Secure Programming via Libraries

A library for information-flow in Haskell (side-effects)

Alejandro Russo (russo@chalmers.se)

Escuela de Ciencias Informáticas (ECI) 2011
UBA, Buenos Aires, Argentina

CHALMERS

Side-effects?

[Russo, Claessen, Hughes 08]

- What about trying to do side-effects inside of the security monad?

```
Sec H (IO ())
```

- Would you run the IO computation?

Malicious Code

- The following code shows malicious side-effects

```
func :: Sec H Char -> Sec H (IO ())
func sec_c = do c <- sec_c
                return $ do putStrLn "The secret is gone!"
                            writeFile "PublicFile" [c]
```

- Important Haskell feature for security: ***by looking the type of a piece of code, it is possible to determine if it performs side-effects!***

Side-effects and Sec

- Trustworthy code

```
module SideEffectsSecT where

import Data.Char
import SecLib.LatticeLH
import SecLib.Trustworthy

import SideEffectsSecU (func) -- Import the untrustworthy function unsafe

secret :: Sec H Char     -- This is the secret to be manipulated by the
                         -- untrustworthy code
secret = return 'X'

execute :: IO ()
execute = reveal $ func secret
```

Side-effects and Sec

- Untrustworthy code

```
module SideEffectsSecU where

import Data.Char
import SecLib.LatticeLH
import SecLib.Untrustworthy


-- Do not execute IO operations inside Sec!
func :: Sec H Char -> Sec H (IO ())
func sec_c = do c <- sec_c
                return $ do putStrLn "The secret is gone!"
                            writeFile "PublicFile" [c]
```

Little Quiz

- What about programs of the following type?

NO

```
Sec H (IO (Sec L Int))
```

NO

```
Sec H (Sec L (IO Char))
```

NO

```
Sec L (Sec H (IO ()))
```

YES

```
Sec L (IO (Sec H Char))
```

Side-effects?

[Russo, Claessen, Hughes 08]

- What about trying to do side-effects inside of the security monad?

```
Sec H (IO ())
```

- We do not know if the side-effects are safe to perform
- What should we do?
- IO is a monad that encapsulates side-effects
- **Let us make another monad that encapsulates safe side-effects!**

Monad SecIO

- It is a monad that performs secure side-effects
  - Side-effects that preserve confidentiality!

It is a computation that
a) can write to security locations above s and
b) which **result, of type a, has confidentiality level at least a**

```
data SecIO s a -- abstract
instance Monad (SecIO s)
```

Monad SecIO

- We show how it works for files
  - It also works for references and sockets (check the library)

```
data SecIO s a
```

It is a computation that
a) can write to security locations above s and
b) which **result, of type a, has confidentiality level at least a**

```
c1 :: SecIO H Int
```

It can write to secret files and returns a secret integer

```
c2 :: SecIO L (Sec H Int)
```

It can write to public and secret files and returns a secret integer

```
c3 :: SecIO L Int
```

It can write to public and secret files and returns public integer

API for SecIO

```
data SecIO s a
instance Monad (SecIO s)

type File s
```

It is a file which content has confidentiality level s

The secure version of the operations to read and write files in Haskell

```
readFileSecIO  :: File s -> SecIO s' (Sec s String)

writeFileSecIO :: File s -> String -> SecIO s ()
```

```
readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()
```

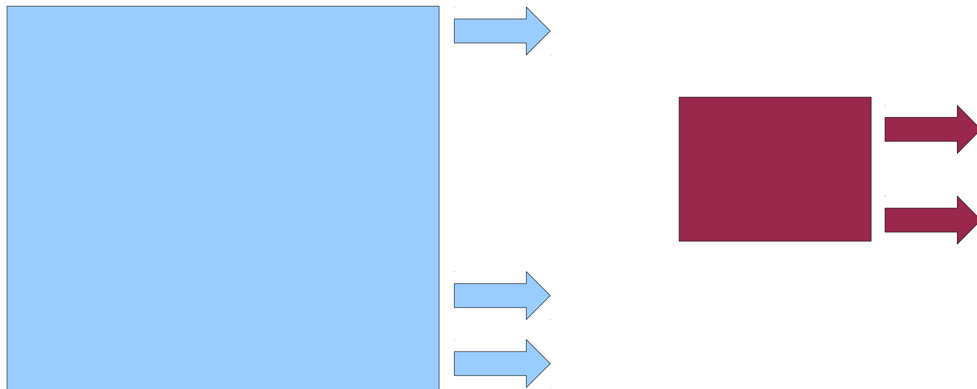
API for SecIO

```
value :: Sec s a -> SecIO s a
```

It pushes any pure secure value into a side-effectful computation

```
plug :: Less sl sh =>
        SecIO sh a -> SecIO sl (Sec sh a)
```

It plugs computations that perform side-effects at a higher level into computations that perform side-effect into lower levels



```
-- Used in trustworthy code
revealSecIO :: SecIO s a -> IO (Sec s a)
```

Small Example

- We want to write a function that copy contents of files
- We do not want *the function to leak information*
- The function should allow copying:
  - a public file into another public file,
  - a secret file into another secret file,
  - a public one into a secret one
- It must avoid *copying a secret file into a public one*
- We will use the library to get the security part of the code right!

