White Paper

Implementing CyaSSL as an SSL Provider on the Android Platform

A Step-by-Step Guide

TABLE OF CONTENTS

I. INTRODUCTION

The Android platform has quickly become one of the most popular mobile operating systems both by developers and end users. As such, security is a high priority, but so is the sometimes-conflicting goal of minimizing resource usage. By default, the Android platform uses OpenSSL to provide Java developers with SSL functionality. By using CyaSSL instead, developers gain a smaller size footprint as well as a faster SSL implementation.

The goal of this White Paper is to provide insight and instruction on how to install CyaSSL as a Java SSL provider alongside OpenSSL on the Android Platform. In doing so, developers will have the option to choose CyaSSL for SSL functionality through the Java API's "javax.net.ssl" package and gain the advantages that the CyaSSL library has to offer.

II. WHAT ARE TLS AND SSL? (IN A NUTSHELL)

TLS (Transport Layer Security) and its predecessor SSL (Secure Socket Layer) are cryptographic protocols that provide security for communications over networks such as the Internet (Transport Security Layer, 2010). Originally created by Netscape, TLS and SSL allow client/server applications to create an encrypted link and ensure that all traffic being sent and received is private and secure. Most typically, SSL and TLS perform unilateral authentication between client and server, meaning that only the server is authenticated. Also supported is bilateral authentication, where both the identity of the server and client are authenticated.

TLS and SSL provide this secure layer through the use of public/private key encryption. A message encrypted with a public key can only be decrypted using the associated private key. The public key is usually publically available, so that anyone can encrypt a message with this key. Only the owner of that public key (the holder of the private key) may decrypt the message once encrypted. There are multiple algorithms that may be used by TLS and SSL to perform this encryption. For an in depth look at TLS and SSL, consider reading the Wikipedia page on the subject:

http://en.wikipedia.org/wiki/Transport_Layer_Security

III. JAVA SECURITY PROVIDERS (A BRIEF OVERVIEW)

The Java platform contains a set of security APIs consisting of several major areas (public key infrastructure, authentication, secure communication, and access control), all of which are only interfaces defining a "contract" for provider implementations to meet. This gives Java programmers the ability to use a single API to gain desired security functionality while still allowing those developers to plug in their desired implementation under that API.

According to the Java Provider documentation, this architecture was designed around three main design principles. The following principles are taken from the Oracle Java SE Documentation (Oracle):

1. Implementation Independence

   Applications do not need to implement security themselves. Rather, they can request security services from the Java platform. Security services are implemented in providers (see provider diagram, below), which are plugged into the Java platform via a standard interface. An application may rely on multiple independent providers for security functionality.

2. Implementation interoperability

   Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.

3. Algorithm extensibility

   The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

Under this provider architecture, multiple providers for a service may be installed side by side. In the case of having multiple providers for a service, each provider is given an order of priority in which it should be used by the Java platform.

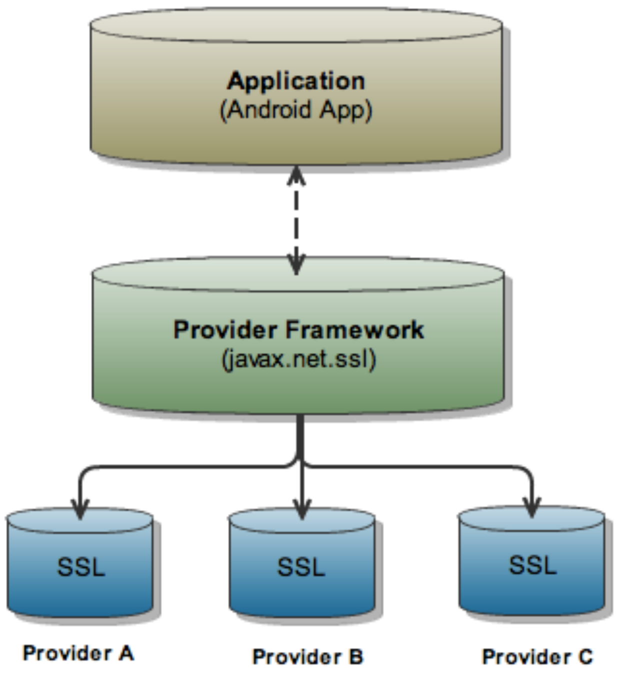

Figure 1:
The structure of the Java Provider framework, showing specifically the javax.net.ssl package and how individual providers can be “plugged in” to the Provider Framework.

