Bad Cryptography Implementation

Ensuring the security and privacy of user data in modern Android applications is paramount. One critical aspect of security is the correct implementation of cryptography. Bad cryptography implementation can lead to numerous vulnerabilities, paving the way for attackers to exploit the system and compromise sensitive data. In Android applications, this can be manifested in several ways, including the use of:

Bad Cryptography Implementation Ways
`Outdated cryptographic algorithms.`
`Hardcoding keys.`
`Improper storage of keys.`
`Not utilizing secure random number generators.`

Poor cryptographic practices can have far-reaching consequences. Users may suffer data loss, identity theft, or financial fraud, while for businesses, bad cryptography can result in the loss of customer trust, legal repercussions, and substantial financial losses. Preventing bad cryptography implementations starts with adhering to well-established cryptographic standards and principles. Developers should use up-to-date cryptographic libraries widely recognized by the security community, avoid hardcoding cryptographic keys, and store keys securely using hardware-backed keystores available in Android.

Static analysis is a powerful method for identifying and preventing poor cryptographic practices. By analyzing source code without executing the application, developers can detect the use of weak algorithms, hardcoded keys, or insecure storage mechanisms. While static analysis is commonly used in penetration testing, integrating it into the development lifecycle promotes a proactive security mindset and helps catch issues early.

Understanding and identifying bad cryptography implementations is essential in safeguarding Android applications from various security threats. In the following example, we will go through the process of identifying and exploiting bad cryptography implementation of an Android application.

Exploiting Bad Cryptography Implementation

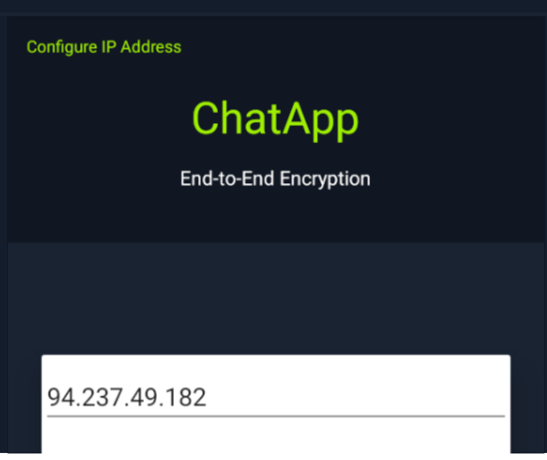
Let's connect to our Android Virtual Device and install the app using `ADB`.

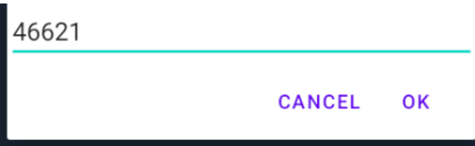
Note: The "adb connect" command is only required when attempting to connect to a remote android device.

Bad Cryptography Implementation

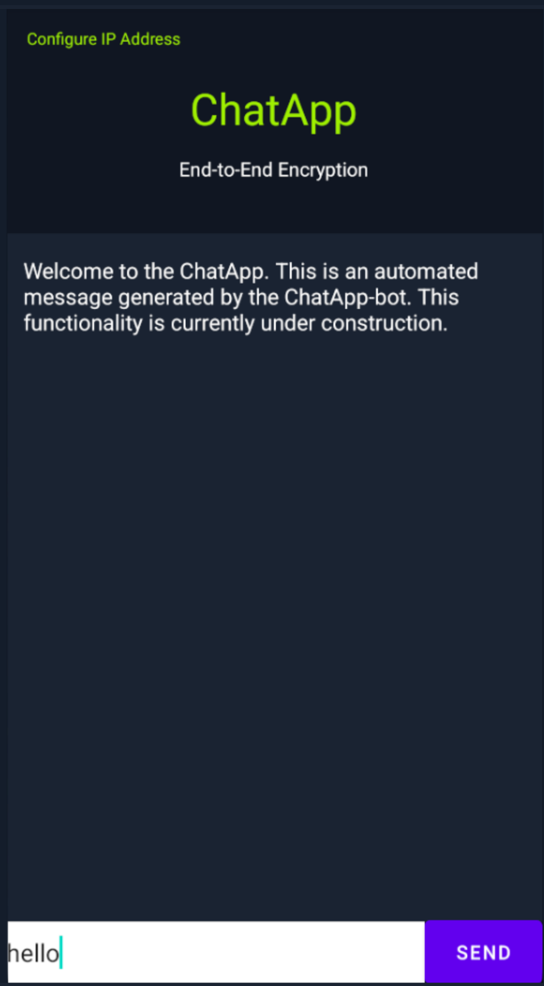
```
rl1k@htb[/htb]$ adb connect
rl1k@htb[/htb]$ adb install myapp.apk

Performing Streamed Install
Success
```