Small Example: Trustworthy code

```
module CopyT where

import SecLib.LatticeLH
import SecLib.Trustworthy

import CopyU (copy)

secret_file :: File H
secret_file = mkFile "SecretFile"

public_file :: File L
public_file = mkFile "PublicFile"

trusted_copy :: Less s s' => (File s -> File s' -> SecIO s' ())
                          -> File s -> File s' -> IO ()

trusted_copy copy_func fs fs' = do sec <- revealIO $ copy_func fs fs'
                                   return $ reveal sec
execute :: IO ()
execute = trusted_copy copy public_file secret_file
```

It imports the untrustworthy copying function

It establishes the confidentiality level of the files

Type for the untrustworthy copying function

It executes the untrustworthy function. Does it preserve confidentiality?

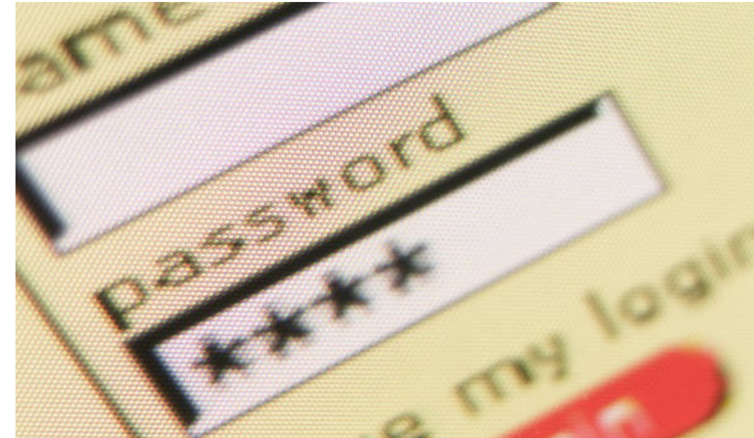
Small Example: Untrustworthy code

```
module CopyU where

import SecLib.LatticeLH
import SecLib.Untrustworthy

copy :: Less s s' => File s -> File s' -> SecIO s' ()
copy file1 file2  = do sec_str <- readFileSecIO file1
                       str     <- value (up sec_str)
                       writeFileSecIO file2 str
```

It provides a function with the type requested by module CopyT

- Can you write the function above in such a way that copies the content of a secret file into a public one?
  - Try it out!
- The type-checker will not allow it

Constructing a Secure Password Administrator



- Linux Password Administrator
  - /etc/passwd

```
bjorn:x:1003:100::/home/andrei:/bin/bash
hana:x:500:100::/home/tsa:
josef:x:1006:100::/home/john:/bin/bash
```

  - /etc/shadow

```
bjorn:$1$0ID5oZxB$0tdKR1VQWQlkJR1Uj7na0:13397:0:99999:7:::
hana:$1$.28fO/M9$aaNMN4SWEKZiGPYoEq9996:13460:0:::::0
josef:$1$UP1uD.28$hi3vYEa20.zgWYNVN/Lq81:13539:0:99999:7:::
```

- Linux Shadow Password HOWTO: Adding shadow support to a C program

> Adding shadow support to a program is actually fairly straightforward. The only problem is that the program must be run by root (or SUID root) in order for the the program to be able to **access** the /etc/shadow file.

