kpsp: Schlussbericht hybride Kryptographie

Marianne Schoch, Pascal Schwarz

4. Juni 2013

1 Abstract

Dieser Teil wird im finalen Bericht enthalten sein.

2 Einleitung / Idee

Die Software erlaubt die Erzeugung von verschlüsselten und signierten Nachrichten sowie deren Entschlüsselung. Die Nachrichten werden in Dateien gespeichert resp. aus Dateien gelesen.

Die Nachricht selbst wird mit einem symmetrischen Verfahren unter Verwendung eines zufälligen Schlüssels verschlüsselt. Dieser Schlüssel wird dem Empfänger der Nachricht ebenfalls übermittelt. Dazu wird der public Key des asymmetrischen Verschlüsselungsverfahrens verwendet.

Um die Integrität der Nachricht sicherstellen zu können wird des Weiteren eine Signatur, wiederum mit Hilfe des asymmetrischen Verschlüsselungsverfahrens und einer zusätzlichen Hashfunktion, der Nachricht beigefügt.

Zusätzlich ermöglichen wir die Erzeugung von Schlüsselpaaren für das asymmetrische Verschlüsselungsverfahren.

3 Theorie

3.1 Nachrichtenformat

3.1.1 Überblick

Die zu übermittelnde Datei besteht aus drei Teilen, die in den nachfolgenden Abschnitten beschrieben werden. Die Teile werden dabei in der folgenden Art markiert:

```
-----BEGIN <Abschnittname> [Option]-----
<Inhalt des Abschnittes, Base64 kodiert>
-----END <Abschnittname>-----
```

3.1.2 Teil KEYCRYPTED

Enthält den zufälligen „Sitzungsschlüssel“, der für das symmetrische Verschlüsselungsverfahren verwendet wird. Als Option wird das verwendete asymmetrische Verschlüsselungsverfahren angegeben. Die Abschnittsmarkierung kann dann beispielsweise folgendermassen aussehen:

```
-----BEGIN KEYCRYPTED RSA-----
<Base64 kodiertes Resultat der RSA-Verschluesselung des Sitzungsschluessels>
-----END KEYCRYPTED-----
```

Die Anwendung von RSA auf den Sitzungsschlüssel wird in Gruppen von 6 Bytes vorgenommen. Dabei werden die Bytes konkateniert und als Zahl intepretiert.

3.1.3 Teil MSGCRYPTED

Für die Verschlüsselung der eigentlichen Nachricht kommt ein symmetrisches Verfahren zum Einsatz. Falls es sich dabei um eine Blockchiffre handelt, wird neben dem Namen des Algorithmus ebenfalls angegeben, in welchem Modus die Blöcke verkettet werden. Falls ein Initialisierungsvektor benötigt wird, wird dieser zufällig erzeugt und in diesem Teil der Nachricht, mit einem „,“ vom Ciphertext separiert, abgelegt.

Kommt AES mit CBC als Modus zum Einsatz, sieht der Abschnitt folgendermassen aus:

```
-----BEGIN MSGCRYPTED AES256 CBC-----
<Base64 kodierter IV>,<Base64 kodiertes Resultat der Verschluesselung>
-----END MSGCRYPTED-----
```

3.1.4 Teil SIGNATURE

Die Signatur wird erzeugt, indem die verschlüsselten Inhalte der Teile KEYCRYPTED und MSGCRYPTED konkateniert werden. Nach der Anwendung eines Hash-Verfahrens wird beispielsweise RSA für die Erstellung der Signatur verwendet. Die Optionen für diesen Teil der Datei enthalten das verwendete Hashverfahren (woraus die Länge des Hashes abgeleitet werden kann) als auch das für die Signatur verwendete Kryptosystem. Ein Beispiel mit SHA256 und RSA sähe demnach so aus:

```
-----BEGIN SIGNATURE SHA256 RSA-----
<Base64 kodiertes Resultat der RSA-Signierung von SHA256(KEYCRYPTED|MSGCRYPTED)>
-----END SIGNATURE-----
```

3.2 Schlüsselformat

3.2.1 RSA

RSA Schlüssel bestehen aus Exponent (e oder d) und dem Produkt der beiden Primzahlen (n). Diese werden in einer Datei mit dem folgenden Format gespeichert:

```
-----BEGIN RSA PUBLIC KEY-----
<Base64 kodierte Binaerdarstellung von e>,<Base64 kodierte Binaerdarstellung von n>
-----END RSA PUBLIC KEY-----
```

3.3 Verwendung

3.3.1 Dateien

plain Datei mit zu verschlüsselndem Inhalt oder Resultat der Entschlüsselung

crypt Datei mit Resultat der Verschlüsselung

rsapriv Datei mit eigenem privatem RSA-Schlüssel

rsapub Datei mit eigenem öffentlichem RSA-Schlüssel

rsapubrecv Datei mit öffentlichem RSA-Schlüssel des Empfängers

3.3.2 Schlüsselerzeugung

Erzeugung eines Schlüsselpaares:

```
./hskeygenerator
```

3.3.3 Ver- und Entschlüsselung

Verschlüsselung:

```
./hsencrypt <asymm. kryptosystem> <hashverfahren> <symm. kryptosystem> <modus> <empfaenger public key> <verschluesselte datei>
```

Entschlüsselung:

```
./hsdecrypt <sender publickey> <verschluesselte datei>
```

4 Implementation

4.1 Hauptfiles

4.1.1 hskeygenerator.hs

```
import System.Environment
import Kpspcrypto.RSAKey
import System.Random

import qualified Data.ByteString.Char8 as B

main =
        do      gen <- newStdGen
                putStrLn "Enter filename:"
                fileName <- getLine
                B.writeFile (fileName ++ "RsaPrivKey") $ getPrivK $ genK gen
                B.writeFile (fileName ++ "RsaPubKey") $ getPubK $ genK gen
                        where
                                getPrivK (x, _) = x
                                getPubK (_, y) = y
```

4.1.2 hsencrypt.hs

```
-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

-- runhaskell -XOverloadedStrings hsencrypt.hs params...

import System.Environment
import System.Random
import qualified Data.ByteString.Char8 as B
import Kpspcrypto.Msg
import qualified Kpspcrypto.MsgCrypted as M
import qualified Kpspcrypto.KeyCrypted as K
import qualified Kpspcrypto.Signature as S

main = do
        args <- getArgs
        handleArgs $ map B.pack args


handleArgs :: [B.ByteString] -> IO()
handleArgs args = do
        if length args /= 7 then do
                printUsage
        else do
                let asym = args !! 0
                let hash = args !! 1
```

```haskell
                let sym = args !! 2
                let blockmode = args !! 3
                let ownprivkey = args !! 4
                let rcptpubkey = args !! 5
                let infile = args !! 6
                pubkey <- B.readFile $ B.unpack rcptpubkey
                privkey <- B.readFile $ B.unpack ownprivkey
                plainFileContent <- B.readFile $ B.unpack infile
                rgen <- getStdGen
                let (mMsgPart, symkey) = M.genMsgPart rgen sym blockmode plainFileContent --generates the crypted Message
                let kMsgPart = K.genMsgPart asym pubkey symkey --generates the crypted Key
                let plainS = [kMsgPart,mMsgPart]
                let sMsgPart = S.genMsgPart asym privkey hash $ plainS --generates the signature
                let msgParts = map (B.pack . show) [kMsgPart,mMsgPart,sMsgPart]
                B.writeFile (B.unpack infile ++ "Encrypted") $ B.intercalate "\n\n" msgParts

printUsage :: IO()
printUsage = do
        putStrLn "you need to call this binary in this way:"
        putStrLn "hsencrypt asymCipher hashAlg symCipher chainingMode privKey publicKey plainFile"
        putStrLn "Example: hsencrypt RSA SHA256 AES256 CBC privkey pubkey plaintext.txt"
```

