# Проектна задача по предметот Информациска безбедност
# Сервер за валидација

Ментори:

Проф. Димитрова Весна
Проф. Михајлоска Христина

Изработиле:
Ана Гаштарова 186001
Андреа Ристевска 186018
Маја Крстевска 186037
Лина Гроздановска 181029

## Вовед

Проектната задача е имплементирана во Java и се состои од клиентска и серверска апликација. Клиентската апликација како влезни параметри, преку аргументи при нејзиното повикување, ги добива јавниот клуч (public key), потписот (signature) и името на датотеката која е потпишана. Потоа, клиентската апликација ги испраќа до серверската апликација податоците за валидација на потписот. Комуникацијата помеѓу клиентската и серверската апликација се одвива преку HTTPS протоколот.

Откако ќе се стартува, серверската апликација чека барање за валидација. Барањето го испраќа клиентската апликација и во него се содржи јавниот клуч, потписот и потпишаната датотеката.

## Потпишување

За креирање на приватниот и јавниот клуч, потписот и потпишувањето на дадотеката се користи OpenSSL (www.openssl.org) која е робустна криптографска алатка за широка употреба. За оваа проектна задача одбравме да се користи RSA-4096 асиметричен private/public алгоритам за енкрипција. Избрантиот алгоритам за енкрипција е помеѓу најбезбедните и најкористените алгоритми денес.

Најпрво се креира приватниот клуч според одбраниот алгоритам за енкрипција (RSA-4096) и потоа се конвертира во PKCS8 енкодирање кое одлучивме да го користиме во проектната задача:

```
openssl genrsa -out privatekeyrsa4096pkcs1.pem 4096
openssl pkcs8 -topk8 -in privatekeyrsa4096pkcs1.pem -inform pem -out
privatekeyrsa4096pkcs8-exported.pem -outform pem -nocrypt
```

Првата команда го генерира приватниот клуч, додека втората команда на OpenSLL прави конверзија на истиот клуч од PKCS1 во PKCS8 енкодинг. Приватниот клуч има форма и содржина слична на следниот пример:

-----BEGIN PRIVATE KEY-----
MIIJQgIBADANBgkqhkiG9w0BAQEFAASCCSwwggkoAgEAAoICAQDVgLrCSDC5mLRL
JY+okYX5MOMGi+bvtRQ9qIQ90d3BO1gAao6ZsbPEFxnOTR9Q3bGsEE5oRlh/FSYS
.
.
kvCjd0ineNZ6OgPVJ/mhPULsZb11+noSUPmFqvClb8SQ0BipbKIcSTIJlQt1ZRZ2
INdXsP5kNlRK181jtU/xtQYfwSjkKA==
-----END PRIVATE KEY-----

Потоа од приватниот клуч се креира јавниот клуч кој треба да биде X509 енкодиран за да може да се користи од Java апликацијата за валидација.

openssl rsa -pubout -outform pem -in privatekeyrsa4096pkcs8-generated.pem -out publickeyrsa4096pkcs8.pem

Јавниот клуч, откако ќе биде изгенериран со користење на  OpenSSL и приватниот клуч, ќе има форма и содржина слична на следниот пример:

-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA1YC6wkgwuZi0SyWPqJGF
+TDjBovm77UUPaiEPdHdwTtYAGqOmbGzxBcZzk0fUN2xrBBOaEZYfxUmEkOFzPbF
.
.
oNta8CSsVrqgFW/tI6+MQwrQFEOcBPCbh6Pr7NbiuR2LrfoJhUJlD5ofz5eM0419
JSS0RvKh0dF3ddlOKV/TQUsCAwEAAQ==
-----END PUBLIC KEY-----

Откако ќе ги имаме приватниот и јавниот клуч можеме да ја потпишеме датотеката користејќи ја повторно OpenSSL алатката.

openssl dgst -sha256 -sign privatekeyrsa4096pkcs8-generated.pem -out signature.sha256 test.txt
openssl base64 -in signature.sha256 -out signaturebase64

Првата команда ја потпишува датотеката test.txt користејќи го приватниот клуч privatekeyrsa4096pkcs8-generated.pem и креира потпис (signature) во бинарен SHA256 формат. За да можеме да го праќаме потписот преку интернет потребно е да го конвертираме во Base64 формат кој е текстуален и погоден за трансфер преку HTTPS. Тоа го правиме со втората команда, која од signature.sha256 креира signaturebase64 датотека.

