Android Malware Analysis

Submitted to:

Dr.Vimala Devi K

Submitted By:

Harshit Mishra	19BCE0799
Alokam Nikhitha	19BCE2555
Shreeyam Sharma	19BCE2700

CSE-3502 – Information Security Management

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School Of Computer Science and Engineering (SCOPE)

April, 2022

Table of Contents

1. Abstract

Android is an open-source operating system with more than a billion users. The amount of sensitive information produced by these technologies are rapidly increasing, which attracts a large number of audiences to develop tools and techniques to acquire that information or to disrupt the device's smooth operation. Despite several solutions being able to guarantee an adequate level of security, day by day the hacker's skills continued to grow, so it remains a permanent challenge for security tools developers.

As a response, several members of the research community are using artificial intelligence tools for Android security, particularly machine learning techniques to classify between healthy and malicious apps.

In this project, we implemented a static analysis framework and machine learning to do that classification.

2. Introduction

Android is an open-source operating system for mobile devices, televisions, automobiles and smart watches with more than a billion users. Therefore, it opens a wide array of attack vectors targeting the user information.

For the protection of the information and devices, Android has several security mechanisms; the most relevant are: a sandbox environment at the kernel level to prevent access to the file system and other resources; an API of permissions that controls the privileges of the applications in the device; security mechanisms at the applications development level; and a digital distribution platform (Google

Play), where the processes are implemented to limit the dissemination of malicious code.

Each application is compiled in an Android Application Package [APK] file, which includes the code of the application (".dex" files), resources, and the AndroidManifest.xml file. This latter is an important element, since it provides most of the information of the security features and configuration of each application. It also includes the information of the API regarding permissions, activities, services, content providers, and the receiving broadcasts.

There are several tools and techniques for the analysis of threats for this operating system. Between the most representative, we have static and dynamic analysis.

2.1 Static Analysis

Static analysis is a technique that assesses malicious behavior in the source code, the data, or the binary files without the direct execution of the application. Its complexity has increased due to the experience that cybercriminals have gained in the development of applications. However, it has been demonstrated that it is possible to avoid this using obfuscation techniques.

2.2 Dynamic Analysis

Dynamic analysis is a set of methods that studies the behavior of the malware in execution through gesture simulations. In this technique, the processes in execution, the user interface, the network connections and sockets opening are analyzed. Alternatively, there already exist some techniques to avoid the processes performed by dynamic analysis, where the malware has the capacity to detect sandbox-like environments and to stop its malicious behavior.

3. Literature Survey

3.1 Machine learning with dynamic analysis

DroidDolphin(2014) is a dynamic analysis framework that uses the GUI, big data, and machine learning for the detection of malicious applications in Android. Its analysis process consists of the extraction of information of the calls to the API and 13 activities, whilst the application executed in virtual environments. The SVM machine learning algorithm with the LIBSVM public library was used, a training dataset of 32,000 benign and malign applications, and a testing dataset of 3,000 healthy applications and 1,000 malicious. The preliminary results showed a precision of 86.1% and an F-score of 0.875.

Wu, Wen-Chieh & Hung, Shih-Hao. (2014). DroidDolphin. 247-252. 10.1145/2663761.2664223.

3.2 Machine learning with static analysis

Sahs and Khan (2012) propose a machine learning system to detect malware in Android devices with the One-Class SVM classification algorithm and static analysis as a technique to obtain the information from the applications. During the development, they used the Androguard tool to extract the information of the APK and the Scikit-learn framework. From the AndroidManifest.xml, they developed a binary vector that contains the information of each permission used for every application and a Control Flow Graph [CFG], which corresponds to an abstract representation of a program. Conversely, the algorithm was implemented with a test set of 2,081 clean apps and 91 malicious, with the information of 5 kernels - about binary vectors, strings, diagrams, sets, and uncommon

permissions - and another one for each application. As a result, they were able to obtain a low rate of false negatives but a high rate of false positives.

J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," 2012 European Intelligence and Security Informatics Conference, Odense, 2012, pp. 141-147, doi: 10.1109/EISIC.2012.34.

3.3 DroidMat: Android Malware Detection through Manifest and API Calls Tracing

Although understanding Android malware using dynamic analysis can provide a comprehensive view, it is still subjected to high cost in environment deployment and manual efforts in investigation. They have proposed a static feature-based mechanism to provide a static analyst paradigm for detecting the Android malware. The mechanism considers the static information including permissions, deployment of components, Intent messages passing and API calls for characterizing the Android applications behavior. In order to recognize different intentions of Android malware, different kinds of clustering algorithms can be applied to enhance the malware modeling capability. They have proposed a mechanism and developed a system, called Droid Mat. First, the Droid Mat extracts the information (e.g., requested permissions, Intent messages passing, etc.) from each application's manifest file, and regards components (Activity, Service, Receiver) as entry points drilling down for tracing API Calls related to permissions. Next, it applies K-means algorithm that enhances the malware modeling capability. The number of clusters are decided by Singular Value Decomposition (SVD) method on the low rank approximation. Finally, it uses KNN algorithm to classify the application as benign or malicious. The experiment result shows that the recall rate of

our approach is better than one of well-known tool, Androguard, published in Black hat 2011, which focuses on Android malware analysis. In addition, Droid Mat is efficient since it takes only half as much time than Androguard to predict 1738 apps as benign apps or Android malware.

3.4 Machine Learning aided Android Malware Classification

In this paper they have proposed two machine learning aided approaches for static analysis of Android malware. The first approach is based on permissions and the other is based on source code analysis utilizing a bag-of-words representation model. The permission-based model is computationally inexpensive, and is implemented as the feature of OWASP Seraphimdroid Android app that can be obtained from Google Play Store. Our evaluations of both approaches indicate an F-score of 95.1% and F-measure of 89% for the

source code-based classification and permission-based classification models, respectively.

Milosevic, N., Dehghantanha, A., & Choo, K. K. R. (2017). Machine learning aided Android malware classification. Computers & Electrical Engineering, 61, 266-274.

3.5 Droid permission miner: Mining prominent permissions for Android malware analysis

In this paper, they have proposed static analysis of android malware files by mining prominent permissions. The proposed technique is implemented by extracting permissions from 436 .apk files. Feature pruning is carried out to investigate the impact of feature length on accuracy. The prominent

features that give way to lesser misclassification are determined using Bi-Normal Separation (BNS) and Mutual Information (MI) feature selection techniques. Results suggest that Droid permission miner can be used for preliminary classification of Android package files.