In this paper, the focus will be on SSL. The javax.net.ssl Java API package is responsible for supplying SSL functionality to the Java platform. The diagram above gives a general overview of how SSL providers, or more generally, providers, are organized within the Java platform. The CyaSSL provider is what will be installed into the Android platform in the following steps.

Java security providers are listed and prioritized in a file called "java.security" on OS X and Linux, or "java.properties" on the Android platform. On OS X, Linux, and Android this file is most likely found at the locations listed below. The details of this file will be covered in section 9, "Configuring a Java Security Provider on Android."

Java Security Properties File:

System/Library/Frameworks/JavaVM.framework/Home/lib/security/java.security	[OS X]
$JAVA_HOME/lib/security/java.security	[Linux]
/libcore/security/src/main/java/java/security/security.properties	[Android]

IV. PREPARING A BUILD ENVIRONMENT

First things first, the local build environment needs to be setup to accommodate for the Android build system. This paper and the included instructions were written based on the requirements and status of the Android build system at the time of writing. Given the speed at which the Android system evolves, procedures and requirements will change over time.

To build the Android source files, either Linux or OS X must be installed on the development machine. Windows is not currently supported. Further, the most current version of OS X, Snow Leopard, is not supported due to incompatibilities with Java 6. The remainder of this paper will assume that the operating system of choice is 32-bit Linux. Some of the following information was collected from the Android project website. For setup information relating to non-32-bit-linux, reference the Android project site (http://source.android.com/source/download.html)

Setting Up Your Machine

The local build environment must have the following installed:

- Git 1.5.4 or newer and the GNU Privacy Guard
- JDK 5.0, update 12 or higher. Java 6 is not currently supported. Instructions for downgrading are explained below.
- flex, bison, gperf, libsdl-dev, libesd0-dev, libwxgtk-dev (optional), build-essential, zip, curl

To install the above packages using apt-get, issue the following command:

```
$ sudo apt-get install git-core gnupg sun-java5-jdk flex bison gperf libsdl-dev libesd0-
dev libwxgtk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev
```

A few additional tools and notes:

- Valgrind (a suite that will help find memory leaks, stack corruption, array bounds overflow, etc.) may be useful during Android development.
  ```
  $ sudo apt-get install valgrind
  ```

- Intrepid (8.10) users may need a newer version of libreadline:
  ```
  $ sudo apt-get install lib32readline5-dev
  ```

Downgrading Java to JDK 1.5

Depending on the build environment, if JDK 1.5 is not the current Java installation, it is necessary to roll back to JDK 1.5. To accomplish this, follow the steps below:

1. Download JDK 1.5 from the Oracle website. At the time of writing it was found here: http://java.sun.com/javase/downloads/5u22/jdk

2. If downloading for Linux, the download will be similar to: jdk-1_5_0_22-linux-i586.bin. The following steps assume that you are installing it at /usr/lib/jvm. Execute the following commands to install:

   ```
   $ sudo mkdir /usr/lib/jvm
   $ sudo mv ~/jdk-1_5_0_22-linux-i586.bin /usr/lib/jvm
   $ cd /usr/lib/jvm
   $ sudo chmod +x jdk-1_5_0_22-linux-i586.bin
   $ sudo ./jdk-1_5_0_22-linux-i586.bin
   ```

   After executing these commands, there will be a /usr/lib/jvm/jdk-1.5.0_22 directory. The original jdk-1_5_0_22-linux-i586.bin may be deleted.

3. Now the symbolic links must be updated to Java 1.5. Execute the following commands to update the most commonly used Java executables so that they will point to the Java 1.5 installation.

   ```
   $ cd /usr/bin
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/java java
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/jar jar
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/javac javac
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/javadoc javadoc
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/javah javah
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/javap javap
   $ sudo ln -sf /usr/lib/jvm/jdk1.5.0_22/bin/javaws javaws
   ```

4. Verify that Java 1.5 has been installed correctly by executing the following command:

   ```
   $ java –version
   ```

   The output of this command should be something that reflects a Java 1.5 installation:

   ```
   java version "1.5.0_22"
   Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_22-b03)
   Java HotSpot(TM) Client VM (build 1.5.0_22-b03, mixed mode, sharing)
   ```

5. Lastly, edit the file /etc/jvm and add the Java 1.5 installation to the top of the list, as follows. An example of what this file should look like can be seen on the top of the following page.

