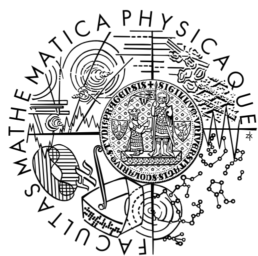


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Karolína Zenknerová

Analysis of Cryptographic Protocols in Mobile Banking Applications (part A)

Department of Algebra

Supervisor of the master thesis: Mgr. Jakub Töpfer
Study programme: Mathematics for Information Technologies

Prague 2025

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I dedicate this work to my family, who supported me throughout my studies and made it possible to get this far, and to Ondra for his support, positive attitude and his help with my strange technical problems that defy logic. I would also like to thank to my supervisor for his patience and guidance. I am very grateful for everything I have learned while working on this thesis.
Long live Ruprechtice!

Title: Analysis of Cryptographic Protocols in Mobile Banking Applications (part A)

Author: Karolína Zenknerová

Department: Department of Algebra

Supervisor: Mgr. Jakub Töpfer, Department of Algebra

Abstract: The thesis analyses cryptographic protocols in mobile banking applications of three banks operating in the Czech Republic. Besides TLS, most of mobile banking applications implement their own cryptographic protocols over TLS. The aim of this thesis is to reverse engineer applications, describe discovered cryptogaphic protocols, identify potential weaknesses and suggest possible improvements. This thesis describes cryptographic protocols in applications MyAir by Airbank, Partners and KB+ by Komerční banka. Emphasis is placed primarily on login and payment protocols. At the end of each chapter on application there is a summary of the key results from a high-level perspective. In order to cover more mobile banking applications, there is also a part B that is examined by another student.

Keywords: cryptographic protocols, analysis of cryptographic protocols, design of cryptographic protocols, applied cryptography, reverse engineering of mobile applications, mobile application security, mobile banking

Název práce: Analýza kryptografických protokolů v aplikacích pro smartbanking (část A)

Autor: Karolína Zenknerová

Katedra: Katedra Algebry

Vedoucí diplomové práce: Mgr. Jakub Töpfer, Katedra Algebry

Abstrakt: Tato práce analyzuje kryptografické protokoly v aplikacích mobilního bankovnictví tří bank působících v České republice. Kromě TLS většina aplikací mobilního bankovnictví implementuje své vlastní kryptografické protokoly nad TLS. Cíl této práce je provést reverzní inženýrství těchto aplikací, popsat nalezené kryptografické protokoly, identifikovat možné nedostatky a navrhnout možná vylepšení. Tato práce popisuje kryptografické protokoly v aplikacích MyAir od Airbank, Partners and KB+ od Komerční banky. Důraz je kladen zejména na přihlášení a platbu. Na konci každé kapitoly o aplikaci jsou shrnuty klíčové výsledky z high-level pohledu. Aby bylo možné pokrýt více aplikací mobilního bankovnictví, je zadána i část B, která je řešena jiným studentem.

Klíčová slova: kryptografické protokoly, analýza kryptografických protokolů, návrh kryptografických protokolů, aplikovaná kryptografie, reverzní inženýrství mobilních aplikací, zabezpečení mobilních aplikací, smartbanking

Contents

Introduction **7**

1 Practical background **8**
- 1.1 General background . . . 8
  - 1.1.1 TLS . . . 8
  - 1.1.2 HTTP and HTTPS . . . 8
- 1.2 JSON Web Token (JWT) . . . 9
- 1.3 Setup and configuration . . . 11
  - 1.3.1 Main idea . . . 11
  - 1.3.2 Software tools . . . 11
  - 1.3.3 Setup . . . 13

2 Theoretical background **14**
- 2.1 RSA . . . 14
  - 2.1.1 ECB . . . 14
  - 2.1.2 PKCS#1 padding . . . 15
- 2.2 AES . . . 16
  - 2.2.1 PKCS#7 padding . . . 16
  - 2.2.2 CBC mode . . . 16
- 2.3 SHA256 . . . 17
- 2.4 HMAC . . . 17
- 2.5 SRP protocol . . . 18
- 2.6 OCRA . . . 21
- 2.7 PBKDF2 . . . 23
- 2.8 ECDH . . . 24
- 2.9 ECDSA . . . 26
- 2.10 ECIES . . . 27
- 2.11 ANSI X9.63 KDF . . . 29
- 2.12 OAuth 2.0 PKCE . . . 30

3 MyAir **32**
- 3.1 Protocol description . . . 32
  - 3.1.1 Overview . . . 32
  - 3.1.2 Notation . . . 34
  - 3.1.3 HMAC . . . 34
  - 3.1.4 Login . . . 35
  - 3.1.5 Encrypt/decrypt by session secret . . . 40
  - 3.1.6 Authorization . . . 42
  - 3.1.7 Registration . . . 65
  - 3.1.8 Change of password (PIN, master_secret) . . . 84
- 3.2 Practical part . . . 94
  - 3.2.1 Cryptographic libraries and sources . . . 94
  - 3.2.2 Setup . . . 95
  - 3.2.3 Reversing . . . 96
  - 3.2.4 Authorization . . . 96

3.3 Conclusion 103

4 Partners **106**
4.1 Protocol description 106
4.1.1 Activation Status 106
4.1.2 Login 112
4.1.3 Transaction 122
4.1.4 Activation 136
4.2 Practical part 141
4.2.1 Cryptographic libraries and sources 141
4.2.2 Setup 142
4.2.3 Reversing 144
4.3 Conclusion 150

5 KB+ **152**
5.1 Protocol description 152
5.1.1 Login 152
5.1.2 Transaction 166
5.2 Practical part 167
5.2.1 Cryptographic libraries and sources 167
5.2.2 Setup 167
5.2.3 Reversing 167
5.3 Conclusion 169

Conclusion **171**

A Attachments **174**
A.1 Our implementation 174
A.2 Scripts for bypassing protections 174
A.3 Example outputs and hooking scripts 174

Introduction

Most of us use mobile banking applications on daily basis for sensitive operations such as payments. Naturally, these operations have to be protected. The connection between the client (mobile application) and the server of the bank is protected by TLS which provides integrity, authentication and enryption using cryptography. Besides TLS, most of mobile banking applications implements their own cryptographic protocols over TLS in order to provide more security. These protocols are not publicly available, since they are part of proprietray mobile applications. In this thesis we will reverse theree mobile banking applications and examine protocols they use.

We want to emphasize that the communication is protected by TLS and thus possible weaknesses in cryptographic protocols do not pose a threat to the security of the application. We focus mainly on cryptographic protocols that we analyse from the theoretical point of view. On the other hand, it is also necessary to solve practical problems associated with the reverse engineering. Most of mobile banking applications contain protections that make reverse engineering of used cryptographic protocols more difficult. Before starting the core part of work we have to overcome these pretections. The practical part of the work consists of the static and dynamic analysis. After the practical reverse engineering it is possible to describe and evaluate cryptographic protocols and algorithms from the theoretical point of view, which we do in this thesis.

The aim of this thesis is to examine three applications, describe the cryptographic protocols they use and discover potential weaknesses. This thesis does not focus on general security of applications. We have chosen MyAir by Airbank, Partners and KB+ by Komerční banka.

Since there is usually problem with protections against reverse engineering implemented by banks, we note that we also tried to revese engineer two more applications which turned out to be too complicated for this thesis. These applications had protection against dynamic analysis which we describe in the first chapter. We provide more details about these applications in the last chapter. Due to the limited time we had to describe KB+ without going into greater detail.

The first chapter describes the practical setup and background. The second chapter desribes the theoretical background - algorithms that appear in following chapters. The following three chapters are about specific applications. The chapter about KB+ provides a less detailed explanation due to time limitations. The last chapter concludes the thesis and provides a comparison of the found protocols.

1 Practical background

1.1 General background

In this section we briefly describe general protocols and concepts that we will see in this thesis.

1.1.1 TLS

TLS stands for Transport Layer Security. It is an encryption layer, that provides encryption, integrity and authentication between two communicating parties. It is commonly used in combination with other protocols, for instance with HTTP which is described in the next subsection. In general it is recommended to use versions TLS 1.2 or TLS 1.3 ([1], [2]) with appropriate configuration. TLS 1.2 is described in RFC 5246 [3] and TLS 1.3 is described in RFC 8446 [4].

TLS connection has two steps: 1) a handshake, where two parties agree on the protocol and cryptographic settings, authenticate using certificates and change encryption keys and 2) the encrypted data exchange.

The authentication uses X.509 certificate [5], [6], it is usual that the server side has to be authenticated, whereas the authentication of the client is not used very often in practice.

X.509 certificate is a standard describing public key certificates. To put it simply, the certificate links the identity of a client or a server with its public key. The certificate should be signed by some trusted certificate authority (CA) which confirms that the client or the server really owns the private key. This public key then can be used for authentication.

In this thesis we examine cryptographic layers developed on the top of the TLS tunnel to enhance overall security and not rely only on TLS.

1.1.2 HTTP and HTTPS

Applications in this thesis use HTTPS when communicationg with the server. That means HTTP (Hypertext Transfer Protocol) with TLS layer. For more information about HTTP see [7], [8], [9]. We overcome this problem using Burp proxy as described below. In the thesis we will capture communication between the server and the client which consists of request and responses. HTTP requests have the following structure described in [7]:

- The first line consists of request method, target (uri) and HTTP version.
- Headers (zero or more), which consist of a name of header and its value.
- Then there is an empty line, after which follows an optional message body (there also does not have to be any).

Example of the request with no message body:

```
GET /discountProgram/v4/smartOverview HTTP/1.1
Host: mb.myair.cz
Device-Type: mobile android
X-Process-Id: prod-MA_ANDROID-20250402104414-618159
Device-Id: FD0D89A3D052FFC92D940BF5AEAE4067A66A5EBD
           2E4E371FE16DFC50300DF6D0
Authorization: Bearer SaERPEfUZckZlLD1mZ3igHwAAAAAA
               B0K_nwAAAGV9bCzjXyocF2ivJYMdIwowZ9lp
               w8zXCIrrvNPDcfoWrI1fF9oeA
User-Agent: Air Bank/10.1.1 (phone; android; 31;
                             Pixel 3 XL; 1440x2621)
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Responses have the same structure but the first line which is a status line. In this line the server gives HTTP version, status code and reason phase.
Example of the response:

```
HTTP/1.1 200 OK
Date: Wed, 02 Apr 2025 08:44:14 GMT
Content-Type: application/json;charset=UTF-8
Content-Length: 90
Expires: 0
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
X-Cnection: close
Set-Cookie: HCI_PERSISTENCE=2198045962.20480.0000;
            expires=Wed, 02-Apr-2025 09:24:14 GMT;
            path=/; Httponly; Secure
J-Session-ID: -
http-host: mb.myair.cz
Real-Clock: 2025-04-02:10:44:14.255 +0200
Set-Cookie: TS01ffcd1c=01212c4b7f417104a31e4379685a
            af235066360593b1ed18e30c62d7dcdd6b9f7cd
            ffe445090ecd654e5dad98e9f7d170d44ad4ec1
            228f9bc3306957d26326529a53965a40;
            Path=/; Domain=.mb.myair.cz;
Vary: Accept-Encoding

{"partlyFilledProgressbarGiftsCount":0,
"filledProgressbarGiftsCount":0,
"newOffersCount":0}
```

1.2 JSON Web Token (JWT)

JWT is a standard describing the format for claims transfered between two parties. The structure of this format contains JSON objects with claims. JWT is

described in RFC 7519 [10]. For more information we recommend a website with JWT Debugger [11] where is a simple explanation of JWT tokens.

In general, the structure consists of:

- Header - contains claims about cryptographic operations that are applied on JWT, that is, an algorithm that is used to sign the token and additional properties.
- Payload - contains claims. According to [11] claims can be registered, public or private. Registered claims are recommended (but not mandatory), public claims are optional (but standard) and private claims are custom. Registered and public claims are registered in the IANA "JSON Web Token Claims" registry.
- Signature - the signature of the header and the payload using the signature algorithm specified in the header.

  Both, header and payload are encoded by base64 and connected using "." before they are proccessed by the signature algorithm. The signature guarantees the integrity. It can also ensure the authenticity.

Example

The following example of JWT was taken from [12].

- **Header**

```
{
    "alg": "HS256",
    "typ": "JWT"
  }
```

- **Payload**

```
{
    "sub": "1234567890",
    "name": "John Doe",
    "admin": true,
    "iat": 1516239022
  }
```

- **Signature secret** - an example secret that is used for signature (UTF-8)


```
a-string-secret-at-least-256-bits-long
```


- **JWT** - the token where Header, Payload and Signature are encoded by base64 and divided by "."


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkw
IiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzO
TAyMn0.KMUFsIDTnFmyG3nMiGM6H9FNFUROf3wh7SmqJp-QV30
```

1.3 Setup and configuration

In this section we will describe how we prepared for the thesis and which tools we used.

1.3.1 Main idea

In this subsection we try to give an idea behind the setup. We will provide more details in following subsections.

Firstly we need to have the application. That is, we have to install the application and open an account at the bank.

To analyze the app we need to capture the communication between the mobile application client and the server of the bank. The captured communication helps us to understand what is happening in the background when we run the application. The communication between the client and the server is performed using HTTPS. In order to read messages we will intercept the traffic using proxy.

Firstly, we need to route the traffic between the mobile and the bank through the computer. For this purpose we create an access point using our computer. The mobile with the application is then connected to this access point and the computer is connected to the Interned via USB cable and the second mobile using USB tethering. Then we can capture HTTPS traffic and read it as HTTP using Burp proxy as described below in 1.3.2.

Another way to analyze the application is static analysis which is done by examining the application code. We can obtain app files of installed applications. The file base.apk then can be decompiled into java code. The code of course differs from the original code and it is less readable. In combination with the captured communication it can give us an idea of what is happening. We note that native libraries can be also decompiled into pseudocode using different tool.

A different way to analyze the application is a dynamic analysis using Frida 1.3.2 that allows us to hook functions, trace them, print their input and output parameters or even rewrite them while the application is running.

For more information we recommend to see the website of OWASP about mobile application security testing [13].

1.3.2 Software tools

In this section we describe some software tools we used.

adb

Android debug bridge (adb) [14] is a commandline tool that enables the communication between computer and the mobile device with Android. It can be installed as a part of Android studio. To use adb with the device connected via USB, the user has to enable USB debugging on the device.

Frida

Frida is a toolkit for dynamic code instrumentation. That means it allows runtime code manipulation. It also lets us to insert JavaScript snippets or our

own library into native applications. Frida is free, its code can be found on Github [15]. There is a website [16], where is a documentation, examples and tutorials. We note that when using Frida for mobile apps analysis, the frida server has to be running on a mobile device.

We used Frida when inspecting applications mainly for dynamically tracing functions calls, that means hooking them and when they were called, printing out their input and output. This helped us to analyze the code. There is a special tool for tracing packages, methods etc. called frida-trace [17]. For tracing native functions we used Frida with Interceptor [18].

Frida can be also used to bypass protections like for example Certificate Pinning, which we will see in 3.2.2. Since bypassing protections is a quite common problem when reverse-engineering, many scripts can be found on the website Frida CodeShare [19]. Bypassing protection typically involves overwriting of some called functions.

Burp Suite

Burp Suite [20] is a proprietary security testing software, which focuses on web applications. It contains many various tools that can be used for testing, however, for our analysis we used only a few of them.

We used Burp proxy which creates a proxy server between the client application and the server. It allows us to capture and analyze the traffic. If needed we can also modify it.

Applications are usually using TLS, which means that the communication between the server and the client is encrypted. In order to inspect or modify the traffic the Burp proxy has to capture it, decrypt it and then encrypt is again before sending. We can assume that the application will not trust the proxy server, because the certificate generated by the Burp proxy will not be signed by any trusted CA (certification authority). To overcome this problem, the Burp proxy can generate a CA certificate unique for every installation that can be downloaded and then installed into a mobile device as a trusted CA certificate. Then the proxy can establish TLS connection with the client and with the server separately and we can see the decrypted communication. The client obtains the certificate by Burp proxy, that is signed by Burp CA, which he trusts.

We note that we used free Community Edition of Burp Suite for our thesis.

jadx

jadx is a Java decompiler. It does not have its own website, but its code can be found on Github [21]. It creates a readable java code from Android Dex and Apk files.

We used it to obtain the code of analyzed applications. The main file of an application can be found as base.apk on the mobile device. The decompiler then produces the code, which we use to analyze the cryptography in applications.

The code obtained from jadx is not the same as the original code, it typically does not preserve names of all packages, classes and methods and it is more complicated to read. It allows us to browse the code, search for specific functions, classes etc. We can also rename classes and methods if we manage to guess what they do and write comments into the code to make it more clear.

Ghidra

Native libraries, that is binary files for ARM architecture, cannot be easily decompiled by jadx. We used Ghidra instead.

Ghidra is a software reverse engineering tool developed by NSA. The code can be found on Github [22]. It also has a website [23]. In addition to assembly instructions, it also translates the instructions into C pseudocode. Although the decompiled pseudocode returned by Ghidra is sometimes not very clear, it is much easier to read than assembly instructions

It also provides a debugger which we did not use. Ghidra alows us to browse created code, search for specific names of functions or classes and rename variables and functions.

ProxyDroid

ProxyDroid is an opensource mobile application which enables you to set and manage proxy. We use it to send our traffic through the Burp proxy.

1.3.3 Setup

The main idea of setup was described above. This section only provides more details.

In the thesis we used rooted device for running analyzed applications.

The access point was created and configured using comand-line tool nmcli. We set the ip address to be 192.168.3.1/24.

We installed all above mentioned software - Burp Suite, Ghidra, jadx, Frida, adb - on the computer and we installed ProxyDroid on the testing mobile phone. We also copied the frida server binary to the mobile, therefore we could run it.

We installed the Burp CA certificate into mobile as described above. Then we added new proxy listener, thus the Burp proxy could listen on 192.168.3.1 on port 12345. Then we set set host in ProxyDroid to 192.168.3.1, TCP port 12345. Therefore the traffic between the mobile application and the bank server could be rerouted through the Burp proxy.

We used adb to run frida server and to obtain all application files that were interesting for us. For instance base.apk or files in shared preferences. We examined the decompiled code of applications and native libraries using jadx and Ghidra. Both of them allow us to rename variables and functions and to save our progress.

The dynamic analysis or some manipulation with the running application, like for example bypassing some protection, can be done using Frida. Consequently when analyzing the application, we ususally run it to perform the dynamic analysis or in addition to the traffic interception. We ususally use commandline options -U, which stands for USB and -F for the application on the foreground or -f with the name of the application. We note that the testing mobile is usually connected via USB, because of adb and frida server.

To summarize it, when starting our dynamic analysis, we have to connect to our AP, start Burp proxy with the right listener, enable proxy with ProxyDroid and usually bypass some protection using Frida. To analyze the code statically we need to use jadx or Ghidra.

2 Theoretical background

This chapter contains a brief description of theoretical cryptographic concepts that are used in following chapters.

2.1 RSA

RSA is a well known public-key cryptosystem, therefore in this paragraph we provide only basic information. For more information, we refer to the section 9.3 in [24]. The following description comes from the same source.

Preparation:

len	...	chosen even integer,
N	...	$N = p \cdot q$, where p and q are randomly chosen primes, $p \neq q$, of the length $\frac{\text{len}}{2}$.
e	...	randomly chosen integer such that $\gcd(e, \varphi(N) = 1$, where φ is Euler's totient function. In [24] they also note that e can be chosen specifically. Then p and q are chosen to satisfy the condition. Usually, $e = 65537$.
d	...	integer such that $d \cdot e \equiv 1 \bmod (p-1) \cdot (q-1)$.

RSA parameters are:

$$\begin{aligned} \text{public key} &= (e, N) \\ \text{private key} &= (d, p, q). \end{aligned}$$

Encryption of the message M (M is an integer, $1 \leq M \leq N - 1$):

$$C = M^e \bmod N.$$

Decryption of the ciphertext:

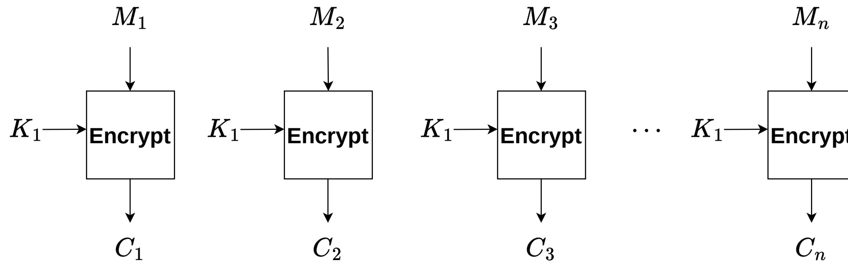
$$M = C^d \bmod N.$$

2.1.1 ECB

ECB (The Electronic Codebook mode) is a mode of encryption, where the message M is divided into blocks $M_1, M_2, \ldots, M_n$. Every block of the message is then processed independently using the same key. ECB mode is defined for block ciphers, see [25], section 4.1.1.

We will see this mode used as a mode for RSA. It is used only in Java and it does not make sense in general, since RSA encryption operates directly on individual blocks. In RSA encryption scheme the message cannot be longer than N, thus the length of the 'block' in this case would be N. We note that the RSA, of course, uses the public key for encryption and the private key for decrytion since it is not a symmetric block cipher.

For completeness we provide the scheme ECB for block ciphers:



The ciphertext is $C = C_1||C_2||C_3||\cdots||C_n$. The decryption is done similarly.

2.1.2 PKCS#1 padding

In symmetric ciphers, padding is used when the algorithm requires some specific block size. In RSA it is used for security reasons. For example we want the output to differ when encrypting two identical plaintexts. By using PKCS#1 padding in RSA algorithm we obtain an input with length equal to the length the modulus N in bytes.

In this thesis we will see PKCS#1 v1.5, the description can be found in [24] in the subsection 9.3.2. There are newer verisons of the padding algorithm. The current version is 2.2 which can be found in RFC 8017 [26].

There is a simple description of a version 1.5 that was taken from [24].

Algorithm 1 PKCS#1v1.5 padding

Let us denote the concatenation by $||$.
INPUT: modulus N that has the length k bytes, message M of lenght at most $k - 11$ bytes
OUTPUT: padded message M

1: Generate random string P of nonzero bytes such that the sum of lengths of P and M is $k - 3$ bytes.
2: **return** padded message $00||02||P||00||M$.

The padding is randomized, thus the output of the padding operation is every time different. PKCS#1 v1.5 is known to have security weaknesses, see for example the paper by Bleichenbacher [27]. The OAEP padding, see the section 9.3.8 in [24], is one of recommended options instead, see [28].

2.2 AES

AES (Advanced Encryption Standard) is a standard block cipher that encrypts a 128-bit block of a plaintext. It offers three key length sizes - 128, 192 and 256 bits. FIPS 197 is a NIST standard that describes it. Since the cipher is frequently used, we consider it a general knowledge and we provide only source without the detailed explanation. For further information see Chapter 3 in [25] or the standard FIPS 197.

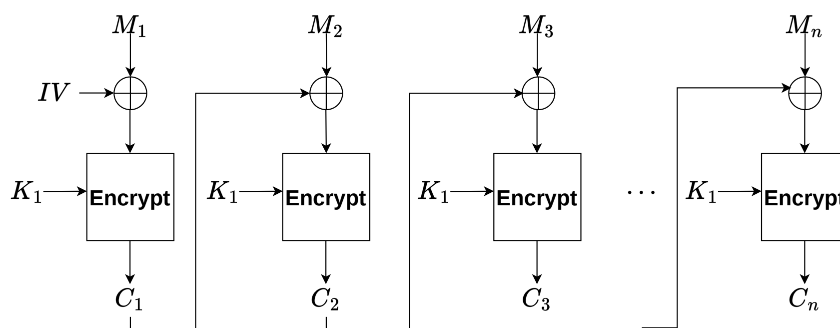
2.2.1 PKCS#7 padding

As before, the padding is used to obtain the required length of the input. This padding is used when the multiple of k bytes is needed. For example when using AES, which takes blocks of the length 128 bits, we usually require the lenght of input to be equal to a multiple of 16 bytes (=128 bits). PKCS#7 padding is described in RFC 2315 [29].

2.2.2 CBC mode

The CBC (The Cipher Block Chaining Mode) is a mode of encryption for block ciphers. It is described in NIST SP 800-38A [30]. For an ilustration, we provide a scheme of the encryption:

$M = M_1||M_2||M_3||\cdots||M_n$



The ciphertext is $C = C_1||C_2||C_3||\cdots||C_n$. The decryption is done analogically. We note that in addition to the input message and the key, we need an IV (inicialization vector). In NIST SP 800-38A [30] we can find several requirements for the IV. They state that IV does not have to be secret, but it must be unpredictable. They mention that we should protect the integrity of IV.

We will see the usage of this mode in AES encryption/decryption. In that case, the blocks M_i and C_i are 128 bits long.

2.3 SHA256

SHA256 is a widely used cryptographic hash function. In general, cryptographic hash functions are used for the integrity protection of a message. A cryptographic hash function maps any input message into string of fixed size. Moreover, cryptographic hash functions must meet several criteria such as collision resistance and one-way security.

Since SHA256 is frequently used, we consider it a general knowledge and we provide only source without detailed explanation. The main sources are NIST standard FIPS 180-4 [31] and RFC 6234 [32].

2.4 HMAC

HMAC is a mechanism used for the message authentication/integration described in RFC 2104 [33] or in NIST standard FIPS 198-1 [34].

Firstly, we will briefly explain the main concept of MAC - message authentication code. MAC is an algorithm which outputs a code that guarantees the authenticity and the integrity of a sent message.

A sender and a receiver of the message share the same secret key. When the sender sends a message, he computes MAC from the message using the shared secret key. The MAC code is then sent together with the message. The receiver can compute the MAC code from the message since he has the same key. If both codes (received and computed) are equal, the message is authenticated and the integrity is checked.

HMAC stands for a particular hash based MAC algorithm. In our case the used hash function is SHA256, in general there are several different options for the hash function. If not stated otherwise, we assume that HMAC uses SHA256 and sometimes we do not write it explicitly. The security of HMAC is dependent on the used hash function. The scheme was designed to resist length extension attacks.

Notation

H	...	a hash function, in our case SHA256 if not explicitely written otherwise
B	...	the block size of the underlying compression function in H, in the case of SHA256 $B = 64$
L	...	the output size of the hash function, in our case 32 bytes
$ipad$	...	byte 36 repeated B times
$opad$	...	byte 5c repeated B times
K	...	shared secret key
m	...	message

According to RFC 2104 [33] the length of the key should be greater than or equal to L. Keys shorter than L can be used but they should not be due to the

security. A longer key does not significantly increase the security. Therefore, according to the NIST standard, if the length is greater than B, then the key is hashed (the output has length L) and padded with zero bytes to get length B. The result is the new key of the proper length. If the key is shorter than B, K is padded with zero bytes to get the length B. The key should be chosen at random and changed regularly.

Algorithm 2 HMAC

INPUT: key K, hash function H, message m
OUTPUT: HMAC code

return $\mathrm{H}(K \text{ XOR } opad||(\mathrm{H}(K \text{ XOR } ipad||m)))$

2.5 SRP protocol

The Secure Remote Password protocol is an asymmetric protocol used for a key exchange and an authentization by password. A protocol of this type is known as "PAKE" which stands for Password-Authenticated Key Exchange. Client proves his knowledge of password and both, cilent and server, agree on the same shared session secret. Only the client knows the password. The protocol has several versions. The current version of the protocol is 6a, which is described in RFC 5054 [35]. The description of SRP 6a without TLS can be found in [36]. There is also a paper describing the version 6 - SRP-6: Improvements and Refinements to the Secure Remote Password Protocol [37].

We provide a desciption of the protocol we will see, the version that was most probably taken from RFC 5054 [35]. We also used RFC 2945 [38] which describes older version of the protocol and a website [36] in order to make the description simpler.

Notation

$\|\|$	...	concatenation of strings
p	...	user's password
I	...	username
s	...	salt
N	...	a large safe prime (that means $N = 2q + 1$, where q is a prime)
g	...	group generator of Z_N^*
H()	...	a hash function
		RFC uses outdated function SHA1
k	...	a constant s. t. $k = H(N)$ XOR $H(g)$
x	...	$x = \mathrm{H}(s\|\|\mathrm{H}(I\|\|" : "\|\|p))$
v	...	verifier, $v = g^x \mathrm{mod} N$
a, b	...	private random ephemeral keys
		chosen by the client and the server respectively
A, B	...	public random ephemeral keys of the client
		and the server respectively,
		s.t. $A = g^a \bmod N$, $B = kv + g^b \bmod N$
PAD(a)	...	pad a with '0' from left to get the length of N in bytes,
		the padding follows the recommendation about padding
		given in RFC 2945 [38]

Before the protocol starts

Before the protocol starts, the client and the server have to agreed on generator g, modulus N. This can be done in advance or it can be sent during the communication by the server. In addition the server also has to know the verifier $v = g^x \bmod N$ which should be stored together with s and username I. The value of x is equal to $x = \mathrm{H}(s||\mathrm{H}(I||" : "||p))$. Both, verifier and username have to be known to the server in advance.

Protocol

The aim of the protocol is to create the shared session secret and confirm the knowledge of the user's password. At the end, the protocol verifies that both sides of the communication have the same session secret K.

Algorithm 3 SRP

CLIENT: Generate random a, compute $A = g^a \bmod N$, send A and I to SERVER.
SERVER: Take v and s that correspond to I, generate random b, compute $B = kv + g^b \bmod N$, where $k = \mathrm{H}(N||\mathrm{PAD}(g))$. Send B and s to CLIENT.
SERVER and **CLIENT** (both): compute u = $\mathrm{H}(\mathrm{PAD}(A)||\mathrm{PAD}(B))$
CLIENT: $x = \mathrm{H}(s||\mathrm{H}(I||" : "||p))$
CLIENT: $S_C = (B - k \cdot g^x)^{(a+ux)} \bmod N$, where $k = \mathrm{H}(N||\mathrm{PAD}(g))$.
CLIENT: $K_C = \mathrm{H}(S_C)$
SERVER: $S_S = (A \cdot v^u)^b \bmod N$
SERVER: $K_S = \mathrm{H}(S_S)$

Now both sides should have the same shared session secret K,
it should hold that $K = K_C = K_S$. The client and the server check it
in the following way:

CLIENT: Compute $M1 = \mathrm{H}(\mathrm{H}(N)\ \mathrm{XOR}\ \mathrm{H}(g)||\mathrm{H}(I)||s||A||B||K_C)$.
CLIENT: Send $M1$ to SERVER.

At this point, SERVER computes $M1$ on its own using K_S instead of K_C. If the value of the result is the same as the received value, the communication can continue. If not, SERVER aborts the communication.

SERVER: Compute $M2 = \mathrm{H}(A||M1||K_S)$.
SERVER: Send $M2$ to CLIENT.

Now CLIENT computes $M2$ using K_C instead of K_S and checks whether the result and received $M2$ are equal. If yes, the protocol proceeded correctly.

SERVER/CLIENT aborts the communication if received $A \equiv 0 \bmod N$ or $B \equiv 0 \bmod N$. SERVER sends B *after* receiving A and $M2$ *after* receiving correct $M1$. CLIENT also aborts communication if $u = 0$.

We note that all computations are computed in the finite field Z_N and thus we compute mod N (see pg. 13 in [39]).

Why does the protocol work?

We can check that the client and the server create the same shared session secret and that the protocol proceeds as expected. Firstly, we will show that $K_C = K_S$.

We have $K_C = \mathrm{H}(S_C)$ and $S_C = (B - k \cdot g^x)^{(a+ux)} \bmod N$,

where $k = \mathrm{H}(N||\mathrm{PAD}(g))$. Then

$$\begin{aligned} S_C &= (B - k \cdot g^x)^{(a+ux)} \bmod N \\ &= (kv + g^b - kg^x)^{(a+ux)} \bmod N \\ &= (kv + g^b - kv)^{(a+ux)} \bmod N \\ &= (g^b)^{(a+ux)} \bmod N \\ &= g^{b(a+ux)} \bmod N. \end{aligned}$$

We also have that $K_S = \mathrm{H}(S_S)$ and $S_S = (A \cdot v^u)^b \bmod N$.

Consequently,

$$\begin{aligned} S_S &= (A \cdot v^u)^b \bmod N \\ &= (A \cdot (g^x)^u)^b \\ &= (g^a \cdot (g^x)^u)^b \\ &= (g^{(a+xu)})^b \\ &= g^{(b(a+ux))}. \end{aligned}$$

Therefore $S_C = S_S$, thus $K_S = K_C$, because $K_C = H(S_C) = H(S_S) = K_S$.

The equality implies that $M1$ computed by the client and $M1$ computed by the server have to be the same if the protocol proceeds correctly. Then $M2$ computed by the server and by the client have to be the same.

The security of the protocol is based, similarly as Diffie-Hellman key agreement protocol, on the discrete logarithm problem. According to RFC [38] this means similar/same reqiurements on the choice of a, b, N, specifically on their lengths. More details about the security and proofs can be found for example in papers by the creator of SRP Thomas Wu [37] and [40] or in the paper by Dennis Dayanikli and Anja Lehmann [41].

2.6 OCRA

OCRA means OATH Challenge-response algorithm described in RFC 6287 [42]. The scheme provides the authentication.

OCRASuite is a string of the form

<Algorithm>:<CryptoFunction>:<DataInput>

and its value sets the mode of the operation. In this thesis we will see the following variant: OCRA-1:HOTP-SHA256-6:QH64-T30S. In more details:

- *Algorithm.* This value specifies the version of OCRA. OCRA-1 is a version descibed in RFC 6287 [42].
- *CryptoFunction.* According to RFC 6287 [42], HOTP is HMAC-based one-time password algorithm which can be found in RFC 4226 [43]. The hash function SHA256 is used for a computation of the HMAC code in HOTP. The value 6 means the number of output digits in the response.

- *DataInput parameters.* The list of input data to OCRA. QH64 means that challenge question is hexadecimal, 64 nibbles (1 nibble = half of byte) long. T30S means that one of the DataInput parameters is a timestamp - the number of time-steps since the Unix epoch. One time-step size is 30 seconds.

The variant relevant for us is the following one-way challenge response:

Algorithm 4 One-way challenge response scheme

Let K be the secret key shared by CLIENT and SERVER,
T is the time-stamp described above,
Q is a challenge question, in our case hexadecimal string 64 nibbles long.

SERVER: Send Q.
CLIENT: Compute R such that R=OCRA(K, {Q, T}).
Send R to SERVER.
SERVER: Compute R using the shared secret key K. Verify that the response R is valid.

ORCA algorithm which we use is HOTP, which is described in RFC 4226 [43]. The main idea is that only the client can compute the correct response R.

Algorithm 5 HOTP

INPUT: Let K be the shared secret, let C be the counter value (8-bytes).
OUTPUT: HOTP

return HOTP(K,C)= Truncate(HMAC-SHA-1(K,C))

In the thesis we will see HMAC-SHA-256 instead of HMAC-SHA-1. During the computation of OCRA, we use {Q,T}, where Q is a question challenge and T is a time-stamp, instead of a counter C, which is given in RFC. Truncate function truncates the output of HMAC as it is given in RFC 4226 [43].

Algorithm 6 Truncate

INPUT: HS = HMAC-SHA-1(K,C). The length of HS is equal to 20 bytes.
OUTPUT: Truncated HS

Step 1:

HS = HS[0]HS[1]...HS[19], where HS is a string
OffsetBits = low-order 4 bits of HS[19]
Offset = conver OffsetBits to number $\in \{0, ..., 15\}$
S = HS[OffsetBits]...HS[OffsetBits+3]
Sbits = last 31 bits of S

Step 2:

Snum = convert S from string to a number $\in 0, ..., 2^{31} - 1$
return Snum mod 10^s

In the OCRASuite we describe $s = 6$. Hovewer, in the following chapters we will see that the truncation will not be needed.

Why does the protocol work?

A client and a server share the same secret key. The main idea is that the client manages to compute the correct response R if and only if he has the correct key. The security relies on HMAC and its underlying hash function used in the scheme.

Also, the response R is OTP - one time password. In this case the OTP depends on the time and a secret key. The password (response R) is valid for only limited time, then it changes. Typically the bruteforce attack on OTP is commonly mitigated by allowing only few attempts. Therfore the bruteforce attack is not possible even if R has only 6 digits.

2.7 PBKDF2

PBKDF2 is a key derivation function that can be found in RFC 2898 [44]. Informally speaking, it uses a key derivation functions to derive a long strong key from the weak and short password. PBKDF2 function is also used for the password hashing [45]. In a simplified way, PBKDF2 uses many iterations of a pseudorandom function, which is typically HMAC-SHA256. Here is the description from RFC 2898 [44]:

Notation

Firstly, we introduce the notation:

p	...	user's password
salt	...	salt
c	...	desired number of iterations
dkLen	...	the length of the derived key, according to RFC in bytes
PRF	...	pseudorandom function, in our case HMAC-SHA256
hLen	...	the length of the PRF, according to RFC in bytes
INT(k)	...	the four-byte hexadecimal value representing the integer k (big-endian representation)

Algorithm

Algorithm 7 PBKDF2

Let || denote the concatenation.
INPUT: p, salt, c, dkLen
OUTPUT: derived key

1: $l = \lceil \frac{\text{dkLen}}{\text{hLen}} \rceil$
2: **for** i, where $1 < i < l$ **do**
3: T_i =F(P,salt,c,i),
where:
F(P,salt,c,i)= U_1 XOR U_2 XOR ... XOR U_c
and
U_1 =PRF(p,salt || INT(i))
U_2 =PRF(p, U_1)
⋮
U_c =PRF(p, U_{c-1})
4: **end for**
5: **return** $T_1||T_2||T_3|| \dots T_l$

In the next chapter we will see the variant with HMAC-SHA256 as a PRF. Unless stated otherwise, we assume the PBKDF2 uses HMAC-SHA256 as a PRF and we do not explicitly write it into a pseudocode. When using PBKDF2 in the following chapters, we use notation PBKDF2(p, salt, c, dkLen).

2.8 ECDH

ECDH (Elliptic Curve Diffie-Hellman) is a key agreement protocol. It is a modification of classical Diffie-Hellman [46] using elliptic curves.

Firslty, we will describe elliptic curves in terms elliptic curve cryptography. The following description is taken from [47]. We describe elliptic curves over $\mathbb{F}_p$ since it is the case we will see in this thesis. The second option would be to use elliptic curves over $\mathbb{F}_{2^m}$.

Elliptic curves

Cryptography uses elliptic curves over finite fields. Elliptic curve E can be described by Weierstrass equation as

$$y^2 = x^3 + ax + b \quad \text{s.t. } a, b \in \mathbb{F}_p,\ 4a^3 + 27b^2 \neq 0.$$

For our purposees we assume that $p \neq 2, 3$. In [47] they give an affine representation of the elliptic curve, where the set of rational points in E over $\mathbb{F}_p$ is denoted by $E(\mathbb{F}_p)$:

$$E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p^2 : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}.$$

The point $\mathcal{O}$ is a point at infinity. The cryptography uses the fact that $E(\mathbb{F}_p)$ has a group structure where $\mathcal{O}$ is an identity element and the operation $+$ at $E(\mathbb{F}_p)$ is defined as it is described in Section 2.3.1 in [47].

An elliptic curve group can be defined using domain parameters. Domain parameters are: a, b in the Weierstrass equation, the finite field $\mathbb{F}$ given by p, the base point G that generates a cyclic subgroup, the order n of this subgroup and cofactor $h = \frac{\#E(\mathbb{F}_p)}{n}$. We denote the number of elements by $\#$.

The protocol is based on the dicrete logarithm problem. We took the following description from [47]. The discrete logarithm problem is the follwing: Let us have cyclic group Gr with generator g of order n. Let us denote the sum (additive group operation) of $k \in \mathbb{N}$ elements $c \in Gr$ as

$$\sum_{i=1}^{i=k} c = [k]g.$$

The discrete logarithm of $m \in Gr$ to the base g is the unique $k \in \mathbb{N}, k < n$, such that [k]g=m. We denote it by $\log_g m$. Then the discrete logarithm problem is to find k if we know g and m. This is considered to be computationally hard for some elliptic curve groups with large n. We note that this is modified case for our problem. In general group does not have to be additive (it is not in the case of classic Diffie-Hellman [46]).

The dicrete logarithm in terms of elliptic curves is the following. Let us have the elliptic curve group given by its domain parameters, where G is a base point and n is the order of generated subgroup. Let us have point P from the subgroup generated by G. Then the problem is to find $k \in \mathbb{N}, k < n$, such that $P = [k]G$. For some curves is this problem computationally difficult. In [47] they call them computationally strong. In the section 2.3.4 in [47] they describe how exactly should be domain parameters set in order to obtain computationally strong elliptic curve group.

Protocol

Finally, we describe the protocol. The description is based on the description in the book Modern Cryptography by Easttom [48] and [49]. Let us have two communicating parties A and B.

Algorithm 8 ECDH

BOTH: Select domain parameters - agree on specific elliptic curve group of order n with base point G.
A: Generate a key pair. Choose private key $s_A \in \mathbb{N}, s_A < n$. Compute public key $P_A = [s_A]G$. Send P_A to B.
B: Generate a key pair. Choose private key $s_B \in \mathbb{N}, s_B < n$. Compute public key $P_B = [s_B]G$. Send P_B to A.
A: Compute shared secret: $K = [s_A]P_B$. If $K = \mathcal{O}$, then stop. Else $K = (x_K, y_K)$ and shared_secret $= x_K$.
B: Compute shared secret: $K = [s_B]P_A$. If $K = \mathcal{O}$, then stop. Else $K = (x_K, y_K)$ and shared_secret $= x_K$.

We can see that both, A nad B, have the same shared_secret, because $K = [s_A][s_B]G$. There is also a variant where K is multiplied by h, see for exampe Section 3.3.2 in [49].

2.9 ECDSA

Elliptica curves can be also used for signature. ECDSA stands for Elliptic Curves Digital Signature Algorithm. The following description of the algorithm was mostly taken from [47] by BSI.

Algorithm 9 ECDSA

Let A be a signer and B be the receiver of the signed message.

Preparation
A,B: Select domain parameters - agree on specific elliptic curve group of order n with base point G.
A: Choose a message M to be signed.
A: Generate key pair. Choose private key $s_A \in \mathbb{N}, s_A < n$. Compute public key $P_A = [s_A]G$.

Signature
A: Choose random $k \in \mathbb{N}, k < n$.
A: $W = [k]G$
A: Let $W = (x_W, y_W)$. Convert x_W from the finite field element to integer.
A: Compute $r = x_W \mod n$. If $r = 0$, then generate a new k and continue as before.
A: $k_{inv} = k^{-1} \mod n$.
A: Compute $h =$HASH$_l(M)$, where HASH is a chosen hash function. The output of the HASH function is truncated to l by taking l leftmost bits. Then the value is converted to integer.
A: $s = k_{inv} \cdot (r \cdot s_A + h) \mod n$. If $s = 0$ then choose again random k and continue again as before.
A: Return signature (r, s)

Verification
B: Verify that r, s are valid values, that is, $r, s \in \mathbb{N}, r, s, < n$.
B: Compute $h =$HASH$_l(M)$. h is converted to integer.
B: $s_{inv} = s^{-1} \mod n$.
B: $u_1 = s_{inv} \cdot h \mod n$
B: $u_2 = s_{inv} \cdot r \mod n$
B: Compute $Q = [u_1]G + [u_2]P_A$. Check that $Q \neq \mathcal{O}$.
B: $Q = (x_Q, y_Q)$
B: Convert x_Q from the finite field element to integer. Then compute $v = x_Q \mod n$.
B: If $v = r$, then the signature is valid. Otherwise it does not.

Why does the signature verification work?

We are interested in the first coordinate of Q. We want to show that if the signature is done correctly (as it is described in the algorithm), then $v = r$ that is, $x_Q \mod n = x_W \mod n$. In the computation we will use the algorithm above, we will substitute corresponding values to obtain the result.

