Universidade de Aveiro

Selective Identity Disclosure

Criptografia Aplicada

Diogo Almeida nº108902

Mestrado Cibersegurança

Professores: André Zúquete e Tomás Silva

22 de dezembro de 2024

Capítulo 1

Introdução

O mundo digital tem estado em constante evolução, e garantir a segurança e a privacidade é um ponto crucial nos dias de hoje. Os métodos tradicionais de identificação, como o cartão de cidadão, apresentam desafios em relação à privacidade dos dados.

O Selective Identity Disclosure (SID) apresenta uma abordagem ao problema de privacidade do cartão de cidadão, permitindo aos utilizadores esconder atributos do cartão, mas mantendo a garantia da autenticidade dos dados. Isto é possível através do uso de um DCC (Digital Citizen Card), um cartão de cidadão digital assinado por uma entidade superior, que garante a autenticidade do mesmo.

Neste projeto, foram desenvolvidas quatro aplicações: uma para criar o cartão digital, uma para que uma entidade superior assine o cartão, uma para esconder atributos do cartão e, por fim, uma para verificar a autenticidade dos dados e garantir que o cartão está íntegro.

Capítulo 2

req_dcc

2.0.1 Ler Dados e gerar pedido

O passo inicialmente implementado foi a leitura do cartão de cidadão utilizando a biblioteca **PyKCS11** do python. Com este passo são retirados os seguintes dados do cartão de cidadão do owner: nome, cc number, pais, organização e data de nascimento.

Listing 2.1: Exemplo dos dados retirados

```
if isinstance(cert_value, bytes):
    # Parse the certificate
    cert = load_der_x509_certificate(cert_value)
    subject = cert.subject
    # Extract details
    full_name = subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
    id_number = subject.get_attributes_for_oid(NameOID.SERIAL_NUMBER)[0].value

    # Additional data (may require custom OIDs for NIF, NSS, Utent, Birth Date)
    country = (
        subject.get_attributes_for_oid(NameOID.COUNTRY_NAME)[0].value
        if subject.get_attributes_for_oid(NameOID.COUNTRY_NAME) else None
    )
    organization = (
        subject.get_attributes_for_oid(NameOID.ORGANIZATION_NAME)[0].value
        if subject.get_attributes_for_oid(NameOID.ORGANIZATION_NAME) else None
    )
```

Depois é pedida uma senha secreta ao owner. Com esta senha é gerado uma máscara para cada atributo utilizando o algoritmo SHA256. Com a máscara e o respetivo valor do atributo é então gerado um valor de commitment utilizando SHA256.

É retirado do cartão de cidadão do utilizador a chave pública do owner e colocada também no pedido.json. O pedido final contém os atributos retirados do cartão, os valores de commitment e a chave pública do owner.

Listing 2.2: JSON do pedido

```
{
    "attributes": [
        {
            "label": "nome",
            "value": "Owner name",
            "commitment": "9
                bff8165f1abf93c99bec5dd17bcda5f9f8bb6e3cc8d6d05577a9b1a2279f9d9"
        },
        {
            "label": "morada",
            "value": "Rua",
            "commitment": "
                eb8efaed4cc544df68730bae65293ca74e8ba448cdd9b893fd94b96fb819e7a4"
        },
```

```
        {
            "label": "data_nascimento",
            "value": "01/01/2000",
            "commitment": "114
                b6d2001154bb8268584ed54d21bf394465285302117c003a78709a4135dd5"
        },
        {
            "label": "cc_number",
            "value": "number"
            "commitment": "
                fee113691dcce3a8ee4a6f2368acc0c8a1d93dfea98abc845053d7cdd2dc137c"
        }
    ],
    "digest_function": {
        "type": "SHA-256"
    },
    "chave_publica_owner": [
        {
            "value":"-----BEGIN PUBLIC KEY-----\nMIIBIjWHPdgIg2s5XEjLCJO+gwTfe\nZQAB
                \n-----END PUBLIC KEY-----\n"
            "description of key nature": "some description of the nature of the
                asymmetric cryptosystem of this key"
        }
    ]
}
```

De seguida a aplicação possui sockets para enviar este pedido para a aplicação do issuer para ser assinada por o mesmo.

