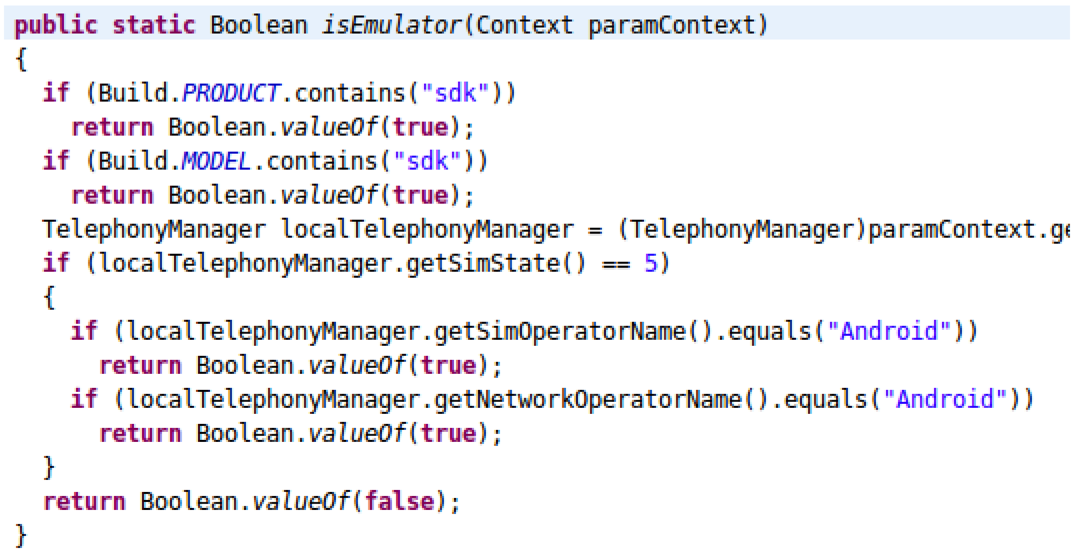
Android malware based on SMS encryption and with KitKat support

maldr0id.blogspot.ch/2014/09/android-malware-based-on-sms-encryption.html

Łukasz



```
public static Boolean isEmulator(Context paramContext)
{
  if (Build.PRODUCT.contains("sdk"))
    return Boolean.valueOf(true);
  if (Build.MODEL.contains("sdk"))
    return Boolean.valueOf(true);
  TelephonyManager localTelephonyManager = (TelephonyManager)paramContext.ge
  if (localTelephonyManager.getSimState() == 5)
  {
    if (localTelephonyManager.getSimOperatorName().equals("Android"))
      return Boolean.valueOf(true);
    if (localTelephonyManager.getNetworkOperatorName().equals("Android"))
      return Boolean.valueOf(true);
  }
  return Boolean.valueOf(false);
}
```

Most of the malware based on the SMS C&C communication channel is not compatible with Android 4.4 KitKat. This is due to the fact that KitKat introduced a concept of one messaging app, which all other apps had to go trough before they send or handle a received text message. This prevented malware from hiding received short messages or sending without saving them in the "Sent" folder.

Well, it wasn't hard to predict that this state of affairs wouldn't last long and that malware authors would eventually catch up. Malware described here not only supports KitKat, but also uses an open-source SMS encryption tool as a basis for its code. Let's have a look at the insides of the new sample (hash: 84e2e9e8430792b583d02d3cc1bf8535) and let's call it SmsSecure, just for the sake of brevity.