   ```
   $ sudo gedt /etc/jvm
   ```

```
# This file defines the default system JVM search order. Each
# JVM should list their JAVA_HOME compatible directory in this file.
# The default system JVM is the first one available from top to
# bottom.

/usr/lib/jvm/jdk1.5.0_22
/usr/lib/jvm/java-6-sun
/usr/lib/jvm/java-6-sun-1.6.0.20
/usr/lib/jvm/java-6-openjdk
/usr
```

(Example Contents of /etc/jvm)

V. THE ANDROID PLATFORM

After the local build environment has been set up, the Android source needs to be downloaded. This section will cover the steps necessary to download the Android platform source and explain how to build both the entire Android platform and only specific projects.

Working with and contributing to the Android platform is done through the use of Git and Repo. *Git* is an open-source version control system designed to handle very large projects that are distributed over multiple repositories (Android Source). In Android, *Git* is used for local operations such as local branching, commits, diffs, and edits. *Repo* on the other hand, is a tool built by Google on top of Git. According to Google, "Repo helps manage the many Git repositories, does the uploads to the revision control system, and automates parts of the Android development workflow. Repo is not meant to replace Git, only to make it easier to work with Git in the context of Android." The following sections follow the steps outlined by the Android Project source documentation (Google).

For more information about using Git and Repo, see the Android Project page, here: http://source.android.com/source/git-repo.html

Installing Repo

To install, initialize, and configure Repo, the following steps should be taken:

1. Make sure there is a ~/bin directory in your home directory, and check to make sure that this bin directory is in your path:
```
$ cd ~
$ mkdir bin
$ echo $PATH
```

2. Download the repo script and make sure it is executable:
```
$ curl http://android.git.kernel.org/repo >~/bin/repo
$ chmod a+x ~/bin/repo
```

Initializing a Repo Client

Before the Android source can be checked out, a Repo client must be initialized. The following steps will initialize a repo client using a specified manifest and prepare your working directory for pulling files from the Android repository.

1. Create an empty directory to hold your working files:
   ```
   $ mkdir mydroid
   $ cd mydroid
   ```

2. Run "repo init" to bring down the latest version of Repo with all of its most recent bug fixes. You must specify a URL for the manifest.
   ```
   $ repo init -u git://android.git.kernel.org/platform/manifest.git
   ```

   If you would like to check out a branch other than the "master", specify it using the –b option:
   ```
   $ repo init -u git://android.git.kernel.org/platform/manifest.git -b cupcake
   ```

3. When prompted, configure Repo with your real name and email address. If you plan on submitting code to the Google platform source, use an email address that is associated with a Google Account.

A successful initialization of Repo will result in a message such as "repo initialized in /mydroid."

Getting the Source Files

After Repo has been set up, in order to pull down files into your working directory from the Android repositories specified in the manifest, run the command "$ repo init." The Android source files will be located in your working directory under their project names.

Verifying Git Tags

Each tag in the Google repository will be signed with a GnuPG key. You can verify tags by importing this public key into your GnuPG key database. To do this, follow the steps outlined in the Android Source Documentation under the "Verifying Git Tags" heading, here: http://source.android.com/source/download.html.

Setting up the Environment

To set up the environment to build the source for a generic device and generic product, run the following two commands. The first sets some environment variables for the Android build system, and the second builds the emulator. Run these commands from the platform root in your working directory.

```
$ source build/envsetup.sh
$ lunch 1
```

Initial Platform Build Process

The steps up to this point have prepared the build environment and the Android platform. The next step is to build the Android platform source. The Android platform uses a monolithic system, meaning that the Android source is one big build tree. Any part of the build can reference any other part of the build via relative filenames.

There are two ways to build the Android platform, either by using the regular "make" command, or the Google "mmm" command. In the Android build system the "mmm" command will build the entire platform, whereas "mm" will build a specific project.

To build the entire platform, change to the working directory, and run the make (or mmm) command:

```
$ cd ~/mydroid
$ make
```

If you wanted to build only a specific project, such as the "libcore" project, change to the "libcore" directory and run the "mm" command:

```
$ cd ~/mydroid/libcore
$ mm
```

VI. THE ANDROID EMULATOR – MAKING LIFE SIMPLER

To make testing and debugging modifications to the Android platform easier, Google has created an Android emulator. This emulator is highly customizable – allowing custom hardware configurations, providing a log output, allowing shell access, and much more.