Quando recebe um pedido assinado por um issuer, esta aplicação possui uma função que vai ler o certificado e ler os dados e verificar se a assinatura e os dados estão corretos de forma a garantir que os dados estão íntegros e que os dados assinados pelo issuer não foram alterados.

Listing 2.3: Verificar integridade dos dados

```
ompromissos_e_chave_publica = []
        for atributo in resposta["attributes"]:
            compromissos_e_chave_publica.append(atributo["commitment"])

        compromissos_e_chave_publica.append(resposta["chave_publica_owner"][0]["
            value"])

        # Gerar os dados para assinar
        dados_para_assinar = json.dumps(compromissos_e_chave_publica, sort_keys=True
            )
        print("Dados para assinar:", dados_para_assinar)

        hash_dados = hashes.Hash(hashes.SHA1(), backend=default_backend())
        hash_dados.update(dados_para_assinar.encode())
        digest = hash_dados.finalize()
        # Verificar a assinatura

        chave_publica.verify(
            assinatura,
            digest,
            ec.ECDSA(hashes.SHA1())
        )

        print("Assinatura valida: os dados estao integros.")
        return True
```

Capítulo 3

gen_dcc

3.0.1 Receber um pedido e assinar

Primeiramente foram gerados uma chave privada e um certificado. Para gerar isto foi utilizado o sistema curvas elipticas do OpenSSL.

Listing 3.1: Gerar certificado e chave privada

```
openssl ecparam -name prime256v1 -genkey -noout -out chave_privada_ec.pem

openssl req -new -x509 -key chave_privada_ec.pem -out certificado_autoassinado.pem -
    days 365
```

O gen_dcc utiliza sockets e fica sempre à espera de um JSON de pedido. Ao receber um pedido é carregada a chave privada gerada anteriormente,

Listing 3.2: Carregar chave privada

```
with open("chave_privada_ec.pem", "rb") as f:
    chave_privada_issuer = serialization.load_pem_private_key(
        f.read(),
        password=None,
        backend=default_backend()
    )
```

São então lidos todos os commitments e a chave pública do owner presente no pedido JSON e estes valores são assinados com a chave privada do issuer.

Listing 3.3: Dados para assinar

```
compromissos_e_chave_publica = []
    for atributo in pedido_dcc["attributes"]:
        compromissos_e_chave_publica.append(atributo["commitment"])

    compromissos_e_chave_publica.append(pedido_dcc["chave_publica_owner"][0]["value"
        ])

    # Gerar os dados para assinar
    dados_para_assinar = json.dumps(compromissos_e_chave_publica, sort_keys=True)
    assinatura_issuer = assinar_dados(chave_privada_issuer, dados_para_assinar)
```

Os dados são assinados utilizando o algoritmo SHA1 para não utilizar o mesmo algoritmo utilizado no req_dcc. São utilizadas curvas elipticas para fazer a assinatura.

Listing 3.4: Assinar dados

```python
def assinar_dados(chave_privada, dados):
    # Calcula o hash dos dados
    hash_dados = hashes.Hash(hashes.SHA1(), backend=default_backend())
    hash_dados.update(dados.encode())
    digest = hash_dados.finalize()

    # Assina os dados com a chave privada
    assinatura = chave_privada.sign(
        digest,
        ec.ECDSA(hashes.SHA1())
    )
    return assinatura
```

É então adicionado ao pedido JSON mais um campo que contém a assinatura dos dados, o timestamp da assinatura, uma descrição da assinatura e o certificado do issuer.

