NetCipher

NetCipher is a library from the Guardian Project to improve the privacy and security of HTTP network communications. In particular, it makes it easier for your app to integrate with Orbot, an Android proxy server that forwards HTTP requests via Tor.

This is a very long chapter. Most likely, you do not need all of it.

It is divided into four main parts:

- An introduction to Tor, Orbot, and NetCipher
- An explanation of how to use a fairly simple API layered atop NetCipher to add its functionality to your app
- An explanation of the more extensive builder API that is included as part of this chapter's set of samples
- An explanation of how that builder API is implemented, showing NetCipher's "raw" API

Prerequisites

This chapter assumes that you have read the core chapters of the book, particularly the one on Internet access. Having read the chapter on SSL is also a very good idea.

Network Security's Got Onions

Maintaining privacy and security on the Internet, in the face of so-called "advanced persistent threats", is a continuous challenge facing many people, particularly those under threats from hostile forces, ranging from organized crime syndicates to your

average rampaging warlord. Tor was created to help deal with this sort of problem; Orbot was created to extend Tor to Android.

A Quick Primer on Tor

Originally named The Onion Router, Tor was created by researchers in the US Naval Research Laboratory back in the mid-1990's, with an eye towards protecting US intelligence communications. In 2006, the technology spun out into an independent non-profit organization, which has continued to improve upon the core Tor software and expand the reach of Tor. Through packages like the Tor Browser Bundle, it is fairly easy for at-risk people to start using Tor to help shroud their communications.

Without getting into the full technical details of Tor — which are *well* beyond the scope of this chapter — Tor basically works by routing a request through a series of relay servers, through a process known as onion routing. Requests are secured through layers of encryption, to keep any two connected relays from knowing the full details of the communications. Some relays serve as "exit nodes", for requests being made of ordinary Web servers. Certain servers — Tor hidden services — are only reachable through Tor; requests made of these servers never leave the Tor network.

Of course, technology like Tor is agnostic in terms of its users and usages, and there have been plenty of examples of people using Tor for illicit purposes, such as the Silk Road. This has a tendency to obscure Tor's benefits to people who need to remain somewhat hidden online, whether from stalkers or other harassers or from the security forces of dictatorships.

Introducing Orbot

The entry path into Tor is usually via some sort of proxy server, that a regular Internet client can connect to. Orbot is one such proxy server, that runs on Android. Apps can use Orbot's HTTP or SOCKS proxies to route requests; those requests will then wind up traversing the Tor network to the end site, whether that site is on the public Internet (reached from a Tor exit node) or a Tor hidden service.

By default, Orbot is limited to `localhost` use, meaning that it does not have open ports that can be reached from other devices on the local WiFi LAN segment (or some subnet of the mobile carrier, if not on WiFi). For an Android app on the same device, this is not a problem, and it in fact simplifies things a fair bit, as there is no

guesswork as to what the IP address should be for the proxy. As we will see, though, finding out exactly how to connect to Orbot is a bit tricky, though with some helper code it is not too bad.

What NetCipher Provides

NetCipher serves two primary roles:

- Make it easier for app developers to tie into Orbot, and Tor by extension
- Provide other stock improvements to network security, particularly surrounding SSL certificates

Bridge to Orbot

While we know that Orbot will be listening on `localhost`, we do not necessarily know the port that it is using for its HTTP proxy. Partly, that is because the user might configure it manually. Partly, that is because there are occasional conflicts with Orbot's default port.

Hence, NetCipher contains some code that will help you find out:

- Is Orbot installed? (and, if not, help get it installed)
- Is Orbot running? (and, if not, help get it running)
- What port is used for the HTTP proxy?

Access to Debian Root Certificate Store

SSL validation relies upon being able to take the SSL certificate for a given HTTP request and trace it back to a known good "root" certificate. For a self-signed SSL certificate, it is its own root, which is why you have to teach HTTP APIs about that self-signed certificate. For more conventional SSL certificates, there is a certificate "chain" from the one you get in an HTTP response that eventually should lead to a root certificate.

The problem is that Android root certificates are part of the OS and therefore are not updated all that frequently, except perhaps on devices that are getting monthly security updates. However, root certificates change, either because new trusted ones get created or because previously trusted ones get removed due to security concerns. While desktop Web browsers often get these changes quickly, apps on Android have two choices:

1. Rely on the system-supplied roster of root certificates and hope for the best
2. Package their own roster of root certificates, and make sure to keep them updated

NetCipher ships with its own copy of the root certificates used by the Debian distribution of Linux, and it will use those root certificates by default. These are newer than the certificates on most Android devices. There are also hooks for you to supply your own root certificate store, in case you want to update the certificates frequently but not necessarily update NetCipher itself frequently (e.g., due to changing APIs).

The Easy API

The `Internet/HTTPStacks` sample application for this chapter serves two roles. First, as with most of the book samples, it illustrates how to use certain APIs. Second, it creates wrapper APIs that simplify the use of NetCipher considerably. Those wrapper APIs are in the form of separate library modules that you could use in other apps if so desired.

There are two sample apps in the project: `bigsample` and `tinysample`.

This chapter will review the `tinysample` app, which is based off of prior samples that show the latest `android` Stack Overflow questions in a `ListView`. In this case, we will use NetCipher to obtain those questions by way of Orbot and Tor, using `HttpURLConnection`.

The `bigsample` app also shows the latest Stack Overflow questions about Android. However, it does this using a `ViewPager` with ten different tabs:

- Four examples of using plain HTTP stacks (`HttpURLConnection`, an independent packaging of HttpClient, OkHttp3, and Volley)
- Four examples of using those in concert with NetCipher
- Two examples using Retrofit with OkHttp3, one with and one without NetCipher

Hence, `bigsample` is a rather complex app, given the sheer number of HTTP stacks involved and trying to minimize the code duplication between them. Also, `bigsample` handles some things that `tinysample` does not, such as confirming that our HTTP connection is indeed going by way of Tor (for the NetCipher editions). The book does not review `bigsample`, due to sheer size.

With that in mind, let's review `tinysample` and see how to hook into NetCipher using the library modules in the overall project.

Choose an HTTP Stack

The sample project offers simplified NetCipher configuration for four major HTTP client implementations (a.k.a., "HTTP stacks"):

- `HttpURLConnection`
- OkHttp3
- Apache's independent HttpClient package
- Volley

There are corresponding library modules for each of those HTTP stacks:

HTTP Stack	Library Module
`HttpURLConnection`	`netcipher-hurl`
OkHttp3	`netcipher-okhttp3`
HttpClient	`netcipher-httpclient`
Volley	`netcipher-volley`

The `netcipher-hurl` library module not only supports `HttpURLConnection`, but it also is a dependency for the other three modules, so if you choose one of those, you will also need `netcipher-hurl`. However, that will be added for you automatically via transitive dependencies, assuming that you have copied the modules (and the `libnetcipher` module) into your project.