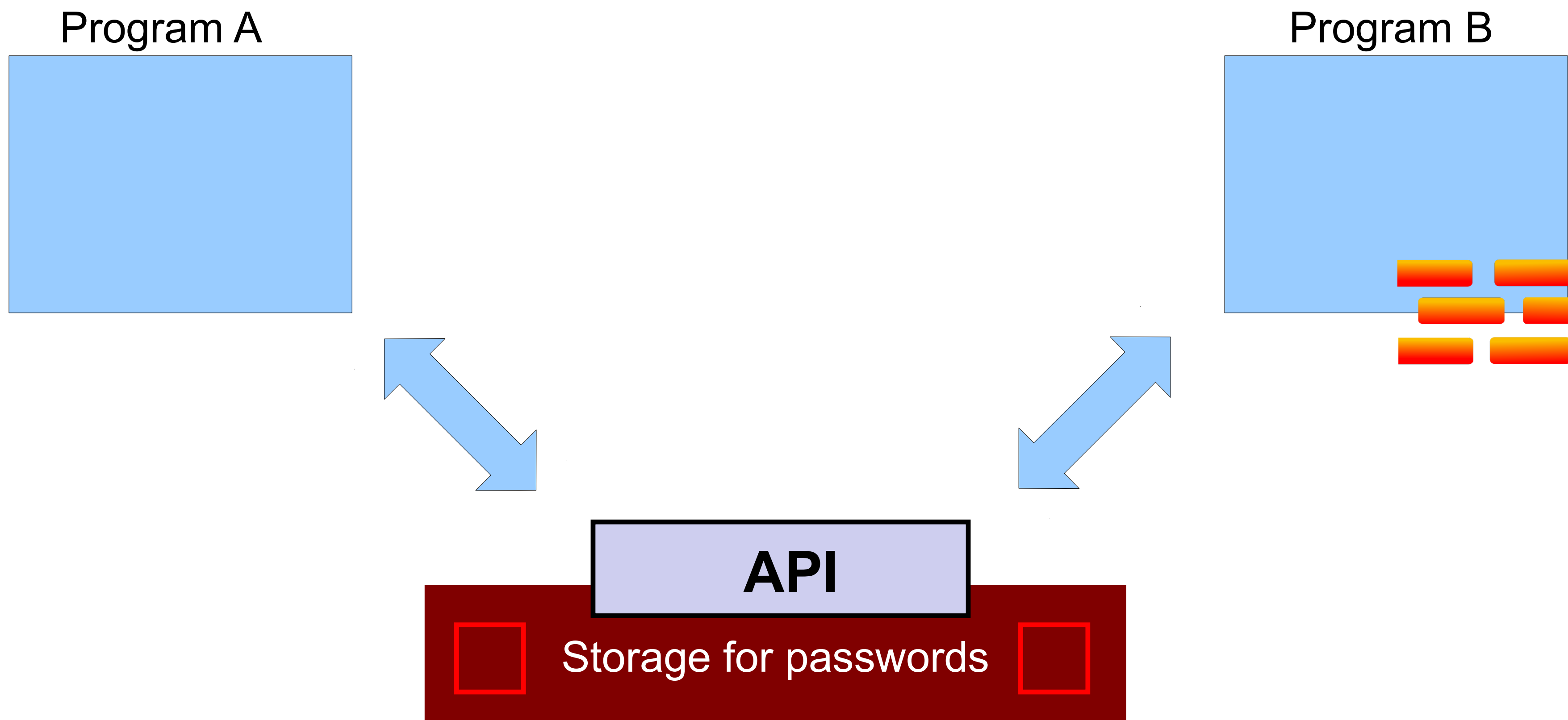
Password Administrator

- What are the security concerns?
  - Give root permission to a program that only needs to authenticate a user
  - Password might be leaked (un)intentionally (dictionary attacks)
- Linux provides an API to access `/etc/shadow`

```
#ifdef HAS_SHADOW
#include <shadow.h>
#include <shadow/pwauth.h>
#endif
```

- File `/etc/shadow` can be accessed by other means (not only by the API)
- We assume the opposite (e.g. in kernel space, remote server, etc)