A. M. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for Android malware analysis," The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014), Bangalore, 2014, pp. 81-86, doi: 10.1109/ICADIWT.2014.681467

4. Proposed work

Other artificial intelligence techniques have also been explored for safety in Android; one is so-called machine learning. Here, the capacity of the systems to learn how to identify malware without being programmed in an explicit way is presented. A large number of proposals use classification algorithms like SVM, Bagging, Neural Network, Decision Tree, and Naive Bayes. Conventional security software requires, on its identification process, human effort that implies time and resources. This might be improved through the use of intelligent tools. One of the more promising paths is the application of machine learning algorithms, which entails an increase in the efficiency in the identification of new threats.

At the present time, some proposals have arisen related to the application of classifiers for the analysis of malicious software in Android; a large number use tools for the download of applications and technologies; hence, the comparison of their results is more complicated.

Having as a foundation a framework of static analysis, we propose an adaptation that uses application databases in published articles with different open-source tools. Additionally, we assessed the framework through the application of several classification algorithms: Naive Bayes, KNeighbors, Random Forest, and Decision Tree.

The proposed framework is composed of four phases: gathering, extraction, features generation, and training and testing.

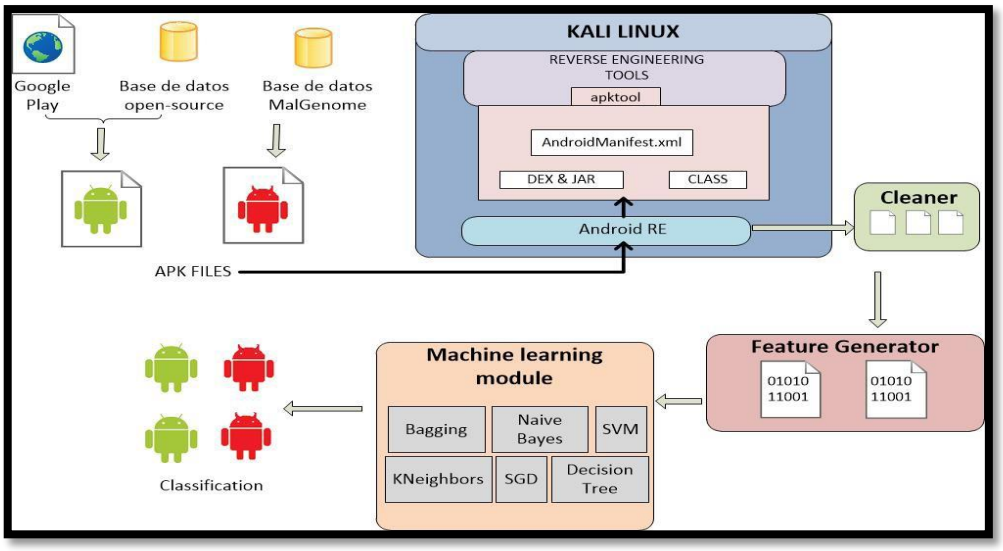


Fig. Proposed framework

5. Data Gathering

We have used the MalGenome Dataset from Kaggle.

We collected a total of 558 APK. The collection is composed of 279 applications with low privileges of the free access dataset and a random selection of 279 malwares of the MalGenome.

6. Extraction

By using ApkTool 2.0.3 tool, we can obtain the AndroidManifest.xml file of the gathered APK. We have used different inbuilt methods of Kali Linux to give us a framework to perform Static Analysis. These frame works include D2J, Dex2Jar, JDGui, etc.

7. Features generator

We developed some tools in Python that create the binary vectors that correspond to the permissions for each application; also, we added a layer regarding the classification of the application (either benign or malign). With these results, we created a new dataset containing the granted permissions for each application in binary.

Training and testing

We created two partitions, one with 71% and the other with 29% of the data for the training and testing phases, respectively. Finally, we trained and verified each machine learning algorithm.

8. Dataset

8.1 Static Analysis

Earlier, a malicious dataset software known as Android Genome Project – MalGenome that was published. The development has a sample of 1,260 malicious applications with 49 different families. The features description of its dataset was also published and, as a part of the results, 1,083 malicious codes (86%) were re-packaged versions of legitimate applications, 36.7% had the capacity to raise privileges, and 45.3% had at their end the subscription of premium message systems. They assessed four security software that in the best case could detect 79.6% of the malicious applications; in the worst case, 20.2%.

Krutz et al. (2015) analyzed a free-access dataset containing more than 1,000 Android applications, more than 4,000 of their versions, and a total of 430,000 commits of online repositories. In addition to their results, it is possible to obtain information regarding the applications under and over privileges per commit category in the repositories and versions.

8.2 Dynamic Analysis

For this approach, we used a set of pcap files from the DroidCollector project integrated by 4705 benign and 7846 malicious applications. All of the files were processed by our feature extractor script, the idea of this analysis is to answer the next question, according to the static analysis previously seen a lot of applications use a network connection, in other words, they are trying to communicate or transmit information, so. is it possible to distinguish between malware and benign application using network traffics?

9. Implementation

We used Python as a development language, and the machine learning tools of Scikit-Learn 0.17. The selected classification algorithms for the experiment were: Naive Bayes, Bagging, KNeighbors, Support Vector Machines (SVM), Stochastic Gradient Descent (SGD), and Decision Tree. During the application of the methodology, we assessed some algorithms with different configurations

Importing the libraries

```
#Importing all libraries required
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestClassifier

from sklearn import preprocessing
import torch
from sklearn import svm
from sklearn import tree
import pandas as pd
from sklearn.externals import joblib
import pickle
import numpy as np
import seaborn as sns
```

Static Analysis

```
# Doing Static Analysis
import pandas as pd
df = pd.read_csv("train.csv", sep=";")
```

```
df = df.astype("int64")
df.type.value_counts()
```

```
1    199
0    199
Name: type, dtype: int64
```

```
df.shape
```

```
(398, 331)
```

Sorting the dataset by different types: Type 1 means malware

```
pd.Series.sort_values(df[df.type==1].sum(axis=0), ascending=False)[1:11]
```

```
android.permission.INTERNET                 195
android.permission.READ_PHONE_STATE         190
android.permission.ACCESS_NETWORK_STATE     167
android.permission.WRITE_EXTERNAL_STORAGE   136
android.permission.ACCESS_WIFI_STATE        135
android.permission.READ_SMS                 124
android.permission.WRITE_SMS                104
android.permission.RECEIVE_BOOT_COMPLETED   102
android.permission.ACCESS_COARSE_LOCATION     80
android.permission.CHANGE_WIFI_STATE          75
dtype: int64
```

```
pd.Series.sort_values(df[df.type==0].sum(axis=0), ascending=False)[:10]
```

```
android.permission.INTERNET                 104
android.permission.WRITE_EXTERNAL_STORAGE    76
android.permission.ACCESS_NETWORK_STATE      62
android.permission.WAKE_LOCK                 36
android.permission.RECEIVE_BOOT_COMPLETED    30
android.permission.ACCESS_WIFI_STATE         29
android.permission.READ_PHONE_STATE          24
android.permission.VIBRATE                   21
android.permission.ACCESS_FINE_LOCATION      18
android.permission.READ_EXTERNAL_STORAGE     15
dtype: int64
```