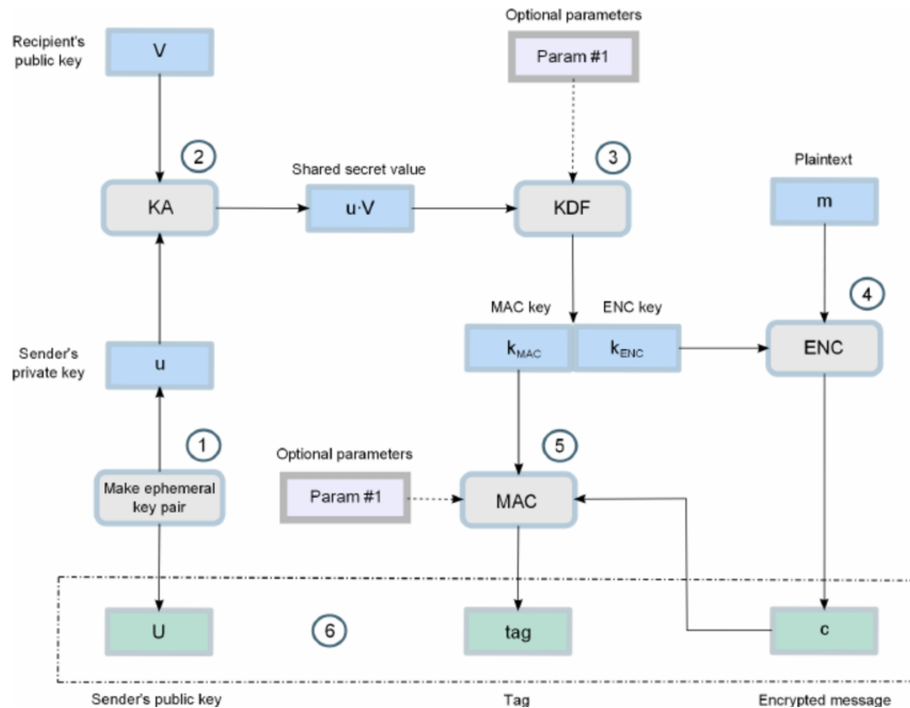
$$\begin{aligned}
Q &= [u_1]G + [u_2]P_A \\
&= [s_{inv} \cdot h]G + [s_{inv} \cdot r]P_A \\
&= [s_{inv} \cdot h]G + [s_{inv} \cdot r][s_A]G \\
&= [h + r \cdot s_A][s^{inv}]G \\
&= [h + r \cdot s_A][s^{-1}]G \\
&= [h + r \cdot s_A][(k_{inv}(r \cdot s_A + h))^{-1}]G \\
&= [h + r \cdot s_A][k^{(-1)(-1)}(r \cdot s_A + h)^{-1}]G \\
&= [k]G \\
&= (x_W, y_W)
\end{aligned}$$

Then

$$v = x_Q \mod n = x_W \mod n = r.$$

2.10 ECIES

ECIES stands for Elliptic Curve Integrated Encryption Scheme. ECIES combines symmetric and asymmetric cryptography based on elliptic curves. There are several standards describing the scheme, most of them is not available for free. There is a paper that describes differences between standards [50]. The following scheme is taken from: [51], where it is labeled as Figure 2.



Denote the sender by A and the receiver by B. From the figure, [49] and [50] we can see that before using the scheme we have to establish follwing algorithms and parameters:

- KA - key agreement function, more precisely ECDH 2.8, according to [49]we should choose which variant of ECDH we will use (whether EDCH with cofactor, that is described in [49] in Section 3.3.2, or not)
- KDF - key derivation function
- MAC - message authentication code
- ENC - symmetric encryption algorithm
- HASH - a hash function
- Domain parameters

Usually standards have a list of supported functions.

Algorithm

As we can see from the above picture, the output of the scheme is an encrypted message and its MAC. In order to obtain the key for symmetric cipher that is used to encrypt the message, sender A and reciever B create a shared secret using asymmetric scheme based on elliptic curves. This shared secret is then used to derive keys for encryption and for MAC. The following description is mostly taken from [49].

Algorithm 10 ECIES

Key deployment
A: Generate private key s_A and public key P_A.
B: Generate private key s_B and public key P_B.

ECIES - encryption
Let M be the message. Let $T = (p, a, b, G, n, h)$ be domain parameters. According to [49] there are also optional shared strings $SharedInfo_1$ and $SharedInfo_2$.
A: Generate an ephemeral pair of keys. Choose a random ephermeral private key $s_A$eph and corresponding ephemeral public key $P_A\text{eph} = [s_A\text{eph}]G$.
A: Compute the shared secret using ECDH. That is, compute $K = [s_A\text{eph}]P_A$ or $K = [h \cdot s_A\text{eph}]P_A$ depending on the variant of ECDH.
A: $K = (K_x, K_y)$. Then the shared secret is $z = K_x$.
A: Derive the secret key using chosen KDF and z, s_key= KDF(z). The key derivation function can also use (see [49]) $SharedInfo_1$ as an input.
A: From s_key obtain the encryption key enc_{key} and input secret to MAC mac_{key} as follows: s_key= $enc_{key}||mac_{key}$, where $||$ denotes the concatenation.
A: Use enc_{key} and ENC algorithm to encrypt the message M.
$C = \text{ENC}(M, enc_{key})$.
A: Compute MAC of the ciphertext C. Optionally, also of the $SharedInfo_2$ as follows: $D = \text{MAC}(mac_{key}, C||SharedInfo_2)$.
A: Send $(P_A\text{eph}, C, D)$ to **B**.

ECIES - decryption
Let $(P_A\text{eph}, C, D)$ be the received ephemeral public key, encrypted message and its MAC. Let $SharedInfo_1$ and $SharedInfo_2$ be optional shared strings.
B: Compute the shared secret using ECDH. That is, compute $K = [s_A\text{eph}]P_A$ or $K = [h \cdot s_A\text{eph}]P_A$ depending on the variant of ECDH.
A: $K = (K_x, K_y)$. Then shared secret is $z = K_x$.
B: Derive encryption key and key to MAC as before: $enc_{key}||mac_{key} = \text{KDF}(z)$. Optionally KDF uses $SharedInfo_1$.
B: Firstly, check the MAC D by computing
$D = \text{MAC}(mac_{key}, C||SharedInfo_2)$.
B: Secondly, decrypt the recieved ciphertext C and obtain the message.
$M = \text{ENC_decrypt}(C, enc_{key})$.

2.11 ANSI X9.63 KDF

This section describes a key derivation function given in standard X9.63. We will see that this function is used in ECIES described above. Since the standard is not publicly available, the following description is taken from the section 3.6.1. in [49]. Our description is simplified, therefore for more detailed description we recommend to see [49].

Before the derivation, we have to establish the hash function HASH we will use and its output length $hash_len$. The input to the KDF is the shared secret z, integer key_len which is the length of the derived key and optionally $SharedInfo$, which are some shared data.

The derivation proceeds as follows:

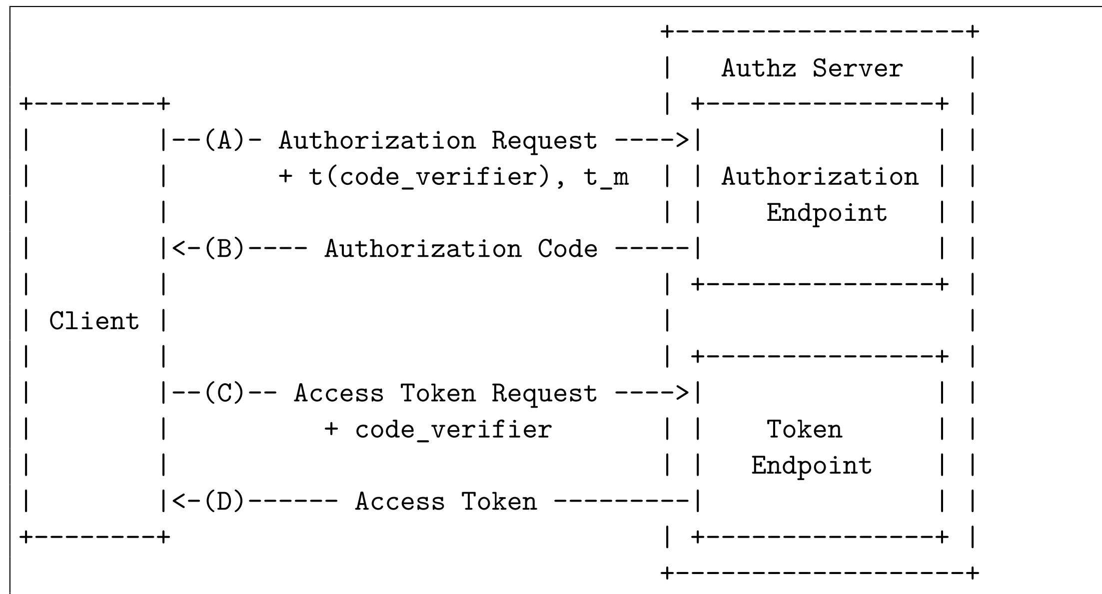
Algorithm 11 ANSI X9.63 KDF

Let $Counter = 0x00000001$. Let us denote the concatenation by $||$.
for (i=1;i$\leq \lceil key_len/hash_len \rceil$; i++) **do**
    $K_i = \text{HASH}(z||Counter||SharedInfo)$
    $Counter = Counter + 0x1$
end for
return $K_1|| \dots ||K_{\lceil key_len/hash_len \rceil}$

2.12 OAuth 2.0 PKCE

In this section we briefly describe OAuth 2.0 PKCE protocol. We will see this protocol in one of the applications. However, the protocol is not very interesting from the point of cryptography, therefore we will not go into details.

The main idea of the protocol is that the client needs to identify to use some service. However, the identity verification is not done by the provider of the service, but some identity provider, authorization server. OAuth 2.0 is the protocol and PKCE means Proof Key for Code Exchange. It is a modification of the protocol OAuth. RFC 7636 [52] explains the reason of this modification and also describes the protocol. In the same RFC there is this diagram of the protocol flow:



Firstly we will explain the notation in the scheme.

The code_verifier is a cryptographically random string generated by the client.

Letter t denotes the transformation and t_m means the transformation method. We will see the version of this protocol where t_m is S256 which is, according to RFC 7636 [52], SHA256 with encoding. The encoding and SHA256 are used as follows: t(code_verifier) is equal to
BASE64URL-ENCODE(SHA256(ASCII(code_verifier))). t(code_verifier) is usually called code_challenge.

Access token can be JWT token that grans the access to the service.
The communication is in RFC described as follows:

- (A) Client creates and stores code_verifier, then it derives t(code_verifier) and sends it with t_m to the authorization endpoint (AE).
- (B) AE responds with authorization code. It is the code that is given to the client after he is authentized/performed authorization. It can be used when requesting the Access token.
- (C) The client sends Access Token Request (using authorization code) with the code_verifier.
- (D) The authorization server then computes t(code_verifier) using saved t_m and compares the result with the value he received earlier. If they are same, it sends Access Token to the client.

In RFC thay claim that then the attacker who intercepts the authorization code cannot send a valid Access Token Request. This is of course the consequence of the used t_m method - in our case it is SHA256, which is a cryptographic hash function.

3 MyAir

In following three sections we will analyze and describe the cryptography used by MyAir, which is a mobile banking application by Airbank.

We opened the bank account in MyAir in a standard way in order to analyse the cryptography. Everything proceeded very quickly and without problems.

The only problem was that we could not to test our hypothesis in the practical part. After the analysis we contacted the bank by e-mail and asked them whether it would possible to send some request to their server to check that our analysis is correct. They politely rejected the request. Then we asked them if they could provide some testing envorinment. They again politely rejected. Thus in the practical part we run different experiment than we originally wanted.

We describe the analysis in three sections. The first is the theoretical part, where we describe used algorithms and protocols. The second is the practical part where we describe the practical aspect of the analysis. The last is the conclusion where we summarize all our comments on the security of analyzed protocols.

In examples of the communication we rewrote all sensitive information by stars "****".

3.1 Protocol description

In this section we describe cryptography in the application MyAir (mobile application by Airbank) from the theoretical point of view. We focus on important functions such as login, payment (transaction) or registration. We describe their algorithmic implementation and if needed we provide the pseudocode.

3.1.1 Overview

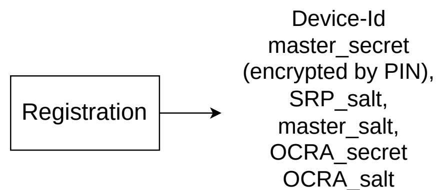
In this section we provide the overview of the chapter. Firstly, we provide a unified notation 3.1.2. Secondly, we describe individual functions and protocols:

- *HMAC*. The application uses HMAC to prove that messages come from the client/server and that the content of messages remains unchanged. The HMAC algorithm is used to sign a body of requests/responses.
- *Registration*. The registration links the device (installation) with the user's account.
- *Login*. To use the application, the user needs to log in, entering the PIN code.
- *Encrypt/decrypt by session secret*. A helper function which can be used to decrypt some information sent by the server.
- *Authorization*. To perform any potentially sensitive operation (like the payment) the client proves the server that he is authorized.
- *Change of PIN*. During the change of PIN by the user, more changes are happening in the background.

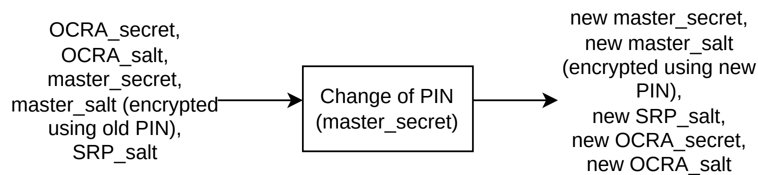
The overview of input/output/shared values

Since many values are shared by more function, for better understanding we provide a simplified graphs. They do not contain everything, however, they show simplified relations between functions and some shared values. The registration and the change of password are shown separately as they generate new values.

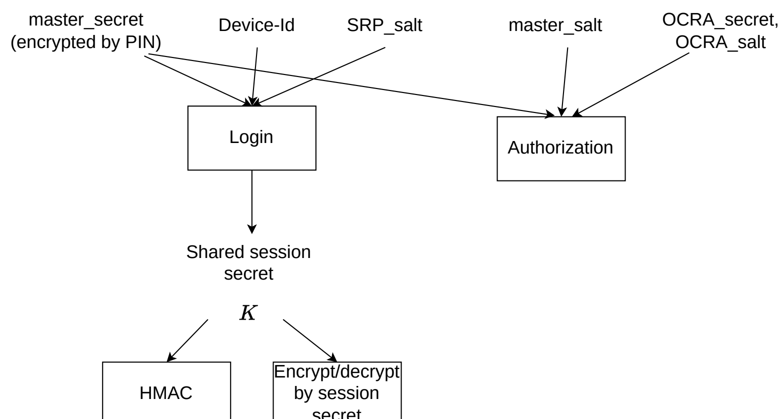
Registration:



Change of password (PIN):



Remaining functions:



Details about every function can be found in following sections.

3.1.2 Notation

For the sake of clarity we provide a summary of some important parameters and their lengths which we use in more sections.

$\|\|$	...	concatenation of strings
K	...	shared session secret, output of SRP protocol, 256 bits
session_AES_key	...	AES key derived from K, 256 bits
session_AES_IV	...	AES IV derived from K, 256 bits
master_secret	...	master secret that is derived during the registration, 256 bits
master_salt	...	salt that was created during the registration, 256 bits
OCRA_secret	...	shared secret that is used in OCRA algorithm, 256 bits
OCRA_salt	...	salt that is used in the authorization, 256 bits
PIN	...	user's PIN that is entered manually, 6 numbers 0-9
auth_key_material	...	key material that is used for deriving AES key in encrypting/decrypting OCRA_secret, 256 bits
server_hmac_key	...	HMAC key for signing the communication from the server to the client, derived at start of every session, 256 bits
client_hmac_key	...	HMAC key for signing the communication from the client to the server, derived at start of every session, 256 bits
SRP_salt	...	salt used in SRP protocol, 256 bits
Device-Id	...	Id of the device, created during the registration it identifies the user (resp. his device) in the communication, 256 bits

3.1.3 HMAC

The HMAC of almost every http request/response is computed in order to confirm the integrity and the authenticity of a message. We described the computation in 2.4. The application generates two HMAC keys as described in the section about login 3.1.4. Keys are generated at the start of every session (after logging in). One of them is for signing messages from the client to the server, while the second is for signing messages from the server to the client. Only the body of the HTTP message is an input to HMAC.

Implementation

We do not provide a pseudocode since it would be a usage of HMAC function with `server_hmac_key` or `client_hmac_key`, depending on the direction of the

communication. HMAC is in the application implemented in Security Module in the function signMessage. For more details about Security Module see the next chapter 3.2.1. Our implementation can be seen in `hmac_komunikace.py` which can be used to chcek/compute HMAC if we know the pair of hmac keys. The body of the request is transformed into bytes before the computation.

Example

In the following example of captured communication, HMAC is computed from the bytes of `{"places":["CARD_FRAME"]}`. The key client_hmac_key is used for signing messages from the client to the server.

```
POST /promo/v3/offer HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
           FD8842A54BBA5D6
...
Hmac: 2a362968b14df245ab28a6e0a9ff25c49d4b705506260580af09f4
      caf9ed76f4
Content-Type: application/json
...

{"places":["CARD_FRAME"]}
```

Conclusion

The usage of HMAC is relatively common. It is desirable to sign messages and check their authenticity and integrity. The variant of HMAC with SHA256 is considered to be secure, see for example Minimum Requirements for Cryptographic Algorithms by NÚKIB [53] or Cryptographic Mechanisms: Recommendations and Key Lengths by BSI [28]. In our case the length of the input key and the output tag are 32 bytes, which is considered to be sufficient.

Problem is that the only part of the request/response that is signed is the body. Therefore the attacker can change the uri the request is sent to or the method or headers. We have verified this attack is possible to perform.

3.1.4 Login

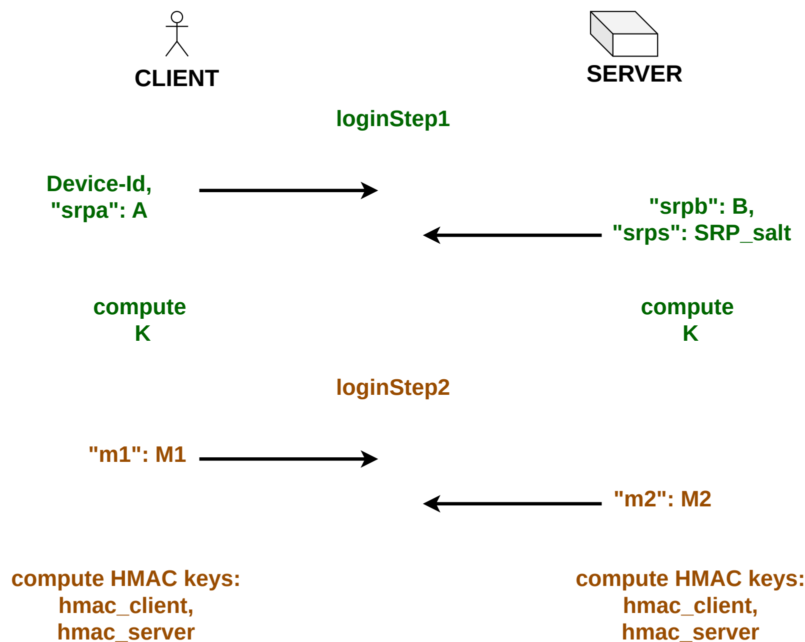
Login is performed using the SRP protocol described in 2.5. It nearly fully matches the algorithm described in RFC, although we can find several minor differences. In this section we give technical details of the login operation and differences. The aim of the algorithm is to log user in, that is, the authentication. It also derives the shared session secret and then uses it to derive two HMAC keys for the current session.

Notation

The notation is similar to the notation in SRP 2.5.

I	...	Device-Id, username in SRP protocol
$K = K_C = K_S$	...	shared session secret
g	...	generator
N	...	modulus, it has length 2048 bits
master_secret	...	the password p which is used in SRP protocol
SRP_salt	...	salt used in SRP protocol
H	...	hash function SHA256
PAD	...	padding by '0' bytes from the left to get the length 256 bytes
A, B, a, b	...	values from the SRP protocol described in 2.5 a has length 2048 bits
$m1 = M1, m2 = M2, k$	...	values from the SRP protocol

Scheme of communication



The communication is divided into two steps: loginStep1 and loginStep2.

loginStep1:

1. CLIENT: wants to log in, sends the value of A as "srpa"
2. SERVER: responds with "srpb": B and "srps": SRP_salt

loginStep2:

3. CLIENT: sends $m1$
4. SERVER: responds with $m2$, this response is the first element that is signed using HMAC

Implementation

Most important parts of login are implemented in Security Module (for more details see the next chapter) in 3 functions: loginStep1, loginStep2 and loginStep3.
In **loginStep1** the exponent a is generated and then A is computed.
In **loginStep2** the shared session secret K is computed using values received from the server, the computation of $m1$ is performed.
In **loginStep3** the value of $m2$ received from the server is checked and two HMAC keys are derived.

From the technical point of view, in the bank they implemented higher layers of the SRP protocol, but all functions at a lower level, for example HMAC, are taken from OpenSSL library [54] as it is described in the next chapter.

We implemented the login using Python programming language in `login.py`.

Parameters and storage

In this subsection we describe where to find parameters. The modulus and the generator in SRP can be found in Security Module (for more details see the next chapter), directly in the decompiled code. They are hardwired into code, thus we know their exact values and they do not change. Values are taken from RFC 5054 [35].

```
g = 0x02
N = 0xac6bdb41324a9a9bf166de5e1389582faf72b6651987ee07fc3192
    943db56050a37329cbb4a099ed8193e0757767a13dd52312ab4b0331
    0dcd7f48a9da04fd50e8083969edb767b0cf6095179a163ab3661a05
    fbd5faaae82918a9962f0b93b855f97993ec975eeaa80d740adbf4ff
    747359d041d5c33ea71d281e446b14773bca97b43a23fb801676bd20
    7a436c6481f1d2b9078717461a5b9d32e688f87748544523b524b0d5
    7d5ea77a2775d2ecfa032cfbdbf52fb3786160279004e57ae6af874e
    7303ce53299ccc041c7bc308d82a5698f3a8d0c38271ae35f8e9dbfb
    b694b5c803d89f7ae435de236d525f54759b65e372fcd68ef20fa711
    1f9e4aff73
```

The exponent a is generated randomly as 256 bytes long number. For the protocol we also need B, s which are received from the server. The password, in the description of SRP protocol 2.5 denoted by p, is our master_secret created during the registration/the change of password. The username, in the description of the protocol denoted by I, is our Device-Id, which is Id of a device created during the registration. The server needs to know the verifier v, username I and salt s (SRP_salt) in advance. All these values are created during the registration or with the change of PIN.

Pseudocode and differences

We do not describe the full SRP protocol since the implementation in the application mostly does not differ from the protocol in RFC 2.5. The hash function is SHA256 as already claimed. Other differences are:

- The value of k is computed in advance and it is hardwired into code. This hardwired k corresponds to the protocol, where $k = \mathrm{H}(N||\mathrm{PAD}(g))$. Padding PAD() is described in 2.5.

- K_C=H(PAD(S_C))
- The value of $M1$ is computed with a padding as follows:

$$\text{fixed_value} = \text{H}(N) \text{ XOR } \text{H}(\text{PAD}(g))$$
$$M1 = \text{H}(\text{fixed_value}||\text{H}(I)||s||\text{PAD}(A)||\text{PAD}(B)||K_C).$$

  The fixed_value is hardwired into the code.

- $M2 = \text{H}(\text{PAD}(A)||M1||K_S)$
- At the end of the protocol, two session hmac keys are created after checking $m2$ from the server:

$$\text{client_hmac_key} = \text{HMAC-SHA256}(K, \text{'client_hmac_key' in bytes})$$
$$\text{server_hmac_key} = \text{HMAC-SHA256}(K, \text{'server_hmac_key' in bytes}).$$

  We can see that the derivation of keys is done by HMAC. This follows the recomendation given by BSI [28] in Appendix B for key derivation after the key exchange.Two HMAC keys are keys for signing messages in the current session. The client_hmac_key is for signing messages from the client to the server and server_hmac_key is for signing messages from the server to the client.

  We note that the RFC 2945 [38] mentions the possibility of precomputing some values in advance to achieve a faster computation.

Request - details

Here is an example of the captured communication while logging in. Notice that the first communication does not have HMAC code, simply because HMAC keys have not been derived yet.

loginStep1

```
POST /security/loginStep1 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
FD8842A54BBA5D6
Content-Type: application/vnd.airbank.mas.loginstep1request+json
...

{"appVersion":"A.10.0.1",
"deviceType":"Pixel 3 XL",
"os":"Android 12",
"srpa":"733e380bab05ee9caa2eebf81d54895d1380b75f7dd0131c5366deac6
2f32f7bc7b0d9033a18d689393b96f71bc62e60d67f9b1fd5dd5b15dc1c67a7e7
0f689105b6fa2e26040c776a7f6a08667344a8ff9b3be7bfdea7a776659bc9c49
f1b6ca84524ad59e9a91884c8c169198ec7b544e9512a0bce684c8476ac7365e2
ad3de7b18384568fa4139dc1e47676136de05b22574d8cb299c5cbb388b57246e
6617694805091c43c1e69e529c7ac6b05f62693c1b3e8bbf2fb5bb2d86ce8d523
```

```
350b96d3cc0d182538cde93a6d13bbaac998fb740388e085d49ee919897e975fa
a663159823835ffafdf6872e61d2b9fd1f80abd457b5d4600f6d64a7de3e553f2",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Fri, 17 Jan 2025 12:52:40 GMT
Content-Type: application/vnd.airbank.mas.loginstep1response+json;
              charset=UTF-8
Content-Length: 597
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-17:13:52:40.124 +0100
...

{"srpb":"6ee53225d3c8697e8d6370fb2c3429bda6307f65df5b013be4f19053c
         56a897b5be06ac214775fa70a9e180048641b2ca419e525bdb192062d
         dee24fcf3f3b2e111363df08684b5ee410211a7e7ae2c71a357b88821
         c53d3758197537d63b3ca6fc0a4b2b53c0428e9853ac79e3a2c7bdf07
         f2810688f78f1c11e1f7ed6446ad8aa83d765358a2a50de57c08cfc79
         98701054a0dacfe84ff13c3a6882548e89f2d8373d7b9e9249d656937
         58329098bfe0b65d210a1092e21586c513ab71287d5c98781b968b382
         6424930651ad1b57539faa507286738cf035f5ba819695fc9a5c06a9c
         3cfb64dc4acdd9973867d5875e7d1e96f21cca5c60a059042c379269",
"srps":"8ac0ea551d826242c91c080e69418d878ffc3af680561bd917ac515945
        4c4d6a"}
```

loginStep2

```
POST /security/v4/loginStep2 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
           FD8842A54BBA5D6
Content-Type: application/json
...

{"deviceType":"Pixel 3 XL",
"m1":"6be2182b8aa38bc855db9cc1f2c22b07a9a3a77f8c2e127102c4aac8
      e6edfbec",
"os":"Android 12",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Fri, 17 Jan 2025 12:52:41 GMT
Content-Type: application/json;charset=UTF-8
...
Hmac: de7f46edc150f806cc8e6d37004e8cb6002287543bb70fa4e56ca07f
      424991e5
...
Http-Host: mb.airbank.cz
```

```
Real-Clock: 2025-01-17:13:52:41.461 +0100
...

{"applicantRef":null,
"contracts":[{"contractNumber":"*******",
             "firstName":"Karolína",
             "id":********,
             "lastName":"Zenknerová",
             "profileId":********,
             "relation":"OWNER",
             ...}],
...
"fullName":"Karolína Zenknerová",
"initializedForLogin":true,
"m2":"ff2ea6301ded64998f3482ab84302833ba5e82625ff49c33a5fc4ec470
      b8bb54",
"swtSetPrimary":false,
"threatsAvoidanceToken":"6d81f2e9-5da4-4bfd-98bf-68e74f9794e2",
"translationVersion":"99.0.58"}
```

Conclusion - differences and security

The implemented SRP protocol is not different from the formal protocol given in RFC 2.5.

The main difference is the usage of SHA256 instead of SHA1, which is an appropriate modification, see for example the section about hash functions in recommendation by BSI [28].

We can assume that if the security is similar to Diffie-Hellman key exchange, then we should require at least same lengths. According to recommendations by BSI [28] or NÚKIB [53] the length of the modulus should be at least 3000 bits/3072 bits. That is not satisfied since our modulus has 2048 bits. A recommendation is that the modulus should certainly be longer in the future. At this moment and for this purpose, 2048 bits is still acceptable.

Private exponent is generated to have also 2048 bits, which is sufficient and secure, but it could be shorter if needed.

The master_secret has 32 random bytes, which is definitely good enough against bruteforce attack, for example in the case that the verifier would be stolen.

There is some padding added, which makes no difference from the security point of view. In RFC 2945 they specifically mention that if the implemetation requires padding, the string should be padded with zeros from the beginning.

Some publicly known values are already precomputed and hardwired into the code which makes no difference from the security point of view.

3.1.5 Encrypt/decrypt by session secret

Encrypt and decrypt by session secret are helper functions that will be used in the section about an authorization 3.1.6 to decrypt received data.

Let K be the session secret (output of SRP protocol above). Encryption/decryption by session secret means encryption/decryption by AES CBC mode with

PKCS7 padding, using AES key and AES IV derived from K. The derivation of the key is performed by HMAC.

Implementation

In the application encryption/decryption by session secret is implemented in Security Module (see the next chapter 3.2.1) as functions encryptBySessionSecret and decryptBySessionSecret.

We implemented the function for decrypting in the `decrypt_card_info.py`, where it is used to decrypt information about our card sent by the server. For both functions, encrypt/decrypt we provide a pseudocode.

Algorithm 12 encryptBySessionSecret

INPUT: message m, K
OUTPUT: encrypted message

session_AES_key = HMAC-SHA256(K,'aes_cbc_key' (in bytes))
session_AES_IV = HMAC-SHA256(K,'aes_cbc_iv' (in bytes))
enc_message = AES_CBC_PKCS7_encrypt(m, session_AES_key, session_AES_IV)
return enc_message

Algorithm 13 decryptBySessionSecret

INPUT: enc_message, K
OUTPUT: message m

session_AES_key = HMAC-SHA256(K,'aes_cbc_key' (in bytes))
session_AES_IV = HMAC-SHA256(K,'aes_cbc_iv' (in bytes))
m = AES_CBC_PKCS7_decrypt(enc_message, session_AES_key, session_AES_IV)
return m

Conclusion

We will see that if the information about card is required more than once, then key and IV are reused. In general that is known to be an issue when encrypting by AES CBC mode, see for example NIST SP 800-38A [30]. In NIST SP 800-38A they state that the IV should not be predictable. From what we have seen, the function is used for encrypting/decrypting information about credit card, that is only a few decryptions. More specifically the number of card and url address. Theoretically, reusing the IV and the key means issues, for example, the attacker can recognize that the first blocks of two messages are the same. In this case these issues are not very likely. Additionaly, the integrity of the message body is protected by HMAC. However, in general we recommend to follow the speciffication.

There is no problem with padding, we assume the attacker cannot sign his message by the correct HMAC key, thus he cannot perform the padding oracle attack.

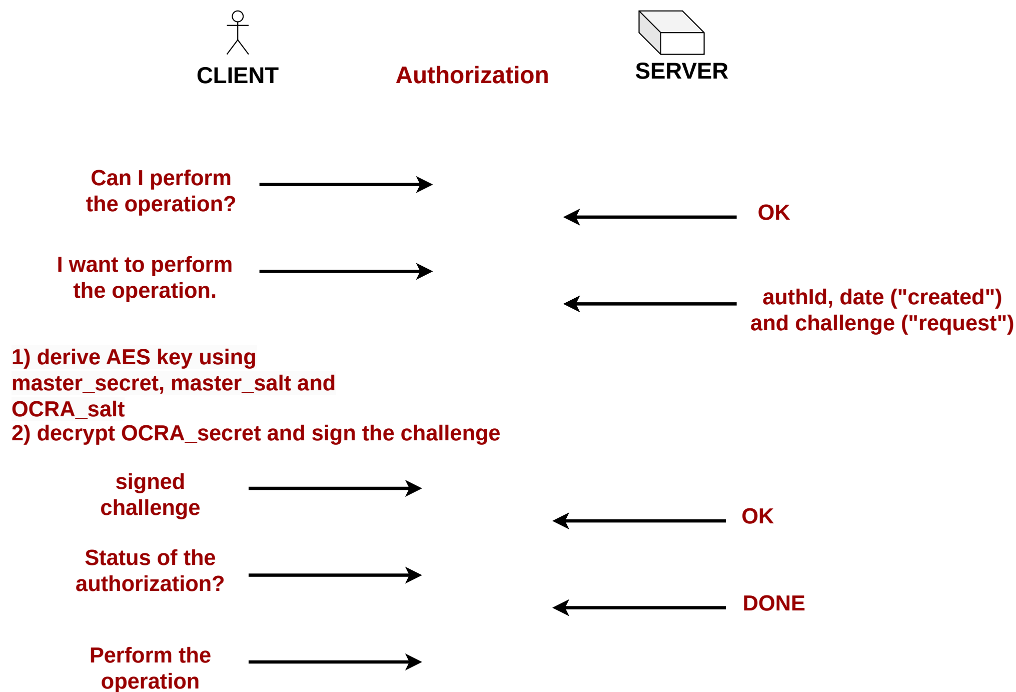
3.1.6 Authorization

Authorization is performed using OCRA-1:HOTP-SHA256-6:QH64-T30S which is descibed in 2.6. Authorization of every important operation is done after the client authenticates using OCRA. Examples of the authorization requests and responses for concrete cases are briefly described in subsections 3.1.6 and 3.1.6.

The protocol proceeds more or less as we would expect. More precisely, the client sends a request with an operation he wants to perform, the response from server contains authId, which is a unique Id number assigned by the server to the operation, time of creation and xml challenge request to sign. The client creates data to sign (input to OCRA) using authId and xml request challenge. The client creates a signature using OCRA_secret and OCRA algorithm. The client sends data to be signed and the signature. If the signature is valid, the client's request with operation is authenticated and the operation is authorized.

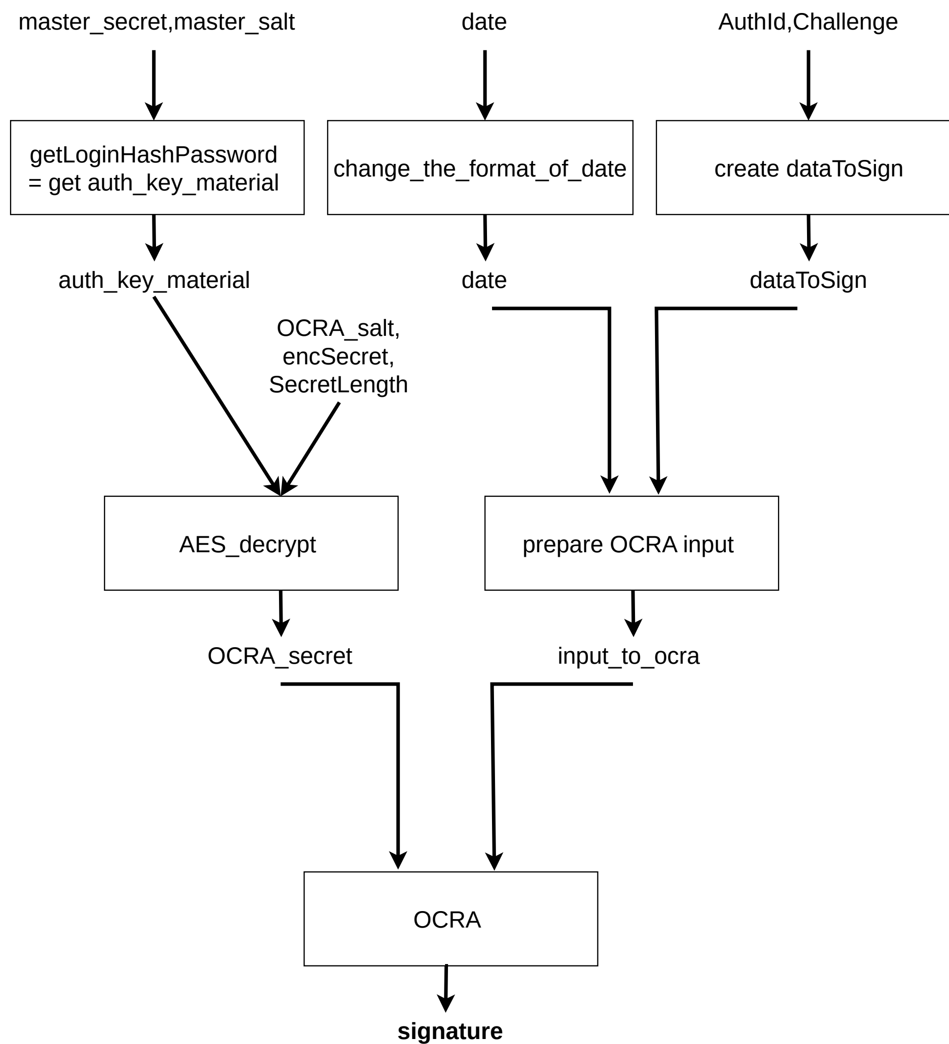
Scheme of communication

We will see several examples of the captured communication. In general, the communication proceeds as follows:



Description

We provide an overview of the signature process:



In following subsections we describe all functions needed for OCRA signature. We also provide their pseudocode.

Notation

Input parameters are:

date	...	date with time received from the server
AuthId	...	unique Id of the operation to be authorized, sent by the server
Challenge	...	challenge to be signed in xml format
OCRA_salt	...	salt used for key derivation, obtained from shared preferences (for more details about the storage see the next chapter)
master_secret	...	secret generated during the registration or during the change of password
master_salt	...	salt generated during the registration or during the change of password
encSecret	...	encrypted OCRA_secret, stored in shared preferences (for more details about the storage see the next chapter)
SecretLength	...	the length of used OCRA_secret, in our case, the length is equal to 32 bytes

Created values are:

auth_key_material	...	key material created using master_salt and master_key
date	...	date after the change of format
dataToSign	...	data to be signed, output of create dataToSign, the value is sent to the server
OCRA_secret	...	decrypted shared OCRA_secret
input_to_ocra	...	input to OCRA algorithm, created using date and dataToSign
signature	...	output from OCRA, value that is sent to the server

Where to get input parameters?

Parameters master_secret and master_salt are created during the registration/the change of password.

AuthId, Challenge and date are received from the server.

The shared secret key needed for OCRA signature, which we denote OCRA_secret, is encrypted by AES and stored in shared preferences in file CSMStorageManager.xml as "encSecret". In the same file we can find OCRA_salt and SecretLength, which is equal to 32 bytes.

OCRA_secret is created and sent by the server after the change of password or after the registration.

We describe all parts of the authorization in more details below. Firstly we describe all functions needed to create OCRA_signature. Then we describe the change of OCRA_secret that follows after the change of password or after the registration.

Create data to sign

In this subsection we show how to get data to sign. That is, we show how to create an input to OCRA algorithm. A computation can be viewed in the pseudocode. The more detailed version implemented in java can be find in attachments as `authorize_sestav_data_to_sign.java`. A.1. Input parameters are request challenge in xml format and authId. Both are received from the server.

Algorithm 14 Create data to sign

Let || denote the concatenation.
INPUT: String authId, String Challenge
OUTPUT: dataToSign

```
1: if Challenge != null then
2:     convert authId and Challenge from String to bytes
3:     return SHA256(SHA256(Challenge) || authId || [0,0,0,0,0,0,0,0])
4: else
5:     return SHA256(authId || [0,0,0,0,0,0,0,0])
6: end if
```

change_the_format_of_date

Here we provide a function that changes the format of date. The input date is received from the server in the form: YYYY-MM-DDThh:mm:ss±hhmm and the function returns hexstring.

Algorithm 15 change_the_format_of_date

INPUT: date (as it was received in response from server)
OUTPUT: date in different format

```
1: date = convert date (string) to unix time (long)
2: date = date/1000
3: date = date/30 (the time step is 30s)
4: date = convert date to hex string and to lower case
5: return date
```

Auth_key_material - getLoginHashPassword

We will need an AES key for decrypting OCRA_secret that is stored in shared preferences as "encSecret". In this subsection we provide a pseudocode for a derivation of a value which we call auth_key_material. This value is used for deriving the AES key needed for the decryption of OCRA_secret. The derivation uses PBKDF2 and values of a master_secret and a master_salt created during the registration/the change of password.

We note that master_secret and master_salt can be obtained from the memory as bytearrays, for more details see the next chapter. The format of master_secret has to be changed before the derivation of auth_key_material as follows: every hexadecimal nibble (half-byte) is expressed as a hexadeximal value using ASCII encoding. The result is a new and longer bytearray which stands for the same master_secret.

Implementation in application

The key material derivation is implemented in Security Module (see 3.2.1) as functions getLoginHashPassword and getLoginHashPasswordForOFT. In the first function, the computation of a key material is straightforward.

Algorithm 16 getLoginHashPassword

INPUT: master_secret, master_salt
OUTPUT: auth_key_material

```
return PBKDF2(master_secret, master_salt, 1000, 32)
```

The second function is similar, but firstly master_secret must be decrypted, because it is not stored as a plaintext. We also mention this in the section about the change of password below. The key for encryption/decryption of master_secret is master_sec_enc_key = PBKDF2(PIN, master_salt, 1000, 32). By PIN code we mean 6 digits that are entered by the user.
The key master_sec_enc_key is XORed to the master_secret to encrypt it. Therefore to decrypt it we have to XOR master_sec_enc_key.

Algorithm 17 getLoginHashPasswordForOFT

INPUT: encrypted_master_secret, PIN, master_salt
OUTPUT: auth_key_material

```
master_sec_enc_key = PBKDF2(PIN, master_salt, 1000, 32)
master_secret = master_sec_enc_key XOR encrypted_master_secret
return PBKDF2(master_secret, master_salt, 1000, 32)
```

Our implementation

We implemented only the first algorithm, because we do not store master_secret and we can suppose we have its value. It is possible to obtain it from the memory, see 3.2.4. The implementation of this function is part of `OCRA.py` A.1.

AES_decrypt

We need to decrypt OCRA_secret since we use it for the signature. The knowledge of the value proves to the server that the request to perform the operation comes from the client and that he is authorized. For this purpose we need to create an AES key, which we derive from OCRA_salt and auth_key_material. The

decryption of encSecret is done in AES_decrypt method in `signature_OCRA.java` A.1, it is called in `OCRA.py` while signing.

Algorithm 18 AES_decrypt

INPUT: auth_key_material, OCRA_salt, encSecret, secretLength
OUTPUT: decrypted Secret

1: new_array = concatenate(OCRA_salt, auth_key_material)
2: key = SHA256(new_array)
3: IV = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
4: decrypted_secret = AES_CBC_no_padding_decrypt(encSecret, key, IV)
5: **return** last secretLength bytes of decrypted_secret

The mode of encryption/decryption - AES_CBC_no_padding is a standard AES_CBC which does not use padding. OCRA_secret is exactly 32 bytes long therefore we do not have to use padding.

Prepare OCRA input

When we have the correct format of date and dataToSign, we can prepare the input.

Algorithm 19 prepare_OCRA_input

Let || denote the concatenation.
INPUT: dataToSign, date
OUTPUT: input data to OCRA

1: date = change_the_format_of_date(date)
2: dataToSign = pad dataToSign with zeros (right-hand side) to get 256 bytes
3: date = pad date with zeros (left-hand side) to get 16 bytes
4: ocra = convert "OCRA-1:HOTP-SHA256-6:QH64-T30S" to bytes
5: ocra = convert ocra to hex string
6: output = ocra || "00" || dataToSign || date
7: output = convert output to bytes
8: **return** output

OCRA signature

Finally, we can compute the signature of prepared data. There is a java version in `signature_OCRA.java`, which is used in `OCRA.py`.

