ARL-TN-0650 ● SEP 2015

US Army Research Laboratory

Monitor Network Traffic with Packet Capture (pcap) on an Android Device

by Ken F Yu

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-0650 ● SEP 2015

US Army Research Laboratory

Monitor Network Traffic with Packet Capture (pcap) on an Android Device

by Ken F Yu
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
Sep 2015	Final	4/15/14–9/1/14

4. TITLE AND SUBTITLE
Monitor Network Traffic with Packet Capture (pcap) on an Android Device

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ken F Yu

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIN-D
2800 Powder Mill Road
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-0650

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The purpose of this report is to provide detailed information on how to effectively build an Android application to monitor network traffic using open source packet capture (pcap) libraries.

15. SUBJECT TERMS
ELIDe, Android, pcap

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Ken F Yu
Unclassified	Unclassified	Unclassified	UU	22	19b. TELEPHONE NUMBER (Include area code) 301-394-3181

Contents

List of Figures

1. Purpose

The purpose of this document is to provide detailed information on how to effectively build an Android application to monitor network traffic using open source packet capture (pcap) libraries.[1]

2. Configuration Used

The following list is the software and hardware used for development and testing:

Operating System (OS): Red Hat Enterprise Linux (RHEL), version 6.5
Android Development Tools (ADTs), Version 22.3.0–887826
Saferoot[2]
Samsung Galaxy S3
Dell Precision T7400
- 8 GB memory
- Intel Xeon X5472 Central Processing Unit (CPU)
  - 64-bit quad and dual-core
  - 3.00 GHz

Open source pcap libraries
Oracle VirtualBox[3] – optional
Android OS ISO for x86[4] – optional
Android Software Development Kit (SDK)[5]
Android Native Development Kit (NDK)[6]

3. Overview

In the field of network monitoring, the pcap Application Programming Interface (API) is a well-known set of open source libraries for capturing network traffic. Because the pcap API is written in C, the codes can be ported to an Android device natively to improve speed performance while monitoring high-volume network traffic.

4. Android Tool Kits Setup

To setup the Android tool kits, download Android SDK[5] and NDK.[6] Then, decompress the 2 files to the local bin path. Afterward, setup the local .bashrc with the correct Android environment settings, assuming that home/user1/bin is where the tool kits are installed. See Fig. 1.

```
export ANDROID_SDK=/home/user1/bin/adt-bundle-linux-
x86_64-20131030/sdk
export ANDROID_NDK=/home/user1/bin/android-ndk-r9
export PATH=$ANDROID_SDK/tools:$ANDROID_SDK/platform-
tools: $PATH
export ANDROID_HOME=/home/user1/bin/adt-bundle-linux-
x86_64-20131030/sdk
```

Fig. 1 Example of .bashrc with Android tool kit settings

Ensure that "ndk-build" command can be executed if NDK is setup correctly. From ECLIPSE, an integrated development environment (IDE), part of the Android SDK, ensure that the SDK location is set to the correct directory. To perform this operation, open Eclipse, go to <Preference->Android>. The SDK location edit line should be set to the path where SDK was installed (i.e., </home/user1/bin/adt-bundle-linux-x86_64-20131030/sdk>).

5. Rooting Android

Because the pcap library requires root privileges, rooting of the Android device is necessary. Refer to the Technical Note (TN), Rooting Android Device[7] for more detail.

6. Setup Android on VirtualBox

To speed up the debugging process without using the actual Android device, or the very slow Android virtual machine, deploy the Android OS on VirtualBox. Refer to the TN, Android Virtual Machine (VM) Setup on Linux[8] for additional information.

7. Cross-Compile Android Native Code on x86

CROSS-COMPILER can create executable code for a processor other than the one on which the compiler is running. I used VirtualBox to run the Android OS VM on an x86 processor. Because this processor is a nonARM-based processor, it requires cross compilation.

To add this capability to a processor that runs on a different platform, create an Android makefile (Application.mk) for the current Android project under the <jni> subdirectory. Make sure to use uppercase characters in the Application.mk file. An example of this file is shown in Fig. 2.

```
APP_STL := gnustl_static
APP_ABI := armeabi armeabi-v7a x86
```

Fig. 2 Example of Application mk file

The <x86> is the keyword used in this makefile to specify the x86 processor type, which is used to cross-compile applications that run on the Android OS using an x86 processor. After the processor type is added, simply perform the normal ndk-build steps[9]. The native code is now ready for deployment.