Modelling and Naive Bayes

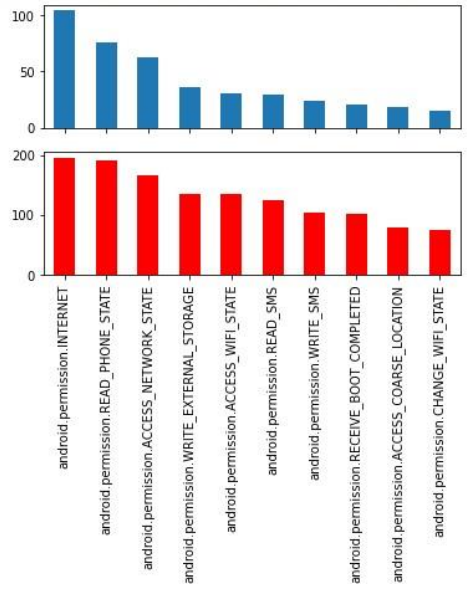
```
#Modelling
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, 1:330], df['type'], test_size=0.20, random_state=42)
```

```
# Naive Bayes algorithm
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# pred
pred = gnb.predict(X_test)

# accuracy
accuracy = accuracy_score(pred, y_test)
print("naive_bayes")
print(accuracy)
print(classification_report(pred, y_test, labels=None))
```

```
naive_bayes
0.8375
              precision    recall  f1-score   support

           0       0.91      0.76      0.83        41
           1       0.78      0.92      0.85        39

    accuracy                           0.84        80
   macro avg       0.85      0.84      0.84        80
weighted avg       0.85      0.84      0.84        80
```

```
import matplotlib.pyplot as plt
fig, axs =  plt.subplots(nrows=2, sharex=True)

pd.Series.sort_values(df[df.type==0].sum(axis=0), ascending=False)[:10].plot.bar(ax=axs[0])
pd.Series.sort_values(df[df.type==1].sum(axis=0), ascending=False)[1:11].plot.bar(ax=axs[1], color="red")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f02246756a0>
```



K Neighbors Algorithm

```
# kneighbors algorithm

for i in range(3,15,3):

    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train, y_train)
    pred = neigh.predict(X_test)
    # accuracy
    accuracy = accuracy_score(pred, y_test)
    print("kneighbors {}".format(i))
    print(accuracy)
    print(classification_report(pred, y_test, labels=None))
    print("")
```

```
kneighbors 3
0.8875
              precision    recall  f1-score   support

           0       0.94      0.82      0.88        39
           1       0.85      0.95      0.90        41

    accuracy                           0.89        80
   macro avg       0.89      0.89      0.89        80
weighted avg       0.89      0.89      0.89        80


kneighbors 6
0.85
              precision    recall  f1-score   support

           0       0.94      0.76      0.84        42
           1       0.78      0.95      0.86        38

    accuracy                           0.85        80
   macro avg       0.86      0.85      0.85        80
weighted avg       0.87      0.85      0.85        80


kneighbors 9
0.8375
              precision    recall  f1-score   support

           0       0.94      0.74      0.83        43
           1       0.76      0.95      0.84        37

    accuracy                           0.84        80
   macro avg       0.85      0.85      0.84        80
weighted avg       0.86      0.84      0.84        80


kneighbors 12
0.825
              precision    recall  f1-score   support

           0       0.91      0.74      0.82        42
           1       0.76      0.92      0.83        38

    accuracy                           0.82        80
   macro avg       0.84      0.83      0.82        80
weighted avg       0.84      0.82      0.82        80
```

Decision tree

```
#Decision Tree
clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Read the csv test file

pred = clf.predict(X_test)
# accuracy
accuracy = accuracy_score(pred, y_test)
print(clf)
print(accuracy)
print(classification_report(pred, y_test, labels=None))
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
0.95
              precision    recall  f1-score   support

           0       0.97      0.92      0.94        36
           1       0.93      0.98      0.96        44

    accuracy                           0.95        80
   macro avg       0.95      0.95      0.95        80
weighted avg       0.95      0.95      0.95        80
```

Dynamic Analysis

```
#Dynamic Analysis
import pandas as pd
data = pd.read_csv("android_traffic.csv", sep=";")
data.head()
```

	name	tcp_packets	dist_port_tcp	external_ips	vulume_bytes	udp_packets	tcp_urg_packet	source_app_packets	remote_app_packets	source_app_bytes	remote_
0	AntiVirus	36	6	3	3911	0	0	39	33	5100	
1	AntiVirus	117	0	9	23514	0	0	128	107	26248	
2	AntiVirus	196	0	6	24151	0	0	205	214	163887	
3	AntiVirus	6	0	1	889	0	0	7	6	819	
4	AntiVirus	6	0	1	882	0	0	7	6	819	

```
data.columns
```

```
Index(['name', 'tcp_packets', 'dist_port_tcp', 'external_ips', 'vulume_bytes',
       'udp_packets', 'tcp_urg_packet', 'source_app_packets',
       'remote_app_packets', 'source_app_bytes', 'remote_app_bytes',
       'duracion', 'avg_local_pkt_rate', 'avg_remote_pkt_rate',
       'source_app_packets.1', 'dns_query_times', 'type'],
      dtype='object')
```

```
data.shape
```

```
(7845, 17)
```

```
data.type.value_counts()
```

```
benign       4704
malicious    3141
Name: type, dtype: int64
```

Data Cleaning and Processing

```
#Data Cleaning and processing
data.isna().sum()
```

```
name                      0
tcp_packets               0
dist_port_tcp             0
external_ips              0
vulume_bytes              0
udp_packets               0
tcp_urg_packet            0
source_app_packets        0
remote_app_packets        0
source_app_bytes          0
remote_app_bytes          0
duracion               7845
avg_local_pkt_rate     7845
avg_remote_pkt_rate    7845
source_app_packets.1      0
dns_query_times           0
type                      0
dtype: int64
```

```
#dropping the columns with high null values
data = data.drop(['duracion','avg_local_pkt_rate','avg_remote_pkt_rate'], axis=1).copy()
```

