Secure Game

Security of Information and Organizations Project changes from First Submission

Rafael Remígio 102435
Bruno Moura 97151
João Correia 104360

February 1, 2023

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro
Year 2022/2023

Contents

1 Changes From First Delivery **2**

1.1 Smart Card Authentication 2

1.2 AES Block Cipher Mode 3

1.3 Cross Language Interoperability 4

1 Changes From First Delivery

1.1 Smart Card Authentication

Our first delivery did not include authentication using the Portuguese citizen smart card due to technical issues. We have added it in now.
We've created a new class "SmartCardSession" for handling PKCS11 sessions. It can sign data using the private key and share it's public key in the form of a modulus and public exponent. We sign data using the SHA-256 hashing algorithm instead of SHA-1 (which was used in the practical classes) for increased robustness.
We used the PyKCS11 python module as a PKCS11 wrapper. It does not include a PKCS11 library, however, so we used OpenSC's to that end.

Signatures for SmartCardSession's methods in the smartcard_reader.py.
Furthermore, we expanded the "crypto.py" cryptographic utility class to include methods for recreating the RSA key object from the modulus and exponent as well as a method for verifying a digital signature from the smart card.

```
import PyKCS11

# PyKCS11 does not include a PKCS11 library. We'll use opensc
#lib = '/usr/lib/x86_64-linux-gnu/pkcs11/opensc-pkcs11.so' # Mint location
    using apt install opensc-pkcs11
lib = '/usr/lib64/pkcs11/opensc-pkcs11.so' # Fedora location using dnf
    install opensc

class SmartCardSession():
    """Smart Card Session Utilities"""

    # static method for creating session object
    @classmethod
    def create(cls, pin : str) -> None:
        """Generates a PyKCS11 session, required Citizen Card PIN code"""
      ...

    def getPublicKey(self) -> tuple[bytes, bytes]:
        """Returns the modulus and pubExponent corresponding to the smart
            card's public key as bytes"""
        ...

    def sign(self,message: bytes) -> bytes:
        """Signs a message and returns the signature"""
      ...

    def close(self) -> None:
        """Closes the PyKCS11 session"""
        ...
```

The only other difference is that “Player.py” and “Caller.py” need an additional PIN parameter from creating the smart card session.

1.2 AES Block Cipher Mode

When faced with the choice of what block cipher mode of operation we should use with AES for symmetric encryption, we originally chose Galois Counter Mode (GCM).
Our reasoning for GCM to be the better choice was that it offered more features than the other modes, such as automatic integrity checking, higher tampering resistance, capability for pre-processing and parallel processing.
This conclusion proved to be foolhardy, as GCM was completely overkill for the use case of deck encryption, as pointed out by Professor Zúquete. It also added unnecessary complexity in the form of the IVs required for cipher operations.
As it turns out, the simpler Electronic Code Book (ECB) is a much better fit to our problem. It doesn’t boast as many qualities as GCM, but it’s good enough for our use case and doesn’t have unnecessary overhead for features we won’t use. Since decks don’t have repeating numbers, pattern propagation is not an issue either.
We learned a valuable lesson about the importance of carefully considering the requirements and constraints of a system to determine what technological solution best satisfies those needs, as opposed to instinctively going for the theoretically “best” option. Although GCM was supposedly the better choice, it’s advantages over ECB were superfluous to the needs of the system while keeping the complexity associated with them.

Previously, with GCM. Note that the same IV is used for every message, compromising robustness.

```
        """Decrypts encrypted data given with given AESGCM key"""

        crypted_data=base64.b64decode(crypted_data.encode('ascii'))
        key=base64.b64decode(key.encode('ascii'))
        cypher = AESGCM(key)
        data = cypher.decrypt(nonce, crypted_data, None)

        return data.decode('ascii')
```