4.1.3 hsdecrypt.hs

```haskell
-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

-- runhaskell -XOverloadedStrings hsencrypt.hs params...

import System.Environment
import Data.List

import Kpspcrypto.Msg
import qualified Kpspcrypto.KeyCrypted as K
import qualified Kpspcrypto.MsgCrypted as M
import qualified Kpspcrypto.Signature as S
import qualified Data.ByteString.Char8 as B

main = do
        args <- getArgs
        handleArgs args

handleArgs :: [String] -> IO()
handleArgs args = do
        if length args /= 3 then do
                printUsage
```

```
        else do
                [ourprivkey, senderpubkey, cryptcontent] <- mapM B.readFile args
                let parts@[keypart,msgcpart,sigpart] = sort $ getMsgParts cryptcontent
                let sigOK = S.verifySig senderpubkey parts
                if sigOK then do
                        let symkey = K.getSymKey ourprivkey keypart
                        let plaintext = M.getPlain symkey msgcpart
                        if "Encrypted" `isSuffixOf` (args !! 2) then do
                                let plainFile = dropEnd 9 $ args !! 2
                                B.writeFile plainFile plaintext
                        else do
                                putStrLn "Output File?"
                                plainFile <- getLine
                                B.writeFile plainFile plaintext
                else do
                        putStrLn "signature or key was wrong, exiting..."

printUsage :: IO()
printUsage = do
        putStrLn "you need to call this binary in this way:"
        putStrLn "hsdecrypt yourPrivKey sendersPublicKey cryptedfile.txt"

dropEnd :: Int -> [a] -> [a]
dropEnd n = reverse . drop n . reverse
```

4.2 Zwischenfiles

4.2.1 KeyCrypted.hs

```
module Kpspcrypto.KeyCrypted where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import qualified Data.Map as M
import Data.Maybe

import Kpspcrypto.Msg
import Kpspcrypto.Pad
import Kpspcrypto.Serial
import qualified Kpspcrypto.Base64 as B64
import qualified Kpspcrypto.RSA as RSA

-- creates a KEYCRYPTED-msgpart using the given asymmetric
-- cipher, the given key for the asymmetric cipher and
-- the given content in encrypted form
genMsgPart :: AsymCipher -> AsymKey -> B.ByteString -> MsgPart
genMsgPart "RSA" akey skey = MsgPart KEYCRYPTED ["RSA"] enckey
        where
```

```
                    enckeyed = map B64.encode [RSA.encrypt akey blocks | blocks <- block 4 skey]
                    enckey = B.intercalate "," enckeyed

getSymKey :: AsymKey -> MsgPart -> B.ByteString
getSymKey akey msg = (fromJust $ M.lookup cipher ciphers) akey msg
        where
                    cipher = head $ options msg

getSymKeyFromRSA :: AsymKey -> MsgPart -> B.ByteString
getSymKeyFromRSA akey msg = B.concat [RSA.decrypt akey $ B64.decode block | block <- B.split ',' $ content msg]

ciphers :: M.Map B.ByteString (AsymKey -> MsgPart -> B.ByteString)
ciphers = M.fromList [("RSA", getSymKeyFromRSA)]


{-----------------------------------
sample data and tests
-----------------------------------}
rsapubkey = "-----BEGIN RSA PUBLIC KEY-----\nBrk=,BAYh\n-----END RSA PUBLIC KEY-----" :: B.ByteString
rsaprivkey = "-----BEGIN RSA PRIVATE KEY-----\nBV0=,BAYh\n-----END RSA PRIVATE KEY-----" :: B.ByteString

-- reicht fuer 4 bytes :)
rsapriv2 = "-----BEGIN RSA PRIVATE KEY-----\nzFEWC0E=,AQro6bcX\n-----END RSA PRIVATE KEY-----" :: B.ByteString
rsapub2 = "-----BEGIN RSA PUBLIC KEY-----\nAQAB,AQro6bcX\n-----END RSA PUBLIC KEY-----" :: B.ByteString

ourdata = "ourdata" :: B.ByteString

simplegentest = genMsgPart "RSA" rsapriv2 ourdata
simplegetKeytest = getSymKey rsapub2 simplegentest
```

4.2.2 MsgCrypted.hs

```
module Kpspcrypto.MsgCrypted (genMsgPart, getPlain) where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import qualified Data.Map as M
import Data.Maybe
import System.Random
import Data.Char

import qualified Kpspcrypto.AES256 as AES
import qualified Kpspcrypto.Base64 as B64
import Kpspcrypto.Msg
import Kpspcrypto.BlockModes
```

```haskell
import Kpspcrypto.Pad

type Key = B.ByteString
type SymCipher = B.ByteString
type ChainMode = B.ByteString

-- create a MSGCRYPTED-part using a random IV and a random Key, also
-- returns the used key for further usage (in the KEYCRYPTED part)
genMsgPart :: StdGen -> SymCipher -> ChainMode -> B.ByteString -> (MsgPart, Key)
genMsgPart rgen "AES256" "CBC" plain = (MsgPart MSGCRYPTED ["AES256","CBC"] (ivenc `B.append` "," `B.append` plainenc), key)
        where
                [key,iv] = rndStrs [32,16] rgen
                plainenc = B64.encode $ (cbc (AES.encode key) iv) plain
                ivenc = B64.encode iv
genMsgPart rgen "AES256" "ECB" plain = (MsgPart MSGCRYPTED ["AES256","ECB"] plainenc, key)
        where
                key = rndStr 32 rgen
                plainenc = B64.encode $ (ecb (AES.encode key) iv) plain
                --only length matters, must be the same as the blocksize of the cipher
                iv = B.replicate 16 '\0'

-- decodes the content of a MSGCRYPTED-part using the supplied key
getPlain :: Key -> MsgPart -> B.ByteString
getPlain key msg = (fromJust $ M.lookup cipher decodingfunctions) key msg
        where
                cipher = head $ options msg

-- decodes the content of a part encrypted using AES
getPlainFromAES :: Key -> MsgPart -> B.ByteString
getPlainFromAES key msg = (modef (AES.decode key) iv) cont
        where
                mode = options msg !! 1
                -- find the function which "unapplies" the block-chaining mode
                modef = fromJust $ M.lookup mode modes
                -- for ecb we have to supply a pseudo-iv, for cbc the iv is
                -- part of the content of the msgpart
                [iv, cont]      | mode == "ECB" = [B.replicate 16 '\0', B64.decode $ content msg]
                                | mode == "CBC" = map B64.decode $ B.split ',' $ content msg

-- maps the option-value in the msgpart-header to the function
-- responsible for decoding a part
decodingfunctions :: M.Map B.ByteString (Key -> MsgPart -> B.ByteString)
decodingfunctions = M.fromList [("AES256", getPlainFromAES)]

-- maps the option-value in the msgpart-header to the function
-- responsible for "unapplying" the block-chaining
```

```haskell
modes :: M.Map B.ByteString ((Block -> Block) -> IV -> B.ByteString -> B.ByteString)
modes = M.fromList [("CBC",uncbc), ("ECB",unecb)]

{-------------------------------------
creating random keys, ivs etc...
-------------------------------------}
-- takes a list of lengths and returns random ByteStrings with
-- these lengths created using the supplied random generator
rndStrs :: [Int] -> StdGen -> [B.ByteString]
rndStrs lengths gen = split lengths allrndstrs
        where
                split [] "" = []
                split (len:lens) tosplit = B.take len tosplit : (split lens (B.drop len tosplit))
                allrndstrs = rndStr (sum lengths) gen

-- creates a random ByteString with given length using the
-- supplied random generator
rndStr :: Int -> StdGen -> B.ByteString
rndStr n gen = B.pack $ rndCL n gen

-- creates a random String with given length using the
-- supplied random generator
-- the reason for creating this method is that prepending
-- single Chars to [Char] is way faster (O(1)) the same operation
-- than on a ByteString (O(n))
rndCL :: Int -> StdGen -> String
rndCL 0 gen = ""
rndCL n gen = chr rc : rndCL (n-1) newgen
        where
                (rc, newgen) = randomR (0,255) gen

{----
tests
----}
runTests :: Bool
runTests = and [testAESECB,testAESCBC]

contents :: [B.ByteString]
contents =      ["the very secret and hopefully somewhat protected plaintext"
                        ,"another, shorter text"
                        ,""
                        ,B.replicate 5000 't'
                        ]

testAESECB :: Bool
testAESECB = and [plain m == getPlain (key m) (msg m) | m <- msgskeys]
        where
                msgskeys = [(genMsgPart rnd "AES256" "ECB" cont,cont) | rnd <- rnds, cont <- contents]
                msg = fst . fst
                key = snd . fst
                plain = snd
```

```
testAESCBC :: Bool
testAESCBC = and [plain m == getPlain (key m) (msg m) | m <- msgskeys]
        where
                msgskeys = [(genMsgPart rnd "AES256" "CBC" cont, cont) | rnd <- rnds, cont <- contents]
                msg = fst . fst
                key = snd . fst
                plain = snd

rnds :: [StdGen]
rnds = [mkStdGen i | i <- [13..63]]
```