Statistical Analysis of data

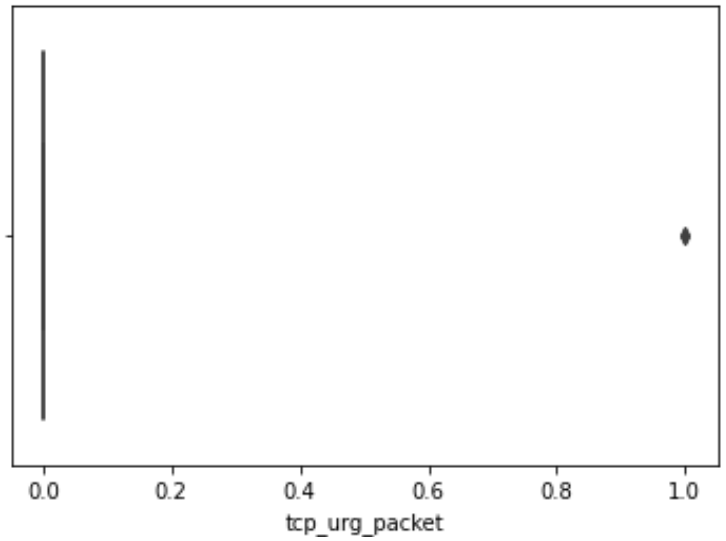
```
#Show all the statistical Analysis of the data
data.describe()
```

	tcp_packets	dist_port_tcp	external_ips	vulume_bytes	udp_packets	tcp_urg_packet	source_app_packets	remote_app_packets	source_app_bytes	remote_app_
count	7845.000000	7845.000000	7845.000000	7.845000e+03	7845.000000	7845.000000	7845.000000	7845.000000	7.845000e+03	7.84500
mean	147.578713	7.738177	2.748502	1.654375e+04	0.056724	0.000255	152.911918	194.706310	2.024967e+05	1.69226
std	777.920084	51.654222	2.923005	8.225650e+04	1.394046	0.015966	779.034618	1068.112696	1.401076e+06	8.23818
min	0.000000	0.000000	0.000000	0.000000e+00	0.000000	0.000000	1.000000	0.000000	0.000000e+00	6.90000
25%	6.000000	0.000000	1.000000	8.880000e+02	0.000000	0.000000	7.000000	7.000000	9.340000e+02	1.04600
50%	25.000000	0.000000	2.000000	3.509000e+03	0.000000	0.000000	30.000000	24.000000	4.090000e+03	3.80300
75%	93.000000	0.000000	4.000000	1.218900e+04	0.000000	0.000000	98.000000	92.000000	2.624400e+04	1.26100
max	37143.000000	2167.000000	43.000000	4.226790e+06	65.000000	1.000000	37150.000000	45928.000000	6.823516e+07	4.22732

Checking for Outliers

```
#checking the outliers in the data
sns.boxplot(data.tcp_urg_packet)
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorator
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7fe8d2196128>
```




```
data.loc[data.tcp_urg_packet > 0].shape[0]
```

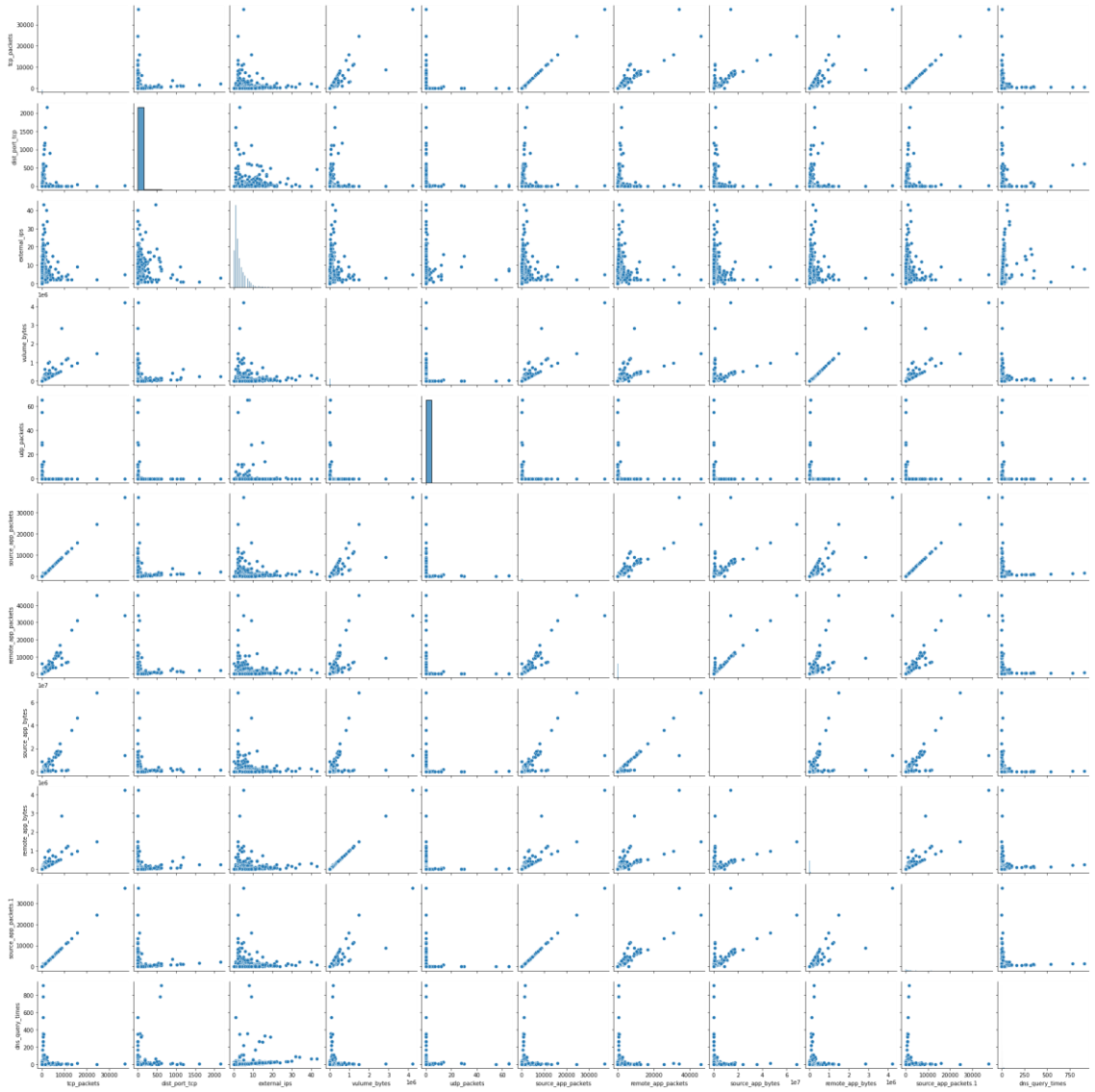
```
2
```

```
#Dropping the column 'tcp_urg_packet' since it has only two columns which are non-zero
data = data.drop(columns=["tcp_urg_packet"], axis=1).copy()
data.shape
```

```
(7845, 13)
```

