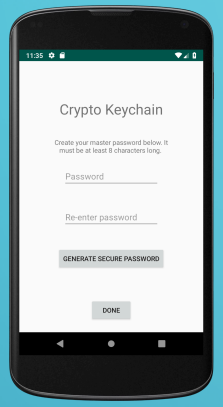
CRYPTO KEYCHAIN

RICHARD GREENBAUM, RUDOLF HERNANDEZ

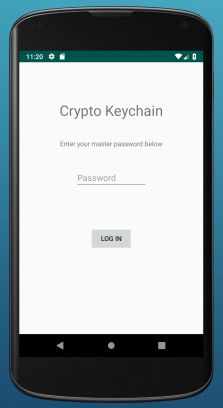
APP FLOW

Splash Screen

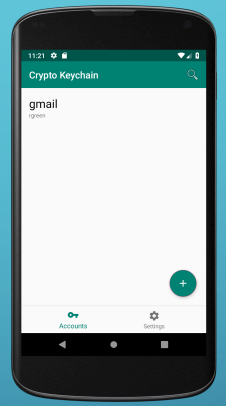
New User



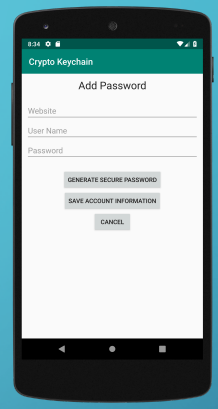
User Login



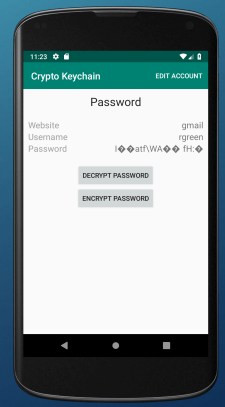
Accounts View



New Account



View/Edit Account



USER CREATION AND AUTHENTICATION

Storing Hash

- Generate hash of master password
- Store the hash in shared preferences

```
private fun hash(password: String) : String {
    var passwordHash :String  = password
    for (x :Int  in 0..5500) {
        Log.d( tag: "middleHash", passwordHash)
        passwordHash = passwordHash.hashCode().toString()
    }
    return passwordHash
}
```

Deriving Key

- Generate random salt
- Create key from password and salt
- Store the salt
- Send key to account view activity

```
private fun createKey(password : String) : ByteArray {
    val random = SecureRandom()
    val salt = ByteArray( size: 256)
    random.nextBytes(salt)
    storeSalt(salt)

    val pbKeySpec = PBEKeySpec(password.toCharArray(),
        salt,  iterationCount: 1324,  keyLength: 256) // 1
    val secretKeyFactory :SecretKeyFactory!  =
        SecretKeyFactory.getInstance( algorithm: "PBKDF2WithHmacSHA1") // 2
    val keyBytes :ByteArray!  =
        secretKeyFactory.generateSecret(pbKeySpec).encoded // 3
      val keySpec = SecretKeySpec(keyBytes, "AES")
    return keyBytes
}
```

USER LOGIN

Verifying Password

- Load the stored password hash from shared preferences
- Generate hash of input password
- Check to see if they are equal

```
var sharedPref : SharedPreferences! =
    PreferenceManager.getDefaultSharedPreferences( context: this)
storedPasswordHash =
    sharedPref.getString( key: "PASSWORD_HASH", defValue: "")

logInBtn.setOnClickListener { it: View!
    var password : String = passwordEt.text.toString()
    var input_password_hash : String = hash(password)
    checkPassword(input_password_hash)
}
```

Deriving Key

- Load the random salt from shared preferences
- Create key from password and salt
- Send key to account view activity

```
private fun createKey(password : String) : ByteArray {
    var sharedPref : SharedPreferences! =
        PreferenceManager.getDefaultSharedPreferences( context: this)
    val salt : ByteArray =
        sharedPref.getString( key: "SALT", defValue: "")
            .toByteArray(Charsets.UTF_8)

    val pbKeySpec = PBEKeySpec(password.toCharArray(),
        salt, iterationCount: 1324, keyLength: 256) // 1
    val secretKeyFactory : SecretKeyFactory! =
        SecretKeyFactory.getInstance( algorithm: "PBKDF2WithHmacSHA1")
    val keyBytes : ByteArray! =
        secretKeyFactory.generateSecret(pbKeySpec).encoded // 3
    return keyBytes
}
```