The following features a chat application where users can send and receive messages. On the left corner, we can tap the `Configure IP Address` to connect to the application's server. A pop-up window will allow us to fill in this information.





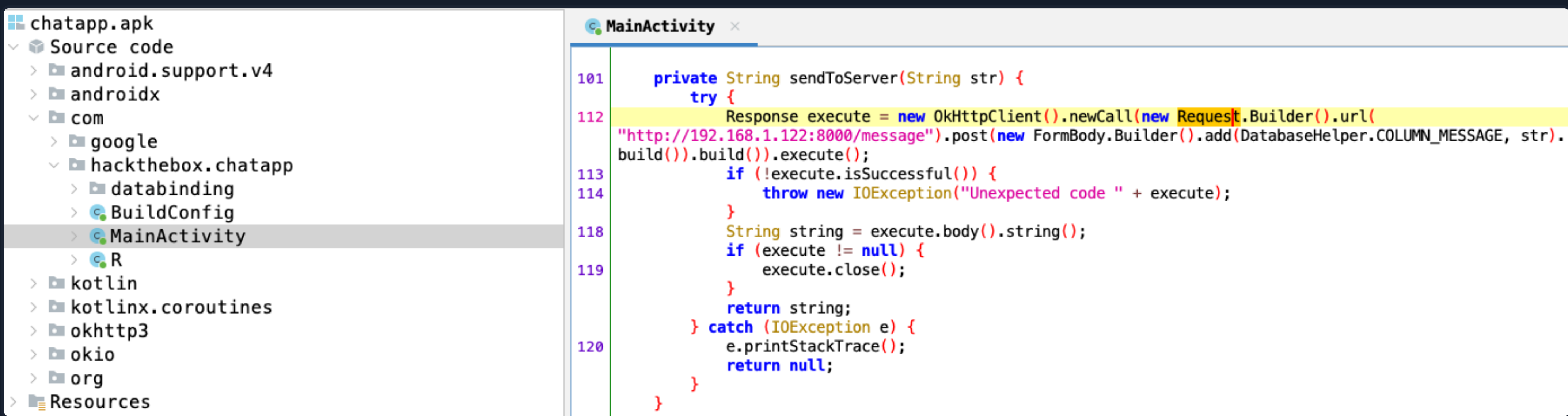
As the application implies, the chat is end-to-end encrypted, which means it should be impossible for someone else to read it. As we can see in the following screenshot, the app allows the user to chat with a bot.



Let's read the application's source code and see how it works. We can open this application with JADX by providing the complete file path to the APK.

Bad Cryptography Implementation

```
rl1k@htb[/htb]$ jadx-gui /path/to/chatapp.apk
```



The application seems to have only one activity: `MainActivity`. Looking at the `MainActivity` under the `Source code` -> `com` -> `hackthebox.chatapp`, we can find hardcoded the URL `http://192.168.1.122:8000/message` that the app uses to communicate with the server. Looking further in the source code reveals the methods `encrypt` and `decrypt`.

```
    private String encrypt(String str) {
        try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            cipher.init(1, new SecretKeySpec(getResources().getString(R.string.secret_key).getBytes(), "AES"
), new IvParameterSpec(getResources().getString(R.string.initialization_vector).getBytes()));
            return Base64.encodeToString(cipher.doFinal(str.getBytes()), 0);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    private String decrypt(String str) {
        try {
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            cipher.init(2, new SecretKeySpec(getResources().getString(R.string.secret_key).getBytes(), "AES"
), new IvParameterSpec(getResources().getString(R.string.initialization_vector).getBytes()));
            return new String(cipher.doFinal(Base64.decode(str, 0)));
        } catch (Exception e) {
            e.printStackTrace();
            return null;
```

```
        }
    }
```

These methods use the AES cryptographic algorithm to encrypt and decrypt the messages sent and received. AES is a symmetric encryption algorithm that operates on fixed-size blocks of data (128 bits for AES) using cryptographic keys of 128, 192, or 256 bits in length. In this example, AES uses the CBC (Cipher Block Chaining) mode of operation, in which each plaintext block is XORed with the previous ciphertext block before being encrypted with the AES algorithm. The first block, having no previous ciphertext block, is XORed with a special block called the Initialization Vector (IV), which is usually random. That means that if we know the encryption algorithm, the encryption key, and the IV, we can decrypt any potentially encrypted sensitive information stored in the app.