Pairplots

```
#Pairplots
sns.pairplot(data)
```




```
scaler = preprocessing.RobustScaler()
scaledData = scaler.fit_transform(data.iloc[:,1:11])
scaledData = pd.DataFrame(scaledData, columns=['tcp_packets','dist_port_tcp',
                                               'external_ips','vulume_bytes','udp_packets','source_app_packets',
                                               'remote_app_packets',' source_app_bytes','remote_app_bytes',
                                               'dns_query_times'])
```

```
#Modelling
X_train, X_test, y_train, y_test = train_test_split(scaledData.iloc[:,0:10], data.type.astype("str"), test_size=0.25, random_state=45)
```

```
data=data[data.tcp_packets<20000].copy()
data=data[data.dist_port_tcp<1400].copy()
data=data[data.external_ips<35].copy()
data=data[data.vulume_bytes<2000000].copy()
data=data[data.udp_packets<40].copy()
data=data[data.remote_app_packets<15000].copy()
```

```
data[data.duplicated()].sum()
```

```
name                  AntiVirusAntiVirusAntiVirusAntiVirusAntiVirusA...
tcp_packets                                                       15038
dist_port_tcp                                                      3514
external_ips                                                       1434
vulume_bytes                                                    2061210
udp_packets                                                          38
source_app_packets                                                21720
remote_app_packets                                                18841
source_app_bytes                                                8615120
remote_app_bytes                                                2456160
source_app_packets.1                                              21720
dns_query_times                                                    5095
type                  benignbenignbenignbenignbenignbenignbenignbeni...
dtype: object
```

```
data=data.drop('source_app_packets.1',axis=1).copy()
```

Naive Bayes Algorithm

```
#Naive Bayes Algorithm
gnb = GaussianNB()
gnb.fit(X_train, y_train)
pred = gnb.predict(X_test)
## accuracy
accuracy = accuracy_score(y_test,pred)
print("naive_bayes")
print(accuracy)
print(classification_report(y_test,pred, labels=None))
print("cohen kappa score")
print(cohen_kappa_score(y_test, pred))
```

```
naive_bayes
0.44688457609805926
              precision    recall  f1-score   support

      benign       0.81      0.12      0.20      1190
   malicious       0.41      0.96      0.58       768

    accuracy                           0.45      1958
   macro avg       0.61      0.54      0.39      1958
weighted avg       0.66      0.45      0.35      1958

cohen kappa score
0.06082933470572538
```

KNeighbors Algorithm

```
# kneighbors algorithm

for i in range(3,15,3):

    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train, y_train)
    pred = neigh.predict(X_test)
    # accuracy
    accuracy = accuracy_score(pred, y_test)
    print("kneighbors {}".format(i))
    print(accuracy)
    print(classification_report(pred, y_test, labels=None))
    print("cohen kappa score")
    print(cohen_kappa_score(y_test, pred))
    print("")
```

```
kneighbors 3
0.8861082737487231
              precision    recall  f1-score   support

      benign       0.90      0.91      0.91      1173
   malicious       0.87      0.85      0.86       785

    accuracy                           0.89      1958
   macro avg       0.88      0.88      0.88      1958
weighted avg       0.89      0.89      0.89      1958

cohen kappa score
0.7620541314671169

kneighbors 6
0.8784473953013279
              precision    recall  f1-score   support

      benign       0.92      0.88      0.90      1240
   malicious       0.81      0.87      0.84       718

    accuracy                           0.88      1958
   macro avg       0.87      0.88      0.87      1958
weighted avg       0.88      0.88      0.88      1958

cohen kappa score
0.7420746759356631
```

```
kneighbors 9
0.8707865168539326
              precision    recall  f1-score   support

      benign       0.89      0.90      0.89      1175
   malicious       0.85      0.83      0.84       783

    accuracy                           0.87      1958
   macro avg       0.87      0.86      0.86      1958
weighted avg       0.87      0.87      0.87      1958

cohen kappa score
0.729919255030886

kneighbors 12
0.8615934627170582
              precision    recall  f1-score   support

      benign       0.88      0.89      0.89      1185
   malicious       0.83      0.82      0.82       773

    accuracy                           0.86      1958
   macro avg       0.86      0.85      0.86      1958
weighted avg       0.86      0.86      0.86      1958

cohen kappa score
0.7100368862537227
```

Random Forest

```
#Random Forest Algorithm
rdF=RandomForestClassifier(n_estimators=250, max_depth=50,random_state=45)
rdF.fit(X_train,y_train)
pred=rdF.predict(X_test)
cm=confusion_matrix(y_test, pred)
accuracy = accuracy_score(y_test,pred)
print(rdF)
print(accuracy)
print(classification_report(y_test,pred, labels=None))
print("cohen kappa score")
print(cohen_kappa_score(y_test, pred))
print(cm)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=50, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=250,
                       n_jobs=None, oob_score=False, random_state=45, verbose=0,
                       warm_start=False)
0.9172625127681308
              precision    recall  f1-score   support

      benign       0.93      0.94      0.93      1190
   malicious       0.90      0.88      0.89       768

    accuracy                           0.92      1958
   macro avg       0.91      0.91      0.91      1958
weighted avg       0.92      0.92      0.92      1958

cohen kappa score
0.8258206083396299
[[1117   73]
 [  89  679]]
```

10. Results and Discussion

Here, we are going to compare between all the algorithms we have used for both static analysis and dynamic analysis.

Static analysis

Algorithm	precision		recall		f1-score		accuracy
	0	1	0	1	0	1	
Naive Bayes	0.91	0.78	0.76	0.92	0.83	0.85	0.84
Kneighbors	0.94	0.85	0.82	0.95	0.88	0.90	0.89
Decision tree	0.97	0.93	0.92	0.98	0.94	0.96	0.95

Table 1. Static Analysis

Dynamic analysis

Algorithm	precision		recall		f1-score		accuracy	cohen kappa score
	B	M	B	M	B	M		
Naive Bayes	0.81	0.41	0.12	0.96	0.20	0.58	0.45	0.0608
Kneighbors	0.90	0.87	0.91	0.85	0.91	0.86	0.89	0.762
Random forest	0.93	0.90	0.94	0.88	0.93	0.89	0.92	0.825

Table 2. Dynamic Analysis

As we can see that in static analysis, the decision tree had the highest result in the classification performance of 97% and 93% respectively for Benign and Malicious apps. Whereas in dynamic analysis, random forest had the highest result in the classification performance of 93% and 90% respectively.

11. Conclusion and Inferences:

We have successfully implemented the static and dynamic analysis frameworks for classifying between the malware and benign apps by using several machine learning algorithms. By the obtained results and the precision scores we can classify the future apps between benign and malware.

12. References

[1] Wu, Wen-Chieh & Hung, Shih-Hao. (2014). DroidDolphin. 247-252. 10.1145/2663761.2664223.