4.2.3 Signature.hs

```
module Kpspcrypto.Signature where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import qualified Data.Map as M
import Data.List
import Data.Maybe

import qualified Kpspcrypto.Base64 as B64
import qualified Kpspcrypto.RSA as RSA
import qualified Kpspcrypto.SHA256 as SHA
import Kpspcrypto.Pad
import Kpspcrypto.Msg

-- create a signature msgpart which contains the signed hash of the other msgparts
genMsgPart :: AsymCipher -> AsymKey -> HashType -> [MsgPart] -> MsgPart
genMsgPart "RSA" akey "SHA256" [kcpart,msgcpart] = MsgPart SIGNATURE ["RSA","SHA256"] signature
        where
                hashed = SHA.hash $ B.concat [content kcpart, content msgcpart]
                signed = map B64.encode [RSA.sign akey blocks | blocks <- block 6 hashed]
                signature = B.intercalate "," signed


-- verifies the signature of the whole msg
verifySig :: AsymKey -> [MsgPart] -> Bool
verifySig akey parts = and $ zipWith (checksig akey) bsigs bs
        where
                [kpart,mpart,spart] = sort parts
                [k,m,s] = map content [kpart,mpart,spart]
                msgh = hashf $ k `B.append` m
                bsigs = [B64.decode block | block <- B.split ',' s]
                bs = block 6 msgh
```

```
                  sigtype = options spart !! 0
                  hashtype = options spart !! 1
                  checksig = fromJust $ M.lookup sigtype checksigs
                  hashf = fromJust $ M.lookup hashtype hashfs

-- contains the hashfunctions, key is the value of the option in the MsgPart-Header
hashfs :: M.Map HashType (B.ByteString -> B.ByteString)
hashfs = M.fromList [("SHA256",SHA.hash)]

-- contains the "check signature" functions, key is the value of the option in the MsgPart-Header
checksigs :: M.Map AsymCipher (AsymKey -> B.ByteString -> B.ByteString -> Bool)
checksigs = M.fromList [("RSA",RSA.checksig)]

{-----------------------------
sample data and tests
-----------------------------}
runTests :: Bool
runTests = and [verifySig pub $ (genMsgPart "RSA" priv "SHA256" otherparts) : otherparts | (pub,priv) <- keys]
      where
                  otherparts = [kcpart,msgcpart]
                  kcpart = MsgPart KEYCRYPTED ["RSA"] "ourkeyourkeyourkeyourkeyourkey"
                  msgcpart = MsgPart MSGCRYPTED ["SHA256","CBC"] "ourdataourdataourdataourdataourdata"

keys :: [(B.ByteString,B.ByteString)]
keys = [("-----BEGIN RSA PUBLIC KEY-----\nAQAB,iUdRIBeyL3qX\n-----END RSA PUBLIC KEY-----"
                   ,"-----BEGIN RSA PRIVATE KEY-----\nH0vn/c/pBfHZ,iUdRIBeyL3qX\n-----END RSA PRIVATE KEY-----")
                  ,("-----BEGIN RSA PUBLIC KEY-----\nAQAB,Y9G9TSdJNf0j\n-----END RSA PUBLIC KEY-----"
                   ,"-----BEGIN RSA PRIVATE KEY-----\nH0jRB6FRS9Th,Y9G9TSdJNf0j\n-----END RSA PRIVATE KEY-----")
                  ,("-----BEGIN RSA PUBLIC KEY-----\nAQAB,Lfh0qKrNlchv\n-----END RSA PUBLIC KEY-----"
                   ,"-----BEGIN RSA PRIVATE KEY-----\nEVl5vcC88PQh,Lfh0qKrNlchv\n-----END RSA PRIVATE KEY-----")
                  ]
```

4.3 Symmetrische Verschlüsselung

4.3.1 AES256.hs

```
module Kpspcrypto.AES256 (encode, decode) where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}
```

```haskell

{--
this module uses Codec.Encryption.AES from the "crypto"-Package
to perform the actual encryption and decryption functions
the Word128-Interface of Codec.Encryption.AES is converted to
a simpler to use Interface using ByteStrings
--}

import qualified Data.ByteString.Char8 as B
import qualified Codec.Encryption.AES as CEAES
import Data.Word
import Data.LargeWord

import Kpspcrypto.Serial

type Block = B.ByteString
type Key = B.ByteString

{------------------
encoding functions
------------------}
-- crypt a single block using ECB and the supplied key
encode :: Key -> Block -> Block
encode key plain = w1282b $ CEAES.encrypt keyw plainw
        where
                keyw = b2w128 key
                plainw = b2w128 plain

{------------------
decoding functions
------------------}
decode :: Key -> Block -> Block
decode key crypted = w1282b $ CEAES.decrypt keyw cryptedw
        where
                keyw = b2w128 key
                cryptedw = b2w128 crypted


{-------
helpers
-------}
-- converts the last 16 Bytes of a BString to a Word128
b2w128 :: B.ByteString -> Word128
b2w128 = fromIntegral . asInt

-- converts a Word128 to a 16 Byte BString
w1282b :: Word128 -> B.ByteString
w1282b s = B.replicate (16-clen) '\0' `B.append` converted
        where
                converted = asStr $ toInteger s
                clen = B.length converted

{----
tests
----}
```

```haskell
-- tests whether a given string (first in tuple) which gets
-- encrypted and then decrypted using the same or different
-- key(s) is (not) the same as the original string

runTests :: Bool
runTests = and [testAES test | test <- tests]

-- runs tests from "tests"
-- compares d(e(plain)) and plain using the supplied eq-function
-- see comment of "tests" for further information
testAES :: (Block,(Block->Block->Bool),(Block->Block),(Block->Block)) -> Bool
testAES (plain,eq,e,d) = plain `eq` (d $ e plain)

-- different keys
key1 = "justAKeyjustBKey" :: B.ByteString
key2 = "justBKeyjustAKey" :: B.ByteString

-- partially apply the encode and decode functions using a key
-- results in functions of the type (Block -> Block)
e1 = encode key1
e2 = encode key2
d1 = decode key1
d2 = decode key2

-- (plaintext, equality-function, encoding function, decoding function)
-- the equality-function should return true if d(e(plain)) matches
-- the expected result, if you decode using another key than the one
-- used for encode, you expect the result to be different from plain,
-- thus you need to supply (/=) as "equality"-function
tests :: [(Block,(Block->Block->Bool),(Block->Block),(Block->Block))]
tests = [(nulls,(==),e1,d1)
                ,(nulls,(/=),e1,d2)
                ,(nulls,(/=),e2,d1)
                ,(nulls,(==),e2,d2)
                ,(as,(==),e1,d1)
                ,(as,(/=),e1,d2)
                ,(as,(/=),e2,d1)
                ,(as,(==),e2,d2)
                ]
        where
                nulls = B.replicate 16 '\0'
                as = B.replicate 16 'a'
```