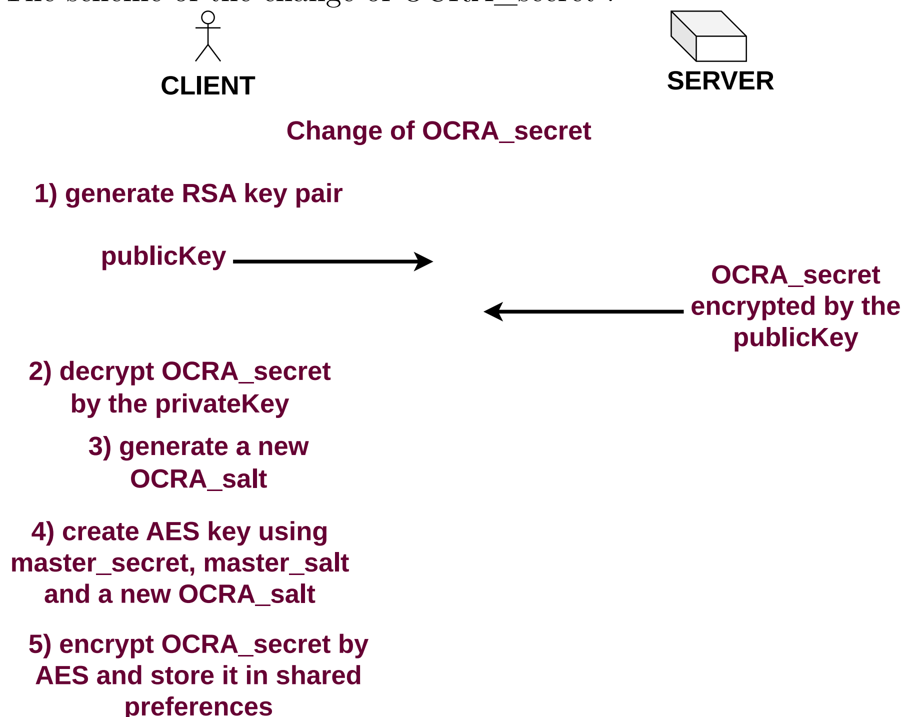
Algorithm 20 OCRA

INPUT: date, OCRA_salt, encSecret, dataToSign, auth_key_material
OUTPUT: signature

1: OCRA_secret = AES_decrypt(auth_key_material, OCRA_salt, encSecret, 32)
2: input_to_ocra = prepare_OCRA_input(dataToSign, date)
3: signature = HMAC-SHA256(OCRA_secret, input_to_ocra)
4: **return** signature as lowercase hex string

Change of OCRA_secret

OCRA_secret is changed after the change of password or after the registration. The scheme of the change of OCRA_secret :



The client generates a temporary RSA key pair, then he sends a request with his public RSA key (2048 bits) and the server responds with OCRA_secret encrypted by the public RSA key of the client. The client then decrypts the secret by his RSA private key, encrypts it by AES using auth_key_material and OCRA_salt and stores it in shared preferences.

The value of OCRA_salt needed for the key derivation in encryption is generated randomly and stored in shared preferences before the encryption.

Algorithm 21 Change of OCRA_secret

CLIENT: Generate RSA key pair (modulus has the length 2048 bits). Send public key publicKey to the SERVER.
SERVER: Encrypt the new OCRA_secret:
encSecret = encrypt_RSA_ECB_PKCS1(OCRA_secret).
Send encSecret to the CLIENT.
CLIENT: Decrypt OCRA_Secret:
OCRA_secret = decrypt_RSA_ECB_PKCS1(encSecret)
CLIENT: Generate a new OCRA_salt.
CLIENT: Encrypt OCRA_secret by AES_CBC_no_padding analogously to decrypting OCRA_secret above using master_secret, master_salt and OCRA_salt.
CLIENT: Store OCRA_salt and encrypted OCRA_secret in shared preferences.

Example of this communication can be seen below in 3.1.6.

Requests and responses - details

In this subsection we provide some additional details about the communication. Data to sign and the signature are converted into base64 before sending to the server. Data to sign can be found in requests as "authOperationReqHash". The signature is under the label "authCode". There is an element called "authElementId", which is simply Device-Id created with the registration of a device. There is also a header in requests which is called "logString".

It is computed during the OCRA computation and it is not signed using HMAC, becuse it is one of headers, not part of the json body. It probably does not have to be included in the communication. It uses RSA public key of the bank that is hardwired into code:

```
pubkey_cert = MIIDlTCCAn2gAwIBAgIJAMx/ZCykcKE9MA0GCSqGSIb3DQEBC
              wUAMGExCzAJBgNVBAYTAkNaMRcwFQYDVQQIDA5DemVjaCBSZX
              B1YmxpYzEPMA0GA1UEBwwGUHJhZ3VlMRcwFQYDVQQKDA5BaXI
              gQmFuayBhLiBzLjEPMA0GA1UEAwwGU1dUTE9HMB4XDTE2MTAx
              NzA4MzcxMVoXDTE3MTAxNzA4MzcxMVowYTELMAkGA1UEBhMCQ
              1oxFzAVBgNVBAgMDkN6ZWNoIFJlcHVibGljMQ8wDQYDVQQHDA
              ZQcmFndWUxFzAVBgNVBAoMDkFpciBCYW5rIGEuIHMuMQ8wDQY
              DVQQDDAZTV1RMT0cwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAw
              ggEKAoIBAQCwkR1NBeMiIPnISUlAdf0ZU+bhufrw9siHvlWOo
              S0t1tSr1ZvM6QQa9fVPRDYvN+Qw/ZRhF6kMTf3SK9dQ5YH5BV
              Z4O6E/CgLOnj1m0lZfmhA9mJ3UA8uVuzg3NWFLYlw5L1dESDw
              dO1wyOH1qjTsNoCNLWVh/GV0bgkqX68+Ea7EKpDRMNvvj4tAG
              /hbi1hxADIEMECkMk/6Xm3oO7cfsH1+Ztk61OkQ+5R7q3La7Y
              vjz10AX4RfLk8qlJarjBkgGNL8HyvxwYybis6UT2bUYg3DDie
              DKTQe+hwstXwoA/WTWhCe/c/8RqzY2XkryJEQw0X6WhZXYu3n
              i24GAMoUdAgMBAAGjUDBOMB0GA1UdDgQWBBThakHrXqYIfo71
              jlZFfY2k+dQUHTAfBgNVHSMEGDAWgBThakHrXqYIfo71jlZFf
              Y2k+dQUHTAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBCwUAA4
              IBAQCCT9ycyjUi7HvnZxMBe4vBCRE7FveUevTMLwzu90MfbrO
              xyLyXhWanGUJAVU9QMB85cm0TC25ja2lFZkb+km2PRqR45Oxe
```

VFPQYuUiXaAxQJyfg5eNMYjylLQOa12tBAV0Z0gJZbiPXIYRQ
SEfoexOIV1B8g6oz5oiVAAUfzlBsW16y+r7eU7wLjf0y3+/Z5
aK0MyuTSyZt3t6kCR0P8zNLcr+p6hBGbH+kXNJwj5PyUFY3QV
qewVBkhLrHLP4X4NO7odZ1nRBBPYoIlFUBEu8Lo1cgY+yj/nM
DuW43oBMSI24dafRW9UqpoiBRhxVmoXUXYtyb4+vxH+a9rCa1C8B

Using ASN1 decoder we can see that the certificate with public RSA key (2048 bits), we denote it pubKey, belongs to Airbank and it is signed by Airbank (it is selfsigned) and according to date it is not valid. It is valid notBefore 2016-10-17 08:37:11 UTC and notAfter 2017-10-17 08:37:11 UTC. However, since the logString is probably only some log, this is not a problem.

There is a pseudocode describing how to get the logString.

Algorithm 22 Computation of logString

INPUT: decSecret (decrypted secret), date, pubKey
OUTPUT: "logString"

1: part_decSecret1 = first 4 bytes of decSecret
2: part_decSecret2 = last 8 bytes of decSecret
3: tmp_array = concatenate(part_decSecret1, part_decSecret2)
4: tmp_array_hashed = SHA256(tmp_array)
5: date = change_the_format_of_date(date)
6: date = pad date with zeros (left-hand side) to get 16 bytes
7: date = convert date to bytes
8: array_to_enc = concatenate(tmp_array_hashed, date, byte 0)
9: logString = encrypt_RSA(array_to_enc) with ECB mode and PKCS1 padding
10: **return** logString

There is also a header "movingFactor". It is the String date converted to long using Java method. It is also outside the json body, thus not signed by HMAC. Here we describe it only for the sake of completeness.

Algorithm 23 Computation of movingFactor

INPUT: date
OUTPUT: "movingFactor"

1: date = change_the_format_of_date(date)
2: date = pad date with zeros (left-hand side) to get 16 bytes
3: **return** Long.parselong(date) ▷ java method

Headers logString and movingFactor are not important as they are part of input data to HMAC and they are probably only for logging.

Implementation in application

Most of the authorization functions are in Java package com.aheaditec.casemobilesdk.airbank. There is also a part that contains cryptography. We do not provide the list of functions as the decompiled version of

the application is not easily readable and describing whole decompiled structure would be probably useless for understanding the algorithm.

Our implementation

We attach code - in `OCRA.py` we call java methods, using java classes `authorize_sestav_data_tosign.java` and `signature_OCRA.java`, that prepare data and generate the signature A.1. Part of the implementation in `OCRA.py` is the function for the derivation of auth_key_material. In `decrypt_card_info.py` we implement the function decrypt_by_session_secret which is described above in 3.1.5. Java methods we use are:

- authorize_sestav_data_to_sign.get_data_to_sign - generates data to sign as described above
- signature_OCRA.create_logString_movingFactor - if requested, it can generate and print logString and movingFactor
- signature_OCRA.hash_and_concatenate_2_arrays_in - concatenates two input byte arrays and prints the result
- signature_OCRA.AES_decrypt - decryption by AES as described above
- signature_OCRA.get_bytes - gets java bytes from hex string
- signature_OCRA.get_date_string - changes the format of the date, the result is suitable for OCRA input
- signature_OCRA.prepare_OCRA_input - prepares input parameters to OCRA as is described above
- signature_OCRA.get_mac - returns the requested mac (for example HMAC_SHA256)
- signature_OCRA.c_OCRA - creates OCRA signature using OCRA_secret and input data
- signature_OCRA.k_OCRA_generation - puts everything together (secret decryption, preparing input, signature), resturns OCRA signature

In the next section we provide some examples of the communication between the client and the server and several details about some authorizated operations.

Card operations

In this section we show examples of card operations which need to be authorized. We note that for card operations we need to get the number ("value") of the card. It changes with every login, but it can be obtained by the corresponding request POST /general/cards HTTP/1.1.

View PIN

In this subsection we try to obtain the PIN code of our card.

The communication proceeds as expected. In 1. and 3. the number of card is sent (its current id). In 4. and 5. we can recognize the OCRA protocol - in 4. we can see the request challenge and in 5. we can see data to sign and OCRA signature in base64 ("authOperationReqHash" and "authCode"). We will focus on the last steps. In the step 8. which is response from server we obtain url that is encrypted by AES with CBC mode using session secret which is described in section 3.1.5. After decrypting it, we obtain usable url. In 9. we send GET request to url which we have obtained. As a response we obtain 10. where the PIN is visible in plaintext under the label "data".

1.
```
POST /cardSensitiveData/canDisplayPin HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
           FD8842A54BBA5D6
...
Hmac: 9fd694b48933bf57df5772651deb39301e1ed7623898dd7d91296e
      11894d48a5
Content-Type: application/json
...

{"value":13411870572853603}
```

2.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:28 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: 6b7a9306a68424e16d30d57e1643f3a92d3c94ec085095fbbeaeaaf4
      b499e299
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:06:28.691 +0100
...

{"resultCode":"OK"}
```

3.
```
POST /cardSensitiveData/displayPin HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD88
           42A54BBA5D6
...
Hmac: 9fd694b48933bf57df5772651deb39301e1ed7623898dd7d91296e1189
      4d48a5
Content-Type: application/json
...

{"value":13411870572853603}
```

4.

```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:29 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: 9cd11955a33f62ca12e947ce76f944ba049f24924d039338a5589538
      4e05a1ed
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:06:29.674 +0100
...

{"allowedElementTypes":["SWT-ONLINE-BIO","SWT-ONLINE"],
"authId":"AuLXVLPs.TpI6v39hnTPLyBfW1CFHDXhCXIhE36C5tbl.0sZsQaVyf
          7WlJEYDP6YEy_uJS9wleGyuEu4XJKwj3LYIg76qsFBFZSDJ4CTSTvb
          2XnKWhHGLPMsCxZEaHiT",
"contractId":********,
"created":"2025-01-08T17:06:29+0100",
"expiration":"2025-01-08T17:09:29+0100",
...
"request":"<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=
                \"yes\"?><ns5:registerPinViewRequest xmlns:ns6=
                \"http://cms.airbank.cz/ws/card/nfc\" xmlns:ns5
                =\"http://cms.airbank.cz/ws/card/sensitive\" xm
                lns:ns8=\"http://cms.airbank.cz/ws/card/nfc/con
                nect\" xmlns:ns7=\"http://cms.airbank.cz/ws/car
                d/payment/3ds\" xmlns:ns2=\"http://homecredit.n
                et/cardmanagement\" xmlns:ns4=\"http://homecred
                it.net/business\" xmlns:ns3=\"http://cms.airban
                k.cz/ws/envelope/delivery\"><ns5:cardId>*******
                </ns5:cardId><ns5:checkMoratorium>true</ns5:che
                ckMoratorium></ns5:registerPinViewRequest>",
"requiresMobileLogin":true}
```

5.

```
POST /authorization/v2/attempt HTTP/1.1
...
Content-Type: application/vnd.airbank.mas.attemptRequest+json
Accept: application/vnd.airbank.mas.attemptResponse+json
logString: KX3W5s9Lov7v6ADzZco38jilkc0Pt9kKwF312Y2QQ2kbBrHpPLJi9
           sBhfqLhE1MOtdvgnKdFoWrP5cCUGzqohV0XR+ulrGYgKr3sLSArlj
           UXXrQBGvBjExjRYwm7jzC4ydo3fP/g8KHaszdHQdL5WZYepZYjNBa
           YYr1olym5+5lj71X6ue9S+OIPjlxDSzUTwr9US0AdcfgLWExgmHq1
           GnEkdvTj6Or8S1COXBueZD2WVt8nqcNsmw0tC15SnbVY3uZRnTpVo
           ln7GvwRufZINuE+hN1bV1xvmCoBNGj57GW4EBLIQH6F4g5HbN9xs6
           jdJGdy74V5Y4ni/9PeZX+rcw==
movingFactor: 57878412
elementType: SWT_ONLINE
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD88
           42A54BBA5D6
...
Hmac: 6a35b28bcd237dd9952f94bfda3c2063feb8d148aa54465eaf7105c1c7
```

```
      331318
Profile-Id: 13355767
...
Host: mb.airbank.cz
...

{"deviceHwId":"",
"authElementId":"4C4426365C34029FDB65EC49D43C231664023E75A9F698A
                12FD8842A54BBA5D6",
"elementType":"SWT-ONLINE",
"authId":"AuLXVLPs.TpI6v39hnTPLyBfW1CFHDXhCXIhE36C5tbl.0sZsQaVyf
         7WlJEYDP6YEy_uJS9wleGyuEu4XJKwj3LYIg76qsFBFZSDJ4CTSTvb2
         XnKWhHGLPMsCxZEaHiT",
"attemptTo":"AUTHORIZE",
"deviceFlags":"AAAAAAAAAAA=",
"geoLocation":"",
"authOperationReqHash":"bvQHa7I6HwxPJ+CIHNdY3YAT9b5a8n8lqqDr\/s
                       IuG6E=",
"authCode":"MI3+itt6Y6B4tbta34K81KwAnZ6d9T3eTeUBSET3lvw="}
```

6.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:34 GMT
Content-Type: application/json;charset=UTF-8
...
Hmac: 191abfa9653b0fab6d4c71a68cb5e5709c0757c245cf246ddb05f6e4
      7672168f
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:06:34.908 +0100
...

{"result":"OK"}
```

7.
```
POST /authorization/status HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12F
           D8842A54BBA5D6
...
Hmac: 80c0b7be5826a99aefc15b720b65714df6522b77b5e824ed5d5e286
      107efc4f2
Content-Type: application/json
...

{"authId":"AuLXVLPs.TpI6v39hnTPLyBfW1CFHDXhCXIhE36C5tbl.0sZsQa
          Vyf7WlJEYDP6YEy_uJS9wleGyuEu4XJKwj3LYIg76qsFBFZSDJ4C
          TSTvb2XnKWhHGLPMsCxZEaHiT"}
```

8.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:38 GMT
```

```
Content-Type: application/json; charset=UTF-8
...
Hmac: ec37341c7466ece4dabad5688db15d3cb08de5f19fd5bcaac780f7ac
      17e6c661
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:06:38.471 +0100
...

{"status":"DONE"}
```

9.
```
POST /cardSensitiveData/fetchDisplayPinResult HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD
           8842A54BBA5D6
...
Hmac: 74ea40f8dd5912225b331fefd3924fceb99c5780867a5db6adbd43fb
      1ef2d855
Content-Type: application/json
...

{"value":"AuLXVLPs.TpI6v39hnTPLyBfW1CFHDXhCXIhE36C5tbl.0sZsQaV
         yf7WlJEYDP6YEy_uJS9wleGyuEu4XJKwj3LYIg76qsFBFZSDJ4CTS
         Tvb2XnKWhHGLPMsCxZEaHiT"}
```

10.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:39 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: ae9f3173828e2fc140f98deb2a31d74d10e88d3fe706ebc19208c9a8
      09b998c8
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:06:39.106 +0100
...

{"responseCode":"OK",
"url":"6214875881699c06b0bf453a69a2226cbf57c3faef068db37260caab
      33f4193b828e1fa7289d4dca30899892360942fc88f63cbd65b1cc55d
      eaa22b8485219d7aea21ee06606d29da4ea2f9a69ec6c18b9060df276
      5be8b7563d489768060cba184ae90d1d393a39fb354c29d6b00696764
      950bbba890e6651de1c7c3a1cb87abc7662dcfac61e658dd05483e8c9
      f3e13d7c522960f0281725926cd0f4f1a34f8d65940020235c255c66c
      0046dce3a97973e2f4df6fc021bc9f202bded37bde40649cef87a319c
      687a3ed7df9dd2b9cd66cd2746b21b4c27952166b0ae72b67f27d31b7
      6c20b31ca7a4c1ae5d25003d1c5527fd1b9b315728447abd1b878bcc9
      9d2c8270834c707e1f0f70d31d17985eaa2acdb5f95977cde965abb84
      995836be33b465a598b2ce9b43d02dfe6edc83b904f6cb3e519b7c717
      8c53d63b4eba37fbde306076392e6692623e6187baaaae1b0e3063618
      0b426557c970df1f881af9073858db9a8d1dca01cf0e9abb91c2d8fc1
```

```
    4de65a1d7d896c44c29c7491cbcc3d33f731dc2b53d56c41424edc744
    2034a69992eea9de19f35c7073aae17763d09446e72b64f62380fc8a3
    63d0a28acac0cd611fb9950925f1459e96ddba467c774c53881b8252d
    6280bb914349542428fa6b0edd2d66cd5d8200c541d0b7c21e62dbd7c
    849197ab84736a3eeff5eacbc57306000564293ae4567143f375af21e
    baaccf60915bd850596829c9072efe476a4ce591fd525ac5bf91f061d
    264281a4625ac7981bb8fcfe88a1328e2597710924a592be227eac570
    14d9675b77b31"}
```

11.
```
GET /ibpin/pin?fiid=2402&requestId=P%2B1ZE90PkCW5ZtJjk3nRLnEVJj
    WLYavY16d7tJEwziOBujW5oA5%2FnlJ%2FfAaQL719lQYV5Acd2WGvwkNP9
    ...
    JcsTwuEqBN9WHMsX2dCQFctSLY%2FQwdmjRFjkJiEp2c%2Bw4kEy6UY%2BO
    %2FTnj%2B7FEIOxtkOAVrb7TU9oil%2B0xJx%2BnEYZbHspXa4tCsKZWd9T
    tw0nT5%2BfB0%2Bs1vPEVbRlMSGCH3K%2F9nvLtboZNel6I3IhXIh7MAkMX
    XrIVkpYdE9I8uw%3D%3D HTTP/1.1
Host: ibpin.ebs.gpe.cz
Accept-Encoding: gzip, deflate, br
User-Agent: okhttp/4.12.0
Connection: close
```

12.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:06:42 GMT
Server: GPE
Strict-Transport-Security: max-age=10886400
X-Frame-Options: SAMEORIGIN
Vary: Host
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Security-Policy: frame-ancestors 'self'
Connection: close
Content-Type: application/javascript; charset=UTF-8
Content-Language: en-US
Content-Length: 44

_jqjsp({"status": "success", "data":"****"})
```

View card info

This operation proceeds similarly as above. The only difference is that instead of the url in step 10. we obtain:

```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:07:14 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: 97b26dcca1cbd9ecba59ca0f7959ea47bc67775dc3fa229c00148792
      3833e493
```

```
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:07:14.441 +0100
...

{"responseCode":"OK",
"cardNumber":"a9aa61d22e493c29833f8e07b1bbb1771f5a1b266ea269ca
              4a54d1f564194c61",
"CVC2":"2dc11a9627a7af294e09f45751a0a1c8"}
```

In the response we can see that the card number and CVC2 are both encrypted. The client uses AES CBC with session secret to decrypt it, which is described in 3.1.5.

Set card limit

Changes of card limits also need to be authorized by the client. There are three options on how to set limits - 1) for a cash withdrawal, 2) a payment in a shop, 3) online payments. In requests we can see the following notation: 1) BASE_DB, 2) POS_DB, 3) INET_ONLY. The setting can be done temporarily or permanently. The choice is also visible in requests. The communication is similar as above, there is no encrypted part and there is are no additional information sent after the authorization.

Transaction

The transaction that sends money from the bank account of client to some chosen account also needs an authorization. The transaction approval proceeds the same as before. There is an example of our transaction, where we send 2Kč with the message for receiver "HOHOHOHOHO".

1.
```
POST /payment/v3/validatePaymentOrder HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A
1          2FD8842A54BBA5D6
...
Hmac: fdab043e67d52a4ec3364093d2bc2107096ad1747c5d96dd2d332
      f0cce3d478a
Content-Type: application/json
...

{"accountInfo":{"accountName":null,
"accountNumber":"*********",
"accountNumberPrefix":null,
"bankCode":"****",
"bankName":null},
"accountTrustLevel":null,
"amountInRequiredCurrency":2,
"confirmedWarnings":[],
"contraEnvelopeID":null,
```

```
"counterpartyId":*********,
"createRule":false,
"debitEnvelopeId":null,
"doorDocumentId":null,
"editableByUser":false,
"emailNotif":false,
"externalTransactionId":null,
"idCategory":null,
"immediateOrReject":false,
"isPayment2Contact":false,
"messageForReceiver":"HOHOHOHOHO",
"messageForSender":null,
"partnerCode":null,
"paymentService":null,
"priority":false,
"processInstantly":true,
"requiredCurrency":"CZK",
"requiredSignedConfirmation":false,
"sourceAccountId":****************,
"specificFields":{"constantSymbol":null,
                  "paymentReasonCode":null,
                  "paymentReasonText":null,
                  "specificSymbol":null,
                  "variableSymbol":null},
"templateId":null,
"transactionHmac":null,
"transactionType":"DTRNCLO",
"updatePaymentOrderId":null,
"validFrom":"2025-01-08T00:00:00+0100[Europe/Prague]"}
```

2.
```
    HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:18:10 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: 0c04cc3f00741ccc8c353f67b7a8c973ccf7532b0e81723ce
      21a67dedc56353f
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:18:10.166 +0100
...

{"passwordRequired":false,
"status":"OK",
"validationErrors":[],
"validationWarnings":[],
"id":null,
"accountTrustLevel":"UNKNOWN"}
```

3.
```
POST /payment/processPaymentOrder HTTP/1.1
Host: mb.airbank.cz
...
```

```
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F69
           8A12FD8842A54BBA5D6
...
Hmac: fdab043e67d52a4ec3364093d2bc2107096ad1747c5d96dd2d
      332f0cce3d478a
Content-Type: application/json
...

{"accountInfo":
    {"accountName":null,
    "accountNumber":"*********",
    "accountNumberPrefix":null,
    "bankCode":"****",
    "bankName":null},
"accountTrustLevel":null,
"amountInRequiredCurrency":2,
"confirmedWarnings":[],
"contraEnvelopeID":null,
"counterpartyId":*********,
"createRule":false,
"debitEnvelopeId":null,
"doorDocumentId":null,
"editableByUser":false,
"emailNotif":false,
"externalTransactionId":null,
"idCategory":null,
"immediateOrReject":false,
"isPayment2Contact":false,
"messageForReceiver":"HOHOHOHOHO",
"messageForSender":null,
"partnerCode":null,
"paymentService":null,
"priority":false,
"processInstantly":true,
"requiredCurrency":"CZK",
"requiredSignedConfirmation":false,
"sourceAccountId":*****************,
"specificFields":
    {"constantSymbol":null,
    "paymentReasonCode":null,
    "paymentReasonText":null,
    "specificSymbol":null,
    "variableSymbol":null},
"templateId":null,
"transactionHmac":null,
"transactionType":"DTRNCLO",
"updatePaymentOrderId":null,
"validFrom":"2025-01-08T00:00:00+0100[Europe/Prague]"}
```

4.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:18:14 GMT
```

```
Content-Type: application/json; charset=UTF-8
...
Hmac: eb2fc1b8e8741c4e3de2d75d239c3c032a9c56caac2ec8a0a
      936b696c712362d
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:18:14.131 +0100

{"allowedElementTypes":["SWT-ONLINE-CONFIRMATION"],
"authId":"o3ZcbI1nxtnrsByAnE9K5l1FtalEYaggt5HGaWmNDOFm3g
          jkk2kNiTccgXe0Ry0Na_k_WP7ITKfDZS1Kmf817GZspue1
          kutRuhWIvxW.i5y6LLGJgML1OuQ6lKTsmhb5",
"contractId":27732074,
"created":"2025-01-08T17:18:14+0100",
"expiration":"2025-01-08T17:21:14+0100",
...
"request":"<?xml version=\"1.0\" encoding=\"UTF-8\" stand
          alone=\"yes\"?><ns5:setPaymentOrderRequest xmln
          s:ns2=\"http://arbes.com/ib/core/ppf/ws/common/
          \" xmlns:ns4=\"http://airbank.cz/common/ws/faul
          t\" xmlns:ns3=\"http://arbes.com/ib/core/ppf/ws
          /obsAutentizationAutorizationWS/\" xmlns:ns6=\"
          http://arbes.com/ib/core/ppf/ws/obsStandingOrde
          rWS/\" xmlns:ns5=\"http://arbes.com/ib/core/ppf
          /ws/obsPaymentOrderWS/\" xmlns:ns8=\"http://arb
          es.com/ib/core/ppf/ws/obsRelationToBankWS/\" xm
          lns:ns7=\"http://arbes.com/ib/core/ppf/ws/obsEn
          ableCollectionWS/\" xmlns:ns13=\"http://airbank
          .cz/obs/ws/OBSUserNameWS\" xmlns:ns9=\"http://a
          rbes.com/ib/core/ppf/ws/obsSipoWS/\" xmlns:ns12
          =\"http://airbank.cz/obs/ws/SessionWS/\" xmlns:
          ns11=\"http://arbes.com/ib/core/ppf/ws/obsPayme
          ntTemplateWS/\" xmlns:ns10=\"http://arbes.com/i
          b/core/ppf/ws/obsContractWS/\" xmlns:ns16=\"htt
          p://airbank.cz/obs/ws/mortgageLoanWS\" xmlns:ns
          15=\"http://airbank.cz/obs/ws/loan\" xmlns:ns14
          =\"http://airbank.cz/obs/ws/obsLoanWS\"><paymen
          tOrder><ns2:validFrom>2025-01-08T00:00:00+01:00
          </ns2:validFrom><ns2:amountInRequiredCurrency>2
          </ns2:amountInRequiredCurrency><ns2:priority>fa
          lse</ns2:priority><ns2:specificFields xsi:type=
          \"ns2:CZSpecificFields\" xmlns:xsi=\"http://www
          .w3.org/2001/XMLSchema-instance\"/><ns2:require
          dCurrency>CZK</ns2:requiredCurrency><ns2:contra
          AccountNumber>*********</ns2:contraAccountNumbe
          r><ns2:contraBankCode>****</ns2:contraBankCode>
          <ns2:transactionType>DTRNCLO</ns2:transactionTy
          pe><ns2:counterPartyId>*********</ns2:counterPa
          rtyId><ns2:editableByUser>false</ns2:editableBy
          User><ns2:messageForReceiver>HOHOHOHOHO</ns2:me
          ssageForReceiver><ns2:emailNotif>false</ns2:ema
```

```
          ilNotif><ns2:idBankAccountDebit>********</ns2:i
          dBankAccountDebit><ns2:additionalInfo/><ns2:add
          ToWhitelist>false</ns2:addToWhitelist><ns2:acco
          untTrustLevel>UNKNOWN</ns2:accountTrustLevel><n
          s2:processInstantly>true</ns2:processInstantly>
          <ns2:isContraAccountInternal>false</ns2:isContr
          aAccountInternal></paymentOrder><validate>false
          </validate><immediateOrRejected>false</immediat
          eOrRejected><paymentOrderEventId>995336c5-44e0-
          42b8-9045-e82501a47ccb</paymentOrderEventId><au
          thLevel>ONE</authLevel><deferPayment>false</def
          erPayment><isPayment2Contact>false</isPayment2C
          ontact><isQuickPaymentOrder>false</isQuickPayme
          ntOrder></ns5:setPaymentOrderRequest>",
"requiresMobileLogin":false}
```

5.

```
POST /authorization/v2/attempt HTTP/1.1
Connection: close
Content-Type: application/vnd.airbank.mas.attemptRequest+json
Accept: application/vnd.airbank.mas.attemptResponse+json
logString: d0nUthbBInvHDMb1SXL5JeTCPJpu+IH/+YIkcSLwBHgbVr5dd+
           mUELTg8LYNZS5CJXCCLQ6tLawTELYvTHkoc8w207HLHXexrSNj
           +OqfQxfVqxn/YXdtLoskPmp2Cx59NxNJl2BLjs+3mZ+qOVkpPJ
           izPgS+P6ASHSFMPZ2Wq9S5X949M4n6Jlkenb06Snvw/WUKN9mq
           kMUBRBtt8AkQLOEijgVG2zvYFjQPc2LJXE26tl6ppwYYAHU7ct
           zpTHUIHH5bbiqfsztMghMLYs0TZxnH6380D0GtcwIf36qV+dwS
           5p1MdWsViefzdcczY9g+3lTpKv4uPBt4NqF0ah73tg==
movingFactor: 57878436
elementType: SWT_ONLINE_CONFIRMATION
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12F
           D8842A54BBA5D6
...
Hmac: bc1d20c1ae0cd40694216eb779fc0f4d5ef91ec6d71c1745556ff8f
      b0709c552
...
Host: mb.airbank.cz
Accept-Encoding: gzip, deflate, br

{"deviceHwId":"",
"authElementId":"4C4426365C34029FDB65EC49D43C231664023E75A9F6
                 98A12FD8842A54BBA5D6",
"elementType":"SWT-ONLINE-CONFIRMATION",
"authId":"o3ZcbI1nxtnrsByAnE9K5l1FtalEYaggt5HGaWmNDOFm3gjkk2k
          NiTccgXe0Ry0Na_k_WP7ITKfDZS1Kmf817GZspue1kutRuhWIvx
          W.i5y6LLGJgML10uQ6lKTsmhb5",
"attemptTo":"AUTHORIZE",
"deviceFlags":"AAAAAAAAAAA=",
"geoLocation":"",
"authOperationReqHash":"z5M3HnrHW9wPhVLlt\/134fwg8hJCRBpkL9G+
                        qlQzfr8=",
"authCode":"TVba7yd9650SgCwkNUyQMxybV1I4528bRm9sDdopXSQ="}
```

6.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:18:15 GMT
Content-Type: application/json;charset=UTF-8
...
Hmac: 239bf903765d23f494e04cac87d032fa81a87f49ea9e9975d427
      1f7be524740f
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:18:15.454 +0100
...

{"result":"OK"}
```

7.
```
POST /authorization/status HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
           FD8842A54BBA5D6
...
Hmac: 788d2ed3a9ffcc32122818990203a5538e2091dae614ecb0375797
      ae8a989e7c
Content-Type: application/json
...

{"authId":"o3ZcbI1nxtnrsByAnE9K5l1FtalEYaggt5HGaWmND0Fm3gjkk2
           kNiTccgXe0Ry0Na_k_WP7ITKfDZS1Kmf817GZspue1kutRuhWI
           vxW.i5y6LLGJgML10uQ6lKTsmhb5"}
```

8.
```
HTTP/1.1 200 OK
Date: Wed, 08 Jan 2025 16:18:19 GMT
Content-Type: application/json; charset=UTF-8
...
Hmac: dca45d5d33aec93d5d97b43e134ac1cd79b98651e97a5799aad375e
      a7ea3eda4
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-01-08:17:18:19.354 +0100
...

{"status":"DONE"}
```

Request details - Change of OCRA_secret

Here we provide an exmaple of the captured communication while changing the value of OCRA_secret (after the change of password):
CLIENT:

```
POST /authorization/initSWToken HTTP/1.1
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD88
```

```
           42A54BBA5D6
...
Hmac: 3d523ab963e43084907915a93fca8c4cfd876ae6bb4db693f6600c756e
      f7b650
...
Host: mb.airbank.cz
Accept-Encoding: gzip, deflate, br

{"publicKey":"MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAxz53QV
              R+T4c4KOOOtjW1aQB7U0JCP1WzX30Xn65ic61o8Z2oMdwQKAmN
              hEo1rhH1Ttk2IkglYUq8dC67kUECSLSBGwfoeZ68oVRrh4SGX0
              80CsX33KmjhWcXClpugUinXcy+MFK\/RVz56rfg0t0DICC2YP5
              s7JLUVN52RvHFbaeVDeDCs1MSKHjR9djwZbc1jFbgyPVCHAdh0
              6Igs3zFDE7t1k+uOhpm+VM0OrHjFwlcBBtxx3I+umoG\/CfRG1
              JLfnMAPuv5HQyv7y1ih2cKr+agQfBBi5exip3Sd\/Olu8eYimq
              6\/Ml3a2FNdlPixqHFoilRGWNf8xjbAPiBY7SK4wIDAQAB",
"previewMode":false}
```

SERVER:

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:29 GMT
...
Hmac: b2ba296fef4e0e9b8cd5758c597691608fb7936360cdfdabaa71da61842
      bb9bb
...

{"encSecret":"faKTgVpiuv4pc8bf9hPXKhqUjkCsK3W+q/FJBvxfhMj13QkRwuo
              IubRWAluUpFzWupf2Q0UK+fAWEgqKrBe0X2kgtVHBh7MwmfCSR8
              35eXY559RXK0U4zxtU2A+W3+BPFC+5ne3HBZkJpW90ST6eSpd/p
              FeK/ilk/fL8yO5pyc+mgLDntbnhBxF/XL8rPGQaGMzvWFbGBfQP
              WXROSdZS/MbkfXEnMMbWDC+75ABdgPyurG/dvNrvjkL6f/iun/F
              Zx7dw2HOSoQqhCyI+V2Tsn98nRj2sxPCj6xmUcj9HmDYjb1FFVL
              m2bGYSrvkojpF2NchIFVfGcoecanrA1vfcSA==",
"encSecretConf":"uPL4kcjGEQPm+j8ROVHw4oa1hCvggo8oKSzdlj0g5ns2VhHz
                 gKeXun6WDl4lpDp9dsx5tBaSrqSAortxTJp10jsE2B6VoG/m
                 E3BlzBz090kf5RU3LaXhtzXKO+Y8GtALI3DJXb7EwxCmDCbZ
                 V0S1e882kLvVGQx8bShl/ML2HopyyPpV6jvtG//SFgIef+Fj
                 AlYGxKeIntszLtVGYHCPlei6hbXqHuJLbTm8yAKmtCh7VR0T
                 AGkj1CTZRvy6rLMMYsac0b70+wV+cVS8Tthp5dAo5Nyl6YBq
                 bgzSWY5v92wJFL6UZZw0VA+8ZWR+ordV0rvaXrxRfoIDoN15
                 ECoNIg==",
"madePrimary":false}
```

Conclusion - differences and security

The implementation in application differs from its specification in RFC [42] in several details - the output of HMAC in OCRA is not truncated to 6 digits. The implementation uses HMAC-SHA256 instead of HMAC-SHA-1. Both changes are acceptable and secure as there is no need to rewrite the code manually and using

SHA256 instead of SHA1 is recommended, see for example recommendation by BSI [28].

We compare the implementation with some of implemetation considerations given in RFC 6287 [42]. Implementation considerations are denoted by IC. We have chosen considerations that are related to the algorithm used by the client.

- IC1 - During the computation, the client uses master_secret (encrypted/decrypted by PIN, which is something the user knows) which follows IC1. All messages are also signed by HMAC.
- IC2 - OCRA_secret has the length 32 bytes - that is also the length of the output of CryptoFunction HOTP-SHA256-6, because the application does not truncate the output to 6 nibbles (half-bytes). Thus IC2 is satisfied.
- IC4 - The output of Create_data_to_sign that can be considered as a challenge question is 32 bytes long. This corresponds to OCRASuite, where is said that the challenge question is hexadecimal and 64 nibbles long. Thus it satisfies IC4, which recommends 20-byte value and requires to have the length at least t, where t denotes the length of truncation output. Since we do not truncate the output, the challenge question is 32 bytes long which is the same as the signature output.
- IC8 - It is recommended to use a session indentifier as an additional input in the computation. The application does not use it in computation. However, it is signed by HMAC with the key corresponding to the current session.
- IC9 - The implementation uses time, but it does not use a counter. Using counter is optional. This could, theoretically, lead to replay attacks (RFC 6287 IC9), however, the application uses authId. This Id is unique for every authorization of operation and it is delivered with the request challenge. We reckon that, if everything works correctly, it could be equivalent to the counter.

  The change of OCRA_secret is done using RSA_ECB_PKCS1 padding. Firstly, the generated modulus is 2048 bits long, which is, as we said above, shorter than the generally recommended value, which is at least 3000 bits. Secondly, the padding PKCS1 of version 1.5 is generally not recommended, see for example recommendations by BSI [28], In this specific case it is not an issue. The value is encrypted by the server and sent to client only once and it is signed by HMAC, thus attacks like, for example, the most famous one described in the paper by Bleichenbacher [55] are not applicable to it. However, it is recommended not to use this version of the padding. The application could use for instance the OAEP padding instead. For more information about OAEP we refer to the section 9.3.8 in [24].

  For storing master_secret the application uses PBKDF2 with 1000 iterations, PIN which has at least 6 decimal values and master_salt. The value of 1000 iterations is from RFC 2898 [44], nowadays the recommended number of iterations is higher. OWASP [56] recommends at least 600 000 iterations for storing of the password.

We note that even if the number of operations is not as high as it is recommended, in this case it is not an issue, because the attack would require a strong attacker. The attacker would have to be able to obtain master_salt and encrypted master_secret from the device. He would also have to obtain OCRA_secret and OCRA_salt from shared preferences. Then he would have to obtain some communication in order to verify the value of PIN, resp. decrypted master_secret. That is, the attacker does not need to actively communicate with the server in order to bruteforce PIN.

The length of salt in PBKDF2 should be at least 128 bits according to NIST SP 800-132 [57], which is satisfied.

OCRA_secret is stored in shared preferences using AES with CBC mode. The application uses the same value of IV, that is zero vector, for every encryption, thus IV is predictable. This implementation does not meet the requirements given in NIST SP 800 38A [30]. On the other hand, in this case, the application encrypts a random plaintext by a random key and uses zero IV only once, which seems to resist known attacks. However, in general we recommend to follow the specification.

The encryption does not use any padding since the length of the secret is 32 bytes, that is, 2 blocks of 128 bits.

3.1.7 Registration

By the registration we mean the registration of a new device, that is, linking the new device (installation) with the bank account.

High-level overview

In this section we provide a brief overview of the registration for better understanding.

Firstly, the server needs the user to authenticate. The registration of the device starts by sending the registration code by SMS to the user. He enters the code into the application and then is asked to turn on his camera and follow the dot on the display with his eyes. The application makes photos of the user while moving the dot. After that, photos are sent to the server. The server verifies photos. We assume it checks the user, comparing his photos with some photo of the user it already has. If the user looks the same/similar and he received the SMS messsage sent to his mobile number, then he is authenticated and the registration can proceed.

The server needs a username (that is, Device-Id) and a verifier of the user password (master_secret) to register the device and to perform a login. The verifier is a sensitive value, an attacker with the verifier could pretend to be the server. In order to obtain a shared secret key that can be used for the encryption of the verifier, the application uses SRP protocol.

The server sends temporary values to the client. The client uses these values and the first part of the SMS code to derive a temporary password (temporary master_secret) and temporary Device-Id. The sever and the client start SRP protocol to derive a stronger shared value. The shared value is used to derive

HMAC keys and AES key and IV. From now on, both, the client and the server, use HMAC to sign their messages.

Then the server sends to the client new (permanent) Device-Id and new SRP_salt. The client generates a new long password, computes the corresponding verifier and encrypts it by AES using the derived key. Subsequently, the client sends the verifier to the server.

Then everything is prepared for the login using SRP since the server has the DeviceId, verifier and SRP_salt. The login proceeds as described in 3.1.4.

The registration is done. The last step is to obtain OCRA_secret, which will the client use to authorize operations. This part proceeds as described in 3.1.6.

Notation

A notation is similar to the notation in SRP 2.5.

Device-Id	...	Device-Id is a hexadecimal identifier of a specific user's device with the installed application given by the server
I (= Device-Id)	...	username in SRP protocol, I = Device-Id
regId	...	temporary Device-Id, we also use it as a username I in SRP protocol
master_secret	...	the password which is used in SRP protocol, see 2.5 and 3.1.4
master_salt	...	salt that is used in encryption of master_secret, see 3.1.6 and 3.1.8
SRP_salt	...	salt used in SRP protocol
regSecret	...	temporary master_secret, temporary password in SRP protocol
AuthId		unique Id of the authorized operation given by the server, in this case, the operation is the registration
g	...	generator
N	...	modulus
H	...	hash function SHA256
PAD	...	padding by '0' bytes from the left to get the length 256 bytes
$A, B, a, b, M1, M2, K$	...	values from the SRP protocol described in 2.5

Scheme of communication

In this subsection we provide a brief overview of the communication. The communication can be divided into three and one additional phase.

- Start of the registration

- Pairing
- Login
- Setting of OCRA_secret

The first phase:

CLIENT

SERVER

Start of the registration

I want to register a new device.

The user's phone number and his date of birth

AuthId, positions of the dot

SMS with a code to the received phone number

User's phone

photos of the user following the dot (at received positions) with his eyes on the screen of the mobile, AuthId

imageUuid for every received photo

I would like to authorize.

