Threat 1: Spoofing

**Threat:** A hack app pretending to be Mapbox sends a fake contact to Contact Manager.

**Mitigation:** To address this threat, Mapbox has a secret authorization code that only it and Contact Manager know. This authorization code is combined with the additional data (the contact name, phone number, and address) and encrypted. The encrypted authorization code is then sent to Contact Manager as another extra in the intent.

When Contact Manager receives the intent, it looks for this encrypted authorization code. Contact Manager knows the encryption and decryption methods, and also knows what the authorization code should be once it has been decrypted. Only once Contact Manager verifies the authorization code does it save the contact.

Mapbox code:

**public static** String buildMac(**final** String key, **final** String data1, **final** String data2, **final** String data3) {  
 String concatenatedMessage = data1 + data2 + data3 + ***authCode***;  
 **return** *encrypt*(concatenatedMessage, key);  
}

Contact Manager code:

**public static boolean** macIsValid(**final** String providedMac, **final** String key, **final** String data1, **final** String data2, **final** String data3) {  
 String decryptedMac = *decrypt*(providedMac, key);  
 Log.*i*(**"mac is valid"**, decryptedMac);  
 **return** decryptedMac.equals(data1 + data2 + data3 + ***authCode***);  
}

...

**if** (!Encryptor.*macIsValid*(providedAuthCode, ***encryptionKey***, mvName, mvAddress, mvNumber)) {  
 **throw new** Exception(**"Auth codes didn't match."**);  
}

Threat 2: Tampering

Threat 3: Repudiation

**Threat:** Mapbox API may modify request sent from Mapbox (e.g. change ‘555 main st’ to ‘555 main st, Hartford Connecticut’).

**Mitigation:** This is a repudiation threat and the danger here is that the Mapbox API will modify a request sent from Mapbox and deny that the request was modified. In order to mitigate this, we needed to be able to prove that this request was modified. We decided to create a private, internal log file that records exactly what the user entered and what was returned by the Mapbox API, as well as timestamps for both. This will allow us to see if the request from Mapbox was modified by the Mapbox API and prove it if this modification is denied.

Mapbox code:

*// Callback method invoked when the user selects the "Search" button from the Navigation menu item***public void** navigationSearchButtonClicked(View view, LinearLayout navAddressBar, EditText addressTextBox, MapView mapView) {  
 Log.*i*(***TAG***, **"navigationSearchButton() called"**);  
 Context ctx = getActivity().getApplicationContext();  
 *// Close the keyboard* InputMethodManager imm = (InputMethodManager) getActivity().getSystemService(Context.***INPUT\_METHOD\_SERVICE***);  
 imm.toggleSoftInput(InputMethodManager.***SHOW\_FORCED***, 0);  
  
 *// Get the address and corresponding LatLng* String address = addressTextBox.getText().toString();  
 String currentDateTimeString;  
 Address location = getAddressObjFromAddress(ctx, address);  
 LatLng latAndLng = **null**;  
  
 *// Write a log file consisting of what the user searched and what Mapbox API returned with timestamps  
 // in order to refute repudiation attacks* **try** {  
 FileOutputStream fileout = ctx.openFileOutput(**"MapboxLog.txt"**, ctx.***MODE\_PRIVATE***);  
 OutputStreamWriter outputWriter = **new** OutputStreamWriter(fileout);  
 currentDateTimeString = DateFormat.*getDateTimeInstance*().format(**new** Date());  
 outputWriter.append(**"["** + currentDateTimeString + **"]: "** + **"User entered - "** + address + **"\n\r"**);  
 currentDateTimeString = DateFormat.*getDateTimeInstance*().format(**new** Date());  
 outputWriter.append(**"["** + currentDateTimeString + **"]: "** + **"Mapbox returned - "** + location.getAddressLine(0) + **"\n\r"**);  
 outputWriter.close();  
 }  
 **catch** (Exception e) {  
 e.printStackTrace();  
 }

Threat 4: Information Disclosure

**Threat:** A hack app intercepts contact information sent by Mapbox.

**Mitigation:** This threat is related to Threat 1: Spoofing. We cannot prevent hack apps from intercepting our data (that’s Android’s job), but we can ensure that any hack app that does intercept our data at can’t decrypt it.

We decided to use the AES method of encryption in order to perform this task. Both Contact Manager and Mapbox have the same algorithms for encrypting and decrypting, and both have been provided our key. Before Mapbox adds the contact data to the intent, it encrypts it. Then, when Contact Manager receives the data, it decrypts it before adding the contact.