Add the Dependencies

Given that the project has the necessary modules, you can add a `compile project()` statement to your `dependencies` closure to pull in the NetCipher HTTP stack integration, along with NetCipher itself.

The `tinysample` uses `HttpURLConnection`, so it pulls in `:netcipher-hurl` as a dependency:

```
apply plugin: 'com.android.application'

repositories {
    maven {
        url "https://s3.amazonaws.com/repo.commonsware.com"
    }
}
```

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.4-beta1'
    releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.4-beta1'
    compile 'com.google.code.gson:gson:2.5'
    compile project(':netcipher-hurl')
}

android {
    compileSdkVersion 23
    buildToolsVersion "23.0.1"

    defaultConfig {
        applicationId "com.commonsware.android.http.tiny"
    }
}
```

(from Internet/HTTPStacks/tinysample/build.gradle)

`tinysample` will use Gson for JSON parsing, plus it uses LeakCanary to check for leaks, so those are listed as dependencies as well.

Set up OrbotInitializer

`OrbotInitializer` is a singleton that manages a lot of the asynchronous communication between your app and Orbot. It is designed to be initialized fairly early on in your app's lifecycle. One likely candidate is to have a custom `Application` subclass, where you override `onCreate()` and set up `OrbotInitializer`.

`tinysample` does this in a custom `SampleApplication` class:

```
package com.commonsware.android.http;

import android.app.Application;
import com.squareup.leakcanary.LeakCanary;
import info.guardianproject.netcipher.hurl.OrbotInitializer;

public class SampleApplication extends Application {
  @Override
  public void onCreate() {
    super.onCreate();

    LeakCanary.install(this);
    OrbotInitializer.get(this).init();
  }
}
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/SampleApplication.java)

This custom `Application` also sets up LeakCanary.

`SampleApplication` is then tied into the app via the `android:name` attribute on the `<application>` element in the manifest:

```
<application
  android:name=".SampleApplication"
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/Theme.Apptheme">
```

(from Internet/HTTPStacks/tinysample/src/main/AndroidManifest.xml)

Choose a Builder

Each module defines a corresponding builder class that can be used to configure NetCipher for use with that stack, with names based on the classes used with those HTTP stacks:

HTTP Stack	Builder Class
`HttpURLConnection`	`StrongConnectionBuilder`
OkHttp3	`StrongOkHttpClientBuilder`
HttpClient	`StrongHttpClientBuilder`
Volley	`StrongVolleyQueueBuilder`

Create a Builder

You will need an instance of your chosen builder class. The simplest way to do that is to call the `forMaxSecurity()` `static` method on the builder class. `forMaxSecurity()` takes a `Context` as a parameter, though it only holds onto the `Application` singleton internally, so any `Context` is safe. `forMaxSecurity()` returns a builder configured for the best protection that NetCipher can offer.

Get a Connection

Then, call `build()` on the builder object. It will take a `StrongBuilder.Callback` object as a parameter, typed for whatever HTTP stack you chose. So, for example, if you went with `StrongConnectionBuilder`, your callback will be a `StrongBuilder.Callback<HttpURLConnection>`.

HTTP Stack	Builder Class	Connection Class
`HttpURLConnection`	`StrongConnectionBuilder`	`HttpURLConnection`
OkHttp3	`StrongOkHttpClientBuilder`	`OkHttpClient`

HTTP Stack	Builder Class	Connection Class
HttpClient	`StrongHttpClientBuilder`	`HttpClient`
Volley	`StrongVolleyQueueBuilder`	`RequestQueue`

You will need to implement three methods on that `Callback`:

- `onConnected()` will be passed an instance of your connection class (e.g., an `HttpURLConnection` instance), ready for your use, configured to hook into NetCipher
- `onConnectionException()` will be passed an `IOException`, if one of those occurs while trying to set up your connection
- `onTimeout()` will be called if Orbot is not installed or we could not connect to it within 30 seconds

Seeing the Builder in Action

The `MainActivity` in `tinysample` creates a `StrongConnectionBuilder` in `onCreate()` and calls `build()` on it to set up a secured `HttpURLConnection`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  try {
    StrongConnectionBuilder
      .forMaxSecurity(this)
      .connectTo(SO_URL)
      .build(this);
  }
  catch (Exception e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }
}
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/MainActivity.java)

`StrongConnectionBuilder` also requires that you call `connectTo()`, before `build()`, to indicate the specific URL for which you want an `HttpURLConnection`. This is unique among the builders. These sorts of per-builder differences are discussed later in this chapter.

`build()` is passed `this`, referencing `MainActivity` itself, which is implementing the `StrongBuilder.Callback` interface:

```
public class MainActivity extends ListActivity implements
  StrongBuilder.Callback<HttpURLConnection> {
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/MainActivity.java)

`SO_URL`, passed into `connectTo()`, is a Web service request URL from the Stack Exchange API, looking for Stack Overflow questions tagged with the `android` tag:

```
  String SO_URL=
    "https://api.stackexchange.com/2.1/questions?"
      + "order=desc&sort=creation&site=stackoverflow&tagged=android";
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/MainActivity.java)

Because `MainActivity` implements the `StrongBuilder.Callback` interface, we have three methods that we need to implement. Two are for error conditions: `onConnectionException()` and `onTimeout()`:

```
  @Override
  public void onConnectionException(IOException e) {
    Toast
      .makeText(this, R.string.msg_crash, Toast.LENGTH_LONG)
      .show();
    Log.e(getClass().getSimpleName(),
      "Exception loading SO questions", e);
    finish();
  }

  @Override
  public void onTimeout() {
    Toast
      .makeText(this, R.string.msg_timeout, Toast.LENGTH_LONG)
      .show();
    finish();
  }
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/MainActivity.java)

The more positive case is `onConnected()`, where we are handed our `HttpURLConnection` set up for NetCipher, and we can retrieve our Web service results. Note that `onConnected()` will be called on the main application thread, so you will need to get your connection over to whatever background thread will be doing your work. In this case, we create a background thread right here to retrieve the JSON, parse it, and use `runOnUiThread()` to update the `ListActivity` with an `ItemsAdapter` to show the parsed Stack Overflow questions:

```
  @Override
  public void onConnected(final HttpURLConnection conn) {
```

```
    new Thread() {
      @Override
      public void run() {
        try {
          InputStream in=conn.getInputStream();
          BufferedReader reader=
            new BufferedReader(new InputStreamReader(in));

          final SOQuestions result=
            new Gson().fromJson(reader, SOQuestions.class);

          runOnUiThread(new Runnable() {
            @Override
            public void run() {
              setListAdapter(new ItemsAdapter(result.items));
            }
          });

          reader.close();
        }
        catch (IOException e) {
          onConnectionException(e);
        }
        finally {
          conn.disconnect();
        }
      }
    }.start();
  }
```

(from Internet/HTTPStacks/tinysample/src/main/java/com/commonsware/android/http/MainActivity.java)

Other than initializing `OrbotInitializer`, setting up the builder, and implementing `StrongBuilder.Callback` somewhere to handle the results, the rest of the code is tied to application logic, not NetCipher itself.

The Rest of the Builder API

The API shown above for getting a NetCipher-secured connection via your favorite HTTP stack is designed for ease of use. However, as shown, it is not very flexible.

The rest of the builder API offers that flexibility, at the cost of some additional code.

Common Configuration Methods

The `StrongBuilder` interface defines the common public API for all four of the builder classes:

```
package info.guardianproject.netcipher.hurl;
```

```
import android.content.Intent;
import java.io.IOException;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;

public interface StrongBuilder<T extends StrongBuilder, C> {
  /**
   * Callback to get a connection handed to you for use,
   * already set up for NetCipher.
   *
   * @param <C> the type of connection created by this builder
   */
  interface Callback<C> {
    /**
     * Called when the NetCipher-enhanced connection is ready
     * for use.
     *
     * @param connection the connection
     */
    void onConnected(C connection);

    /**
     * Called if we tried to connect through to Orbot but failed
     * for some reason
     *
     * @param e the reason
     */
    void onConnectionException(IOException e);

    /**
     * Called if our attempt to get a status from Orbot failed
     * after a defined period of time. See statusTimeout() on
     * OrbotInitializer.
     */
    void onTimeout();
  }

  /**
   * Call this to configure the Tor proxy from the results
   * returned by Orbot, using the best available proxy
   * (SOCKS if possible, else HTTP)
   *
   * @return the builder
   */
  T withBestProxy();

  /**
   * @return true if this builder supports HTTP proxies, false
   * otherwise
   */
  boolean supportsHttpProxy();

  /**
   * Call this to configure the Tor proxy from the results
   * returned by Orbot, using the HTTP proxy.
   *
```

```
     * @return the builder
     */
     T withHttpProxy();

    /**
     * @return true if this builder supports SOCKS proxies, false
     * otherwise
     */
     boolean supportsSocksProxy();

    /**
     * Call this to configure the Tor proxy from the results
     * returned by Orbot, using the SOCKS proxy.
     *
     * @return the builder
     */
    T withSocksProxy();

    /**
     * Replaces system-supplied keystore with one based on Debian.
     * Use this if you are keeping your app up to date with the
     * latest NetCipher library and are supporting older devices
     * (e.g., Android 4.4 and lower).
     *
     * @return the builder
     * @throws CertificateException
     * @throws NoSuchAlgorithmException
     * @throws KeyStoreException
     * @throws IOException
     * @throws UnrecoverableKeyException
     * @throws KeyManagementException
     */
    T withDefaultKeystore()
      throws CertificateException, NoSuchAlgorithmException,
      KeyStoreException, IOException, UnrecoverableKeyException,
      KeyManagementException;

    /**
     * Applies your own custom keystore, instead of either the
     * system-supplied keystore or the default NetCipher keystore.
     *
     * @param keystore a loaded KeyStore ready for use
     * @return the builder
     */
    T withKeystore(KeyStore keystore)
      throws KeyStoreException, NoSuchAlgorithmException,
      IOException, CertificateException,
      UnrecoverableKeyException, KeyManagementException;

    /**
     * Call this if you want a weaker set of supported ciphers,
     * because you are running into compatibility problems with
     * some server due to a cipher mismatch. The better solution
     * is to fix the server.
     *
     * @return the builder
     */
    T withWeakCiphers();

    /**
```

```
   * Builds a connection, applying the configuration already
   * specified in the builder.
   *
   * @param status status Intent from OrbotInitializer
   * @return the connection
   * @throws IOException
   */
  C build(Intent status) throws IOException;

  /**
   * Asynchronous version of build(), one that uses OrbotInitializer
   * internally to get the status.
   *
   * @param callback Callback to get a connection handed to you
   *                 for use, already set up for NetCipher
   */
  void build(Callback<C> callback);
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/StrongBuilder.java)

Proxy Configuration

Five of the methods are tied into choosing what proxy protocol should be used with Orbot.

`forMaxSecurity()`, under the covers, uses `withBestProxy()`, which chooses the best proxy for the situation. Right now, the implementation chooses the SOCKS proxy where that is supported, falling back to the HTTP proxy where it is not.

The `supportsHttpProxy()` and `supportsSocksProxy()` methods indicate whether a given builder supports these proxy types.

The `withHttpProxy()` and `withSocksProxy()` methods tell the builder that you want to use that specific proxy. Use these with care, making sure that the proxy you want is supported. `withBestProxy()` is a far better choice overall.

Other Configuration

`forMaxSecurity()` also calls `withDefaultKeystore()`, indicating that we should try to use NetCipher's built-in roster of root certificates. If you prefer, you can call `withKeystore()` and supply your own `KeyStore` of root certificates to use. Note that you will be responsible for initializing this `KeyStore` yourself, which usually involves baking a password into your app.

`withWeakCiphers()` expands the roster of SSL ciphers that NetCipher allows the HTTPS connection to use. Normally, NetCipher tries to avoid ciphers with known

security issues. However, that may cause problems with some servers, if NetCipher and the server cannot negotiate a common cipher. `withWeakCiphers()` allows NetCipher to use more ciphers, to perhaps overcome the negotiation problem, with the cost of possibly weaker security.

Differences Between the Stacks

While each of the builders supports the `StrongBuilder` API, there are some differences between the implementations.

StrongConnectionBuilder

As noted previously, before calling `build()`, you need to call `connectTo()` to supply the URL (as a `String` or `URL`) that you want to connect to. The other builders give you objects that you can reuse across many requests (e.g., OkHttp3's `OkHttpClient`), but that was not possible with `HttpURLConnection`.

To help make this a bit easier, `StrongConnectionBuilder` supports the copy constructor. You can create a master `StrongConnectionBuilder` with your base configuration, then make a copy, call `connectTo()` on the copy, then call `build()` on the copy, throwing away the copy when you are done.

StrongHttpClientBuilder

The builder for Apache's independent packaging of HttpClient for Android extends Apache's own `HttpClientBuilder`. As a result, you can call all the normal `HttpClientBuilder` methods in addition to calling the `StrongBuilder` methods. The noteworthy exception is that the standard zero-parameter `build()` offered by `HttpClientBuilder` is not supported.

StrongOkHttpClientBuilder

OkHttp3 does not support SOCKS proxies. Hence, `supportsSocksProxy()` returns `false`, causing `withBestProxy()` to fall back to the HTTP proxy.

StrongVolleyQueueBuilder

This builder class adheres to the `StrongBuilder` API without any changes.

Inside the Builder API

If all you want to do is use that code, you are set.

If you want to understand how that code works, or you want to understand more about NetCipher's own API, this section is for you.