Currently, with ECB.

```
def sym_encrypt(cls, key: bytes, data) -> bytes:
        """Encrypts data given with given AESGCM key"""

        key=base64.b64decode(key.encode('ascii'))
        data=bytes(str(data), 'ascii')

        cipher = Cipher(algorithms.AES(key), modes.ECB())
```

```
    # Set up padder
    padder = sym_padding.PKCS7(128).padder()

    # Add padding to the bites
    padded_data = padder.update(data) + padder.finalize()

    # create encryptor
    encryptor = cipher.encryptor()

    # encrypt the message
    ct = encryptor.update(padded_data) + encryptor.finalize()

    return base64.b64encode(ct).decode('ascii')
```

1.3 Cross Language Interoperability

Our system features a novel method for card generation, in which every player shuffles the playing deck as they sign it with their symmetric key.
Once the generation process has been completed and everyone knows the clear deck, users unshuffle the deck in the reverse order it was shuffled in to determine the card of every player. The first N numbers of the deck as shuffled by a given player is their playing card, where N is the card length.
More information about our method can be found at our previous report.
It was vital that whenever a user shuffles the deck, it is done deterministically as to allow the others users to unshuffle it. To achieve that, we used a method that deterministically shuffles and unshuffles a list given a seed. The seed is the player's deck encryption key, as it is a secret up until the point everyone shares theirs at the end of the deck generation.
Original shuffling methods. Note the use of the random module.

```
# https://crypto.stackexchange.com/q/78309
def deterministic_shuffle(self, ls, seed : str):
    """Deterministically shuffles a list given a seed"""
    random.seed(seed)
    random.shuffle(ls)
    return ls
# https://crypto.stackexchange.com/q/78309
def deterministic_unshuffle(self, shuffled_ls, seed : str):
    n = len(shuffled_ls)
    # perm is [1, 2, ..., n]
    perm = [i for i in range(1, n + 1)]
    # apply sigma to perm
    shuffled_perm = self.deterministic_shuffle(perm, seed)
    # zip and unshuffle
    zipped_ls = list(zip(shuffled_ls, shuffled_perm))
    zipped_ls.sort(key=lambda x: x[1])
```

```
        return [a for (a, b) in zipped_ls]
```

One issue we had not considered raised by Professor Zúquete is that our methods rely on Python's module for random number generation. This means that a bingo client written in a programming language other than Python could not play with our Python client as it would shuffle the deck differently, even though they both follow the same communication protocol.
To fix this, we refactored those functions to use a cross-language method for random number generation. The solution we came up with is to hash the seed using MD5 and treating each of the sixteen bytes of the digest as a pool of random integers to be used by the Fisher-Yates list shuffling algorithm. If we run out of numbers in the pool, we generate it again using a different nonce.

Shuffling using MD5. The unshuffling method remains unaltered.

```
 # Fisher-Yates Algorithm https://favtutor.com/blogs/shuffle-list-python
    @classmethod
    def deterministic_shuffle(cls, ls : list, seed : str):
        """Deterministically shuffles a list given a seed using the Fisher-
            Yates Algorithm"""

        rng = [] # array of 16 random numbers as generated from a MD5 hash
        nonce = 0 # nonce for generating the hash in case the original rng
            pool runs out
        for i in range(len(ls)-1, 0, -1):

            # if the random number pool is empty, generate more numbers
            if rng == []:
                digest = hashlib.md5((seed+str(nonce)).encode()).digest() #
                    MD5 hash of the seed + the nonce as a source of
                    uniqueness
                rng = [x for x in digest] # convert bytes to array of ints
                nonce += 1 # increase the nonce so that the next digest is
                    different

            # selects a random index using the random number
            j = rng.pop() % i
            ls[i],ls[j] = ls[j], ls[i] # swaps two items

        return ls

    # https://crypto.stackexchange.com/q/78309
    @classmethod
    def deterministic_unshuffle(cls, shuffled_ls : list, seed : str):
        """Reverses the deterministic shuffle and returns the original list
            """
        n = len(shuffled_ls)
```

```
# perm is [1, 2, ..., n]
perm = [i for i in range(1, n + 1)]
# apply sigma to perm
shuffled_perm = cls.deterministic_shuffle(perm, seed)
# zip and unshuffle
zipped_ls = list(zip(shuffled_ls, shuffled_perm))
zipped_ls.sort(key=lambda x: x[1])
return [a for (a, b) in zipped_ls]
```