The second part of received SMS code, imageUuids, dot positions, AuthId

OK

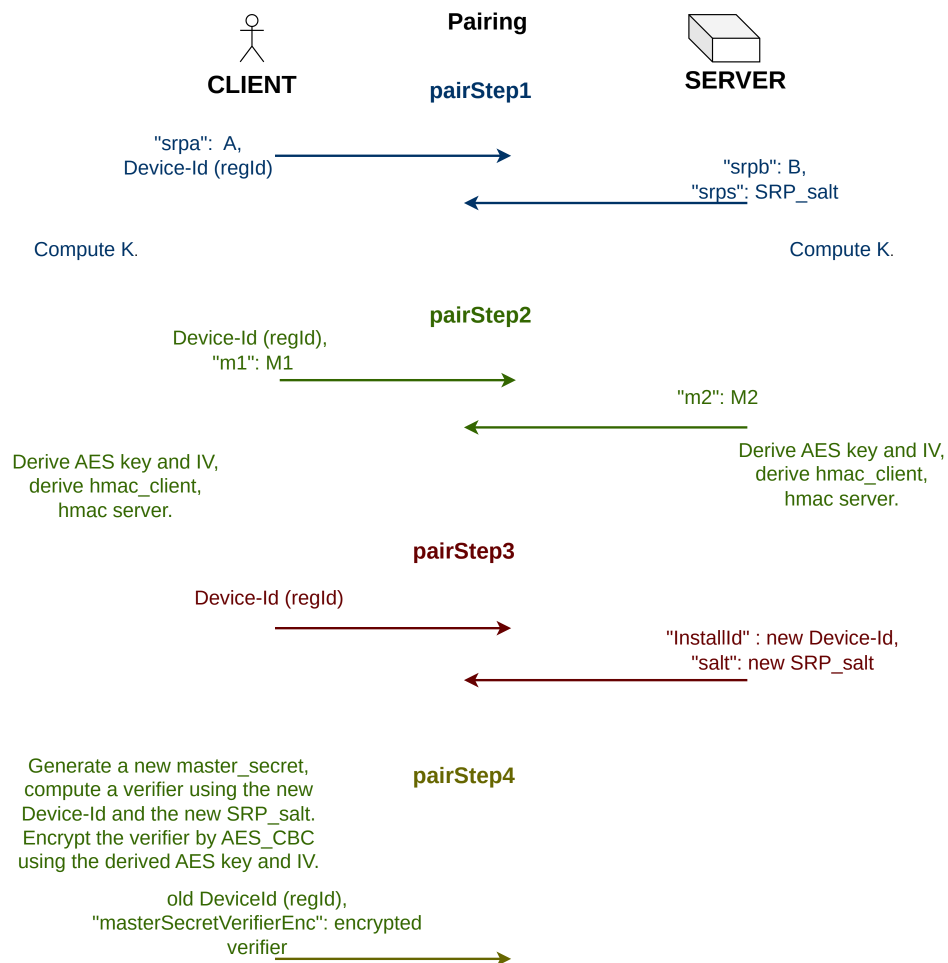
AuthId

regId, regSecretEnc

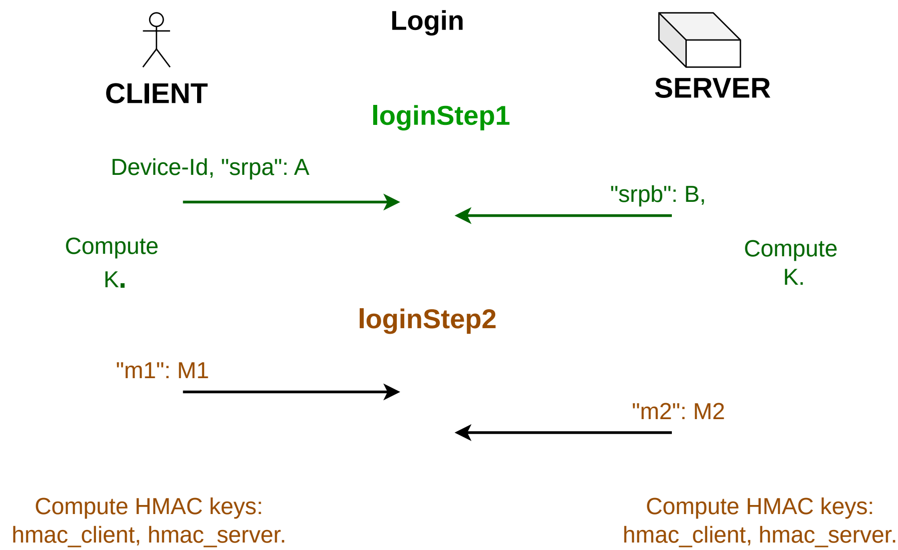
Use the first part of the SMS code to decrypt regSecret.

Then, in PairStep1, Device-Id = regId and password master_secret = regSecret.

The second phase:



The third phase:



The last phase is the setting of the new OCRA_secret and OCRA_salt. The process is the same as in the 3.1.6 while changing the OCRA_secret.

Implementation

Most of the registration is implemented in the Security Module (see the next chapter) in functions registrationStep1, registrationStep2, registrationStep3, registrationStep4, registrationFinalize, loginStep1, loginStep2, loginStep3.

We implemented the registration in the Python programming language in `registration.py`.

Description and pseudocode

Most of the first part is pretty much straightforward, because there is almost no cryptography.

Algorithm 24 Start of the registration

CLIENT: Send user's date of birth, phone number and deviceName entered by the user during the registration.
SERVER: Send AuthId, positions of the dot and minimum number of valid segments to the client.
USER: Receive an SMS message with the code in the following format: NNNNNN - MMMMMM, where N,M $\in \{0, \ldots, 9\}$.
USER: Turn on the camera and follow the dot on the screen with your eyes (the dot is moving according to received positions).
CLIENT: Capture photos of the user following the dot. Send AuthId and photos one by one to the server.
SERVER: Send "imageUuid" (id number) for every received photo one by one to the client.
CLIENT: Try to authorize - send AuthId, the second part of the SMS code (MMMMMM) and dot positions with imageUuids corresponding to photos.
SERVER: If everything works correctly, respond with "OK".
CLIENT: Send AuthId.
SERVER: Send regId (6 characters - 0-9A-Z), regSecretEnc (10 characters - 0-9A-Z) to the client.

CLIENT: Decrypt regSecretEnc using the first part of the SMS code (NNNNNN) and the function described below:
decrypted_regSecret = Decrypt_regSecretEnc(regSecretEnc,NNNNNN)
CLIENT: Set Device-Id = regId,
master_password = decrypted_regSecret

The function that decrypts regSecret:

Algorithm 25 Decrypt_regSecretEnc

Let INT() denote the tranformation of the input with the base 36 into the integer with the base 10.
Let TO36S() denote the tranformation of the input into the integer with the base 36 (instead of decimal or hexadecimal) and then to the string.
INPUT: regSecretEnc, key (the first part of the SMS code)
OUTPUT: decrypted regSecret

1: key = transform key to bytes
2: hashed_key=SHA256(key)
3: hashed_key = TO36S(hashed_key)
4: regSecret = new array of 10 characters
5: $i = 0$
6: **while** $i < 10$ **do**
7: $j = i + 1$
8: regSecret[i]=TO36S((INT(regSecretEnc[i]) + 36 - INT(hashed_key[j])) mod 36)
9: **end while**

We note that the minimum number of valid segments denotes the number of

required photos. We will see that it is equal to 5 in the communication. The second part of the algorithm is similar to the SRP protocol. We describe both, a client part and a server part. However, we have the code of the client side, not of the server, thus sometimes we can only guess what the server does.

Algorithm 26 Pairing

INPUT: I = Device-Id = regId, p = master_secret = regSecret

pairStep1

CLIENT: Generate a, send $A = g^a \bmod N$ to the SERVER. Also send I = Device-Id and information about the mobile device, for example list of installed applications.
SERVER: Take verifier v that corresponds to Device-Id, generate temporary SRP_salt, generate random b, compute $B = kv + g^b \bmod N$, where $k = \mathrm{H}(N||\mathrm{PAD}(g))$. Send B and SRP_salt to the CLIENT.
(Both) SERVER and **CLIENT**: Compute u = H(PAD(A)||PAD(B)).
CLIENT: $x = \mathrm{H}(s||\mathrm{H}(I||":"||p))$
CLIENT: $S_C = (B - k \cdot g^x)^{(a+ux)} \bmod N$, where $k = \mathrm{H}(N||\mathrm{PAD}(g))$.
CLIENT: $K_C = \mathrm{H}(PAD(S_C))$
SERVER: $S_S = (A \cdot v^u)^b \bmod N$
SERVER: $K_S = \mathrm{H}(PAD(S_S))$

pairStep2

CLIENT:
$M1 = \mathrm{H}(\mathrm{H}(N) \text{ XOR } \mathrm{H}(\mathrm{PAD}(g)))||\mathrm{H}(I)||\mathrm{SRP_salt}||\mathrm{PAD}(A)||\mathrm{PAD}(B)||K_C)$
CLIENT: Send $M1$ to the SERVER.
SERVER: Compute and validate $M1$ using K_S. If $M1$ is correct, compute $M2 = \mathrm{H}(PAD(A)||M1||K_S)$.
SERVER: Send $M2$ to the CLIENT.
CLIENT: Check $M2$ using K_C.
Let us denote $K_C = K_S$ by K.
SERVER and **CLIENT** (both): Compute
AES_key=HMAC-SHA256(K, 'aes_cbc_key' in bytes),
AES_IV=HMAC-SHA256(K, 'aes_cbc_iv' in bytes),
server_hmac_key=HMAC-SHA256(K, 'server_hmac_tmpkey' in bytes),
client_hmac_key=HMAC-SHA256(K, 'client_hmac_tmpkey' in bytes).

pairStep3

CLIENT: Send Device-Id (regId) to the SERVER. Sign message by HMAC using client_hmac_key.
SERVER: Create and send "InstallId": new Device-Id and "salt": new SRP_salt to the CLIENT. Sign message by HMAC using server_hmac_key.

pairStep4

CLIENT: Generate a new master_secret and a master_salt.
CLIENT: Compute x = H(SRP_salt||H(Device-Id||":"||master_secret)).
CLIENT: Compute a verifier $v = g^x \bmod N$.
CLIENT: encrypted_v=AES_CBC_PKCS7_encrypt(AES_key, AES_iv, v)
CLIENT: Send encrypted_v as "masterSecretVerifierEnc" to the SERVER. Sign message by HMAC using client_hmac_key.
SERVER: Server can decrypt the verifier using derived AES_key and AES_IV.

After the pairing part the server has the verifier and SRP_salt and the client has master_secret, which is encrypted by PIN as described in 3.1.6 and 3.1.8. The client also has Device-Id, therefore he can log in. It seems from the decompiled code that the condition on the nonzero B is checked during the pairing.

The login part is already described in 3.1.4. It includes loginStep1 and loginStep2.

After the user is logged in, he receives a new OCRA_secret as described in 3.1.6.

The user is then prompted to choose if the device will be used for the transactions approval. If the answer is yes, the operation is authorized using OCRA as described above in 3.1.6. We mention it only because it is the part of the registration in the application, but it is not part of the registration process, the user can decide that he does not want to perform the operation.

Storage and parameters

Photos are made by the user during the registration in the application. The SMS code is sent to user's mobile using the number he had already registered before.

In the pairing and login phase, we need a generator g and a modulus N. They are hardwired into the code and we have already seen them in 3.1.4.

The values of Device-Id and master_secret are firstly only temporary, in the pairing phase we receive a new permanent Device-Id for our device and we create a new permanent master_secret. All remaining parameters are created during the registration or receiverd from the server.

We note that the private exponent and temporary parameters as SRP_salt or master_secret in pairStep1 and pairStep2 are shorter than usual in SRP protocol. For comparsion, we provide lengths:

- temporary Device-Id (regId) ... 6 numbers in base-36 (6 bytes using ASCII)
- Device-Id ... 32 bytes
- temporary master_secret (regSecret) ... 10 numbers in base-36 (10 bytes using ASCII), it can be computed using 6 digits
- master_secret ... 32 bytes
- temporary SRP_salt ... 16 bytes
- SRP_salt ... 32 bytes
- private exponent a of the client in pairStep1 ... 32 bytes
- privete exponent a of the client in loginStep1 ... 256 bytes

The master_secret is stored encrypted using PIN as we have already said:

$$\text{encrypted_master_secret} = \\ = \text{PBKDF2(PIN,master_salt,1000,32) XOR master_secret.}$$

Request details

In this subsection we provide a caputured communication.

1) Start of the registration

For the sake of readability, we do not include full images. Images are in jpeg format and encoded in base64.

```
POST /security/register HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
..

{"activeOperations":true,
"birthDay":"****-**-**T00:00:00+0200[Europe/Prague]",
"callingCode":"+420",
"deviceName":"pixel",
"initialPreviewMode":false,
"phoneNumber":"*********"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:34:17 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:34:17.535 +0100
...

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj",
"activeCheck":{"minValidSegmentCount":5,
               "segments":[{"dotPosition":"BOTTOM_LEFT"},
                           {"dotPosition":"BOTTOM_RIGHT"},
                           {"dotPosition":"TOP_LEFT"},
                           {"dotPosition":"BOTTOM_LEFT"},
                           {"dotPosition":"TOP_LEFT"},
                           {"dotPosition":"BOTTOM_LEFT"}]}}
```

2) Start of the registrarion - *photos*

```
POST /public/facePictureUpload HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj",
```

```
"biometryConsentTimestamp":
                "2025-02-12T16:35:06.774+0100[Europe/Prague]",
"image":"/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoSUNDX1BST0ZJTEUAAQEAAAI
         YAAAAAAIQAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAAABhY3NwAAAAAA
         AAA...vZRomlUa+TDYzW5JK54lrOY714bOKjf8AuZZj39klyuPav//Z"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:07 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:07.348 +0100
...

{"imageUuid":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55d2ee0
              fc80597633c6f5ea4efeb1818dc3062984f24bb02591245a4d9
              5884f85b63d0830c180233d3fed31ace3f5bb5026ae4f23193f
              549ff51"}
```

```
POST /public/facePictureUpload HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj",
"biometryConsentTimestamp":
                "2025-02-12T16:35:06.800+0100[Europe/Prague]",
"image":"/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoSUNDX1BST0ZJTEUAAQEAAAI
         YAAAAAAIQAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAAABhY3NwAAAAAA
         AAA...MO2NgEbSJHdRWcZxJFY0jhH03B8teWxIOt+yi8+WN27TKR//Z"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:08 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:08.505 +0100
...

{"imageUuid":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55d2ee0
              fc80597633c6f85efe25e6762c9090d29a83027521b57cc0dce
              15f51076438833277cb02c3c81b6f84482d0df3445bb89d1f81
              339742d"}
```

```
POST /public/facePictureUpload HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...
```

```
{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJKvI
           lRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZkagjRM
           bssqKXgCAxk.HRyyjj",
"biometryConsentTimestamp":
                "2025-02-12T16:35:06.813+0100[Europe/Prague]",
"image":"/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoSUNDX1BST0ZJTEUAAQEAAAIYA
        AAAAAIQAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAABhY3NwAAAAAAAAAAA
        AAA...nGnYui1pw1AVKZOQwZMeXqkpeIGtWwE4V9MODc3iHokudjcr/9k="}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:10 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:10.269 +0100

{"imageUuid":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55d2ee0
              fc80597633c6fd511885f88f0a889080decf29f4548dee28cec
              e24b7e6908fa98019c1ca16ea2052dff43f2b2b412e0ea29a5e
              f82e557"}
```

```
POST /public/facePictureUpload HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj",
"biometryConsentTimestamp":
                "2025-02-12T16:35:06.826+0100[Europe/Prague]",
"image":"/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoSUNDX1BST0ZJTEUAAQEAAAI
         YAAAAAAIQAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAABhY3NwAAAAAA
         AAA...qFuhdMuA99klNx5mBfSZVxTWhwOYOBh3W6S43Ht3mXT//2Q=="}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:11 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:11.589 +0100
...

{"imageUuid":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55d2ee0
              fc80597633c6fd149293ac8f8ab6be1754b08d1fcbcf70087aa
              e5413abcb4ec37b86f2bb6efa8ec5357cc5813e25dfdee945bc
              ab48d2a"}
```

```
POST /public/facePictureUpload HTTP/1.1
Host: mb.airbank.cz
```

```
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj",
"biometryConsentTimestamp":
                "2025-02-12T16:35:06.839+0100[Europe/Prague]",
"image":"/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoSUNDX1BST0ZJTEUAAQEAAAI
         YAAAAAAIQAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAAABhY3NwAAAAAA
         AAA...p5Fa+00apcPZkc0Ew11sv1SVqqRTptc2Gw6SNwNwksfZdV//Z"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:14 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:14.364 +0100
...

{"imageUuid":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55d2ee0
              fc80597633c6f8d1fade2b5aeb8d6e1f1c73b1091d4736611cb
              18f73026719b5a02c94f8005694fa19aaa0870869723e296b36
              dbfe79c"}
```

3) Start of the registrarion - ***authorization using SMS code and photos***

```
POST /public/multistepAttempt HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: notPairedDevice_9fdb80fa73704ff
...

{"attemptTo":"AUTHORIZE",
"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
          vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
          gjRMbssqKXgCAxk.HRyyjj",
"authOperationReqHash":null,
"deviceFlags":null,
"deviceHwId":null,
"geoLocation":null,
"smsEncrypted":null,
"elementTypes":
    [{"authCode":"105826","authElement":"SMS"},
     {"authCode":"d03c6f9e48a351ef23d97ff862ab605595acb804e49b55
                  d2ee0fc80597633c6f5ea4efeb1818dc3062984f24bb02
                  591245a4d95884f85b63d0830c180233d3fed31ace3f5b
                  b5026ae4f23193f549ff51|BOTTOM_LEFT|
                  |d03c6f9e48a351ef23d97ff862ab605595acb804e49b5
                  5d2ee0fc80597633c6f85efe25e6762c9090d29a830275
                  21b57cc0dce15f51076438833277cb02c3c81b6f84482d
```

```
                  0df3445bb89d1f81339742d|BOTTOM_RIGHT|
                  |d03c6f9e48a351ef23d97ff862ab605595acb804e49b5
                  5d2ee0fc80597633c6fd511885f88f0a889080decf29f4
                  548dee28cece24b7e6908fa98019c1ca16ea2052dff43f
                  2b2b412e0ea29a5ef82e557|TOP_LEFT|
                  |d03c6f9e48a351ef23d97ff862ab605595acb804e49b5
                  5d2ee0fc80597633c6fd149293ac8f8ab6be1754b08d1f
                  cbcf70087aae5413abcb4ec37b86f2bb6efa8ec5357cc5
                  813e25dfdee945bcab48d2a|BOTTOM_LEFT|
                  |d03c6f9e48a351ef23d97ff862ab605595acb804e49b5
                  5d2ee0fc80597633c6f8d1fade2b5aeb8d6e1f1c73b109
                  1d4736611cb18f73026719b5a02c94f8005694fa19aaa0
                  870869723e296b36dbfe79c|TOP_LEFT",
        "authElement":"FACE_BIOMETRY"}]}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:17 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:14.982 +0100
...

{"result":"OK"}
```

```
POST /security/fetchRegisterResult HTTP/1.1
Host: mb.airbank.cz
X-Process-Id: prod-MA_ANDROID-20250212163521-877699
User-Agent: Air Bank/10.1.0 (phone; android; 31; Pixel 3 XL; 1440x2621)
Device-Id: notPairedDevice_9fdb80fa73704ff
Content-Type: application/json;charset=UTF-8
Content-Length: 141
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

{"authId":"YPRs2W3wfniivMgxStkhNohH1ZaBhfCNwtNlq1GpcgzY8pCzb5DJK
           vIlRIrCBs8DlMQmzKJOEaJbXotOmememsTqdHNS2vTIutCb2dWZka
           gjRMbssqKXgCAxk.HRyyjj"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:21 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:21.849 +0100
...

{"regId":"PO2LEK",
"regSecretEnc":"4EIR9WMSDM"}
```

4) Pairing - *pairStep1*

```
POST /security/pairStep1 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: PO2LEK
...

{"appVersion":"A.10.1.0",
"deviceType":"Pixel 3 XL",
"os":"Android 12",
"srpa":"a9781f0cc75595e50a29d491d102be1d471a71086ca67fc780875f6b5a
        c2559a25f6e3d43743a4d0043d3cecfe1dfc31995ffa7f2a5e2890a69a
        8addbac765c114a80aee4a11775053d9aea112812f69144b09e5731e13
        bbea4250cf0e140d98958864be0d57c4a9a930899c56cd36126fef2265
        a65d47758dbbb6bafd45ac7fed85c73061d5b847273cd5a403e3041544
        6fa8506084131fbf01f782193a34354c18fec32bbd721a70f1f05aedca
        11596ba70fe299a2493d697d8c136cc7c94ac732e1e177f5500e8caa8f
        6fff30364155b28daece8566c3ce207e6675dd6b773da3602e46ea2475
        54ad0a9537ac54e2fb24a110335ce84c26c49126eeb37000",
"advancedDeviceIdentification":
        {"androidGSFID":"****************",
        "androidHWFingerPrint":"********************************",
        "androidID":"****************",
        "deviceProducer":"Google",
        "deviceUserName":"Pixel 3 XL",
        "installedApplications":
            "com.android.internal.display.cutout.emulation.noCutout,
             ...
             com.google.android.apps.restore",
        "iosIdentifierForVendor":null,
        "languageOS":"cs",
        "screenResolution":"1440x2621"},
"platform":"ANDROID"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:33 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:32.110 +0100
...

{"srpb":"78067bc97400ec8eb9cc2a226c860f69572dff37d0300c232d65e4fbd
         8fbd747db8057b41fc2d017e0ab93631a4e41541fdd20c8535f982295
         a33ddb10cad39de9595e66bc78170a14b72d519448f27254ad9ddb1e2
         af6df0a8f22b2d0fbc6fceea0bb46c166defad0b73c041d99f7964624
         edff5a6d9fe21ae1d63f90228824b98eaa49dc1df3732808a6be6fcfe
         76d88ed4573dcd7c093834435f3e998624a06002a1e4f43d705a2722d
         32e62f6405c70f6233fbe9284192af1b10afd925728d39e6cce1fc4d5
         c5e8f0abad501df737dc0ec45b1f6d76ea72fe684e30a50f37ec76629
         9254301fe550a8b5339016338b3222424d5177fcf188a4dd3667f79e",
```

```
"srps":"D609E21A09F3BB0C3D3749123A2FF20D"}
```

4) Pairing - *pairStep2*

```
POST /security/pairStep2 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: PO2LEK
...

{"m1":"80349306d21b2fc8514c6a705ec470a925f2ab306018f8f6a4e11343cb12
       a260"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:34 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:34.466 +0100
...

{"m2":"8b2e0605d9d836de9e907e4da05c9b8ece109b90b89f5308ecb759712cac
       24ec"}
```

5) Pairing - *pairStep3*

```
POST /security/pairStep3 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: PO2LEK
Hmac: 354185f1da351226cdbf053ffbaf70eb6a8cca5146c83128fbdc245ec7de0708
...
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:43 GMT
...
Hmac: d4e27c489969d009df0d9047aa7154e0d2c5d47edfe7a9412e09fdc59ad68336
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:43.676 +0100
...

{"installId":"3E19DD1D38CB1F6191993A8458F7C4FAC43CD713CA80B861EE760896
              6B1BCC37",
"salt":"647e499610facb65bd4d911222caf916c62ff3c3a542240b9f7f2d018d3b47
        00"}
```

6) Pairing - *pairStep4*

```
POST /security/pairStep4 HTTP/1.1
Host: mb.airbank.cz
```

```
Device-Id: PO2LEK
Hmac: fa53820e9378adfc417d82c5e2da24da7a45f1e790c0a325c5633a0f8f4d04b8
...

{"masterSecretVerifierEnc":"ec189834e53c36df33fd61607a5e41cb6cb5df7bb1
                            921fec707ba5472aac7eafaafb3eeb5b0b86d1a38c
                            baa56ddca354ea37f86f17ffc33f5af617ad7ea17f
                            d5f2779f7c53c96c6857339a03d494623c527f5a54
                            0c5b4cc503e77e70fdd2a16c0dfe33b114bef62060
                            860ec81ad5546c4b3a95dd53f8b1d5cdb449f2f4b7
                            43bfda6237c422e0dd052d19157837c977c0957fcb
                            1ffeb2898fd82cb9a3621082fcd213976e246e9185
                            f7f51b3751f784b8555784a3b9e36a13a6a117de6f
                            70f47515ae21ad4d846c8bb88422198caa8210b60f
                            7d7699f05a1a116b7b1c607a21b0e0192d0994102b
                            b92a29b37e8656ab09f9f60795d4fe33077df3b4d3
                            d4b231113b51725ad8f2281553cea5184a067c63"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:45 GMT
...
Hmac: 80d2afd7a618bbe9aceb0987cabb1e2b2470a7d9dcfe183b6007b7f7cc380482
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:44.988 +0100
...
```

7) Login - *loginStep1*

```
POST /security/loginStep1 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 3E19DD1D38CB1F6191993A8458F7C4FAC43CD713CA80B861EE760896
           6B1BCC37
...

{"appVersion":"A.10.1.0",
"deviceType":"Pixel 3 XL",
"os":"Android 12",
"srpa":"a5b128c7a70a60f4c873cdc6c31b84bfaacf93e423ab4aa4c90e3c5a97ae
        5e0977e5678f283a0f4cf2fd936d1a994843a2b9ceee08a7c62eb8f3ef73
        277a260541ce1443bae310170da09e43981006578801d447995fe7e59fe0
        83d4e5025e3e50d4e7bdab44305dc3659fd4863655ddbea954c18bd33ee2
        272c32d3bc96bd84b8bff58c5020466b2a19a7c6cd6b6228fdc94bf3494c
        1c030ac66ef3ff5fa9b7faf5939e17aed36f83fd57641b37255b01e7b7aa
        61447edbdb13860d13b8013c36d3f5d53e49f68a267d1db42113d80b6d0e
        ca10fb5f8688af3692d78f49810d4e04fa755c5bb8bf15e8358eb049df98
        53e1162cc37bf0ca75109b2b4a8d8faa",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:46 GMT
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:46.184 +0100
...

{"srpb":"178ebe1e34907e743b39fe31277480ab5d61be5327db9ca053d6ed69c94
         ba5e17f83b0fedb1dc77d6cb72f9398c224b1f3b56745554695a80e304f
         86ea0765946d605ec92fcb8eddbcdbe0f48c381049eb1a1e82c43afaca0
         58375a63ffe0f87c8e9181668ea5fb7a48c15d7fb81227aabca4e211d9b
         3c1ce4e7cf6b41a99161912f083ebac922e18c50535c388117a932b1432
         0e6a426a80b0ad9e0f0e17cb0cfb849f109063cd24b7a6b19c9dbb216cd
         ccf5ceafe51580b98f214b1d8d77f5be7359293e2c73fcda811c02a9cae
         fee6cd0c86342bf688154e3b2df9bb5f7070585cc512200f9d47c2e2ad1
         468e419159340139cbae567ae5d1a78af31b6df6",
"srps":"647e499610facb65bd4d911222caf916c62ff3c3a542240b9f7f2d018d3b
        4700"}
```

8) Login - *loginStep2*

```
POST /security/v4/loginStep2 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 3E19DD1D38CB1F6191993A8458F7C4FAC43CD713CA80B861EE7608966
           B1BCC37
...

{"deviceType":"Pixel 3 XL",
"m1":"d7b28b54a728d3161742a8b9bf000aa4370685f93d116efd1e7d13ea4a857e
      61",
"os":"Android 12",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Wed, 12 Feb 2025 15:35:47 GMT
...
Hmac: 6c2562794c7a2d75b3cb3e9b8887ba502cbce2a34ed9c0dec6157e4ed33f
      d009
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-12:16:35:46.984 +0100
...

{"applicantRef":null,
"contracts":[{"contractNumber":"*******",
              "firstName":"Karolína",
              "id":********,
              "lastName":"Zenknerová",
              "profileId":********,
```

```
              "relation":"OWNER",
              ...}],
...
"fullName":"Karolína Zenknerová",
"initializedForLogin":true,
"m2":"628ca92611f78ed1f6a9a354bc23ce25284fdf63a0e3ca22f17bf69707c
      5862d",
"swtSetPrimary":false,
"threatsAvoidanceToken":"92fbdb6b-e3ff-495a-9c80-4779d20d7f06",
"translationVersion":"99.3.4"}
```

9) OCRA_secret

The example of the communication has already been shown in 3.1.6.

Conclusion - security and differences

The registration uses the SRP protocol - firstly to create a shared secret and use it to encrypt the new verifier and then to log in. SRP in the registration is implemented in the same manner as in 3.1.4 with same differences and same security considerations.

The login part of the algorithm is completely the same as in 3.1.4, therefore we omit the description.

In general there is no bigger issue in the registration part, we only mention the following.

As in the section about login, the modulus should be longer than 2048 bits in future, see for example recommendation given by NÚKIB [53] or BSI [28]. The exponent is sufficiently large, see the paper about Diffie-Hellman key agreement with short exponents [58]. It could be longer if needed.

The registration part is similar to login, but it uses much shorter values. The password regSecret, temporary master_secret, is "encrypted" using the first part of the SMS code, which is very short (6 digits) and it has only 10 hexadecimal nibbles. As a general key for encryption the value is too short and it is possible to brute-force. In this case it is not an issue, since we use these values in SRP protocol just once to derive a stronger (randomized and longer) value. We use this value to derive the key for the encryption of the verifier. We again note that the verifier is sensitive value, therefore it should be protected by encryption. It seems as a good idea to derive a stronger key. If the attacker could brute-force the encryption key and would obtain the verifier, he could pretend to be the server.

The verifier is encrypted by AES CBC with PKCS7 padding. The padding cannot be attacked since the message is generated only during the registration. Moreover, it is signed by HMAC, therefore not easily changable. The HMAC key is derived by using HMAC-SHA256 and temporary shared session secret K. The value of K is output of the SRP protocol with the temporary weak password. Using HMAC with the shared session secret for the key derivation is considered to be correct, see for example the recommendation given by BSI [28][Appendix B].

The encryption by created AES key and IV is performed only once, therefore there is no problem with reusing these values.

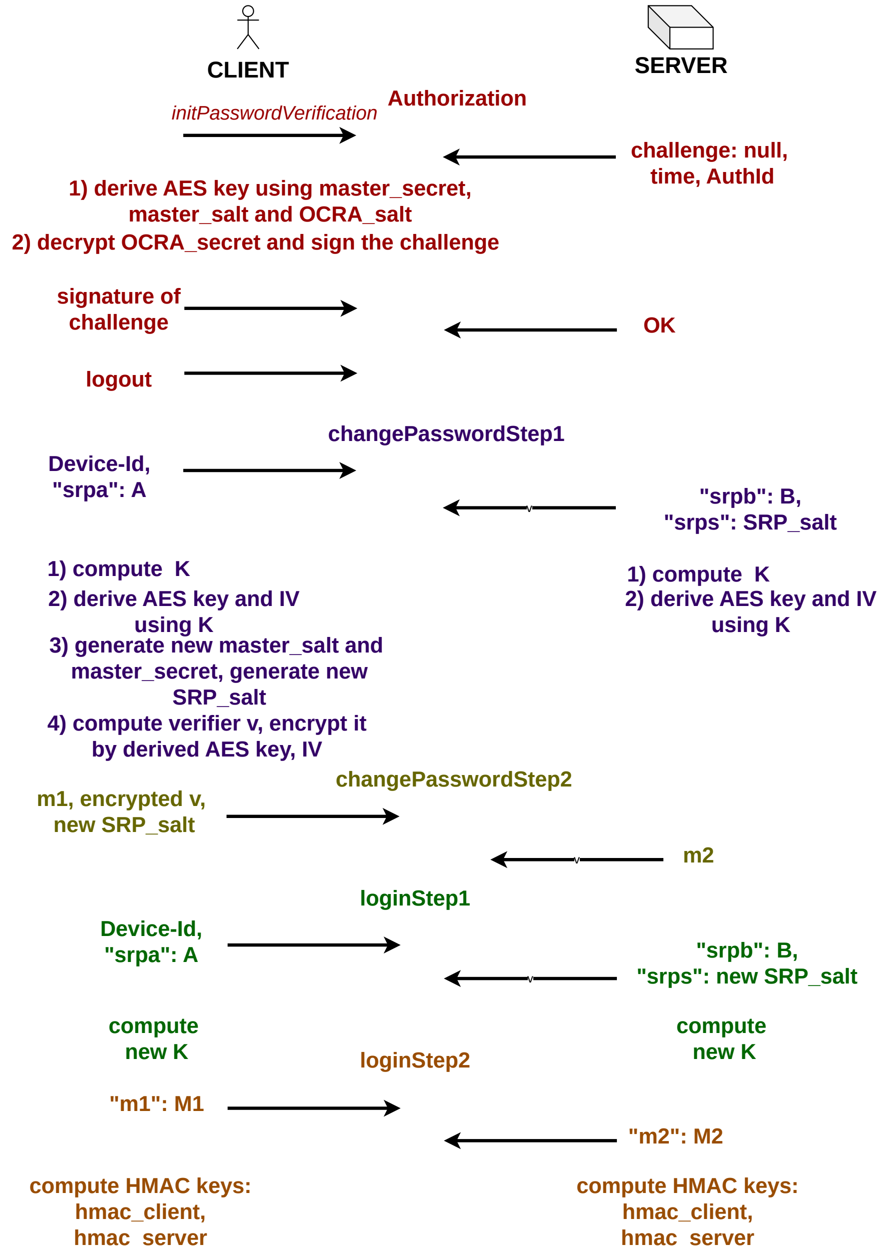
3.1.8 Change of password (PIN, master_secret)

In this section we describe the change of PIN and master_secret. The change includes both - the change of PIN, that is manually entered by the user and the change of master_secret (see 3.1.4) that is generated by the application. The change of PIN/master_secret uses both, SRP and OCRA.

Let us recall that the master_secret value is important and sensitive, because it is a part of our login and we use it to derive the shared session secret K in the SRP protocol. The value of PIN is used only to encrypt/decrypt master_secret and its change is not visible in the communication. Therefore, we do not describe the change of it in the pseudocode, but in the paragraph below it.

Scheme of communication

We provide an overview of the communication:



Figuratively, the communication is divided into three parts - authorization, change of password and login. **Authorization** is provided by OCRA as described above in the section about the authorization3.1.6. The only difference is that there is no challenge visible in the communication. That happens because the challenge is null (in the java code).
The second part, **change of password**, is similar to SRP in login. During this part a new password and salt are generated. At the end of this part the server obtains a new verifier. This part is described below in more detail. The third part is **login** which is same as in the 3.1.4.

Implementation

The first part (authorization using OCRA) is implemented as described in 3.1.6.

The second part is implemented in Security Module in three functions: changePasswordStep1, changePasswordStep2 and changePasswordFinalize. Only first two are important for us, because they contain cryptography.

The third part (login) is implemented in Security Module as described in 3.1.4.

For more details about Security Module see the next chapter 3.2.1.

We implemented the change of password in `change_password.py`.

Description and pseudocode

We describe the change of password part only, because the authorization and login parts are described above. We provide a pseudocode for the client side of the communication.

Notation

Firstly, let us introduce the notation which is similar to SRP 2.5.

I	...	Device-Id, username in SRP protocol
g	...	generator
N	...	modulus
master_secret	...	the password which is used in SRP protocol, see 2.5
master_salt	...	salt that is used in the encryption of master_secret and OCRA_secret, see 3.1.6 and 3.1.8
SRP_salt	...	salt used in SRP protocol
H	...	hash function SHA256
PAD	...	padding by ‘0’ bytes from the left to get the length 256 bytes
A, B, a, b	...	values from the SRP protocol described in 2.5
AuthId		unique Id of the authorized operation given by the server
OCRA_secret, OCRA_salt		values used in the authorization, see 3.1.6

Parameters and storage

In this paragraph we explain how to obtain parameters.

For the detailed description of the first part, OCRA authorization, see 3.1.6. Everything is the same - time and AuthId are received from the server, OCRA_salt and OCRA_secret are stored in shared preferences etc.

The third part, SRP login, is described above in 3.1.4.

The second part is very similar to the SRP login. Values of N and g are hardwired into the code, we have seen them in 3.1.4. Both - master_secret and master_salt are created during the registration/the change of password. The master_secret is encrypted using PIN as it is described below in 3.1.8. Username I is Device-Id created during the registration. B, and SRP_salt are received from the server.

Pseudcode

Algorithm 27 changePasswordStep1

INPUT: g, N, I
OUTPUT: -

1: Generate a, compute $A = g^a \mod N$. Send I and A to the server.
2: Receive SRP_salt and B from the server.

Algorithm 28 changePasswordStep2

INPUT: B,SRP_salt from the server
OUTPUT: -

1: compute u = H(PAD(A)||PAD(B))
2: compute x = H(SRP_salt||H(I||" : "||master_secret))
3: $S_C = (B - k \cdot g^x)^{(a+ux)} \mod N$, where k = H(N||PAD(g))
4: K = H(PAD(S_C))
5: $M1$ = H(H(N) XOR H(PAD(g)))||H(I)||SRP_salt||PAD(A)||PAD(B)||K)
6: Randomly generate new_master_secret, new_master_salt and new_SRP_salt. Everything has length 32 bytes.
7: Compute x = H(new_SRP_salt||H(I||" : "||new_master_secret)).
8: Compute verifier $v = g^x \mod N$.
9: AES_key=HMAC-SHA256(K,"aes_cbc_key" in bytes)
10: AES_IV=HMAC-SHA256(K,"aes_cbc_iv" in bytes)
11: encrypted_v=AES_CBC_PKCS7_encrypt(AES_key, AES_IV, v)
12: Send $M1$, encrypted_v, new_SRP_salt to the server.

From the decompiled code it seems that the condition on the nonzero B is checked during changePasswordStep2.

The value of PIN is used to encrypt master_secret which we have already seen in 3.1.6. More exactly, the application computes

encrypted_master_secret =
= PBKDF2(PIN,master_salt,1000,32) XOR master_secret.

Then the value of encrypted master_secret is stored. Therefore, if we want to change the PIN and master_secret (which happens at the same time), we have to decrypt the master_secret before we use it to create the shared secret K. Thus,

before the computation of x in changePasswordStep2 we compute

master_secret =
PBKDF2(PIN,master_salt,1000,32) XOR encrypted_master_secret.

The PIN code is entered manually. Then, after generating the new_master_secret, we have to encrypt it and store it. Consequently, the user creates a new PIN and and he enters it manually. The application computes

encrypted_new_master_secret =
PBKDF2(new PIN,new_master_salt,1000,32) XOR new_master_secret.

Then it stores the encrypted_new_master_secret. It can be only decrypted using the new PIN code of the user. The value of PIN is not stored on the device.

Notice that the value $M2$ from the server is received in changePasswordStep2, but not checked by the client. This does not follow the RFC 2945 [38], however, it seems that in this case it does not cause any problems. If something went wrong, the login in the next part would not proceed correctly.

Request - details

For illustration, we provide an example of the captured communication during the change of password.

1) Authorization

This part is described in the section about authorization 3.1.6.

```
POST /security/initPasswordVerification HTTP/1.1
Host: mb.airbank.cz
...
Content-Type: application/json;charset=UTF-8
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD
           8842A54BBA5D6
...
Hmac: acaf99d38b85369e2595eed4852f0cebd7fdfb348dd885f6e738038f
      15d22c1f
...
Connection: keep-alive
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:14 GMT
...
Hmac: 1232d7b80ada9306deb58deccd4d73e352b9682831dcda9fa3d8c1c61
      b6d72b8
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-07:16:16:14.407 +0100
...

{"allowedElementTypes":["SWT-ONLINE"],
```

```
"authId":"lkh7GP0Be2jlfyrfXbDMPFwoheFLKZ12BPUYdjU6lhbL7.2Zz6k0tQKM
          yN0KFrlj.ZeK0xisHzAi3ZOOPUY0bOdelYSRTTpjmY4_kNAjwMfq_Q7k
          l4BOGWYNlXURYy8v",
"created":"2025-02-07T16:16:14+0100",
"expiration":"2025-02-07T16:19:14+0100",
...
"requiresMobileLogin":true}
```

```
POST /authorization/v2/attempt HTTP/1.1
...
logString: XbBACORtsaGTRqyY8tOHp70yYmR8Sq3dD1sQKcCGBwOydsu3A4PTW7/
           UzxN3VNQWu4l/aibr6SWgM/dsTejbJjB402yAl5ZiECh3e8JxZF33hC
           PLBotO55PFrAwUz+eOZ9qvE1q7zNIW3kCctBszE0bkceg59ZFtGijX0
           6MNXd6+y98ni9m/YoOT4t2OsmLEQt/WdNd+6fgIpwO21koqo15cLmG/
           xdIFQDioWtK457aG2dqLEBEL2s/4yTj495Dhoy6eg/TUaPOvu0fMAKZ
           kRaHnVdUhTCPdcb81i3xSrIbBgrgIvaFXSL+oKBzfCL7JFpHBJPXe1i
           3gGogQcLldQg==
movingFactor: 57964712
elementType: SWT_ONLINE
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD8842
           A54BBA5D6
...
Hmac: ab1a9d74b237bf2edcc96d4dd08c85adc3983a31b29ce2634e482e830612
      8999
...
Host: mb.airbank.cz
Accept-Encoding: gzip, deflate, br

{"deviceHwId":"",
"authElementId":"4C4426365C34029FDB65EC49D43C231664023E75A9F698A12
FD8842A54BBA5D6",
"elementType":"SWT-ONLINE",
"authId":"lkh7GP0Be2jlfyrfXbDMPFwoheFLKZ12BPUYdjU6lhbL7.2Zz6k0tQKM
          yN0KFrlj.ZeK0xisHzAi3ZOOPUY0bOdelYSRTTpjmY4_kNAjwMfq_Q7k
          l4BOGWYNlXURYy8v",
"attemptTo":"AUTHORIZE",
"deviceFlags":"AAAAAAAAAAA=",
"geoLocation":"",
"authOperationReqHash":"JVfTEzabXwzB1egT2m3VfcP13wiOLLV43QO+7X7o0h0=",
"authCode":"D4WxSt9eFtwM4XXtuDY+AjMKjrBEFg5iSJWxoHljTPM="}
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:16 GMT
...
Hmac: 425feb1da377fb2f5c19dc7cc908c5b80b15db3a137a0d54f89f3ae09551004c
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-07:16:16:16.032 +0100
...

{"result":"OK"}
```

2) Logout

```
POST /security/logout HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD8
           842A54BBA5D6
...
Hmac: acaf99d38b85369e2595eed4852f0cebd7fdfb348dd885f6e738038f1
      5d22c1f
...
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:21 GMT
...
Hmac: 91cc2eb4f094aa1a5e15f7fb97154e9ba25e621c329d277a6492cb69c
      b6e8c94
...
Http-Host: mb.airbank.cz
Real-Clock: 2025-02-07:16:16:21.782 +0100
...
```

3) changePasswordStep1

```
POST /security/changePasswordStep1 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A1
           2FD8842A54BBA5D6
...

{"appVersion":null,
"deviceType":null,
"os":null,
"srpa":"8d3aa2d7c953568f6fdb33084c2557d5fc0e80c3485b7e2f96e
        1e35cee8d5c3ea36e450a2d257fbde4fc98e39e02a378c59075
        689bc312aaca997a7670b8797b0e93213d9b47b0c3de69d1376
        b0904676b5a95ab75c07a7c7ca616bf133e349a1ac1776e11b5
        88324e5dfef86be8107a78f01b00a5781318933e8653a1eff77
        f42fc61ba3afd35c603f21e8d17c7c8eb54e7551cab8d77f157
        5fe2deb904837be61a14527553c38731dd9255d63562de984b4
        5b1cdd690a25a233b62db28e4cfe0555ba1abd95b53a1eb95ed
        34f52f161f9cf01c1c8779bc660300936026be60658ef5cdce9
        0d0dd4c92e01811f47f580611cf19a2ee81a49b5acdbb3c3ddf
        5b",
"secretType":null}
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:23 GMT
```

```
...
Http-Host: mb.airbank.cz
...
{"srpb":"672c9e204ce005d229cc95b4e347afad7615473707e3437
         d7e22b8c0c328b6c3b400021f30c01407076b799289247c
         51ff30d27666b5506dfa6b830dd79a32fea070760c211ca
         4ad99e48a5cbf92013580d822135278938ad4fc4ba51719
         2406e584aea5ee3aa3ca7d4b8edacda745fc9dfb3ae6dd8
         f7ed280e4056b032fe25641336c9bbdffe3e063929c0f3e
         43ce808ebbac8b29600862b194ca13fa7475dc1957617f4
         0361342c8a057ea802184490c941d7402be75b855f827b8
         0ca363e60b8b8a7d2605eabf8c324089538ecc05ca73fd2
         67adf5d817ee2e674c271733a278946b6d446ac50769e2b
         26b89e3d8a0937f9f2f727992042092aa9f5fcadff",
"srps":"132d10db11e95cdcc93bcde9763c0add362258e9689107c0
        223add4dd81ca5f1"}
```

4) changePasswordStep2

```
POST /security/v4/changePasswordStep2 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A
           12FD8842A54BBA5D6
...

{"deviceType":null,
"m1":"23b87220202f440c34da5de783eace35c183a1e1ca09d7316a926
      90fd840ae2e",
"os":null,
"secretType":null,
"newMasterSecretSalt":"74507ebb0de8bc45ad531dc9b6fecf85c76b9
                       bf3a59ef0d9661242cc601eab14",
"newMasterSecretVerifierEnc":"c6e4f028f6b7d9a82224c6a215f0a3
                              d745b062fd91ca0241ecf3c616b790
                              d21fba07328f7012277d5b5c7d3f61
                              b3a2d9c1770fe680a66ac07118568e
                              3d448b2724fe41a700c6d42ebb498f
                              02b7cf515f4eace3dafc9b11293d01
                              c69d749a33bb92bd5bfed88cb0d80a
                              a080770309df390c9f05b05058d928
                              8594ef74afc07936ac95b569ddc016
                              e3e475d11dfea6ee794430679a3c71
                              2172439e9f8cf8f066197bef1800a2
                              5dbe3c619badb7d0b764ba6e902307
                              3dd423b13a718fa96b8c037bd211da
                              09cedb120f3a1ca2a0fd059aa204b2
                              7b2b3defd6e293d5212791319aad84
                              b0f9515a477bf16c32bd9f724212f8
                              b70226cc645c0a1e801e01a8e1b69e
                              15183caed3bb14a42b3c9c0abfa19e
```

```
                              cf60"}
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:24 GMT
...
Hmac: 209581a5121ae2fec3456c02c96c44c87018ee1aeed97ad413269de9
      fa32a521
...
{"applicantRef":null,
"contracts":[{"contractNumber":"*******",
            "firstName":"Karolína",
            "id":********,
            "lastName":"Zenknerová",
            "profileId":********,
            "relation":"OWNER",
            ...}],
...,
"fullName":"Karolína Zenknerová",
"initializedForLogin":true,
"m2":"4eb832c217c217568690259b6076f15450768c7d44d678d55c2ab76e1
      26bf747",
"swtSetPrimary":false,
"threatsAvoidanceToken":"6d81f2e9-5da4-4bfd-98bf-68e74f9794e2",
"translationVersion":"99.3.4"}
```

5) loginStep1

Login is described in 3.1.4 in more detail.

```
POST /security/loginStep1 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD
           8842A54BBA5D6
...

{"appVersion":"A.10.1.0",
"deviceType":"Pixel 3 XL",
"os":"Android 12",
"srpa":"67f96a8365cece51f4add9873faa87851749e7e6cc4a55bed5b53f
        bde224dea9e8730022a706b364b74f8c0dbdae149aa5b5b140b873
        f814456941050a13a54d792d91c516035f7abd4f4c1fe05d0ffcf5
        d2597ab4f8eb49fad94356f554a840c75e7e5f5458da182435066e
        893b8107c60f555c8305237156879c77d4c1957f88e37eb529909c
        5337443f595372d996b656ef802fb4b6d8cbcf844ec2cebe515510
        83abf505216ac049b2bd8fa02f7ad3f71dfea9cc89632a116dd40b
        8227d8f6b28102378883e1a73479b6d7f0a58a122b843c6278fc71
        5c0b91f63dd9bf8e02e8601b422a43f61009905662d4c365a90b8c
        2f79c43512e93117e28324fef2",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:25 GMT
...
Http-Host: mb.airbank.cz
...

{"srpb":"6114c046fa2674053ff6ace30465c5808a288e99f5e16bdc635be6
         5d3e1cfd52682e475928ecb579c0b8870f105fe65e2ee5e98d7fd4
         ee33c7b489afba38254cd83babb22a820484b5df8a18980a7c5640
         8dd4151955433dc83f53007d58cf468471ca1dc53156d0f2877d17
         c638b70d2127625e1d1e42549298696c8116f430583089ff224bfb
         d5f25081162b8a109a920c01f92205e2a380c63dbb724c1794b606
         a0cdb1ceb0f9ffb3912b206bbafcedeaed28a1954cfbd9a845456f
         25b1edd5a1b84ed5e29db82632ac0a79577161f599174770de5e7a
         c47a4930c03cd8c6363dd45670a071d5eebb3ce1337903f94ffc28
         527583904b63e785c837d58497",
"srps":"74507ebb0de8bc45ad531dc9b6fecf85c76b9bf3a59ef0d9661242c
        c601eab14"}
```

6) loginStep2

```
POST /security/v4/loginStep2 HTTP/1.1
Host: mb.airbank.cz
...
Device-Id: 4C4426365C34029FDB65EC49D43C231664023E75A9F698A12FD884
           2A54BBA5D6
...

{"deviceType":"Pixel 3 XL",
"m1":"10346262399ddc0bfc1123b01f83cfe9b1312bb3f4be0eab9ef931ce744
      81a90",
"os":"Android 12",
"secretType":"PASSWORD"}
```

```
HTTP/1.1 200 OK
Date: Fri, 07 Feb 2025 15:16:27 GMT
...
Hmac: 186b2b4bb66e11d2826666b4121385e066041cb56f31afad3fcbfac4d0
      5eb72c
...
Http-Host: mb.airbank.cz
...
{"applicantRef":null,
"contracts":[{"contractNumber":"*******",
             "firstName":"Karolína",
             "id":********,
             "lastName":"Zenknerová",
             "profileId":********,
             "relation":"OWNER",
             ...}],
```

```
...
"fullName":"Karolína Zenknerová",
"initializedForLogin":true,
"m2":"c42ec53658231b868e2a3c7fb9b696ffec3087af90c7fd51a74d22194a
      3df172",
"swtSetPrimary":false,
"threatsAvoidanceToken":"6d81f2e9-5da4-4bfd-98bf-68e74f9794e2",
"translationVersion":"99.3.4"}
```

7) Change OCRA_secret after the change of password

We have seen the corresponding example in the section about the authorization 3.1.6.