Looking closer at the method `encrypt,` we can see the `R.string.secret_key` and `R.string.initialization_vector`. This indicates that the key and IV are retrieved from the `strings.xml` in order for the encryption to happen. Reading the content of the `strings.xml` file reveals the encryption key `z5sR2v8y*AqKl7w!`.

```
    <string name="searchview_clear_text_content_description">Clear text</string>
    <string name="searchview_navigation_content_description">Back</string>
    <string name="secret_key">z5sR2v8y*AqKl7w!</string>
    <string name="side_sheet_accessibility_pane_title">Side Sheet</string>
```

A few lines below, the IV `p0o9i8u7y6t5r4e3` also appears.

```
    <string name="icon_content_description">Dialog Icon</string>
    <string name="initialization_vector">p0o9i8u7y6t5r4e3</string>
    <string name="item_view_role_description">Tab</string>
```

Further reading shows how the app stores messages in a local database:

```
    private void saveMessageToDatabase(String str, String str2) {
        SQLiteDatabase writableDatabase = this.databaseHelper.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(DatabaseHelper.COLUMN_MESSAGE, str);
        contentValues.put(DatabaseHelper.COLUMN_DIRECTION, str2);
        writableDatabase.insert(DatabaseHelper.TABLE_NAME, null, contentValues);
    }

    /* renamed from: com.example.myapplication.MainActivity$DatabaseHelper */
    /* loaded from: classes.dex */
    class DatabaseHelper extends SQLiteOpenHelper {
        public static final String COLUMN_DIRECTION = "direction";
        public static final String COLUMN_ID = "id";
        public static final String COLUMN_MESSAGE = "message";
        public static final String DATABASE_NAME = "messages.db";
        public static final String TABLE_NAME = "encrypted_messages";

        public DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, (SQLiteDatabase.CursorFactory) null, 1);
        }

        @Override // android.database.sqlite.SQLiteOpenHelper
        public void onCreate(SQLiteDatabase sQLiteDatabase) {
            sQLiteDatabase.execSQL(
"CREATE TABLE encrypted_messages (id INTEGER PRIMARY KEY AUTOINCREMENT, message TEXT, direction TEXT)");
        }

        @Override // android.database.sqlite.SQLiteOpenHelper
        public void onUpgrade(SQLiteDatabase sQLiteDatabase, int i, int i2) {
            sQLiteDatabase.execSQL("DROP TABLE IF EXISTS encrypted_messages");
            onCreate(sQLiteDatabase);
        }
    }
```

Let's enumerate the app's local storage via `ADB`. First, gain root shell access:

Bad Cryptography Implementation

```
rl1k@htb[/htb]$ adb root
rl1k@htb[/htb]$ adb shell

emu64x:/ #
```

Next, we find application's package name. While the app is running, we issue the following command.

Bad Cryptography Implementation

```
emu64x:/ # pm list packages | grep chatapp

package:com.hackthebox.chatapp
```

Now that we know the application's package name, let's list the content of its local directory.

Bad Cryptography Implementation

```
emu64x:/ # ls -l /data/data/com.hackthebox.chatapp/

total 24
drwxrws--x 2 u0_a227 u0_a227_cache 4096 2023-09-13 16:13 cache
drwxrws--x 2 u0_a227 u0_a227_cache 4096 2023-09-14 16:38 code_cache
drwxrwx--x 2 u0_a227 u0_a227       4096 2023-09-13 16:19 databases
```

The above command reveals the subdirectory `databases`. Listing its contents, we see the database `messages.db`.

