Selective Identity Disclosure

tomas.matos@ua.pt, 108624
goncalomf@ua.pt, 107853

December 26, 2024

1 Introduction

This project is the second challenge given in the Applied Cryptography course from the Cybersecurity MSc at the University of Aveiro. The main objective is to develop system that allows the owner of a Digital Citizen Card (DCC) to control the selective disclosure of his identity attributes in a secure and privacy-preserving manner. The detailed assignment can be found here.

2 Cryptosystem for Key Pairs

2.1 Owner

The keys used for the owner's digital identity are those embedded within the Portuguese Citizen Card. The **public key** can be retrieved from the certificates stored on the card when extracting the identity data, and it plays a crucial role in verifying the authenticity of data associated with the cardholder. The **private key** remains secure and is not extractable, ensuring that sensitive cryptographic operations are kept protected, it is stored securely within the card's chip and is never exposed, thereby preventing unauthorized access.

Despite that, it is still possible to sign data using the private key, provided that the owner has access to his authentication PIN. This PIN, suppostly known only by the owner, is used to unlock the private key and perform digital signing operations, which are essential for authenticating the cardholder's identity in various secure transactions or digital interactions. This security mechanism ensures that the cardholder retains control over his private key while enabling secure and verifiable signing of data.

2.2 Issuer

To generate the key pair and the self-signed certificate, we developed a Python script that utilizes the **cryptography** library. The script begins by generating an RSA key pair, consisting of a private key and a public key. The generated keys are saved in PEM format, a standard for encoding cryptographic objects.

Here is the function that generates the RSA key pair and saves them:

Listing 1: Key Generation

```
def generate_rsa_key_pair():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()

    # Save private key to PEM format
    with open("private_key.pem", "wb") as private_pem_file:
        private_pem = private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        )
        private_pem_file.write(private_pem)
        print("Private␣key␣saved␣to␣private_key.pem")

    # Save public key to PEM format
    with open("public_key.pem", "wb") as public_pem_file:
        public_pem = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        public_pem_file.write(public_pem)
        print("Public␣key␣saved␣to␣public_key.pem")

    return private_key, public_key
```

Once the keys are generated, the next step is to create a self-signed certificate. This certificate is digitally signed by the private key and contains information about the subject and issuer (in this case, both are the same). The certificate includes the public key, the validity period, and other essential information, such as the serial number and basic constraints indicating that the certificate is a CA (Certificate Authority) certificate.

Below is the function to generate the self-signed certificate:

Listing 2: Generate self-signed Certificate

```
def generate_self_signed_certificate(private_key, public_key):
    subject = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, "PT"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, "Aveiro"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, "Aveiro"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, "DETI"),
        x509.NameAttribute(NameOID.COMMON_NAME, "deti.pt"),
    ])

    issuer = subject

    certificate = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(public_key)
        .serial_number(x509.random_serial_number())
        .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
        .not_valid_after(datetime.datetime.now(datetime.timezone.utc) + datetime.
            timedelta(days=365))
        .add_extension(
            x509.BasicConstraints(ca=True, path_length=None), critical=True
        )
        .sign(private_key, hashes.MD5(), default_backend())
    )

    # Save certificate to a PEM file
    with open("self_signed_certificate.pem", "wb") as f:
        f.write(certificate.public_bytes(serialization.Encoding.PEM))
    print("Self-signed␣certificate␣saved␣to␣self_signed_certificate.pem")
```

In a certificate (like a self-signed certificate), there are two important roles: the subject and the issuer.

- **Subject**: Represents the entity the certificate is about — for example, the individual or organization the certificate is issued to (the owner of the certificate).
- **Issuer**: Refers to the entity that issues the certificate — typically a trusted certificate authority (CA) that vouches for the identity of the subject.

In a self-signed certificate, the issuer is the same as the subject because the same entity is both generating and signing the certificate. Essentially, the entity is both asserting its own identity (as the subject) and signing its own certificate (as the issuer).

3 Extract Data from CC

To extract the identity attributes, a card reader is used, and to specifically load data from the Portuguese Citizen Card (CC), the library **pteidlibj** is utilized in a Java program.

