<u>Project description</u>:
This project uses JavaCard to implement a Credit Card on a Smart Card. The smart card controls access to the data via encryption and authentication which the host application can access in a secure way.

- Installation script: ant
- Application
    - PIN confirmation
        - Security question on third false input
        - Set PIN during installation with installation script
    - Create private/public key pair
    - Interfaces
        - Get public key
        - Accept public key for encryption (after PIN confirmation)
        - Set session key
        - Get details (card holder name, card number, card expiry date, security code)
    - Confidentiality: encrypt with public key
    - Authenticity: add hash code before encryption

This application is based on a smart card client/host. The smart card holds confidential credit card information such as:

Cardholder name
Account number
Card security code
Card expiry date
Card pin number

The host can become authorised to access information on the card after entering the correct pin within three attempts. Using RMI an authorised host has the ability to: change the pin and retrieve the confidential information and secret message.
If the requests/replies are not encrypted or authenticated, the card blocks the access to the information.

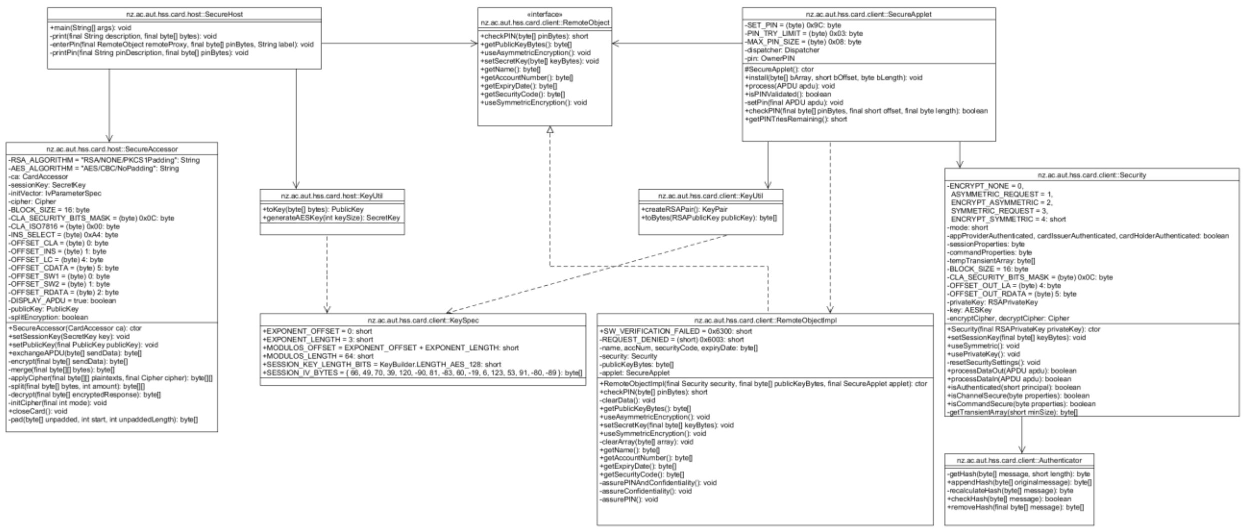
<u>Security features:</u>

- Confidentiality: This application uses asymmetric and symmetric encryption/decryption to ensure attackers cannot read messages transmitted between the host and client. The smart card client generates a RSA key pair and sends the public key from this key pair to the host. The host uses the public key to encrypt the PIN when attempting to gain authorisation. Once authorised, the host can send the session key (encrypted with the client's public key) - generated using the client's public key. Once the client receives the session key, they can use it to encrypt information sent to the host (rather than encrypting with the public key)
- Authentication: The hash codes we were going to use and append to the message risked memory exhaustion due to limited memory on the emulator. Instead of using usual hash codes, we used an XOR hash. This involved taking the first byte of a message and XORing this with the second byte. The byte resulted from this was XORed with the next byte and so on. When no more bytes in the message need to be XORed, the final byte is appended to the message before it gets encrypted and sent.
  Upon receiving a message, the message is decrypted and the XOR method is applied to the message (excluding the last byte). The result from the XOR method is compared to the final byte. If they match than the message can be regarded as authentic.

<u>How to counter attacks:</u>

- Tampering attacks – The smart card uses an XOR hash code to verify whether the message has been altered or not, as the hash code is likely to change after altering a message.
- Brute force attacks - The smart card clears its details and denies the host access after three incorrect attempts of entering the pin.
- The application also counters attacks in which an attacker could read the PIN (or any other confidential information) as it is transmitted, by using encryption to make all messages confidential. The only plaintext message is the public key of the smart card

<u>Class diagram</u> (image version in *doc/class-diagram.png*):