Bad Cryptography Implementation

```
emu64x:/ # ls -l /data/data/com.hackthebox.chatapp/databases/

total 28
-rw-rw---- 1 u0_a227 u0_a227 20480 2023-09-14 16:40 messages.db
-rw-rw---- 1 u0_a227 u0_a227     0 2023-09-14 16:40 messages.db-journal
```

Fortunately, AVD has `sqlite3` preinstalled. Let's try to list the tables of this database using the instruction `.tables` in the `sqlite3` client.

Bad Cryptography Implementation

```
emu64x:/ # sqlite3  /data/data/com.hackthebox.chatapp/databases/messages.db

SQLite version 3.32.2 2021-07-12 15:00:17
Enter ".help" for usage hints.
sqlite> .tables
android_metadata    encrypted_messages
```

This reveals the table `encrypted_messages`. Finally, we can issue the following query to list the entries of this table.

Code: sqlite

```
sqlite> select * from encrypted_messages;

1|cTI/ewGOxoi+COl9gbceJGU7pEtLgbn9dAGCO3bkJaA=
|OUTGOING
2|i86d39WVaIHcU/Drli+uAJwsGP76I5VkN3pfpsJ1jqI=|INCOMING
3|HXnPXiKgrqqGrgjzRkEiAw==
|OUTGOING
4|Je7FNN9AbbMQ6vwP+vGyGD6GHcHec11ws7Yciovnw5GWCo85ETVIxRgedAmnH4petYowGtvEmnsLQRzC3PhH8pIBCTfHun1hjcBI+Vl2N8nxH3vwHYX+2nMhxlk
```

The database contains the encrypted conversations generated by the application. By combining the information previously discovered through source code analysis—specifically the encryption key, algorithm, and initialization vector—we can attempt to decrypt these messages. To do so, we will create the following Android application that performs the decryption.

Code: java

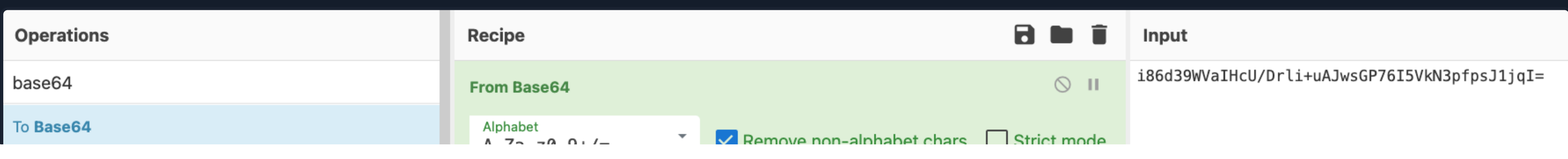
```
package com.example.myapplication;

import android.os.Bundle;
import android.util.Base64;
```