After loading the library, we proceed by setting up the reader and checking if the card is present. Once it is detected, the program retrieves the identity data associated with the card, including the personal information and certificates. Below is the code snippet that demonstrates how the data is loaded from the Citizen Card:

Listing 3: CcData.java

```
        PTEID_ReaderSet.initSDK();

        PTEID_ReaderContext reader = PTEID_ReaderSet.instance().getReader();

        if (reader.isCardPresent()) {
            try {
                PTEID_EIDCard card = reader.getEIDCard();
                PTEID_EId eid = card.getID();

                PTEID_Certificates certs = card.getCertificates();

                for (long i = 0; i < certs.countAll(); i++) {
                    PTEID_Certificate cert = certs.getCert(i);
                    byte[] certBytes = cert.getCertData().GetBytes();
                    CertificateFactory factory = CertificateFactory.getInstance("X.509
                        ");
                    InputStream certInputStream = new ByteArrayInputStream(certBytes);
                    Certificate certificate = factory.generateCertificate(
                        certInputStream);
                    PublicKey publicKey = certificate.getPublicKey();

                    String pemKey = exportToPEM(publicKey);
                    RSAPublicKey rsaPublicKey = (RSAPublicKey) publicKey;
                    BigInteger modulus = rsaPublicKey.getModulus();
                    int keySize = modulus.bitLength();
                    System.out.println("Public␣Key:␣\n" + pemKey);
                    System.out.println("Key␣Size:␣" + keySize + "␣bits");

                    break;
                }


                String nome = eid.getGivenName();
                String sobrenome = eid.getSurname();
                String nrCC = eid.getDocumentNumber();

                ....

                System.out.println("Name:␣" + nome);
                System.out.println("Surname:␣" + sobrenome);
                System.out.println("Document␣Number:␣" + nrCC);

                ....
```

By running this program with a Citizen Card (CC) in the reader, it is possible to extract various identity attributes such as the **Name** (given name and surname), **Document Number** (unique identifier for the Citizen Card), **Gender**, **Date of Birth**, **Document Version**, and **Document Type**. It also retrieves validity-related information, including the **Validity Start Date** and **Validity End Date**, along with other personal details like **Height**, **Civilian ID Number**, **Tax Number**, **Social Security Number**, and **Health Number**. Additional information about the **issuing entity**, **Local of Request**, and parental details such as **Father's Name** and **Mother's Name** are also available. The program also extracts the cardholder's **Nationality** and **Document PAN** Moreover, is retrievered the **Photo** of the cardholder in both raw and encoded formats. Finally, the Public Key associated with the card are extracted from the digital **certificates** stored on the card.

4 Aplications

4.1 Request DCC (req_dcc)

To use this application, the user must have access to a card reader. Once the application is running, a menu is displayed, allowing users to request for a Digital Citizen Card.

The following sections will detail the application flow for each request in chronological order.

4.1.1 Get CC Data

When a user requests a Digital Citizen Card (DCC), the process begins with extracting data from the Portuguese Citizen Card (CC) using the previously explained Java code.

Since the application is developed in Python, we executed the Java code in a subprocess using Python's **subprocess** library. The output from the Java program (stdout) was then processed and converted into a dictionary for further use.

Listing 4: Get data

```python
def get_portuguese_cc_data():
    stdout, stderr = run_java_program()

    if stdout:
        extracted_data = extract_data(stdout)
        return extracted_data

    else:
        print("Failed␣to␣execute␣the␣Java␣program.")
        print(stderr)
        return None
```

4.1.2 Create Incomplete DCC

From the extracted data, the next step is to create an incomplete Digital Citizen Card (DCC) ready to be signed by the issuer. The function create_dcc, shown below, takes the Citizen Card (CC) data as input, secures it using a user-provided secret, and generates a set of commitments for the identity attributes. These commitments ensure the integrity and privacy of the data before issuing the DCC.