The Android application can use the Java Native Interface (JNI) to communicate with the native codes. As a result, there is no need to be concerned about building a separate executable for native calls. However, if the application is required to spawn a native process, then it is necessary to have a separate build for each processor-based executable. In order to deploy the native executable from the Android package, this executable must be copied or saved under the <asset> path of the project directory.

There are many options that allow a programmer to distribute processor-based executables. The easiest method is to have a separate processor build package for deployment. Another more complicated option is to create a separate path for a processor-based directory under the project's <asset> path. For example, <project/asset/x86> for x86 and <project/asset/arm> can be used for ARM processors. To deploy the correct type of executable when running the application, the CPU type should be obtained. The processor type on the device can be retrieved by using the BufferedReader class from the device's </proc/cpuinfo> file. Figure 3 shows an example of how to retrieve the CPU information.

```
Private String getCpuInfo()
{
      StringBuffer sb = new StringBuffer();
      sb.append(“abi: “).append(Build.CPU_ABI).append(“\n”);
      if (new File(“/proc/cpuinfo”).exists()) {
            try
           {
                  BufferedReader br = new BufferedReader(new
FileReader(new File(“/proc/cpuinfo”)));
                  String data;
                  while ((data = br.readLine()) != null) {
                        sb.append(aLine + “\n”);
                  }
                  if (br != null) {
                        br.close();
                  }
            } catch (IOException e) {
                  e.printStackTrace();
            }
     }
      // info from the cpu file
      return sb.toString();
}
```

Fig. 3 Example of how to get CPU information in code

After the CPU type is obtained, the AssetManager in <android.content.res.AssetManager> API can be used to extract the native executable. The codes shown in Fig. 4 are an example of how to retrieve files from the asset folder.

```
Private void copyFileFromAssets(String assetPath, String
filename, String toPath)
     {
         AssetManager assetManager = this.getAssets();

         InputStream in = null;
         OutputStream out = null;

         try {
             in = assetManager.open(assetFilename);
             out = new FileOutputStream(newFileName);

             byte[] buffer = new byte[1024];
             int read;
             while ((read = in.read(buffer)) != -1) {
                 out.write(buffer, 0, read);
             }
             in.close();
             in = null;
             out.flush();
             out.close();
             out = null;
         } catch (Exception e) {
             Log.e(mServiceName, e.getMessage());
         }
     }
```

Fig. 4 Example of retrieving files from the asset folder

8. Building Application with Native Codes

8.1 Calling Native Codes Using JNI

One way for an Android application to interact with native codes is to use JNI. JNI provides the cross-platform between Java and C/C++. When calling C from Java, it is necessary to have the Java-to-C calling method be static. The keyword “static native” should be used. The C++ method has to be static as well. The example in Fig. 5 illustrates how Java calls C using JNI.

```
Public int callNativeExample()
{
      String config = “config”;
      String pcap = “pcap”;
      String datafile = “file”;
      String savefile = “save”;
      // calls the native code
      int ret = naInitElide(config, pcap, datafile,
saveFile);
}

// JNI native call declaration
private static native int naInitElide(String config, String
pCap, String datafile, String saveFile);
private static native String naStartElideTest();
private static native void naStopService();

static {
      System.loadLibrary(“TestJni”);
}
```

Fig. 5 Example of Java calling C using JNI

The above examples show 3 methods: 1) Method naInitElide() passes 4 strings to the JNI interface method and returns an integer; 2) Method naStartElideTest() has no passing parameter and returns a string; and 3) Method naStopElideTest() has no passing and no return parameters. These methods belong to the native library called “TestJni”. The library must be declared as static. After these methods are created and saved as a Java file, the javah utility program can be used to create the C/C++ interface header. An example of the usage is shown in Fig. 6.

```
Javah –verbose –classpath $ANDROID_SDK/platforms/android-
19/android.jar:bin/classes –jni –d jni
com.example.elidetest.ElideTestService
```

