cipher mode, like stated in line 5, it must be set to `Cipher.DECRYPT_MODE` followed by the private key argument.

The daemons signature, which provides the authenticity of the outgoing OTP is created by using the RSA key pair and a SHA256 digest-algorithm.

```
public byte[] signRSA(byte[] _data, PrivateKey _privKey) {
    final Signature signature;
    try {
        signature = Signature.getInstance("SHA256withRSA" );
        signature.initSign(_privKey);
        signature.update(_data);
        return signature.sign();
    } catch (Exception e) {
        // exception handling
    }
    return signature.sign();
}
```

The outgoing "One-time password" (OTP) 3.1.1 is an essential and crucial part of the Bluetooth Proximity Authentication. As the name already indicates OTPs are only valid for one session and must further be random, unique and non-reproducible[8], making it a challenge to create them.

To fulfill these specifications, the OTP is created using a SHA256 hash, a digest algorithm based on accumulated hardware and software serial numbers of the system running the daemon, as well as a random initialization vector (IV). On every successful authentication, daemon increases the IV, which alternates the hash, resulting in a new OTP. The particular chosen serial numbers are the following:

BIOS serial number
+ Mainboard serial number
+ Operation system serial number
+ Memory chip hardware number
+ Initialization vector

= daemon's OTP

The Windows serial numbers are retrieved using the Windows Management Instrumentation Command-line (WMIC) tool and on Linux is Hardware Annotation Library (HAL) necessary to obtain the serial numbers. WMIC is always available on Windows systems, where on the other hand HAL is not. Therefore, in order to function properly must the system running daemon install HAL beforehand.

3.4.4 Application

The daemon provides the **Bluetooth Proximity Authentication as 2nd Factor** by utilizing the above-explained technologies and two configured routes, which are called by clients implementing the additional authentication factor:

/checkForAuthToken

/addNewDevice

The first route */checkForAuthToken* must be equipped with the query Parameter "level", mentioning the desired security level of the authentication (explained in 3.2). A second query Parameter “hmac” must be appended when the desired security level equals “3”, specifying the keyphrase with which the HMAC is constructed. In the case of security level “3” when “hmac” is not provided the system behaves the same way as if the provided “hmac” is invalid.

Once a client calls the *checkForAuthToken*, this route executes the query request for Bluetooth devices, which act as Bluetooth tokens. It proceeds further with the verification, depending on the forwarded “level”, requesting a OTP or HMAC by the Bluetooth device.

The second route */addNewDevice* allows adding an additional Bluetooth device to function as authenticating Bluetooth token. On the execution of the route, it queries the surrounding for new Bluetooth devices that have not been added yet. The execution of the route is triggered by the daemon administrator when a new Bluetooth device should be in the surrounding. The administrator of the daemon chooses a Bluetooth device by selecting the proper one from the list of the available choices.

3.5 Firefox Extension

The intention of this thesis’ Firefox extension is to extend the login scheme of a visiting website by adding an additional authentication factor, utilizing the Bluetooth proximity authentication offered through the daemon’s API (see 3.4).

Before the usual website login takes place the daemon must return a positive result for an authenticated Bluetooth token. This is enforced by temporarily disabling the mandatory ’password’ input field of the website’s login scheme, until the reception of the daemon’s response, which results in an intercepted login scheme.

Firefox extensions are developed using Javascript, requiring the Node.js[3] runtime environment and the package manager “jpm”.

The immediate search, followed by the deactivation of input fields with type ’password’ is accomplished by so-called ’content scripts’, which allows to “access and modify the content of web pages”[31] and occurs every time a new web page is opened. Such content scripts are added by the extensions main code and are loaded into the visiting web page.

The content script of the thesis’ Firefox extension instantiates an Event-Listener on the browsers ’window’ object, which gets dispatched on the ’load’ event and queries for the ’password’ input fields by iterating the HTML

[3]Node.js - https://nodejs.org/en/

source code. A found input field gets deactivated by setting the attribute 'disabled' to 'true'. To re-enable it, the attribute has to be set back to 'false'.

```
1   window.addEventListener("load", function(event) {
2     var pwdInputs = [];
3     var inputs = document.getElementsByTagName("input");
4     for (var i=0; i < inputs.length; i++) {
5       if (inputs[i].getAttribute("type") === "password") {
6         inputs[i].setAttribute("disabled", "true");
7         pwdInputs.push(inputs[i]);
8       }
9     }
10    if(pwdInputs.length >= 1) {
11      self.port.emit("makeRESTCall");
12    }
13  }, false);
```

When a 'password' input field was found and deactivated, the content script informs the main code of the extension that a REST call to the daemon is necessary, so it can verify for authenticating Bluetooth token (seen in line 11).

The REST call to the daemon is made using the XMLHttpRequest API, requiring the URL of the daemon's API. EventListener is listening for 'load' or 'error' event. Once the event has been caught, it either receives the daemon's response or handles the unavailability. If the daemon's response is successful, the Firefox extension checks for the 'foundBT' header's value in the response. If the value is 'true', the 'password' input field gets unblocked. In the case of a 'false' value, the input field stays disabled. On another hand, if the daemon's response is unsuccessful, the input field stays blocked as well.

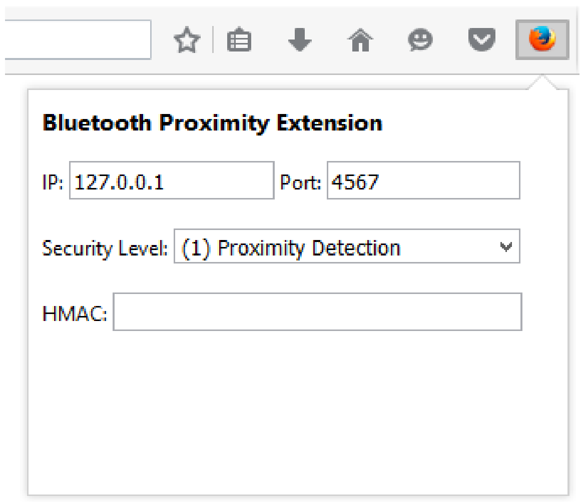
Since the daemon's API accepts only secure HTTP connections it might be necessary to add the daemon's certificate to the browser's trusted certificates.

The daemon's API URL is set by specifying its IP address and port, as well as the desired authentication security level, in the extension's interface, shown in figure 3.3. A HMAC key must be set when the authentication security level is set to level "(3) Proximity D. + OTP + HMAC".

3.6 Linux PAM Client

Pluggable authentication modules (PAM) "are at the core of user authentication in any modern Linux distribution"[19] and provide independent authentication methods, enabling how individual applications authenticate users on Linux. The combination of different modules constructs and strengthens the Linux authentication scheme, as well as applications using PAM.

The Linux PAM Client of this thesis offers an authentication module


Bluetooth Proximity Extension

IP: 127.0.0.1 Port: 4567

Security Level: (1) Proximity Detection

HMAC:

Figure 3.3: Screenshot of the Firefox extension interface

which implements the verification for authenticating Bluetooth tokens.

The communication between the authentication module and the daemon behaves in the same way as described in section 3.5. In this case, the authentication module takes the place of the Firefox extension.

PAM are made available by editing the configuration file of an application, that authenticates users, found in `/etc/pam.d/` or `/etc/pam.conf` and adding a line following this syntax:

type **control-flag** **module-path** **module-arguments**

The first part specifies the type of authentication that is being used for this module. There are four different **types**, where each type is corresponding to a different aspect of the authorization process. Depending on the use of the PAM module a particular type must be set. The available types are displayed in table 3.1.

Followed by the authentication type is the **control-flag** defining[4] how strong the PAM enforcement is when the authentication fails. A overview of the available control-flags gives table 3.2.

