kpsp Team14
Hybride Kryptographie

06.06.2013

Marianne Schoch, Pascal Schwarz

Inhalt

- Überblick Aufgabenstellung
- Verwendete Verfahren
- Nachrichtenformat
- Aufteilung Funktionalität
- Code Auszüge
- Demo

Aufgabenstellung

- Ver- und Entschlüsselung von Nachrichten
  - Vertraulichkeit
- Signierung von Nachrichten
  - Authentizität, Integrität
- Hybride Kryptographie
  - Nachricht symmetrisch verschlüsselt
  - Key für symmetrisches Verfahren mittels asymmetrischem Verfahren geschützt
  - Signatur unter Zuhilfenahme von Hashfunktion

Verwendete Verfahren

- Hashfunktion
  - SHA256
- Symmetrische Verschlüsselung
  - AES256
  - ECB, CBC
- Asymmetrische Verschlüsselung
  - RSA
- Base64
- Einbindung weiterer Kryptosysteme möglich

Nachrichtenformat

- Unterteilung
  - KEYCRYPTED, MSGCRYPTED, SIGNATURE

```
----BEGIN KEYCRYPTED RSA----
Ks6QTu1hlTvf,CdDzMYESK8p+,U/yCrIYxC9Sx,AToBLF6rBEGN,ZfWWLW
+lTIhG,LK4lCROsimxp,NgKB7Uumb3xH,rFgQutb/Z4U=
----END KEYCRYPTED----

----BEGIN MSGCRYPTED AES256 CBC----
xcLSjn0uHMZKqkOVuZWvOg==,Q47mg7/9AV2eZBxePGqQl9lWn3M2ZNiGL28vPK
DsflbWHVsK3g2MnVzvnPUB2iKD
----END MSGCRYPTED----

----BEGIN SIGNATURE RSA SHA256----
h62LINm9tA5e,Dh52nDFyKqjE,qeghhKVXc+Y6,KEreLBSm1rUr,tQc
+a5rDBRKJ,FtH7ek5TDtmC
----END SIGNATURE----
```

Aufteilung Funktionalität

- Hauptfiles
  - hskeygenerator, hsencrypt, hsdecrypt
- Zwischenfiles
  - KeyCrypted, MsgCrypted, Signature
- Weitere
  - Msg, RSA, RSAKey, SHA256, AES256, Pad, Serial, BlockModes, Base64

Code - Hauptfiles

- Optionen als Argumente beim Programmaufruf

```
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
        let sym = args !! 2
```

- Optionen interaktiv

```
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
```

Code – RSAKey/Random

```
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)
            (getQ, newGen') = (randomR (2^35, 2^37) newGen)
            d = genD p q
            n = p*q
```

Code – Hauptfile Encrypt / Msg

```
let (mMsgPart, symkey) = M.genMsgPart rgen sym blockmode plainFileContent
let kMsgPart = K.genMsgPart asym pubkey symkey
let plainS = [kMsgPart,mMsgPart]
let sMsgPart = S.genMsgPart asym privkey hash $ plainS
let msgParts = map (B.pack . show) [kMsgPart,mMsgPart,sMsgPart]
B.writeFile (B.unpack infile ++ "Encrypted") $ B.intercalate "\n\n" msgParts
```

- show msgpart

```
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) ++ "----"
```

- msgpart

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

Code – Zwischenfile/Keycrypted

```haskell
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)]
```

Code – SHA256

```
// Preprocessing, dann Unterteilung
// in 512bit-Chunks
for each chunk
    // break chunk into sixteen 32-bit
    // big-endian words w[0..15]
    // Extend the sixteen 32-bit words
    // into sixty-four 32-bit words

    // Initialize hash value for this chunk:
    a := h0; b := h1; c := h2; d := h3
    e := h4; f := h5; g := h6; h := h7

    // Main loop:
    for i from 0 to 63
        // Einiges an Bitoperationen, dann:
        h := g; g := f; f := e; e := d
        d := c; c := b; b := a; a := temp

    // Add this chunk's hash to result so far:
    h0 := h0 + a; h1 := h1 + b
    h2 := h2 + c; h3 := h3 + d
    h4 := h4 + e; h5 := h5 + f
    h6 := h6 + g; h7 := h7 + h

// Produce the final hash value:
digest := hash := h0 append h1 append h2
append h3 append h4 append h5 append h6 append
h7
```

```haskell
hash :: B.ByteString -> B.ByteString
hash msg = B.concat $ map w2b h
    where
        h = foldl perchunk hs preprocessed
        preprocessed = chunks $ preprocess msg
```

```haskell
perchunk :: [Word32] -> B.ByteString -> [Word32]
perchunk curhash chunk = zipWith (+) curhash looped
```

```haskell
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
```

Demo

- Erzeugung Nachricht
- Erzeugung Keys
- Verschlüsselung
- Entschlüsselung