What We Need to Do to Use NetCipher

There are two key steps to plug your code into NetCipher: adding the root certificate keystore and adding the Orbot proxy to your HTTP client for whatever HTTP implementation that you are using (e.g., an `OkHttpClient` for OkHttp3). Adding the proxy is a bit involved, simply because Orbot is a separate app, which may or may not be installed or running at the present time.

Adding the Keystore

First, you need to add a keystore that contains those Debian root certificates. This involves creating a `TrustManager` that is based on those certificates. But, as developers have become aware, it is important to create a `TrustManager` that is well-written, as Google is kicking insecure implementations out of the Play Store.

The Debian root certificate keystore, at the moment, is packaged as a raw resource in NetCipher. This poses some problems for other library modules that try to use NetCipher due to the way that `R` values get generated. The sample app for this project has a separate copy of that keystore in `assets/`, to make it easier for both application and library modules to reference this keystore.

The problem is that this keystore will itself become out of date. It will be important for you to keep NetCipher up to date, so your copy of the keystore can be current with respect to any new or removed root certificates.

I Can Haz Orbot?

Orbot may not be installed. In that case, you cannot use it, unless the user elects to install it.

NetCipher's `OrbotHelper` class has an `isOrbotInstalled()` `static` method that returns a simple `boolean` indicating whether or not Orbot is installed.

If it is not, `OrbotHelper` has another `static` method, `getOrbotInstallIntent()`, that returns an `Intent` that you can use with `startActivity()` to help the user install Orbot from the Play Store or F-Droid.

So, for example, if Orbot is installed, you continue to set it up, but if Orbot is not installed, you offer an action bar item or a `Preference` that, when tapped, triggers the install `Intent` to go install Orbot.

Getting the Proxy Port

If Orbot is installed, it may or may not be running, and even if it is running, it may or may not be fully connected yet to Tor. Orbot takes a while to establish a Tor connection, which is a common issue with Tor clients.

Orbot supports a status broadcast. If you send this broadcast, Orbot will send broadcasts back to you based on the changes in Orbot's status.

Once Orbot has a Tor connection, the status return broadcast will tell you the proxy port, based on user configuration and any dynamic changes that were needed to avoid collisions with other apps.

`OrbotHelper` has a `requestStartTor()` `static` method that sends the broadcast. `OrbotInitializer` uses a different approach, to improve upon `OrbotHelper` in a few ways:

- Managing the `BroadcastReceiver` needed to listen for the response(s) from Orbot
- Caching the Orbot status across configuration changes
- Confirming that the installed copy of Orbot is indeed really Orbot, and not a hacked version signed with the wrong signing key

We will explore the implementation of `OrbotInitializer` later in this chapter.

Confirming the Connection

Given that Orbot is running and says that it has a Tor connection, and given that you have a port number for the Orbot proxy server, you can teach your HTTP client API about that proxy. The details of that vary by HTTP client, and we will explore those details later in the chapter.

However, for your user's sake, it would be a good idea to try to confirm that you have all of this set up properly and that your HTTP client will be communicating over Tor.

There does not appear to be a formal way of going about this, unfortunately. The `bigsample` sample app demonstrates hitting a Tor status URL that returns a JSON payload, indicating whether or not your request came via Tor.

Inside OrbotInitializer

To understand the builders, we must first understand `OrbotInitializer`. `OrbotInitializer` can:

- install Orbot if it is not available
- start Orbot if it is available
- get the status information about Orbot, including proxy ports to use

These things require IPC, such as a broadcast to Orbot to start it up (if needed) and get its status. IPC in Android is largely asynchronous, and that gets to be a pain when it comes to configuration changes destroying and recreating our activities and fragments.

The Interfaces

`OrbotInitializer` relies upon two interfaces for notifying builders (or other clients) about our connection with Orbot.

One is `StatusCallback`, for overall status events:

```
package info.guardianproject.netcipher.hurl;

import android.content.Intent;

/**
 * Callback interface used for reporting Orbot status
 */
public interface StatusCallback {
  /**
   * Called when Orbot is operational
   *
   * @param statusIntent an Intent containing information about
   *                     Orbot, including proxy ports
   */
  void onEnabled(Intent statusIntent);

  /**
```

```
   * Called when Orbot reports that it is starting up
   */
  void onStarting();

  /**
   * Called when Orbot reports that it is shutting down
   */
  void onStopping();

  /**
   * Called when Orbot reports that it is no longer running
   */
  void onDisabled();

  /**
   * Called if our attempt to get a status from Orbot failed
   * after a defined period of time. See statusTimeout() on
   * OrbotInitializer.
   */
  void onStatusTimeout();

  /**
   * Called if Orbot is not yet installed. Usually, you handle
   * this by checking the return value from init() on OrbotInitializer
   * or calling isInstalled() on OrbotInitializer. However, if
   * you have need for it, if a callback is registered before
   * an init() call determines that Orbot is not installed, your
   * callback will be called with onNotYetInstalled().
   */
  void onNotYetInstalled();
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/StatusCallback.java)

The three callback methods that will concern developers most here are `onEnabled()` (Orbot is ready to go), `onStatusTimeout()` (Orbot is installed but we did not get a response in a timely fashion), and `onNotYetInstalled()` (Orbot does not exist on the device).

There is also `InstallCallback`, an inner interface of `OrbotInitializer`, that handles events related to the Orbot installation process:

```
  /**
   * Callback interface used for reporting the results of an
   * attempt to install Orbot
   */
  public interface InstallCallback {
    void onInstalled();
    void onInstallTimeout();
  }
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Setting up OrbotInitializer

`OrbotInitializer` is a singleton:

```
  private static volatile OrbotInitializer INSTANCE;
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Clients access that singleton via the static `get()` method, which lazy-creates the `OrbotInitializer` if needed:

```
  synchronized public static OrbotInitializer get(Context ctxt) {
    if (INSTANCE==null) {
      INSTANCE=new OrbotInitializer(ctxt);
    }

    return(INSTANCE);
  }
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

The `OrbotInitializer` `private` constructor holds onto the `Application` singleton (for use as non-leakable `Context`) and a `Handler` tied to the main application thread. You can set that up by calling `getMainLooper()` on the `Looper` class (returning the `Looper` tied to the main application thread), then passing that `Looper` instance to the `Handler` constructor:

```
  private OrbotInitializer(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
    this.handler=new Handler(Looper.getMainLooper());
  }
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

`OrbotInitializer` maintains collections of `StatusCallback` and `InstallCallback` implementations:

```
  private WeakSet<StatusCallback> statusCallbacks=new WeakSet<>();
  private WeakSet<InstallCallback> installCallbacks=new WeakSet<>();
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Here, `WeakSet` is a utility class that wraps a `HashSet` and holds onto all of its elements through `WeakReferences`:

```
package info.guardianproject.netcipher.hurl;

import java.lang.ref.WeakReference;
import java.util.HashSet;
import java.util.Iterator;
```

```
import java.util.Set;

// inspired by https://github.com/explodes/easy-android/blob/master/src/io/explod/android/collections/weak/
WeakList.java

/**
 * Weak implementation of a set. Elements are held weakly and
 * therefore may vanish due to GC, but that is all hidden by the
 * implementation.
 *
 * @param <T> The type of data that the set "holds"
 */
public class WeakSet<T> implements Iterable<T> {
  private final Set<WeakReference<T>> items=
    new HashSet<WeakReference<T>>();

  /**
   * Add an item to the set. Under the covers, this gets wrapped
   * in a WeakReference.
   *
   * @param item item to add
   * @return true if added successfully, false otherwise
   */
  public boolean add(T item) {
    return(items.add(new WeakReference<T>(item)));
  }

  /**
   * Removes an item from the set. Under the covers, uses
   * the WeakIterator to find this, also cleaning out dead
   * wood along the way.
   *
   * @param item item to remove
   * @return true if item removed successfully, false otherwise
   */
  public boolean remove(T item) {
    final Iterator<T> iterator=iterator();

    while (iterator.hasNext()) {
      if (iterator.next()==item) {
        iterator.remove();

        return(true);
      }
    }

    return(false);
  }

  /**
   * Used to support Iterable, so a WeakSet can be used in
   * Java enhanced for syntax
   *
   * @return a WeakIterator on the set contents
   */
  @Override
  public Iterator<T> iterator() {
    return(new WeakIterator());
  }
```

```
  // inspired by https://github.com/explodes/easy-android/blob/master/src/io/explod/android/collections/
weak/WeakIterator.java

  /**
   * Iterator over the contents of the WeakSet, skipping over
   * GC'd items
   */
  class WeakIterator implements Iterator<T> {
    private final Iterator<WeakReference<T>> itemIterator;
    private T nextItem=null;

    /**
     * Constructor. Creates the itemIterator that is the
     * "real" iterator for the underlying collection. Calls
     * moveToNext() to set the iterator (and nextItem) to the
     * first non-GC'd entry.
     */
    WeakIterator() {
      itemIterator=items.iterator();
      moveToNext();
    }

    /**
     * @return true if we have data, false otherwise
     */
    @Override
    public boolean hasNext() {
      return(nextItem!=null);
    }

    /**
     * Moves to the next item, skipping over GC'd items.
     *
     * @return the current item before the move
     */
    @Override
    public T next() {
      T result=nextItem;

      moveToNext();

      return(result);
    }

    /**
     * Removes whatever was last returned by next()
     */
    @Override
    public void remove() {
      itemIterator.remove();
    }

    private void moveToNext() {
      nextItem=null;

      while (nextItem==null && itemIterator.hasNext()) {
        nextItem=itemIterator.next().get();

        if (nextItem==null) {
          remove();
```

```
        }
      }
    }
  }
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/WeakSet.java)

This way, even if a client of `OrbotInitializer` fails to unregister a listener, we will not have a memory leak.

`OrbotInitializer` then has add/remove methods to manipulate those collections of callbacks. `removeStatusCallback()`, `addInstallCallback()` and `removeInstallCallback()` just update the collections using `add()` and `remove()`. `addStatusCallback()` does that and a little bit more, as we will see in the next section.

There are also `statusTimeout()` and `installTimeout()` configuration methods, to override the default timeouts for status checks and installations:

```
/**
 * Sets how long of a delay, in milliseconds, after trying
 * to get a status from Orbot before we give up.
 * Defaults to 30000ms = 30 seconds = 0.000347222 days
 *
 * @param timeoutMs delay period in milliseconds
 * @return the singleton, for chaining
 */
public OrbotInitializer statusTimeout(long timeoutMs) {
  statusTimeoutMs=timeoutMs;

  return(this);
}

/**
 * Sets how long of a delay, in milliseconds, after trying
 * to install Orbot do we assume that it's not happening.
 * Defaults to 60000ms = 60 seconds = 1 minute = 1.90259e-6 years
 *
 * @param timeoutMs delay period in milliseconds
 * @return the singleton, for chaining
 */
public OrbotInitializer installTimeout(long timeoutMs) {
  installTimeoutMs=timeoutMs;

  return(this);
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Monitoring Orbot Status

The key method of `OrbotInitializer` is `init()`, designed to be called as part of setting up `OrbotInitializer` at the outset:

```
  /**
   * Initializes the connection to Orbot, revalidating that it
   * is installed and requesting fresh status broadcasts.
   *
   * @return true if initialization is proceeding, false if
   * Orbot is not installed
   */
  public boolean init() {
    Intent orbot=OrbotHelper.getOrbotStartIntent(ctxt);
    ArrayList<String> hashes=new ArrayList<String>();


hashes.add("A4:54:B8:7A:18:47:A8:9E:D7:F5:E7:0F:BA:6B:BA:96:F3:EF:29:C2:6E:09:81:20:4F:E3:47:BF:23:1D:FD:5B");

hashes.add("A7:02:07:92:4F:61:FF:09:37:1D:54:84:14:5C:4B:EE:77:2C:55:C1:9E:EE:23:2F:57:70:E1:82:71:F7:CB:AE");

    orbot=
      SignatureUtils.validateBroadcastIntent(ctxt, orbot,
        hashes, false);

    if (orbot!=null) {
      isInstalled=true;
      handler.postDelayed(onStatusTimeout, statusTimeoutMs);
      ctxt.registerReceiver(orbotStatusReceiver,
        new IntentFilter(OrbotHelper.ACTION_STATUS));
      ctxt.sendBroadcast(orbot);
    }
    else {
      isInstalled=false;

      for (StatusCallback cb : statusCallbacks) {
        cb.onNotYetInstalled();
      }
    }

    return(isInstalled);
  }
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

NetCipher's `OrbotHelper` class has a `getOrbotStartIntent()` method that constructs an `Intent` designed to start up Orbot and get its status.

We then use `SignatureUtils`, from the CWAC-Security library, to validate that Orbot is installed and, more importantly, that it is the real Orbot, and not some hacked-and-repackaged edition that somebody installed by accident from somewhere. `validateBroadcastIntent()` takes:

- a `Context`
- the `Intent` that you would like to broadcast, typically with `setPackageName()` to narrow it down to a particular app (here, handled by `getOrbotStartIntent()`)
- the SHA-256 hash (or hashes) of the public signing key for the app
- a `boolean` indicating what you want to have happen if a signature mismatch is found, where `true` means "throw a `SecurityException`" and `false` means "just ignore it"

There are three possible responses from `validateBroadcastIntent()`:

- a `SecurityException`, if you passed `true` for the fourth parameter, which would mean an invalid copy of Orbot was installed
- `null`, which means we could not find Orbot (or, if you passed `false`, we could not find a valid copy of Orbot)
- a copy of the original `Intent`, augmented with the `ComponentName` of the actual `BroadcastReceiver` that should receive this "broadcast"

In that latter case, we know that Orbot is installed and properly signed.

For many apps, there will be only one SHA-256 hash of the signing key, in which case you just pass the `String` of that hash. However, some distribution channels for you to use different signing keys. Amazon is one, as they will sign it with their own key. F-Droid is another, as they want to build the app from source, and therefore wind up signing it with their key. Orbot is distributed through the Play Store and F-Droid, and so there are two hashes to consider.

If Orbot is installed and validated, we:

- note that Orbot is installed; clients can call `isOrbotInstalled()` to check this
- use the `Handler` to get control after the designed timeout
- register a `BroadcastReceiver` to get the response from Orbot
- send the broadcast using the refined `Intent` from `validateBroadcastIntent()`

Conversely, if Orbot is not installed, we note that fact and call `onNotYetInstalled()` on any `StatusCallback` objects that are registered at present.

`init()` returns a `boolean`, indicating if Orbot was installed. Between that and `isOrbotInstalled()`, clients can know whether or not Orbot needs to be installed. We will see how to install Orbot in the next section.

Ideally, we get our status broadcast delivered to the `orbotStatusReceiver`:

```
private BroadcastReceiver orbotStatusReceiver=new BroadcastReceiver() {
  @Override
  public void onReceive(Context ctxt, Intent intent) {
    if (TextUtils.equals(intent.getAction(),
      OrbotHelper.ACTION_STATUS)) {
      String status=intent.getStringExtra(OrbotHelper.EXTRA_STATUS);

      if (status.equals(OrbotHelper.STATUS_ON)) {
        lastStatusIntent=intent;
        handler.removeCallbacks(onStatusTimeout);

        for (StatusCallback cb : statusCallbacks) {
          cb.onEnabled(intent);
        }
      }
      else if (status.equals(OrbotHelper.STATUS_OFF)) {
        for (StatusCallback cb : statusCallbacks) {
          cb.onDisabled();
        }
      }
      else if (status.equals(OrbotHelper.STATUS_STARTING)) {
        for (StatusCallback cb : statusCallbacks) {
          cb.onStarting();
        }
      }
      else if (status.equals(OrbotHelper.STATUS_STOPPING)) {
        for (StatusCallback cb : statusCallbacks) {
          cb.onStopping();
        }
      }
    }
  }
};
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

If this is the status `Intent`, and the status is `STATUS_ON`, we:

- cache this `Intent` for later use
- cancel the timeout
- call `onEnabled()` on the `StatusCallback` objects

This receiver may be called with other statuses, indicating that Orbot is warming up and such. Those are passed along to the `StatusCallback` objects using appropriate methods (e.g., `onStarting()`).

If we do not get response by the time of the timeout, the `onStatusTimeout` `Runnable` is triggered by the `Handler` and our `postDelayed()` call:

```
private Runnable onStatusTimeout=new Runnable() {
  @Override
  public void run() {
    ctxt.unregisterReceiver(orbotStatusReceiver);

    for (StatusCallback cb : statusCallbacks) {
      cb.onStatusTimeout();
    }
  }
};
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Here, we just unregister the receiver (as we assume Orbot is not going to respond) and call `onStatusTimeout()` on all the `StatusCallback` objects.

If Orbot is ready to go, when we call `onEnabled()` on the `StatusCallback` objects, we pass along the `Intent` itself that was sent to us via the broadcast. That `Intent` contains extras that detail the proxy ports that are available. There is a `build()` method on `StrongBuilder` that takes the `Intent` and configures the HTTP connection, using that proxy information. We will examine that method, and the rest of the `StrongBuilder` family of builders, later in this chapter.

Installing Orbot

If desired, the app can call `installOrbot()` on the `OrbotInitializer` singleton, to kick off installation of Orbot:

```
public void installOrbot(Activity host) {
  handler.postDelayed(onInstallTimeout, installTimeoutMs);

  IntentFilter filter=
    new IntentFilter(Intent.ACTION_PACKAGE_ADDED);

  filter.addDataScheme("package");

  ctxt.registerReceiver(orbotInstallReceiver, filter);
  host.startActivity(OrbotHelper.getOrbotInstallIntent(ctxt));
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Here, we:

- set up a separate timeout

- register a receiver for `ACTION_PACKAGE_ADDED`, so we can watch for newly-installed apps during the period before the timeout
- start up the Play Store or F-Droid using an `OrbotHelper`-supplied `Intent`

Note that for `ACTION_PACKAGE_ADDED` and related package broadcasts, you have to have `addDataScheme("package")` in the `IntentFilter`. Otherwise, you will not receive the broadcast.

With luck, the user will install Orbot, and we will find out about that in `orbotInstallReceiver`:

```
private BroadcastReceiver orbotInstallReceiver=new BroadcastReceiver() {
  @Override
  public void onReceive(Context ctxt, Intent intent) {
    if (TextUtils.equals(intent.getAction(),
      Intent.ACTION_PACKAGE_ADDED)) {
      String pkgName=intent.getData().getEncodedSchemeSpecificPart();

      if (OrbotHelper.ORBOT_PACKAGE_NAME.equals(pkgName)) {
        isInstalled=true;
        handler.removeCallbacks(onInstallTimeout);
        ctxt.unregisterReceiver(orbotInstallReceiver);

        for (InstallCallback cb : installCallbacks) {
          cb.onInstalled();
        }

        init();
      }
    }
  }
};
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

If this is the right broadcast, and if the package that was installed is Orbot, we:

- cancel the timeout
- unregister the receiver
- call `onInstalled()` on all of the `InstallCallback` objects
- call `init()` to try to get Orbot running and get its status

But, if the timeout is reached first, `onInstallTimeout` gets triggered:

```
private Runnable onInstallTimeout=new Runnable() {
  @Override
  public void run() {
    ctxt.unregisterReceiver(orbotInstallReceiver);

    for (InstallCallback cb : installCallbacks) {
      cb.onInstallTimeout();
```

```
    }
  }
};
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/OrbotInitializer.java)

Here, we just unregister the receiver and call `onInstallTimeout()` on the `InstallCallback` objects, letting the caller know that the user elected not to install Orbot (or perhaps got distracted by a kitten).

Inside the NetCipher Builders

You may have chosen some other HTTP stack, beyond the ones shown here. Or, you may have other reasons why you want to understand the "nuts and bolts" of attaching NetCipher to an HTTP stack. The following sections will review the `Strong...Builder` family of classes, to explain what is done in each to teach that HTTP stack how to use the Orbot proxy and how to use NetCipher's root certificate store.

Each of the HTTP stack NetCipher integrations is isolated in its own library module in the sample project. Each depends on a central `libnetcipher` library module, containing the latest `master` branch of NetCipher itself. Ideally, the library modules would rely upon a NetCipher artifact. However, the most recent such artifact (`info.guardianproject.netcipher:netcipher:1.2`) was released in June 2015 and is significantly behind the `master` branch.