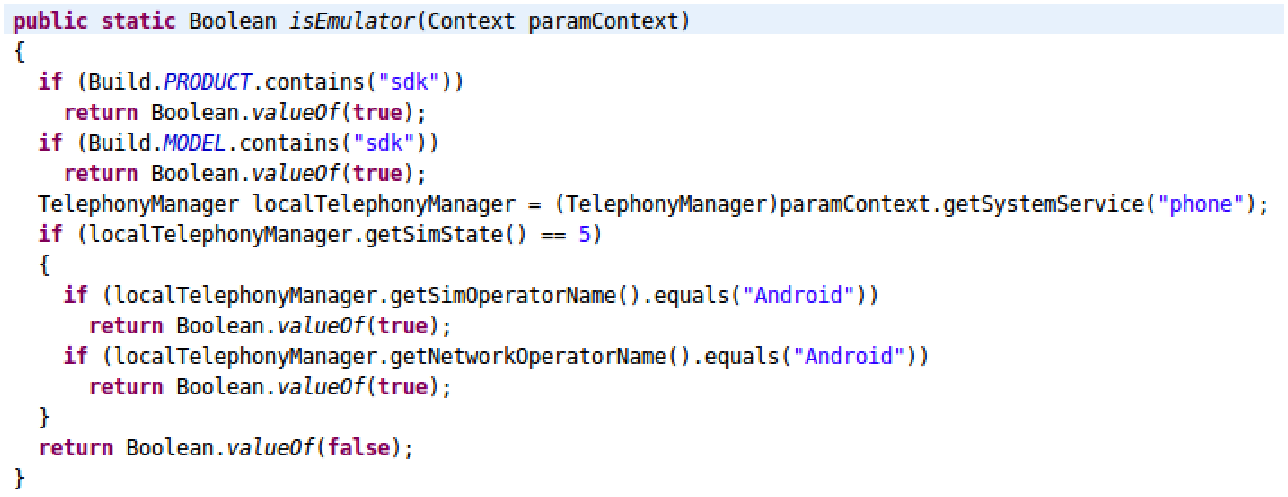
Open-source encryption

I have to say that I did lie a bit. Well, I didn't tell the whole truth. Commands sent trough the SMS channel are not encrypted per se. Only the code for secure messaging is used as a basis for this malware, presumably because it supports Android 4.4 out-of-the-box and it is easier to encourage users to install "Secure SMS" application as a main messaging app. It has the same package name, org.thoughtcrime.securesms and hides main portion of its code in the x* (namely xlibs, xservices, xpack, xbroadcast) subpackages. These are responsible e.g. for sending received text messages to the C&C server.

There are two channels of communication: one trough unencrypted text messages and the other one trough POST requests, which body is encrypted using blowfish. So, all of this talk about "encrypted SMS" is just a scam.

Emulator detection and other techniques

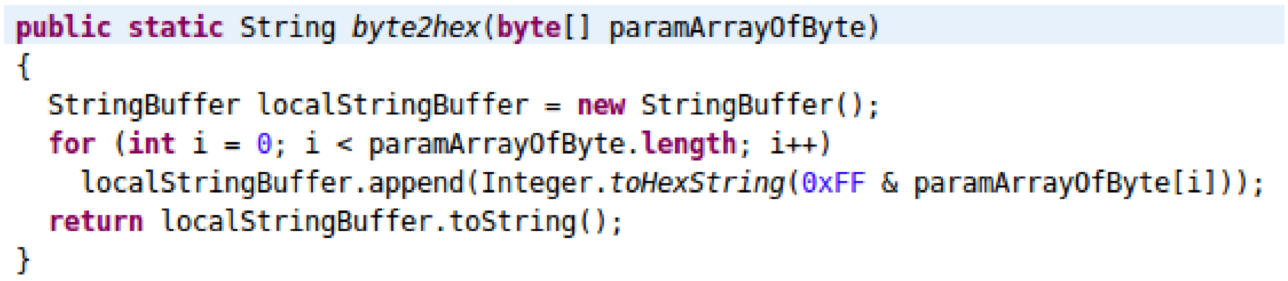
What we are used to is that the Android malware increasingly has a code that detects whether it is run on the emulator or not. There are couple of usual checks (e.g. phone model) and a couple of unusual checks illustrated below.



```java
public static Boolean isEmulator(Context paramContext)
{
  if (Build.PRODUCT.contains("sdk"))
    return Boolean.valueOf(true);
  if (Build.MODEL.contains("sdk"))
    return Boolean.valueOf(true);
  TelephonyManager localTelephonyManager = (TelephonyManager)paramContext.getSystemService("phone");
  if (localTelephonyManager.getSimState() == 5)
  {
    if (localTelephonyManager.getSimOperatorName().equals("Android"))
      return Boolean.valueOf(true);
    if (localTelephonyManager.getNetworkOperatorName().equals("Android"))
      return Boolean.valueOf(true);
  }
  return Boolean.valueOf(false);
}
```

Not that frequent technique is checking for the Network Operator and SIM Operator name. These names are hardcoded in the emulator and are equal to Android.

Another interesting feature is the configuration. Configuration is stored as the Raw Resources, because it is encrypted. Encryption method is Blowfish/CBC (IV="12345678") and the key is stored in the resource cleverly called blfs.key and the configuration is encrypted in two files named in the same cunning way: config.cfg and config1.cfg. However, if you try to just decode the configs using the key provided, you will fail. Key is somewhat "obfuscated" using a byte2hex function outlined below.



```java
public static String byte2hex(byte[] paramArrayOfByte)
{
  StringBuffer localStringBuffer = new StringBuffer();
  for (int i = 0; i < paramArrayOfByte.length; i++)
    localStringBuffer.append(Integer.toHexString(0xFF & paramArrayOfByte[i]));
  return localStringBuffer.toString();
}
```

It takes every byte in the key file, renders it to a hexstring and then construct a new string of this hexstrings. The function that gets the key limits it size to 50 bytes. So instead of the key:

NfvnkjlnvkjKCNXKDKLFHSKD:LJmdklsXKLNDS:<XObcniuaebkjxbcz

we get:

4e66766e6b6a6c6e766b6a4b434e584b444b4c4648534b443a

Really clever obfuscation technique. After decrypting the configuration files we get two beautiful XMLs:

```
<?xml version="1.0" encoding="utf-8"?>
        <config>
          <data rid="25"
                shnum10="" shtext10="" shnum5="" shtext5="" shnum3="" shtext3="" shnum1=""
shtext1=""
                del_dev="0"
                url_main=""
                url_data=""
                url_sms=""
                url_log=""
                phone_number="+43676800505476"
 download_domain="ttt"
                ready_to_bind="0" />

        </config>
```

```
<?xml version="1.0" encoding="utf-8"?>
        <config>
          <data rid="25"
                shnum10="" shtext10="" shnum5="" shtext5="" shnum3="" shtext3="" shnum1=""
shtext1=""
                del_dev="0"
                url_main=""
                url_data=""
                url_sms=""
                url_log=""
                phone_number="15555215554"
                ready_to_bind="0" />
        </config>
```

See if you can spot the fake one. Anyhow, that's it for the text-based C&C, but what about the HTTP-based dropzone? Well, there isn't any URL hard coded. The dropzone URL comes via the text command.

SMS commands

Each command is send in a following form (the originating number is completely ignored):

<control_code> <command> <number> <service_code>

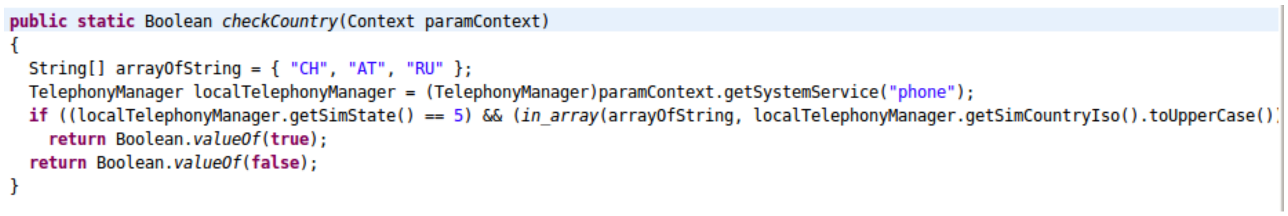
Two last values are optional. control_code should be one of the 6-digit predefined codes. service_code is a number to which the logs will be send (as a text messages) and will be updated no matter what the issued command is. What are the possible commands? Well, there are eleven of them, listed below.

- START - starts redirecting all text messages to the number (service_code is ignored) by setting the RTB (ready to bind) value to 2 and sending the "Service Started" SMS to the number.
- STARTB - starts redirecting all text messages to the number URL (URLs are for some reason called "buffers") by setting the RTB value to 1 and sending the message as above (but via HTTP).
- STOP - command complimentary to the START and STARTB commands. Sets the RTB value to 0 and sends "Service stopped" SMS/POST request to the number.
- DEL - removes the application. Just like that. But you still have to send the correct control_code. Sends back the "Delete command received" to the number.
- SETB - sets what the author calls "Buffers". These are all the URLs needed for the HTTP communications. URLs are set according to the following pattern: number/[1|2|3|4].php. First one is used for a ping-back, second one is a dropzone for the captured messages, third one is for the configuration update and the last one is for the logs (e.g. that some action has started).
- CLEARB - clears the "buffers" i.e. the URLs mentioned above.
- SETP - sets the phone number that will overwrite any number that comes in the text message. Useful for using the fake number values in the subsequent commands.
- CLEARP - reverts the above setting.
- CLEAR - does both the CLEARB and CLEARP.
- LOCK - set the password for the device to number and locks the device. I don't think this feature is enabled in the mentioned sample, as the sample does not require the Device Admin privileges.
- UNLOCK - clears the password and locks the device (which makes it unlockable after the lock above).

The application also supports blocking messages from some numbers. These numbers should be contained in the "filters" variable (separated by the semicolon) and can only be updated via the HTTP or in the starting configs.

Summary

This is a fine example of the malware evolution - now completely compatible with KitKat. It even checks whether you have set it as a main messaging app already and displays a pestering notification if not. It also tries to do some targeting based on the SIM operator (as pictured below). However, this function is never used in the code, as it was probably dropped at some point by the authors. Anyhow, the targeted countries were supposed to be Austria, Russia and Switzerland.



```
public static Boolean checkCountry(Context paramContext)
{
  String[] arrayOfString = { "CH", "AT", "RU" };
  TelephonyManager localTelephonyManager = (TelephonyManager)paramContext.getSystemService("phone");
  if ((localTelephonyManager.getSimState() == 5) && (in_array(arrayOfString, localTelephonyManager.getSimCountryIso().toUpperCase()
    return Boolean.valueOf(true);
  return Boolean.valueOf(false);
}
```

However, my favorite part is how they extensively log everything and use "setted" instead of "set".

Update

@seckle_ch reports that the SmsSecure is used in conjunction with the retefe banking trojan, targeting Austria, Switzerland and Sweden.

Android malware goes Mono and Lua - part 2 (Ransomware)

KLM's "We'll keep you grounded" Programme

Using World War II techniques to fight ransomware