Listing 5: Create Incomplete DCC

```python
def create_dcc(cc_data):
    password = input("--␣Insert␣a␣secret:␣")
    commitments = []

    for label, value in cc_data.items():
        if value is None or label == "public_key_pem" or label == "key_size":
            continue
        mask = derive_mask(password, label)

        commitment = create_commitment(label, value, mask)
        commitments.append({
            "label":label,
            "value": value,
            "commitment": commitment
        })

    dcc = {
        "identity_attributes": commitments,
        "digest_function": "SHA-1",
        "public_key": {
            "value": cc_data["public_key_pem"],
            "key_size": cc_data["key_size"]
        }
    }
    return dcc
```

The function **derive_mask** generates a unique mask for each attribute by hashing the combination of the secret password and the attribute name and the **create_commitment** combines the attribute name, value, and mask into a hash, creating a secure commitment for that attribute. Both use the SHA-1 hashing algorithm to ensure data integrity and work together to securely prepare the data for the next steps.

Listing 6: Commitment Functions

```
def derive_mask(password, attribute_name):
    return hashlib.sha1(f"{password}{attribute_name}".encode()).hexdigest()

def create_commitment(attribute_name, attribute_value, mask):
    combined = f"{attribute_name}{attribute_value}{mask}".encode()
    return hashlib.sha1(combined).hexdigest()
```

Incomplete DCC model:

```
{
    "identity_attributes": [
        {
            "label": "name",
            "value": "",
            "commitment": ""
        },
        {
            "label": "surname",
            "value": "",
            "commitment": ""
        },
        ...
    ],
    "digest_function": "",
    "public_key": {
        "value": "",
        "key_size": "3072"
    }
}
```

4.1.3 Comunicate with gen_dcc

Once the incomplete DCC is created, it is time to communicate with the gen_dcc application to finalize the DCC. After the application processes the request and sends the completed DCC back, the issuer's signature is verified. If the verification is successful, the DCC is stored.

Listing 7: Comunication

```
message = {"type": "request", "dcc": dcc}
response = send_receive_dictionary(message)
final_dcc = response["dcc"]

issuer_sign = final_dcc["issuer_signature"]["value"]
issuer_cert = final_dcc["issuer_signature"]["certificate"]
verify_issuer = validate_issuer_signature(issuer_sign, issuer_cert, final_dcc)

if verify_issuer:
    print("DCC␣is␣valid␣and␣signed␣by␣the␣trusted␣issuer.")

    with open(f'dccs/dcc_{final_dcc['identity_attributes'][12]['value']}.json'
        , 'w') as json_file:
        json.dump(final_dcc, json_file, indent=4)

else:
    print("DCC␣validation␣failed␣or␣issuer␣signature␣is␣invalid.")
```

4.2 Generate DCC (gen_dcc)

This application simulates a trusted entity that serves as the root of trust for every Digital Citizen Card (DCC). By acting as the authority, it ensures that each DCC is securely issued and recognized as authentic.

To create a realistic simulation, this application operates as a server, allowing it to handle requests from the **req_dcc** application.

Once a DCC request is made the flow happens like this:

4.2.1 Receive and Load Data

First of all, the incomplete DCC data is received and the issuer private key and certificate is loaded:

Listing 8: Load Data

```
        json_data = json.loads(buffer.decode('utf-8'))
        dcc_data = json_data["dcc"]

        issuer_private_key = load_issuer_private_key('issuer/private_key.pem')
        issuer_cert = load_issuer_certificate('issuer/self_signed_certificate.pem')
```

4.2.2 Issuer Sign Incomplete DCC

In this section, the issuer processes and signs the incomplete DCC received. The data to be signed, as shown in the code snippet above, consists of a list containing all the commitments and the public key of the DCC owners.

Listing 9: Filter Data to Sign

```
        only_commitment = []
        for att in dcc_data["identity_attributes"]:
            only_commitment.append(att["commitment"])

        only_commitment.append(dcc_data["public_key"])

        serialized_only_commitment = json.dumps(
                only_commitment,
                separators=(',', ':')
            ).encode('utf-8')

        issuer_signature = sign_with_issuer_key(
            issuer_private_key,
            serialized_only_commitment
        )
```

The **sign_with_issuer_key** function generates a cryptographic signature for the serialized DCC data using the issuer's private key. First, it hashes the data with the SHA-512 algorithm. Then, the function signs the hashed data using the RSA-PKCS1v1.5 padding scheme, which is a widely used method for secure digital signatures.