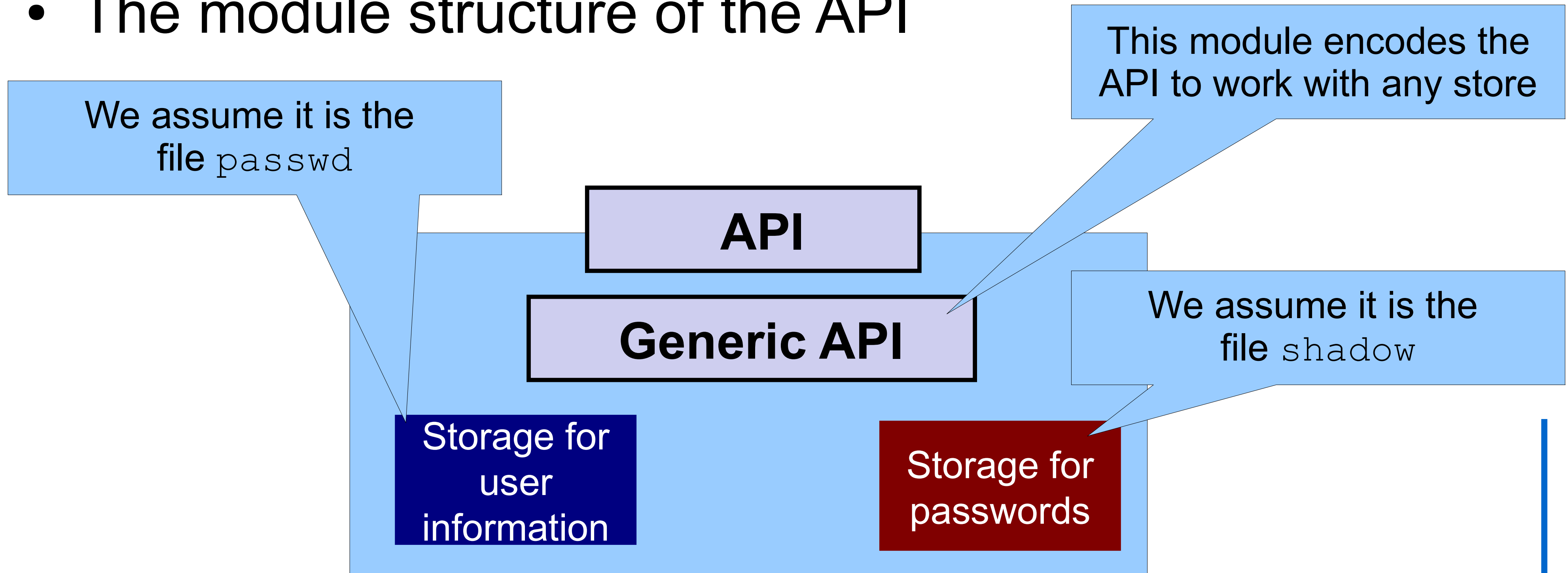
More graphically



	Required root access	Confidentiality
C program + shadow.h	**YES**	**NO**
Haskell program + SecLib	**NO**	**YES**

Password Administrator

- Let us implement the API in Haskell
  - Recall that shadow password are only accessible via the API
- The module structure of the API



GenericAPI

```
module GenericAPI
  ( getSpwdName, putSpwd, getNames )
where

import Spwd
```

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Cypher }
```

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
```

Store for user information

Store for password

```
putSpwd :: FilePath -> Spwd -> IO ()
```

Store for password

```
getNames :: FilePath -> IO [Name]
```

Store for user information

API

```
module API
  (
        getSpwdName
      , putSpwd
      , getNames
  )
where

import Spwd
import qualified GenericAPI as GenericAPI (getSpwdName, putSpwd, getNames)

passwd :: FilePath
passwd = "./passwd"

shadow :: FilePath
shadow = "./shadow"

getSpwdName :: Name -> IO (Maybe Spwd)
getSpwdName = GenericAPI.getSpwdName passwd shadow

putSpwd :: Spwd -> IO ()
putSpwd = GenericAPI.putSpwd shadow

getNames :: IO [Name]
getNames = GenericAPI.getNames passwd
```

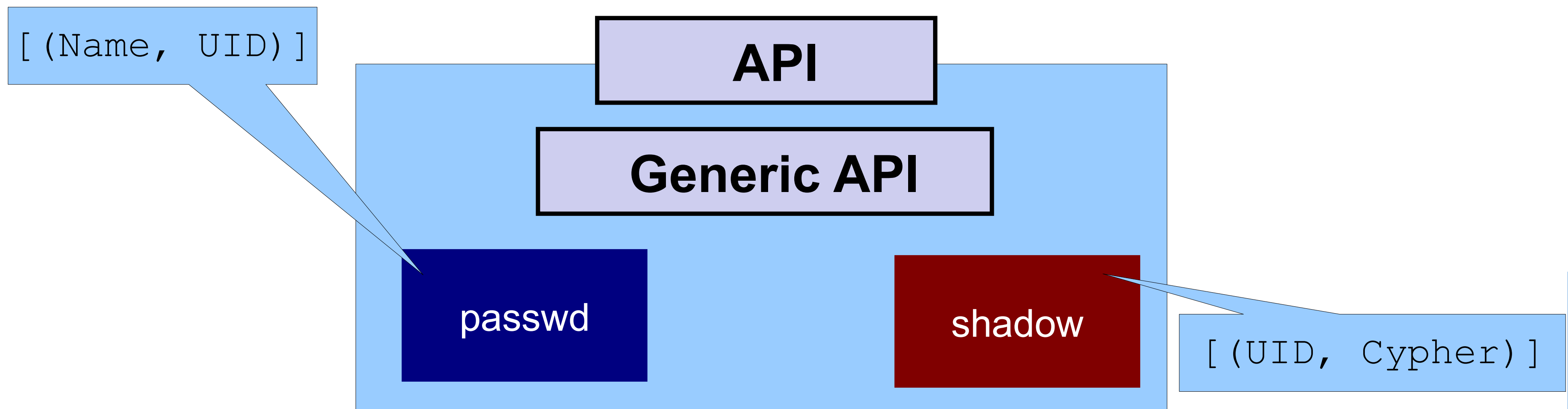
Store of user information

Store for passwords

The module applies the generic API interface to specific stores

Implementing getSpwdName

- Some internals of the implementation
  - It is not the most advance password administrator
  - You can do it better!
  - It is only for pedagogical purposes