[4]Definition from http://www.tuxradar.com/content/how-pam-works

Table 3.1: Available PAM module types

type	Definition
auth	Module determines if the user is allowed to access the service, if their passwords has expired, etc. E.g. requests or validates a password.
account	Module verifies that the access of user is allowed and who they claim to be, by a password or using biometrics.
session	Module configures and manages user sessions, but it can also perform additional tasks that are needed to allow access.
password	Module is used to change user passwords.

Table 3.2: PAM control-flags

control-flag	Definition
required	If the module fails, PAM immediately returns a failure result to the application; no further modules in the stack are called.
requisite	If the module fails, PAM returns a failure result to the application but it will continue to call the next module in the stack.
sufficient	If the module succeeds, PAM returns a ’pass’ result to the application and no further modules in the stack are called
optional	The pass/fail result of the module is ignored, which generally means that the module is being called to perform some operation, rather than participating in the pass/fail decision for the stack.

The last arguments describe the module’s location and its arguments.

In order to add this thesis’ Linux PAM Client, the following syntax is needed:

type	auth
control-flag	required
module-path	<path>/bt-pam.so
module-arguments	<path>/certificate.pam
	<ip-address>:<port>
	<level>
	<hmac-keyword>

The module-arguments define the certificate location, IP address and port, as well as the security level of the daemon. A fourth module argument for the HMAC key is necessary when the chosen security level is 3.

PAM are developed using the C programming language and require the PAM library 'libpam' which offers an interface to the methods which need to be overwritten. Upon compiling the PAM, the used libraries must be set. In addition to the libraries, during the process of compiling PAM, the "-fPIC" flag must also be set. This flag tells the compiler to generate position-independent code, enabling the PAM to run memory position independent. The Linux PAM Client of this thesis requires cURL[5] to construct and send HTTPS to the daemon.

Both, curl and 'libpam' library must be available in development environment and can be downloaded using `apt-get install libcurl4-openssl-dev libpam0g-dev`.

A basic implementation for a PAM module contains the import of the header files by the PAM library 'libpam', followed by the definition which authentication type(s) will be supported by the module.

```
/* Include PAM headers */
#include <security/pam_appl.h>
#include <security/pam_modules.h>

/* Define which PAM interface the PAM module provides */
#define PAM_SM_AUTH     // def. for authentication
#define PAM_SM_ACCOUNT  // def. for account
#define PAM_SM_SESSION  // def. for session
#define PAM_SM_PASSWORD // def. for password
```

Each definition automatically requires the implementation of the service functions for their authentication type. These functions are used to either implement the new authentication method or "should return PAM_SERVICE-_ERR and write an appropriate message to the system log".[26]

For example, the PAM module offers a new authentication method for the type 'auth' (PAM_SM_AUTH) with the implementation of the following methods:

```
// ... include's and define's
PAM_EXTERN int pam_sm_authenticate(pam_handle_t *pamh, int flags, int
    argc, const char **argv) {
  // implemention of the authentication interface
}

// defined but not implemented
PAM_EXTERN int pam_sm_setcred(pam_handle_t *pamh, int flags, int argc,
     const char **argv) {
  // implementation to modify the credentials of the user with respect
```

[5]cURL - https://curl.haxx.se/

```
18     // to the corresponding authorization scheme
19   }
```

Each function has the same arguments, where the first “pam_handle_t” is the handler by PAM. Secondly, the “flags” argument allows setting constants such as ’PAM_SILENT’, to not emit any messages or ’ PAM_-DISALLOW_NULL_AUTHTOK’ to prevent NULL as allowed user. The last two, “argc and argv arguments are taken from the line appropriate to this module”.[26].

Aside from the normal CURL setup for HTTPS connections, the following lines are especially important in the context of this thesis:

```
20     // ... instantiate and initialize 'curl'
21
22     curl_easy_setopt(curl, CURLOPT_URL, URL);
23     /* Certificate check */
24     curl_easy_setopt(curl, CURLOPT_SSLKEYTYPE, "PEM");
25     curl_easy_setopt(curl, CURLOPT_CAINFO, certlocation);
26     /* Connection will terminate by certificate miss-match  */
27     curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 1L);
28
29     // ... continue with setup and perform curl
```

The flags CURLOPT_SSLKEYTYPE (line 4) and CURLOPT_CAINFO (line 5) specify the format and location of the forwarded daemon certificate for the CURL setup. CURLOPT_SSL_VERIFYPEER set to 1L ($\hat{=}$ TRUE) indicates, that the connection immediately terminates once the verification of the peer certificate, compared with the forwarded certificate, fails.

3.7 Windows Desktop client

Unlike the previous client implementations, that extended login schemes, the Windows Desktop client uses the Bluetooth proximity authentication to secure workstations by automatically signing out and redirecting the user to the lock screen once no authenticating token could be found within a specified time-interval. This implementation approach shows how the security of a system can be passively enhanced and benefits by the Bluetooth Proximity Authentication.

The Windows Desktop client is, like the daemon (section 3.4), implemented using Java. The sign-out and redirection to the lock screen is achieved by accessing the Java runtime environment and executing a Windows command-line command:

```
1   private void lockscreen() {
2     try {
3       final String path = System.getenv("windir") + File.separator + "
        System32" + File.separator + "rundll32.exe";
4       Runtime runtime = Runtime.getRuntime();
```

```
    Process pr = runtime.exec(path + " user32.dll,LockWorkStation");
    pr.waitFor();
  } catch (IOException | InterruptedException e) {
    e.printStackTrace();
  }
}
```

Establishing a connection and sending a REST call to the daemon's API is accomplished by instantiating a 'HttpsURLConnection', forwarding the daemon's connection URL (see line 15). However, before opening the connection, a 'Keystore' object is constructed and a certificate is added (line 2-4). The objective of this is to verify, using the added certificate, that the outgoing connection is genuinely interfacing with the daemon's API once the connection is resolved (line 23).

```
1   private void establishConnection() {
2     KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
3     keystore.load(null, null);
4     keystore.setCertificateEntry("BT Daemon", mBTx509Cert);
5
6     TrustManagerFactory tmf = TrustManagerFactory.getInstance(
      TrustManagerFactory.getDefaultAlgorithm());
7     tmf.init(keystore);
8
9     SSLContext sslCtx = SSLContext.getInstance("TLS");
10    sslCtx.init(null, tmf.getTrustManagers(), null);
11
12    SSLSocketFactory sslFactory = sslCtx.getSocketFactory();
13
14    URL url = new URL(getHostURL());
15    HttpsURLConnection connection = (HttpsURLConnection) url.
      openConnection();
16    connection.setSSLSocketFactory(sslFactory);
17    try {
18      connection.connect();
19
20      Certificate[] listCerts =
21        getConnection().getServerCertificates();
22        for (Certificate cert : listCerts) {
23          cert.verify(mBTPublicKey);
24        }
25    } catch (Exception e) {
26      lockscreen();
27      connection.disconnect();
28    }
29  }
```

Unlike the previous client implementation, the Windows Desktop client has a graphical user interface (GUI), displayed in figure 3.4, to configure the parameters. It offers input fields to specify the IP address and port of the daemon, as well as a field to specify the HMAC key (when necessary). If HMAC key is not needed, its field will be disabled. The security level of the Bluetooth proximity authentication is selectable by a drop-down list. The displayed time, adjustable by a slider, indicates the interval on how often the verification check occurs for authenticating tokens.