Користејќи OpensSLL можеме да направиме и проверка на потписот и потпишаната датотека за да сме сигурни дека целата постака за генерирање на клучевите и потпишувањето е исправно направена. Тоа го правиме со следната команда:

openssl dgst -sha256 -verify publickeyrsa4096pkcs8.pem -signature signature.sha256 test.txt

За оваа валдиација се користи јавниот клуч, потписот како и потпишаната датотека. OpenSLL како резултат од извршената команда за валидација може да даде одговор Verified Ok доколку проверката е исправна, или Verified Failure доколку проверката не е исправна.

Откако ќе потврдиме дека клучевите, потписот и потпишаната датотека се исправно генерирани со користење на OpenSSL можеме да продолжиме со нивно испраќање преку нашата клиентска програма до нашиот сервер за валидација.

## Сервер за HTTPS комуникација во Java

Имплементацијата на проектната задача се состои од серверска апликација и клиентска апликација кои комуницираат преку HTTPS протоколот. Пред започенеме со пишување на двете апликации потребно е да генерираме keystore и truststore кои Јава ги користи за зачувување на сертификатите потребни за SSL заштитена комуникација. За оваа проектна задача го користиме keystore-от кој е даден во примерите на JDK (Java Development Kit) и истиот ќе го употребиме како keystore и како truststore. Во него се наоѓа и сертификат за localhost (127.0.0.1) кој ќе ни овомозжи да го користиме локалниот компјутер за развој и проверка на серверската и клиентската апликација.

Серверската апликација користи SSLContext, SSLServerSocket и SSLSocket објекти од Java за да овозможи HTTPS комуникација. Најпрво се креира SSLContext од кого се добива SSLSocketFactory, а од SSLSocketFactory се креира SSLSocekt и се поставува да слуша барања за валидација на одреден комуникациски порт (во нашата задача, тоа е портот 9999). Како што е прикажано на следниот листинг:

```
    SSLContext sslContext = this.createSSLContext();

     try{
        SSLServerSocketFactory sslServerSocketFactory = sslContext.getServerSocketFactory();

        SSLServerSocket sslServerSocket = (SSLServerSocket)
sslServerSocketFactory.createServerSocket(this.port);

        System.out.println("Signature Validation server started over SSL");

        while(!isServerDone){
           SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
           new ServerThread(sslSocket).start();
        }
     } catch (Exception ex){
        ex.printStackTrace();
     }
```

Креирањето на SSLContext и неговото иницијализирање се состои од вчитување и иницијализирање на дадениот KeyStore кој ги содржи сертификатите потребни за воспоставување на SSL конекција. Потребно е дадениот keystore (testkeys) да биде зачуван

на истата локација со програмата, како и да биде снимен на локацијата каде е инсталиран Java Development Kit (JDK) во папката /lib/security/ и да ја замени постоечката cacerts датотека.

Потоа, се креира т.н. KeyManager објект со користење на иницијализираниот KeyStore и пасвордот за тој KeyStore (во нашиот пример пасвордот е passphrase). Практичната задача користи ист keystore и truststore во Јава.

На крај, се иницијализира SSLContext објектот за Transport Layer Security (TLS) верзија 1.3 со користење на веќе иницијализираните KeyStore и TrustManager објекти.

Креирањето на SSLContext е дадено на следниот листинг:

```
try{
        //Vcituvanje i instanciranje na keyStore od dadeniot file
        KeyStore keyStore = KeyStore.getInstance("JKS");
        keyStore.load(new FileInputStream("testkeys"),"passphrase".toCharArray());

        // Kreiranje na KeyManager
        KeyManagerFactory keyManagerFactory =
KeyManagerFactory.getInstance("SunX509");
        keyManagerFactory.init(keyStore, "passphrase".toCharArray());
        KeyManager[] km = keyManagerFactory.getKeyManagers();

        //kreiranje na trust manager
        TrustManagerFactory trustManagerFactory =
TrustManagerFactory.getInstance("SunX509");
        trustManagerFactory.init(keyStore);
        TrustManager[] tm = trustManagerFactory.getTrustManagers();

        //Inicijalizacija na SSLContext objektot za TLSv1
        SSLContext sslContext = SSLContext.getInstance("TLSv1");
        sslContext.init(km,  tm, null);

        return sslContext;
    } catch (Exception ex){
        ex.printStackTrace();
    }
```

Серверот покренува посебен Thread за секој нов клиент кој ќе се обрати за комуникација. Со тоа се овозможува паралелно опслужување на повеќе клиентски апликации. Секој од тредовите независно ја води комуникацијата со својот клиент. Комуникацијата почнува со т.н. handshake и воспоставување на сесија. Исто така, се креираат и Stream објекти за размена на податоци со клиентот, еден InputStream и еден OutputStream преку кои се добиваат и испраќаат податоци од/до клиентот.