Before using the emulator, it needs to be downloaded. It comes bundled with the Android SDK, which is available for download here: http://developer.android.com/sdk/index.html.

Once the SDK has been downloaded, a variety of tools will be found in the <Android-SDK>/tools directory, where <Android-SDK> is the root directory of the SDK. These tools will include the emulator, and the Android Debug Bridge (adb).

VII. JAVA SSL PROVIDER COMPONENTS OVERVIEW

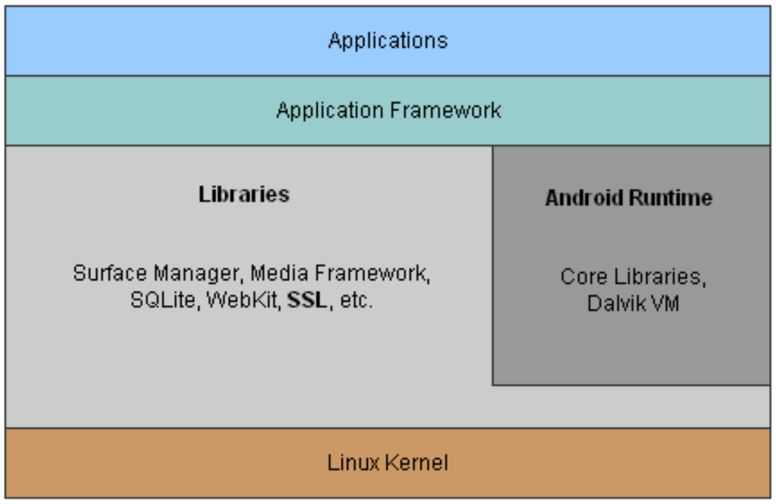
Applications
Application Framework
Libraries
Surface Manager, Media Framework, SQLite, WebKit, SSL, etc.
Android Runtime
Core Libraries, Dalvik VM
Linux Kernel

Figure 2: Android platform layer composition

The CyaSSL Java SSL provider is composed of two main parts: the CyaSSL shared library and the Java provider code (using JNI to communicate between Java and the CyaSSL C library). The Android platform is divided into several layers, which are shown in Figure 2, above. The two layers being affected during the SSL Provider installation are the Libraries and Android Runtime layers. In order to continue, the CyaSSL Java SSL Provider must be downloaded from the yaSSL website, here:

http://ww.yassl.com/yaSSL/Download_More.html

CyaSSL is a C language based SSL library targeted for embedded and RTOS environments, primarily because of its small size and speed. CyaSSL is also commonly used in applications for standard operating environments because of its royalty free pricing and cross platform support. It supports the industry standards up to the current TLS 1.2 level, and is up to 20 times smaller than OpenSSL. User benchmarking and feedback reports dramatically better performance when using CyaSSL versus OpenSSL.

CyaSSL Shared Library

The CyaSSL shared library is compiled (at the time of writing) from the CyaSSL 1.5.6 source code by the Android build system into the shared library "libcyassl.so." This library contains all the functions that would be found in the CyaSSL library on a regular desktop installation and is the foundation of the CyaSSL Java SSL provider.

The source files which will be installed into the Android platform are found in the provider download under the /external/cyassl directory.

Java Provider Code

The Java provider code utilizes JNI to provide communication between Java code and native C and C++ code. Because of this, there are two separate parts: the Java code files and the native C++ file.

The source files which will be installed into the Android platform are found in the provider download under the /libcore/yassl directory. This directory contains both the Java code and the C++ native code.

VIII. INSTALLATION OF THE CYASSL SHARED LIBRARY

In this document, <Android-Platform> is used to represent the location where the Android platform source root is located. The CyaSSL Java SSL provider is dependent upon the CyaSSL shared library; therefore, the shared library will be installed first.

1. Since we are installing a new library, we're going to create a new folder under the "/external" directory in the Android platform. Most third party shared libraries being placed into the Android platform should be installed in the "/external" directory. To do this, copy the "cyassl" directory from "src/external/cyassl" of the provider download to the "/external" directory of the Android platform. This folder should now be located at:

   <Android-Platform>/external/cyassl

   These source files will be compiled into libcyassl.so by the Android build system using the rules in the "/external/cyassl/src/Android.mk" file.