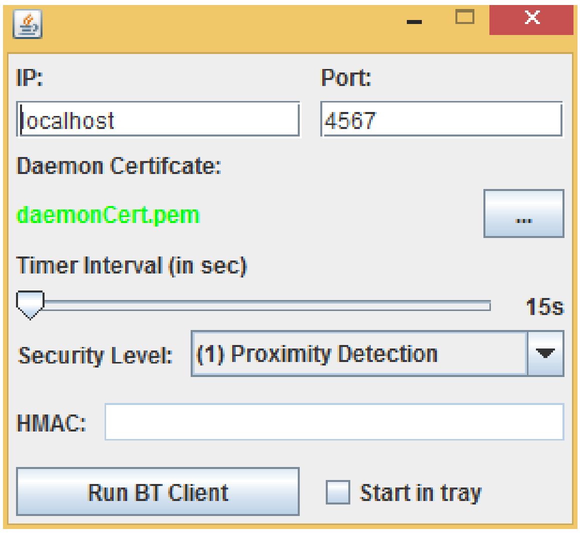

IP: localhost
Port: 4567
Daemon Certifcate: daemonCert.pem
Timer Interval (in sec) 15s
Security Level: (1) Proximity Detection
HMAC:
Run BT Client
Start in tray

Figure 3.4: Graphical user interface of the Windows Desktop client

3.8 Authentication tool

The authentication tool, in the case of this thesis an Android application, is the third, additional component of the Bluetooth Proximity authentication and enables the second and third security level (described in section 3.2). Android as an ecosystem for the authentication tool is not mandatory, any other solution is feasible if it is executable on the authenticating Bluetooth token and fulfills the requirements, explained in section 3.3.

One of the tools key requirements is that the execution environment must be the same Bluetooth device, which functions as the authenticated Bluetooth token. Through this authentication tool an interface to the daemon's is offered to exchange OTP via a Bluetooth connection. Furthermore, it allows the specification of the mandatory keyphrase when a HMAC is requested (security level three). Aside from that, the authentication tool must hold also the RSA public key of the daemon.

Before the first OTP exchange occurs, the daemon requests the RSA public key of the authentication tool. After the public key is obtained, every future transmission is secured and encrypted using the authentication tools public key, making the received OTP only decryptable by the corresponding private key held by the authentication tool.

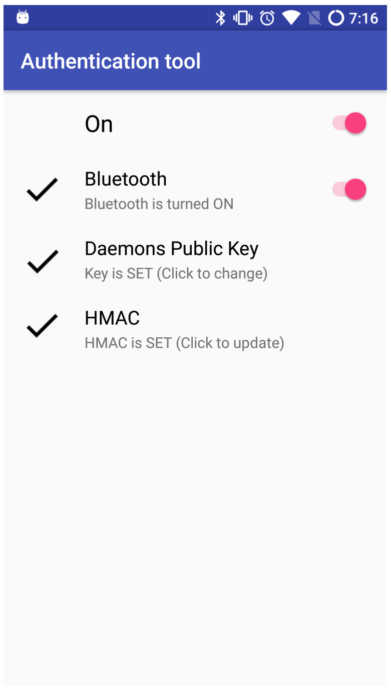
However, before the authentication tool decrypts a received OTP, its origin is verified using the daemon's public key. If the daemon is not validated as the OTPs origin, the tool declines the OTP and terminates the Bluetooth connection.

The Bluetooth setup, querying and transferring of data is based on the BluetoothChat Sample[6] provided by Google and has been extended to the further requirements of the authentication tool:

Allowing a constant authentication and exchange of OTP or HMAC the Android application must have a background service continuously running, which is listening and awaiting an incoming Bluetooth connection by the daemon. The Bluetooth connection attempt by the daemon must be established using the matching Bluetooth UUID[7] of the Bluetooth Server hosted by the authentication tool.

To stop, (re-)start or change the configuration, like the daemon's certificate or HMAC keyphrase, the independently running service is accessible by an Android activity (Figure 3.5).

To minimize and simplify overhead uses the authentication tool the respectively same encryption, decryption, verification and validation implementation like the daemon, explained in the section 3.4.3.


7:16

Authentication tool

On

✓ Bluetooth
Bluetooth is turned ON

✓ Daemons Public Key
Key is SET (Click to change)

✓ HMAC
HMAC is SET (Click to update)

Figure 3.5: Android activity which accesses the service of the authentication tool

[6]BluetoothChat sample - https://github.com/googlesamples/android-BluetoothChat
[7]UUID - Universally Unique IDentifier

The activity, displayed in figure 3.5, interacts with the authentication tools service and allows to interrupt and restart its execution. The daemon's public key, as well as the HMAC keyphrase, is easily set and updated within the activity. Furthermore. it is possible to enable or disable the Bluetooth interface on the smartphone.

Chapter 4

Results

This chapter presents the results of the implementation that have been described earlier in chapter 3. It discusses the added security, performance and the overcome obstacles during the implementation. While at the same it highlights open concerns, that could not be solved or came up during the development and tries to give pointers for possible solutions. Each section of this chapter covers one of the three components of this thesis's 2FA with Bluetooth proximity solution.

4.1 Overall results

Predominantly, the implementation results of the daemon (which offers the **Bluetooth Proximity as 2nd Factor authentication** and the provided clients are a success:

The approach to decouple and publicize the authentication process using a daemon benefits the availability of the authentication factor through connecting clients and makes the Bluetooth Proximity authentication available even for instances which do not have a Bluetooth interface.

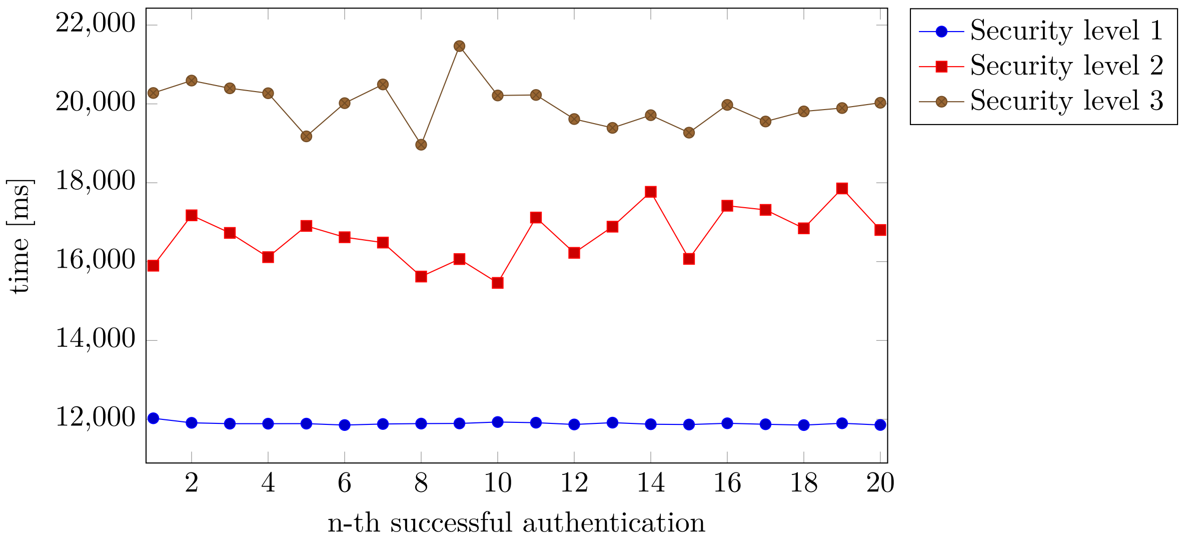
A client, which links the connection to the daemon and the extending login scheme, is easily created and requires only the abilities to make secure REST API calls as well as a validation scheme to verify that the client is interfacing with the correct daemon. Further, the client needs to read the REST call's response header and react on the returned value.