Во комуникацијата со клиентот, серверот добива три пораки кои се еден текстуален ред енкодиран во Base64 формат кој е соодветен за транспорт преку HTTP и HTTPS протоколите. Првата порака е јавниот клуч, втората порака е потписот додека третата порака е потпишаната датотека. За крај на комуникацијата клиентот испраќа празна порака.

Следниот листинг го прикажува примањето на овие пораки:

```
while((line = bufferedReader.readLine()) != null){

        if(line.isEmpty()){
           break;
        }
        else
        {
           if (count == 0)
           {
              //se prima soodrzinata na javniot kluc i se dekodira od Base64 format
              //se kreira javniot kluc od dobienata soodrzina
              count++;

              byte[] decoded = Base64
              .getDecoder()
              .decode(line);

              X509EncodedKeySpec pubKeySpec = new X509EncodedKeySpec(decoded);

              KeyFactory keyFactory = KeyFactory.getInstance("RSA");
              pubKey = keyFactory.generatePublic(pubKeySpec);
           }
           else
              if (count == 1)
              {
                 // se prima sodrzinata na potpisot
                 // se kreira insnca na Signature objektot spored koristeniot algoritam
                 // se dekodira sodrzinata na potpisot od Base64 format
                 count++;

                 sig = Signature.getInstance("SHA256withRSA");
                 sig.initVerify(pubKey);
                 sigToVerify = Base64.getDecoder().decode(line);
              }
              else
                 if (count == 2)
                 {
                    // se prima sodrzinata na datotekata koja e potpisana
                    // se dekodira od Base64 i se dodava na Signature objektot
```

```
                    // potoa se pravi verifikacijata i se pecati rezultatot na ekran
                    count = 0;

                    byte[] data = Base64.getDecoder().decode(line);
                    sig.update(data, 0, data.length);

                    verifies = sig.verify(sigToVerify);
                    System.out.println("signature verifies: " + verifies);
                }
            }
        }
```

Јавниот клуч се пренесува во Base64 текстуален формат кој после се декодира и од декодираниот податок се креира т.н. X509EncodedKeySpec објект од кој после тоа се креира јавниот клуч како PublicKey објект.

Со добивањето на втората порака, се креира т.н. Signature објект кој за нашата задача треба да подржува SHA256withRSA тип на потписи. Добиениот потпис се декодира од Base64 форматот во кој беше испратен и се зачувува за верификација на потпишаната датотека.

Потпишаната датотека исто така се испраќа во Base64 формат и веднаш се декодира. Декодираната датотека, која сега е во оригиналиот формат, се проследува на веќе иницијализираниот Signature објект и заедно со декодираниот потпис се врши валидација.

Овие операции на прием и обработка на примените податоци, како и верификацијата на потписот се прикажани на претходниот код.

За крај, серверот одговара со „HTTP/1.1 200“ порака и со резултатот од валидацијата на потпишаната датотека.

## Клиент за HTTPS комуникација во Java

Клиентската апликација при стартување прима три параметри: името на датотеката со јавниот клуч, името на датотеката со потписот во Base64 формат и името на потпишаната датотека.

На почеток, слично како и кај серверската апликација, се креираат SSLContext и SSLSocket објекти. Креирањето и иницијализацијата на SSLContext објектот е исто како и кај серверската апликација, прикажано погоре, со користење на KeyStore и TrustStore објектите.

Откако ќе се иницијализираат овие објекти, клиентската апликација покренува посебен Thread во кој се одвива комуникацијата со серверот. Комуникацијата почнува со т.н. handshake со серверката апликација и креирање на комуникациска сесија. Потоа се креираат Stream објектите кои се користат за размена на податоци. Исто како и серверската

апликација, и клиентската апликација користи InputStream и OutputStream објекти за двонасочна комуникација со серверот.