Implementing getSpwdName

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing  -> return Nothing
               Just id   -> return $ Just (case lookup id sh of
                                    Nothing -> error "Error!"
                                    Just c  -> Spwd { uid = id ,
                                                      cypher = c})
```

pw :: [(Name, UID)]

sh :: [(UID, Cypher)]

```
parse_passwd :: FilePath -> IO [(Name,UID)]
parse_shadow :: FilePath -> IO [(UID,Cypher)]
```

Using the API

- Programs using that API can build up more sophisticated functions

```
module Auxiliaries where

import Data.Maybe
import Spwd
import API

-- Function to suggest a user name
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                     $ filter (\(_,b) -> b == False)
                              [ (name', name' `elem` ns)
                               | n <- [0..], let name'= name ++ show n]
```

- How does it work?
  - User “david” is in the system, then it suggests “david0”. If “david0” is in the system, then it suggests “david1”, etc.
- Could someone implement some unintended behaviour in this function?

Using the API

```
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       f ns
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b == False)
                                     [ (name', name' `elem` ns)
                                      | n <- [0..], let name' = name ++ show n ]


f :: [Name] -> IO ()
f ns = do lst <- f' ns
          writeFile "foo" (show lst)
          return ()

       where f' []      = return []
             f' (n:ns) = do spwd <- getSpwdName n
                            lst  <- f' ns
                            return $ (n, (cypher $ fromJust spwd)) : lst
```

What is this?

It is copying the passwords to a file

Modifying the API?

- We see two versions of `suggest_name`
  - Built on the password adminstrator API
- To identify the one violating confidentiality, we looked at the code and think for a bit
  - *Code revision*
- Let us use the `SecLib` to automatically enforce confidentiality
  - In that manner, we do not need to do code review!
  - Of course, we still need to do testing for correctness

Marking the Secret Data

- How do we start?
  - Indicating which are the secrets (passwords) in our program

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Cypher }
```

GenericAPI: Secure Version

```
module GenericAPI
  ( getSpwdName, putSpwd, getNames )
where
import SecLib.LatticeL
import SecLib.Untrustwo
import Spwd
```

```
type UID    = Int
type Cypher = String
type Name   = String

data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }
```

```
-- getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
-- putSpwd :: FilePath -> Spwd -> IO ()
-- getNames :: FilePath -> IO [Name]

getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)
```

Store for user information

Store for password

This function does not write to any file

```
putSpwd :: File H -> Swpd -> SecIO H ()
```

Store for password

This function writes to a secret file

```
getNames :: File L -> SecIO s [Name]
```

This function does not write to any file

API: Secure Version

```
module API
  (
        getSpwdName
      , putSpwd
      , getNames
  )
where

import Spwd
import qualified GenericAPI as GenericAPI (getSpwdName, putSpwd, getNames)

import SecLib.Trustworthy
import SecLib.LatticeLH

passwd :: File L
passwd = mkFile "./passwd"

shadow :: File H
shadow = mkFile "./shadow"

getSpwdName :: Name -> SecIO s (Maybe Spwd)
getSpwdName = GenericAPI.getSpwdName passwd shadow

putSpwd :: Spwd -> SecIO H ()
putSpwd = GenericAPI.putSpwd shadow

getNames :: SecIO s [Name]
getNames = GenericAPI.getNames passwd
```

This module is trustworthy

It assigns the security level of each store. That is why this module is trustworthy!

As the unsecure version but it returns a `SecIO` instead as `IO`

Summarizing

- We have a new API

```
data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }

getSpwdName :: Name -> SecIO s (Maybe Spwd)

putSpwd :: Spwd -> SecIO H ()

getNames :: SecIO s [Name]
```

- Any program that wants to use the API needs to use `SecLib`
- Confidentiality is then provided!
    - No need for root permission

Using the Secure API

- Remember the *well-behaved* function to suggest a user name?
  - Let us try to reimplemented using the secure API

```
module Auxiliaries where

import Data.Maybe
import Spwd
import API

-- Function to suggest a user name
suggest_name :: Name -> SecIO s Name
suggest_name name =
    do ns <- getNames
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b == False)
                                     [ (name', name' `elem` ns)
                                       | n <- [0..], let name'= name ++ show n]
```

Using the Secure API

- Remember the bad-*behaved* function to suggest a user name?

```
suggest_name :: Name -> IO Name
suggest_name name =
    do ns <- getNames
       f ns
       case (name `elem` ns) of
            False -> return name
            True  -> return $ fst $ head
                            $ filter (\(_,b) -> b ==
                                     [ (name', n      elem` ns)
                                      | n <- [0..], let name' = name ++ show n ]


f :: [Name] -> IO ()
f ns = do lst <- f' ns
          writeFile "foo" (show lst)
          return ()

       where f' []     = return []
             f' (n:ns) = do spwd <- getSpwdName n
                            lst  <- f' ns
                            return $ (n, (cypher $ fromJust spwd)) : lst
```

It will not work!