Listing 3.5: Novo campo do JSON

```
"Issuer_signature_over_comminments_and_public_key": [
        {
            "value": "304502204d7b2e9190a4c87a967996cbe997218bf",
            "timestamp": 1734465734.8392873,
            "description": "Assinatura com Elliptic Curve (ECDSA) e SHA-256",
            "issuer_certificate": "-----BEGIN CERTIFICATE-----\
                nMIIB0TCCAXegAwIBAgIUfn4Iv91THVrwq1C0oHgww0tZIVcwCgYIKoZIzj0EAwIw\
                nPjELMAkGA1UEBhMCUFQxDzANBgNVBAgMBkF2ZWlybzEPMA0GA1UEBwwGQXZ\n-----
                END CERTIFICATE-----\n"
        }
    ]
```

É então enviado o DCC final para o owner e este guarda-o num ficheiro json.

Capítulo 4

gen_min_dcc

4.0.1 Esconder atributos do DCC

Nesta aplicação, primeiramente é pedido ao owner um DCC em formato JSON. Caso o utilizador passe um DCC com formato válido são então apresentados ao owner todos os atributos presentes no mesmo. O owner pode então escolher os atributos que pretende apresentar no min_dcc final de forma a não ter de apresentar todos os dados.

Depois de escolhidos os atributos é pedida a senha que foi usada para gerar as máscaras na aplicação inicial. São então geradas novamente as máscaras para os atributos que o owner pretende manter visiveis.

Listing 4.1: Gerar máscaras

```
"attributes": [
            {
                "label": atributo["label"],
                "value": atributo["value"],  # valor original do atributo
                "mask": encode_base64(gerar_mascara(atributo["label"], senha))
            }
            for atributo in dcc_final["attributes"]
            if atributo["label"] in atributos_visiveis
        ],
```

Os commitments de todos os atributos são adicionados a uma lista e colocados no JSON do min_dcc em conjuntos com os atributos visíveis e as máscaras destes atributos.

Listing 4.2: Adicionar commitments

```
"commitments": [
            attr['commitment']
            for attr in dcc_final['attributes']
        ],
```

De seguida estes dados são assinados utilizando a chave privada do cartão de cidadão então o cartão de cidadão físico ainda é necessário neste passo. É utilizado RSA com o algoritmo SHA512.

Listing 4.3: Assinar dados

```
# Gerar a assinatura do DCC
    dcc_filtrado_str = json.dumps(dcc_filtrado, sort_keys=True)
    print("Dados a assinar: ", dcc_filtrado_str)
    # assinatura = assinar_dcc(chave_privada_owner, dcc_filtrado_str)
    assinatura, mecanismo = assinar_com_chave_privada(dcc_filtrado_str)

    # Adicionar a assinatura ao DCC
    timestamp = time.time()
    dcc_filtrado["Owner_signature"] = {
        #bytes to string
        "signature_value": assinatura.hex(),
```

```
        "timestamp": timestamp,
        "crypto_system": str(mecanismo)
    }
```

Listing 4.4: Ler chave privada e assinar

```
def assinar_com_chave_privada(dados):
    try:
        # Caminho da biblioteca PKCS#11
        lib = '/usr/local/lib/libpteidpkcs11.so'
        pkcs11 = PyKCS11Lib()
        pkcs11.load(lib)
        slots = pkcs11.getSlotList()
        if not slots:
            print("Nenhum cartao encontrado.")
            exit()

        # Selecionar o primeiro slot
        slot = slots[0]
        session = pkcs11.openSession(slot, PyKCS11.CKF_SERIAL_SESSION | PyKCS11.
            CKF_RW_SESSION)

        # Localizar a chave privada no cartao
        priv_key_obj = None

        try:
            priv_key_obj = session.findObjects([(PyKCS11.CKA_CLASS, PyKCS11.
                CKO_PRIVATE_KEY), (PyKCS11.CKA_LABEL, "CITIZEN AUTHENTICATION KEY")
                ])[0]
            mechanism = PyKCS11.Mechanism(PyKCS11.CKM_SHA512_RSA_PKCS)
            assinatura = session.sign(priv_key_obj, dados, mechanism)
            assinatura_bytes = bytes(assinatura)  # Converter para bytes
            session.closeSession()
            print("Assinatura gerada com sucesso!")
            return assinatura_bytes, "CKM_SHA512_RSA_PKCS"
```