[2] J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," 2012 European Intelligence and Security Informatics Conference, Odense, 2012, pp. 141-147, doi: 10.1109/EISIC.2012.34.

[3] D. Wu, C. Mao, T. Wei, H. Lee and K. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on Information Security, Tokyo, 2012, pp. 62-69, doi: 10.1109/AsiaJCIS.2012.18.

[4] Milosevic, N., Dehghantanha, A., & Choo, K. K. R. (2017). Machine learning aided Android malware classification. Computers & Electrical Engineering, 61, 266-274.

[5] A. M. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for Android malware analysis," The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014), Bangalore, 2014, pp. 81-86, doi: 10.1109/ICADIWT.2014.6814679.

[6] D. E. Krutz et al., "A Dataset of Open-Source Android Applications," 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, 2015, pp. 522-525, doi: 10.1109/MSR.2015.79.

[7] Wen-Chieh Wu and Shih-Hao Hung. 2014. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems (RACS '14). Association for Computing Machinery, New York, NY, USA, 247–252. DOI:https://doi.org/10.1145/2663761.2664223

[8] Urcuqui López, Christian & Navarro, Andres. (2016). Machine learning classifiers for android malware analysis. 1-6. 10.1109/ColComCon.2016.7516385.

[9] Batyuk, Leonid & Herpich, Markus & Camtepe, Seyit & Camtepe, Karsten & Raddatz, Aubrey- automatic assessment and mitigation of unwanted and malicious activities within Android.

AUDIT REPORT OF ANDROID MALWARE ANALYSIS

Version 1.0

Table of Contents

1.1 Purpose and Scope:

The purpose and scope of this document is to study and record a security assessment of android applications. Here we have generated our own android malware/trojan application using Kali Linux's msfconsole and our purpose is to study the application, list out all its vulnerability, test its working behind the scene and see how does our selected list of features from previous work fall in line with this application and is it safe to call that application as malicious based on our findings from different cryptographic and machine learning models.

1.2 Objective:

i. To create an android malware which is workable and not has limitations beyond our knowledge.
ii. Inspect security aspects of that application using static analysis
iii. Review the permission set of that android application and conclude if the application is malicious.
iv. Use Dex2Jar to convert the apk file into workable andoid-manifest.xml file and open all its ports and scan them.
v. Use JDGui and carefully inspect all the hidden functions of the application and how does it impact our system.
vi. Record all the set of rules which violate general working and are even identified as malicious from our machine learning analysis.
vii. Providing set of permissions which should be blocked with immediate effect to turn the application benign.

1.3 Constraints

- **Hardware Constrains:** To check the working of android malware, we need a good hardware to host virtual box and install Kali-Linux in it with a minimum of 4 cores. And 10 GB of storage allocation for smooth functioning.
- **Software Constraints:** We need to have msfconsole working in our machine. We further need to have JDGui preinstalled in our machine. Apart from these we need up and running high end browser like Mozilla or Google's Chrome.
- **Internet Speed Limitations:** we in our process need to install Kali-Linux and that is about 2.4 GB and takes about 3 hours to download with a stable and high-end network speed. Hence if there is not access to internet, the analysis is going to be impossible.
- **Multi-variate OS information:** Since here we are working on Android applications in Kali-Linux, we are ought to face inaccuracy issues. The same XML file may behave in a different fashion while working actually in Kali and might be more dangerous while installed in any stock android.
- **Scope of Audit Engagement:** Since we are analyzing only static working, it is highly possible that XML file might use obfuscation technique to hide from such testing and work behind the scene in malicious manner.

1.4 Components and Definitions

This document uses the term system, malware, android, security, testing, information security, vulnerability and penetration extensively. For better understanding of this document, their definitions are provided as follows:

System – A system is any of the following:

- Computer System (e.g., minicomputer, mainframe computer)
- Network Security (e.g., Local Area Network)
- Network Domain
- Host (e.g., a computer system)
- Network nodes, routers, switches and firewalls
- Network and/or computer application on each computer system.

Malware – Malware is an intrusive software that is designed to damage and destroy computers and computer systems. Malware is a contraction for “malicious software”. Examples of common malware includes viruses, worms, Trojan viruses, spyware, adware, and ransomware. It is any software intentionally designed to cause disruption to a computer, server, client, or computer network, leak private information, gain unauthorized access to information or systems, deprive users access to information or which unknowingly interferes with the user’s computer security and privacy.

Network – Android is a versatile working framework in view of an altered adaptation of the Linux piece and other open-source programming, planned basically for touchscreen cell phones, for example, cell phones and tablets. Android is created by a consortium of designers known as the Open Handset Alliance and industrially supported by Google. It was revealed in November 2007, with the principal business Android gadget, the HTC Dream, being sent off in September 2008.

Security – it is the quality or state of being secures, such as freedom from danger, freedom from fear or anxiety, freedom from the prospect of being laid off. In computer science, security is a term used to define a state of well-being of information, network resources or messages. Network security is a broad term that covers a multitude of technologies, devices and processes. In its simplest term, it is a set of rules and configurations designed to protect the integrity, confidentiality and accessibility of computer networks and data using both software and hardware technologies.

Testing – Network testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Network testing can also provide an objective, independent view of the network to allow the business to appreciate and understand the risk of network implementation. In general, testing is finding out how well something works. In terms of human beings, testing tells what level

of knowledge or skill has been acquired.

Information Security – Information security refers to the processes and methodologies which are designed and implemented to protect print, electronic, or any other form of confidential, private and sensitive information or data from unauthorized access, use, misuse, disclosure, destruction, modification, or disruption.

Vulnerability – Vulnerability is a weakness or flaw in software, hardware, or organizational processes, which when compromised by a thread, can result in security breach. Network vulnerabilities can either be physical or non-physical. Non-physical network vulnerabilities typically refer to anything related to data and software. Vulnerable Operating System that the IT department does not update will leave the entire system susceptible to threat actors. If a virus or malware downloads into the Operating System, it could potentially infect the whole system. Whereas the physical vulnerabilities for network includes actions like storing an on-site server in a rank closet and securing it with a lock or requiring a code to access a secure point of entry.

Penetration – a network penetration test is the process or act of identifying, recording the security vulnerabilities in applications, systems and topology by intentionally using various malicious techniques to evaluate the network's security, or lack of, response.