nz.ac.aut.hss.card.host::SecureHost
+main(String[] args): void
-print(final String description, final byte[] bytes): void
-enterPin(final RemoteObject remoteProxy, final byte[] pinBytes, String label): void
-printPin(final String pinDescription, final byte[] pinBytes): void
«interface»
nz.ac.aut.hss.card.client::RemoteObject
+checkPIN(byte[] pinBytes): short
+getPublicKeyBytes(): byte[]
+useAsymmetricEncryption(): void
+setSecretKey(byte[] keyBytes): void
+getName(): byte[]
+getAccountNumber(): byte[]
+getExpiryDate(): byte[]
+getSecurityCode(): byte[]
+useSymmetricEncryption(): void
nz.ac.aut.hss.card.client::SecureApplet
-SET_PIN = (byte) 0x9C: byte
-PIN_TRY_LIMIT = (byte) 0x03: byte
-MAX_PIN_SIZE = (byte) 0x08: byte
-dispatcher: Dispatcher
-pin: OwnerPIN
#SecureApplet(): ctor
+install(byte[] bArray, short bOffset, byte bLength): void
+process(APDU apdu): void
+isPINValidated(): boolean
-setPin(final APDU apdu): void
+checkPIN(final byte[] pinBytes, final short offset, final byte length): boolean
+getPINTriesRemaining(): short
nz.ac.aut.hss.card.host::SecureAccessor
-RSA_ALGORITHM = "RSA/NONE/PKCS1Padding": String
-AES_ALGORITHM = "AES/CBC/NoPadding": String
-ca: CardAccessor
-sessionKey: SecretKey
-initVector: IvParameterSpec
-cipher: Cipher
-BLOCK_SIZE = 16: byte
-CLA_SECURITY_BITS_MASK = (byte) 0x0C: byte
-CLA_ISO7816 = (byte) 0x00: byte
-INS_SELECT = (byte) 0xA4: byte
-OFFSET_CLA = (byte) 0: byte
-OFFSET_INS = (byte) 1: byte
-OFFSET_LC = (byte) 4: byte
-OFFSET_CDATA = (byte) 5: byte
-OFFSET_SW1 = (byte) 0: byte
-OFFSET_SW2 = (byte) 1: byte
-OFFSET_RDATA = (byte) 2: byte
-DISPLAY_APDU = true: boolean
-publicKey: PublicKey
-splitEncryption: boolean
+SecureAccessor(CardAccessor ca): ctor
+setSessionKey(SecretKey key): void
+setPublicKey(final PublicKey publicKey): void
+exchangeAPDU(byte[] sendData): byte[]
-encrypt(final byte[] sendData): byte[]
-merge(final byte[][] bytes): byte[]
-applyCipher(final byte[][] plaintexts, final Cipher cipher): byte[][]
-split(final byte[] bytes, int amount): byte[][]
-decrypt(final byte[] encryptedResponse): byte[]
-initCipher(final int mode): void
+closeCard(): void
-pad(byte[] unpadded, int start, int unpaddedLength): byte[]
nz.ac.aut.hss.card.host::KeyUtil
+toKey(byte[] bytes): PublicKey
+generateAESKey(int keySize): SecretKey
nz.ac.aut.hss.card.client::KeyUtil
+createRSAPair(): KeyPair
+toBytes(RSAPublicKey publicKey): byte[]
nz.ac.aut.hss.card.client::Security
-ENCRYPT_NONE = 0,
ASYMMETRIC_REQUEST = 1,
ENCRYPT_ASYMMETRIC = 2,
SYMMETRIC_REQUEST = 3,
ENCRYPT_SYMMETRIC = 4: short
-mode: short
-appProviderAuthenticated, cardIssuerAuthenticated, cardHolderAuthenticated: boolean
-sessionProperties: byte
-commandProperties: byte
-tempTransientArray: byte[]
-BLOCK_SIZE = 16: byte
-CLA_SECURITY_BITS_MASK = (byte) 0x0C: byte
-OFFSET_OUT_LA = (byte) 4: byte
-OFFSET_OUT_RDATA = (byte) 5: byte
-privateKey: RSAPrivateKey
-key: AESKey
-encryptCipher, decryptCipher: Cipher
+Security(final RSAPrivateKey privateKey): ctor
+setSessionKey(final byte[] keyBytes): void
+useSymmetric(): void
+usePrivateKey(): void
-resetSecuritySettings(): void
+processDataOut(APDU apdu): boolean
+processDataIn(APDU apdu): boolean
+isAuthenticated(short principal): boolean
+isChannelSecure(byte properties): boolean
+isCommandSecure(byte properties): boolean
-getTransientArray(short minSize): byte[]
nz.ac.aut.hss.card.client::KeySpec
+EXPONENT_OFFSET = 0: short
+EXPONENT_LENGTH = 3: short
+MODULOS_OFFSET = EXPONENT_OFFSET + EXPONENT_LENGTH: short
+MODULOS_LENGTH = 64: short
+SESSION_KEY_LENGTH_BITS = KeyBuilder.LENGTH_AES_128: short
+SESSION_IV_BYTES = { 66, 49, 70, 39, 120, -90, 81, -83, 60, -19, 6, 123, 53, 91, -80, -89 }: byte[]
nz.ac.aut.hss.card.client::RemoteObjectImpl
+SW_VERIFICATION_FAILED = 0x6300: short
-REQUEST_DENIED = (short) 0x6003: short
-name, accNum, securityCode, expiryDate: byte[]
-security: Security
-publicKeyBytes: byte[]
-applet: SecureApplet
+RemoteObjectImpl(final Security security, final byte[] publicKeyBytes, final SecureApplet applet): ctor
+checkPIN(byte[] pinBytes): short
-clearData(): void
+getPublicKeyBytes(): byte[]
+useAsymmetricEncryption(): void
+setSecretKey(final byte[] keyBytes): void
+useSymmetricEncryption(): void
-clearArray(byte[] array): void
+getName(): byte[]
+getAccountNumber(): byte[]
+getExpiryDate(): byte[]
+getSecurityCode(): byte[]
-assurePINAndConfidentiality(): void
-assureConfidentiality(): void
-assurePIN(): void
nz.ac.aut.hss.card.client::Authenticator
-getHash(byte[] message, short length): byte
+appendHash(byte[] originalmessage): byte[]
-recalculateHash(byte[] message): byte
+checkHash(byte[] message): boolean
+removeHash(final byte[] message): byte[]