4.3.2 SHA256.hs

```haskell
module Kpspcrypto.SHA256 (hash) where

-- http://en.wikipedia.org/wiki/SHA-2

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

```

```haskell
import qualified Data.ByteString.Char8 as B
import Data.Word
import Data.Bits
import Data.Char
import Text.Printf -- for tests only

import Kpspcrypto.Pad
import Kpspcrypto.Serial

-- apply SHA256 to given data, result will always be 32 bytes long
hash :: B.ByteString -> B.ByteString
hash msg = B.concat $ map w2b h
        where
                h = foldl perchunk hs preprocessed
                preprocessed = chunks $ preprocess msg

-- adds padding (in the form 0x80[00]*) until there are 4 bytes left for the size
shapad :: B.ByteString -> B.ByteString
shapad input = fill $ input `B.snoc` chr 0x80
        where
                -- 56bytes are 448bits
                fill unfilled
                        | lenmod == 56 = unfilled
                        | otherwise = unfilled `B.append` B.replicate remaining '\0'
                lenmod = (B.length input) + 1 `mod` 64 -- +1 because we already added a byte
                --120 because 62 `mod` 64 results in -2, which we cant use for replicate
                remaining = (120 - lenmod) `mod` 64

-- adds padding and size, output length will always be a multiple of 64 bytes
preprocess :: B.ByteString -> B.ByteString
preprocess input = shapad input `B.append` lenAsBStr
        where
                len = 8 * B.length input --in bits
                lenAsBStr = B.pack
                        ['\NUL','\NUL','\NUL','\NUL'
                        ,chr $ shiftR (len .&. 0xFF000000) 24
                        ,chr $ shiftR (len .&. 0x00FF0000) 16
                        ,chr $ shiftR (len .&. 0x0000FF00) 8
                        ,chr $ len .&. 0x000000FF
                        ]

-- prepares a chunk, executes mainloop and adds the result to the hash so far
perchunk :: [Word32] -> B.ByteString -> [Word32]
perchunk curhash chunk = zipWith (+) curhash looped
        where
                broken = map b2w $ block 4 chunk
                expanded = expandwords broken
                looped = mainloop 0 expanded curhash

```

```
-- executes 64 SHA2-Rounds on a chunk
mainloop :: Int -> [Word32] -> [Word32] -> [Word32]
mainloop 64 _ h = h
mainloop i w [a,b,c,d,e,f,g,h] = mainloop (i+1) w [temp2,a,b,c,newd,e,f,g]
        where
                s1 = (e `rotateR` 6) `xor` (e `rotateR` 11) `xor` (e `rotateR` 25)
                ch = (e .&. f) `xor` ((complement e) .&. g)
                temp = h + s1 + ch + ks!!i + w!!i
                newd = d + temp;
                s0 = (a `rotateR` 2) `xor` (a `rotateR` 13) `xor` (a `rotateR` 22)
                maj = (a .&. (b `xor` c)) `xor` (b .&. c)
                temp2 = temp + s0 + maj

-- expands the 16 Word32s to 64 Word32s, according to SHA256 spec
expandwords :: [Word32] -> [Word32]
expandwords cw
        | length cw == 64 = cw
        | otherwise = expandwords $ cw ++ [newword cw]

-- creates the next Word for the expansion
newword :: [Word32] -> Word32
newword cw = cw!!(i-16) + s0 + cw!!(i-7) + s1
        where
                i = length cw
                s0 = (cw!!(i-15) `rotateR` 7) `xor` (cw!!(i-15) `rotateR` 18) `xor` (cw!!(i-15) `shiftR` 3)
                s1 = (cw!!(i-2) `rotateR` 17) `xor` (cw!!(i-2) `rotateR` 19) `xor` (cw!!(i-2) `shiftR` 10)

-- converts 4 Bytes to a Word32
b2w :: B.ByteString -> Word32
b2w = fromInteger . asInt

w2b :: Word32 -> B.ByteString
w2b = asStr . toInteger

-- break input into 512bit blocks
chunks :: B.ByteString -> [B.ByteString]
chunks = block 64

{---
Data
---}
-- initial hash
hs :: [Word32]
hs =
        [0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19]

-- round constants, in every iteration of the innermost
-- loop (mainloop), one of these values is used
ks :: [Word32]
```

```
ks =
        [0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
        ,0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174
        ,0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da
        ,0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967
        ,0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85
        ,0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070
        ,0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
        ,0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
        ]

{------------------------------------------
some tests
data from wikipedia and manual execution
of echo -ne "input" | sha256sum on linux
------------------------------------------}

-- printf "%08x" wandelt einen Int in die Hexdarstellung um
-- ueblicherweise werden hashes in Hex ausgegeben
hex :: B.ByteString -> B.ByteString
hex = B.pack . printf "%08x" . asInt

-- true if no tests failed
runtests :: Bool
runtests = and [testHash test | test <- tests]

testHash :: (B.ByteString,B.ByteString) -> Bool
testHash (input,expected) = (hex $ hash input) == expected

tests = [("","e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855")
                ,("\0","6e340b9cffb37a989ca544e6bb780a2c78901d3fb33738768511a30617afa01d")
                ,("a","ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb")
                ,("hallo","d3751d33f9cd5049c4af2b462735457e4d3baf130bcbb87f389e349fbaeb20b9")
                -- 55 bytes stay in one chunk
                ,(B.replicate 55 'a',"9f4390f8d30c2dd92ec9f095b65e2b9ae9b0a925a5258e241c9f1e910f734318")
                -- 56 bytes and more cause perchunk to be called at least two times
```

```haskell
                    ,(B.replicate 56 'a',"b35439a4ac6f0948b6d6f9e3c6af0f5f590ce20f1bde7090ef7970686ec6738a")
                    ,(B.replicate 57 'b',"2dd0a6d14520f410e18bd2f443f0ff2e7389dfaf9242bb9257730fc190e8085d")
                    ,("Franz jagt im komplett verwahrlosten Taxi quer durch Bayern",
                        "d32b568cd1b96d459e7291ebf4b25d007f275c9f13149beeb782fac071")
                    ,("Frank jagt im komplett verwahrlosten Taxi quer durch Bayern",
                        "78206a866dbb2bf017d8e34274aed01a8ce405b69d45db30bafa00f5eeed7d5")
                    ,(B.replicate 120 'a' `B.append` B.replicate 1000 'Z',
                        "f44da844b446469e8a3c928e6f696b3994e404b1388282267e932744bb")
                    ]
```