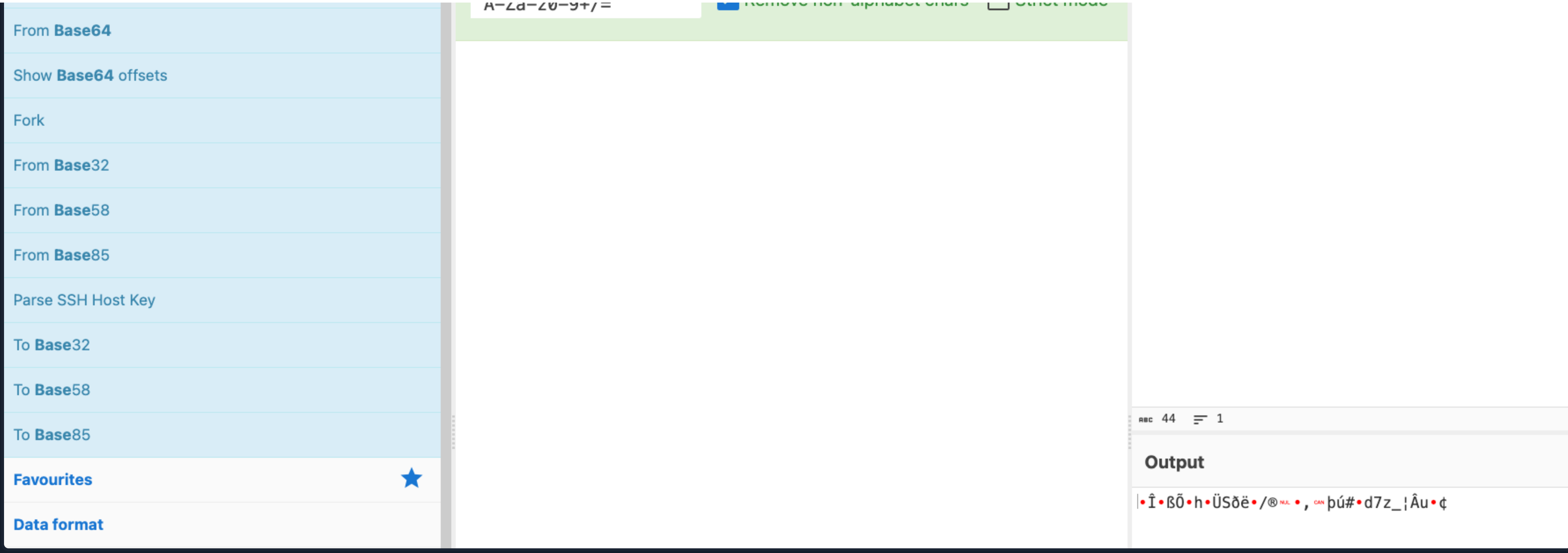
```
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;
import java.nio.charset.StandardCharsets;
import java.security.Key;
import java.security.spec.AlgorithmParameterSpec;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        String secretKey = "z5sR2v8y*AqKl7w!";
        String initializationVector = "p0o9i8u7y6t5r4e3";

        String[] encryptedMessages = {
                "cTI/ewGOxoi+COl9gbceJGU7pEtLgbn9dAGCO3bkJaA=",
                "i86d39WVaIHcU/Drli+uAJwsGP76I5VkN3pfpsJ1jqI=",
                "HXnPXiKgrqqGrgjzRkEiAw==",
                "Je7FNN9AbbMQ6vwP+vGyGD6GHcHec11ws7Yciovnw5GWCo85ETVIxRgedAmnH4petYowGtvEmnsLQRzC3PhH8pIBCTfHun1hjcBI+Vl2N8nx
        };

        for (String encryptedMessage : encryptedMessages) {
            try {
                String decryptedMessage = decrypt(encryptedMessage, secretKey, initializationVector);
                Log.d("Decrypted Message", decryptedMessage);
            } catch (Exception e) {
                Log.e("Decryption Error", "Error while decrypting", e);
            }
        }
    }

    public static String decrypt(String cipherText, String key, String iv) throws Exception {
        byte[] cipherData = Base64.decode(cipherText, Base64.DEFAULT);
        byte[] keyData = key.getBytes(StandardCharsets.UTF_8);
        byte[] ivData = iv.getBytes(StandardCharsets.UTF_8);

        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        Key secretKeySpec = new SecretKeySpec(keyData, "AES");
        AlgorithmParameterSpec ivSpec = new IvParameterSpec(ivData);
        cipher.init(Cipher.DECRYPT_MODE, secretKeySpec, ivSpec);

        byte[] decryptedData = cipher.doFinal(cipherData);
        return new String(decryptedData, StandardCharsets.UTF_8);
    }
}
```

Running the above app in Android Studio will decrypt and print the following messages in the `Logcat` window.