Conclusion - differences and security

The first part of the change of password is the authorization, which uses OCRA. We described this part already in 3.1.6.

In the rest of the change of password we use SRP protocol twice. SRP is implemented in the same manner as in the login, therefore there are the same differences and security considerations as in 3.1.4.

The login part is exactly the same as in the 3.1.4.

The part with change of password uses SRP to create the shared secret and then to derive the key that is used for encrypting a new verifier. In contrast to the registration, we use the previous master_secret which has the same length (256 bits) as the generated new master_secret. However, the main idea is the same. Also the encryption by AES CBC with PKCS7 padding is the same as in the registration. The message from the client is not signed by HMAC, but there is $M1$ proving that he has the same shared secret K. We can assume that the server checks the value of $M1$ before proceeding. We note that the in the change of password the value of $M2$ from the server is not checked, and encrypted verifier is sent with $M1$. This does not follow the SRP protocol, on the other hand, it seems that it does not cause any problems. In general we recommend to follow RFC or standard.

3.2 Practical part

In this chapter we will cover the practical part of our analysis of MyAir by Airbank. We used three versions of the application for our analysis, 9.0.1, 9.3.1 and 10.1.1.

3.2.1 Cryptographic libraries and sources

In this section we describe where to find the code that can help us analyze the cryptography in the application.

Firstly, the application file can be found in /data/app/. For example on our device the application is in the folder

```
/data/app/~~F9jnhGJbD8n-1YPRVAwtgA==/
cz.airbank.android-TW857BUA3W-qa34hmWdm9A==
```

There is a file of the application base.apk that can be decompiled using java decompiler jadx 1.3.2. We used it mainly to analyze the authorization.

By the decompilation we have also found out that most of cryptographic protocols and primitives in MyAir can be found in Security Module library.

We searched for words from captured communication, for example "loginStep1" to locate the usage of a cryptography in the application. By this searching we found the java package cz.airbank.mobile.securitymodule. The Security Module native library is loaded as follows:

```
public class SecurityModule {
    private static final String TAG = "SecurityModule";
    private long handle = init();

    static {
        System.loadLibrary(TAG);
    }
```

The library can be found in the same folder as base.apk
as lib/arm64/libSecurityModule.so. The library is a native library, thus we used Ghidra 1.3.2 to decompile it and analyze the code. From the decompiled code we deduced that the original code was written in C/C++. As we have said, the library provides most of cryptographic functions in the application. The authorization is handled separately. Similarly as before, we have found the correct part of the code, where the authorization is performed. The authorization is included in the java package com.aheaditec.casemobilesdk.airbank. The cryptography can be found in com.aheaditec.casemobilesdk.airbank.crypto. This library uses shared preferences, which can be found in /data/data/cz.airbank.android/shared_prefs.

The registration additionally uses java package
cz.airbank.mb.android.pairing.face or cz.airbank.mb.android.pairing.face.dot, where the first phase of the registration is implemented.

3.2.2 Setup

Most of the setup was already described in 1.3.

Certificate pinning

The application additionally uses a certificate pinning. The certificate pinning is a technique that is used to improve security of a TLS connection 1.1.1.The application communicates with the server through the proxy provided by BurpSuite. We installed the CA certificate into the mobile to enable the connection between Burp proxy and the mobile through TLS, see 1.3.

When the application creates the connection, the certificate signed by Burp CA is checked, but its public key is not the public key of the server, it is the public key of Burp proxy.

Then, due to the certificate pinning, the value of the public key of the server is checked against the pinned value (the real public key of the server). When values

are not equal, which are not, the application starts, but instead of the login screen there is a notification that the bank is working on some improvement or fix. For more details about certificate pinning we refer to the article about it [59].

It is possible to bypass the certificate pinning and make the application work. We used Frida 1.3.2 with the script that enables bypassing the pinning. We found the script on the website Frida CodeShare [19]. The script can be found in attachments as `android-pinning-bypass_2.js` A.2. We run the script as follows:

- `frida -U -f cz.android.airbank -l android-pinning-bypass_2.js`

Then the pinning is switched off and the application works.

3.2.3 Reversing

In this section we give more details about reversing the code and analysing it.

Communication

We capture the communication using Frida and Burpsuite. It is our starting point, after discovering some key words in the communication we can continue with exploring the static code and tracing functions.

3.2.4 Authorization

The authorization can be found in the java package com.aheaditec.casemobilesdk.airbank 3.2.1. To find it in the code decompiled by jadx, we use parts of captured communication, for example we can search for the usage of the word "authOperationReqHash" or "logString". By the search we can find the package and then by seraching further and exploring we can finally find the cryptographic part. We can try to hook the whole package that provides the authorization using the following command after the initial setup:

- `frida-trace -U -f cz.airbank.android -j ’*ahead*!*’`

Part of the output for illustration:

```
...
| b.b("CM3AuthorizationManager",
            "Generating signature for transaction approving")
    | a.a()
    | <= 8000
 b.h()
    | b.g("CSMStorageManager.encSecret")
    |     | b.s("CSMStorageManager.encSecret", null)
    |     | <= "5B8C9B76B2B2E0B922...B646980535A1"
    |     | f.h("5B8C9B76B2B2E0B922...B646980535A1")
    |     | <= [91,-116,-101,118,-78,-78,...,-74,70,-104,5,53,-95]
    | <= [91,-116,-101,118,-78,-78,...,-74,70,-104,5,53,-95]
 <= [91,-116,-101,118,-78,-78,...,-74,70,-104,5,53,-95]
 c.a()
```

```
<= "Bils5ZEN89SC7D...Nd_yuOCyI9GFRFbVI3"
a.f()
<= null
a.m("Bils5ZEN89SC7D...Nd_yuOCyI9GFRFbVI3", null, [0,0,0,0,0,0,0,0])
   | f.c("Bils5ZEN89SC7D...Nd_yuOCyI9GFRFbVI3")
   | <= [66,105,108,115,53,90,...,121,73,57,71,70,82,70,98,86,73,51]
<= [-15,82,87,25,87,-84,...,8,66,-10,98,-92,-71,124,-64,103,106]
f.i([-15,82,87,25,87,-84,...,8,66,-10,98,-92,-71,124,-64,103,106])
<= "F152571957AC2614B029D18EF3E190A21B29F3
...
```

We compared obtained values with the decompiled code and then we implemented our version in java using parts of the decompiled code. We compared output ouf our script with the captured communication.

Security Module

Reversing the Security Module was more complicated. Firstly we will describe static analysis.

In Ghidra we could see that all functions imported to application are in the folder Java_cz_airbank_mobile_securitymodule_SecurityModule_.
We can choose functions that are interesting for us, for example loginstep1:

Decompile: Java_cz_airbank_mobile_securitymodule_SecurityModule_loginStep1 - (libSecurityModule.so)

```

SecurityModule *
Java_cz_airbank_mobile_securitymodule_SecurityModule_loginStep1
          (long *param_1,undefined8 param_2,long param_3,SecurityModule *param_4)

{
  char *pcVar1;
  long lVar2;
  undefined4 uVar3;
  undefined8 uVar4;
  undefined8 uVar5;
  undefined8 uVar6;
  char *pcVar7;
  SecureString aSStack_90 [24];
  SecureString aSStack_78 [24];
  SecureString aSStack_60 [24];
  long local_48;

  lVar2 = tpidr_el0;
  local_48 = *(long *)(lVar2 + 0x28);
  if (param_4 != (SecurityModule *)0x0) {
    pcVar7 = (char *)0x0;
    if ((param_1 != (long *)0x0) && (param_3 != 0)) {
      pcVar7 = (char *)(**(code **)(*param_1 + 0x548))(param_1,param_3,0);
    }
    pcVar1 = "";
    if (pcVar7 != (char *)0x0) {
      pcVar1 = pcVar7;
    }
                    /* try { // try from 0019f938 to 0019f93f has its CatchHandler @ 0019fb20 */
    sm::SecureString::SecureString(aSStack_60,pcVar1);
    if (((param_3 != 0) && (param_1 != (long *)0x0)) && (pcVar7 != (char *)0x0)) {
                    /* try { // try from 0019f954 to 0019f963 has its CatchHandler @ 0019fb04 */
      (**(code **)(*param_1 + 0x550))(param_1,param_3,pcVar7);
    }
                    /* try { // try from 0019f964 to 0019f96b has its CatchHandler @ 0019fb18 */
    sm::SecureString::SecureString(aSStack_78);
                    /* try { // try from 0019f96c to 0019f973 has its CatchHandler @ 0019fb10 */
    sm::SecureString::SecureString(aSStack_90);
                    /* try { // try from 0019f974 to 0019f987 has its CatchHandler @ 0019fb0c */
    uVar3 = sm::SecurityModule::loginStep1(param_4,aSStack_60,aSStack_78,aSStack_90);
                    /* try { // try from 0019f98c to 0019f99b has its CatchHandler @ 0019fb08 */
    param_4 = (SecurityModule *)
```

```
Decompile: loginStep1 - (libSecurityModule.so)
1
2  /* sm::SecurityModule::loginStep1(sm::SecureString const&, sm::SecureString&, sm::SecureString&) */
3
4  undefined4 __thiscall
5  sm::SecurityModule::loginStep1
6            (SecurityModule *this,SecureString *param_1,SecureString *param_2,SecureString *param_3)
7
8  {
9    long lVar1;
10   SharedSecretData *this_00;
11   ulong uVar2;
12   ulong extraout_x1;
13   undefined4 uVar3;
14   SecureData *this_01;
15   SecureData aSStack_90 [24];
16   SecureData aSStack_78 [24];
17   SecureData aSStack_60 [24];
18   long local_48;
19
20   lVar1 = tpidr_el0;
21   local_48 = *(long *)(lVar1 + 0x28);
22   if (*(int *)(*(long *)this + 4) == 0x10) {
23     CryptoUtils::PBKDF2((CryptoUtils *)param_1,(SecureString *)(*(long *)this + 0x28),
24                         (SecureData *)0x3e8,(ulong)param_3);
25                     /* try { // try from 0019c2c4 to 0019c2cb has its CatchHandler @ 0019c484 */
26     SecureData::SecureData(aSStack_78);
27                     /* try { // try from 0019c2d4 to 0019c33f has its CatchHandler @ 0019c490 */
28     CryptoUtils::XOR(aSStack_60,(SecureData *)(*(long *)this + 0x40),aSStack_78);
29     this_01 = *(SecureData **)(this + 0x18);
30     if (this_01 != (SecureData *)0x0) {
31       SecureData::~SecureData(this_01 + 0xc0);
32       SecureData::~SecureData(this_01 + 0xa8);
33       SecureData::~SecureData(this_01 + 0x90);
34       SecureData::~SecureData(this_01 + 0x78);
35       SecureData::~SecureData(this_01 + 0x60);
36       SecureData::~SecureData(this_01 + 0x48);
37       SecureData::~SecureData(this_01 + 0x30);
38       SecureData::~SecureData(this_01 + 0x18);
39       SecureData::~SecureData(this_01);
40       operator.delete(this_01);
41     }
42     this_00 = (SharedSecretData *)operator.new(0xd8);
43                     /* try { // try from 0019c344 to 0019c347 has its CatchHandler @ 0019c474 */
44     SRP::SharedSecretData::SharedSecretData(this_00);
45     *(SharedSecretData **)(this + 0x18) = this_00;
```

If we would open some cryptographic function, for example HMAC_SHA256, we could see that the function HMAC is called inside it. From the code in the HMAC function we can see that it is taken from OpenSSL library [54]:

```
Decompile: HMAC - (libSecurityModule.so)
1
2  uchar * HMAC(EVP_MD *evp_md,void *key,int key_len,uchar *d,size_t n,uchar *md,uint *md_len)
3
4  {
5    uchar *md_00;
6    int iVar1;
7    HMAC_CTX *ctx;
8    ENGINE *pEVar2;
9    uchar auStack_94 [64];
10   uint local_54;
11
12   md_00 = (uchar *)0x367fe0;
13   if (md != (uchar *)0x0) {
14     md_00 = md;
15   }
16   ctx = (HMAC_CTX *)CRYPTO_zalloc(0x20,"crypto/hmac/hmac.c",0x89);
17   if (ctx != (HMAC_CTX *)0x0) {
18     iVar1 = HMAC_CTX_reset(ctx);
19     if (iVar1 == 0) {
20       HMAC_CTX_free(ctx);
21       ctx = (HMAC_CTX *)0x0;
22     }
23     else {
24       if ((key == (void *)0x0) && (key_len == 0)) {
25         iVar1 = HMAC_Init_ex(ctx,&DAT_002edafe,0,evp_md,(ENGINE *)0x0);
26       }
27       else {
28         iVar1 = HMAC_Init_ex(ctx,key,key_len,evp_md,(ENGINE *)0x0);
29       }
30       if (((((iVar1 != 0) && (ctx->md != (EVP_MD *)0x0)) &&
31             (iVar1 = EVP_DigestUpdate((EVP_MD_CTX *)(ctx->md_ctx).digest,d,n), iVar1 != 0)) &&
32            ((ctx->md != (EVP_MD *)0x0 &&
33             (iVar1 = EVP_DigestFinal_ex((EVP_MD_CTX *)(ctx->md_ctx).digest,auStack_94,&local_5
34             iVar1 != 0)))) &&
35           ((iVar1 = EVP_MD_CTX_copy_ex((EVP_MD_CTX *)(ctx->md_ctx).digest,
36                                        (EVP_MD_CTX *)(ctx->md_ctx).flags), iVar1 != 0 &&
37            ((iVar1 = EVP_DigestUpdate((EVP_MD_CTX *)(ctx->md_ctx).digest,auStack_94,(ulong)loca
38             iVar1 != 0 &&
39             (iVar1 = EVP_DigestFinal_ex((EVP_MD_CTX *)(ctx->md_ctx).digest,md_00,md_len), iVar
40             ))))) {
41         EVP_MD_CTX_reset((ctx->md_ctx).engine);
42         EVP_MD_CTX_reset((ctx->md_ctx).flags);
43         EVP_MD_CTX_reset((ctx->md_ctx).digest);
```

The structure is the same as well as used names and labels which can be found in OpenSSL.

Therefore, we can see that programmers used cryptographic functions from OpenSSL and they created and added higher layers of the program. For example they created the SRP protocol which uses cryptographic functions from OpenSSL. We can also notice that input parameters are of different types - input parameters of the function called from java are standard, for instance long, whereas inner functions implemented in the library have input parameters of the type SecureString or SecureData, which are classes created by programmers of this module. Functions taken from OpenSSL have standard input parameter types. We will use this fact later.

Since the code in Ghidra is not easily readable, we can trace chosen functions using Frida 1.3.2. That means we can see input and output of every called and hooked function. In Frida it is possible to trace java functions quite easily using frida-trace. However, the Security Module library is a native library, therefore we have to use a different approach. We can hook native functions using Frida and inside it to use Interceptor which allows us to identify input and output arguments when the native function is called.

For tracing we need to know the name of functions which can be found in the decompiled code in Ghidra. We can search names of functions in Global

namespace using options "Search → Program Text". We also have to be able to read the input/output arguments of a native function.

This is the point where the fact that the OpenSSL uses well known data types of input/output parameters becomes useful. Reading input/output with SecureString/SecureData would be probably more demanding, since we would have to understand the structure. Therefore, we decided to trace OpenSSL functions which was much easier since the used data types or structures are easily readable. For example we used the fact that in SRP we have to compute g^a. This operation uses OpenSSL function BN_mod_exp. This strucure uses BigNumber where the pointer points at the memory where the value of the number is stored. As an example we provide the code that helped us to uderstand how the login works in the attachment `hook_native_login.js` A.3.

For illustration, we provide the beginning of this code:

```
var Jni2= Module.getExportByName('libSecurityModule.so', 'BN_mod_exp');
Interceptor.attach(Jni2,
{onEnter:
   function(args){
   console.log("Vstupuji do BN_mod_exp")
   console.log(args[0],args[1],args[2],args[3],args[4],args[5],args[6])
   console.log("Pokus ziskat BN ...");
   console.log("Generator: ");
   console.log(hexdump(ptr(args[1].readPointer())));
   console.log("Exponent: ");
   console.log(hexdump(ptr(args[2].readPointer())));
   console.log("Modulus: ");
   console.log(hexdump(ptr(args[3].readPointer())));
   console.log(hexdump(ptr(args[0])),hexdump(ptr(args[1])),
               hexdump(ptr(args[2])),hexdump(ptr(args[3])),
               hexdump(ptr(args[4])),hexdump(ptr(args[5])));
    },
onLeave: function(retval)
{console.log("return BN_mod_exp: ");
    console.log(retval);}})
...
```

After the initial setup described in 1.3 and 3.2.2, supposing that the script bypassing the certificate pinning is running, we will run the script named for example `hook_login.js` as follows:

- frida -U -F -l hook_login.js -o output_hook_login.txt

The ouptut of the script is stored in `output_hook_login.txt`. Here we provide an example of this output. We deleted addresses, to save space. There is "#" in the place of addresses.

```
...
Pokus ziskat BN ...
Generator:
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
#  02 00 00 00 00 00 00 00 80 7b 8a 7e 75 00 00 00  .........{.~u...
```

```
#  01 41 00 00 00 00 68 28 00 00 00 00 00 00 00 00  .A....h(........
#  22 20 37 00 00 00 00 00 00 00 00 00 00 00 00 00  " 7.............
#  01 40 00 00 00 00 91 11 00 00 00 00 00 00 00 00  .@..............
#  00 00 7c 42 00 00 00 00 00 00 00 00 00 00 00 00  ..|B............
#  01 c1 00 00 00 00 96 29 00 00 00 00 00 00 00 00  .......)........
#  6c 04 00 00 1c 00 00 00 6c 04 00 00 00 00 00 00  l.......l.......
#  01 01 01 00 00 00 da 56 00 00 00 00 00 00 00 00  .......V........
#  a0 a5 8c fe 75 00 00 00 00 00 00 00 00 00 00 00  ....u...........
#  01 61 00 00 00 00 d7 cd 00 00 00 00 00 00 00 00  .a..............
#  22 20 37 00 22 20 00 00 00 00 00 00 00 00 00 00  " 7." ..........
#  01 40 00 00 00 00 c8 4a 00 00 00 00 00 00 00 00  .@.....J........
#  00 b8 88 41 00 00 00 00 00 00 00 00 00 00 00 00  ...A............
#  01 c1 00 00 00 00 cf 72 00 00 00 00 00 00 00 00  .......r........
#  00 00 c0 41 00 00 c0 41 00 00 c0 41 00 00 00 00  ...A...A...A....
#  01 40 00 00 00 00 ac b0 00 00 00 00 00 00 00 00  .@..............
Exponent:
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
#  49 77 bd ab bf 8d a2 5b be f4 06 14 e1 99 ba fd  Iw.....[........
#  02 bb 96 cc b0 7d 07 44 3a c7 4b 9a 55 5c 22 98  .....}.D:.K.U\".
#  b1 c9 6d ea ce 2d 5e d7 4d ac 51 02 42 e2 e9 51  ..m..-^.M.Q.B..Q
#  7d 75 2d fb 5a d5 2e a2 d2 99 46 3b 82 79 69 9c  }u-.Z.....F;.yi.
#  1f 2a 54 94 94 21 56 7f d9 bb 60 17 58 13 ab 0a  .*T..!V...`.X...
#  ba f2 91 7d fb 0d be 54 ff 22 93 0f be 5e 6b 9d  ...}...T."...^k.
#  66 af b6 19 06 dc 35 ce f2 d4 02 6d 60 6d 87 33  f.....5....m`m.3
#  28 97 5b 96 d3 0a bf 74 ea f9 c8 d0 fa be ff c6  (.[....t........
#  d6 2a 3c cc 1d ee b3 d3 90 2f 24 ab 96 0c 16 82  .*<....../$.....
#  10 c0 fa 26 de 51 55 38 69 81 00 c6 08 02 e9 c3  ...&.QU8i.......
#  bb 4c 31 3d b0 31 29 61 92 56 b4 e6 71 22 3c 28  .L1=.1)a.V..q"<(
#  59 64 04 e7 7d cb d0 ff 90 7c 77 ba 9a 2e 55 a2  Yd..}....|w...U.
#  82 b4 cd 08 7e 0f 3e da 19 8c fc 82 23 92 52 23  ....~.>.....#.R#
#  62 bf 71 11 26 28 6c 6e 8f ca 4a 5a a6 8d 49 c2  b.q.&(ln..JZ..I.
#  43 ba 38 ae 1b 13 14 49 1d c8 68 6f 80 ef 7b e5  C.8....I..ho..{.
#  17 ec af fd 3a 0c 92 92 2f d1 34 c0 e2 89 bb 6a  ....:.../.4....j
Modulus:
    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
#  73 ff 4a 9e 1f 11 a7 0f f2 8e d6 fc 72 e3 65 9b  s.J.........r.e.
#  75 54 5f 52 6d 23 de 35 e4 7a 9f d8 03 c8 b5 94  uT_Rm#.5.z......
#  b6 fb db e9 f8 35 ae 71 82 c3 d0 a8 f3 98 56 2a  .....5.q......V*
#  d8 08 c3 7b 1c 04 cc 9c 29 53 ce 03 73 4e 87 af  ...{....)S..sN..
#  e6 7a e5 04 90 27 60 61 78 b3 2f f5 db fb 2c 03  .z...'`ax./...,.
#  fa ec d2 75 27 7a a7 5e 7d d5 b0 24 b5 23 45 54  ...u'z.^}..$.#ET
#  48 77 f8 88 e6 32 9d 5b 1a 46 17 87 07 b9 d2 f1  Hw...2.[.F......
#  81 64 6c 43 7a 20 bd 76 16 80 fb 23 3a b4 97 ca  .dlCz .v...#:...
#  3b 77 14 6b 44 1e 28 1d a7 3e c3 d5 41 d0 59 73  ;w.kD.(..>..A.Ys
#  74 ff f4 db 0a 74 0d a8 ea 5e 97 ec 93 79 f9 55  t....t...^...y.U
#  b8 93 0b 2f 96 a9 18 29 e8 aa fa d5 fb 05 1a 66  .../...).......f
#  b3 3a 16 9a 17 95 60 cf b0 67 b7 ed 69 39 08 e8  .:....`..g..i9..
#  50 fd 04 da a9 48 7f cd 0d 31 03 4b ab 12 23 d5  P....H...1.K..#.
#  3d a1 67 77 75 e0 93 81 ed 99 a0 b4 cb 29 73 a3  =.gwu........)s.
#  50 60 b5 3d 94 92 31 fc 07 ee 87 19 65 b6 72 af  P`.=..1.....e.r.
#  2f 58 89 13 5e de 66 f1 9b 9a 4a 32 41 db 6b ac  /X..^.f...J2A.k.
```

```
Error: access violation accessing 0x4
    at <anonymous> (frida/runtime/core.js:145)
    at t (frida/runtime/hexdump.js:8)
    at onEnter (/home/kaja/backup/Diplomka/hook_native_login.js:18)
return BN_mod_exp:
...
```

The full output file is in attachements as
`attachment_login_output_master_thesis.txt` A.3.

The script is in attachments as `hook_native_login.js` A.3.

We used the code in Ghidra, captured communication and values obtained from Frida to get the idea of how the application works. We wrote the code, obtained secret values from Frida output and we compared the result with the captured communication.

Registration

Most of the registration is written in Security Module, thus the reversing was the same as above. The only difference is the start of the registration - photos sending and the decryption of the received regSecret. This part can be found in jadx by searching for "regId" or for example "regSecretEnc". After filtering out irrelevant results, we can identify the part of the code that decrypts the "regSecret" - it is in the package cz.airbank.mb.android.pairing.face. Since the code was clear, it sufficed to copy the code and compare the result with the input to function in Security Module.

Experiment

We experimentally verified our hypothesis by writing our own code, capturing values when running oprations and then comparing our result values with values captured in the communication. By experiments we successfully verified our implementations of HMAC, Login, Registration, Change of Password, Authorization and decryption by session secret. We did not verified The change of OCRA_secret.

3.3 Conclusion

In this chapter we will conclude the results mainly in terms of security.

Firsly, we could not analyze the application fully as we wanted, since the bank did not approved our request. Therefore some parts of the work are only assumptions, for example we cannot verify whether the server checks the signature. Hovewer we know we analyzed most of functions correctly, since we were able to perform the same computations as the application. We verified it experimentally using values from the captured communication and from the memory.

The application mostly uses a native library. It helped us that they do not implement lower cryptographic layers. Instead of that they use functions from OpenSSL which are easier to track.

Secondly, we analyzed the cryptograpy from the theoretical point of view. In this chapter we will conclude the results mainly in terms of security.

Imagine we do not trust TLS.

The SRP protocol partially ensures forward secrecy, since it generates a new unique key for every session. The forward secrecy is ensured for values using the secret generated during the SRP protocol - HMAC keys and keys for encryption by session secret.

The secret that is used by OCRA - OCRA_secret - is always the same, hence this part of the protocol does not provide forward secrecy. Also the master_secret is always the same.

HMAC is used to sign only body of requests. It means that it does not provide integrity of the whole communication, only of body of requests/responses. It is possible to change uri, method or headers and the request passes as valid. The attacker can send his/her own requests with the body and HMAC of some captured request. It does not allow the attacker to read important information, since sensitive transferred data are encrypted using session secret. Also their sending has to be authorized.

Let us consider the following cases of attackers:

1. The attacker that can capture and modify the communication.
2. The attacker that can obtain data from the mobile device.
3. The attacker that can do both - capture the communication and obtain data from the mobile device.

The scheme is vulnerable as follows:

1. The attacker in this position can theoretically resend forged modified requests using valid HTTP body and its HMAC. This could be fixed by signing all important parts of the request. More sensitive parts of the protocol use SRP and OCRA and thus the protocol requires some knowledge that only the client has. Ususally it is impossible for attacker to obtain this knowledge even when he actively sends request to the client and server, because of protocols design. Also the sensitive data are sent encrypted, therefore the attacker cannot read them.
2. The attacker in this possition cannot do anything useful. The attacker cannot obtain master_secret by bruteforcing PIN, because even though he can try to decrypt it, he has not way how to verify it. We can assume that the server blocks the user after several unsuccessful attempts to log in. Without master_secret he cannot decrypt OCRA_secret.
3. The attacker in this position can do everything as previous two attackers. Moreover, in this possition the attacker can bruteforce PIN and therefore obtain master_secret that is encrypted using PIN. He can do this attack offline which we have already described in the section about OCRA 3.1.6.

From the practical point of view, the application does not make much effort to stop the reverse engineering. There are ways how to make the application more resistant against the reverse engineering, keys extraction etc. For example, the application could detect the usage of Frida.

The application also does not use keystore. If it would use the keystore it would be much more difficult extract keys and therefore to reverse engineer the cryptography. For the attacker the keystore would mean more complications. However the thesis is about the cryptography, therefore we do not discuss more practical issues in details.

4 Partners

In following three chapters we will analyse and describe the cryptography used in the application by Partners versions 1.51.46 and 1.60.44.

As in the previous application, the first part describes protocols and algorithms, the second part is about practical reversing and the third part is the conclusion.

We also opened the bank account in Partners in a standard way in order to analyse the cryptography in the application. Interesting fact is that the bank does not offer internet banking. The only way how to manage the account online is via the mobile application.

We also tried to contact someone responsible to ask him whether we could send a request to their server in order to check the analysis. We did not received an reasonable answer thus we omitted this experiment. However we checked most of the algorithms by their implementation and comparing our result values with captured values.

This chapter consists of three sections - theoretical, practical and conclusion. In examples of the communication we rewrote all sensitive information by stars "****".

4.1 Protocol description

In this section we describe theoretical analysis of the application by Partners.

The first is the description of activation status exchange which is an operation that is usually performed several times during the session.

The second is the description of login and transaction.

The last is the description of the account linknig/registration of the device that is called Activation. Activation is described only briefly.

The application uses PowerAuth protocol by Wultra. Cryptographic operations are implemented in the native library from Wultra which has a documentation on Github [60]. We use this documentation to help us to analyse the crypto. We describe the reversing of the library in the next section.

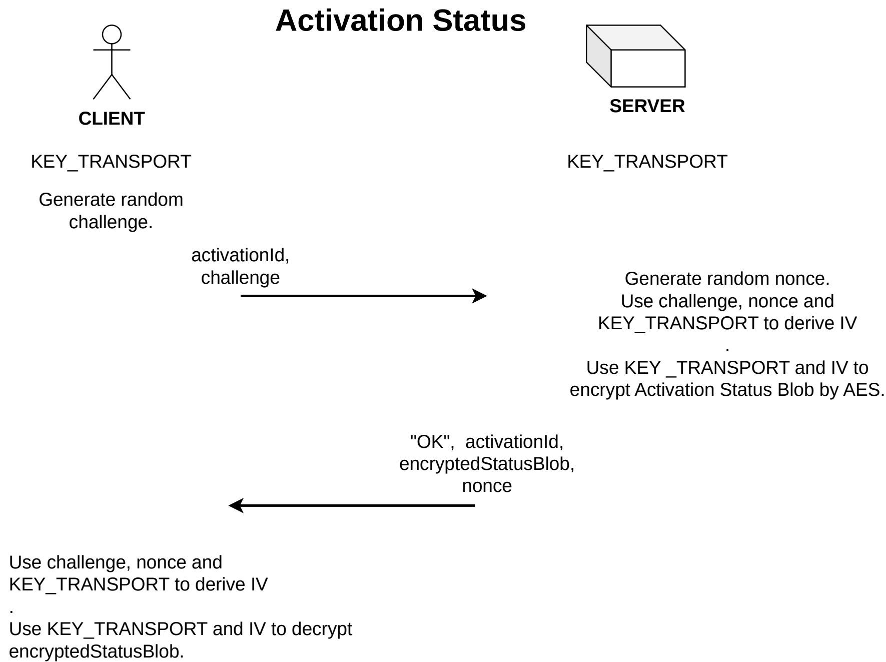
4.1.1 Activation Status

The activation status is a status sent from the server to the client. The status says whether the client is active, blocked or non-active and the client can display the information based on that to the user. The activation status decides the screen that is shown to the user.

The activation status is sent before the login at the start of every session and then during the session and when the user logs out. It is also sent during the transaction operation. That is the reason why we describe sending of the activation status in this separate section.

Overview

The communication proceeds as follows:



Activation Status Blob

The activation status is sent encrypted as a part of Activation Status Blob. The Activation Status Blob has the following structure. The description is taken from the documentation [60].

```
0xDECODED1 1B:${STATUS} 1B:${CURRENT_VERSION}
1B:${UPGRADE_VERSION} 5B:${RESERVED}
1B:${CTR_BYTE} 1B:${FAIL_COUNT}
1B:${MAX_FAIL_COUNT} 1B:${CTR_LOOK_AHEAD}
16B:${CTR_DATA_HASH}
```

The Status Blob has 32 bytes.

The start of Status Blob is the word "decoded", "1" is the version of Status Blob format.

`${STATUS}` is a byte that describes the activation status. It says the client which information it should display to the user. The STATUS can be one of the following options:

- `0x01 - CREATED`
- `0x02 - PENDING_COMMIT`
- `0x03 - ACTIVE`
- `0x04 - BLOCKED`
- `0x05 - REMOVED`

The usual value is `ACTIVE - 0x03`.

`${CURRENT_VERSION}` is a version of the used cryptographic protocol, in our case `0x03`, because the client and server use PowerAuth version "3.1" which is visible in the captured communication.

`${UPGRADE_VERSION}` is the maximal version of PowerAuth protocol that is supported, in our case `0x03`.

`${RESERVED}` are five bytes reserved for some future use, in the current version they have no meaning.

`${CTR_BYTE}` is the least significant byte of the current value of the counter.

`${FAIL_COUNT}` is a byte that gives us the number of failed attempts.

`${MAX_FAIL_COUNT}` the maximal allowed number of failed attempts. In our case it is equal to `0x05`.

`${CTR_LOOK_AHEAD}` the number that gives the number of iterations that server looks ahead when it validates the signature. In our case this value is equal to `0x14`. That is equal to 20 in decimal.

`${CTR_DATA_HASH}` hash of the current value of the counter (the server side), the value is 16 bytes long. We did not checked, however we can assume that the hash function will be the same as the function used to iterate the counter. The iteration of the counter is describe in 4.1.2

Algorithm

Activation Status

Before sending the activation status, client and server have shared secret key KEY_TRANSPORT. According to the documentation this key should be stored in the same manner as the KEY_SIGNAURE_POSSESION, which we describe in the Section 4.1.2. The following code was taken from the documentation [60] (with slight adjustments because of readibility). Key derivation functions used in the code are described below.

Program 1 Ecryption/decryption of Activation Status Blob

Let us denote concatenation by ||.

1. Both, client and server calculate KEY_TRANSPORT_IV as:

```
SecretKey KEY_TRANSPORT_IV = KDF(KEY_TRANSPORT, 3000)
```

2. Client chooses random 16 bytes long STATUS_CHALLENGE and sends that value to the server:

```
byte[] STATUS_CHALLENGE = Generator.randomBytes(16)
```

3. Server chooses random 16 bytes long STATUS_NONCE and calculates STATUS_IV as:

```
byte[] STATUS_NONCE = Generator.randomBytes(16)
byte[] STATUS_IV_DATA =  STATUS_CHALLENGE || STATUS_NONCE
byte[] STATUS_IV = getBytes(KDF_INTERNAL(
                    KEY_TRANSPORT_IV, STATUS_IV_DATA))
```

4. Server uses KEY_TRANSPORT as key and STATUS_IV as IV to encrypt the status blob:

```
encryptedStatusBlob = AES.encrypt(statusBlob, STATUS_IV,
                        KEY_TRANSPORT, "AES/CBC/NoPadding")
```

5. Server sends encryptedStatusBlob and STATUS_NONCE as response to the client.

6. Client receives encryptedStatusBlob and STATUS_NONCE and calculates the same STATUS_IV and then the client decrypts the status data:

```
byte[] STATUS_IV_DATA = STATUS_CHALLENGE ||STATUS_NONCE
byte[] STATUS_IV = getBytes(KDF_INTERNAL(
                     KEY_TRANSPORT_IV, STATUS_IV_DATA))
byte[] statusBlob = AES.decrypt(
                        encryptedStatusBlob, STATUS_IV,
                         KEY_TRANSPORT, "AES/CBC/NoPadding")
```

KDF

The following code was taken from the documentation [60]:

```
public SecretKey kdfDeriveSecretKey(SecretKey secret, long index)
{
byte[] bytes = ByteBuffer.allocate(16).putLong(index).array();
byte[] iv = new byte[16];
byte[] newKeyBytes = AES.encrypt(bytes, iv, secret);
return KeyConversion.secretKeyFromBytes(newKeyBytes);
}
```

The first row of the KDF function in above code takes the index, converts it to hexadecimal value. Then it adds padding with zeros from the left to get 16 bytes and puts it in the byte array.

The second row creates an array of 16 zero bytes.

The encryption is done by AES-128 CBC without padding.

KDF internal

The following code was taken from the documentation [60]:

```
public SecretKey deriveSecretKeyHmac(SecretKey secret,
                                                 byte[] index)
{
byte[] derivedKey = Mac.hmacSha256(secret, index);
byte[] newKeyBytes = ByteUtil.convert32Bto16B(derivedKey32);
return KeyConversion.secretKeyFromBytes(newKeyBytes);
}
```

The conversion of 32 bytes to 16 bytes is done by XORing the first half of the value with the second half of the value byte by byte.

Example

```
POST /pid-enrollment/v1/pa/v3/activation/status HTTP/2
Host: api.pbapi.cz
...
User-Agent: cz.pbktechnology.partners.client/1.60.40
            PowerAuth2/1.7.12
            (Android 12, Samsung SM-N935F)
...

{"requestObject":
  {"activationId":"dc69576b-8a39-48d7-a145-7523fff02f50",
   "challenge":"9g2xpjxwBR3OxCXOQHE2Ng\u003d\u003d"}}
```

```
HTTP/2 200 OK
...

{"status":"OK",
 "responseObject":
  {"activationId":"dc69576b-8a39-48d7-a145-7523fff02f50",
   "encryptedStatusBlob":"j1W5NXs4Yx8HW6p7N6TiI1kS9AFVi+D
                          gFXaWW/8xRgc=",
   "nonce":"EAQLFZPdLfymVj7j8UsROw==",
   "customObject":
        {"activationFlags":[
          "PID_UUID=1a6e04a0-8e0d-4cad-99e2-bec0ba1f0bc7",
          "PROCESS_COMPLETION_TIME=2025-04-11T15:53:00.475274",
          "PROCESS_TYPE=IDENTITY_VERIFICATION",
```

```
        "PROCESS_UUID=d020f815-b7b5-4edc-833e-eac961f0e667"]}
}}
```

Implementation

In application

The decoding of the encrypted Activation Status Blob by the client is done in the library `libPowerAuth2Module.so` in the function called `Session_decodeActivationStatus`. We describe the reversing of the library in the next chapter.

Our implementation

We implemented the decoding of the Activation Status Blob in `activation_code_decrypt.py` which we attach A.1.

Conclusion - Security

The algorithm is the same as it is described in the documentation, which we checked by reversing the corresponding code in Ghidra. We captured the TRANSPORT_KEY and values sent in the communication, we implemented the algorithm and we verified that the activation status was decoded correctly. We describe the reversing in the next chapter.

KDF functions in this section are not used to generate a secret key, but to generate IV. Therefore we will not evaluate them as key derivation functions. However, the usage of HMAC in "KDF_internal" is correct for key derivation, therefore it is good enough for IV derivation. Since the message (challenge + nonce) is always different, the output IV is alway different. "KDF" derives the secret value from the secret key KEY_TRANSPORT. We could not find any reason for this derivation. The KEY_TRANSPORT could be used as an input value to KDF_internal directly. However, this derivation is not bad, only unnecessary.

According to [30], IV for AES CBC has to be unpredictable, but it does not have to be secret. It says that there are two recommended ways to generate an unpredictable IV: "The first method is to apply the forward cipher function, under the same key that is used for the encryption of the plaintext, to a nonce. The nonce must be a data block that is unique to each execution of the encryption operation. For example, the nonce may be a counter, as described in Appendix B, or a message number. The second method is to generate a random data block using a FIPS- approved random number generator."

The usage of HMAC in the application for the derivation of IV using key dependent value as a secret is similar to the first recommended derivation of IV.

The server could simply generate a random IV using a FIPS-approved random number generator and send it with the encrypted data. The integrity would be satisfied - if IV would be changed, then the client could not decrypt Status Blob correctly. However, in [30] there is nothing about the protocol including both sides. We observed that the challenge value adds some resistance against the attack, where the attacker captures the encryptedStatusBlob and nonce from the server. If the challenge would not be included, then the attacker could pretend to

be the server and send this response to the client in different session. From this point of view the usage of challenge and nonce separately makes sence and it is more secure.

Since the length of the encrypted Status Blob is 32 bytes, the application does not have to use padding for the encryption. This is satisfied. The CBC mode requires unpredictable IV which is also satisfied by the protocol. The integrity of IV should be protected according to [30]. The integrity is theoretically protected - if challenge or nonce would be changed during the transfer, then the client could not decrypt the received Status Blob correctly.

Practically, we changed nonce when capturing the communication and the client did nothing, therefore even if the integrity of IV is theoretically satisfied, practicaly it is not.

The encryption of Status Blob is a little bit more complicated than needed, but for given purpose it works correctly. There is no need to derive a different secret key from KEY_TRANSPORT using "KDF".