A key advantage of the Bluetooth Proximity authentication is that it has no limitations to a particular subset of Bluetooth devices. However, the consciousness if a Bluetooth device is suited to function as authenticated Bluetooth token lays by the daemon's administrator, especially for the described second or third security level.

4.2 Results and security of the daemon

A disadvantage of the Bluetooth technology itself, which affects the daemon is the long query time for Bluetooth devices. A Bluetooth query using the *BlueCove* library takes in average around 11.694 seconds to finish. Although it is acceptable in the average case scenario, when it comes to the authentication by the daemon, it results in an increase of the authentication time and holds off the user login. The complete Bluetooth authentication process using the first security level takes about 12 seconds, whereas in comparison second security level needs 17 seconds and third 20 seconds. The average time was calculated using the arithmetic mean over 20 successful authentications using the Windows Desktop client. The overall test results for each security level are displayed in the plot 4.1.

Figure 4.1: Authentication time for each security level



Furthermore, it is not possible with this thesis solution to have multiple authentication requests at once, which causes requests to queue and get blocked in the meantime. A solution for this would be to extend and handle the connecting clients in multithreading manner. This would elevate the limitation of one client at a time to (theoretically) 7 clients at once. The limitation of specifically seven clients is due to the restriction of the 3bit length of the network address in picconet.[9]

The appliance of strong encryption and validation, on the other hand, is a key advantage during the OTP exchange via Bluetooth, by the daemon and connected client. Through this strict enforcement, the origin of the

OTP is verified and is only decryptable by the desired recipient. This thesis' implementation uses the hash-based one-time password (HOTP) as its OTP. As explained in section 3.1.1 the HOTP requires a key for the underlying HMAC. This key is the concatenation of various serial numbers of the system running the daemon.

While OTP, in general, restrains replay attacks, the concatenation of the above-mentioned serial numbers prevents the deception of the authentication daemon. The reason for this is because the serial numbers are bound to the system running the daemon. If an attacker is able to obtain a copy of the authentication daemon, the copied instance does not continue to generate valid OTP for the future authentication, since it uses the serial numbers of the new system.

Aside from the protection against replay attacks, which is achieved by session token, the OTP of the daemon is equipped with additional security precautions. In details, the daemon's API is further secured by accepting only secure HTTP connections, as well as the daemon's authenticity towards connecting clients assured due to the certificate securing the HTTPS connection. The used certificate is signed with an RSA public-private key pair which has the key size of 2048Bit. For development purposes, the certificate is self-signed; however, in production state, it is essential to sign it by an endorsed certificate authority (CA).

Further, the communication and the data exchange between the daemon's Bluetooth interface and the authentication tool (3.8) is secured through encryption and decryption using yet another independent RSA keypair with a key size of 2048Bit. The necessary random initialization vector (IV) of the RSA keypair is automatically generated using the *SecureRandom* class provided by the Java cryptography extension (JCE). The unpredictability of the IV is guaranteed since the *SecureRandom* implementation complies with the RFC1750 standard[33]. Every sent data also contains the daemon's signature which verifies and validates the origin.

The chosen encryption and decryption cipher mode, the Electronic Codebook (ECB) with a PKCS1 Padding might not be a good cipher since it may leak information about the plaintext. The leaked information could allow attackers to detect a common prefix or substrings and may help them to forge or even create valid OTP. However, for the proof of concept of this thesis, the cipher is sufficient enough and the exchange to more sophisticated cipher mode should be easily accomplished. Since the authentication tool of this thesis utilizes the same encryption configuration it is recommended to change it as well.
Aside from the wrong selected cipher, the daemon fulfills the CIA (**C**onfidentiality, **I**ntegrity and **A**vailability) triad of information security utilizing the above-mentioned security measurements.

4.3 Results and security of the provided clients

The client implementation, in form of a Mozilla Firefox Extension, has been successfully implemented and performs as intended, albeit there are restrictions on websites for which the Firefox extension is applicable. The reason is that in order to send REST calls, the visiting website must allow "Cross-origin resource sharing" (CORS). CORS allows the access and requests resources from external domains, which are needed in the scope of the Firefox extension to start the REST call. Some websites like Google[1] or Amazon[2], allow CORS whereas others like GitHub[3] do not.

In addition to that, in the planning phase of the Firefox extension, it was considered to include the restriction to prevent the autoform filling up the login credentials prior to the daemon's positive response for a Bluetooth authentication token. The intention was to use Mozilla Firefox *nsILoginManager*[4], which has according to the documentation, the methods *findLogins* and *fillForm* to achieve this. However, it seems to be the case that in the meantime from the last revision (15th March 2015) of the documentation until the publication of this thesis, Mozilla removed the *fillForm* function. Due to that removal, it was not possible to implement this planned feature.

The Firefox extension abstains from a custom certificate input or validation implementation and utilizes the Firefox's provided solution. In addition to this, it verifies the validity of the entered IP and port of the daemon. Due to the fact that the daemon's API only accepts secure HTTP connections, the extension sends secure REST calls. Insecure calls are never sent or get immediately rejected by the daemon. If the extension interfaces a deceived daemon or a mismatching certificate, the connection immediately terminates.
The implementation of PAM is not well documented, yet it was a fairly straightforward process using the "Linux-PAM Module Writers' Guide" "to write a module that conforms to the Linux-PAM standard"[26] and achieved in the end.

The installation and extension of PAM is simple. However, it is crucial to specify the correct path to the daemon's certificate, indicating that an incorrect location or invalid certificate would fail the authentication and eventually might lock the user out. In addition, it is required to specify the PAM module at the appropriate position in the configuration file. Misplacing the line would either not enforce the additional second factor or might weaken instead of strengthening the authentication scheme. The proper control-flag (see 3.6) for the PAM module is important as well.

[1]https://google.com/

[2]https://amazon.com/

[3]https://github.com/

[4]https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Reference/Interface/nsILoginManager

The underlying cURL[5] implementation of the Linux PAM sends the authentication request securely to the daemon. Aside from that, it also verifies the daemon's certificate and terminates the connection immediately on a mismatching, missing or invalid certificate. The location of the certificate which is used to compare the current connection is specified as PAM argument in the configuration file.

Using an open-source library like cURL may distrass and raise concerns when you would like to improve an existing login or extend an authentication process. However, cURL has been proven to be a reliable and secure libray and reflects this by the over 200 companies using it. cURL is under continuous development with dedicated and free contributors which support the fast prevention against vulnerabilities. A security audit initiated by the cURL project owners and conducted by the third-party company Cure53 [6] in August 2016 identified eleven vulnerabilities, which were until then undiscovered. It may sounds like a lot, but only three out of the eleven vulnerabilities had a *high* severity rank and Cure53 stated nevertheless that "the overall impression of the state of security and robustness of the cURL library was positive" [3]. The found issuses were immediately assessed and have been resolved in the version 7.51.0. An overview of previous cURL versions and their vulnerabilities can be found online[7].
The Windows Desktop Client is the simplest client provided with this thesis. Nevertheless, it shows effectively how an authentication and active user session can be passively strengthened by the provided Bluetooth Proximity authentication. The client is supported from Windows XP onwards and requires only the Java Runtime Environment.

The REST call to the daemon is established using a secure HTTP connection, which is verified using the supplied daemon's certificate of the Windows Desktop Client. When the verification of the connection fails, the Windows lock screen gets invoked as it would normally when no authenticating Bluetooth token is found.