4.4 BlockModes.hs

```haskell
module Kpspcrypto.BlockModes where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import Data.Bits

import Kpspcrypto.Pad
import Kpspcrypto.Serial

type Block = B.ByteString
type IV = Block

cbc :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
cbc cipher iv plain = B.concat $ docbc cipher iv $ pad blocklen plain
        where
                blocklen = B.length iv

docbc :: (Block -> Block) -> IV -> [Block] -> [Block]
docbc _ _ [] = []
docbc cipher iv (x:xs) = cblock : docbc cipher cblock xs
        where
                ivi = asInt iv
                xi = asInt x
                ivxorb = asStr $ ivi `xor` xi
                cblock = cipher ivxorb

uncbc :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
```

```haskell
uncbc cipher iv crypt = unpadblocks $ douncbc cipher iv $ block blocklen crypt
        where
                blocklen = B.length iv

douncbc :: (Block -> Block) -> IV -> [Block] -> [Block]
douncbc _ _ [] = []
douncbc cipher iv (x:xs) = plain : douncbc cipher x xs
        where
                ivi = asInt iv
                xdec = asInt $ cipher x
                plain = asStr $ ivi `xor` xdec

ecb :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
ecb cipher iv plain = B.concat [cipher pblock | pblock <- pblocks]
        where
                blocklen = B.length iv
                pblocks = pad blocklen plain

unecb :: (Block -> Block) -> IV -> B.ByteString -> B.ByteString
unecb cipher iv crypt = unpadblocks [cipher cblock | cblock <- cblocks]
        where
                blocklen = B.length iv
                cblocks = block blocklen crypt
```

4.5 Asymmetrische Verschlüsselung

4.5.1 RSA.hs

```haskell
module Kpspcrypto.RSA (encrypt, sign, decrypt, checksig) where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import qualified Kpspcrypto.Base64 as B64
import Kpspcrypto.Serial

type KeyFileContent = B.ByteString
-- first part is e or d, second is n
type Pubkey = (B.ByteString, B.ByteString)
type Privkey = Pubkey


{----------------
public functions
----------------}
encrypt :: KeyFileContent -> B.ByteString -> B.ByteString
encrypt key msgIn = asStr $ modexp msg e n
        where
                e = toInt eIn
                n = toInt nIn
```

```haskell
                msg = asInt msgIn
                (eIn,nIn) = fromFile key

sign :: KeyFileContent -> B.ByteString -> B.ByteString
sign = encrypt

decrypt :: KeyFileContent -> B.ByteString -> B.ByteString
decrypt = encrypt

checksig :: KeyFileContent -> B.ByteString -> B.ByteString -> Bool
checksig pubkey sig msg = encrypt pubkey sig == msg

{-------------------
helper functions
-------------------}
-- modexp b e n returns b^e mod n
-- slightly modified from http://pastebin.com/m142c0ca
modexp :: Integer -> Integer -> Integer -> Integer
modexp b 0 n = 1
modexp b e n
        | even e = (modexp b (e `div` 2) n) ^ 2 `mod` n
        | otherwise = (b * modexp b (e-1) n) `mod` n

{------------------------------------
less related utility functions
------------------------------------}
-- converts an Integer to Base64 encoded ByteString
toStr :: Integer -> B.ByteString
toStr = B64.encode . asStr

-- reads a Base64 encoded Integer from a ByteString
toInt :: B.ByteString -> Integer
toInt = asInt . B64.decode

-- extracts key from file
fromFile :: B.ByteString -> Pubkey
fromFile file = (e,n)
        where
                contentline = B.lines file !! 1
                [e,n] = B.split ',' contentline

{---------------------------------------------------------
sample data (from
http://de.wikipedia.org/wiki/RSA-Kryptosystem)
---------------------------------------------------------}
rsapubkey = "-----BEGIN RSA PUBLIC KEY-----\nFw==,jw==\n-----END RSA PUBLIC KEY-----" :: B.ByteString
rsaprivkey = "-----BEGIN RSA PRIVATE KEY-----\nLw==,jw==\n-----END RSA PRIVATE KEY-----" :: B.ByteString
rsarecvpubkey = "-----BEGIN RSA PUBLIC KEY-----\nBrk=,BAYh\n-----END RSA PUBLIC KEY-----" :: B.ByteString
rsarecvprivkey = "-----BEGIN RSA PRIVATE KEY-----\nBV0=,BAYh\n-----END RSA PRIVATE KEY-----" :: B.ByteString

exmsg = toStr 7
```

```haskell

expub = (toStr 23, toStr 143)
expriv = (toStr 47, toStr 143)

expub2 = (toStr 1721, toStr 263713)
expriv2 = (toStr 1373, toStr 263713)

smalltest = [checksig rsapubkey (sign rsaprivkey str) str | str <- map B.singleton ['\0'..'\140']]
smalltest2 = [checksig rsarecvpubkey (sign rsarecvprivkey str) str | str <- map B.singleton ['\0'..'\255']]
```

4.5.2 RSAKey.hs

```haskell
module Kpspcrypto.RSAKey (genK) where

import qualified Data.ByteString.Char8 as B
import System.Random
import Kpspcrypto.Serial
import qualified Kpspcrypto.Base64 as B64


type P = Integer
type Q = Integer
type Privkey = B.ByteString
type Pubkey = B.ByteString

genK :: StdGen -> (Privkey, Pubkey)
genK rgen = (genPrivK rgen, genPubK rgen)

genPrivK :: StdGen -> Privkey
genPrivK rgen = begin `B.append` toStr (getD $ genKeys rgen) `B.append` "," `B.append` toStr (getN $ genKeys rgen) `B.append` end
        where
                begin = "-----BEGIN RSA PRIVATE KEY-----\n"
                end = "\n-----END RSA PRIVATE KEY-----"
                getN :: (Integer, Integer) -> Integer
                getN (_, n) = n         --oder getN = snd
                getD :: (Integer, Integer) -> Integer
                getD (d, _) = d         --oder getD = fst

genPubK :: StdGen -> Pubkey
genPubK rgen = begin `B.append` toStr 65537 `B.append` "," `B.append` toStr (getN $ genKeys rgen) `B.append` end
        where
                begin = "-----BEGIN RSA PUBLIC KEY-----\n"
                end = "\n-----END RSA PUBLIC KEY-----"
                getN :: (Integer, Integer) -> Integer
                getN = snd


genKeys :: StdGen -> (Integer, Integer)
genKeys rgen = (d, n)
        where
                p = head $ genPrime getP
```

```
                            q = head $ genPrime getQ
                            (getP, newGen) = (randomR (2^34, 2^35-1) rgen) -- different range for p and q to ensure p!=q
                            (getQ, newGen') = (randomR (2^35, 2^37) newGen)
                            d = genD p q
                            n = p*q


--calculate d (decoding) e * d = 1 mod phi(N)
genD :: P -> Q -> Integer
genD p q = if scnd (extendedEuclid 65537 phiN) > 0 then scnd (extendedEuclid 65537 phiN) else phiN + scnd (extendedEuclid 65537 phiN)
        where
                phiN = (p-1)*(q-1)
                scnd (_, x, _) = x

--helper functions
extendedEuclid :: Integer -> Integer -> (Integer, Integer, Integer)
extendedEuclid a b
                                    | b == 0 = (a, 1, 0)
                                    | otherwise = (d,t,s - (div a b)*t)
                                            where
                                                    (d,s,t) = extendedEuclid b (a `mod` b)

--first version with pattern matching -> chose guards to do sth different for once ;)
--extendedEuclid a 0 = b == 0 = (a, 1, 0)
--extendedEuclid a b = (d,t,s - (div a b)*t)
--                                          where
--                                                  (d,s,t) = extendedEuclid b (a `mod` b)

--create a list containing primes to get random p and q
genPrime :: Integer -> [Integer]
genPrime n = if even n then genPrime (n+1) else [x | x <- [n,n+2..], isPrime x]



isPrime :: Integer -> Bool
isPrime x = null [y | y <- takeWhile (\y -> y*y <= x) [2..], x `mod` y == 0]


-- converts an Integer to Base64 encoded ByteString
toStr :: Integer -> B.ByteString
toStr = B64.encode . asStr
```