O min_dcc vai ter então uma assinatura do owner em cima de todos os dados com a chave do seu cartão. O min_dcc final vai ser guardado com o seguinte formato:

Listing 4.5: Min DCC

```
{
    "commitments":[
                "9bff8165f1abf93c99bec5dd17bcda5f9f8bb6e3cc8d6d05577a9b1a2279f9d9",
                "eb8efaed4cc544df68730bae65293ca74e8ba448cdd9b893fd94b96fb819e7a4",
                "114b6d2001154bb8268584ed54d21bf394465285302117c003a78709a4135dd5",
                "fee113691dcce3a8ee4a6f2368acc0c8a1d93dfea98abc845053d7cdd2dc137c"],
    "digest_function": {
        "type": "SHA-256"
    },
    "attributes": [
        {
            "label": "nome",
            "value": ("Nome", "Mask")
        },
        {
            "label": "morada",
            "value": ("Rua Romana", "Mask")
        }
    ]
    "chave_publica_owner": [
        {
```

```
            "value":"-----BEGIN PUBLIC KEY-----\
                nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAs8Bj62Wm2oPErsnH21zi\
                nz4iVZ1Sj5FyoT7bf8NhmBvOg52fTFYgks3/oXtYn5\n-----END PUBLIC KEY
                -----\n"
            "description of key nature": "some description of the nature of the
                asymmetric cryptosystem of this key"
        }
    ],
    "Issuer_signature": [
        {
            "value": "signature",
            "timestamp": timestamp,
            "description": " description of the asymmetric cryptosystem",
            "issuer_certificate": "issuer_certificate"
        }
    ],
    "Owner_signature": [
        {
            "value": "signature",
            "timestamp": timestamp,
            "description": " description of the asymmetric cryptosystem"
        }
    ]
}
```

Foi então gerado o Cartão de Cidadão digital com apenas os dados necessários.

Capítulo 5

check_dcc

5.0.1 Verificar integridade do DCC

Nesta aplicação é pedido ao utilizador um JSON de min_dcc para ser lido e verificado. É dado então ao utilizador a opção de mostrar os dados visíveis presentes no DCC.

Figura 5.1: Ver dados do min_dcc

O utilizador tem também a opção de verificar a integridade dos dados. Quando o utilizador seleciona esta opção, primeiro é lido certificado da entidade superior (issuer) que assinou os dados. É verificado utilizando os commitments e a chave pública do owner presentes no ficheiro se a assinatura está correto e se os dados estão íntegros.

Listing 5.1: Verificar assinatura do issuer

```
try:
        # Extrair dados da assinatura
        assinatura_data = resposta['Issuer_signature'][0]
        assinatura = bytes.fromhex(assinatura_data['value'])
        certificado_pem = assinatura_data['issuer_certificate']

        # Carregar o certificado
        certificado = load_pem_x509_certificate(certificado_pem.encode('utf-8'),
            default_backend())
        chave_publica = certificado.public_key()
        # Concatenar os valores dos commitments e a chave publica do owner
        compromissos_e_chave_publica = []
        for atributo in resposta["commitments"]:
            compromissos_e_chave_publica.append(atributo)

        compromissos_e_chave_publica.append(resposta["chave_publica_owner"][0]["
            value"])

        # Gerar os dados para assinar
        dados_para_assinar = json.dumps(compromissos_e_chave_publica, sort_keys=True
            )

        hash_dados = hashes.Hash(hashes.SHA1(), backend=default_backend())
        hash_dados.update(dados_para_assinar.encode())
        digest = hash_dados.finalize()

        # Verificar a assinatura
        chave_publica.verify(
            assinatura,
            digest,
            ec.ECDSA(hashes.SHA1())
        )
```

```
30
31          print("Assinatura do Issuer valida: os dados estao integros.")
32          return True
```

Depois é verificada a assinatura do Owner para garantir a integridade dos dados do cartão.