Mapbox / Contact Manager code:

**public class** Encryptor {  
 **public static** String encrypt(**final** String unencryptedMessage, **final** String hexKey) {  
 **try** {  
 *// Convert message and key to bytes* **final byte**[] hexKeyAsBytes = Base64.*decode*(hexKey, Base64.***DEFAULT***);  
 **final byte**[] encodedMessage = unencryptedMessage.getBytes(Charset.*forName*(**"UTF-8"**));  
  
 *// Get the cipher, block size, and secret key spec* **final** Cipher cipher = Cipher.*getInstance*(**"AES/CBC/PKCS5Padding"**);  
 **final int** blockSize = cipher.getBlockSize();  
 **final** SecretKeySpec secretKeySpec = **new** SecretKeySpec(hexKeyAsBytes, **"AES"**);  
  
 *// Build a random IV* **final byte**[] ivData = **new byte**[blockSize];  
 **final** IvParameterSpec ivParameterSpec = *buildIvParameterSpec*(ivData);  
  
 *// Encrypt the message* cipher.init(Cipher.***ENCRYPT\_MODE***, secretKeySpec, ivParameterSpec);  
 **final byte**[] encryptedMessage = cipher.doFinal(encodedMessage);  
  
 *// Concatenate the IV and the encrypted message* **final int** messageLength = ivData.**length** + encryptedMessage.**length**;  
 **final byte**[] ivAndEncryptedMessage = **new byte**[messageLength];  
 System.*arraycopy*(ivData, 0, ivAndEncryptedMessage, 0, blockSize);  
 System.*arraycopy*(encryptedMessage, 0, ivAndEncryptedMessage, blockSize, encryptedMessage.**length**);  
  
 *// Return the result as a string* **return** Base64.*encodeToString*(ivAndEncryptedMessage,Base64.***DEFAULT***);  
 } **catch** (InvalidKeyException e) {  
 **throw new** IllegalArgumentException(**"Key argument is not a valid AES key: "** + e.getMessage(), e);  
 } **catch** (GeneralSecurityException e) {  
 **throw new** IllegalStateException(**"Unexpected exception during encryption: "** + e.getMessage(), e);  
 }  
 }  
  
 **public static** String decrypt(**final** String ivAndEncryptedMessage, **final** String hexKey) {  
 **try** {  
 *// Covert message and key to bytes* **final byte**[] hexKeyAsBytes = Base64.*decode*(hexKey, Base64.***DEFAULT***);  
 **final byte**[] encodedIvAndEncryptedMessage = Base64.*decode*(ivAndEncryptedMessage, Base64.***DEFAULT***);  
  
 *// Get the cipher, block size, and secret key spec* **final** Cipher cipher = Cipher.*getInstance*(**"AES/CBC/PKCS5Padding"**);  
 **final int** blockSize = cipher.getBlockSize();  
 **final** SecretKeySpec secretKeySpec = **new** SecretKeySpec(hexKeyAsBytes, **"AES"**);  
  
 *// Get the IV* **final byte**[] ivData = **new byte**[blockSize];  
 System.*arraycopy*(encodedIvAndEncryptedMessage, 0, ivData, 0, blockSize);  
 **final** IvParameterSpec ivParameterSpec = **new** IvParameterSpec(ivData);  
  
 *// Get the encrypted message* **final int** encryptedMessageLength = encodedIvAndEncryptedMessage.**length** - blockSize;  
 **final byte**[] encryptedMessage = **new byte**[encryptedMessageLength];  
 System.*arraycopy*(encodedIvAndEncryptedMessage, blockSize, encryptedMessage, 0, encryptedMessage.**length**);  
  
 *// Decrypt the message* cipher.init(Cipher.***DECRYPT\_MODE***, secretKeySpec, ivParameterSpec);  
 **final byte**[] encodedMessage = cipher.doFinal(encryptedMessage);  
  
 *// Return the result as a string* **return new** String(encodedMessage, Charset.*forName*(**"UTF-8"**));  
 } **catch** (InvalidKeyException e) {  
 **throw new** IllegalArgumentException(**"Key argument does not contain a valid AES key: "** + e.getMessage(), e);  
 } **catch** (BadPaddingException e) {  
 Log.*e*(**"Decrypt"**, **"BadPaddingException: "** + e.getMessage());  
 **return null**;  
 } **catch** (GeneralSecurityException e) {  
 **throw new** IllegalStateException(**"Unexpected exception during decryption: "** + e.getMessage(), e);  
 }  
 }  
  
 **public static** IvParameterSpec buildIvParameterSpec(**byte**[] ivData)  
 **throws** NoSuchAlgorithmException {  
 **final** SecureRandom rnd = SecureRandom.*getInstance*(**"SHA1PRNG"**);  
 rnd.nextBytes(ivData);  
 **return new** IvParameterSpec(ivData);  
 }  
}

Threat 5: Elevation of Privilege