2. Open <Android-Platform>/build/core/prelink-linux-map.map and add a new entry for libcyassl.so under the heading "# libraries for specific apps or temporary libraries." It should look similar to the following:

   ```
   libcyassl.so        0x9C500000 # [~1M] for external/cyassl
   ```

   Note that libraries should be aligned on 1MB boundaries.

3. Open the file <Android-Platform>/dalvik/libnativehelper/Android.mk and add libcyassl.so to the "shared_libraries" list.

IX. INSTALLATION OF THE JAVA SSL PROVIDER

Now that the shared library has been installed, the rest of the provider may be installed.

1. The existing SSL provider in Android (Apache Harmony using OpenSSL), is located in the "/libcore" directory. The CyaSSL provider will be installed there as well for consistency. To begin, copy the "yassl" directory from "src/libcore/yassl" of the provider source to the "/libcore" directory of the Android platform. This folder should now be located at:

   <Android-Platform>/libcore/yassl

2. The CyaSSL SSL Provider initialization method (in the native C++ code) must be registered with the Android platform so that the native methods can be registered with the Dalvik VM at runtime. Unlike a desktop Java installation, Dalvik handles JNI slightly different in that it requires a function to be written to explicitly register every native method that needs to be made available to the JVM. This method is the one that needs to be added to libnativehelper's Register.c file. Open the file <Android-Platform>/dalvik/libnativehelper/Register.c.

   Add the "register_com_yassl_xnet_provider_jsse_NativeCrypto" method under the entry for the existing provider. When added, it should look as follows:

   ```
   if (register_org_apache_harmony_xnet_provider_jsse_NativeCrypto(env) != 0)
   goto bail;
   if (register_com_yassl_xnet_provider_jsse_NativeCrypto(env) != 0)
   goto bail;
   ```

3. The last step is to configure the provider to act as the default SSL provider for Java. To do this, open the "security.properties" file (located at <Android-Platform>/libcore/security/src/main/java/java/security/security.properties). Make the following changes to configure the CyaSSL provider:

   a. Add the following line to the list of providers. This line needs to be above the default "org.apache.harmony.xnet.provider.jsse.JSSEProvider" provider. Note the numbers beside each provider. These reflect the priority of the provider. It might be necessary to re-number this list after inserting the new provider.

      ```
      "security.provider.3=com.yassl.xnet.provider.jsse.JSSEProvider"
      ```

   b. Change the "ssl.SocketFactory.provider" entry to point to the new CyaSSL Provider. After modification, it should read as follows:

      ```
      "ssl.SocketFactory.provider=com.yassl.xnet.provider.jsse.SocketFactoryImpl"
      ```

X. TESTING PLATFORM MODIFICATIONS

At this point, the CyaSSL provider is fully installed into the Android platform. The next step is to build the platform with the new provider installed and make sure there are no build errors come up. If no errors arise during the platform build, the provider should then be loaded into the emulator to make sure the platform loads correctly with the new provider installed.

Re-building the Android Platform

To rebuild the entire Android platform, follow the steps as they were executed before. All commands should be run from the Android platform source root. The build process can take a significant amount of time depending on the build environment.

```
$ source build/envsetup.sh          [Sets environment variables]
$ lunch 1                           [Builds the emulator]
$ make                              [Builds the Android Platform]
```

On a side note, keep in mind that it is possible to rebuild only one project (such as the CyaSSL shared library) to test that the shared library builds correctly using the "mm" command:

```
$ cd external/cyassl
$ mm
```

The result of the complete Android platform build process is three main image files:

```
<Android-Platform>/out/target/product/generic/ramdisk.img
<Android-Platform>/out/target/product/generic/system.img
<Android-Platform>/out/target/product/generic/userdata.img
```

ramdisk.img – A small partition which is mounted as read-only by the kernel at boot time, it only contains /init and a few configuration files. It is used to start /init which will boot the rest of the system images and run the init procedure.

system.img – A partition image that will be mounted as / and contains all system binaries. In other words, this is the image file that contains all of the changes that were made above. This image will be the highest concern when testing with the emulator.

userdata.img – This image is only used when the –wipe-data option is used with the emulator. In a normal emulator execution, a default userdata image will be used. The –wipe-data option copies the contents of userdata.img into the default userdata image, therefore, not saving any data from a previous session.

