Using crypto in Haskell

Sharif Olorin <sio@tesser.org>

Ambiata

August 23, 2017

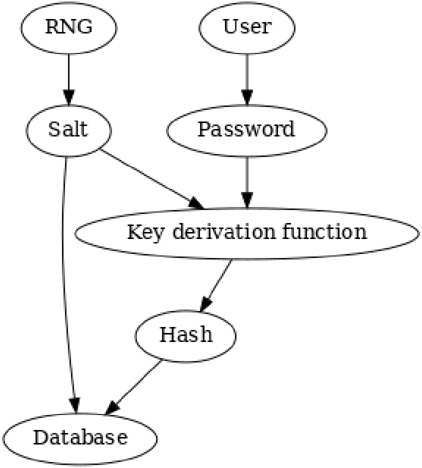
```
#include <stddisclaimer.h>
```

```
# cloc usr/src/openssl | head

Language     files      blank    comment     code
----------------------------------------------------------
C              867      33658      33632    249878

# cloc usr/src/hs-tls | head

Language     files      blank    comment     code
----------------------------------------------------------
Haskell        69        1393      1199      8518
```

...so why so many terrible libraries?

A basic authentication framework

- Securely store user credentials.
- Implement authentication without leaking data.


RNG
User
Salt
Password
Key derivation function
Hash
Database

```haskell
newtype User = User Text

newtype Password = Password Text

newtype Salt = Salt ByteString

newtype Hash = Hash ByteString

data Credential = Credential !Salt !Hash

data Verification = Verified | NotVerified
```

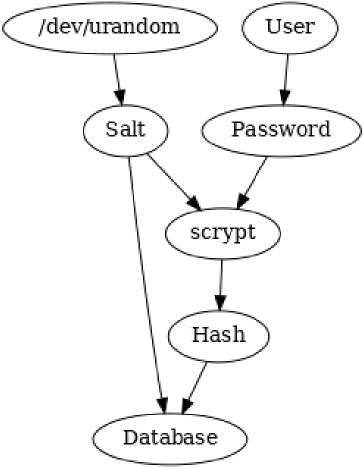
```haskell
authenticate :: User -> Password -> IO Verification

register :: User -> Password -> IO ()
register :: User -> Password -> IO ()
register username pass = do
  salt <- readEntropy
  storeUser salt $ hash salt pass
```

```haskell
authenticate :: User -> Password -> IO Verification
authenticate username pass =
  lookupUser username >>= \(mu :: Maybe Credential) ->
    case mu of
      Just cred ->
        pure $ verify pass cred
```

```haskell
authenticate :: User -> Password -> IO Verification
authenticate username pass =
 lookupUser username >>= \(mu :: Maybe Credential) ->
  case mu of
   Just cred ->
    pure $ verify pass cred
   Nothing ->
    pure $ verify "" fakeCred
```

```haskell
authenticate :: User -> Password -> IO Verification
authenticate username pass =
  lookupUser username >>= \(mu :: Maybe Credential) ->
    case mu of
      Just cred ->
        verify pass cred
      Nothing -> do
        _ <- verify pass fakeCred
        pure NotVerified
```

```haskell
verify :: Password -> Credential -> IO Verification
verify pass (Credential salt h) = do
  h' <- hash salt pass
  case h `constEq` h' of
    True ->
      pure Verified
    False ->
      pure NotVerified
```


/dev/urandom
User
Salt
Password
scrypt
Hash
Database

On comparison

```haskell
-- | Message authentication code wrapper.
newtype MAC = MAC ByteString
  deriving (Eq, Show)

-- | Verify origin and integrity of message.
authenticate :: MAC -> Message -> Key -> Verification
authenticate mac msg key =
  let
    mac' = computeMAC msg key
  in
  case mac == mac' of
    True ->
      Verified
    False ->
      NotVerified
```

```
eq :: ByteString -> ByteString -> Bool
eq a@(PS fp off len) b@(PS fp' off' len')
  | len /= len'                = False
  | fp == fp' && off == off' = True
  | otherwise                  = compareBytes a b == EQ

compareBytes :: ByteString -> ByteString -> Ordering
compareBytes (PS fp1 off1 len1) (PS fp2 off2 len2) =
    accursedUnutterablePerformIO $
      withForeignPtr fp1 $ \p1 ->
      withForeignPtr fp2 $ \p2 -> do
        i <- memcmp (p1 `plusPtr` off1)
                    (p2 `plusPtr` off2)
                    (min len1 len2)
        return $! case i `compare` 0 of
                    EQ  -> len1 `compare` len2
                    x   -> x
```

```c
bool const_cmp(uint8_t *buf1,
               size_t s1,
               uint8_t *buf2,
               size_t s2) {
        size_t i;
        uint8_t acc = 0;
        if (s1 != s2) {
                return FALSE;
        }
        for (i = 0; i < s1; i++) {
                acc |= buf1[i] ^ buf2[i];
        }
        if (acc == 0) {
                return TRUE;
        }
        return FALSE;
}
```

Testing

- Property tests for everything.
- But especially C code.
- Good generator coverage is essential.
- Timing tests.
- Consider supplementing with statistical tests where appropriate.

```haskell
prop_verify_timing =
  forAll (arbitrary :: Password) $ \pass ->
    (t, r) <- run . withCPUTime $ verify pass fakeCred
    stop $ conjoin [
        r === NotVerified
      , t >= minHashTime
      ]
  where
    withCPUTime a = do
      t1 <- liftIO getCPUTime
      r <- a
      t2 <- liftIO getCPUTime
      pure (t2 - t1, r)
```

Some sample “red flags” in crypto packages

- Using a bespoke implementation of an established primitive for no good reason.
- Trivial (or missing) testsuite.
- Over-enthusiastic use of ‘unsafePerformIO‘.
- Derived ‘Eq‘ instances for authentication codes or signatures.
- Unsanitary combination of entropy sources (e.g., sequential reading and concatenation).

Suspicious...

```
verifyCredential
  :: Password
  -> Salt
  -> Hash
  -> Verification
```

RUN

```
verifyCredential
  :: ByteString
  -> ByteString
  -> ByteString
  -> Bool
```

Morals?

- Timing is a side-effect.
- Laziness is not always your friend.
- It’s possible to have too many pure functions.
- C is not literally Satan...
- ...as long as you have QuickCheck.

Thanks!

https://github.com/olorin/slides

<sio@tesser.org>