Fig. 6 Example of compiling C/C++ header file from Java using javah

As shown in Fig. 6, a C/C++ based header file will be created based on the number of native methods that are created. In this example, I assume that ElideTestService is an <ElideTestService.java> file and that <com.example.elidetest> is the namespace. The “-classpath” option points to where the <android.jar> file is. The –d option specifies where to place the output header files. The header file can be used as definition for the methods used later—just need to fill in the contents. An example of the C/C++ implementation using JNI is shown in Fig. 7.

```
#include <jni.h>
#include “com_example_elidetest_ElideTestService.h”
/*
 * Class:     com_example_elidetest_ElideTestService
 * Method:    naInitElide
 * Signature:
(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljav
a/lang/String;)I
 *
 * (JNIEnv *, jclass, jstring, jstring, jstring, jstring);
 */
JNIEXPORT jint JNICALL
Java_com_example_elidetest_ElideTestService_naInitElide
     (JNIEnv *env, jclass clazz, jstring config, jstring
pCap, jstring datafile, jstring saveFile)
{
     // converts Java String to const char*
     const char *_datafile = env->GetStringUTFChars
(datafile, 0);
    params.datafile = _datafile;

     const char *_config = env->GetStringUTFChars (config,
0);
    params.configFile = _config;

     const char *_pCap = env->GetStringUTFChars (pCap, 0);
     params.pcapFile = _pCap;

     const char *_saveFile = env->GetStringUTFChars
(saveFile, 0);
     params.pcapSaveFile = _saveFile;

     // release buffers allocated by the env
     env->ReleaseStringUTFChars (config, _config);
     env->ReleaseStringUTFChars (datafile, _datafile);
     env->ReleaseStringUTFChars (pCap, _pCap);
     env->ReleaseStringUTFChars (saveFile, _saveFile);

     // do your stuffs here

     return 0;
}
```

Fig. 7 Example of using JNI in C

The C/C++ method name uses a very strict declaration for the Android JNI. The method name must follow the correct naming convention for it to work properly. For this reason, the javah program comes in very handy because it creates the header automatically. Once the native codes are written, compile the codes using

the usual <ndk-build> command via command shell under the project's <jni> directory. It will build the native codes object library for you.[9]

8.2 Calling Native Codes from an Android Application

The purpose of calling native codes from an Android application is to allow the application itself to have the ability to start as "root". This application can spawn a process that requires administrative privileges. Under the current design Android development requirement, an Android Graphical User Interface (GUI) application cannot directly call any native API that requires administrative permissions, as in the case of pcap's <pcap_open_live()> method. To overcome this obstacle, the application needs to spawn the super user (su) process, which requires the user to grant permission. After "su" is granted by the user, this process will need to spawn a shell process and then calls the process that requires "root" access. An example of the "su" granting process using native code is illustrated in Fig. 8.

```
private static java.lang.Process mElideProcess = null;
private 9escry9 bCreated = false;
private DataOutputStream mOS = null;
try {
      mElideProcess = Runtime.getRuntime().exec(“su”, null,
null);

      // Attempt to write a file to a root-only
      mOS = new
DataOutputStream(mElideProcess.getOutputStream());
mReader = new BufferedReader(new
InputStreamReader(mElideProcess.getInputStream()));

      if(null != mOS && null != mReader)
            bCreated = true;
}
catch (Exception e)
{
      // Can’t get root!
      mRoot = false;
      mRootMsg = “createProcess failed: su “ +
e.getMessage();
}
if (bCreated == true)
{
      try {
            mOS.writeBytes(“id\n”);
            mOS.flush();

            String currUid = mReader.readLine();
            9escry9 exitSu = false;
            if (true == currUid.contains(“uid=0”))
            {
                  // do something after permission is
granted.
                  // ie. Call your own process
                  // pcapElide is an example of the command
                  // that is using
                  try {
                        String cmd = “./pcapElide”;
                        mOS.writeBytes(cmd);
                        mOS.flush();
                  } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                  }

            }
      }
}
```

Fig. 8 Example of the “su” granting process using native code

9. Retrieve Live Network Traffic via pcap

There are many ways to capture live network traffic on an Android device. However, one way to capture network traffic is to use the pcap library in native codes. This method is the most preferred for optimal performance in obtaining network traffic data for analyses.