It is not possible to write a value of type `Sec H Cypher` into a public file

The result of f' is a list of type `[(Name, Sec H Cypher)]` instead of `[(Name, Cypher)]`

Implementing the Secure API (getSpwdName)

- Recall

```
data Spwd = Spwd { uid :: UID, cypher :: Sec H Cypher }

getSpwdName :: Name -> SecIO s (Maybe Spwd)

putSpwd :: Spwd -> SecIO H ()

getNames :: SecIO s [Name]
```

- We set up the types of the secure API
- How do we implement it?
  - We will see how to do one of the primitives (the rest is homework!)

Implementing Secure Version of getSpwdName

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
getSpwdName :: File H FilePath -> Name -> Sec IO s (Maybe Spwd)
getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing   -> return Nothing
               Just id    -> return $ Just (case lookup id sh of
                                            Nothing  -> error "Error!"
                                            Just c  -> Spwd { uid = id ,
                                                              cypher = c})
```

pw :: [(Name, UID)]

sh :: Sec H [(UID, Cypher)]

```
parse_passwd :: FilePath -> IO [(Name,UID)]
parse_shadow :: FilePath -> IO [(UID,Cypher)]
parse_shadow :: FilePath -> Sec IO (Sec H [(UID,Cypher)])
```

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: FilePath -> FilePath -> Name -> IO (Maybe Spwd)
getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing   -> return Nothing
               Just id    -> return $ Just (case lookup id sh of
                                          Nothing  -> error "Error!"
                                          Just c   -> Spwd { uid = id ,
                                                         cypher = c})
```

```
pw :: [(Name, UID)]
```

```
sh :: Sec H [(UID, Cypher)]
```

```
parse_passwd :: FilePath -> IO [(Name,UID)]
parse_shadow :: FilePath -> IO [(UID,Cypher)]
```

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)

getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing   -> return Nothing
               Just id    -> return $ Just (case lookup id sh of
                                         Nothing  -> error "Error!"
                                         Just c  -> Spwd { uid = id ,
                                                          cypher = c})
```

pw :: [(Name, UID)]

sh :: Sec H [(UID, Cypher)]

```
parse_passwd :: File L -> SecIO s [(Name,UID)]
parse_shadow :: File H -> SecIO s (Sec H [(UID,Cypher)])
```

We need to adapt these functions as well! (homework)

Implementing Secure Version of getSpwdName