Of these, system.img is of the highest concern. This is the image that contains the majority of the system, and all of the changes that have been made with the addition of the CyaSSL SSL Provider.

Emulator Execution

At this point, the Android SDK should already be downloaded. If not, refer to Section 6, "The Android Emulator – Making Life Simpler." Once the SDK has been downloaded, the emulator will be located in the <Android-SDK>/tools directory.

To use the emulator, an Android Virtual Device must first be created. Android Virtual Devices are configurations of emulator options, which in turn allow developers to better model a physical android device. They hold configuration information such as a hardware profile, a mapping to a system image, a dedicated storage area, and much more. To create an Android Virtual Device, the "android" application is used. This application (android) is found under the tools directory of the SDK as well. A new Virtual Device may be created using the following command (issued from the /tools directory):

```
$ android create avd -n <desired-name> -t <target-version>
```

Where <desired-name> is what the Android Virtual Device will be called and <target-version> is the desired target platform. Available targets can be viewed by running the following command:

```
$ android list targets
```

After the Android Virtual Device has been created, the emulator can be loaded with the built images using the following command: (Note that although this command spans multiple lines, it should be written on a single)

```
$ emulator -avd <virtual-device-name> -system <Android-Platform>/out/
target/product/generic/system.img -data <Android-Platform>/out/
target/product/generic/userdata.img -ramdisk <Android-Platform>/
out/target/product/generic/ramdisk.img
```

There are several other useful emulator options that may be added to the above command if desired. A few are listed below, but for a complete list see the official Android Emulator webpage (http://developer.android.com/guide/developing/tools/emulator.html)

-verbose	[Verbose Output]
-nocache	[Don’t use a cache]
-show-kernel	[Print Kernel messages to the terminal window]

Once the emulator is running, the logcat output can be viewed by running the following command in a new terminal window (Assuming the current directory is <Android-SDK>/tools):

```
$ adb logcat
```

XI. WRITING AN ANDROID APPLICATION USING SSL

After the CyaSSL Java SSL provider has been installed and built into the Android platform, it is helpful to create a simple Android application that makes use of the javax.net.ssl package to make sure the provider was installed correctly. Details regarding the setup of the Android Application development environment will be omitted, and the focus will lie on the application code itself. For instructions on setting up a development environment for Android applications (Using Eclipse), please refer to the official Android documentation (http://developer.android.com/sdk/index.html).

The following steps will explain how to write a simple SSL client application for Android. This application is very simple and minimalistic – acting only as a simple client that makes a “GET” request to “www.google.com” using SSL. A TextView is used to display text to the Android screen. After creating a new project, the following code will go inside of the “public void onCreate()” method. For source code examples similar to this, download the CyaSSL Java SSL Provider from the yaSSL website and look in the ‘examples’ directory: (http://yassl.com/yaSSL/Download_More.html).

1. First off, set the ContentView and create a new TextView, allowing it to scroll and setting a default text size:

```
setContentView(R.layout.main);
TextView tv = new TextView(this);
tv.setMovementMethod(new ScrollingMovementMethod());
tv.setTextSize(12);
```

2. Create a Socket Factory using the default Socket Factory (as defined in the java.properties file):

```
// Create a socket factory
SSLSocketFactory f = (SSLSocketFactory) SSLSocketFactory.getDefault();
```

3. Create a Socket and print out the Socket information. Also create a BufferedReader and BufferedWriter for reading and writing to the socket, and a new string to hold the response from Google. The printSocketInfo(...) method will be shown later.

```
// Create a socket, print socket info
SSLSocket c = (SSLSocket) f.createSocket("www.google.com", 443);
printSocketInfo(c, tv);

BufferedWriter w = new BufferedWriter(new
                              OutputStreamWriter(c.getOutputStream()));
 BufferedReader r = new BufferedReader(new
                              InputStreamReader(c.getInputStream()));
String m = null;
```

4. Send our GET message and flush the BufferedWriter:

```
w.write("GET / HTTP/1.0\n\n");
w.flush();
```

5. While we are still getting information back through the socket from www.google.com, append it to the TextView. Close the BufferedReader, BufferedWriter, and SSLSocket:

```
while((m = r.readLine()) != null){
        tv.append(m+"\n");
}
w.close();
r.close();
c.close();
```