4.4 Results and security of the Authentication tool

The authentication tool, which reflects and offers the general concept of a higher-up security level than just proximity authentication, has been implemented as Android application and consists of two components:

A background service which handles incoming requests that have been sent by the daemon and foreground application which is needed to configure the authentication tool. Depending on the incoming requests reacts the

[5]cURL - https://curl.haxx.se/
[6]Cure53 - https://cure53.de/
[7]cURL vulnerabilities overview - https://curl.haxx.se/docs/vulnerabilities.html

client by returning the OTP or its public key. The exchange of the OTP (or public key) is done automatically once it is assured that the request origin is genuinely the authentication daemon. While this approach certainly favors the authentication time, the missing authentication consent could be seen as a disadvantage. A received OTP is saved in authentication tool's private sharedPrefeneces.

The foreground application allows setting the daemon's certificate, which is used to encrypt the outgoing OTP, as well as specifying the HMAC key phrase. The clients own public-private encryption keypair is saved in the "Trusted Execution environment" (TEE) storage, which is a secure isolated environment that runs in parallel with the operating system. By saving the keypair in the TEE the valuable encryption and decryption keys are secured from an intruder since they are only accessible by the same application that stored them. The feature to save the keypair in this storage is only available on Android device running Android 6.0 ("Marshmallow") or higher. Nonetheless, it would be possible to implement the authentication tool for earlier Android versions, starting from Android 2.1 ("Elair"). However, the advantage of mentioned secure storage location for the key pair would be disregarded.

Saving the OTP in the sharedPreferences protects the access from other application, yet might it be at risk when the Android device is (intentional or unintentional) rooted since any sharedPrefences of every app is accessible by a user with root privilege. While on the contrary, the mandatory encryption of the OTP before the transmission the security favors.

Chapter 5

Discussion

In this chapter the importance of second-factor authentication, alongside the thesis's provided **Bluetooth Proximity Authentication as 2nd Factor** will be discussed and further in relation to similar, existing and novel solutions compared.

The overall aim of this study was to cover the theoretical aspects and advantages of second-factor authentication, as well as providing a proof-of-concept for utilizing Bluetooth proximity detection as an additional authentication factor. The prototype includes a daemon, which functions as an authorization agent, besides three sample clients.

As mentioned in *Related Work* has second-factor authentication highly influenced our everyday lives, starting from the early days of its existence. However, the financial sector is undeniably its largest field of application. The tendency to especially use this security feature in this field has been early observed and predicted by Bruce Schneier, a cryptography security specialist, as he states: "I predict that banks and other financial institutions will spend millions of dollars outfitting their users with two-factor authentication tokens."[23]

He further elaborates the changes in security attacks and raises concerns about possible attack surfaces like 'Man-in-the-Middle Attacks' (MITM), where an attacker bypasses the authentication by implanting a middleware between the user and the authenticating system. The middleware blocks the user's authentication attempt while in the meantime it captures the login credentials and further mandatory authentication information. The attacker could then misuse this information for his or her own possible malicious intentions. While various second-factor systems are vulnerable to MITM attacks is the authentication scheme provided with this thesis not prone to such attacks[23]. The reason for this is the introduction of one-time passwords (OTP), which are exchanged with a third entity, the Bluetooth token's 'authentication tool', and requested during the verification request. The significance of the OTP is the characteristic that an attacker is not able to create without or predict an OTP. This due to the fact that the OTP are based on unique serial and part numbers of the system running the daemon, which eliminates the possible attempt as well of using a copy of the daemon for the implanted middleware, whether an instance of the daemon can be obtained or not. Further is any communication between the daemon and this 'authentication tool' with each other RSA public key, encrypted. Only with the corresponding private key, it is possible to decrypt the sent data. Its decryption is only possible via the corresponding private key held by the daemon. Additionally to the encryption is every newly sent OTP signed by the daemon and upon arrival verified by Bluetooth token's entity.

However, the resistance against MITM is only granted through the introduction of the different security level by the authenticating daemon (see section 3.2), starting from security level 2. Therefore, the first security level ("Proximity only") is the weakest, which offers a malicious third party the possibility to impersonate the authenticated Bluetooth token by spoofing its Bluetooth address. *Spoofing* describes the technique to falsify the MAC[1] of a Bluetooth device. The MAC is essentially used to identify the device and is further used to connect and communicate to mentioned device. An attacker could use Bluetooth Spoofing to masquerade his/her own Bluetooth MAC with the MAC of an authenticating Bluetooth device and therefore imitate it. There is no known way to de-masquerade or reveal that a Bluetooth device is imitating another, yet through of the introduction of the security level and the request of OTP an additional challenge is introduced which secures the authentication scheme. The same conclusion and subsequentially resolution were made by Czeskis et al.[4]

With the introduction of the different authentication security level and the exchange of OTP, through the "authentication tool", comes an overhead of additional authentication time which needs to be weight in with the favored security. The more secure the authentication is, the longer the verification takes time, which can be seen in the comparison of the three security level. The most secure authentication, using security level 3, needs 8 seconds in average more compared to the weakest with security level 1 with 12 seconds. While the second security level only needs 17 seconds in total. Another weak point by the higher-up security level is the fact that they limit the eligibility of suitable Bluetooth devices acting as authenticated Bluetooth devices. Normally, with security level 1, every Bluetooth device is feasible to act as an authentication token. However, due to the need of the 'authentication tool' starting from security level 2, only Bluetooth devices with matching requirements (described in section 3.8) are possible. Since in the most cases a secure yet fast authentication favored is the second security

[1]Media access control address (MAC) - the unique identifier for a network communication interface

level the best compromise, even though it comes with its limitation.

The additional overhead time during the verification due to Bluetooth query and the cryptographic appliance is a common and shared occurrence in the similar solution by Czeskis et al. suggests, stating “an average login time of 24.5 seconds for a 2-factor login”[4]. Additionally to the similar authentication time is the resemblance with the thesis’s creation of a distributed system functioning as an authentication instance (in the context of the thesis: the daemon) as well as a connecting client utilizing the system. Further, Czeskis et al.’s solution share the idea of an ’authentication tool’ where previously sent token are compared and determining the success of an authentication. In the case of Czeskis et al. the connecting client is a ’Chromium’ web browser[2], which can be seen equivalent to a Mozilla Firefox web browser equipped with the thesis’ provided sample client, the Firefox extension (see section 3.5).

However, in contrast of the similarity is the Bluetooth feature that has been integrated directly into the web browser by the scholars Czeskis et al. The query for Bluetooth devices is managed inside the web browser and accomplished using the Bluetooth interface of the system running the web browser. The result of the found Bluetooth devices and the return of requested token by the device will be forwarded to the authentication instance which finally determines whether to grant access or not. With this thesis proposed solution, the Bluetooth query is only initiated by the Firefox extension (or any other client), the query request will be forwarded to the daemon, which is seemingly the authentication instance as well. The Bluetooth query is executed using the daemons Bluetooth interface and the access permission, minding the evaluation of the received OTP (when requested), will be returned to the client. This procedure is a decoupled solution, which is distinctly different to other novel or proposed second-factor authentication that use Bluetooth Proximity. The advantage of it is that it provides a fairly endless range to use authentication scheme and offers an uncomplicated way to provide clients without having the requirement to have Bluetooth interface. Nevertheless, an implementation of a client does not come without obstacles or possible weak points, but this vary depending on the application area of the client, which will be highlighted more detailed later on.

Another point is the thesis’ encryption and signature practice which are nearly identical to a proposal for an extended authentication scheme to enhance the security of mass storage devices by Eldefrawy et al.[5]. They are using the same concept to validate data changes and exchange by a known user with an authentication server. The usage of advanced cryptographic methods, in addition to applying a timestamp, is similar to the thesis’s solution using RSA and the construction of OTP or HMAC. Additionally,

[2]Chromium browser - https://www.chromium.org/Home

Eldefrawy et al. are creating a physical token by saving an unique signature on the message storage device, constructed using the mass storage exclusive identifier and an user-id-password combination. The mass storage device alone, as a physical token, would not be sufficiently secured against impersonation attacks. However, in association with the added authentication server is such an attack prevented.