4.6 Base64.hs

```
module Kpspcrypto.Base64 (encode, decode) where

-- http://www.haskell.org/haskellwiki/DealingWithBinaryData
```

```haskell
-- http://en.wikipedia.org/wiki/Base64

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import qualified Data.Vector as V
import qualified Data.Map as M
import Data.Bits
import Data.Maybe
import Data.Char

{----------------
public functions
----------------}
-- encode a given ByteString using Base64
-- the output length will be a multiple of 4
encode :: B.ByteString -> B.ByteString
encode input = encodeR $ addpad input

-- decode Base64 encoded content of a ByteString
-- input length must be a multiple of 4
decode :: B.ByteString -> B.ByteString
decode encoded = B.take resultlen decWithTrash
        where
                (unpadded, padlen) = unpad encoded
                decWithTrash = decodeR unpadded
                resultlen = B.length decWithTrash - padlen

{----------------
encoding helpers
----------------}
-- recursively substitute 3 bytes with the 4 bytes
-- that result from Base64-encoding
-- the padding length is required for marking the
-- amount of added padding in the final output
encodeR :: (B.ByteString, Int) -> B.ByteString
encodeR ("",_) = ""
encodeR (x,padlen)
        | B.length x /= 3 || padlen == 0 = subs next `B.append` encodeR (rest, padlen)
        -- otherwise: last 3 bytes and we have padding
        | otherwise =
                if padlen == 1 then B.init (subs x) `B.append` "="
                else B.take 2 (subs x) `B.append` "=="
        where
                (next,rest) = B.splitAt 3 x
                -- convert to 6-bit-values, find the corresponding char and
                -- convert the [Char] to a ByteString
                subs input = B.pack $ map (table V.!) (toB64BitGroups input)

```

```haskell
-- splits a ByteString (with length 3) into four 6-bit values
toB64BitGroups :: B.ByteString -> [Int]
toB64BitGroups x = [
        shiftR (ord (B.index x 0)) 2,
        shiftL (ord (B.index x 0) .&. 3) 4 .|. shiftR (ord (B.index x 1)) 4,
        shiftL (ord (B.index x 1) .&. 15) 2 .|. shiftR (ord (B.index x 2)) 6,
        ord (B.index x 2) .&. 63]

-- expands an input to a multiple of 3 Bytes for processing
-- second element of tuple is the length of the added padding
addpad :: B.ByteString -> (B.ByteString, Int)
addpad x = (x `B.append` B.replicate padlen '\0', padlen)
        where
                len = B.length x
                padlen = (3 - len `mod` 3) `mod` 3

-- base64 index table
-- contains the allowed characters in the Base64 output
table :: V.Vector Char
table = V.fromListN 64 (['A'..'Z'] ++ ['a'..'z'] ++ ['0'..'9'] ++ ['+','/'])

{----------------
decoding helpers
----------------}
-- combines 4 6-bit values into a ByteString with length 3
fromB64BitGroups :: [Int] -> B.ByteString
fromB64BitGroups x = B.pack $ map chr ords
        where
                ords = [
                        shiftL (x !! 0) 2 .|. shiftR (x !! 1) 4,
                        shiftL ((x !! 1) .&. 15) 4 .|. shiftR (x !! 2) 2,
                        shiftL ((x !! 2) .&. 3) 6 .|. (x !! 3)]

-- removes the padding from a Base64-encoded input
-- second element in the returned tuple is the amount
-- of bytes to be discarded after decoding
unpad :: B.ByteString -> (B.ByteString, Int)
unpad x = (
        B.takeWhile (/= '=') x `B.append` B.replicate padlen '0',
        padlen )
        where
                padlen = B.length (B.dropWhile (/= '=') x)

-- recursively substitute 4 "Base64-Bytes" with 3 Bytes from
-- the plaintext which was encoded
decodeR :: B.ByteString -> B.ByteString
decodeR "" = ""
decodeR x = subs next `B.append` decodeR rest
        where
                (next,rest) = B.splitAt 4 x
                -- convert ByteString to [Char], restore the original 4
                -- 6-bit-values and convert them to the original 3 Bytes
```

```
                subs input = fromB64BitGroups [fromJust (M.lookup c tableR) | c <- B.unpack input]

-- contains the 6bit-values for the allowed chars in Base64 encoded data
tableR :: M.Map Char Int
tableR = M.fromList [(table V.! v ,v) | v <- [0..63]]

{-----
tests
-----}
runTests :: Bool
runTests = and [str == (decode . encode $ str) | str <- teststrings]

-- returns some strings of various lengths
teststrings = map B.pack [replicate i (chr $ i+j+k) ++ replicate j (chr $ 7*i+9*j) ++ replicate k (chr $ 11*i+3*k+2*j) |
                i <- [0..10], j <- [0..10], k <- [0..10]]
```

4.7 Hilfsfiles

4.7.1 Pad.hs

```
module Kpspcrypto.Pad (pad, unpad, unpadblocks, block) where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import Data.Char

-- separate input into blocks and (always!) add padding
-- if input length mod blocklength == 0 we add a block that
-- only contains padding
pad :: Int -> B.ByteString -> [B.ByteString]
pad n input
        -- not the last block, recurse on following blocks
        | B.length input >= n = next : pad n rest
        -- last block: add padding
        | otherwise = [input `B.append` (B.replicate padlen padchar)]
        where
                (next,rest) = B.splitAt n input
                padlen = n - B.length input
                padchar = chr padlen

-- split input into block of given length
block :: Int -> B.ByteString -> [B.ByteString]
block n "" = []
block n x = next : (block n rest)
        where
                (next,rest) = B.splitAt n x

-- undo "pad n", uses "unpadblocks"
```

```haskell
unpad :: Int -> B.ByteString -> B.ByteString
unpad n input = unpadblocks $ block n input

-- remove the padding in the last block
unpadblocks :: [B.ByteString] -> B.ByteString
-- last block: at least one byte is padding, because
-- pad always adds a padding
unpadblocks [x] = B.take padlen x
        where
                n = B.length x
                padlen = n - (ord $ B.last x)
-- not the last block: recursion on following blocks
unpadblocks (x:xs) = x `B.append` unpadblocks xs

{----
tests
----}
runTests :: Bool
runTests = and [(unpad len $ B.concat $ pad len s) == s | s <- testInputs, len <- [1..30]]

testInputs :: [B.ByteString]
testInputs = [B.replicate i 'a' | i <- [1..200]]
```