The function returns a structured signature object, which includes the hexadecimal representation of the signature, a timestamp for when the signature was created, and a description of the signature algorithm used.

Listing 10: Sign DCC

```
        def sign_with_issuer_key(private_key, data_to_sign):
            digest = Hash(hashes.SHA512())
            digest.update(data_to_sign)
            hashed_data = digest.finalize()

            signature = private_key.sign(
                hashed_data,
                padding.PKCS1v15(),
                Prehashed(hashes.SHA512())
            )

            full_sign = {
                "value": signature.hex(),
                "timestamp": datetime.now(timezone.utc).isoformat(),
                "description": "Issuer␣signature␣using␣SHA-512␣and␣RSA-PKCS1v1.5"
            }

            return full_sign
```

DCC model:

```
{
    "identity_attributes": [
        {
            "label": "name",
            "value": "",
            "commitment": ""
        },
        {
            "label": "surname",
            "value": "",
            "commitment": ""
        },
        ...
    ],
    "digest_function": "",
    "public_key": {
        "value": "",
        "key_size": "3072"
    },
    "issuer_signature": {
        "value": "",
        "timestamp": "",
        "description": "",
        "certificate": ""
    }
}
```

4.2.3 Finalize and Send DCC

With the DCC completed, a response message is sent to the the **req_dcc** application.

Listing 11: Finalize and send

```
        dcc_data["issuer_signature"] = issuer_signature
        dcc_data["issuer_signature"]["certificate"] = issuer_cert

        response = {
            "status": "success",
            "type" : "dcc_complete",
            "dcc": dcc_data
        }

        conn.sendall(json.dumps(response, default=str).encode('utf-8'))
```

4.3 Generate minimalistic DCC (gen_min_dcc)

This application is used by the owner of a Citizen Card after their DCC has been created and saved. It allows the user to restrict the identity attributes displayed on the digital card. When the application is running, a menu is presented, enabling the user to generate a minDCC by loading the JSON corresponding to his complete DCC.

4.3.1 Validate DCC Issuer

The first step in generating the minimalistic version of the DCC is to ensure that the loaded DCC is valid and signed by a trusted issuer. This process involves verifying the issuer's signature to confirm that the DCC's data has not been tampered with and that it was issued by a legitimate entity. The minDCC generation can only proceed if the validation is successfully completed.

Listing 12: Validate issuer

```
verify_issuer = validate_issuer_signature(issuer_sign, issuer_cert, dcc_data)

if verify_issuer:
    print("DCC␣is␣valid␣and␣signed␣by␣the␣trusted␣issuer.")
    ...
else:
    print("DCC␣validation␣failed␣or␣issuer␣signature␣is␣invalid.")
```

4.3.2 Remove Attributes

In this section the user is able to exclude the identity attributes that he wants:

Listing 13: Validate issuer

```
attributes = dcc_data["identity_attributes"]
labels = [dic["label"] for dic in attributes]
civilian_id_number = ""
to_remove = []
idx = 0

for label in labels:
    if label == "civilian_id_number":
        civilian_id_number = attributes[idx]["value"]
        continue

    response = input(f"␣␣Remove␣attribute␣'{label}'?␣(y/n)").strip()

    if response.lower() == "y":
        to_remove.append(label)
        idx += 1

new_labels = [label for label in labels if label not in to_remove]
```

4.3.3 Generate Final minDCC

Although the goal is to generate a minimalistic DCC, certain key attributes must be retained in this version. These include the commitments of all attributes, the digest function data, the owner's public key, and the issuer's signature:

Listing 14: Creation of minDCC

```
        dcc_min = {}
        only_commitment = []

        for att in dcc_data["identity_attributes"]:
            only_commitment.append(att["commitment"])

        dcc_min["commitment"] = only_commitment
        dcc_min["digest_function"] = dcc_data["digest_function"]
        dcc_min["public_key"] = dcc_data["public_key"]
        dcc_min["issuer_signature"] = dcc_data["issuer_signature"]
```

Based on the attributes selected by the owner to be kept, a mask is generated to protect the sensitive data. The mask is created using the owner's secret, ensuring that the values remain hidden while still being verifiable.