SECURE PASSWORD GENERATION

- Hard code arrays for each value type
- Create one final array by combining the value type arrays
- Generate random integers in range 0..len(final_array-1) and append the value at that index to the password string

```
fun generatePassword() : String {
    var lower : Array<String> = arrayOf("a","b","c","d","e","f","
    val upper : Array<String> = arrayOf("A","B","C","D","E","F","
    val num : Array<String> = arrayOf("0","1","2","3","4","5","6"

    val final : Array<String> = lower + upper + num

    var password = ""
    for (i : Int in 0..10) {
        Log.d( tag: "num", i.toString())

        var index : Int = (0..final.size-1).random()
        password += final.get(index)
    }
    return password
}
```

```
// Create 16 bytes of random data; package it into an IvParameterSpec object
SecureRandom ivRandom = new SecureRandom();
byte[] iv = new byte[16];
ivRandom.nextBytes(iv);
IvParameterSpec ivSpec = new IvParameterSpec(iv);

// Encrypt password using AES-CBC mode and PKCS-7 Padding Scheme
Cipher cipher = null;
try {
    cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (NoSuchPaddingException e) {
    e.printStackTrace();
}

try {
    cipher.init(Cipher.ENCRYPT_MODE, passwordKey, ivSpec);
} catch (InvalidAlgorithmParameterException e) {
    e.printStackTrace();
} catch (InvalidKeyException e) {
    e.printStackTrace();
}

byte[] temp = password.getText().toString().getBytes(Charset.defaultCharset());
try {
    temp = cipher.doFinal(temp);
} catch (BadPaddingException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
}
```

ENCRYPTING ACCOUNT PASSWORDS

- The 256-bit AES key is retrieved from the app's MainActivity.
- A PRNG is used to create a random IV.
- A Cipher object is initialized and the password is encrypted using AES-CBC mode and PCKS7 padding.

```
Cipher cipher = null;
try {
    cipher = Cipher.getInstance("AES/CBC/PKCS7Padding");
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (NoSuchPaddingException e) {
    e.printStackTrace();
}

// Retrieve the key derived from the user's master password
passwordKey = ((MainActivity)getActivity()).passwordKey;

// Create an IvParameterSpec object from the IV used to encrypt the account information
iv = android.util.Base64.decode(currentAccount.getIv(), Base64.DEFAULT);
Log.w(TAG, currentAccount.getIv());
IvParameterSpec ivSpec = new IvParameterSpec(iv);

// Create Cipher object, decrypt encrypted password
try {
    cipher.init(Cipher.DECRYPT_MODE, passwordKey, ivSpec);
    byte[] decrypted = cipher.doFinal(ciphertext);
    temp = new String(decrypted);
    return temp;
} catch (InvalidAlgorithmParameterException e) {
    e.printStackTrace();
} catch (InvalidKeyException e) {
    e.printStackTrace();
} catch (BadPaddingException e) {
    e.printStackTrace();
} catch (IllegalBlockSizeException e) {
    e.printStackTrace();
}
```

DECRYPTING ACCOUNT PASSWORDS

- The 256-bit AES key is retrieved from the app's MainActivity.
- The IV is retrieved from the app database.
- A Cipher object is initialized and the password is decrypted using the IV that was used to encrypt it.

ACCOUNT SEARCH

- The search function returns all accounts that have the account name or username equal to the search query.

```
@Override
public boolean onQueryTextChange(String newText) {
    newText = newText.toLowerCase();
    ArrayList<Account> searchList = new ArrayList<>();
    for (Account account : mAccounts){
        if (account.getUsername().contains(newText) ||
                account.getAccount_name().contains(newText)){
            searchList.add(account);
        }
    }
    if (searchList.isEmpty()) {
        mRecyclerView.setVisibility(View.GONE);
        emptyView.setVisibility(View.VISIBLE);
    }
    else {
        mRecyclerView.setVisibility(View.VISIBLE);
        emptyView.setVisibility(View.GONE);
    }
    ((AccountsAdapter)mAdapter).setFilter(searchList);
```

ATTACKER MODEL

- Goals of the attacker include:
  - Obtaining the master password
  - Obtaining the password based encryption key
  - Obtaining specific account passwords stored in the app
  - Predicting the output of the secure random password generator
- We assume that the attacker will gain access to the memory of the device and will know the cryptographic primitives that we use
- We also assume that the attacker is able to use the app from the login screen

ATTACK FEASIBILITY

- The state space for an 8 digit master password is ~70^8 password
  - A brute force attack on even the weakest passwords would take thousands of years
- The state space of the encryption key with 256-bit AES is 1.1*10^77
- Predicting the output of the secure password generator would require predicting java's random number generator