```
getSpwdName :: File L -> File H -> Name -> SecIO s (Maybe Spwd)
getSpwdName passwd shadow user =
      do  pw  <-  parse_passwd passwd
          sec_sh  <-  parse_shadow shadow
          case lookup user pw of
               Nothing  -> return Nothing
               Just id   -> return $
                      Just $ Spwd { uid = id ,
                                    cypher = do sh <- sec_sh
                                                case lookup id sh of
                                                  Nothing -> error "Error!"
                                                  Just c  -> return c }
```

pw :: [(Name, UID)]

SecIO

sh :: Sec H [(UID, Cypher)]

Sec H

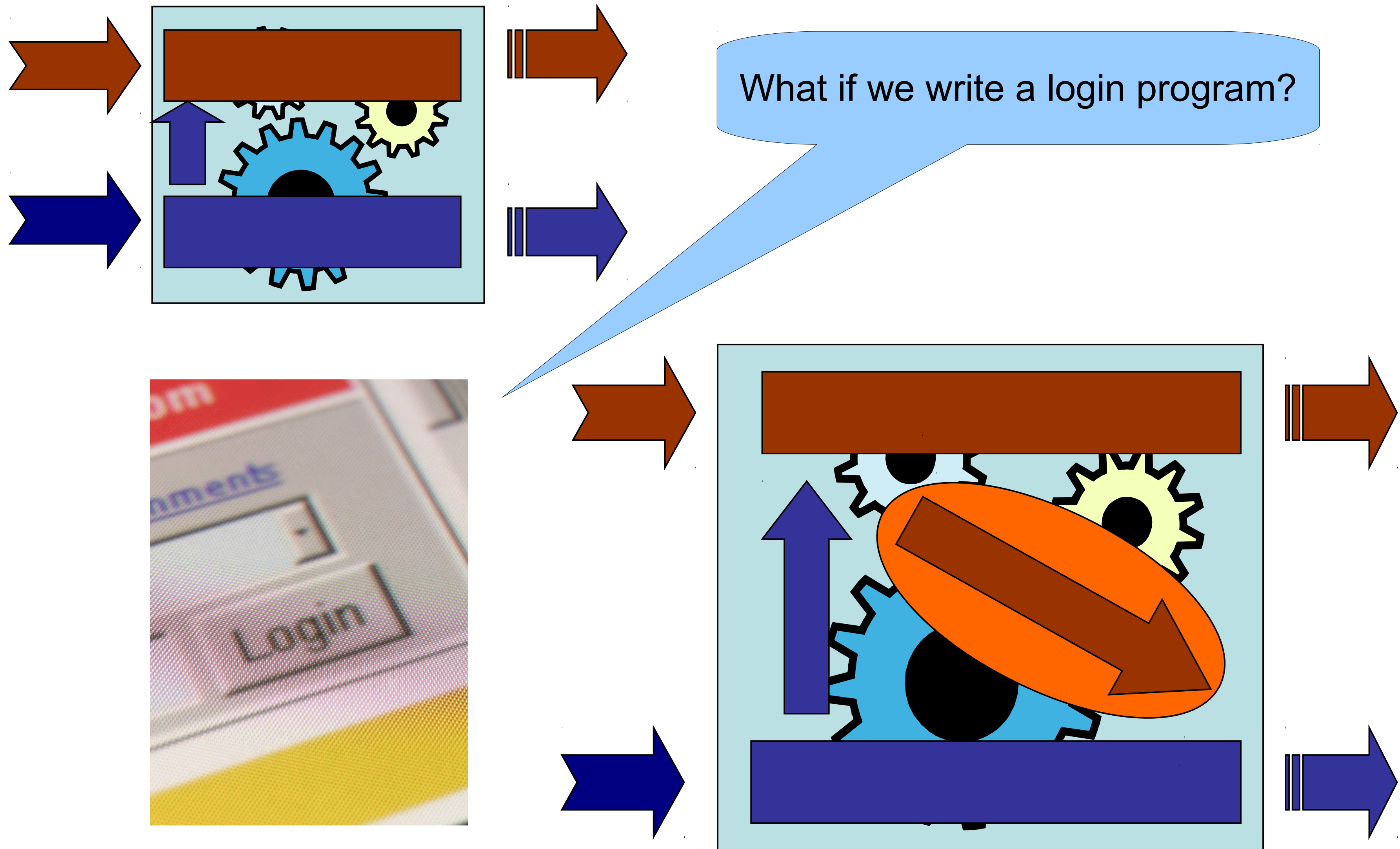
```
parse_passwd :: File L -> SecIO s [(Name,UID)]
parse_shadow :: File H -> SecIO s (Sec H [(UID,Cypher)])
```

We need to adapt these functions as well! (homework)

General Guidelines

- Take a non-secure version of some code that you wrote
- Indicate in your program (datatypes and API) which data is confidential
  - As we did with `Spwd` and `getSpwdName`
- Indicate the confidentiality level of your external resources
  - As we did with files `passwd` and `shadow`
- Once the types are in place (`Sec` **`H`**, `SecIO s`, `SecIO L`) just adapt the code to type-check!

Declassification



Declassification

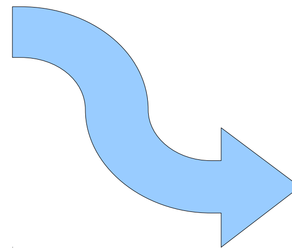
[Sabelfeld, Sands 07]

- Login program: it is necessary to leak information that depends on secrets
  - cypher spwd == input_user
- Dimensions and principles of declassification
  - **What** information can be leak?
  - **When** can information be leaked?
  - **Where** in the program is safe to leak information?
  - **Who** can leak information?
- How to be certain that our programs leak what they are supposed to leak?

Declassification in the Library

- The library handle different kind of *declassificaiton policies*
- *Declassification policies are programs!*
  - Trustworthy code defines them
  - Controlled at run-time

```
module DeclPolicies where

import SecLib.Trustworthy

...
```



```
module X where

import SecLib.Untrustworthy

...
```

Declassification in the Library

- The library defines *combinators* for different declassification policies (**what**, **when**, **who**)
  - It is possible to combine dimension of declassification
  - “When event X happens, you can declassify information Y provided that the code is running by Z”
- In the course: **what**

Escape Hatches

- Declassification is performed by functions
  - Terminology: *escape hatches* [Sabelfeld, Myers 04]
- In the library: a escape hatch is just a function of type

```
Less sl sh => Sec sh a -> SecIO s (Sec sl b)
```

It indicates that information can flow to the lower levels in the lattice

We leave this type “free” (see later)

About the Type for Espace Hatches

- Why `SecIO`?