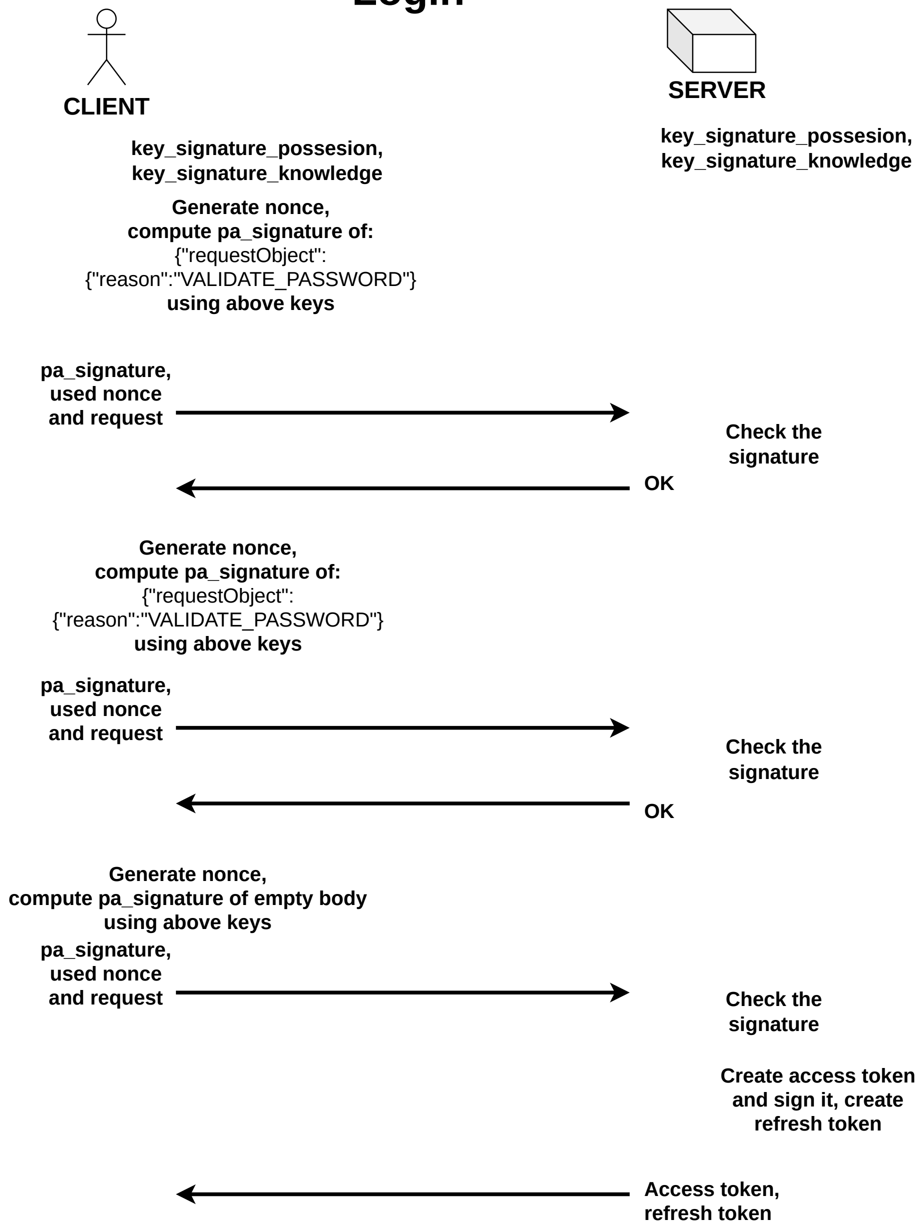
This request and response with status can be also theoretically seen as an authentication of the server. No one else can encrypt the StatusBlob. The attacker can resend this value, but that is all. He cannot use it in different session because it corresponds to the received challenge. He also cannot modify it, because the client would not be able to decrypt it correctly.

Practically, the client does nothing when receiving the wrong encrypted Activation Status Blob. We tried to modify the response from the server to see what happens. Hence the application does not use this cryptogrpahic part as an authentication.

4.1.2 Login

Overview and description

Firstly the client asks for the activation status as it is described in the previous section. After receiving the status, the login operation proceeds as follows:



Firstly, the client signs the request using shared secret pair of keys it has. The server checks the signature. If correct, it responds with "OK". This is done twice. We could not find any reason why this should be repeated, using different nonce generated by the client and the same request data.

Then the client sends a request requiring access tokens, equivalently, he sends a request to log in. The request has an empty body and it is signed. The server responds with a JWT access token and a refresh token. The same access token is then sent to the server with every request. The client proves that he is logged in by sending this token.

pa_signature

In this section we describe the signature algorithm. The algorithm is part of the login potocol.

Firstly we describe how keys are derived and stored and what is the hash based counter used in the signature. Then we show how to obtain input data to the signature in the subsection Create data to sign.

Keys derivation and storage

The signature algorithm uses two keys: `key_signature_possesion` and `key_signature_knowledge`. The derivation of keys is described in the documentation on Github [60]. The output of the Activation operation (linking the device with the user's account) is one shared master secret KEY_MASTER_SECRET. From this value two keys are derived. The following code is taken from the documentation, where the key derivation is described.

```
SecretKey KEY_SIGNATURE_POSSESSION = KDF.derive(
                                        KEY_MASTER_SECRET, 1);
SecretKey KEY_SIGNATURE_KNOWLEDGE = KDF.derive(
                                        KEY_MASTER_SECRET, 2);
```

The function used for key derivation (KDF) was described in 4.1.1. However, the application stores derived vaues and their does not have to be repeated. Keys are: key_signature_possesion and key_signature_knowledge. Both values are 128 bits long.

Since our application uses both these keys for signing we need to obtain them from storage when starting the signature algorithm. Both keys are stored encrypted by AES-128 CBC without padding where IV are zero bytes.

The signature possesion key is encrypted using a value connected to the device as an encryption key, it is 128 bits long.

The signature knowledge key is encrypted by the key derived from PIN as follows:

$$\text{AES_key_knowledge} = \text{PBKDF2}(\text{PIN}, \text{salt}, 10\,000)$$

where the PRF in PBKDF2 is not HMAC-SHA256, but HMAC-SHA1. The PBKDF2 function uses 10 000 iterations. The PBKDF2 is described in the chapter Theoretical background 7. The IV are zero bytes. This encryption corresponds to the description written in the documentation. Therefore obtaining signature keys from the storage can be described in the pseudocode as:

Algorithm 29 get_signature_keys

INPUT: PIN, device_related_key, encrypted_key_signature_possesion, encryted_key_signature_knowledge
OUTPUT: key_signature_possesion, key_signature_knowledge

1: AES_key_knowledge = PBKDF2(PIN, salt, 10 000)
2: AES_IV = 16 zero bytes
3: key_signature_possesion = AES_CBC_decrypt(device_related_key, AES_IV, encrypted_key_signature_possesion)
4: key_signature_knowledge = AES_CBC_decrypt(AES_key_knowledge, AES_IV, encryted_key_signature_knowledge)

Hash based counter

The signature algorithm uses a hash based counter which is iterated as follows:

Algorithm 30 Iterate_counter

INPUT: CTR - counter, hex string, 16 bytes
OUTPUT: iterated counter CTR

new_counter = SHA256(CTR)
new_counter_part1 = first 16 bytes of new_counter
new_counter_part2 = last 16 bytes of new_counter
new_counter = new_counter_part1 XOR new_counter_part2
return new_counter

The algorithm used in the module corresponds to its documentation (version of the protocol 3.1+), which we verified. The counter is iteraterd after every signature. The initial random value of the counter is set during the Activation.

Create data to sign

In this paragraph we describe how to create data to be signed. We remind that the signature algorithm is used to sign HTTP requests. The implementation corresponds with the documenetation, therefore we used the code written in the documentation. Firstly, we need following input parameters, some of them are part of the sent request:

- **REQUEST METHOD** - usually "POST"
- **REQUEST_URI_IDENTIFIER** - identifier of given URI of the resource encoded as Base64, the resource is for example "/auth/v2/pid-login" when logging in
- **APPLICATION_SECRET** - 16 bytes encoded by base64, identifies the application
- **NONCE** - 16 bytes encoded by base64, the nonce is generated by the client

- **REQUEST_BODY** - body of the signed request, transformed to bytes and encoded by base64. If there is no body, the algorithm uses "{}" instead.

Then data to be signed denoted by DATA are created as follows:

$$\begin{aligned} \text{REQUEST_DATA} = &\text{REQUEST_METHOD} \\ &|| \text{ REQUEST_URI_IDENTIFIER} \\ &|| \text{ NONCE} \\ &|| \text{ REQUEST_BODY} \end{aligned}$$

$$\text{DATA} = \text{REQUEST_DATA} \ || \ \text{APPLICATION_SECRET},$$

where || denotes the concatenation.

Signature

The signature is performed as it is described in the documentation. We provide the pseudocode for our specific case. Before starting the signature we need the DATA that are created as described above and we also need keys - possesion and knowledge - that are created as it is described above. We need to know the value of the hash based counter CTR which is also described above.

Firstly, we obtain signature components which are used to create the signature. The following algorithm is the algorithm from the documentation written for our specific case, which is reason why it looks differently.

Algorithm 31 pa_signature - get_signature_components

INPUT: DATA, CTR, key_signature_possesion, key_signature_knowledge
OUTPUT: signature_components

```
derived_key = HMAC(key_signature_possesion, CTR)
component1 = HMAC(derived_key, DATA)

derived_key1 = HMAC(key_signature_knowledge,CTR)
derived_key2 = HMAC(key_signature_knowledge, CTR)
derived_key = HMAC(derived_key1,derived_key2)
component2 = HMAC(derived_key,DATA)

signature_components=[component1, component2]
return signature_components
```

HMAC functinon uses SHA256 as PRF.

The next step is using signature components to produce a signature. This is done simply as follows:

Algorithm 32 pa_signature - signature

INPUT: signature_components
OUTPUT: signature

```
1: signature = ""
2: for (i=0, i<2, i=i+1) do
3:     component = last 16 bytes of signature_components[i]
4:     append component to signature
5: end for
6: signature = encode signature by base64
7: return signature
```

The result of the signature algorithm can be seen in the request header as "pa_signature"

Tokens

In this section we describe tokens used as a proof that the user is logged in.

Tokens sent between the client and the server are JWT tokens, therefore they have a standard structure. A general structure of tokens was described in 1.2.

Description

The access token is meant to prove that the client is logged in. It is valid for limited time. This time limit is specified in the response from server. The refresh token is sent when the access token expires. The server then sends new access and refresh tokens.

From the below example we can see that in the access token the signature is performed using RS256, which stands for RSA with SHA256. More specifically, the algorithm uses RSA with PKCS1 v1.5 paddding. That means the input is hashed by SHA256 and then it is signed by the server's private key.

The refresh token uses HS512 which stands for HMAC with SHA512 used as PRF.

Both algorihms are specified in RFC 7518 [61].

Example

The example of two tokens in the captured communication in the next section 4.1.3 can be decoded to:

- **Access token**


```
{
    "alg": "RS256",
    "typ": "JWT",
    "kid": "Ay1_24xWfNYzWYsuaXw25tCqgUKyIMqo570Ch5OuGwc"
}
```

```
{
    "exp": 1747305212,
```

```
    "iat": 1747304912,
    "jti": "52e8cd97-8cc7-41f3-81b6-e62edc8fdb1a",
    "iss": "https://login-pid.pbapi.cz/realms/pid",
    "aud": [
      "world-daily",
      "world-pid",
      "account"
    ],
    "sub": "64b8b9ce-42f8-42ee-ac87-c35d1ca0e3d9",
    "typ": "Bearer",
    "azp": "frontend-mobile-app",
    "sid": "9d4385ac-6c2f-44ea-a208-b2de5f8fa9a1",
    "acr": "1",
    "realm_access": {
      "roles": [
        "default-roles-pid",
        "offline_access",
        "uma_authorization"
      ]
    },
    "resource_access": {
      "world-daily": {
        "roles": [
          "DAILY_ACTIVATED"
        ]
      },
      "world-pid": {
        "roles": [
          "PID_ACTIVATED"
        ]
      },
      "account": {
        "roles": [
          "manage-account",
          "manage-account-links",
          "view-profile"
        ]
      }
    },
    "scope": "email pid_attributes profile",
    "email_verified": false,
    "user_attributes": {
      "pidUuid": "1a6e04a0-8e0d-4cad-99e2-bec0ba1f0bc7"
    },
    "preferred_username": "1a6e04a0-8e0d-4cad-99e2-bec0ba1f0bc7"
}
```

- **Refresh token**

```
{
    "alg": "HS512",
```

```
    "typ": "JWT",
    "kid": "c00d9025-9f39-4323-9048-8b15c61154c7"
}
```

```
{
"exp": 1747306712,
"iat": 1747304912,
"jti": "44b1c85f-6e3b-405f-9f46-7c16799cf221",
"iss": "https://login-pid.pbapi.cz/realms/pid",
"aud": "https://login-pid.pbapi.cz/realms/pid",
"sub": "64b8b9ce-42f8-42ee-ac87-c35d1ca0e3d9",
"typ": "Refresh",
"azp": "frontend-mobile-app",
"sid": "9d4385ac-6c2f-44ea-a208-b2de5f8fa9a1",
"scope":"email pid_attributes basic profile web-origins acr roles"
}
```

For decoding we used the decoder on website jwt.io [12].

Login protocol

The login protocol uses the pa_signature and all needed values were described above in the section about pa_signature. To summarize it, the login protocol is the following:

Algorithm 33 Login

CLIENT: application_secret, counter CTR
CLIENT and SERVER: Same shared keys: key_signature_possesion and key_signature_knowledge

---

for (i=0,i<2,i++) **do**
  CLIENT: method = "POST", uri_identifier = "/pa/signature/validate"
  CLIENT: pa_nonce = random 128 bit nonce
  CLIENT: body = ""requestObject":"reason":"VALIDATE_PASSWORD""
  CLIENT: Create input data to the signature algorithm using body, pa_nonce, application_secret and request details.
  CLIENT: Compute pa_signature using input data, CTR, key_signature_possesion and key_signature_knowledge.
  CLIENT: Iterate counter CTR.
  CLIENT: Send pa_signature,nonce and application_secret to SERVER
  SERVER: Validate the signature, if it is correct, return "OK"
end for
CLIENT: method = "POST", uri_identifier = "/auth/v2/pid-login"
CLIENT: pa_nonce = random 128 bit nonce
CLIENT: body = "{}"
CLIENT: Create input data to the signature algorithm using body, pa_nonce, application_secret and request details.
CLIENT: Compute pa_signature using input data, CTR, key_signature_possesion and key_signature_knowledge.
CLIENT: Iterate counter CTR.
CLIENT: Send pa_signature, nonce and application_secret to the SERVER
SERVER: Validate the signature, if it is correct, generate a JWT access token and a refresh token.
SERVER: Send tokens to CLIENT.

We note that before above protocol starts, server and client exchange the request and response with the activation status described in the previous section 4.1.1.

Implementation

The signature is implemented in the method signHTTPRequest of the class Session. This method is called from the native library which we describe in the next chapter.

We implement the pa_signature in `pa_signature.py` which we attach A.1.

Conclusion - security

The login protocol uses pa_signature to prove that the user knows the password.

In this section we make a conclusion about the used cryptography.

We will start the conclusion by key derivation and storage and then we will continue with the pa_signatue and with the login protocol.

The signature algorithm uses two keys. Both keys are encrypted using AES and were derived as it is described in 4.1.2. AES encryption uses zero bytes as IV. This is not unpredictable, however in this case it is not an issue since the only value encrypted by the same key is the plaintext that is derived form random master_secret.

According to the documentation the first AES key is related to the device. We managed to obtain it during our analysis.

The second key is encrypted by the key derived from the PIN. The function that is used to derive the AES key is PBKDF2 with HMAC-SHA1 and 10 000 operations. This does not correspond to the recommendation given by OWASP [56] where PBKDF2 with HMAC-SHA256 and at least 600 000 iterations are recommended.

On the other hand, the attack would require an attacker that would manage to capture the communication and obtain the value of the counter in the time of the communication. Then the attacker could guess PIN which has 6 digits, compute PBKDF2 and try to decrypt key by AES. Then the attacker would need to check whether the decrypted key_signature_possesion is correct by computing the signature and checking that it corresponds to the captured signature. This is not very probable, the value of the counter changes after every signature and there is no feasible method how to efficiently reconstruct it.

The pa_signatue implemented in the application probably corresponds to the algorithm written in the documentation. We cannot say if it is exactly the same since the algorithm in the documentation is also only pseudocode.

The algorithm pa_signature uses HMAC with SHA256 for derivation of keys, which is correct option according to recommendation given by BSI [28]. The algorithm also uses HMAC with SHA256 for the signature. The key length should be greater then or equal to 128 bits according to the same recommendation. The length of the output should be ≥ 128 according to redommendation by BSI [28] and ≥ 96 according to NÚKIB [53]. Both recommendations are satisfied.

The hash based counter ensures that the signature is not simply repeatable and that the derived key is refreshed after every signature. The periodic key refreshment is recommended in RFC 2104 [33].

We could not find out why the second component is derived in a more complicated way. It seems unnecessary, since the algorithm could use just one HMAC instead of three to derive the key as it is done with the first component.

Some randomness is added by using nonce.

The protocol implemented in the application starts by two signatures of the same body and then the client signs an empty body. This means that the signature is performed three times before the server sends JWT token. This does not make any sense, there is no reason to perform the same signature three times. By only one signature, the server only checks that the user has shared keys, key_signature_possesion and key_signature_knowledge. It also ensures the integrity of the message which is in our case ""requestObject":"reason":"VALIDATE_PASSWORD"" or "{}".

This login protocol is vulnerable to man in the middle attack. The server does not sign any responses and thus the client has no proof that he is communicating with the real server.

Theoretically, only values in the first response with the activation status 4.1.1

are guaranteed to be from the server. This answer also includes a hash of the current value of the counter. Practically, the application do nothing when the encrypted activation status is wrongly decrypted.

Therefore the attacker could pretend to be the server, capture first three login requests by the client. Then he would stop the communication and use the captured requests for his own login. In this moment the attacker knows correct signatures even if he does not know the counter and keys. This would not be possible if the nonce was generated and sent by the server. The server should also authenticate.

JWT tokens also can be captured and transfered together with a message changed by the attacker. Since the request/response is not signed, this is possible.

To summarize it, the pa_signture could be simplified by using HMAC to derive a key and then to use HMAC to sign DATA. This can be done for both keys. PBKDF2 could use the recommended function as SHA256 with more iterations even if the attack is not very probable. The protocol could be simplified by checking only one signature instead of three. But since the protocol in this form is vulnerable, we would recommend to use different protocol instead. We also recommend to sign every response/request sent by the client and sent by the server.

Alternative protocols that could be used are challenge-response protocols which use shared secret, such as already mentioned OCRA 2.6. The crucial is that the challenge (in our case nonce) is generated and sent by the server. The protocol can be run in both directions, to check both - the client and the server. The used protocol should also be resistant against replay attacks, thus the usage of the counter is desirable and correct.

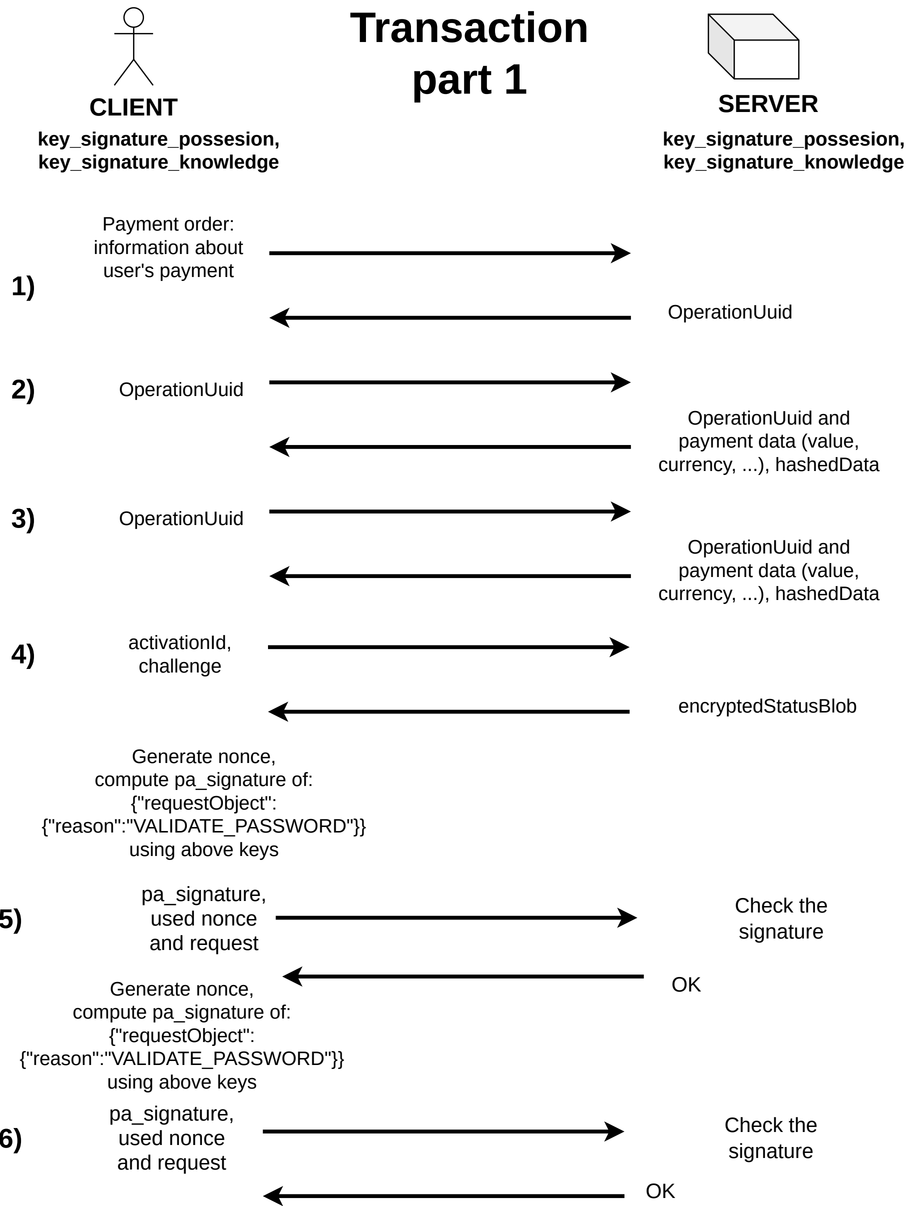
4.1.3 Transaction

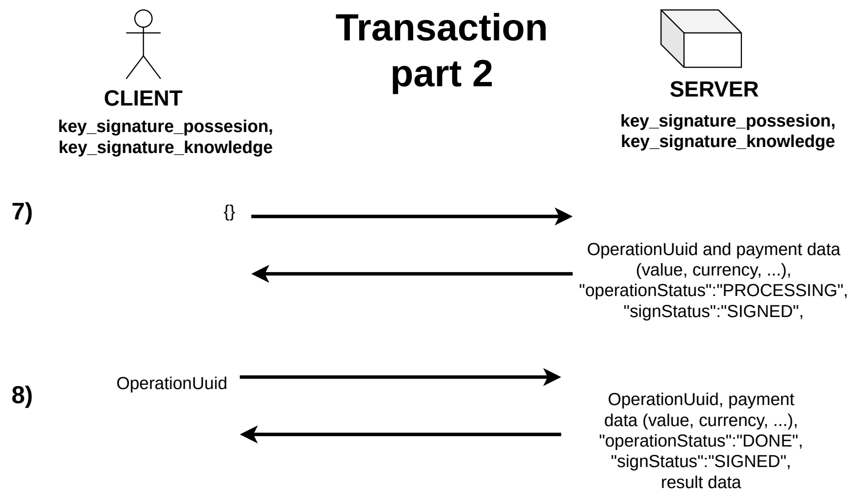
Overview and description

The transaction also uses pa_signature.

Transaction

The transaction proceeds as follows:


CLIENT
key_signature_possesion,
key_signature_knowledge
Transaction
part 1
SERVER
key_signature_possesion,
key_signature_knowledge
1)
Payment order:
information about
user's payment
OperationUuid
2)
OperationUuid
OperationUuid and
payment data (value,
currency, ...), hashedData
3)
OperationUuid
OperationUuid and
payment data (value,
currency, ...), hashedData
4)
activationId,
challenge
encryptedStatusBlob
Generate nonce,
compute pa_signature of:
{"requestObject":
{"reason":"VALIDATE_PASSWORD"}}
using above keys
5)
pa_signature,
used nonce
and request
Check the
signature
OK
Generate nonce,
compute pa_signature of:
{"requestObject":
{"reason":"VALIDATE_PASSWORD"}}
using above keys
6)
pa_signature,
used nonce
and request
Check the
signature
OK



We will describe all steps.

In 1) the client sends information about the payment he wants to perform. The server responds with OperationUuid, which is the unique identifier for this operation.

In 2) the client sends OperationUuid and the server responds with the payment data with details such as the receiver, currency, date, client ip address, etc. It also sends the hash of data.

The 3) is exactly the same as 2). We do not know why this request is repeated twice.

In 4) the client asks for Activation Status, he sends the activation Id and challenge and receives encryptedStatusBlob and nonce. We describe this request and response in 4.1.1.

In 5) the signature is performed. The client generates the nonce and computes the signature of a request body, which is {"reason":"VALIDATE_PASSWORD"}. If the signature is correct, the server responds with "OK".

The 6) is exactly the same as 5) with different nonce. It is the same as in the login and we do not know the reason.

In 7) the client receives again payment data with status of the operation and the signature.

In 8) the operation is done and the client again receives the payment data. He also receives the result data, which are the same as the payment data.

Transaction protocol

In this section we give more detailed description. The transaction uses pa_signature which was already described in 4.1.2.

Algorithm 34 Transaction

CLIENT: application_secret, counter CTR
CLIENT and SERVER: Same shared keys: key_signature_possesion and key_signature_knowledge

1)
CLIENT: Send a request with requested operation "CREATE_PAYMENT_ORDER" to SERVER.
CLIENT: Send payment order to SERVER. The payment order contains the accountUuid, amount and currency, the number of recipient account etc.
SERVER: Give the opearation unique number operationUuid, and send it to CLIENT.
2) + 3)
for (i=0,i<2,i++) **do**
 CLIENT: Send server the request with received operationUuid.
 SERVER: Create "data", which is structure containing all information about the payment order such as details about the account, the paid value and currency, message for sender and receiver, the amount of money on the account after the payment, client's IP address etc.
 SERVER: Send data, operationUuid, time of creation and expiration, status of the operation which is "WAITING_FOR_SIGN", hash of data, etc. to CLIENT
end for
4)
CLIENT: Generate challenge. Send challenge, activationId to SERVER.
SERVER: Generate nonce. Use challenge, nonce and KET_TRANSPORT to encrypt Status Blob as described in 4.1.1.
SERVER: Send encryptedStatusBlob to CLIENT.
CLIENT: Decrypt encryptedStatusBlob.
5) + 6)
for (i=0,i<2,i++) **do**
 CLIENT: method = "POST", uri_identifier = "/pa/signature/validate"
 CLIENT: pa_nonce = random 128 bit nonce
 CLIENT: body = ""requestObject":"reason":"VALIDATE_PASSWORD""
 CLIENT: Create input data to the signature algorithm using body, pa_nonce, application_secret and request details.
 CLIENT: Compute pa_signature using input data, CTR, key_signature_possesion and key_signature_knowledge.
 CLIENT: Iterate counter CTR.
 CLIENT: Send pa_signature,nonce and application_secret to SERVER
 SERVER: Validate the signature, if it is correct, return "OK"
end for
7)
CLIENT: Send a request with empty body to SERVER.
SERVER: Send respose with the same data as before, operationUuid and with operationStatus "PROCESSING" to CLIENT
8)
CLIENT: Send request with operationUuid to SERVER
SERVER: Send response with operationStatus "DONE", with operation data which are same as before and with the result data which is a result of the operation. Result data are the same as data.

Parameters and storage

Parameters and storage of keys needed for pa_signature which is the only cryptographic operation in the transaction were already described in the previous section about login in 4.1.2.

Implementation

The transaction operation also uses pa_signature which is a implemented in the libPowerAuth2Module.so as we have already said in the section about login.

We have implemented pa_signature in `pa_signature.py` as we have already said.

Conlusion - security

The algorithm used for pa_signature is the same as before and therefore conclusions about the security of the algorithm and the storage of keys are same as in the Conclusion in the previous section 4.1.2.

The protocol is vulnerable to the man in the middle attack. We can see that even if it is claimed that the payment order data are signed, the only signatures that are computed and checked by the server are pa_signatures of ""requestObject":"reason":"VALIDATE_PASSWORD"". Therefore an attacker in the middle of communication can resend pa_signature request and responses and he can modify the payment order. He can for example rewrite the amount of money that is sent, therefore he can force the user to pay more money even if it is not visible in the application. We verified that this attack is possible by using Burp proxy.

The easiest solution to this problem could be to sign all messages in both directions. If the payment data would be signed, then the attacker could not modify them. There is no need to sign the message:
""requestObject":"reason":"VALIDATE_PASSWORD""
instead of payment order data. Another option would be to use classical asymmetric cryptography such as RSA or elliptic curves and sign the data (probably with time and the unique ID of the payment) by the private key of the client.

Communication

In this section we give an example of captured communication where the login and the transaction is visible.

Login

```
POST /pid-enrollment/v1/pa/v3/signature/validate HTTP/2
Host: api.pbapi.cz
...
X-Powerauth-Authorization: PowerAuth pa_version="3.1",
    ...
    pa_application_key="AbNrYAGrPLYvjGhRg7OuTg==",
    pa_nonce="rXLk9pZ98H9DNS5zCsWp/A==",
    pa_signature_type="possession_knowledge",
```

```
    pa_signature="lXJhDoLVPW3RhYrZp4DYo2OQARGdmNstqRfjbklOSas="
...
User-Agent: cz.pbktechnology.partners.client/1.51.46 PowerAuth2/1.7.10
             (Android 12, Samsung SM-N935F)
...

{"requestObject":{"reason":"VALIDATE_PASSWORD"}}
```

```
HTTP/2 200 OK
Date: Thu, 15 May 2025 10:28:31 GMT
...

{"status":"OK"}
```

```
POST /pid-enrollment/v1/pa/v3/signature/validate HTTP/2
Host: api.pbapi.cz
...
X-Powerauth-Authorization: PowerAuth pa_version="3.1",
        ...
        pa_application_key="AbNrYAGrPLYvjGhRg7OuTg==",
        pa_nonce="/n52qrjDeUhVFy4MHXF4dg==",
        pa_signature_type="possession_knowledge",
        pa_signature="T3Jxjynrzdc8Nxel3Xf6NUoMiNFhbw9AOIPaRLo3JSk="
Content-Type: application/json
User-Agent: cz.pbktechnology.partners.client/1.51.46 PowerAuth2/1.7.10
            (Android 12, Samsung SM-N935F)
...

{"requestObject":{"reason":"VALIDATE_PASSWORD"}}
```

```
HTTP/2 200 OK
Date: Thu, 15 May 2025 10:28:31 GMT
...

{"status":"OK"}
```

```
POST /pid-mobilegateway-auth/v2/pid-login HTTP/2
Host: api.pbapi.cz
...
X-Powerauth-Authorization: PowerAuth pa_version="3.1",
    ...
    pa_application_key="AbNrYAGrPLYvjGhRg7OuTg==",
    pa_nonce="Gk6Bzl2hrIOSu8PNVNCowA==",
    pa_signature_type="possession_knowledge",
    pa_signature="YiKA1/pOkyFC9CQTraQ2XSdQkNOeUBjV8QJOtouIInM="
...
```

```
HTTP/2 200 OK
Date: Thu, 15 May 2025 10:28:32 GMT
```

```
...

{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkI
                 iA6ICJBeTFfMjR4V2ZOWXpXWXN1YVh3MjV0Q3FnVUt5SU
                 1xbzU3MENoNU91R3djIn0.eyJleHAiOjE3NDczMDUyMTI
                 sImlhdCI6MTc0NzMwNDkxMiwianRpIjoiNTJlOGNkOTct
                 OGNjNyO0MWYzLTgxYjYtZTYyZWRjOGZkYjFhIiwiaXNzI
                 joiaHR0cHM6Ly9sb2dpbi1waWQucGJhcGkuY3ovcmVhbG
                 1zL3BpZCIsImF1ZCI6WyJ3b3JsZC1kYWlseSIsIndvcmx
                 kLXBpZCIsImFjY291bnQiXSwic3ViIjoiNjRiOGI5Y2Ut
                 NDJmOC00MmVlLWFjODctYzM1ZDFjYTBlM2Q5IiwidHlwI
                 ...
                 aeommWuAsTNpyCtuivKY30f_D_mCrLUMI3N_24D_TPXuL
                 83Hvz3_tmD75sQ3f25dium2hBAomgkdW4ZVcOB1pU5P7p
                 WCkkM1ll8RJIxnm0mQRohETEAJJLb3DkxCQKWPbbGFzxv
                 rGWzY0L-DqQakUHSgbFQ-Vjlozfnho5uamou8MhSFb7CO
                 JmODA1nmu2Jm7OUKk6sdpytKQqydLxDekbwW6eo1Ix_-m
                 uyJO1--RYl7VhYkES6-UqwciijsLiC4YXWVikZKCA",
"expires_in":300,
"refresh_token":"eyJhbGciOiJIUzUxMiIsInR5cCIgOiAiSldUIiwia2lkIi
                 A6ICJjMDBkOTAyNS05ZjM5LTQzMjMtOTA0OC04YjE1YzYx
                 MTU0YzcifQ.eyJleHAiOjE3NDczMDY3MTIsImlhdCI6MTc
                 0NzMwNDkxMiwianRpIjoiNDRiMWM4NWYtNmUzYi00MDVmL
                 TlmNDYtN2MxNjc5OWNmMjIxIiwiaXNzIjoiaHR0cHM6Ly9
                 sb2dpbi1waWQucGJhcGkuY3ovcmVhbG1zL3BpZCIsImF1Z
                 CI6Imh0dHBzOi8vbG9naW4tcGlkLnBiYXBpLmN6L3JlYWx
                 tcy9waWQiLCJzdWIiOiI2NGI4YjljZS00MmY4LTQyZWUtY
                 WM4Ny1jMzVkMWNhMGUzZDkiLCJ0eXAiOiJSZWZyZXNoIiw
                 iYXpwIjoiZnJvbnRlbmQtbW9iaWxlLWFwcCIsInNpZCI6I
                 jlkNDM4NWFjLTZjMmYtNDRlYS1hMjA4LWIyZGU1ZjhmYTl
                 hMSIsInNjb3BlIjoiZW1haWwgcGlkX2F0dHJpYnV0ZXMgY
                 mFzaWMgcHJvZmlsZSB3ZWItb3JpZ2lucyBhY3Igcm9sZXM
                 ifQ._EakCb7OUuSJKMZ6ZTDvqdzICI8IyCExpwneQv6TNC
                 5yqShuvt0E0INqprgsISF2t1FlYkm3AeQLFalENvmjRw",
"refresh_expires_in":1800,
"token_type":"Bearer",
"not-before-policy":0,
"session_state":"9d4385ac-6c2f-44ea-a208-b2de5f8fa9a1",
"scope":"email pid_attributes profile"}
```


Then the communcation continues. The client sends requests with obtained JWT access token.

Transaction

In this section we give an example of captured communication while performing transaction. We used QR code to send 1 Kč with a message "POKUS2".

```
POST /bank-customergateway-qr/v3/v3/bank/qr/payment-order/parse HTTP/2
Host: api.pbapi.cz
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
```

```
...

{"qrChain":"SPD*1.0*ACC:CZ**********************AM:1*CC:CZK
            *DT:20250518*PT:IP*MSG:POKUS2"}
```

```
HTTP/2 200 OK
...

{"amount":{"value":1,"currencyCode":"CZK"},
"paymentCodes":{},
"recipient":{"iban":"CZ**********************",
            "message":"POKUS2"},
"dueDate":"2025-05-18"}
```

```
POST /bank-customergateway-signs/v2/v2/signs/sign-list HTTP/2
...
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
...

{"requestedOperation":"CREATE_PAYMENT_ORDER"}
```

```
HTTP/2 200 OK
...

{"signList":[]}
```

```
POST /bank-customergateway-paymentorders/v3/v3/bank/payment-orders/
    prepare HTTP/2
Host: api.pbapi.cz
...
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
...

{...
"amount":{"value":1,"currencyCode":"CZK"},
"note":"",
"dueDate":"2025-05-18",
"paymentCodes":{"constant":null,"specific":null,
                "variable":null},
"recipient":{"iban":"CZ**********************","message":"POKUS2"}}
```

```
HTTP/2 200 OK
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941"}
```

```
POST /bank-customergateway-signs/v2/v2/signs/sign-detail HTTP/2
...
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
```

```
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941"}
```

```
HTTP/2 200 OK
Date: Sun, 18 May 2025 10:55:56 GMT
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941",
"operation":"CREATE_PAYMENT_ORDER",
"operationStatus":"WAITING_FOR_SIGN",
"data":"{\"pidUuid\":\"*******-****-****-****-************\",\"from
        AccountUuid\":\"********-****-****-****-************\",\"fro
        mAccountIban\":\"CZ**********************\",\"fromAccountNam
        e\":\"Běžný účet\",\"amount\":{\"value\":1.0,\"currencyCode\
        ":\"CZK\"},\"senderInformation\":{\"iban\":\"CZ************
        *********\",\"accountName\":\"Běžný účet\",\"accountUuid\":\
        "********-****-****-****-************\",\"messageForSender\"
        :\"\"},\"dueDate\":\"2025-05-18\",\"updatedDueDate\":\"2025-
        05-18\",\"paymentCodes\":{},\"recipient\":{\"iban\":\"CZ****
        *****************\",\"message\":\"POKUS2\",\"name\":\"ZENKN
        EROVÁ KAROLÍN\"},\"accountTransferPaymentType\":\"INSTANT\",
        \"parentSignatureRequired\":false,\"note\":\"\",\"availableB
        alance\":{\"value\":5,\"currencyCode\":\"CZK\"},\"threatMark
        \":{\"client_ip\":\"**.***.***.***\",\"app_session_id\":\"1c
        e5c7f8-bfb6-4e53-a035-49e3ad548ee1\",\"app_device_id\":\"1af
        29da3e4005663\",\"security_item_type\":\"password\",\"applic
        ation_id\":\"android\"},\"handle\":\"!YTseXSiwgFVmIddqJnyD3G
        \",\"bankRequestedOperation\":\"CREATE_PAYMENT_ORDER\"}",
"signStatus":"READY",
"authorizationMethod":"SECOND_FACTOR",
"createdAt":"2025-05-18T10:55:56.007896Z",
"hashedData":"$2a$10$kUtd0Y5jZy3Ho3V4qRHrVOpRHRyVqKHtH3/nMFWS/68asHC
             Rztx2W",
"expiration":"2025-05-18T11:00:55.936476Z",
"groupIndex":0}
```

```
POST /bank-customergateway-signs/v3/v3/signs/sign-detail HTTP/2
...
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941"}
```

```
HTTP/2 200 OK
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941",
"operation":"CREATE_PAYMENT_ORDER",
"operationStatus":"WAITING_FOR_SIGN",
"data":"{\"pidUuid\":\"*******-****-****-****-************\",\"from
```

```
        AccountUuid\":\"********-****-****-****-************\",\"fro
        mAccountIban\":\"CZ**********************\",\"fromAccountNam
        e\":\"Běžný účet\",\"amount\":{\"value\":1.0,\"currencyCode\
        ":\"CZK\"},\"senderInformation\":{\"iban\":\"CZ*************
        *********\",\"accountName\":\"Běžný účet\",\"accountUuid\":\
        "********-****-****-****-************\",\"messageForSender\"
        :\"\"},\"dueDate\":\"2025-05-18\",\"updatedDueDate\":\"2025-
        05-18\",\"paymentCodes\":{},\"recipient\":{\"iban\":\"CZ****
        ******************\",\"message\":\"POKUS2\",\"name\":\"ZENKN
        EROVÁ KAROLÍN\"},\"accountTransferPaymentType\":\"INSTANT\",
        \"parentSignatureRequired\":false,\"note\":\"\",\"availableB
        alance\":{\"value\":5,\"currencyCode\":\"CZK\"},\"threatMark
        \":{\"client_ip\":\"**.***.***.***\",\"app_session_id\":\"1c
        e5c7f8-bfb6-4e53-a035-49e3ad548ee1\",\"app_device_id\":\"1af
        29da3e4005663\",\"security_item_type\":\"password\",\"applic
        ation_id\":\"android\"},\"handle\":\"!YTseXSiwgFVmIddqJnyD3G
        \",\"bankRequestedOperation\":\"CREATE_PAYMENT_ORDER\"}",
"signStatus":"READY",
"authorizationMethod":"SECOND_FACTOR",
"createdAt":"2025-05-18T10:55:56.007896Z",
"hashedData":"$2a$10$kUtd0Y5jZy3Ho3V4qRHrVOpRHRyVqKHtH3/nMFWS/68asHC
             Rztx2W",
"expiration":"2025-05-18T11:00:55.936476Z",
"groupIndex":0}
```

```
POST /pid-enrollment/v1/pa/v3/activation/status HTTP/2
...
User-Agent: cz.pbktechnology.partners.client/1.51.46
            PowerAuth2/1.7.10 (Android 12, Samsung SM-N935F)
...

{"requestObject":{"activationId":"********-****-****-****-**
                                  **********",
                  "challenge":"m9jrveYvgTTt3A3jEiRQuw\u003d\u003d"}}
```

```
HTTP/2 200 OK
...

{"status":"OK",
"responseObject":{
  "activationId":"********-****-****-****-************",
  "encryptedStatusBlob":"MpxVtOHfHXA2auZJxxrffuoFSMbCK1Qm
                         aeHg6YsxRpw=",
  "nonce":"fpySwjh6+RGzr/lSjcFNrA==",
  "customObject":
    {"activationFlags":[
        "PID_UUID=1a6e04a0-8e0d-4cad-99e2-bec0ba1f0bc7",
        "PROCESS_COMPLETION_TIME=2025-04-11T15:53:00.475274",
        "PROCESS_TYPE=IDENTITY_VERIFICATION",
        "PROCESS_UUID=d020f815-b7b5-4edc-833e-eac961f0e667"]}}}
```

```
POST /pid-enrollment/v1/pa/v3/signature/validate HTTP/2
Host: api.pbapi.cz
X-Powerauth-Authorization:
    PowerAuth pa_version="3.1",
    ...
    pa_application_key="AbNrYAGrPLYvjGhRg7OuTg==",
    pa_nonce="/dZ9pXF1aq9wSmT5BmLjjA==",
    pa_signature_type="possession_knowledge",
    pa_signature="5pTtt0L8fs0QN/3f7haKFvFYW0QPxwToADFDNyo1Lbw="
Content-Type: application/json
User-Agent: cz.pbktechnology.partners.client/1.51.46
            PowerAuth2/1.7.10 (Android 12, Samsung SM-N935F)
...

{"requestObject":{"reason":"VALIDATE_PASSWORD"}}
```

```
HTTP/2 200 OK
...

{"status":"OK"}
```

```
POST /pid-enrollment/v1/pa/v3/signature/validate HTTP/2
Host: api.pbapi.cz
...
X-Powerauth-Authorization:
    PowerAuth pa_version="3.1",
    ...
    pa_application_key="AbNrYAGrPLYvjGhRg7OuTg==",
    pa_nonce="8IHK2Ic6MMCyy9QVj+Ws0Q==",
    pa_signature_type="possession_knowledge",
    pa_signature="a6cn1MuRoiu3zhC1W2XihxSXi4DXXvetpoaXytYuThs="
...
User-Agent: cz.pbktechnology.partners.client/1.51.46
            PowerAuth2/1.7.10 (Android 12, Samsung SM-N935F)
...

{"requestObject":{"reason":"VALIDATE_PASSWORD"}}
```

```
HTTP/2 200 OK
...

{"status":"OK"}
```

```
POST /bank-customergateway-signs/v2/v2/signs/
     sign-list-with-operation-data HTTP/2
Host: api.pbapi.cz
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
...

{}
```

```
HTTP/2 200 OK
Date: Sun, 18 May 2025 10:56:12 GMT
...

{"signList":[{
    "operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941",
    "operation":"CREATE_PAYMENT_ORDER",
    "operationStatus":"PROCESSING",
    "signStatus":"SIGNED",
    "data":"{\"pidUuid\":\"********-****-****-****-************\",
           \"fromAccountUuid\":\"********-****-****-****-********
           ****\",\"fromAccountIban\":\"CZ**********************\
           ",\"fromAccountName\":\"Běžný účet\",\"amount\":{\"val
           ue\":1.0,\"currencyCode\":\"CZK\"},\"senderInformation
           \":{\"iban\":\"CZ**********************\",\"accountNam
           e\":\"Běžný účet\",\"accountUuid\":\"********-****-***
           *-****-************\",\"messageForSender\":\"\"},\"due
           Date\":\"2025-05-18\",\"updatedDueDate\":\"2025-05-18\
           ",\"paymentCodes\":{},\"recipient\":{\"iban\":\"CZ****
           ******************\",\"message\":\"POKUS2\",\"name\":\
           "ZENKNEROVÁ KAROLÍN\"},\"accountTransferPaymentType\":
           \"INSTANT\",\"parentSignatureRequired\":false,\"note\"
           :\"\",\"availableBalance\":{\"value\":5,\"currencyCode
           \":\"CZK\"},\"threatMark\":{\"client_ip\":\"**.***.***
           .***\",\"app_session_id\":\"1ce5c7f8-bfb6-4e53-a035-49
           e3ad548ee1\",\"app_device_id\":\"1af29da3e4005663\",\"
           security_item_type\":\"password\",\"application_id\":\
           "android\"},\"handle\":\"!YTseXSiwgFVmIddqJnyD3G\",\"b
           ankRequestedOperation\":\"CREATE_PAYMENT_ORDER\"}",
    "expiration":"2025-05-18T11:00:55.936476Z"}]}
```

```
POST /bank-customergateway-signs/v3/v3/signs/sign-detail HTTP/2
...
Authorization: Bearer eyJhbGciOiJSUzI1Ni...IT7n0qXHbfaFxYk_yckwtbg
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941"}
```

```
HTTP/2 200 OK
...

{"operationUuid":"96bb6e54-ad4c-4d54-8228-718f7a281941",
"operation":"CREATE_PAYMENT_ORDER",
"operationStatus":"DONE",
"data":"{\"pidUuid\":\"********-****-****-****-************\",\"from
         AccountUuid\":\"********-****-****-****-************\",\"fr
         omAccountIban\":\"CZ**********************\",\"fromAccountN
         ame\":\"Běžný účet\",\"amount\":{\"value\":1.0,\"currencyCo
         de\":\"CZK\"},\"senderInformation\":{\"iban\":\"CZ*********
         *************\",\"accountName\":\"Běžný účet\",\"accountUui
```

```
         d\":\"********-****-****-****-************\",\"messageForSe
         nder\":\"\"},\"dueDate\":\"2025-05-18\",\"updatedDueDate\":
         \"2025-05-18\",\"paymentCodes\":{},\"recipient\":{\"iban\":
         \"CZ**********************\",\"message\":\"POKUS2\",\"name\
         ":\"ZENKNEROVÁ KAROLÍN\"},\"accountTransferPaymentType\":\"
         INSTANT\",\"parentSignatureRequired\":false,\"note\":\"\",\
         "availableBalance\":{\"value\":5,\"currencyCode\":\"CZK\"},
         \"threatMark\":{\"client_ip\":\"**.***.***.***\",\"app_sess
         ion_id\":\"1ce5c7f8-bfb6-4e53-a035-49e3ad548ee1\",\"app_dev
         ice_id\":\"1af29da3e4005663\",\"security_item_type\":\"pass
         word\",\"application_id\":\"android\"},\"handle\":\"!YTseXS
         iwgFVmIddqJnyD3G\",\"bankRequestedOperation\":\"CREATE_PAYM
         ENT_ORDER\"}",
"signStatus":"SIGNED",
"authorizationMethod":"SECOND_FACTOR",
"createdAt":"2025-05-18T10:56:13.745995Z",
"resultData":"{\"pidUuid\":\"********-****-****-****-************\",
              \"fromAccountUuid\":\"********-****-****-****-********
              ****\",\"fromAccountIban\":\"CZ**********************\
              ",\"fromAccountName\":\"Běžný účet\",\"amount\":{\"val
              ue\":1.0,\"currencyCode\":\"CZK\"},\"senderInformation
              \":{\"iban\":\"CZ**********************\",\"accountNam
              e\":\"Běžný účet\",\"accountUuid\":\"********-****-***
              *-****-************\",\"messageForSender\":\"\"},\"due
              Date\":\"2025-05-18\",\"updatedDueDate\":\"2025-05-18\
              ",\"paymentCodes\":{},\"recipient\":{\"iban\":\"CZ****
              ******************\",\"message\":\"POKUS2\",\"name\":\
              "ZENKNEROVÁ KAROLÍN\"},\"accountTransferPaymentType\":
              \"INSTANT\",\"parentSignatureRequired\":false,\"note\"
              :\"\",\"availableBalance\":{\"value\":5,\"currencyCode
              \":\"CZK\"},\"threatMark\":{\"client_ip\":\"**.***.***
              .***\",\"app_session_id\":\"1ce5c7f8-bfb6-4e53-a035-49
              e3ad548ee1\",\"app_device_id\":\"1af29da3e4005663\",\"
              security_item_type\":\"password\",\"application_id\":\
              "android\"},\"handle\":\"!YTseXSiwgFVmIddqJnyD3G\",\"b
              ankRequestedOperation\":\"CREATE_PAYMENT_ORDER\"}",
"expiration":"2025-05-18T11:00:55.936476Z",
"groupIndex":0}
```

JWT handling

In this subsection we give an example of two operations with JWT. The first is just an example of a request with access token. The second example shows the usage of the refresh token.