1.5 Auditing Phases

The audit approach along with auditing phases are as follows:

1.5.1 Selection Phase:

Here review exercises are chosen utilizing a gamble-based approach. Inward review meets with authority and the executives during the advancement of the yearly review intend to talk about dangers and expected obstructions to meeting goals. This plan is drawn closer by the Executive and Audit Committee of the Board of Trustees, so essentially, here we characterize every one of the tests and frameworks that we will assess and conclude them fully supported by that product. For our situation it will be a product in energy area and henceforth we would characterize these subtleties to the Network head or Manager around there. We would be evaluating the **transactional database, energy production logs, satellite paths and record logs, internal network design of tower** and **OS permission details.**

1.5.2 Planning:

Each review requires arranging, beginning from characterizing the extension and objective to creating review steps to meet the goal. Inward review directs an entry meeting with the executives to examine the reason for review, risk factors, and other strategies. The board is remembered for the arranging stage and the subtleties are kept in arranging and checking update. Hence, in here we work

together with the supervisory crew of our product and examine with them regarding the reason why do we want to do this review, to get an image of anticipated results. Further, we discuss with them all the risk factors that they would be expecting at end of this audit report. In our case, we would be focusing on **network strength, shield against outside attacks, risk factors of internal attacks, risk factor of satellite being compromised, energy outsource details etc.**

1.5.3 Field work:

During the field work stage, evaluators lead the means distinguished in the arranging system. Steps frequently incorporate leading meetings, exploring regulations, strategies and best work on, checking test exchanges, dissecting informational collections, and directing studies. In here the clients are kept educated regarding the review cycle through ordinary status gatherings. We, in here, list all the steps we would be taking to file the audit report. Some sample test steps are as follows:

- Interviewing the employees of their roles in the company and their administrative privileges to understand the risk of internal attacks.
- Reviewing energy logs to find if there are any mis-leading information with regards to energy output.
- Studying the network topology and firewall configurations to understand the network security.

- Studying advanced security mechanisms of routers, firewalls, positioning etc. to identify the shield against outside attacks.
- Satellite path coverage, details of path access, details of administrative access and etc. to understand the risk factor of satellite being compromised.

1.5.4 Reporting:

Evaluators lead a leave meeting with the board at the finish of the hands-on work to examine the consequences of the review, explicit discoveries and suggestions and different perceptions. Examiners impart these to the executives through a review perception notice and request that administration furnish a reaction with a remedial activity plan. Henceforth, in here, we would furnish the ware with our review report and take their survey on it. We will look for all the organization changes and consent assessment they will do after our report accommodation. We will likewise be giving them a few standard guidelines and proposal rehearses that ought to be taken by that ware.

1.5.5 Follow-up:

All review suggestions and the board remedial activities plans are circled back to give confirmation that plans are carried out. Restorative activity designs that don't seem, by all accounts, to be moderate are accounted for every year to the president and Executive and Audit Committee.

1.6 Auditing Tasks:

a) Permission Scanning: Procedures must be implemented to find active devices on the network by making use of features in the network protocols to signal devices and await a response. This is used to evaluate network security. We would be using various ICMP ping, and ICMP echo commands that can be used to find ICMP vulnerabilities and potentially zombie systems. We can further use ping commands to understand actual network configuration and find any deviation from standard norms as specified by the company. Further, to evaluate the network strength we will have to reconfigure the firewalls and understand their filtering mechanisms. We would need to see if they are working on "allow all that is not specifically denied" or "deny all that is not specifically allowed" mechanisms.

b) Class Scanning: Tools should be configured to identify security weaknesses and flaws in systems and the software which can be exploited by attackers. It is very similar to identifying the network security. By evaluating the network security, we will get most of the potential vulnerabilities of a particular system. Here we will be finding the key break-in points to the system by any outside intruder. We would also be evaluating data-leakages that could be used by an outsider or insider for eavesdropping. Further, we would be studying the encryption and decryption mechanism of data transfer to see if the data transfer is subject to confidentiality risks.

c) Services Review: Security auditing must be enabled on all components that support logging. Logs provide sufficient data to support comprehensive audits to study and analyze the effectiveness, compliance, of current security policies. As described earlier, we will be studying satellite's path and record logs and energy logs to find any discrepancies in them. These details will be used to study the risk factors of various network components in our commodity including the satellite function and data leakages which would in-turn help in recognizing vulnerabilities.

d) Session Review: Each user session must be validated with passwords. Session details such as time of log-in, log-out, IP address, host details must be stored in log files which can be used for future verification. Since our commodity is not typically service provider, we might not have session logs of users, but we can use the session logs of their employees to understand any risk from internal attacks. These sessions as network level must also be monitored and logged by firewalls and other networking devices. These details can give us insight to any details that an employee should not get access to and any other database schema that shouldn't be accessed by any specific group of employees.

e) Virus Detectors: Each host system must be configured with anti-virus applications to ensure that chances of system damage from external software is minimized. Tools related to malware detection and removal must be implemented at application level. We can also configure

firewalls to act as semi-antivirus and prevent any death-of-ping or malicious packet to enter our system. We should also configure system in such a way that any anomalies that might occur, must immediately be flagged and reported upon. The system should automatically isolate its key resources in case the anti-virus identifies severe data breach or network imbalances.

1.7 Auditing Methods:

a) AndroidManifest.XML static analysis: We sign our trojan.apk file and decode it. The process of decode is combination of decryption and decompression where the encrypted cipher information is converted into plain text human readable information and the compressed information is converted to its original format to cover all the aspects. The signing of trojan file is done using apktool pre-built in our Kali platform and it supports all readable methods.

b) JDGui Analysis: The decompressed and decrypted file of androidmanifest.xml is transferred to JDGui platform. JDGui is a standalone graphical utility that displays Java source codes of ".class" files. You can browse the reconstruction source code with JD-GUI for instant access to methods and fields. Not only this, it also supports a method to search the various keywords to enhance and fasten the process of static analysis.

c) Application Design review: Processes are implemented to review the security architecture of the organization. This includes reviewing and monitoring systems both at the application and the network layer. For the application layer, the antivirus systems are audited to ensure they are updated as per the latest databases. For the network layer, firewall, IPS systems are reviewed to ensure the system security is not compromised.

d) Document Review: All security and technical document of the application are thoroughly reviewed to understand their document policies and ethic. This will also help us understand if there are any core vulnerabilities in policy design of our application's system.

e) Permission Details: Since we are focused on android applications. Our main focus to analyze its permissions and understand if there are any core vulnerabilities in its permission itself. It is one of the easiest methods to begin our testing with.

1.8 Audit Report:

Permission Scanning:

S.NO	Check	Findings
1.	Go through system permission using AndroidManifest.XML file in Mozilla browser.	There were many permissions that had something to do with malicious intent. We have found out a list of various permissions which does not have much to do with functioning of application, instead are kept only for snooping in our cryptanalysis.
2.	Assess the activity of android permissions.	Many permissions although were not part of our analysis in static detection, we found out that those permissions had nothing to do with application and were redundant. We were not able to map the use-case of those permissions and hence we categorized them into malicious as well.