A weak point of the thesis' provided Firefox extension is the fact that the source code of any loaded JavaScript file is accessible and clearly readable to the Firefox user, and therefore to any possible intruder as well. It is possible to obfuscate the source code, yet it would only diminish the attack surface slightly. This exposure simply allows to either replace or modify the extension's content-script, i.e. the IP address of the daemon. Additionally, the user could decide to disable the global JavaScript execution by Firefox, which would result that the extension is not instantiated and the additional second factor can not be applied. When Firefox itself becomes a target by an intruder, the general rejection of invalid certificates could be disabled and exposes Firefox to any malicious attacks, but further would cause the Firefox extension to accept (due to the missing certificate validation) any connection to a deceived daemon instance.

A shared vulnerability of the Linux PAM client and the Windows Desktop client is the imminent failure of the authentication verification, when the certificate that validates the daemons certificate, gets deleted or exchanged with an invalid or corrupt replacement. The result is that the user is immediately locked out and is not able to successfully log in again. Fixing the issue could be difficult without a second user, with permission to re-add the certificate. The same applies for an interrupted network connection, where a change in the configuration might be needed to patch the connection problem.

Aside from the advantages and disadvantages of this thesis compared to others, how does the future look like for second-factor authentication in general? Being now as mentioned before an essential part of people's daily life is it secure enough and how can there be or is might there be already a better model of authentication?

Bruce Schneier, the cryptography security specialist, mentioned in 2005 that "Two-factor authentication is not useless. It works for local log in, and it works within some corporate networks. But it won't work for remote authentication over the Internet. [sic]"[23]. With his prediction he shows his disapproval to the usage over the internet, yet shows the current use and acceptance, predominantly in online-banking, the opposite. It is also not anticipated that this tendency will drop, due to the surrounding encourage in the security dependent areas. Regarding the future development you can observe a slowly going trend towards biometric second-factor authentication, despite and mind in privacy concerns, with a tendency to use biometrics as an authentication system alone, i.e. already existing with smartphones [2].

However, for the lone and broader use of biometric authentication should the conscious recognition of a living person be more be developed.

Chapter 6

Conclusion

In this chapter we will discuss what has been achieved, as well we will discuss the future work that could be done to add or to improve upon the current progress if need be.

6.1 Summary

The main purpose of this thesis is the evaluation of Two-factor authentication with the aim to introduce and develop a new authentication factor by using Bluetooth Proximity detection. Aside from the theoretical part offers this thesis a daemon, which implements mentioned authentication method accessible as an interface. Accompanying the daemon there are three client implementations (Mozilla Firefox extension, Linux PAM module and Windows Desktop client) provided, where each utilizes the daemon's interface to extend an existing login scheme and establish a second-factor authentication. The developed authentication method queries for nearby Bluetooth devices and compares the result with known devices that have been previously chosen to function as authenticating token. Additionally, to the verification for known nearby Bluetooth device, a matching OTP or HMAC can be requested from the authenticating device, which essentially strengthens the authentication and requires an authentication tool, running on the Bluetooth device.

6.2 Future Outlook

The development of a new authentication method by using Bluetooth Proximity Detection and its application as additional authentication factor is an interesting area and definitely worth a look into.

The concept to decouple and centralize the authentication method on an accessible daemon is beneficial, since it removes the overhead to implement the authentication method for each client utilizing it. Continuing, it might be a good idea to introduce other authentication methods for clients due to the scalability of the daemon's API.

Speaking against the usage of Bluetooth Proximity as an additional authentication factor is the long time to query Bluetooth devices data, which creates a long authentication process and holds off the user login.

Chapter 7

Manual

In this chapter a closer look at provided daemon, clients and authentication tool in practice. The daemon, as well as the provided Windows Desktop Client, are executed on system with an Intel i5-4200U CPU and 8GB RAM running Windows 8.1 Pro (64Bit). The used Java Development as well Runtime Environment is version 1.8.0_92. The Linux PAM has been added to the login scheme of Kali Linux 2016.1, a Debian-based Linux distribution. The Firefox extension is added to the Mozilla Firefox web browser version 47.0.1 on the mentioned machine running Windows 8.1 Pro. The authentication tool runs on a HTC One (m7) with the operating system Android, version 6.0.1 ("Marshmallow").

7.1 Daemon

7.1.1 Prerequisites

Before the daemon can be started, the system which is going to run the daemon must specify Java in its system environment variables. On Linux systems it is required to have **HAL**[1] installed, which is used to retrieve the needed system variables that are needed for the OTP. Further, the daemon requires a "Java Keystore", which is the database holding the certificate that is used to secure and validate the daemon as an instance. The "Java Keystore" and the corresponding (for developing and test purposes, self-signed) certificate is created using "keytool", a program provided by Java. The "Java Keystore" is created by executing the "keytool" program from command-line with the following command: *keytool -genkey -alias alias -keyalg RSA -keystore keystore-name.jks -keysize 2048*. This command, where the alias of the certificate as well the keystore filename is user-specified, outputs the keystore file which must be placed in the same directory in which the daemon is. The certificate associated with the keystore is later

[1]Hardware Annotation Library (HAL)

required by the windows Desktop (section 7.4) and Linux PAM (see 7.3) client and therefore needs to be exported in PEM format using: *keytool -exportcert -alias alias -file certificate.pem -rfc -keystore keystore.jks.*

Since the daemon uses strong cryptography methods, the standard policy of the Java Cryptography Extension (JCE) must be updated with a newer and stronger Unlimited Strength Jurisdiction Policy[32]. This is achieved by replacing the "local_policy.jar" and "US_export_policy.jar" in the "lib\security" folder of Java with the stronger updated version.

7.1.2 Setup

The daemon is started from command-line using the command "java -jar 2fa-daemon.jar". When executing the daemon for the first time, an initialization setup starts which helps to configure and add the first authenticating Bluetooth token to the daemon:

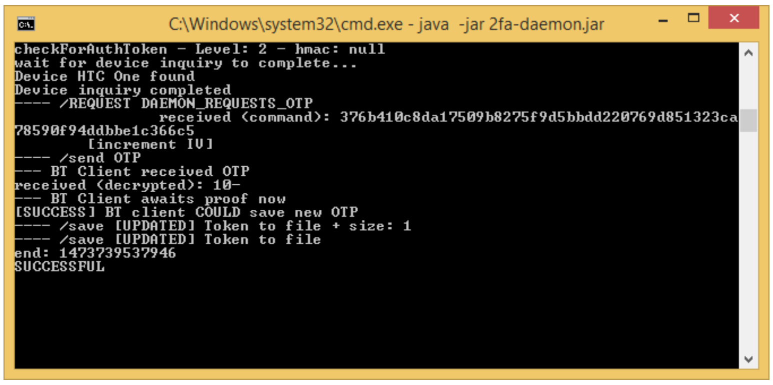
The setup starts by generating a public-private keypair, which is used to encrypt, beforehand sending OTP, and decrypt received OTP or HMAC by the authenticating Bluetooth token. The generated public key of the keypair, which is stored in the same directory as the daemon must be forwarded to the device running the authentication tool (see 7.5) when a security level of 2 or 3 is intended. The authentication tool requires the daemon public key to verify received and send encrypted OTP to the daemon. After the keypair generation, the daemon queries for the first time for nearby Bluetooth devices. After successful device discovery, the user must select a device, as well as the desired authentication security level (see chapter 3.2). By selecting a Bluetooth device, the daemon saves the encrypted hash of the device's Bluetooth address, along with further information like its public key and the sent OTP, when the chosen security level is higher level 1.

