**Threat Mitigation Report**

The threats outlined in Table 1 include five (5) of the six (6) threats from the STRIDE acronym. Mitigations were applied for examples of Spoofing, Tampering, Repudiation, Information Disclosure, and Elevation of Privilege. For each section below, the steps for mitigation have been documented and described.

ID: T1

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Element | Threat | Impact |
| Spoofing | ContactManager | A hack app pretending to be Mapbox sends a fake contact to ContactManager | The fake contact may contain harmful code and cause ContactManager to crash. |

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.

**Testing:** To test our code, we sent intents to Contact Manager with malformed macs. We did this using a number of methods: using the wrong key, using the wrong authorization code, and not prepending the data properly to the authorization code. Contact Manager successfully ignored the data in these intents, and thus the threat was mitigated.

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."**);  
}

ID: T2

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Element | Threat | Impact |
| Tampering | ContactManager | A snuck malware may modify contact information saved in ContactManager | Contacts stored in ContactManager become useless |

Threat 2: Tampering

**Threat:** A snuck malware may modify contact information saved in ContactManager

**Mitigation:** To address this threat, ContactManager has a secret authorization code that only it knows. This authorization code is combined with the additional data (the contact name, phone number, and address) and encrypted.

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 use the contact.

**Testing:** To test our code, we modified the contact information while Contact Manager was running and it identified the contacts were incorrect and didn’t display them.

Contact Manager code:

**final Button addBtn = (Button) findViewById(R.id.btnAdd);**

**addBtn.setOnClickListener(new View.OnClickListener() {**

**@Override**

**public void onClick(View view) {**

**Contacts.add(new Contact(Encryptor.encrypt(nameTxt.getText().toString(),encryptionKey), Encryptor.encrypt(phoneTxt.getText().toString(),encryptionKey), Encryptor.encrypt(emailTxt.getText().toString(), encryptionKey), /\*Encryptor.encrypt(\*/addressTxt.getText().toString()/\*,encryptionKey)\*/, imageURI));**

**populateList();**

**Toast.makeText(getApplicationContext(), nameTxt.getText().toString() + " has been added to your Contacts!", Toast.LENGTH\_SHORT).show();**

**}**

**});**

...

**if (mv\_name != null && mv\_name != "") {**

**name.setText(Encryptor.decrypt(mv\_name,encryptionKey));**

**}**

**if (mv\_email != null && mv\_email != "") {**

**email.setText(Encryptor.decrypt(mv\_email,encryptionKey));**

**}**

**if (mv\_phone != null && mv\_phone != "") {**

**phone.setText(Encryptor.decrypt(mv\_phone, encryptionKey));**

**}**

**if(mv\_address != null && mv\_address != ""){**

**address = (TextView) view.findViewById(R.id.cAddress);**

**if(use\_hyperlinks) {**

**address.setMovementMethod(LinkMovementMethod.getInstance());**

**address.setText(Html.fromHtml("<a href=\"http://www.google.com\"> " + /\*Encryptor.decrypt(\*/mv\_address/\*, encryptionKey)\*/ + " </a> ").toString());**

**address.setMovementMethod(LinkMovementMethod.getInstance());**

**} else {**

**address.setText(/\*Encryptor.decrypt(\*/mv\_address/\*, encryptionKey)\*/);**

**}**

ID: T3

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Element | Threat | Impact |
| Repudiation | Mapbox | Mapbox API may modify request sent from Mapbox (e.g. change ‘555 main st’ to ’555 main st, Hartford, Connecticut’) | Get a wrong GeoJSON file |

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.

**Testing:** The mitigation of this threat was tested internally in the code. This was necessary because the log file is a private, internal file within the emulated phone so it could not be accessed physically. We used similar code shown below except we used a stream reader in order to verify the contents of the log file. We were able to confirm that the user input and what was returned from the Mapbox API along with timestamps were successfully recorded accurately. We were also able verify this data using a debugger within Android Studio.

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();  
 }

ID: T4

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Element | Threat | Impact |
| Information Disclosure | Data flow: From Mapbox to ContactManager | A hack app intercepts contact information sent by Mapbox | The contacts are exploited to unauthorized users |

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 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.

**Testing:** To test our threat mitigation, we sent malformed data to Contact Manager through the intent. This includes data encrypted using the wrong key in addition to unencrypted data. In each case, Contact Manager was unable to decrypt the data and thus ignored the malicious attack.

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);  
 }  
}

ID: T5

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Element | Threat | Impact |
| Elevation of Privilege | Mapbox | Mapbox can access certain key resources in Android system (e.g. current location) without authorization | A snuck malware may use Mapbox to modify key resources in Android system |

Threat 5: Elevation of Privilege

**Threat:** Mapbox can access certain key resources in Android system (e.g. current location) without authorization

**Mitigation:** The AndroidManifest.xml file is the application’s manifest file located in its root directory. It contains essential information about the app to the Android system. The system requires this information before running any of the application’s code. It also names the Java package for the application, describes the components of the application, determines which processes will host application components, declares which permissions others must have to interact with the application’s components, lists the instrumentation classes, declares the minimum level of the Android API required by the application, and lists the libraries the application must be linked against. Although it contains many details, the most relevant aspect of the AndroidManifest.xml file is the permission declaration. The application only has access to the permissions declared in this manifest which are listed below. Because of this, MapBox could not access other key resources in the Android system. These limitations are imposed to protect critical information and code which could be used to distort the user experience which could include modifying key resources. The <uses-permission> element in the manifest protects features of the Android system from access by an application. The application must request access with the user being prompted with a question to grant access to the particular resource. Also located in the AndroidManifest.xml file are permissions to protect the components of the application itself such as activities and services. This provides an additional level of security for the application itself; thus, the manifest file protects from unauthorized access to key resources of the Android system as well as the internal workings of the application. The AndroidManifest.xml file cannot be edited while the application is running.

**Testing:** We tried requesting permissions that we did not specify in the android manifest in our code. Android blocked all attempts to do so, so this threat is considered mitigated.

*<?***xml version="1.0" encoding="utf-8"***?>*<**manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.mapbox.mapboxsdk.android.testapp"**>  
  
 <**uses-permission android:name="android.permission.INTERNET"**/>  
 <**uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION"**/>  
 <**uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"**/>  
 <**uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"**/>  
 <**uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"**/>  
  
 <**supports-screens  
 android:anyDensity="true"  
 android:largeScreens="true"  
 android:normalScreens="true"  
 android:smallScreens="true"  
 android:xlargeScreens="true"** />  
  
 <**application  
 android:debuggable="true"  
 android:allowBackup="true"  
 android:hardwareAccelerated="true"  
 android:label="@string/appName"  
 android:icon="@drawable/ic\_launcher"  
 android:theme="@style/AppTheme"**>  
 <**activity  
 android:name="com.mapbox.mapboxsdk.android.testapp.MainActivity"  
 android:configChanges="orientation|screenSize|uiMode"  
 android:label="@string/appName"**>  
 <**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"**/>  
 <**category android:name="android.intent.category.LAUNCHER"**/>  
 </**intent-filter**>  
 </**activity**>  
  
 </**application**>  
  
</**manifest**>