Следниот код ја прикажува имплементацијата на комуникацијата со серверот, испраќањето на пораките и примањето на одговорот од серверот:

```
sslSocket.setEnabledCipherSuites(sslSocket.getSupportedCipherSuites());

        try{
            //Pocetok na komunikacijata so rakuvanje
            sslSocket.startHandshake();

            //Vospostavuvanje na sesija i komunikacija so serverot
            SSLSession sslSession = sslSocket.getSession();

            //Pecatenje na infromacija za konekcijata so serverot
            System.out.println("SSLSession :");
            System.out.println("\tProtocol : "+sslSession.getProtocol());
            System.out.println("\tCipher suite : "+sslSession.getCipherSuite());

            //Streamovi za komunikacija so serverot
            InputStream inputStream = sslSocket.getInputStream();
            OutputStream outputStream = sslSocket.getOutputStream();

            BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(inputStream));
            PrintWriter printWriter = new PrintWriter(new OutputStreamWriter(outputStream));

            //Podgotovka na podatocite i nivno isprakanje do serverot
            printWriter.println(ReadPublicKey());
            printWriter.println(ReadSignature());
            printWriter.println(ReadFile());
            printWriter.println("");
            printWriter.flush();

            //Cekanje na odgovor od serverot i negovo ispisuvanje na ekranot
            String line = null;
            while((line = bufferedReader.readLine()) != null){
                System.out.println("Input : "+line);

                if(line.trim().equals("HTTP/1.1 200\r\n")){
                    break;
                }
            }

            sslSocket.close();
```

```
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Клиентската апликација потребно е да ги подготви пораките во соодветен формат пред да ги испрати до серверската апликација. Ова е потребно за да може правилно да се пренесат преку HTTPS протоколот, односно мораат да бидат во Base64 формат.

Најпрво се вчитува јавниот клуч од датотеката која е внесена како прв аргумент при стартувањето на клиентската апликација. Јавниот клуч при неговото генерирање е во X509 формат кој е Base64 и може да се пренесува преку интернет. Но, OpenSSL во јавнит клуч запишува и информација за почеток и крај на клучот која Java не ја очекува. Затоа е потребно да се отстанат првата и последната линија од текстуалиот формат на јавниот клуч, како и целата содржина на јавниот клуч да се зачува во еден String од еден текстуален ред. Подготвувањето на јавниот клуч пред да се проследи до серверот е дадено со следниот код:

```
try {
    FileInputStream keyfis = new FileInputStream(strPK);
    byte[] encKey = new byte[keyfis.available()];
    keyfis.read(encKey);

    keyfis.close();

    String stringBefore = new String(encKey);

    //cistenje na soodrzinata na javniot kluc
    String stringAfter = stringBefore
        .replace("-----BEGIN PUBLIC KEY-----","")
        .replace("-----END PUBLIC KEY-----","")
        .replaceAll("\\s", "")
        .trim();

    return stringAfter;
    }
    catch (Exception e) {
        System.err.println("Caught exception " + e.toString());
    };
```

Следната порака која се испраќа до серверот е потписот. При неговото генерирање со OpenSSL потписот е конвертиран во Base64 формат што значи дека е веќе спремен за испраќање, но повторно е потребно да се промени и да биде во еден String без нови редови. Подготовката на пораката со потписот е дадена со следниот код:

```
try {
    FileInputStream keyfis = new FileInputStream(strSign);
```

```
byte[] encKey = new byte[keyfis.available()];
keyfis.read(encKey);

keyfis.close();

String stringBefore = new String(encKey);

//cistenje na soodrzinata na potpisot
String stringAfter = stringBefore
   .replaceAll("\\s", "")
   .trim();

return stringAfter;
}
catch (Exception e) {
   System.err.println("Caught exception " + e.toString());
};
```

Третата порака која се испраќа до серверот ја содржи потпишаната датотека. Бидејќи таа датотека може да биде во различен формат потребно е да ја конвертираме во Base64 и да ја испратиме до серверот, кој потоа ќе ја декодира во нејзниот оригинален формат. Подготовката на пораката која ја содржи потпишаната датотека е прикажана на следниот код:

```
try {
    FileInputStream keyfis = new FileInputStream(strFile);
    byte[] encKey = new byte[keyfis.available()];
    keyfis.read(encKey);

    keyfis.close();

    //encodiranje vo Base64 za da moze da se isprati preku HTTPS
    String stringAfter = Base64
    .getEncoder()
    .encodeToString(encKey);

    return stringAfter;
    }
    catch (Exception e) {
       System.err.println("Caught exception " + e.toString());
    };
```

Откако ќе бидат испратени трите пораки до серверската апликација, клиентската апликација испраќа празна порака со која означува дека ги испратила сите пораки. Потоа, го чека одговорот од серверската апликација и го прикажува на екран.

## Референци

Во изработката на оваа практична задача ги користевме официјалните упатства и примери од Oracle кои се дел од Java Development Kit.

За програмирање на HTTPS комуникацијата помеѓу клиент и сервер апликации ни послужи упатството дадено на следниот линк: https://docs.oracle.com/javase/10/security/sample-code-illustrating-secure-socket-connection-client-and-server.htm#JSSEC-GUID-A4D59ABB-62AF-4FC0-900E-A795FDC84E41

Додека, за валидација на потписот и програмирањето на делот од апликациите за работа со клучеви и потписи ни послужи следното упатство:
https://docs.oracle.com/javase/tutorial/security/apisign/index.html

За работа со OpenSSL ги користевме примерите и упатствата дадени на официјалната страница на OpenSSL www.openssl.org како и разни примери за употреба кои се достапни на интернет.