Once at least one authentication token is added, the initialization of the daemons API starts by setting HTTPS as only accepted connection method. After securing the connection, different API routes are loaded. The daemon is now ready to receive API requests.

The daemon adminstrator can add additional Bluetooth device through the daemon's API route /addNewDevice. By calling this route in a web browser the daemon Bluetooth discovery gets triggered and the same device selection procedure during the setup is ecxecuted.

7.1.3 Authentication

When the daemon receives an authentication request, it is visible in the command-line while running the daemon. It then checks once again for nearby Bluetooth devices, querying for a device which has been previously chosen to act as the authentication token. When the second or third security level is demanded, the daemon requests the OTP (or HMAC) by a found and authorized Bluetooth token (seen in figure 7.1).



```
checkForAuthToken - Level: 2 - hmac: null
wait for device inquiry to complete...
Device HTC One found
Device inquiry completed
---- /REQUEST DAEMON_REQUESTS_OTP
                received (command): 376b410c8da17509b8275f9d5bbdd220769d851323ca
78590f94ddbbe1c366c5
        [increment IV]
---- /send OTP
--- BT Client received OTP
received (decrypted): 10-
--- BT Client awaits proof now
[SUCCESS] BT client COULD save new OTP
---- /save [UPDATED] Token to file + size: 1
---- /save [UPDATED] Token to file
end: 1473739537946
SUCCESSFUL
```

Figure 7.1: Screenshot of the daemon while performing the authentication using security level 2

On a positive response, a new OTP is sent to the Bluetooth token. Afterward, the daemon is ready to receive further authentication requests. If the query for authorized Bluetooth is unsuccessful, the daemon returns a negative authorization response to the client. The same occurs when it receives an invalid OTP by the authentication tool.

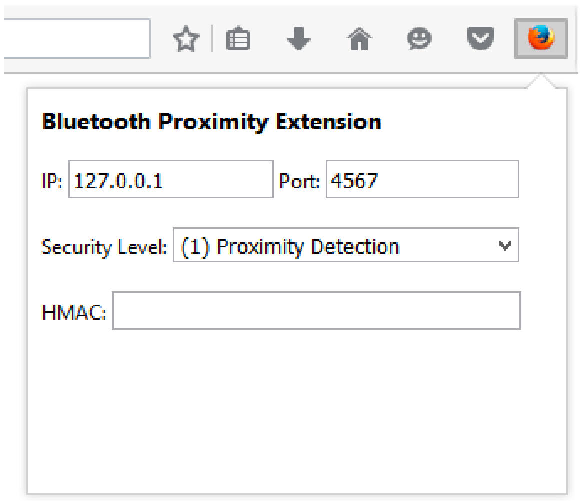
7.2 Firefox extension

The Mozilla Firefox extension is installed by dragging and dropping the “2fa-firefox-extension.xpi” on a running Mozilla Firefox instance and by confirming its installation.

In the next step, the daemon’s parameters like its IP address and port, as well as the desired authentication level must be specified in the extension’s settings. The settings are accessible from the shortcut in the top right extension bar (see figure 7.2). On security level 3, the last input field of the settings form must be filled with the keyphrase of the HMAC.
Subsequently, the daemon’s certificate must be added. This is achieved by visiting its ’/hello’ route and confirm to trust the website, which adds the certificate to Mozilla Firefox’s certificate manager.

Once all the steps are completed, sends the extension automatically, when a login scheme is found, the authentication requests to the daemon.

Further, the password input field gets disabled, which intervenes the website’s login and the Firefox extension awaits the authentication response by the deamon. On a positive response the input field is again enabled so that the user can proceed with the login authentication. Otherwise, on a negative response, the input field stays disabled.



Bluetooth Proximity Extension

IP: 127.0.0.1 Port: 4567

Security Level: (1) Proximity Detection

HMAC:

Figure 7.2: Screenshot of the extension's interface

7.3 Linux PAM client

A prerequisite and requirement, which needs to be installed before the Linux PAM client, is cURL[2] with OpenSSL for the Linux distribution since it is utilized in the Linux PAM client.

The Linux PAM client is installed using the provided *makefile* which is located along the source files. The user needs to open a terminal session, change into directory of mentioned makefile and execute the shell command 'make' command. Followed by 'make install'. The script automatically install the PAM client in the set default location for PAM modules for Debian-based Linux distributions (*libsecurity*. Therefore the installation path might need to be adjusted, depending on the Linux distribution. This is accomplished by modifying the parameter 'PAM_DST' in the makefile.

To apply the Linux PAM client to the Linux login scheme, the user needs to modify and add a line to the configuration file *common-auth* located in *etcpam.d*. For example, an added line for an authentication with security level 2 would look like as following:

auth required /lib/security/bt-pam.so
/cert/certificate.pam 192.168.1.3:3456 2

The example adds this Linux PAM client as **required auth**entication module, indicates it's installation path and informs it about the requested security level (here: 2) and daemons certificate as well as location (IP address

[2]cURL - https://curl.haxx.se/download.html

and port). The sixth parameter, the keyphrase for the HMAC, is in the example not set, since it's only needed with the 3rd security level.

Once the configuration file is saved, the Linux PAM client is already applied and get's invoked with next login at the Linux login scheme.

7.4 Windows Desktop client

The Windows Desktop is, like the daemon, implemented using Java and needs to be be started from the command line using the command "java -jar 2fa-windows-client.jar". On the first startup, unless it has been specified differently later, starts the Windows Desktop client with maximized window and its graphical user interface (GUI) which is presented in figure 7.3).

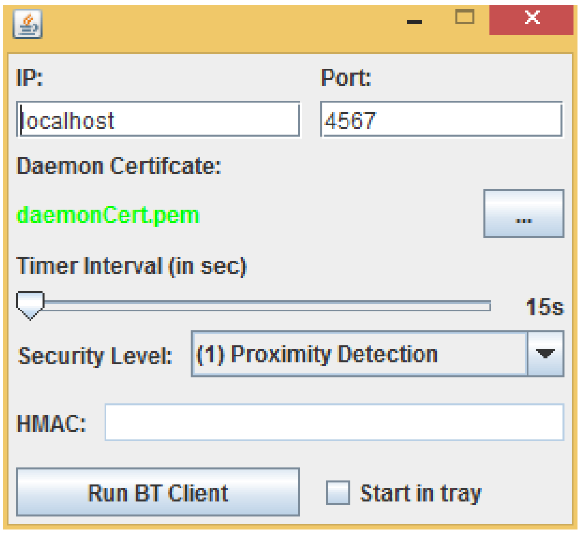

IP: localhost
Port: 4567
Daemon Certifcate: daemonCert.pem
Timer Interval (in sec) 15s
Security Level: (1) Proximity Detection
HMAC:

Figure 7.3: Graphical user interface of the Windows Desktop client

Before the client can secure the Windows session, the required parameters, like the IP address and port of the daemon, must be specified in designated input fields of the GUI. Further, the certificate, which has been obtained earlier, in section 7.1.1, must be located and set in the client. In the next step, the security level, as well as the time interval between the authentication requests, have to be selected. The time interval slider offers the user a time frame in 15-second steps, starting from 15 seconds to one hour.

Once all these settings are set, the Windows Desktop client can be started. From now on, the Windows Desktop client sends the authorization requests to the daemon and immediately invokes the Windows lock screen when an authorized Bluetooth token could not be found.

7.5 Authentication tool

The authentication tool of this thesis is provided as "2fa-authentication-tool.apk" and needs to be installed as third-party Android application. Conclusively the user must allow "unknown sources" in the security settings in order to allow the installation from the smartphone's internal or external storage. Further, the authentication tool requires the permission to access the smartphone's storage to query and selected the daemon's public key (see 7.1.2). Therefore, the app needs to have app permission"storage" granted. The permission can be set by enabling it in the point 'permissions' in the 'App info' settings of Android.