The owner is prompts the same secret as the one inserted in the **req_dcc** app. Then, for each attribute that the owner has chosen to keep the corresponding mask is derived using the derive_mask function. The mask ensures that the value of the attribute is obfuscated, while still allowing it to be validated later.

Listing 15: Mask Creation

```
        password = input("--␣Insert␣the␣secret:␣")

        attributes = [
            {"label":dic["label"], "value": (dic["value"], derive_mask(password, dic["
                label"]))}
            for dic in attributes if dic["label"] in new_labels
        ]

        dcc_min["identity_attributes"] = attributes
```

To complete the minDCC, the only remaining step is to add the owner's signature over the key attributes previously mentioned:

Listing 16: Complete minDCC

```
        dccmin_data_to_sign = json.dumps(dcc_min).encode('utf-8')

        signature, timestamp = sign_with_cc(dccmin_data_to_sign)

        dcc_min["owner_signature"] = {
            "value": signature.hex(),
            "timestamp" : timestamp,
            "description": "RSA,␣using␣the␣Citizen␣Card␣private␣key␣for␣signing."
        }

        with open(f'min_dccs/dcc_min_{civilian_id_number}.json', 'w') as json_file:
            json.dump(dcc_min, json_file, indent=4)
```

Finally, the completed minDCC is saved as a JSON file, using the civilian's ID number to create a unique filename.

The **sign_with_cc** function is responsible for signing the minDCC data using the owner's Citizen Card private key. It begins by checking for a slot containing the label "CARTÃO DE CIDADAO" (Citizen Card) to identify the correct token. Once the correct slot is found, it opens a session and searches for the private key labeled 'CITIZEN AUTHENTICATION KEY' within the session. If the private key is found, the function proceeds to sign the data using the RSA-PKCS mechanism with SHA-256.

Listing 17: Sign with CC

```
def sign_with_cc(data_to_sign):
    slots = pkcs11.getSlotList()

    for slot in slots:
        token_info = pkcs11.getTokenInfo(slot)
        if 'CARTAO␣DE␣CIDADAO' in token_info.label:
            session = pkcs11.openSession(slot)
            try:
                priv_key = session.findObjects([
                    (CKA_CLASS, CKO_PRIVATE_KEY),
                    (CKA_LABEL, 'CITIZEN␣AUTHENTICATION␣KEY')
                ])[0]

                mechanism = Mechanism(CKM_SHA256_RSA_PKCS)
                signature = session.sign(priv_key, data_to_sign, mechanism)
                session.closeSession()

                return bytes(signature), datetime.now(timezone.utc).isoformat
                    ()
            except Exception as e:
                session.closeSession()
                raise Exception(f"Failed␣to␣sign␣data:␣{e}")

    raise Exception("Citizen␣Card␣not␣found␣or␣private␣key␣not␣accessible.")
```

minDCC model (depends on the removed attributes):

```
{
    "commitment": [],
    "digest_function": "",
    "identity_attributes": [
        {
            "label": "civilian_id_number",
            "value": [
                "",
                ""
            ]
        }
        ...
    ],
    "public_key": {
        "value": "",
        "key_size": ""
    },
    "issuer_signature": {
        "value": "",
        "timestamp": "",
        "description": "",
        "certificate": ""
    },
    "owner_signature": {
        "value": "",
        "timestamp": "",
        "description": ""
    }
}
```

4.4 Validate minimalistic DCC (check_dcc)

This application is designed for users who want to verify the integrity of a minimalistic DCC. Upon launching the application, users are presented with a menu that allows them to load a minDCC, validate the issuer and owner signatures, verify the integrity of the commitments, and review the data contained within the digital card.