- Access token

```
GET /bank-kliprbff-fe/v2/bundle/utilization HTTP/2
Host: api.pbapi.cz
...
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSl
               dUIiwia2lkIiA6ICJBeTFfMjR4V2ZOWXpXWXN1YVh
```

```
                              ...
               fN1WDTCHY_28lKjnmXzE7yreh467OOTSBvqcrO9JQ
...
User-Agent: okhttp/4.12.0
```

```
HTTP/2 200 OK
Date: Fri, 16 May 2025 12:44:37 GMT
...

{"bundleName":"Pro jednoho",
"bundlePrice":{"value":0,"currencyCode":"CZK"},
"bundleProductCode":"OB_BUNDLE_INDIVIDUAL",
"pidBundleRole":"DRIVER",
...
```

- Refresh token

```
POST /realms/pid/protocol/openid-connect/token HTTP/1.1
Host: login-pid.pbapi.cz
...

client_id=frontend-mobile-app&
grant_type=refresh_token&
refresh_token=eyJhbGciOiJIUzUxMiIsInR5cCIgOiAiSldUIiwia
2lkIiA6ICJjMDBkOTAyNS05ZjM5LTQzMjMtOTA0OC04YjE1YzYxMTU0
YzcifQ.eyJleHAiOjE3NDc0MDEyNTksImlhdCI6MTc0NzM5OTQ1OSwi
anRpIjoiNGI3YWJkOTMtMzdiNC00YTU1LWExMzYtMmQ0YTdmMjAzYmI
2IiwiaXNzIjoiaHR0cHM6Ly9sb2dpbi1waWQucGJhcGkuY3ovcmVhbG
1zL3BpZCIsImF1ZCI6Imh0dHBzOi8vbG9naW4tcGlkLnBiYXBpLmN6L
3JlYWxtcy9waWQiLCJzdWIiOiI2NGI4YjljZS00MmY4LTQyZWUtYWM4
Ny1jMzVkMWNhMGUzZDkiLCJ0eXAiOiJSZWZyZXNoIiwiYXpwIjoiZnJ
vbnRlbmQtbW9iaWxlLWFwcCIsInNpZCI6IjI3YWVlOGJhLTM0MWMtND
VhMC1iMTlmLWM4ZGQ5ZTM1OGE2OSIsInNjb3BlIjoiZW1haWwgcGlkX
2F0dHJpYnV0ZXMgYmFzaWMgcHJvZmlsZSB3ZWItb3JpZ2lucyBhY3Ig
cm9sZXMifQ.XVNV6yNpR0ZXfIhjn_8oW3Q8UTvG_TCnqgubcaH-7xiX
4DMdXaLBmS7VdLwBDdiTCxElt4Yx5IgoZxnk6-rRzA
```

```
HTTP/2 200 OK
...

{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
                 ia2lkIiA6ICJBeTFfMjR4V2ZOWXpXWXN1YVh3Mj
                              ...
                 q1NwC5mriVrmA9RomLJje87BQu7DCxzqdAeyvuw",
"expires_in":300,
"refresh_expires_in":1800,
"refresh_token":"eyJhbGciOiJIUzUxMiIsInR5cCIgOiAiSldUIiwi
```

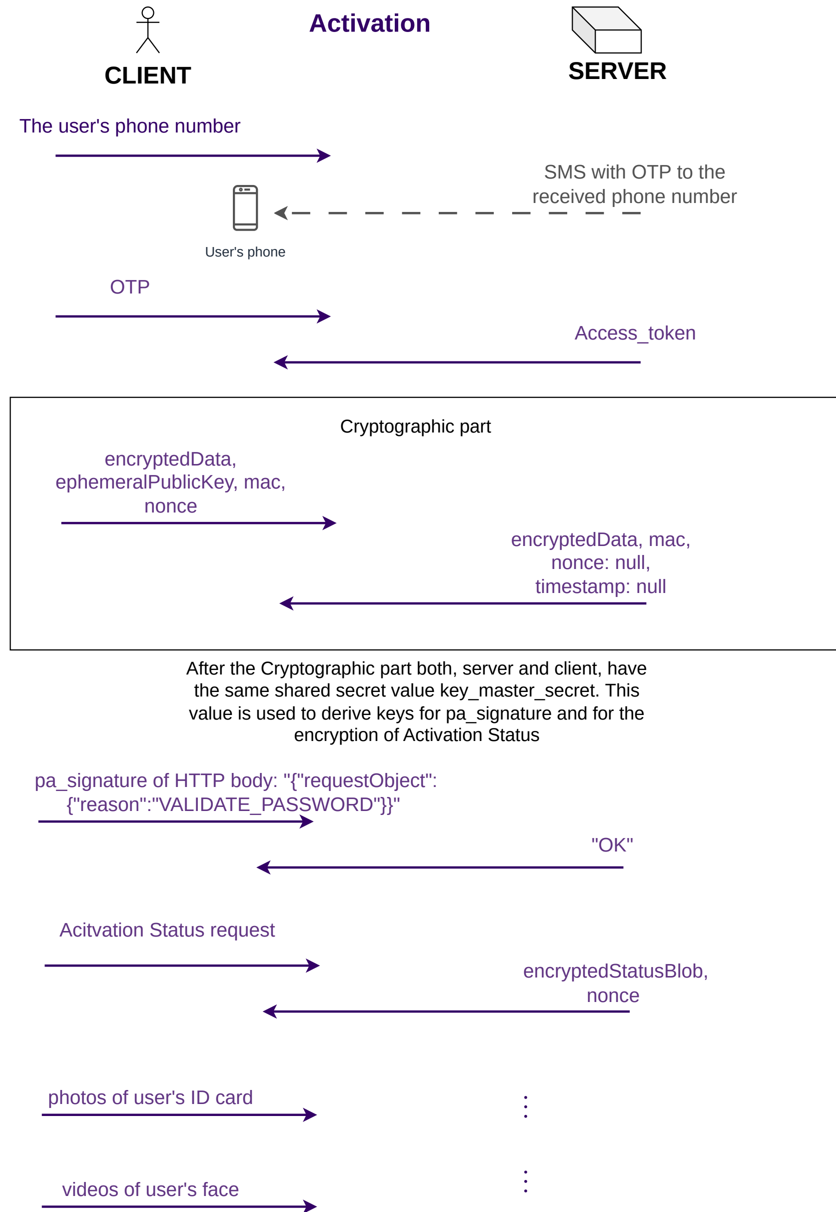
```
                a2lkIiA6ICJjMDBkOTAyNS05ZjM5LTQzMjMtOTA0
                                        ...
                6f-xiA6FK_prZexHwxEt9rbOePXYgo2kxI5tYyObg",
"token_type":"Bearer","not-before-policy":0,
"session_state":"27aee8ba-341c-45a0-b19f-c8dd9e358a69",
"scope":"email pid_attributes profile"}
```

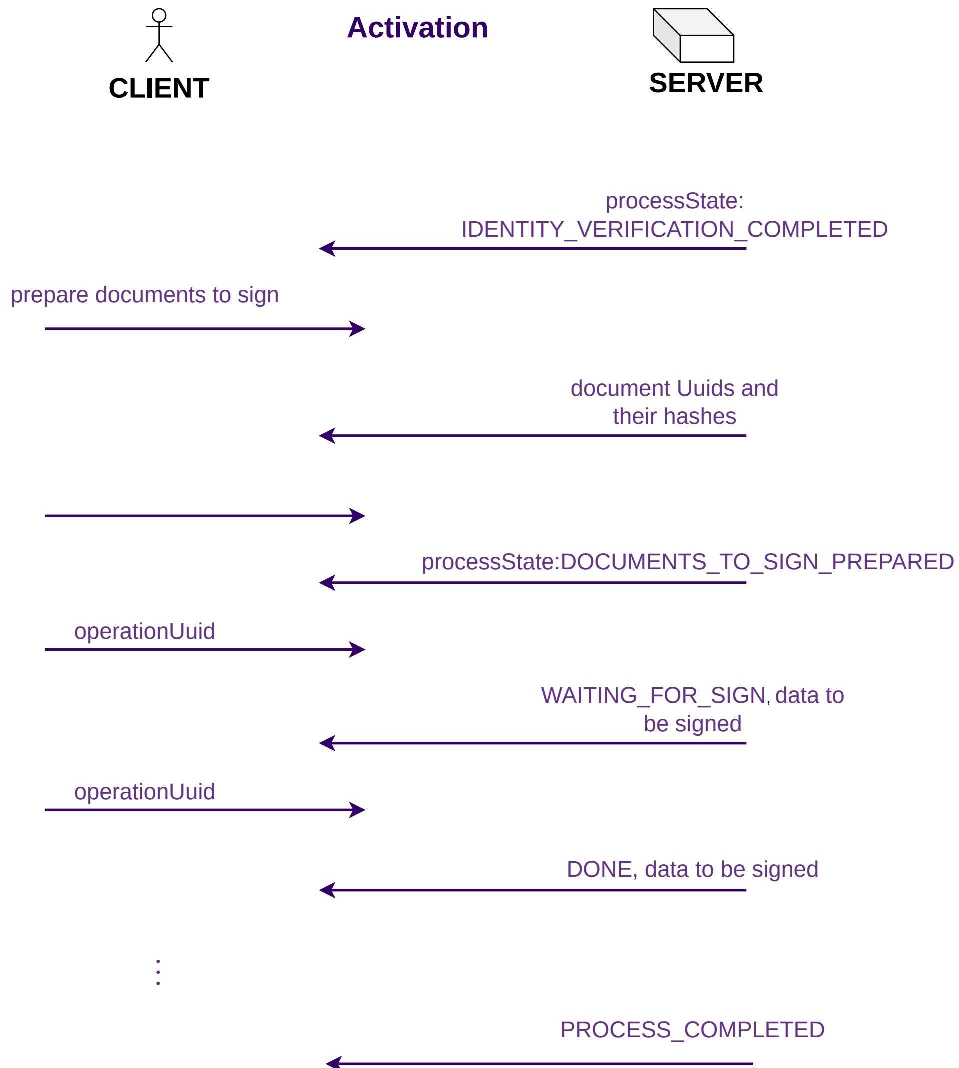
4.1.4 Activation

In this section we briefly describe activation. Due to time reasons, we did not analysed this part as carefully as all other sections. However, we used the captured communication and documentation [60] to understand the cryptographic scheme.

Overview

We provide a simplified scheme of the communication. We used the captured communication to create this scheme. The activation proceeds as follows:


Activation
CLIENT
SERVER
The user's phone number
SMS with OTP to the received phone number
User's phone
OTP
Access_token
Cryptographic part
encryptedData, ephemeralPublicKey, mac, nonce
encryptedData, mac, nonce: null, timestamp: null
After the Cryptographic part both, server and client, have the same shared secret value key_master_secret. This value is used to derive keys for pa_signature and for the encryption of Activation Status
pa_signature of HTTP body: "{"requestObject": {"reason":"VALIDATE_PASSWORD"}}"
"OK"
Acitvation Status request
encryptedStatusBlob, nonce
photos of user's ID card
videos of user's face



The result of the activation is shared `key_master_secret` which is an important value, since it is used to derive all shared cryptographic keys/secrets.

Cryptography in the activation

The Activation includes pa_signature which we have already described in 4.1.2 and Activation Status which we have described in 4.1.1. Following descriptions are based or straightly taken from the documentation. The result of the operation is `KEY_MASTER_SECRET` which is used to derive `KEY_TRANSPORT`, `KEY_SIGNATURE_POSSESSION` and `KEY_SIGNATURE_KNOWLEDGE` as follows:

```
KEY_SIGNATURE_POSSESSION = KDF.derive(KEY_MASTER_SECRET, 1);
KEY_SIGNATURE_KNOWLEDGE = KDF.derive(KEY_MASTER_SECRET, 2);
KEY_TRANSPORT = KDF.derive(KEY_MASTER_SECRET, 1000);
```

We have already described KDF function in 4.1.1.

The application creates the secret key using asymmetric scheme ECIES which is described in 2.10.

We give an example of the cryptographic part of the Activation, where the scheme is used to derive a key and to encrypt activation data:

```
POST /pid-enrollment/v1/pa/v3/activation/create HTTP/2
Host: api.pbapi.cz
...
X-Powerauth-Encryption: PowerAuth version="3.1"
                        application_key="AbNrYAGrPLYvjGhRg7OuTg=="
...

{"encryptedData":"mNptPquNihyCVmThvqg1I+ClaF2QbyscqF7O+Ik5di2OPU4H
                  RC6i1YTHP4GeIbL8EFwHJIl3pUlRf2EZwEaldvK3XimEZ0yE
                  ...
                  sArNxDkA5ySDmiITIIwTyAxYQBUX/k/t0t/2m81YYiZTlhMB
                  iyXuGKDE5VYn0mlTvufHiBw2Zjm3XzLqzAlKTY6A9/kEahjZ
                  crz4IbUpuYw\u003d",

"ephemeralPublicKey":"Apm3Tvz6bHKDes8yChP4I9UaCxR4CueZIKmyZysvhMho",
"mac":"O78eWG09rt6XDz1gYeBpsGeV/HJWOhd2o8AcOfVAe+Q\u003d",
"nonce":"ntjj9emFBsUPg+m+UIMm1w\u003d\u003d"}
```

```
HTTP/2 200 OK
Date: Sun, 15 Jun 2025 10:13:39 GMT
...

{"encryptedData":"mNptPquNihyCVmThvqg1I+ClaF2QbyscqF7O+Ik5di0V+YGM
                  McMh4fql0W0o64nFsKrgjLEKZg8ONwV5KyYiQWQc+H5Oh8/F
                  ...
                  DAC6v/L+owUsITowv6PfSaK9Vs0OsdVgHaK3oDAKcyzM35Qe
                  OxmKxBfxDgg=",
"mac":"KuYO0H+sYeonhh/oAFyHgB/HnJVm9MXobZb3cAlB4dI=",
"nonce":null,
"timestamp":null}
```

According to the documentation [60] the scheme ECIES is implemented using ECDH with secp256r1 curve as a key agreement function. The scheme uses X9.63 KDF function with SHA256 as a hash function. We have described the X9.63 KDF in 2.11.

The elliptic curve used in the algorithm is denoted by secp256r1. More precisely, it denotes elliptic curve domain parameters. The description can be found for example in [62]. According to [62] "sec" stands for "Standards for Efficient

Cryptography", "p" denotes that chosen parameters are over $\mathbb{F}_p$, "256" is the length in bits of the field size, "r" says that the parameters were chosen verifiably random and "1" is a sequence number. Recommended parameters can be found in the section 2.4.2 in [62]

According to the documentation, the master secret is the result of ECDH in ECIES scheme. The following code is taken from the documentation.

```
KEY_MASTER_SECRET = ByteUtils.convert32Bto16B(
                              ECDH.phase(KEY_SERVER_PRIVATE,
                                         KEY_DEVICE_PUBLIC))
```

The function that converts 32 to 16 bytes is the XOR of the left side of the bytearray with the right side of the bytearray. This is the most important result of the cryptography performed during the Activation since this secret is used to derive rest of secrets/keys.

The documentation also gives an idea about the encryption. According to the documentation the data that are sent from the client to the server are encrypted twice by ECIES. Also the response from the server is encrypted twice. Two layers are performed because the request by the client is firstly received by the enrollment server which is a front-end facing server and then it is transfered to PowerAuth server which is a hidden server application. During the activation the client sends activation data to the server. According to the documentation, activation data contains device name and extra attributes. The reponse from the server includes, among other things, an initial value of the hash based counter.

We have not found exact description of the structure of encrypted data. We can see that the first 2 blocks of encrypted data by the client and of encrypted data by the server are the same. The only symmetrical algorithm that is metioned in the documentation is AES CBC mode with 128 bit key. This means that the server uses the same key and also the same IV as the client. This does not correspond to the recommendation given by NIST. This should not be done - the IV should be unpredictable. However, we do not know exactly what is going on, because we have not examined it in detail.

Conclusion - security

We have not thoroughly examined the Activation and its implementation in the application, therefore we do not have proper coclusion. However, the ECIES scheme is recommended by BSI [28]. Algorithms that are used are in general recommended and ECIES seems as a good choice fot transfering initial state of the counter and creating master_secret. The captured communication seems a bit unexpected, since both encryted data have same two blocks. This does not have to be an isssue, we do not know enough to make a conclusion. The protocol is partially vulnerable against the man in the middle attack, since the client and the server do not have any shared secret in advance. Therefore the attacker could transfer the communication with OTP. Then he could theoretically continue establishing the shared master_key with the server, pretending to be the client. The attacker cannot pretend to be the server when establishing the shared

secret with the client, since he does not know server's private key. If the attaker would continue in the communication, the client would recognize that he is not communcating with the server and he would stop the communication. Therefore the attacker would not have any photos of ID or video and thus he could not convince the server that he is the real client.

4.2 Practical part

In this section we describe practical part of analysis. We note that the application uses PowerAuth protocol by Wultra.

4.2.1 Cryptographic libraries and sources

In this section we describe where to find the application file and the cryptograpic library used in the application. As in the previous case, the application file can be found in `/data/app/`. On our device we have the following folder:

```
/data/app/~~PvLY3ar9oe9F-KH-q3haTg==/
cz.pbktechnology.partners.client-EZTS-RJHNnfSJW7au8oMoA==
```

In this folder we can find base.apk file that can be used to obtain the decompiled code. Also there is a file split_config.arm64_v8a.apk from which we can extract a cryptographic library libPowerAuth2Module.so. We again used java decompiler jadx 1.3.2.

As in the previous application by Airbank, the library is loaded as follows:

```
static {
    System.loadLibrary("PowerAuth2Module");
}
```

This method is used several times in the code in different classes that implement PowerAuth protocol.

The protocol is fully implemented in the library, therefore we again used Ghidra 1.3.2 to obtain the readable code in pseudo C.

The library implements the PowerAuth protocol that was developed by Wultra [63]. There is a documentation on Github [60] describing all functions, classes and methods used in the library. Hence we had a clue what should we look for in the decompiled code.

When examining the library, we discovered that at least some parts of the library were taken from OpenSSL. The following code includes a string with EVP_MAX_MD_SIZE which is a constant used in OpenSSL [64].

```
20      return uVar1;
21    }
22                      /* WARNING: Subroutine does not return */
23    FUN_00214e00("assertion failed: ctx->digest->md_size <= EVP_MAX_MD_SIZE",&DAT_00151509,0);
24  }
25
```

Later, this was confirmed in the documentation, where is written that the client-side algorithms provider is OpenSSL (libCrypto).

4.2.2 Setup

The setup is already described in 1.3. However the application had several more protections that we had to bypass in order to capture the communication.

Certificate pinning

The application uses the certificate pinning, that is, it compares the certificate of the server with the received value of the public key. We know that our proxy server has the different value of the public key, therefore the aplication refuses to connect. We bypassed this protection in the same way as in the section 3.2.2 where we analysed the application by Airbank.

Root detection

We have noticed that the application checks whether the mobile device is rooted. When is detects root, the application does not offer you the login screen, it writes that the root access was detected on the device, therfore it cannot start. When searching in the decompiled code we have discovered two libraries that could check root. One of them is RootBeer [65] and the second is JailMonkey [66]. Both these libraries have to be bypassed separately. Both are well known and it is not difficult to find a Frida script that bypasses them. By simple searching we have found two such scripts on Frida CodeShare [19]. They can be found in attachments A.2 as `jailmonkey_bypass.js` and `rootbeer_byapss.js`.

Others

Few seconds after starting the frida, the application rebooted the mobile device. That made the analysis more difficult. However, we have noticed that there are java packages called `com.threatmark.mobile.android` and `cz.pbktechnology.partners.client.threatmark` which we managed to trace before rebooting. By using frida-trace we found the exact place in the code which caused the problem:

```java
public void startThreatMark() {
    try {
        ThreatMark.start(new StartParameters(this.reactContext,
            BuildConfig.THREATMARK_ANDROID_URL,
            2048, new HttpBasicAuthCredentials(),
            "tmSdkKeyAndroid.pem",
            null, false, Collections.emptyList(),
            getValidSslTrustManager(), new CookiesListener() {
    @Override
    public void onCookiesResult(Map<String, String> map) {
    for (String str : map.keySet()) {
    PbkThreatMarkModule.this.cookiesMap.putString(str,
                                                    map.get(str));
    }
    }
    }));
    } catch (ThreatMarkAlreadyActiveException e) {
        Log.d(this.TAG, "Starting SDK when already active.", e);
    } catch (ThreatMarkException e2) {
        Log.d(this.TAG, "Unknown ThreatMark exception.", e2);
        Sentry.captureException(e2);
    } catch (Exception e3) {
        Log.d(this.TAG, "General exception", e3);
        Sentry.captureException(e3);
    }
}
```

By tracing we discovered that the problem is calling the method `ThreatMark.start`. Therefore we wrote our own frida script that overwrites the method `startThreatMark()`. Instead of calling the the method `ThreatMark.start` the method does not do anything.

```javascript
Java.perform(function() {
var PbkThreatMarkModule = Java.use(
"cz.pbktechnology.partners.client.threatmark.PbkThreatMarkModule"
                                                                );
PbkThreatMarkModule.startThreatMark.implementation =
                                        function () {
console.log(`PbkThreatMarkModule.startThreatMark is called`);
                                                    };
});
```

The script can be found in attachments as `script_modification.js` A.2.

When starting the application we also use a script for bypassing multiple protections that can be found in attachments as `muliple_bypass.js` A.2. This script was taken from Frida CodeShare [19].

To summarize it, we start the application by:

```
frida -U -f cz.pbktechnology.partners.client
    -l rootbeer_bypass.js
```

```
    -l jailmonkey_bypass.js
    -l script_modification.js
    -l muliple_bypass.js
    -l android-pinning-bypass_2.js
```

4.2.3 Reversing

The native library libPowerAuth2Module could be decompiled using Ghidra. In contrast to the Airbank Security Module, this library was more difficult to reverse. Only several functions were exported which meant that the rest of them had to be searched and hooked by their address. Also there were many changes made by the optimizer. On the other hand, the library has a documentation on Github which helped us to reconstruct the code.

We used frida tool Interceptor in the following way to hook the functions given by their address:

```
const ghidraImageBase = 0x00100000;
const moduleName = 'libPowerAuth2Module.so';
const moduleBaseAddress = Module.findBaseAddress(moduleName);
const fnAddress = moduleBaseAddress.add(
                        0x0027e9d8 - ghidraImageBase);

Interceptor.attach(fnAddress,
    {onEnter:
        function(args){
            ...
```

The code was taken from stackoverflow.com [67].

pa_signature

By hooking all functions called from PowerAuth module we noticed that there is the class `Session` with method `signHTTPRequest` that is called during the login. We inspected the captured communication and we found the signed request, we could see the input to the signature when hooking. We found the function in jadx:

```
public native SignatureResult signHTTPRequest(
                    SignatureRequest signatureRequest,
                    SignatureUnlockKeys signatureUnlockKeys,
                    int i);
```

The method is called from PowerAuth module therefore we used Ghidra as it is mentioned above. The function was complicated, however we managed to localize parts of the code where the key derivation and the signature are performed.

pa_signature - key derivation

We knew the key derivation should be done as it is written in the documentation. According to the documentation, both signature keys, possesion and knowledge are derived from the master key that was derived during the Activation (account linking).

They also say that the possesion signature key should be encrypted using a client device fingerprint. The documentation does not describe the derivation of the encryption key. The derivation of the key is not part of the module.

The knowledge key is encrypted using the PIN. The key derivation and storage of the knowledge key is described in the documentation. The code from the documentation is following:

```
char[] password = "1234".toCharArray();
byte[] salt = Generator.randomBytes(16);
int iterations = 10000;
int lengthInBits = 128;
SecretKey encryptionKey = PBKDF2.expand(password,
                          salt, iterations, lengthInBits);
byte[] iv = Generator.zeroBytes(16);
byte[] keyKnowledgeBytes = KeyConversion.getBytes(
                          KEY_SIGNATURE_KNOWLEDGE);
byte[] C_KEY_SIGNATURE_KNOWLEDGE = AES.encrypt(keyKnowledgeBytes,
                          iv, encryptionKey, "AES/CBC/NoPadding");

// Store `C_KEY_SIGNATURE_KNOWLEDGE` and `salt`.
```

We know which functions should be visible in the pseudocode produced by Ghidra when decompiling the library. Therefore, we can look for AES_CBC_decrypt and PBKDF2 which internally uses HMAC with SHA1 (which can also be found in documnetation). The HMAC can be found by constants as we will see in the next subsection about revesing the signature. To find PBKDF2 we used chatGPT. We found a function that looked differently and seemed to be important. We asked the chatbot what does this decompiled code do and we obtained the answer. Very similarly we recognized AES decrypt which was useful. We checked both these answers by hooking these functions using their address. Both were correct.

By hooking these functions we discovered that the knowledge signature key is obtained exactly as it is described in the documentation. We obtained salt, key, etc. and we implemented our code and then compared results. We discovered that the possesion signature key is stored encrypted by AES_CBC without padding with zero IV, using the value (128 bits) related to device as an encryption key. This is not explicitely written in the documentation.

pa_signature - signature

The signature should be done as follows, this code is taken from the documentation of the library [63].

```java
/**
* Compute the signature components for given data using provided
* keys and current counter.
* @param data - data to be signed
* @param signatureKey - array of symmetric keys
* used for signature
* @param CTR_DATA - hash based counter
*/
List<byte[]> computeSignatureComponents(byte[] data,
                List<SecretKey> signatureKeys, byte[] CTR_DATA) {
    // ... compute signature components
    List<byte[]> signatureComponents = new ArrayList<byte[]>();
    for (int i = 0; i < signatureKeys.size(); i++) {
        byte[] KEY_SIGNATURE = KeyConversion.secretKeyFromBytes(
                                        signatureKey.get(0));
        byte[] KEY_DERIVED = Mac.hmacSha256(KEY_SIGNATURE,
                                            CTR_DATA);

        // ... compute signature key using more than one keys,
        //     at most 2 extra keys
        // ... this skips the key with index 0 when i == 0
        for (int j = 0; j < i; j++) {
            KEY_SIGNATURE = KeyConversion.secretKeyFromBytes(
                                    signatureKey.get(j + 1));
            KEY_DERIVED_CURRENT = Mac.hmacSha256(KEY_SIGNATURE,
                                                CTR_DATA);
            KEY_DERIVED = Mac.hmacSha256(KEY_DERIVED_CURRENT,
                                                KEY_DERIVED);
        }
        // ... sign the data
        byte[] SIGNATURE_COMPONENT = Mac.hmacSha256(KEY_DERIVED
                                                    , DATA);
        // ... keep it in the list
        signatureComponents.add(SIGNATURE_COMPONENT);
    }
     return signatureComponents;
}
```

More details about input parameters and functions can be found in the previous chapter 4.1.2.

```java
/**
 * Compute the signature for HTTP request purposes for given data
 * using provided keys and current counter.
 * @param data - data to be signed
 * @param signatureKey - array of symmetric keys u
 *                      used for signature
 * @param CTR_DATA - hash based counter
 */
String computeOnlineSignature(byte[] data,
                List<SecretKey> signatureKeys, byte[] CTR_DATA) {
// ... at first, calculate signature components
List<byte[]> signatureBinaryComponents =
    computeSignatureComponents(data, signatureKeys, CTR_DATA);

// ... now convert components into one Base64 string
byte[] signatureBytes = new byte[signatureKeys.size() * 16];
for (int i = 0; i < signatureComponents.size(); i++) {
  byte[] SIGNATURE_COMPONENT = signatureBinaryComponents.get(i);
  // ... append last 16 bytes from SIGNATURE_COMPONENT
  // to signature bytes
  ByteUtils.copy(SIGNATURE_COMPONENT, 16,
        signatureBytes, i * 16, 16);
}
// ... final conversion to Base64
return Base64.encode(signatureBytes);
}
```

From the documentation we can see that HMAC is used several times when singning. Therefore, we try to find HMAC in the pseudocode generated by Ghidra. We discovered the part of the code with ipad and opad constants and then we discovered different part of the code that performed SHA256. SHA256 has typical constants that were visible in the code.

The part of the code where we can see ipad and opad constants:

```
97    else {
98      if ((int)param_3 < 0) {
99        return false;
100     }
101     memcpy(&local_170,param_2,(ulong)param_3);
102     local_34 = param_3;
103   }
104   if (local_34 != 0x90) {
105     memset((void *)((long)&local_170 + (ulong)local_34),0,(ulong)(0x90 - local_34));
106   }
107   local_e0 = local_170 ^ 0x3636363636363636;
108   uStack_d8 = uStack_168 ^ 0x3636363636363636;
109   uStack_d0 = uStack_160 ^ 0x3636363636363636;
110   uStack_c8 = uStack_158 ^ 0x3636363636363636;
111   local_c0 = local_150 ^ 0x3636363636363636;
112   uStack_b8 = uStack_148 ^ 0x3636363636363636;
113   uStack_b0 = uStack_140 ^ 0x3636363636363636;
114   uStack_a8 = uStack_138 ^ 0x3636363636363636;
115   local_a0 = local_130 ^ 0x3636363636363636;
116   uStack_98 = uStack_128 ^ 0x3636363636363636;
117   uStack_90 = uStack_120 ^ 0x3636363636363636;
118   uStack_88 = uStack_118 ^ 0x3636363636363636;
119   local_80 = local_110 ^ 0x3636363636363636;
120   uStack_78 = uStack_108 ^ 0x3636363636363636;
121   local_60 = local_f0 ^ 0x3636363636363636;
122   uStack_58 = uStack_e8 ^ 0x3636363636363636;
123   uStack_68 = uStack_f8 ^ 0x3636363636363636;
124   uStack_70 = uStack_100 ^ 0x3636363636363636;
125   iVar3 = FUN_002146b0(param_1[2],lVar4,param_5);
126   if (iVar3 != 0) {
127     lVar5 = param_1[2];
128     iVar3 = FUN_00215644(lVar4);
129     iVar3 = FUN_0021481c(lVar5,&local_e0,(long)iVar3);
130     if (iVar3 != 0) {
131       local_e0 = local_170 ^ 0x5c5c5c5c5c5c5c5c;
132       uStack_d8 = uStack_168 ^ 0x5c5c5c5c5c5c5c5c;
133       uStack_d0 = uStack_160 ^ 0x5c5c5c5c5c5c5c5c;
134       uStack_c8 = uStack_158 ^ 0x5c5c5c5c5c5c5c5c;
135       local_c0 = local_150 ^ 0x5c5c5c5c5c5c5c5c;
136       uStack_b8 = uStack_148 ^ 0x5c5c5c5c5c5c5c5c;
137       uStack_b0 = uStack_140 ^ 0x5c5c5c5c5c5c5c5c;
138       uStack_a8 = uStack_138 ^ 0x5c5c5c5c5c5c5c5c;
139       local_a0 = local_130 ^ 0x5c5c5c5c5c5c5c5c;
140       uStack_98 = uStack_128 ^ 0x5c5c5c5c5c5c5c5c;
141       uStack_90 = uStack_120 ^ 0x5c5c5c5c5c5c5c5c;
```

The part of the code with SHA256:

```
152      param_2 = param_2 + 4;
153      param_3 = param_3 + -1;
154      auVar45 = NEON_rev32(auVar45,1);
155      auVar72 = NEON_rev32(*pauVar15,1);
156      auVar38 = NEON_rev32(*pauVar18,1);
157      auVar40 = NEON_rev32(*pauVar1,1);
158      auVar86._0_4_ = auVar45._0_4_ + 0x428a2f98;
159      auVar86._4_4_ = auVar45._4_4_ + 0x71374491;
160      auVar86._8_4_ = auVar45._8_4_ + -0x4a3f0431;
161      auVar86._12_4_ = auVar45._12_4_ + -0x164a245b;
162      auVar82 = NEON_sha256su0(auVar45,auVar72,4);
163      auVar45 = NEON_sha256h(auVar34,auVar36,auVar86,4);
164      auVar86 = NEON_sha256h2(auVar36,auVar34,auVar86,4);
165      auVar88 = NEON_sha256su1(auVar82,auVar38,auVar40,4);
166      auVar110._0_4_ = auVar72._0_4_ + 0x3956c25b;
167      auVar110._4_4_ = auVar72._4_4_ + 0x59f111f1;
168      auVar110._8_4_ = auVar72._8_4_ + -0x6dc07d5c;
169      auVar110._12_4_ = auVar72._12_4_ + -0x54e3a12b;
170      auVar72 = NEON_sha256su0(auVar72,auVar38,4);
171      auVar82 = NEON_sha256h(auVar45,auVar86,auVar110,4);
172      auVar86 = NEON_sha256h2(auVar86,auVar45,auVar110,4);
173      auVar72 = NEON_sha256su1(auVar72,auVar40,auVar88,4);
174      auVar96._0_4_ = auVar38._0_4_ + -0x27f85568;
175      auVar96._4_4_ = auVar38._4_4_ + 0x12835b01;
176      auVar96._8_4_ = auVar38._8_4_ + 0x243185be;
177      auVar96._12_4_ = auVar38._12_4_ + 0x550c7dc3;
178      auVar38 = NEON_sha256su0(auVar38,auVar40,4);
179      auVar45 = NEON_sha256h(auVar82,auVar86,auVar96,4);
180      auVar86 = NEON_sha256h2(auVar86,auVar82,auVar96,4);
181      auVar38 = NEON_sha256su1(auVar38,auVar88,auVar72,4);
182      auVar111._0_4_ = auVar40._0_4_ + 0x72be5d74;
183      auVar111._4_4_ = auVar40._4_4_ + -0x7f214e02;
184      auVar111._8_4_ = auVar40._8_4_ + -0x6423f959;
185      auVar111._12_4_ = auVar40._12_4_ + -0x3e640e8c;
186      auVar40 = NEON_sha256su0(auVar40,auVar88,4);
187      auVar82 = NEON_sha256h(auVar45,auVar86,auVar111,4);
188      auVar86 = NEON_sha256h2(auVar86,auVar45,auVar111,4);
189      auVar40 = NEON_sha256su1(auVar40,auVar72,auVar38,4);
190      auVar97._0_4_ = auVar88._0_4_ + -0x1b64963f;
191      auVar97._4_4_ = auVar88._4_4_ + -0x1041b87a;
192      auVar97._8_4_ = auVar88._8_4_ + 0xfc19dc6;
193      auVar97._12_4_ = auVar88._12_4_ + 0x240ca1cc;
194      auVar88 = NEON_sha256su0(auVar88,auVar72,4);
195      auVar45 = NEON_sha256h(auVar82,auVar86,auVar97,4);
196      auVar86 = NEON_sha256h2(auVar86,auVar82,auVar97,4);
```

We tried to hook all functions connected to the start of HMAC and SHA256. Usually input parameters are pointers or the function could not be easily hooked as in the case of SHA256. However, we managed to hook the function where SHA256 is called. We used a script that is in attachments as `Partners_hook_native_partial_input_to_SHA256.js` A.3. Then we reconstructed the algorithm using captured values from which we could obtain hmac keys, input data and counter. We also used the documentation.

Activation Status

Activation Status decoding is implemented in libPowerAuth2Module in the class Session in the method decodeActivationStatus. We analysed this method very similarly as pa_signature and key derivations. We had the documentation, therefore we were looking for HMAC and AES functions in decodeActivationStatus. We found them (and functions above them) very similarly as before. We hooked them and as before we wrote our own implementation and we used captured values to be sure that the code is written correctly.

4.3 Conclusion

In this section we conclude the results.

As in the previous chapter, we did not examined the application as much as we wanted, therefore some parts of work are only assumptions. On the other hand, we analysed most of the functions correctly, since we managed to performed the same computations as the application.

Similarly to the previous chapter, cryptography is implemented in the native library which we had to reverse. In this chapter we conclude main results in terms of security.

As in the previous chapter, imagine, theoretically, that we do not trust TLS.

Then the protocol does not in general provide integrity. The only part that is signed and thus it provides the integrity and authentication are login/transaction requests from the client with string "VALIDATE_PASSWORD" or with empty body. Also, theoretically, the encrypted Activation Status and sent challenge/nonce cannot be changed, because the client could not decrypt it. Practically, it can be changed as we have already said. The integrity of everything else but several signed requests is not ensured.

It partially provides forward secrecy. Since the counter is transferred encrypted, based on hash function, we cannot simply compute its previous value and thus we cannot compute any previous signature even if we know both keys. On the other hand, the encryption of the Activation Status uses always the same key, therefore in this part the protocol does not provide the forward secrecy.

Let us consider the following attackers:

1. The attacker that can capture and modify the communication.
2. The attacker that can obtain data from mobile device.
3. The attacker that can do both - capture the comunication and obtain data from the mobile device.

The protocol is vulnerable as follows:

1. The attacker in this position can perform the man in the middle attack, since the scheme does not autheticate the server and does not provide integrity. In the case of login it means that he can pretend to be the server and capture signed requests, which he can later use for his own login. In the case of the transaction, the attacker can entirely modify the transaction, for example amount of money, number of receiver's account, etc. We checked that the attack on transaction is possible.
2. The attacker in this position can probably obtain KEY_TRANSPORT and KEY_SIGNATURE_POSSESION since they are encrypted by some value related to the device. He can theoretically obtain the value of the counter.
3. The attacker in this position can do everything that previous two attackers. He could theoretically try to bruteforce PIN, resp. try to obtain KEY_SIGNATURE_KNOWLEDGE. He would need to have KEY_SIGNATURE_POSSESION, input salt to PBKDF2, the correct value of counter in the time of the captured communication and the captured

communication for checking the result. This is not very probable, moreover the value of the counter cannot be not easily computed backwards.

We note that the PowerAuth uses its own cryptographic schemes for the signature and the encryption. They are not part of any standard. Although we cannot see any problem, usually is better to follow some standard. At least some of functions look more complicated than needed.

The application does not use keystore for storing cryptographic keys. If it used keystore it would be more difficult to obtain keys when reverse engineering.

5 KB+

In this chapter we analyse the cryptography used by KB+, version 2.0.2. As in the previous two cases, we had to open an account. Due to the lack of time we did not check our results by writing our own script. Some parts of the analysis may be probably inaccurate and incomplete. That is also the reason why we did not contact the bank.

This chapter also consists of three sections - theoretical part, where are the protocols described, practical part and conclusion. In examples of the communication we rewrote all sensitive information by stars `"****"`.