To open network traffic, the <pcap_open_live()> method is used. The 2 API methods to retrieve network data once the device is opened are <pcap_next()> and <pcap_loop()>. I performed experiments using both API methods. At constant network traffic, I discovered that if an Android device receives data speed lower than 100 kbps, the 2 API methods did not have any performance issues. However, with constant incoming data rate higher than 250 kbps, <pcap_next()> does not respond to the caller. In other words, the API just stayed in an infinite loop without returning back to the caller. On the other hand, <pcap_loop()> uses the callback handler capability, and it has no problem handling any network speed. If the network speed is too fast, it simply drops the packets and continues its required request. An example of how to open and retrieve network data using pcap is shown in Fig. 9.

```
pcap_t* getPCapFileDescrpitor(const string &pcapOrDevice)
{
      pcap_t*11escry = NULL;
      char errbuf[PCAP_ERRBUF_SIZE];   // 256
      bpf_u_int32 pMask;            /* subnet mask */
      bpf_u_int32 pNet;             /* ip address*/

      // use this if it is an offline file
      //11escry = pcap_open_offline(pcapOrDevice.c_str(),
errbuf);

      pcap_if_t *alldevs, *d;
      char dev_buff[64] = {0};
      int i =0;

      // Prepare a list of all the devices
      if (pcap_findalldevs(&alldevs, errbuf) == -1)
      {
            cout<< “no device found\n”;
            return NULL;
      }

      cout << “flags: “ << alldevs->flags << “\n”;

      // Print the list to user
      // so to see the right selection is made
      bool bFoundDevice = false;
      for (d=alldevs; d; d=d->next)
      {
        if (pcapOrDevice == d->name)
            // requested device found
            bFoundDevice = true;
      }

      if (bFoundDevice == false)
      {
            cout << “device “ << pcapOrDevice << “ cannot be
found as active\n”;
            return 11escry;
      }

      // fetch the network address and network mask
      pcap_lookupnet(pcapOrDevice.c_str(), &pNet, &pMask,
errbuf);

      // Now, open device for sniffing none promiscuous = 0
      11escry = pcap_open_live(pcapOrDevice.c_str(), // name
of the device
```

```
         65536, // guarantees that the whole packet will be
captured on all the link layers
         0,     // none promiscuous mode
         1000,              // read timeout 1 sec
         errbuf);         // error buffer

     // no longer need device list
     pcap_freealldevs(alldevs);
 }
/* Callback function invoked by libpcap for every incoming
packet */
void packet_handler(u_char *param, const struct pcap_pkthdr
*header, const u_char *pkt_data)
{
     // do your own processing here
}
// example of opening device
pcap_t *pCap = getPCapFileDescrpitor("wlan0");

// example of setting pcap_loop after device is opened
if (pcap_loop(pCap, -1, (pcap_handler)packet_handler, NULL)
== -1)
{
     // problem with pcap_loop
}
```

Fig. 9 Example of how to open and retrieve network data using pcap

10. Conclusions

Network traffic capturing on Android using pcap with native codes is relatively new. This report serves as a guide for any developer creating an application on Android to take advantage of the proven open source speedy performance to monitor or capture network traffic using pcap.

11. References

1. TCPDump and LibPCAP. [accessed 24 July 2014]. http://www.tcpdump.org/.
2. Saferoot: Root for VRUEMJ7, MK2, and Android 4.3. [accessed 24 July 2014]. http://forum.xda-developers.com/showthread.php?t=2565758.
3. Oracle VirtualBox. [accessed 24 July 2014].https://www.virtualbox.org/.
4. Android x86 ISO. [accessed 24 July 2014]. http://www.android-x86.org/download/.
5. Android SDK. [accessed 24 July 2014]. http://developer.android.com/sdk/index.html/.
6. Android NDK. [accessed 24 July 2014]. http://developer.android.com/tools/sdk/ndk/index.html/.
7. Yu KF. Rooting Android device. Adelphi (MD): US Army Research Laboratory (US); September 2015. Report No.: ARL-TN-0706.
8. Yu KF. Android Virtual Machine (VM) setup on Linux. Adelphi (MD): US Army Research Laboratory (US); September 2015. Report No.: ARL-TN-0651.
9. Ritchey RP, Payer GS, Harang RE. Compilation of a network security/machine learning toolchain for Android ARM platforms. Adelphi (MD): US Army Research Laboratory (US); July 2014. Report No.: ARL-CR-0739. Also available at

   http://www.arl.army.mil/www/default.cfm?technical_report=7125.

List of Symbols, Abbreviations, and Acronyms

ADT	Android Development Tool
API	Application Programming Interface
CPU	Central Processing Unit
GUI	Graphical User Interface
IDE	integrated development environment
JNI	Java Native Interface
NDK	Native Development Kit
OS	Operating System
pcap	packet capture
RHEL	Red Hat Enterprise Linux
SDK	Software Development Kit
su	super user
TN	Technical Note
VM	Virtual Machine

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIRECTOR US ARMY RSRCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT

1 (PDF) GOVT PRINTG OFC A MALHOTRA

1 (PDF) DIRECTOR US ARMY RSRCH LAB RDRL CIN D K YU

INTENTIONALLY LEFT BLANK.