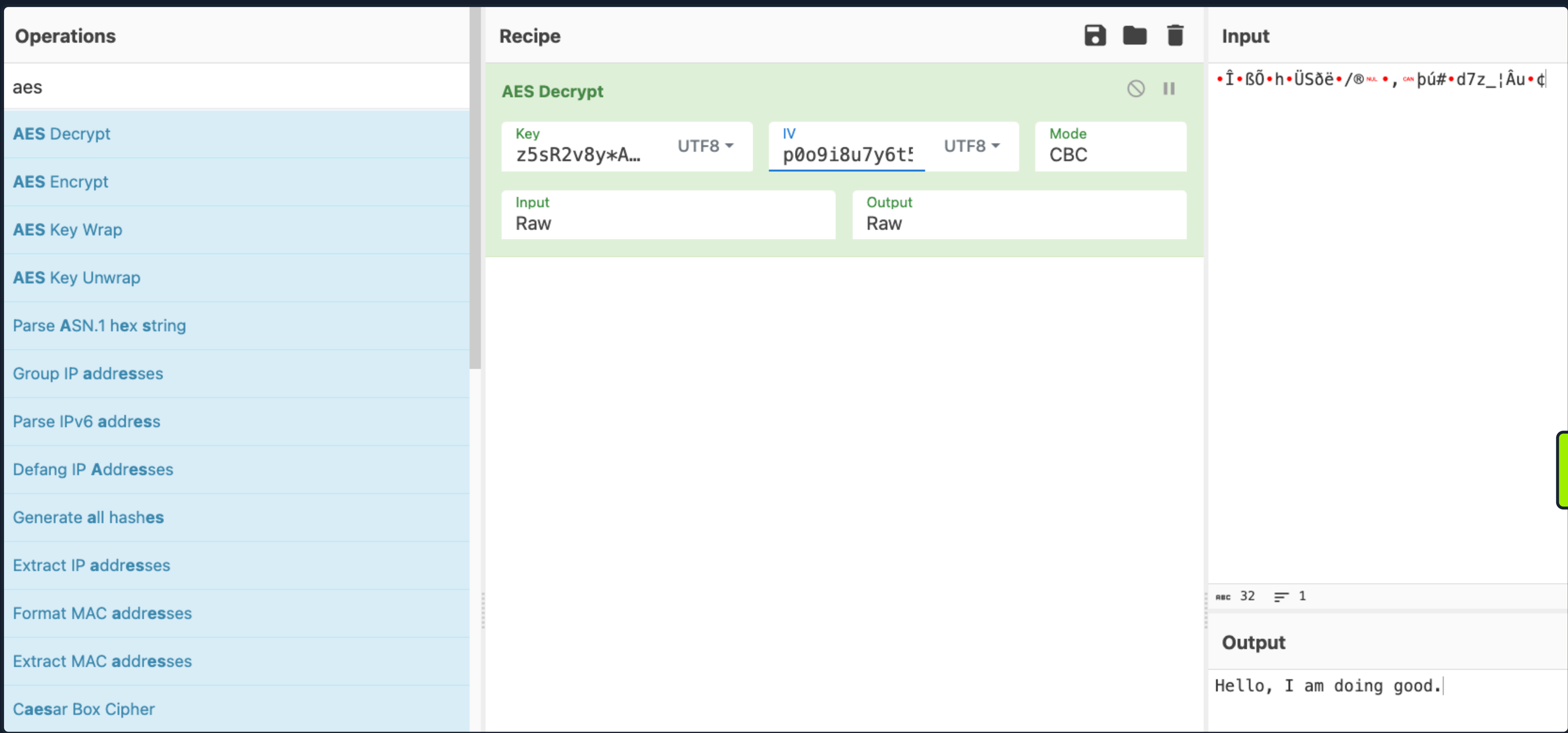
```
hey, how are you doing?
Hello, I am doing good.
hello
Welcome to the ChatApp. This is an automated message generated by the ChatApp-bot. This functionality is currently under constraction.
```

Another useful way of decrypting the above messages is using the CyberChef. From the left side menu, we select `From base64` to convert the cyphertext from base64 string to raw bytes.





Then, we remove the `From Base64` and select the `AES Decrypt` on the left side menu. Next, we configure the parameters with the key and the IV and give as an input the output we got from the `From Base64`,



The cyphertext is successfully decrypted, and both approaches work.

Connect to Pwnbox

Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location

UK 26ms

Terminate Pwnbox to switch location

Start Instance

∞ / 1 spawns left

Waiting to start...

Enable step-by-step solutions for all questions

Questions

Cheat Sheet

Answer the question(s) below to complete this Section and earn cubes!

Target(s): Click here to spawn the target system!

+ 3 What is the decrypted message stored in the app's database? Format: HTB{Th1s_1s_a_Fl4g}

Submit your answer here...

+10 Streak pts Submit myapp_bad_crypto.zip

Previous Next

Cheat Sheet

? Go to Questions

Table of Contents

Extracting and Enumerating APK Files

- Introduction
- Disassembling the APK
- Understanding Smali

Analyzing Application's Source Code

- Reading Hardcoded Strings
- Bad Cryptography Implementation
- Reversing Hybrid Apps
- Reading Obfuscated Code
- Deobfuscating Code

Analyzing Native Libraries

- Reversing Shared Objects
- Reversing DLL Files

Application Patching

Authentication Bypass

Modifying Game Apps

License Verification Bypass

Root Detection Bypass

Skills Assessment

Skills Assessment

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left