4.7.2 Msg.hs

```haskell
module Kpspcrypto.Msg where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import Text.Regex.Posix

type AsymCipher = B.ByteString -- z.B. "RSA"
type AsymKey = B.ByteString -- kompletter inhalt des pub- oder privatekeyfiles
type HashType = B.ByteString -- z.B. "SHA256"

sampleMsgStr =  "-----BEGIN KEYCRYPTED RSA 8-----\nbla//+ba\n-----END KEYCRYPTED-----\n\n" `B.append`
                                  "-----BEGIN MSGCRYPTED AES256 CBC-----\nbla+bl/ubb\n-----END MSGCRYPTED-----\n\n" `B.append`
                                  "-----BEGIN SIGNATURE SHA256 RSA 8-----\nbl/+ubb+/+lubb\n-----END SIGNATURE-----\n\n"

-- possible types of messageparts
data MsgType = KEYCRYPTED | MSGCRYPTED | SIGNATURE deriving (Show, Read, Eq, Ord)

-- this can hold any type of messagepart
```

```
data MsgPart = MsgPart  {       msgtype :: MsgType
                                                , options :: [B.ByteString]
                                                , content :: B.ByteString
                                                } deriving (Read, Eq)

makeMsg :: (MsgType, [B.ByteString]) -> B.ByteString -> MsgPart
makeMsg (msgtype, options) content = MsgPart msgtype options content

-- make the msg print in the way we want and expect it in a file
instance Show MsgPart where
        show msg = "-----BEGIN " ++ show (msgtype msg) ++ " " ++
                B.unpack (B.intercalate " " (options msg)) ++
                "-----\n" ++ B.unpack (content msg) ++ "\n" ++
                "-----END " ++ show (msgtype msg) ++ "-----"
-- alternative for intercalate: foldl (\acc option -> acc `B.append` " " `B.append` option) "" (options msg)

-- make the msg sortable by type
instance Ord MsgPart where
        compare a b = compare (msgtype a) (msgtype b)

-- interprets the first line of a msgpart
readHdr :: B.ByteString -> (MsgType, [B.ByteString])
readHdr hdr = (msgtype, msgoptions)
        where
                -- drop 10 drops "-----BEGIN "
                contents = B.words . B.takeWhile (/= '-') . B.drop 10
                msgtype = read . B.unpack . head $ contents hdr
                msgoptions = tail $ contents hdr

-- interpret a ByteString as MsgPart
-- the input has to be in the right form
readMsg :: B.ByteString -> MsgPart
readMsg input = makeMsg (readHdr headerLine) content
        where
                headerLine = head $ B.lines input
                -- outermost 'init' is for removing the trailing \n added by unlines
                content = B.init . B.unlines . init . tail $ B.lines input

-- gets messageparts from a file
-- http://stackoverflow.com/questions/7636447/raise-no-instance-for-regexcontext-regex-char-string
getMsgParts :: B.ByteString -> [MsgPart]
getMsgParts input = map readMsg regmatches
        where
                regmatches = getAllTextMatches (input =~ regex :: AllTextMatches [] B.ByteString)
                regex = "-----BEGIN [A-Z0-9 ]+-----\n[a-zA-Z0-9+/=,]+\n-----END [A-Z0-9]+-----" :: B.ByteString
```

4.7.3 Serial.hs

```haskell
module Kpspcrypto.Serial where

-- needed for using string-literals with ByteString
-- see http://hackage.haskell.org/packages/archive/bytestring/0.10.2.0/doc/html/Data-ByteString-Char8.html
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString.Char8 as B
import Data.Char

-- interpret the bytes of a ByteString as
-- Integer, result will always be positive
asInt :: B.ByteString -> Integer
asInt str = f . reverse $ B.unpack str
        where
                f "" = 0
                f (x:xs) = toInteger (ord x) + 256 * f xs

-- convert a positive Integer to a ByteString
asStr :: Integer -> B.ByteString
-- edge case here because we dont want 0 to become "", but
-- we need a ""-edge-case in the helper function
asStr 0 = B.singleton '\0'
asStr i = B.pack . reverse $ f i
        where
                f 0 = []
                f i = chr (fromInteger (i `mod` 256)) : f (i `div` 256)
```

5 Testfälle

5.1 Key Generator

Der Keygenerator soll zufällige RSA-Keys erzeugen. Wir können ihn nach der Kompilation mittles *ghc -XOverloadedStrings hskeygenerator.hs* aufrufen, um Keys für unseren Sender und den Empfänger zu generieren:

```
[iso@iso-t530arch tmp]$ ./hskeygenerator
Enter filename:
sender
[iso@iso-t530arch tmp]$ ./hskeygenerator
Enter filename:
receiver
```

Die Inhalte der erzeugten Schlüsselfiles sehen dann wie folgt aus:

```
[iso@iso-t530arch tmp]$ cat senderRsaPubKey
-----BEGIN RSA PUBLIC KEY-----
AQAB,uLEhNs/uRmG9
-----END RSA PUBLIC KEY-----

iso@iso-t530arch tmp]$ cat senderRsaPrivKey
-----BEGIN RSA PRIVATE KEY-----
```

```
Ia9kwGVnj+Bh,uLEhNs/uRmG9
-----END RSA PRIVATE KEY-----

[iso@iso-t530arch tmp]$ cat receiverRsaPubKey
-----BEGIN RSA PUBLIC KEY-----
AQAB,OSD0b/N9Btp5
-----END RSA PUBLIC KEY-----

[iso@iso-t530arch tmp]$ cat receiverRsaPrivKey
-----BEGIN RSA PRIVATE KEY-----
EbzFqCfx729x,OSD0b/N9Btp5
-----END RSA PRIVATE KEY-----
```

Wie erwünscht enthalten die jeweils zueinander gehörigen Private- und Public-Keys den selben Wert für N („uLEhNs/uRmG9“ bei sender, „OSD0b/N9Btp5“ bei receiver). Bei beiden Schlüsselpaaren wird für e der Wert 65537 verwendet, d hingegen unterscheidet sich zwischen den Schlüsselpaaren.

In diesem Fall wurden die folgenden Schlüssel erzeugt. Die Werte können mittels folgendem Befehl (nach dem Laden von hsencrypt.hs) zurückgewonnen werden:

```
Kpspcrypto.Serial.asInt $ Kpspcrypto.Base64.decode "Ia9kwGVnj+Bh"
```

- Sender e: 65537, d: 621380992428485369953, n: 3406964452648185913789
- Receiver e: 65537, d: 327197112392121020273, n: 1053839058196540873337

5.2 Ver- und Entschlüsselung von Nachrichten

5.2.1 Vorbereitung

Mit den folgenden Befehlen werden (unter Linux) zufällige Testnachrichten erstellt und deren Hashwerte (für den späteren Vergleich) ermittelt:

```
[iso@iso-t530arch tmp]$ dd bs=1k count=1 if=/dev/urandom of=1k.msg
1+0 records in
1+0 records out
1024 bytes (1.0 kB) copied, 0.000435418 s, 2.4 MB/s
[iso@iso-t530arch tmp]$ dd bs=1k count=100 if=/dev/urandom of=100k.msg
100+0 records in
100+0 records out
102400 bytes (102 kB) copied, 0.0180468 s, 5.7 MB/s
[iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
1b13213582917f5b36751ab66bf22ae19233e259de319181acbeb64d712a3559  100k.msg
```

5.2.2 Verschlüsselung ECB

Für diese Tests werden die zuvor erzeugten Schlüssel verwendet. Wir verschlüsseln die beiden Nachrichten mit den öffentlichen Schlüssel des Empfängers und signieren den Inhalt der Nachricht mit Hilfe des privaten Schlüssels des Senders:

```
[iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 ECB
    senderRsaPrivKey receiverRsaPubKey 1k.msg
[iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 ECB
    senderRsaPrivKey receiverRsaPubKey 100k.msg
```