```
Less sl sh => Sec sh a -> SecIO s (Sec sl b)
```

There is an **internal state** that determines if declassication can proceed

- Why `s` is “free”?
  - The state might change when applying a escape hatch. However, that change can only be *observed if declassification fails or succeed.*
  - *Since we are termination-insensitive is like no-effect is produced*

Some Declassification Combinators

- Base combinator
  - It always succeed in declassifying

```
hatch :: Less sl sh =>
         (a -> b) -> Sec sh a -> SecIO s (Sec sl b)
```

It applies an arbitrary function

- What combinator (how often)

Escape hatch

```
ntimes :: Int -> (Sec sh a -> SecIO s (Sec sl b))
          -> IO (Sec sh a -> SecIO s (Sec sl b))
```

How many times can be applied per run

It creates a counter

Module Login (Trustworthy)

- This module sets up
  - The confidentiality level of the resources (stdin/stdout)
  - The escape hatches
- It calls the untrustworthy module that implements the login
  - We assume that the login function provided by the untrustworthy module fulfill its specification, but we want to guarantee that it is also secure.

Module Login (Trustworthy)

```
module Login (login) where

import Spwd
import qualified ULogin as ULogin (login)

import SecLib.Trustworthy
import SecLib.LatticeLH

check :: Sec H (String, Cypher) -> SecIO s (Sec L Bool)
check = hatch (\(input, key) -> input == key)

check3 :: IO (Sec H (String, Cypher) -> SecIO s (Sec L Bool))
check3 = ntimes 3 check

screen :: Screen L
screen = mkScreen ()
```

Escape hatch to declassify if the input provided matches the password

The escape hatch can only be applied, at most, 3 times per run

stdin/stdout is a public channel

Module Login (Trustworthy)

The type of the function provided by the untrustworthy

```
safe_login :: ( Screen L
                -> (Sec H (String, Cypher) -> SecIO s (Sec L Bool))
                -> SecIO L ()
              )
              -> IO ()

safe_login expected_login = do esc_hatch <- check3
                               run $ expected_login screen esc_hatch
                               return ()

login = safe_login ULogin.login
```

It provides with the screen and escape hatch to the untrustworthy login

Module Ulogin (Untrustworthy)

```
login :: Screen L
         -> (Sec H (String, Cypher) -> SecIO L (Sec L Bool))
         -> SecIO L ()
login scr eh
   = do putStrLnSecIO scr "Welcome!"
        putStrSecIO scr "login:"
        user <- getLineSecIO scr
        spwd <- getSpwdName user
        case spwd of
             Nothing     -> putStrLnSecIO scr "Invalid user!"
             Just spwd   -> do b <- verify 3 spwd scr eh
                               if b then putStrLnSecIO scr "Launching shell!"
                                    else error "Access denied!"
```

- Very similar to a login function written without `SecIO`

Module Ulogin (Untrustworthy)

```
verify 0 _ scr _ =
     do putStrLnSecIO scr "Maximum number of tries reached!"
        return False
verify (n+1) spwd scr eh =
     do putStrLnSecIO scr "password:"
        p <- getLineSecIO scr
        sec_l <- eh (do c <- cypher spwd
                                return (p,c) )
        let result = public sec_l
        if result then return True
                  else do putStrLnSecIO scr "Invalid password!"
                          verify n spwd scr eh
```

It applies the escape hatch

Put together the password and the input provided by the user into `Sec H`

Function login

- What do we know about it?

```
module Login (login) where
```

- It preserves confidentiality (non-interference) but allows to declassify some information
  - Escape hatch
- Login cannot, for example, send the password into a public file
- Login cannot apply the escape hatch more than 3 times
  - Limit the number of bits to be leaked per run

SecLib: Pros

- Provides confidentiality
  - Type-system and abstraction provided by the module system in Haskell
- Only check types and some imports (no code revision)
- Light-weight library (342 LOC)
  - Polymorphic secure code for free!
- Promise to be practical
  - Simple (Monads)
  - Side-effects: files, references, stdin/stdout, etc.
- Support for declassification
  - It is the most experimental part of the library
  - Room for innovation here!

`SecLib`: Cons

- Static security lattice
  - Dynamic security levels?
  - Mutual-distrust environments
- Timing channel
  - Usually a difficult channel to close up
- It relies on Haskell's type-safety (no cheating) and that abstraction is respected (modules system)
  - SafeHaskell is coming soon!