With the start of the application the authentication tool's interface is presented. Like with the earlier present sample clients, the authentication tools needs to be configured. First the daemon's certificate must be forwarded to device and with the ease of the application interface added and and selected. Secondly, Bluetooth must be enabled, if not already turned on, which is also possible within the app. In the final step, the user must when the authentication request demands it (security level equals three) set the HMAC keyphrase.

Once all these steps have been fulfilled, the authentication background service can be started and the smartphone is ready to receive the first OTP. The feature of the background service is that it continues to run and is therfore able to receive further OTP (or HMAC) even when the application is closed. Certainly, it is possible to turn off the background service by disabling it using the authentication tool's interface.

From now on, the authentication tool, allows to enhance strenghtens the Bluetooth proximity authentication by exchanging tokens in the format of OTP and HMAC with the daemon.

References

Literature

[1] Fadi Aloul, Syed Zahidi, and Wasim El-Hajj. “Multi factor authentication using mobile phones”. *International Journal of Mathematics and Computer Science* 4.2 (2009), pp. 65–80 (cit. on pp. 3, 6, 7, 9).

[2] Rasekhar Bhagavatula et al. “Biometric authentication on iphone and android: Usability, perceptions, and influences on adoption”. In: *In Proc. USEC*. 2015 (cit. on p. 44).

[3] Cure53 et al. *Pentest-Report cURL 08.2016*. Tech. rep. Aug. 2016. URL: https://cure53.de/pentest-report_curl.pdf (cit. on p. 39).

[4] Alexei Czeskis et al. “Strengthening User Authentication Through Opportunistic Cryptographic Identity Assertions”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 404–414. URL: http://doi.acm.org/10.1145/2382196.2382240 (cit. on pp. 42, 43).

[5] M. H. Eldefrawy, M. K. Khan, and H. Elkamchouchi. “The Use of Two Authentication Factors to Enhance the Security of Mass Storage Devices”. In: *Information Technology: New Generations (ITNG), 2014 11th International Conference on*. Apr. 2014, pp. 196–200 (cit. on p. 43).

[6] How-To Geek. *ATM Skimmers Explained: How to Protect Your ATM Card*. image. URL: http://cdn5.howtogeek.com/wp-content/uploads/2014/07/automated-transaction-machine-or-atm.jpg (cit. on p. 7).

[7] Neil Haller et al. *A One-Time Password System*. RFC 2289. http://www.rfc-editor.org/rfc/rfc2289.txt. RFC Editor, Feb. 1998. URL: http://www.rfc-editor.org/rfc/rfc2289.txt (cit. on p. 13).

[8] Steffen Hallsteinsen, Ivar Jorstad, et al. “Using the mobile phone as a security token for unified authentication”. In: *Systems and Networks Communications, 2007. ICSNC 2007. Second International Conference on*. IEEE. 2007, pp. 68–68 (cit. on pp. 4, 8, 24).

[9] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.1a: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Wireless Personal Area Networks (WPAN)". *IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002)* (June 2005), pp. 1–700 (cit. on p. 36).

[10] RSA Security Inc. *RSA SecurID 800 Authenticator*. datasheet. An optional note. 2008 (cit. on p. 6).

[11] Kingpin. *Attacks on and Countermeasures for USB Hardware Token Devices*. 2000 (cit. on p. 11).

[12] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. http://www.rfc-editor.org/rfc/rfc2104.txt. RFC Editor, Feb. 1997. URL: http://www.rfc-editor.org/rfc/rfc2104.txt (cit. on p. 14).

[13] M. Lafkih et al. "Application of new alteration attack on biometric authentication systems" (Nov. 2015), pp. 1–5 (cit. on p. 12).

[14] J. Lang, B. Rajagopalan, and D. Papadimitriou. *Generalized Multi-Protocol Label Switching (GMPLS) Recovery Functional Specification*. RFC 4426. RFC Editor, Mar. 2006 (cit. on p. 13).

[15] J. C. Liou and S. Bhashyam. "A feasible and cost effective two-factor authentication for online transactions" (June 2010), pp. 47–51 (cit. on pp. 9, 12).

[16] Josh Mandel, Austin Roach, and Keith Winstein. *MIT Proximity card vulnerabilities*. Tech. rep. (cit. on p. 12).

[17] D. M'Raihi et al. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. http://www.rfc-editor.org/rfc/rfc6238.txt. RFC Editor, May 2011. URL: http://www.rfc-editor.org/rfc/rfc6238.txt (cit. on p. 13).

[18] PayPal. *PayPal TAN Generator*. image. URL: http://presse.ebay.de/sites/default/files/101101_1.jpg (cit. on p. 8).

[19] Floris Lambrechts Peter Hernberg. *User Authentication HOWTO*. May 2000. URL: http://tldp.org/HOWTO/User-Authentication-HOWTO/x115.html (cit. on p. 26).

[20] Thanasis Petsas et al. "Two-factor Authentication: Is the World Ready?: Quantifying 2FA Adoption". EuroSec '15 (2015), 4:1–4:7 (cit. on p. 11).

[21] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". *Communications of the ACM* 21.2 (1978), pp. 120–126 (cit. on p. 14).

[22] Nikita Saple et al. “Securing Computer Folders using Bluetooth and Rijndael Encryption”. *International Journal of Current Engineering and Technology* 5.1 (2015), pp. 397–400 (cit. on p. 10).

[23] Bruce Schneier. “Two-factor Authentication: Too Little, Too Late”. *Commun. ACM* 48.4 (Apr. 2005), pp. 136–. URL: http://doi.acm.org/10.1145/1053291.1053327 (cit. on pp. 41, 44).

[24] Aladdin - Securing the global village. “The Security Advantages of Hardware Tokens over Software Tokens for PKI Applications” (May 2006) (cit. on p. 12).

[25] Yubico. *YubiKey NEO*. image. URL: https://www.yubico.com/wp-content/uploads/2012/09/YubiKey-NEO-on-Phone.jpg (cit. on p. 11).

Online sources

[26] Thorsten Kukuk Andrew G. Morgan. Aug. 2010. URL: http://www.linux-pam.org/Linux-PAM-html/mwg-expected-of-module-overview.html (cit. on pp. 29, 30, 38).

[27] Google. 2016. URL: https://support.google.com/nexus/answer/6093922?hl=en (cit. on p. 14).

[28] Google. URL: https://www.google.com/landing/2step/ (cit. on p. 9).

[29] Ted Hersey. 2016. URL: http://all.net/CID/Attack/papers/Replay.html (cit. on pp. 12, 13).

[30] Scott Lindley. Nov. 2015. URL: http://www.mnec.com/article/securing_campus_contactless_card_based_access_control_systems (cit. on p. 12).

[31] Mozilla Developer Network and individual contributors. June 2016. URL: https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Content_Scripts (cit. on p. 25).

[32] Oracle Technology Network. URL: http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html (cit. on p. 49).

[33] Oracle. URL: http://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html (cit. on p. 37).

[34] Margaret Rouse. May 2015. URL: http://searchsecurity.techtarget.com/definition/multifactor-authentication-MFA (cit. on p. 5).

[35] Margaret Rouse. URL: http://searchsecurity.techtarget.com/definition/one-time-password-OTP (cit. on p. 13).

[36] Bundesamt fuer Sicherheit in der Informationstechnik. 2013. URL: https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/_content/m/m04/m04133.html (cit. on p. 5).

[37] Chester Wisniewski. Jan. 2014. URL: https://nakedsecurity.sophos.com/2014/01/31/the-power-of-two-all-you-need-to-know-about-2fa/ (cit. on p. 4).