4.4.1 Validate Issue Signature

To validate the issuer's signature, only the signature value and the issuer's self-signed certificate, which is embedded in the loaded data, are required. Once the signature is applied to the commitments within the digital citizen card and the owner's public key, this validation ensures that the issuer's identity is genuine and that the data has remained unaltered since it was signed.

More specifically, if any commitment or the owner's public key has been changed, whether intentionally or unintentionally, the validation will return the message: **"DCC validation failed or issuer signature is invalid."**

Although the validation primarily focuses on the commitments and the public key, it can also assess the integrity of identity attributes, such as their values and associated labels, using their mask. This allows for a deeper level of verification, as it is explained in the 4.4.3 section, ensuring that all aspects of the digital card remain consistent and trustworthy.

Listing 18: Validate Issue Signature

```
        issuer_sign = dcc_data["issuer_signature"]["value"]
        issuer_cert = dcc_data["issuer_signature"]["certificate"]

        verify_issuer = validate_issuer_signature(issuer_sign, issuer_cert, dcc_data)

        if verify_issuer:
            print("DCC␣is␣valid␣and␣signed␣by␣the␣trusted␣issuer.")
        else:
            print("DCC␣validation␣failed␣or␣issuer␣signature␣is␣invalid.")
```

4.4.2 Validate Owner Signature

To validate the owner's signature, the process requires the owner's public key, which is embedded within the minDCC, along with the signature itself. First, the data signed by the owner must match the data that the owner really signed, so it is necessary to exclude the signature from the full minDCC. Once it is removed the pre-processment is finished and the signature can be validated.

If the validation fails, we can be certain that the minDCC has been altered and therefore cannot be trusted. This ensures the integrity of the digital card, confirming that the owner's consent is properly reflected in the signed data.

Listing 19: Validate Owner Signature

```
        signature = dcc_data["owner_signature"]["value"]
        public_key_data = dcc_data["public_key"]["value"]

        dcc_without_signature = {key: dcc_data[key] for key in dcc_data if key != "
            owner_signature"}

        serialized_data = json.dumps(dcc_without_signature).encode('utf-8')

        is_valid = verify_signature(public_key_data, serialized_data, bytes.fromhex(
            signature))
        if not is_valid:
            print("Signature␣verification␣failed.␣Data␣integrity␣compromised.")
        else:
            print("Signature␣verified␣successfully.␣Data␣integrity␣intact.")
```

4.4.3 Validate Commitments

Validating the commitments involves verifying that the data presented in the identity attributes in the minDCC matches the expected commitments. This requires the commitments and the identity attributes, which include the masks used to generate those commitments. By creating new commitments based on the identity attributes and comparing them with the existing commitments in the minDCC, we can ensure that the data remains unchanged. If any discrepancies are found between the generated commitments and those stored in the minDCC, it indicates a compromise in the data's integrity.

Listing 20: Validate Commitments

```
        commitments = dcc_data["commitment"]
        attributes = dcc_data["identity_attributes"]

        are_included = True
        for at in attributes:
            value = at["value"][0]
            mask = at["value"][1]
            label = at["label"]
            commit = create_commitment(label, value, mask)

            if commit not in commitments:
                print("Changed␣->␣", value, "␣(value)␣or␣", label, "␣(label)␣or␣",
                    mask , "␣(mask)␣")
                are_included = False

        if are_included:
            print("Data␣integrity␣intact.")
        else :
            print("Data␣integrity␣compromised.")
```

4.4.4 Check Data

This option allows users to review the identity attributes stored in the minDCC. It displays each attribute's label and value, and shows the CC image data if the owner has chosen to include it in the minDCC.

Listing 21: Check Data

```
        attributes = dcc_data["identity_attributes"]

        for at in attributes:
            value = at["value"][0]
            label = at["label"]

            if label == "image_bytes":
                display_image(value)
                continue

            print(label, ":␣", value)
```

References

Manual de SDK – Middleware do Cartão de Cidadão and Chat GPT