Class Scanning:

S.NO	Check	Findings
1.	Checking the available classes in our application.	We found out that there were 7 classes in our XML file. The first of them was out main class to support the loading of application and the other 6 were supportive classes labelled a.class, b.class, c.class and so on.
2.	Analyzing class information and their accessibilities	In our analysis we found out that most of the classes had to do with permissions which were classified by us as either dangerous or not needed.
3.	Finding Zombie Classes or compromised classes	None of the classes were found to be zombie or redundant. All the class were active and were functioning in the way the code was designed for them.
4.	Checking that all data inputs are validated and they are not vulnerable to common XML attacks and XML or	All database queries and XML requests are validated with parameterized queries and hence is secure.

	SQL injection attacks like query tempering and XML external entity attacks.	

Services Review:

S.NO	Check	Findings
1.	Verify user identification and the types of events performed by them in the services entries.	Here we are checking all the information of users which are being logged and monitored by the application. This is done in order to categorize users and using that information send them catered advertisements and attract their needs.
2.	Checking of undefined services	Services is the best place for attackers to hide their malicious intents. We checked all the classes and found out that none of the classes had any instance of such undefined services or features.

3.	Verify that the service logs are aggregated and protected from illegal or unauthorized access and modification.	The logs data is integrated and stored on a centralized server. Log injection can be attempted and might possibly be successful.

Session Review:

S.NO	Check	Findings
1.	Verifying session times of each class with respect to their use case profile.	The application was not under any internal attack as classes had similar use time as their use case profiles expected them to have.
2.	Verifying Session invalidation after session log-out.	When the users navigate away from the application without logging out properly, exploitation of the established user session might happen.
3.	Verifying if the session cookies are recorded optimally.	All the session activities were logged in console as expected. Even the warnings and exceptions were recorded optimally.

4.	Verifying that session IDs are unique and long.	Once a user has logged on to a system, they are granted a unique session ID that allows for secure communication between the user and web app for the valid session.
5.	Verify the session ID and timeout after specified period of inactivity.	Some session hijacking to log in to the users account is possible and hence we need to improve upon this vulnerability.

Virus Detector:

S.NO	Check	Findings
1.	Checking if the system has employed necessary software recommendations to counter cyber-attacks.	The system had necessary software in-place to combat mediocre cyber-attacks.
2.	Checking that software targets critical system areas to detect and remove active malware.	Their anti-virus software ensures that there are no active threats by checking running processes and important registry and disk sections. It also checks for

		malicious browsers plug-ns and rootkits.
3.	Checking that application detects virus over POP3, HTTP, SMTP, IMAP and FTP protocols.	A full antivirus uses a scanning engine and virus signature database to protect against virus infected over protocols.
4.	Checking firewall configurations.	Firewalls were all configured properly and were filtering traffic packets precisely.
5.	Checking physical device positions and firewall locations.	The firewall although was configured properly, it could have been better placed to increase the security of our system.

2.1 Conclusion:

In this report we have documented necessary network security aspects of our application. We have gone through its permission record, class design, functioning, session record, log reviews, cookies management and unauthorized functioning.

The security of our application was not at all placed well. The androidmanifest.xml file had all the malicious signatures that we initially thought of. Since we had ourselves prepared that trojan file, we were sure that the permissions were not under tha part of obfuscation and we were not going to have much difficulty in finding most of the vulnerabilities. But this is not true in real-time analysis. There, programmers try to hide all these malicious intents and can even hide from box-testing. Our static analysis gave us all the permissions which were not at all required and hence we were sure if the application possessed any of these permissions, the application is going to be malicious.

The class scanning suggested that class required a few services declaration which we couldn't identified and hence classified them as malicious, which might not be the case.

The vulnerability scanning suggested that system, more or less, is secured. Again, a few systems can be targeted to be compromised with, but their configurations can easily be updated to deny any such possibilities.

The session management again suggested that sessions are being managed judiciously and records are maintained optimally to support clear and precise revival of them.

The virus detectors were in place and working well, but their positioning needs to be changed. The firewalls were loosely placed only in the entry point and exit point of our system, they need to be configured at another location to prevent servers from being attacked.

Hence in this report, a detailed audit has been carried out for data integration to extract hidden information. Relevant techniques such as network scanning, log analysis and correlation, virus/malware scanning, port scanning, and session managements have been employed to find out all-possible scenarios which can lead to security issues. Based on this audit, the key checks and findings have been listed out as well. Based on this audit, the user can take necessary steps to fix existing security flaws to make the overall application more secure and reliable.

2.2 Recommendations and Solutions:

2.2.1 Permission Configuration:

Observation: It was observed that many of the permissions were not at all required.

Recommendation: The users of the application are advised to disable those permissions even if the application ask you to enable them for smoother functioning. These permissions had nothing to do with functioning of the applications but are only for their snooping activities.

2.2.2 Password Controls:

Observation: Ensuring that adequate logical access controls are applied correctly to all systems and are enforced by the system helps to prevent unauthorized access through password guessing. Although passwords policies on the network require complex passwords to be entered and that they are required to be changed every 90 days and the user account is locked out after three unsuccessful password attempts, the password length is currently set to six characters long and history of only six previous passwords is maintained. Failure to implement strong password controls increases the likelihood that unauthorized users may gain access to the network.

Recommendation: Management should enhance the current password policies for the Corporate Network and for recommended practice within all council applications. This should require a minimum password length of eight characters to be enforced and restrict the history of previous passwords that can be used to thirteen.

2.2.3 Use of Emails – Monitoring:

Observations: A periodic review will help ensure that use of the council's email system by its stall is in compliance with its stated policy, that is, not used other than for council business. Although, there email filters applied for the acceptable use of emails, there is no compliance check performed on acceptable email usage. Monitoring of email usage is only performed if it requested by a service manager. There is a risk of the email system being used for excessive non council which could impact on service delivery.

Recommendation: We recommend that periodic email monitoring and compliance checks are performed on the use of the Council's email systems to identify any instance of potential excessive usage during core working hours, this information should be routinely provided to management for information purposes.

2.2.4 Port Permissions:

Observation: The application was accessing many open ports of android device which could lead to issues with online attacks

Recommendation: We need to close all the open ports after their use is done with. Even if a particular port is in use, we should search in APK file if that port is in access and close it for mean time if there is information transfer possible. If not, we can carry on to use that port without any issues.

2.2.5 Data Sharing Permissions:

Observation: There were many permissions which were sharing user data with third party. This included information such as call records, message records, messages, storage information, internally stored files and many others.

Recommendation: We need to carefully inspect if these permissions lie in accordance to use case of our android application. If not, we have to disable those permissions. For example, if the application is a voice recorder, then it is bound to access our storage and record our voice, but if it is a payment wallet or gateway, then there is no use for it to record our voice and have permission to our microphone.