6. Set the ContentView to the TextView:

```
setContentView(tv);
```

7. Here is the printSocketInfo(...) function, which prints information about the socket and session that have been created:

```
public void printSocketInfo(SSLSocket s, TextView tv)
  {
     tv.append("\n Socket class: "+s.getClass());
     tv.append("\n Remote address = "+s.getInetAddress().toString());
     tv.append("\n Remote port = "+s.getPort());
     tv.append("\n Local socket address = "+s.getLocalSocketAddress().toString());
     tv.append("\n Local address = "+s.getLocalAddress().toString());
     tv.append("\n Local port = "+s.getLocalPort());
     tv.append("\n Need client authentication = "+s.getNeedClientAuth());

     SSLSession ss = s.getSession();

     tv.append("\n Cipher suite = "+ss.getCipherSuite());
     tv.append("\n Protocol = "+ss.getProtocol());
     tv.append("\nPeer Host = "+ ss.getPeerHost());
      tv.append("\nPeer Port = "+ ss.getPeerPort());
     tv.append("\nIs Valid = "+ss.isValid());
     try {
         tv.append("\nPeer Principal Name = " + ss.getPeerPrincipal().getName());
     } catch (SSLPeerUnverifiedException ex) {
         Logger.getLogger(SSLTest.class.getName()).log(Level.SEVERE, null, ex);
     }

     String[] names = ss.getValueNames();
```

```
        for(String name: names){
            tv.append("\nName = "+name);
        }
        tv.append("\n\n");
    }
```

XII. IMPORTING ADDITIONAL CERTIFICATES INTO ANDROID

Just like a driver's license identifies a person, a certificate provides identification for the online world – identifying things such as servers and clients. SSL certificates are issued individually to a specific domain and server and are authenticated by an SSL Certificate provider. When a browser connects to a server, the server sends its certificate to the browser, allowing the browser to verify the server to which it has connected.

Additional certificates may be added to an Android phone by loading them from an SD card through the Settings menu:

1. Go to Phone Settings
2. Click on Location & Security
3. Click on Install from SD Card

The only supported certificate format at the time of writing is the .p12 format.

XIII. TROUBLESHOOTING

For help resolving issues relating to the material covered in this White Paper, please contact support@yassl.com.

XIV. CONCLUSION

In this White Paper, we have covered how to install a Java SSL provider into the Android platform, more specifically, the CyaSSL Java SSL Provider. The Android build process was explained briefly, along with the Android emulator. By using CyaSSL in the Android platform for application development instead of OpenSSL, developers are able to leverage both the speed and size advantages of the CyaSSL library. The CyaSSL Java SSL provider can be downloaded from the yaSSL Website for OS X, Linux, and Android. Additionally, yaSSL offers downloads for CyaSSL, yaSSL, yaSSH, and the yaSSL Embedded Web Server.

XV. ABOUT THE AUTHOR

Chris Conlon
yaSSL
chris@yassl.com

Chris Conlon is a developer at yaSSL who recently began working with the Android platform. Finding a balance between outdoor adventures and computing, Chris enjoys continually learning and strives to bring new and helpful things to the technology community.

yaSSL (yet another SSL) is an open source Internet security company who's primary product is an embedded SSL Library called CyaSSL, but also creates yaSSL, yaSSH, and the CyaSSL Embedded Web Server. Its primary users are programmers building security functionality into their applications and devices. yaSSL employs the dual licensing model, like MySQL, so it is available under GPL and it is also available under commercial license terms. Support and consulting are available for yaSSL as well. For any comments, questions, or inquiries, please contact info@yassl.com.

yaSSL
1627 West Main St., Suite 237
Bozeman, MT 59715 USA
http://www.yassl.com

General Questions	**Licensing Questions**	**Support:**
Email: info@yassl.com	Email: licensing@yassl.com	Email: support@yassl.com
Phone: +1 (206) 369-4800		

XVI. REFERENCES

Google. (n.d.). Get Android Source Code. Retrieved from http://source.android.com/source/download.html

Oracle. (n.d.). How to Implement a Provider in the Java ™ Cryptography Architecture. Retrieved from http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html

Transport Layer Security. (2010, August 16). In Wikipedia, The Free Encyclopedia. Retrieved from http://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=379191826

yaSSL. (2010). Retrieved from http://www.yassl.com/yaSSL/Products.html