Listing 5.2: Verificar assinatura do owner

```
1  def validar_assinatura_owner(chave_publica, dados_para_validar, assinatura):
2      try:
3          chave_publica = serialization.load_pem_public_key(
4              chave_publica.encode('utf-8'),
5              backend=default_backend()
6          )
7
8          # Converter a assinatura de hexadecimal para bytes
9          assinatura_bytes = bytes.fromhex(assinatura)
10
11         dados_bytes = dados_para_validar.encode('utf-8')
12
13         # Verificar a assinatura utilizando a chave publica
14         chave_publica.verify(
15             assinatura_bytes,  # A assinatura a ser validada
16             dados_bytes,  # O hash dos dados originais
17             padding.PKCS1v15(),  # O esquema de padding
18             hashes.SHA512()  # O algoritmo de hash utilizado
19         )
20
21         print("Assinatura do Owner valida: os dados estao integros.")
22
23     except ValueError as e:
24         print(f"Erro ao verificar a assinatura: {e}")
25     except Exception as e:
26         print(f"Assinatura do Owner invalida ou documento adulterado: {e}")
```

Como são enviadas as máscaras dos atributos visíveis é possível calcular o commitment value desse atributo. Então depois de serem verificadas as assinaturas do owner e do issuer são então calculados os commitments values e é verificado se eles estão presentes na lista de commitments para garantir que os valores não foram alterados.

Listing 5.3: Verificar commitments

```
1  def verificarCompromisso(nome_atributo, valor_atributo, mascara, lista_compromissos)
       :
2      # Calcular o compromisso
3      compromisso_calculado = calcular_compromisso(nome_atributo, valor_atributo,
           mascara)
4
5      # Verificar se o compromisso calculado e igual ao compromisso fornecido
6      if compromisso_calculado in lista_compromissos:
7          print(f"O compromisso do atributo '{nome_atributo}' e valido.")
8      else:
9          print(f"O compromisso do atributo '{nome_atributo}' e invalido.")
```

Listing 5.4: Calcular commitments

```
1  # Funcao para calcular o compromisso
2  def calcular_compromisso(nome_atributo, valor_atributo, mascara):
3      compromisso = hashlib.sha256()
```

```
    compromisso.update(nome_atributo.encode())  # Adiciona o nome do atributo
    compromisso.update(valor_atributo.encode())  # Adiciona o valor do atributo
    compromisso.update(mascara)
    return compromisso.hexdigest()
```

É então apresentado ao utilizador os resultados:

```
Menu:
1. Ver Dados do DCC
2. Validar Integridade dos Dados
3. Sair
Escolha uma opção (1/2/3): 2
Assinatura do Issuer válida: os dados estão íntegros.
Assinatura do Owner válida: os dados estão íntegros.
Verificando compromisso do atributo 'nome', valor: 'DIOGO FILIPE ROSÁRIO DE ALMEIDA,' máscara: 'eFBPsKciKPTD0QodRpteg93tQypKjl3ArzrFiycEmv8='
O compromisso do atributo 'nome' é válido.
Verificando compromisso do atributo 'data_nascimento', valor: '2003-02-05 12:00:00+00:00,' máscara: 'HyatIHZgcH37nCUcYywcEEV3Q+7IzHigtvsTGNQzjK0='
O compromisso do atributo 'data_nascimento' é válido.
```

Figura 5.2: Resultados do check_dcc

Capítulo 6

Algoritmos e sistemas criptográficos

Forma usados os seguintes algoritmos e sistemas criptográficos:

- Digest Function: SHA256
- Criptosystem owner key pair e signature algorithm: RSA com SHA512
- Criptosystem issuer key pair e signature algorithm: Elliptic Curve Digital Signature Algorithm (ECDSA) com SHA1