Den Inhalt des 1k-Files schauen wir uns an (Zeilenumbrüche innerhalb der Msg-Teile wurden manuell hinzugefügt):

```
[iso@iso-t530arch tmp]$ cat 1k.msgEncrypted
-----BEGIN KEYCRYPTED RSA-----
KB25vJ46fR06,ISPN+nHCYyt3,AdvzIdRlMOpz,LFpPe8rS7WCM,NOgghBG7w74E,
Li21sI2vAGBH,GfDWpqTKuuP+,KgFI/+tsr+NX
-----END KEYCRYPTED-----

-----BEGIN MSGCRYPTED AES256 ECB-----
NC5ja4U25ZMXVRQAFlYn0aIJcr26nx+gEYlARbw52QHDoQnywSEZRTFXuQ5K4kSvx1AiW
L+s719SBx0OGi/o+JOKp+cutp4ArUSHK7aWdBVrJpTi6a0Bj0fyKr5xsAVGZAfC3XCx14
li16+05/p+fOwN8LzixmqQ2R0T49n+iI7n/9Uw1wu1UbYPfjgBIeXT8HGHc+GevYAkrXv
QHXVdbFaBZXQBQWWcfr+A0rbfOkxG2bDh5FwR7WF+7PFDK7h2peXSFJ/nFu4SSLBVbEED
PFbbhG/fS+IcQ4y5Mu5/ICfc2WeZg8r83cRhu2nDJouOzQXM8qxvLrpY0IZ5xhbB2b0mT
vg01KaD9mp4UrtxDvsqLs9kwuGgMKruqKolM3C49zx3uhBSb0T4uF/2hgowPuhrN0Xppd
ytMuMvstvJGImPEuj+CAFJ6GbFbBQj5xwlvsgx3tsYiCzTe6A62m0yuATioFDAuGB7A6a
tdgDLiNyfV3oNFGuBukIe1UAZyz5xDWyfCbR0bG8Ok+38oXqMzRrdyv/zhtwZaugnhnLa
H/Eu8H3AmqoMz6hVp6xDtX72HcYu/FXOaRtFZsH6PWEPJdu02uCGQ7G/1dUM2frG6SPSj
lfRxPpUmQTkoDOtk51t/nv2BQUqcwYuDHWHzL4wIw/+wAY1xe5LU/WiPd9/3Galv4saan
wDsrpFqvbgMdPkOW6wNHBu4Vl8KG4TPRpymwYKvwhg2hIU01RT+zJ+ezVsgrl74cNj5DP
3NrlNdiKhRVmd7oAJPHde2g6ZhdB1YrcAhSsiVOq118UiKpbn/LfobGUKOTa51/wollRs
riC4uU4ULyXix0C+WHLHdGd/xwaGmoBSBf0h+I/fUZ97xw6fgdN0oyfek75tdQpiPI18X
NHVwYd2qAdH/BTuM+ODuqjgPcuunUzJXfGJpQVDBPPQh6akzIyyHfQMBJN0N4o1jfUKL6
CFWaHmXRQpCnVkFwsKP5NlOYfpjsY7N34OmrqAOZP/wIBYs+HjcF4YxirE98iOcII5Rsf
O5i/wtQiyXgc/kFkY93uluwrZQT0OJKWzctH0isSIRPqGk3ySliB7Ceh7spqMIVPuPpn4
W63yFtf9XrzDtW8gMv/roX/0HJmeoW1ysjPjJ+teT6lZ50REmh4/LFGjkRQx2n1x+wflC
wwTusYA/1I/lzZRDvHhpljHGX1x8H9fYJekBBKYmPu6ufS0uEFS2UwaHFHalwIXuE2kVj
3Bfiop5rqgZZimVfoFKnFjo+V0d4SXuuqAVv+IFnPorG4nXThHf2TN5h6cn40xrlOfrM3
iRLB/CMCuvyvaV0luRLSKmYbrjDQr9ShFhLf+ER6Mp0eVxp8GfxnKdkCCpiHjrTM4eVLM
IoIMH1s=
-----END MSGCRYPTED-----

-----BEGIN SIGNATURE RSA SHA256-----
i7XYWSqqv8b7,P3ERQK9xnLYD,kMwA8TdQpUcb,OmazloPVsEod,YllxmJXn8mmf,
FmK2J2p4xgnI
-----END SIGNATURE-----
```

Im Header des MSGCRYPTED-Teils ist wie erwartet die ECB-Option gesetzt.

5.2.3 Entschlüsselung ECB

Die beiden Nachrichten können mit Hilfe des privaten Schlüssels des Empfängers und des öffentlichen Schlüssels des Senders entschlüsselt werden (dabei werden die in unserem Fall noch vorhandenen Original-Plaintext-Dateien überschrieben). Die SHA256-Summen der (neuen) Plaintext-Dateien entsprechenen denjenigen vor der Ver- und Entschlüsselung.

```
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey 1k.msgEncrypted
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey 100k.msgEncrypted
[iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
1b13213582917f5b36751ab66bf22ae19233e259de319181acbeb64d712a3559  100k.msg
```

5.2.4 Verschlüsselung CBC

Die Nachrichten werden mit den selben Keys diesmal im CBC-Modus verschlüsselt und der Header des MSGCRYPTED-Teils überprüft:

```
[iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 CBC senderRsaPrivKey receiverRsaPubKey 1k.msg
[iso@iso-t530arch tmp]$ ./hsencrypt RSA SHA256 AES256 CBC senderRsaPrivKey receiverRsaPubKey 100k.msg
[iso@iso-t530arch tmp]$ head -5 1k.msgEncrypted
-----BEGIN KEYCRYPTED RSA-----
DiUzRSdJA72E,EdXLBpnsii33,HL4nUeyNrdUn,BL/Saw/ZQls8,IDrIpuKmMjkJ,
Hrx7GSmFoNQV,NQOf4xGv0XMJ,KtodRCoiyeZp
-----END KEYCRYPTED-----

-----BEGIN MSGCRYPTED AES256 CBC-----
....
```

5.2.5 Entschlüsselung CBC

Wiederum Entschlüsseln wir die Nachrichten mit den geeigneten Keys und kontrollieren die Hashwerte:

```
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey 1k.msgEncrypted
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey 100k.msgEncrypted
[iso@iso-t530arch tmp]$ sha256sum 1k.msg 100k.msg
fd6df86538db0013a9f943b2d8a03d52a5d6a40cbe3243408167dc15e29a855d  1k.msg
b99126267061a7ca4a63a0a59ec6e4b331d63e1dd852e2f4ba4fb72b98048a5d  100k.msg
```

Die Hashwerte stimmen überein, woraus geschlossen werden kann, dass die Ver- und Entschlüsselung der Nachrichten keinen Informationsverlust zur Folge hat.

5.2.6 Versuch der Entschlüsselung von modifizierten Crypt-Files

In diesem Test wird in der CBC-verschlüsselten Datei 1k.msgEncrypted eine Anpassung innerhalb einer der drei Msg-Teilen vorgenommen und versucht, die Nachricht zu entschlüsseln:

```
[iso@iso-t530arch tmp]$ nano 1k.msgEncrypted
[iso@iso-t530arch tmp]$ ./hsdecrypt receiverRsaPrivKey senderRsaPubKey 1k
    .msgEncrypted
signature or key was wrong, exiting...
```

Wir erhalten die erwartete Fehlermeldung und die Datei wurde nicht entschlüsselt.