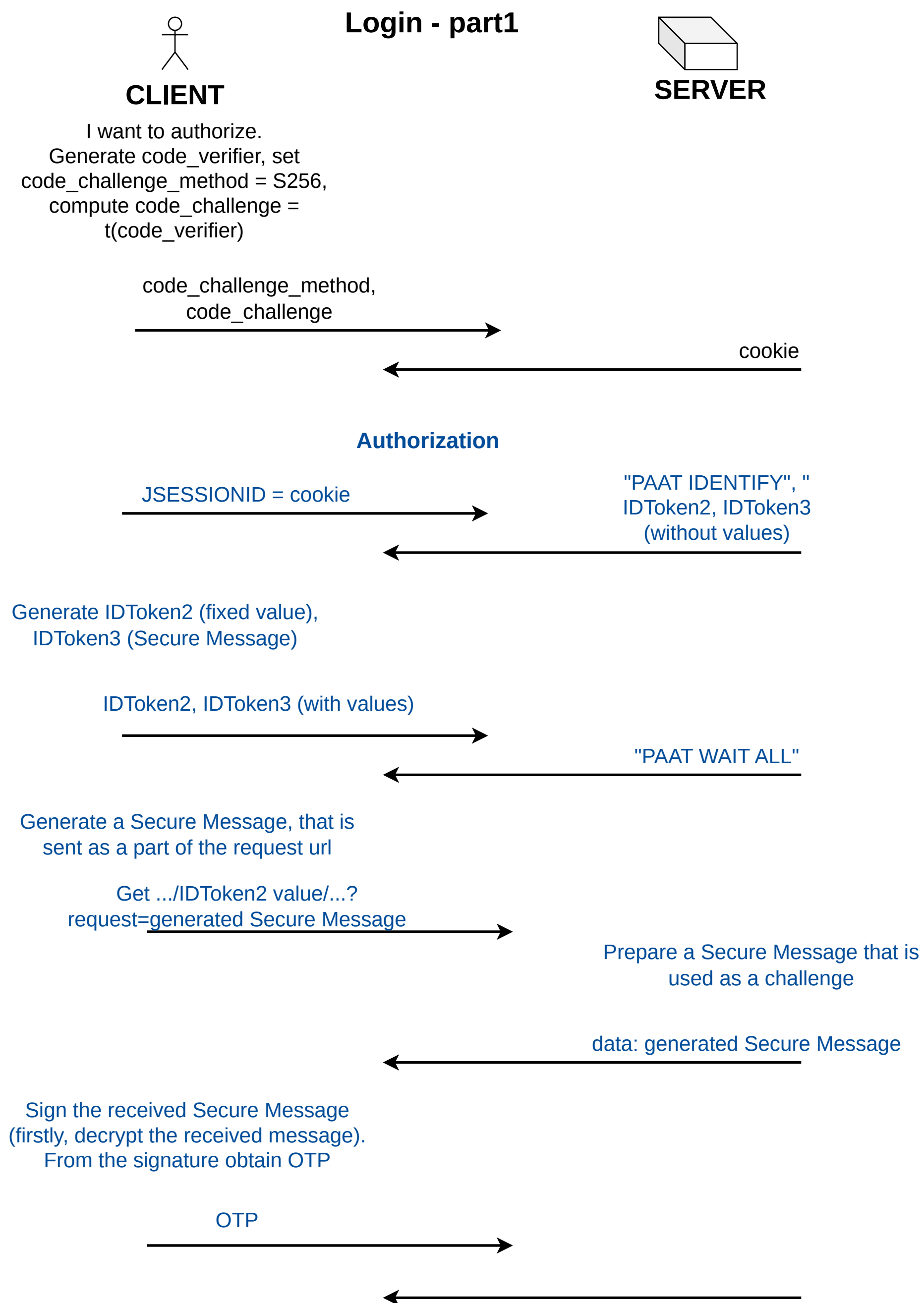
5.1 Protocol description

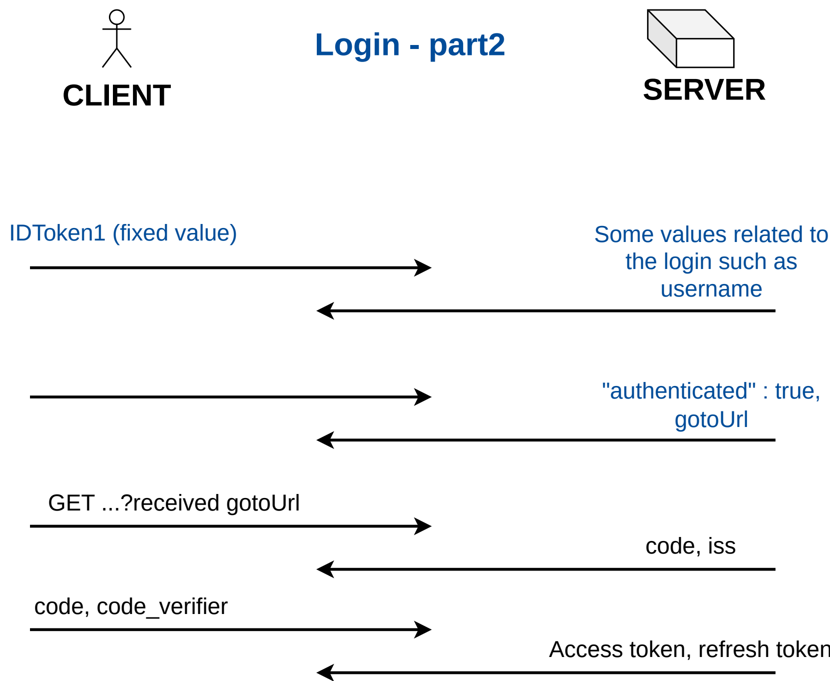
In this section we describe Login and Transaction operations.

5.1.1 Login

The login operation uses OAuth 2.0 PKCE for authentization/authorization. During this protocol, the client has to authenticate and also authorize the login operation to the authorization server. For the authorization the protocol uses OTP generated by Digipass SDK. In following section we provide an overview of the login protocol.

Overview





OAuth

The application uses OAuth 2.0 PKCE which we have briefly described in 2.12. For the authentication/authorization between steps (A) and (B) in the protocol the application uses cryptography provided by Digipass SDK which we describe in the next section.

Digipass SDK

To encrypt and sign messages, to authentize or to generate OTP, the client uses Digipass SDK by OneSpan. All important methods can be found in Java package `cz.kb.caas.mobile.security.token.device` in the class `DigipassCryptoEngineImpl`. The documentation by OneSpan is available here [68].

Static and dynamic vectors

Digipass uses static and dynamic vectors to store values related to the cryptography.

Static vector does not change. According to the documentation [68] it contains flags that specify the configuration, more exactly parameter set of the cryptographic applications and for example the customer master key. We could see that the static vector is sent during the registration in plaintext, therefore we can conclude that in general the static vector does not contain any sensitive values.

On the other hand, according to the documentation [68], the dynamic vector should change with every OTP or signature generation. In contains sensitive values - Digipass secrets. We will see that they are usually stored encrypted.

Due to the obfuscation and the lack of time we did not manage to obtain the exact structure or values from vectors.

Both vectors are used to create one class.

createSecureChannelInformationMessage

When sending a message using a Digipass package, there is created a special class for the message. This class is called `SecureMessage` and it is created by calling method `createSecureChannelInformationMessage` with parameter containing the message. Firstly, we will describe the supposed structure of the `SecureMessage` as we can see it in the method.

There are 9 fields in te class.

- **f1**: byte 0
- **f2**: byte 36
- **f3**: IDToken2 - Hexadecimal string obtained from static/dynamic vector. It is fixed, because it does not change even when registering a new device.
- **f4**: 8 random bytes
- **f5**: The message transformed to hexadecimal string. The message can be encrypted.
- **f6**: Boolean that says whether the message is encrypted. From the code it seems that the encryption is also dependent on the input to the function that performs or does not perform the encryption.
- **f7**: byte 1
- **f8**: Signature computed from all previous fields, using the secret key.
- **f9**: All previous fields transformed to the string.

Although the encryption is, theoretically, optional we will see that messages created by `createSecureChannelInformationMessage` are usually encrypted. The encryption performed in the method seems to be done as follows.
Firstly the key has to be derived. The key derivation uses the hash function SHA256, PBKDF2 and AES. As an input it uses static/dynamic vector, IDToken2 - which is in field **f3**, random bytes that are in **f4** and DeviceFingerprint.

From the code it seems that firstly the value of IDToken2 is hashed using SHA256. The output of SHA256 is then used as an input to the function that performs PBKDF2 together with some fixed bytes, that are visible in the decompiled code:

```
/* loaded from: classes.dex */
public final class n {

    /* renamed from: a */
    public static final byte[] f13826a = {-80, -123, 42, -101, -7, -42, 35, 66, 12, 24, 78, -103, 87, 77, -82, -95};

    /* renamed from: b */
    public static final byte[] f13827b = {71, 109, 122, 15, Base64.padSymbol, -2, -54, -71, -112, -96, -31, 73, 95, ByteCompanionObject.MAX_VALUE, 109, -36};

    /* renamed from: c */
    public static final byte[] f13828c = {10, -73, 56, 92, -81, 66, -84, 34, -62, 92, 107, -4, 116, 80, -8, 64};

    /* renamed from: d */
    public static final byte[] f13829d = {-124, -16, 27, 4, -32, 115, 113, -58, -117, 55, 76, 104, 32, 81, -58, -108};
```

The function performing PBKDF2 could be roughly described as follows:

Algorithm 35 Partial KDF

INPUT: fb - fixed bytes transformed to string, output_from_SHA256, DeviceFingerprint, class StatAndDynVect connected to staticand dynamic vectors
OUTPUT: partially_derived_key

1: Combine fb, DeviceFingerprint and some bytes from static/dynamic vectors (using StatAndDynVect) into one bytearray bArr
2: derived_salt = SHA256(bArr)
3: partially_derived_key = PBKDF2(key = ouput_from_SHA256, number_of_iterations = 1000, salt = derived_salt, derived_value_output_length = 16)
4: **return** partially_derived_key

PBKDF2 seems to be using SHA256 as an underlying hash function.

From the code it seems that after this algorithm there is also some other encryption function, however we could not trace it by frida-trace, therefore we do not know whether it is called. If yes, then it uses the partially derived key to decrypt some secret value from the dynamic vector. We just note that the function is roughly the following:

Algorithm 36 Partial KDF - AES decryption

INPUT: partially_derived_key - output from previous algorithm
OUTPUT: partially_derived_or_decrypted_key2

1: input = value from the dynamic vector, probably encrypted key
2: IV = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
3: derived_key = AES128-CBC_dec(partially_derived_key, input, IV)
4: **return** partially_derived_or_decrypted_key2

After this, the key derivation uses AES as follows.

Algorithm 37 Partial KDF - AES

INPUT: partially_derived_or_decrypted_key2 - output from previous algorithm
OUTPUT: derived key

1: input = [-78,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15]
2: derived_key = AES128-ECB_enc(partially_derived_or_decrypted_key2, input)
3: **return** derived_key

Sencondly, the key is used to encrypt the message.

The message is encrypted using AES as follows:

Algorithm 38 SecureMessage encryption

INPUT: derived_key, message (**f5**), random bytes **f4**
OUTPUT: encrypted_message

1: IV=**f4** padded with 0 to obtain legth 16 bytes
2: enc_message = AES128-CTR_enc(derived_key, message **f5**, nonce = **f4**)
3: **return** enc_message

We note that this can be computed without the knowledge of the PIN and thus the encryption can be performed before the user logs in. The encrypted message then replaces the plaintext message in the field **f5**.
Now we will describe the signature.

Firstly, the key has to be derived/decrypted. This is done exactly same as above when deriving key for encryption. The only difference is the last step. When encrypting key by AES128-ECB, the input is

$$[-77, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15].$$

Only the first byte is different.

Secondly, the signature is performed by HMAC-SHA256.

Algorithm 39 SecureMessage signature

INPUT: derived_key, SecureMessage class
OUTPUT: signature

1: SecureMessage_str = map SecureMessage to string
2: signature = HMAC-SHA256(derived_key, SecureMessage_str)
3: **return** signature

Therefore the `SecureMessage` in the method `createSecureChannelInformationMessage` is created in the following way.

Fields **f1,f2** and **f7** will be assigned specific values given above. IDToken2 is extracted from static/dynamic vector and assigned to **f3**. Also 8 random bytes are generated and assigned to **f4**. The input message is transformed to bytes and to hexstring. Then it is asssigned to **f5**. The field **f6** is set to false. The message is then encrypted as described above and assigned to **f5**. The field **f6** is set to true. Then the class is signed by HMAC as described above. The signature is assigned to **f8**. Then everything is transformed into string and assigned to **f9**.

decryptSecureMessageBody

When receiving an encrypted message from the server that has to be decrypted, the method `decryptSecureMessageBody` is called. Firstly, the string describing the class `SecureMessage` is parsed and the class `SecureMessage` is created. Then the client checks that the signature in **f8** is correct. When it is equal to the HMAC computed by the client, the message can be decrypted. The encrypted message is in **f5**. The decryption is analogous to the encryption. The key is derived in the same manner and the only difference is that the AES256-CTR is now in the

decryption mode. Random bytes from the received message are used as a nonce. After the decryption, the obtained message is assigned to **f5** and **f6** is set to false.

generateSignature

The client authentizes by computing and sending OTP (one time password). This is done by the method `generateSignature` and not by the method `generateOtp`, which we would expect.

Since the code performing the signature is heavily obfuscated and due to the lack of time we did not manage to reverse the code completely. However, from the decompiled code we got at least some idea which we will describe in this section. Please note that the description may be inaccurate. We will start by high-level overview.

The method `generateSignature` receives a challenge from the server that has to be signed. The challenge is sent in the form of `SecureMessage`. Firstly, the message is parsed into the class `SecureMessage` and the message in **f5** is decrypted. Then the signatue can be performed. In order to do that, the user has to enter his PIN which is then used to decrypt the secret that is used for the signature. The input to the signature algorithm is not only the message, but whole string describing the class. This string is hashed by SHA256 and the result is divided into four parts that are then processed by the signature algorithm.

The signature algorithm needs a key for the signature, let us denote it key_sign, and the input data. From the decompiled code it seems that the signature key is encrypted by the key derived from user's PIN. Let us denote this key key_user. It seems that the key_sign is encrypted and stored as a part of the dynamic vector. This also corresponds to the documentation. The key_user is derived using DeviceFingerprint, some values from static/dynamic vector, one of them will probably be IDToken2, and the PIN value. The derivation seems to be using SHA256 several times and then PBKDF2 with 1000 iterations, where the key is PIN and salt is a hash of array that combines the rest of values. Moreover, from the code it seems that one of the values in salt derivation is the output from WhiteBoxAES decryption with CTR mode. Input values to this AES are fixed in the code. It also seems that AES uses some tables which are part of the application. After obtaining key_user we can decrypt the value of key_sign. It is encrypted by AES128-CBC with zero IV.

In general we could see that the signature process is given by flags, which are probably part of the static/dynamic vector. We managed to obtain most of them. Despite that, the signature function was obfuscated and difficult to read, therefore, we only describe what we have seen in the decompiled code. Firstly the string "CSMS " is encrypted using some key from the dynamic vector. The signature uses 4 bytes from the DeviceFingerprint processed by SHA256 and some transformation. The current time is obtained and assigned to some variable or field two times in the code, however it seems that the part of the code working with time steps is skipped. Also the part where we can see (after a little bit of reversing) the string "OCRA-3:HOTP-SHA1-" and "C-Q" or "Q" seems to be skipped. In the code we can see that there is one interesting method called three times. This method performs the encryption of the input by AES CBC and then XORs two halves of the output. In the code we can also see the while loop processing four string from the hash of the signed message.

Login protocol

Algorithm 40 Login protocol

CLIENT: Generate code_verifier.
Compute code_challenge=S256(code_verifier).
Send code_challenge and code_challenge_method to SERVER.
CLIENT: Generate IDToken2. IDToken2 is a fixed hexstring that is computed from dynamic/static vectors. Generate IDToken3. IDToken3 is a SecureMessage that is constructed using method createSecureChannelInformationMessage. The input message to this method can be for example following:


```
{"paat":{"appVersion":"2.0.2", "voiceOver":false,
"dictionaryVersion":203,"sdkVersion":"2.11.4",
"language":"en","apiLevel":"31","versionCode":"2.0.2"},
"instance":"FEF1899184-7","operation":"identifyInstance",
"deviceId":*******,"timestamp":"20250716T093839.962Z"}
```


The structure is always the same, values (for example timestamp, instance) can change. This message is encrypted and the class SecureMessage is hashed, both operations using static/dynamic vectors, DeviceFingerprint and randomly generated bytes as it is described in 5.1.1.
SERVER: Check the hash and decrypt the message. When everything is correct the protocol can proceed.
CLIENT: Generate a SecureMessage similarly as before. The only difference is that it constains the following message (where values change).


```
{"paat":{"appVersion":"2.0.2","voiceOver":false,
"dictionaryVersion":203,"sdkVersion":"2.11.4",
"language":"en","apiLevel":"31","versionCode":"2.0.2"},
"instance":"FEF1899184-7","operation":"getPendingOperations",
"deviceId":*******,"timestamp":"20250716T093840.500Z"}
```


Send a request to SERVER where part of the url is the session instance and the generated SecureMessage.
SERVER: Check the hash and decrypt the message. Generate a challenge and use it to generate an encrypted SecureMessage. The message inside the SecureMessage contains some bytes that change, some fixed bytes and it always contains following strings.

```
directbanking.common, NDB_MOBILE, 81
```

Send generated SecureMessage to CLIENT with label 'data'.
CLIENT: Check the hash and decrypt the message using decryptSecureMessageBody. The whole message may looks as follows:


```
02010A066148647A4877000B066148647A745100820A1464697265637462616E
6B696E672E636F6D6D6F6E00830A0A4E44425F4D4F42494C4500010238310000
```


Sign the received SecureMessage class (string in **f9**) by calling method generateSignature. When calling this method, the user has to enter PIN. The result of this method is OTP consisting of 8 digits. Send OTP to server.

SERVER: If OTP is correct, then the CLIENT is authenticated. Send gotoUrl to CLIENT.
CLIENT: Send request using url received from the server.
SERVER: Send code and iss.
CLIENT: Send Access Token Request with code and code_verifier.
SERVER: Check the code_verifier by computing S256(code_verifier) and comparing the value with code_challenge. If the code_verifier is correct, send access and refresh tokens to CLIENT.

Example of communication

1.
```
POST /api/security/v1/auth/authorize?response_type=
     code&redirect_uri=http%3A%2F%2Flogin.kb.cz%2Fc
     allback&client_id=NDB_MOBILE&scope=directbanki
     ng.common&nonce=0Xp5NdTJNT3ZE7PWyIeozw&code_ch
     allenge=0DLHWCqG8wqee71QiEvzMiVKS-RwdrPbNqzWD2
     N3waI&code_challenge_method=S256 HTTP/2
Host: gateway.kb.cz
...

{}
```

2.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:26 GMT
X-Content-Type-Options: nosniff
...
```

3.
```
POST /api/security/v1/auth/resetAuthChain?sso=NDB&lang=cs
     &variant=app&method=PAAT&methodState=ACTIVE HTTP/2
Host: gateway.kb.cz
...

{"frontendData":{"secDashboard":{"version":2,"score":44.0}}}
```

4.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:26 GMT
...

{"authenticated":false,
"authData":{"template":"PAAT_IDENTIFY",
"callbacks":
[{"type":"TextOutputCallback","input":[],
         "output":[{"name":"message","value":
         "/*{ \"operationId\":\"cfbf0a11c82a4e2eba8c4e0e27fa
          ba8b\" }*/"}, {"name":"messageType","value":"4"}]},
{"type":"NameCallback","input":[{"name":"IDToken2","value":""}],
        "output":[{"name":"prompt","value":"ID:"}]},
```

```
{"type":"NameCallback","input":[{"name":"IDToken3","value":""}],
        "output":[{"name":"prompt","value":"Message:"}]}]},
"error":"OK",
...
"currentChain":"paat~app",
"operationInfo":{"type":"LOGIN",
    "state":"AUTHORIZATION_PENDING",
    "definitionDescription":"Přihlášení",
    "securityLevel":1,
    "timestamp":"20250715201126",
    ...
    "oauth2ClientName":"KB+",
    ...
}}
```

5.
```
POST /api/security/v1/auth/stepAuthChain HTTP/2
Host: gateway.kb.cz
...

{"template":"PAAT_IDENTIFY",
"callbacks":
[{"type":"TextOutputCallback"},
{"type":"NameCallback",
    "input":[{"name":"IDToken2","value":"FEF1899184"}]},
{"type":"NameCallback",
    "input":[{"name":"IDToken3","value":"09015FFF1CFAB06
              2BB5187B72A5B62DF182CBEE78AC8BBC9D70A66E39
              B65015F37DEC954E5B45E59F048AC48F0D46CA3D2B
              3EC6F885D4C62EC9A07011C6EB23E2BD96B326BD5E
              86A1284C2C3B382EAFE9DF123D59FB047250E1EE58
              96919E1B6B2BD74632E100F5D5400F514034B39549
              2AF8F6B97A7C0699D614FE2B0C338B62C5C5F49E85
              2FAAEACD0F07DE5EA6477BB62A7C00578309813211
              5519C9FA6CA3B9779B31D695D6653E70A8DB7EECAA
              3103C52BDBAE43F4D685DF7D6F142EC9D9B05E4987
              F0FF24E6BA1429432475957088O1D15A12AC38AC88
              2D530D59820C17A779C28CCD832FF842FD1F636A5D
              13605EFD2D4151A020079CF846585E68A56B880B8D
              40E010AC6E1F4329E446107A5C60F739DF7DEAFC3C
              63B34"}]}]}
```

6.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:27 GMT
...

{"authenticated":false,
"authData":{"template":"PAAT_WAIT_ALL",
"callbacks":
[{"type":"TextOutputCallback","input":[],"output":
         [{"name":"message","value":"/*{ \"loginCode\": \"9TQ\",
         \"expiration\": \"20250715201356\" }*/"},
```

```
         {"name":"messageType","value":"4"}]},
{"type":"PollingWaitCallback","input":[],"output":
        [{"name":"waitTime","value":"500"},
        {"name":"message","value":"Waiting for response..."}]},
{"type":"ConfirmationCallback","input":[{"name":"IDToken3",
        "value":99}],"output":[{"name":"prompt","value":""},
        {"name":"messageType","value":0},
        {"name":"options","value":["Use offline login"]},
        {"name":"optionType","value":-1},
        {"name":"defaultOption","value":0}]}]},
...
"operationInfo":{"type":"LOGIN",
"state":"AUTHORIZATION_PENDING",
...
}}
```

7.

```
GET /api/security/v1/operations/FEF1899184-7/
    ?request=09015FFF1CFAB0BF221A5D676BCB0310
    CB31456D9112E688CDA749D200555535E70916F83
    8A4C42CD27F233BBF4FEEFC2E8BC8417E04A857CD
    C0E28E591B25AB9322B61D98EB2B10024C53BF086
    67165316D822DAA1CEA9755A8B5255745CC3B4A00
    4E02D7B3CA166559D9B2C759618142CCF61FAD71C
    EC28BA984522B1C301B0B96916C03CF6393F1D880
    F4B414B9C2ACBE4F0708C056603B2E657C769E9FA
    99C536750B1E5BB18513B0B8B6CA4E2991534099E
    20D023BA4695621F1558B8425F6408211CFE72ED5
    1B8D11B94EB7726E80A0EAAB52C5058784445A58D
    5398007F0077C70D5EE5F8327BBE55E34C4637684
    44DD67371C3325230629F422BF8BA33F10218EB77
    69AFC908AF735F2EA93315C910E2837698D5B52C0
    3DE84EB0F&_queryId=cfbf0a11c82a4e2eba8c4e
    0e27faba8b&tmData=TS01cf8e91%3D01b8c95dc8
    31fc14a14feeb291f54a820a8cfc3652e1b7d5634
    a2b0750b9d9ce03008472b3e5656a977082dd723f
    961b5369579b129d24db20e2aab0ba1f8aaf84330
    9bf65%7EpK4902tFXuR3Lyo%3DLwdiiZssQHDh0wm
    IhSB133V7zo0YzogG%7EqF2376dMHkT4Nsp%3DVAU
    G9PLxBYOPqRdqnQhZU3xOKd50jw0r HTTP/2
Host: gateway.kb.cz
...
```

8.

```
HTTP/2 200 OK
...

{"remainingPagedResults":-1,
"result":
  [{"data":"00C15FFF1CFAB05E720E803604BB6F13639819AAB879676
            C2B14CE5B912DF373599FB1FEF729CF653BBDB8E8CA44F5
            8650B8841400439ACDC9D8DBB60500CC029BD2BA35622BE
            A129448D68752971270B36638714188FD",
```

```
  "expiresIn":148,
  "expiry":"2025-07-15T20:13:56.000Z",
  ... }],
  ...}
```

9.

```
PATCH /api/security/v1/operations/FEF1899184-7/cfbf0a11c82a4e2e
      ba8c4e0e27faba8b/ HTTP/2
Host: gateway.kb.cz
...

[{"operation":"add",
"field":"/otp",
"value":"17636766",
"metadata":{"rooted":"0",
            "recognition":"password",
            ...]
```

10.

```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:31 GMT
...

{}
```

11.

```
POST /api/security/v1/auth/stepAuthChain HTTP/2
Host: gateway.kb.cz
...

{"template":"PAAT_WAIT_ALL",
"callbacks":
[{"type":"TextOutputCallback"},
{"type":"PollingWaitCallback"},
{"type":"ConfirmationCallback","input":
        [{"name":"IDToken1","value":"99"}]}]}
```

12.

```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:32 GMT
...

{"authenticated":false,
"authData":{"template":"ADAPTIVE_SECURE_TOKEN",
"callbacks":[
{"type":"TextOutputCallback",
   "input":[],"output":[
   {"name":"message",
   "value":"/*{ \"userSchema\": \"KBID\",
   \"userIdent\": \"*********\",
   \"token\": { \"username\": \"******\",
   \"token\": \"U0RUawE7A...Jzn76uEn1AA==\" },
   \"oidHash\": \"e08eaf2f...371df7\" }*/"},
   {"name":"messageType","value":"4"}]}]},
```

```
...
"operationInfo":{
    "type":"LOGIN","state":"AUTHORIZATION_PENDING",
    "definitionDescription":"Přihlášení",
    ...
}}
```

13.
```
POST /api/security/v1/auth/stepAuthChain HTTP/2
...

{"template":"ADAPTIVE_SECURE_TOKEN",
"callbacks":[{"type":"TextOutputCallback"}]}
```

14.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:33 GMT
...

{"authenticated":true,
"gotoUrl":"https://login.kb.cz/autfe/finishAuth?sess=hr9bLEPc
           _hTINRfeafC71szWpKk9eQr3",
"error":"OK"}
```

15.
```
GET /api/security/v1/auth/finishAuth?sess=hr9bLEPc_hTINRfeafC71sz
    WpKk9eQr3 HTTP/2
Host: gateway.kb.cz
...
```

16.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:33 GMT
...

{"code":"9SbnCu7Mfu26vosYBWEueNlTAzU",
"iss":"https%3A%2F%2Fcaas.kb.cz%2Fopenam%2Foauth2",
"client_id":"NDB_MOBILE"}
```

17.
```
POST /api/security/v1/oauth2/access_token HTTP/2
Host: gateway.kb.cz
...

client_id=NDB_MOBILE&redirect_uri=http%3A%2F%2Flog
          in.kb.cz%2Fcallback&grant_type=authoriza
          tion_code&code=9SbnCu7Mfu26vosYBWEueNlTA
          zU&code_verifier=N0AcFcXk-N1VXTv57_tC0A5
          NcQA76a-CeuHxjgQsUTH8Mr1V4NB5cdSWYTolBKn
          q_Is4AmDu2QXbfDeEVyXX_Q
```

18.
```
HTTP/2 200 OK
Date: Tue, 15 Jul 2025 20:11:33 GMT
...
```

```
{"access_token":"eyJ0eXAiOiJKV1QiLCJraWQiOiJmL1h6VzBaZzls
                VHJtMnRjd2tNeFhlQmN5QjQ9IiwiYWxnIjoiUlMyN
                ...
                ajaCRmbTJZmjuIakAXWA5bkslrAFwOnOROBZygGCK",
"refresh_token":"eyJ0eXAiOiJKV1QiLCJraWQiOiJmL1h6VzBaZzlsV
                 ...
                 O8yXWGRAnuZjATSJalZTpD2P1BoVjI3J4XsGvUlwT",
"scope":"directbanking.common",
"token_type":"Bearer",
"expires_in":299,
"nonce":"OXp5NdTJNT3ZE7PWyIeozw"}
```


Conclusion and security

In this section we conclude the results. We have not managed to obtain exact algorithm for dynamic vector updating or generation of the signature (our OTP). Hence we cannot evaluate it from the cryptographic point of view. However we have an idea how the encryption and the signature of the secure message works thus we can describe them and evaluate their cryptography.

The key that is used to decrypt the secret stored in the dynamic array is derived using SHA256 and PBKDF2 with 1000 iterations. The number of the iterations is too low for password storage, however, it seems that the derivation uses known non-protected values that are connected to the device (such as DeviceFingerprint) and the user or values taken from the static/dynamic vector and thus the number of iterations does not matter that much, because if we can obtain values from the device there is nothing to brute-force. It seems that the reason for the encryption of the secret is not to store it in plaintext.

The secret is encrypted by AES128-CBC with fixed and predictable IV. In this case, when encrypting one block of the random plaintext it does not have to be an issue, however we do not know details, whether the key or secret changes etc.

The following AES128-ECB encryption that encrypts fixed bytes using obtained secret is just the next step in the key derivation. Because of this step, different keys are derived for AES encryption and for HMAC. AES encryption itself is not a proper key derivation function and ECB mode is not in general recommended for encryption.

The message is encrypted by AES128-CTR. CTR mode is recommended for the encryption by for example BSI [28], however we have to ensure that the input IV is nonce. This is satisfied, because in this case the nonce is randomly generated block of 8 bytes. The second half is the couner, therefore the nonce is padded with zeros.

The usage of HMAC-SHA256 is in general recommended for the signature. In the RFC 2104 [33] they state that keys should be chosen at random and periodically refreshed. We cannot say whether the key is periodically refreshed. It is possible the keys are refreshed, since the secret used for derivation is part of the dynamic vector, but we have not seen the change in the code.

The signature algorithm was unreadable in the limited time and therefore we cannot make any proper conclusion about its security or used algorithms.

The protocol does not ensure the integrity and authenticity of the communication, but it ensures the integrity and the authenticity of messages inside the `Secure Message` class. Everything else can be changed. The best practice is to sign every request and response, then the attacker cannot modify the communication.

In this case the man-in-the-middle attacker can transfer the communication in login protocol and then continue in the communication with server, pretending to be the client, using captured Access token.

Since messages are encrypted and signed by shared keys, the attacker cannot modify them without someone noticing. Messages also contain some nonce values which provide more randomization. One random nonce is a part of the message and the second is for example the timestamp inside the message body. The attacker could theoretically try to resend `SecureMessages` comming from the client. They would probably look valid, but it would not make much sence, since the core of the authorization is in the signature and OTP. Sent messages also contain timestamp, which probably makes them valid for only limited time. He could also try to repeat the message that contains the challenge from the server, but even if the client would respond with valid OTP, it would have no use for the attacker.

The most important part of the login protocol is the signature. The authorization of login operation is done by signing the challenge from the server and returning the OTP. From the side of the server, the man-in-the-middle attacker cannot effectively modify the challenge, since it is encrypted and signed by keys he does not have. From the side of the client he cannot try to guess OTP, since we can assume only limited number of attempts is allowed. The attacker can only transfer all these values.

OAuth 2.0 PKCE ensures that the attacker cannot use the captured code, which we have already mentioned in 2.12.

To summarize it, functions for the encryption and signature are chosen correctly and their usage ensures the integrity and authentization of the secure message structure.

The key derivation could be performed using different functinons, since AES ECB is not a key derivation function. Also the number of iterations of PBKDF2 should be in general higher, see the recommendation given by [56].

We also recommend to sign every request/response since the integrity of the communication is not ensured but `SecureMessages`.

The usage of the challenge-response algorithm for the authorization seems as a good idea, even though we do not know details. The challenge always contains random bytes, therefore it changes every time, which is desirable. From the code it also seems that the signature could use timestamp if configured properly.

5.1.2 Transaction

The signature is almost the same as the login. There are two differences.

The signed challenge contains important payment details. That is, account numbers, amount of money and currency. It also includes more bytes like the login challenge message, some of them can create a structure of the message.

The second difference is that the request with OTP also contains operationWysiwysData which is signed jwt token with with all details about the payment.

Since the algorithm for the payment is the same, the conclusion about the security would also be the same. We only note that the man-in-the-middle attacker can modify everything that is not signed, however, the last check that the user can see is obtained from the decrypted challenge which cannot be modified.

5.2 Practical part

In this section we briefly describe the practical part of the reversing.

5.2.1 Cryptographic libraries and sources

Similarly as before we obtained base.apk file from the mobile device.

We discovered the interesting package by searching for words in the captured communication.

Main methods of Digipass can be found in the package cz.kb.caas.mobile.security.token.services in the class `CryptoEngineServiceImpl`. The rest of cryptographic classes and methods are in the packages created during the obfuscation - therefore they have names as `Wa.d.f` and similar. Whole solution by Digipass is obfuscated.

5.2.2 Setup

The main setup was described in 1.3.

The application detects the rooted device, but the detection does not have to be bypassed. The only similar protection in the application is the certificate pinning which we have already described in 3.2.2. We used the same script `android-pinning-bypass_2.js` which we attach to bypass it.

- `frida -U -f cz.kb.ndb -l android-pinning-bypass_2.js`

5.2.3 Reversing

We used frida-trace 1.3.2 and jadx 1.3.2 for reversing. Pritning out inputs and outputs of traced methods helped us to deduce what they do. For illustration, this is part of the obfuscated signature function:

```
            bArr = c10;
            byteArrayOutputStream2.write(bArr12, z12 ? 1 : 0, 8);
            p.i(bArr12);
            r10 = z12;
        }
    }
    byte[] byteArray3 = byteArrayOutputStream2.toByteArray();
    p.i(bArr);
    j(eVar, a10, bArr12, byteArray3, r10);
    p.i(byteArray3);
    C2846b c2846b8 = a10.f13792e;
    if (c2846b8.f13769h) {
        bArr12[r10] = (byte) (bArr12[r10] ^ bArr12[4]);
        bArr12[1] = (byte) (bArr12[1] ^ bArr12[i11]);
        b13 = 2;
        bArr12[2] = (byte) (bArr12[2] ^ bArr12[i13]);
        bArr12[3] = (byte) (bArr12[3] ^ bArr12[7]);
    } else {
        b13 = 2;
    }
    if (a10.f13800m != b13 || c2846b8.f13774m || c2846b8.f13776o) {
        boolean z16 = c2846b8.f13779r;
        byte b29 = cVar.f13862b;
        if (!z16) {
            byte[] n10 = n(a10, bArr12);
            if (c2846b8.f13773l && ((b16 = a10.f13800m) != 2 || c2846b8.f13774m)) {
                C2846b c2846b9 = a10.f13792e;
                if (b16 == 1) {
                    n10[0] = (byte) ((n10[0] & 15) + (b29 << 4));
                } else if (b16 == 0) {
                    n10[0] = (byte) (b29 + 48);
                } else if (b16 == 2 && c2846b9.f13774m) {
                    n10[0] = (byte) Integer.toString(b29).charAt(0);
                }
            }
            if (!c2846b8.f13775n || ((b15 = a10.f13800m) == 2 && !c2846b8.f13776o)) {
                b14 = 0;
            } else {
                C2846b c2846b10 = a10.f13792e;
                byte b30 = (byte) (a10.f13801n % 10);
                if (b15 == 1) {
                    b14 = 0;
                    n10[0] = (byte) ((n10[0] & 240) + b30);
                } else {
                    b14 = 0;
                    if (b15 == 0) {
                        n10[1] = (byte) (b30 + 48);
                    } else if (b15 == 2 && c2846b10.f13776o) {
```

We mentioned before that we managed to obtain most of parameters that give us the configuration of the signature generation. We used frida-trace Web UI, where we just slightly modified the existing auto-generated script. We asked ChatGPT [69] to help us with the modification. Here we provide the script that could be useful when reverse engineering the signature method in future.

```
defineHandler({
  onEnter(log, args, state) {
    log(`e.a(${args.map(JSON.stringify).join(', ')})`);
  },
  onLeave(log, retval, state) {
    if (retval !== undefined) {
      log(`<= ${JSON.stringify(retval)}`);
      log(" a " + retval.a);
      log(" b " + retval.b);
      log(" c " + retval.c);
      log(" d " + retval.d);
      log(" e " + retval.e)
      log(" f " + retval.f);
      log(" g " + retval.g);
      log(" h " + retval.h);
      log(" i " + retval.i);
      log(" j " + retval.j);
      log(" k " + retval.k);
      log(" l " + retval.l);
      log(" m " + retval.m);
      log(" n " + retval.n);
      log(" o " + retval.o);
      log(" p " + retval.p);
      const trida = retval.e.value;
      log("retval.e = " + trida);
      const javaLangClass = trida.getClass();
      const fields = javaLangClass.getDeclaredFields();

      fields.forEach(function (field) {
      try {
        field.setAccessible(true);
        const value = field.get(trida);
        log("     " + field.getName() + " = " + value);
      } catch (e) {
        log("     " + field.getName() + " = err (" + e + ")");
      }});
    }
  }
});
```

5.3 Conclusion

In this section we conclude the results. Because of the limited time we did not manage to fully analyse the cryptography in the application. Therefore, some of the results may be imprecise or just assumptions.

As before, let us assume we do not trust TLS. The protocol does not provide integrity. Only `SecureMessages` sent by the server or by the client are encrypted and signed, thus the protocol provides integrity, authentication and confidentiality of these messages. Also the signature could not be produced by the man-in-the-middle attacker. The integrity of everything else is not ensured.

We do not know whether the protocol provides the forward secrecy. From what we have seen it does not, but we do not know how the dynamic vector changes

and therefore it is possible that the change in the dynamic vector provides forward secrecy. It would have to generate a new key for every session that would be difficult to compute backwards.

Let us again consider the following attackers:

1. The attacker that can capture the communication.

2. The attacker that can obtain data from the mobile device.

3. The attacker that can do both - capture the communication and obtain data from the mobile device

The protocol seems to be vulnerable as follows:

1. The attacker in this position can perform the man in the middle attack, since the scheme does not provide the integrity or authentication of every sent request/response. The attacker can transfer the login and then use the captured jwt to communicate with the server instead of the client. Or he could similarly communicate with the client while pretending to be the server.

   On the other hand, the login/transaction itself cannot be authorized by anyone else than the user who knows PIN.

2. It seems that the attacker in this position could easily obtain at least some keys - for example keys for the encryption and HMAC.

3. The attacker in this position can do everything that previous two attackers. As in the previous case he could theoretically try to bruteforce PIN and try to repeat the signature in the captured communcation. On the other hand we do not have enough information to make a conclusion whether this is possible in the protocol.

From the practical point of view, the application is obfuscated which complicates the reverse engineering. However, it does not make much effort to prevent the capturing of the communication or to prevent the debugger usage.

Conclusion

The aim of this thesis was to analyse the cryptographic layers developed over the TLS tunnel in three mobile banking applications. We analysed the cryptography used by MyAir, Partners and KB+.

We will conclude main results of the analysis.

1. Assume that the attacker can perform the passive man in the middle attack.

   MyAir uses HMAC on every request and response, but it signs only the body of the request/response. Therefore the attacker can send arbitrary request where he adds the body and HMAC header from some captured request and his request seems valid as a part of the current session.

   In Partners the attacker can transfer the communication during the login protocol and then he can use the captured Access token and send his own request to the server. That would grant him the access.

   The situation is same in KB+ where the attacker can use captured JWT token to communicate with the server and pretend to be the client.

2. Assume the attacker can perform the active man in the middle attack. This attacker can do everything as the previous pasive attacker.

   The application MyAir signs all request/response bodies, therefore they cannot be modified, although it is possible to perform the attack described above. However, sensitive operations like login or the payment cannot be affected.

   The Partners application seems to be the most vulnerable against the active man in the middle attack from examined applications. Firstly, the attacker can capture signed login requests from the client pretending to be the server, then abort the communication with the client and use captured requests for his own login. Secondly, the attacker can modify payment transaction data since they are not signed. He can change the number of an account, amount of money, etc. This can be done in both directions thus the change is not visible in the application for the user.

   KB+ does not sign request/responses, but it does not have the same problem since it encrypts and signs transaction data and during the authorization it signs partially randomized challenge sent by the server.

3. Let us consider a different situation, let us suppose the attacker can obtain data from the device.

   Then in MyAir the attacker cannot obtain any important value, only salts and similar values.

   In Partners the attacker could probably obtain KEY_TRANSPORT and KEY_SIGNATURE_POSSESION that is encrypted by the key associated with the device. Theoretically, the attacker can also obtain the counter. However, this is not sufficient for the authorization. The KEY_TRANSPORT can be used to decrypt the encrypted Activation Status.

In KB+ it seems that the attacker can obtain at least some keys that are derived from values stored on the device. That would suffice for singing or encryption, but it would not be enough for the authorization. We note that the application was not examined in sufficient detail to make the conclusion.

4. Let us imagine an attacker that can obtain data from the device and also capture the communication, but he cannot modify it. This attacker can do everything that can do attackers in 1) and 3). Moreover, he could attack as follows.

   In the case of MyAir he could capture the the communcation with the authorization using OCRA. The attacker can obtain OCRA_salt, master_salt, encrypted master_secret and encrypted OCRA_secret. He could theoretically try to bruteforce PIN, he could try to decrypt master_secret and then use it to decrypt OCRA_secret and compute the signature using captured data. If the signature would be same as in the communication, he would know that he has correct master secret, correct PIN and correct OCRA_secret.

   In the case of Partners, if the attacker could obtain the data sometime before the signature was performed and then to capture the communication, he could also try to bruteforce PIN and to obtain KEY_SIGNATURE_KNOWLEDGE. We note that we need to know the value of the counter in advance, because it cannot be easily computed backwards.

   We did not examine KB+ as thoroughly as we wanted to, therefore we cannot make conclusions. It seems that the secret used in the signature algorithm is encrypted by the key that was derived using PIN and PBKDF2 with 1000 iteration which is possible to bruteforce. Then the attacker could try to decrypt the signature secret. But since we do not know enough, this is just an assumption.

5. The last possible situation is the attacker that can modify the comunication (active man in the middle) and he can also obtain stored data. This attacker is the most powerful and he can do everything as previous attackers. However, he does not seem to have any bigger advantage if we compare it with the attacker in 4).

The analysis of applications was not only theoretical, but most of the work was practical. Besides usual technical issues, we have also encountered several bigger problems, which have not yet been mentioned in this thesis. Many applications have stronger and better protection against the reverse engineering than those we have described. Some of these protections are difficult to bypass, and it is not possible to use a generic script on them.

Firstly, some of applications use debugger detection - then it is difficult to use frida to find the exact place in the code that has to be bypassed. As an example we can mention SmartBanking by Unicredit. The appliacation lets the debugger run before the login, then it detects frida and ends it. The application is also unusually obfuscated, which makes bypassing really difficult. We did not manage to bypass these protections in given time.

Some of applications have interesting protection against the proxy and capturing the communication. For example ČSOB Smart and ČSOB Smart Klíč. We also did not managed to bypass this protection in given time.
ČSOB Smart Klíč also detects the rooted device. However, this protection can be easily bypassed by writing frida script that rewrites one specific method.

Unfortunately, it cannot be predicted in advance which of the given applications has strong protections and which will be easy to bypass. That is one of the reasons why we did not have time to examine the last application in full detail.

In future it would be interesting to examine KB+ more thoroughly. It would be also interesting to examine applications which have stronger protection against the reverse engineering and therefore we did not have time for their analysis.

A Attachments

A.1 Our implementation

- activation_code_decrypt.py.txt
- change_password.py.txt
- decrypt_card_info.py.txt
- hmac_komunikace.py.txt
- login.py.txt
- OCRA.py.txt
- pa_signature.py.txt
- registration.py.txt
- signature_OCRA.java.txt
- authorize_sestav_data_to_sign.java.txt

A.2 Scripts for bypassing protections

- script_modification.js.txt
- rootbeer_bypass.js.txt
- multiple_bp.js.txt
- jailmonkey_bypass.js.txt
- android-pinning-bypass_2.js.txt

A.3 Example outputs and hooking scripts

- Partners_hook_native_partial_input_to_SHA256.js.txt
- hook_native_login.js.txt
- attachement_login_output_master_thesis.txt