Each of those library modules also depends on the HTTP stack itself, with the exception of the `HttpURLConnection` code, as `HttpURLConnection` is part of the Android SDK.

StrongBuilderBase

Three out of the four `Strong...Builder` classes extend from a `StrongBuilderBase` class, supplied in `netcipher-hurl`. The exception is `StrongHttpClientBuilder`, as it extends HttpClient's own `HttpClientBuilder` and uses the delegate pattern to wrap a `StrongBuilderBase`:

```
package info.guardianproject.netcipher.hurl;

import android.content.Context;
import android.content.Intent;
import java.io.IOException;
import java.io.InputStream;
import java.net.InetSocketAddress;
import java.net.Proxy;
```

```
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;
import info.guardianproject.netcipher.proxy.OrbotHelper;

/**
 * Builds an HttpUrlConnection that connects via Tor through
 * Orbot.
 */
abstract public class
  StrongBuilderBase<T extends StrongBuilderBase, C>
  implements StrongBuilder<T, C> {
  private final static String PROXY_HOST="127.0.0.1";
  private final static String TRUSTSTORE_TYPE="BKS";
  private final static String TRUSTSTORE_PASSWORD="changeit";
  protected final Context ctxt;
  protected Proxy.Type proxyType;
  protected SSLContext sslContext=null;
  protected boolean useWeakCiphers=false;

  /**
   * Standard constructor.
   *
   * @param ctxt any Context will do; the StrongBuilderBase
   *             will hold onto the Application singleton
   */
  public StrongBuilderBase(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
  }

  /**
   * Copy constructor.
   *
   * @param original builder to clone
   */
  public StrongBuilderBase(StrongBuilderBase original) {
    this.ctxt=original.ctxt;
    this.proxyType=original.proxyType;
    this.sslContext=original.sslContext;
    this.useWeakCiphers=original.useWeakCiphers;
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public T withBestProxy() {
    if (supportsSocksProxy()) {
      return(withSocksProxy());
    }
    else {
      return(withHttpProxy());
    }
  }
```

```
/**
 * {@inheritDoc}
 */
@Override
public boolean supportsHttpProxy() {
  return(true);
}

/**
 * {@inheritDoc}
 */
@Override
public T withHttpProxy() {
  proxyType=Proxy.Type.HTTP;

  return((T)this);
}

/**
 * {@inheritDoc}
 */
@Override
public boolean supportsSocksProxy() {
  return(true);
}

/**
 * {@inheritDoc}
 */
@Override
public T withSocksProxy() {
  proxyType=Proxy.Type.SOCKS;

  return((T)this);
}

/**
 * {@inheritDoc}
 */
@Override
public T withDefaultKeystore()
  throws CertificateException, NoSuchAlgorithmException,
  KeyStoreException, IOException, UnrecoverableKeyException,
  KeyManagementException {

  /*
    NOTE: Trying to use the raw resource from netcipher
    itself proved to be extremely troublesome. This module
    has a copy of the same keystore BKS file in assets/, and
    this code pulls that keystore from that location.
   */

  InputStream in=ctxt
    .getResources()
    .getAssets()
    .open("debiancacerts.bks");

  KeyStore trustStore=KeyStore.getInstance(TRUSTSTORE_TYPE);
```

```
    trustStore.load(in, TRUSTSTORE_PASSWORD.toCharArray());

    return(withKeystore(trustStore));
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public T withKeystore(KeyStore keystore)
    throws KeyStoreException, NoSuchAlgorithmException,
    IOException, CertificateException,
    UnrecoverableKeyException, KeyManagementException {

    TrustManagerFactory tmf=TrustManagerFactory
      .getInstance(TrustManagerFactory.getDefaultAlgorithm());

    tmf.init(keystore);
    sslContext=SSLContext.getInstance("TLSv1");
    sslContext.init(null, tmf.getTrustManagers(), null);

    return((T)this);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public T withWeakCiphers() {
    useWeakCiphers=true;

    return((T)this);
  }

  public SSLContext getSSLContext() {
    return(sslContext);
  }

  public int getSocksPort(Intent status) {
    if (status.getStringExtra(OrbotHelper.EXTRA_STATUS)
      .equals(OrbotHelper.STATUS_ON)) {
      return (status.getIntExtra(OrbotHelper.EXTRA_PROXY_PORT_SOCKS,
        9050));
    }

    return(-1);
  }

  public int getHttpPort(Intent status) {
    if (status.getStringExtra(OrbotHelper.EXTRA_STATUS)
      .equals(OrbotHelper.STATUS_ON)) {
      return (status.getIntExtra(OrbotHelper.EXTRA_PROXY_PORT_HTTP,
        8118));
    }

    return(-1);
  }

  protected SSLSocketFactory buildSocketFactory() {
    SSLSocketFactory result=
```

```
      new SniFriendlySocketFactory(sslContext.getSocketFactory(),
        useWeakCiphers);

    return(result);
  }

  public Proxy buildProxy(Intent status) {
    Proxy result=null;

    if (status.getStringExtra(OrbotHelper.EXTRA_STATUS)
      .equals(OrbotHelper.STATUS_ON)) {
      if (proxyType==Proxy.Type.SOCKS) {
        result=new Proxy(Proxy.Type.SOCKS,
          new InetSocketAddress(PROXY_HOST, getSocksPort(status)));
      }
      else if (proxyType==Proxy.Type.HTTP) {
        result=new Proxy(Proxy.Type.HTTP,
          new InetSocketAddress(PROXY_HOST, getHttpPort(status)));
      }
    }

    return(result);
  }

  @Override
  public void build(final Callback<C> callback) {
    OrbotInitializer.get(ctxt).addStatusCallback(
      new OrbotInitializer.SimpleStatusCallback() {
        @Override
        public void onEnabled(Intent statusIntent) {
          OrbotInitializer.get(ctxt).removeStatusCallback(this);
          try {
            callback.onConnected(build(statusIntent));
          }
          catch (IOException e) {
            callback.onConnectionException(e);
          }
        }

        @Override
        public void onNotYetInstalled() {
          OrbotInitializer.get(ctxt).removeStatusCallback(this);
          callback.onTimeout();
        }

        @Override
        public void onStatusTimeout() {
          OrbotInitializer.get(ctxt).removeStatusCallback(this);
          callback.onTimeout();
        }
      });
  }
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/StrongBuilderBase.java)

The declaration of the class is rather unusual: `StrongBuilderBase` uses Java generics to declare that it has a template class `T`... that extends from itself. This is

because `StrongBuilderBase` implements the builder API pattern *and* we want subclasses to also be able to add builder-style methods. The problem with the builder API and a statically-typed language like Java is that if the builder methods on `StrongBuilderBase` simply return `this`, that is typed as an instance of `StrongBuilderBase`, not the concrete subclass. This "type erasure" means that you may not be able to call builder methods in the order you want. Specifically, if you tried calling a subclass' builder method after calling a `StrongBuilderBase` builder method, the compiler would complain, as it would not recognize the subclass' builder method as being available on an instance of `StrongBuilderBase`. The `T extends StrongBuilderBase` declaration, and having the `StrongBuilderBase` builder methods return `(T)this`, ensures that we do not lose the subclass' type when we call builder methods. In other words, welcome to Java.

The `withHttpProxy()` and `withSocksProxy()` methods simply note what you want, for later use in the `build()` methods. Similarly, `withWeakCiphers()` simply notes the fact that you want this setting for later use.

The `withDefaultKeystore()` method loads the NetCipher default Debian-based root certificate keystore, to use to configure our SSL connections. In `libnetcipher`, this keystore is a raw resource. However, using resources from one library to another can be a problem. In particular, the author of this book could not get Gradle and Android Studio to expose `R` values from `libnetcipher` in the other library modules. To work around this, `netcipher-hurl` and `netcipher-httpclient` have a copy of the same keystore as an asset, which does not rely on `R` values and therefore can be accessed without issue. `withDefaultKeystore()` gets an `InputStream` on that asset and uses it to initialize a `KeyStore`, using a hardcoded password.

`withDefaultKeystore()` then delegates to a `withKeystore()` method. You might use this yourself, instead of `withDefaultKeystore()`. For example, if NetCipher is not updated sufficiently frequently, you might elect to have your own keystore of known-safe root certificates; you could load those into the `Strong...Builder` classes using `withKeystore()`. `withKeystore()` sets up a `TrustManagerFactory`, initializes from the `KeyStore`, then creates an `SSLContext` and initializes it from the `TrustManager` array supplied from the `TrustManagerFactory`. Later on, we can use that `SSLContext` when configuring the HTTP stacks.

`getSocksPort()` and `getHttpPort()` look inside the status `Intent` and return the ports for those proxies, defaulting to 9050 and 8118, respectively.

`buildSocketFactory()` creates an `SSLSocketFactory` based on the `SSLContext` and the `useWeakCiphers` flag. However, rather than use a stock `SSLSocketFactory` implementation, `buildSocketFactory()` returns an `SniFriendlySocketFactory`. This wraps around an `SSLSocketFactory` and adds in smarts to allow Server Name Indication (SNI) to work on Android. `SniFriendlySocketFactory` also extends a `NetCipher`-supplied `TlsOnlySocketFactory`, which handles the cipher negotiation process, including whether or not to support weaker ciphers based on configuration.

`buildProxy()` configures a `Proxy` object based on the preferred proxy type, the port for that proxy, and whether Orbot is actually up and running. This, along with `buildSocketFactory()`, gets used by the subclasses of `StrongBuilderBase` for configuring the various HTTP stacks.

The asynchronous `build()` implementation does not depend upon a particular HTTP stack, and so it is implemented in `StrongBuilderBase`. It uses `OrbotInitializer` to get the status `Intent`. `OrbotInitializer` will call `onEnabled()` immediately if the status `Intent` was retrieved earlier and cached; otherwise, it will call `onEnabled()` once we have a status `Intent` to use. `onEnabled()`, in turn, uses the stack-specific `build()` method to provide the connection to the `Callback`.

HttpURLConnection

The `netcipher-hurl` library module only depends upon `libnetcipher`, since `HttpURLConnection` is a part of the Android SDK.

`StrongConnectionBuilder` extends the `StrongBuilderBase` class from the previous section:

```
package info.guardianproject.netcipher.hurl;

import android.content.Context;
import android.content.Intent;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.Proxy;
import java.net.URL;
import java.net.URLConnection;
import java.security.KeyManagementException;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import javax.net.ssl.HttpsURLConnection;
```

```
import javax.net.ssl.SSLSocketFactory;

/**
 * Builds an HttpUrlConnection that connects via Tor through
 * Orbot.
 */
public class StrongConnectionBuilder
  extends StrongBuilderBase<StrongConnectionBuilder, HttpURLConnection> {
  private URL url;

  /**
   * Creates a StrongConnectionBuilder using the strongest set
   * of options for security. Use this if the strongest set of
   * options is what you want; otherwise, create a
   * builder via the constructor and configure it as you see fit.
   *
   * @param ctxt any Context will do
   * @return a configured StrongConnectionBuilder
   * @throws Exception
   */
  static public StrongConnectionBuilder forMaxSecurity(Context ctxt)
    throws Exception {
    return(new StrongConnectionBuilder(ctxt)
      .withDefaultKeystore()
      .withBestProxy());
  }

  /**
   * Creates a builder instance.
   *
   * @param ctxt any Context will do; builder will hold onto
   *             Application context
   */
  public StrongConnectionBuilder(Context ctxt) {
    super(ctxt);
  }

  /**
   * Copy constructor.
   *
   * @param original builder to clone
   */
  public StrongConnectionBuilder(StrongConnectionBuilder original) {
    super(original);
    this.url=original.url;
  }

  /**
   * Sets the URL to build a connection for.
   *
   * @param url the URL
   * @return the builder
   * @throws MalformedURLException
   */
  public StrongConnectionBuilder connectTo(String url)
    throws MalformedURLException {
    connectTo(new URL(url));

    return(this);
  }
```

```
  /**
   * Sets the URL to build a connection for.
   *
   * @param url the URL
   * @return the builder
   */
  public StrongConnectionBuilder connectTo(URL url) {
    this.url=url;

    return(this);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public HttpURLConnection build(Intent status) throws IOException {
    URLConnection result;
    Proxy proxy=buildProxy(status);

    if (proxy==null) {
      result=url.openConnection();
    }
    else {
      result=url.openConnection(proxy);
    }

    if (result instanceof HttpsURLConnection && sslContext!=null) {
      SSLSocketFactory tlsOnly=buildSocketFactory();
      HttpsURLConnection https=(HttpsURLConnection)result;

      https.setSSLSocketFactory(tlsOnly);
    }

    return((HttpURLConnection)result);
  }
}
```

(from Internet/HTTPStacks/netcipher-hurl/src/main/java/info/guardianproject/netcipher/hurl/StrongConnectionBuilder.java)

There are two flavors of the `connectTo()` method, one taking a simple `String`, the other taking a `URL`. The `String` edition simply creates a `URL` and delegates to the `URL` edition, which holds onto your chosen URL.

The reason why we need the URL comes from how we use the `Proxy`. We have to supply that via a call to `openConnection()` on a `URL`, which implies that we have a `URL` to work with. That is handled in the `build()` method, which also calls `setSSLSocketFactory()` on the `HttpsURLConnection` (for SSL requests), so we can handle the cipher negotiation and enable SNI support.

OkHttp3/Retrofit

The `netcipher-okhttp` library module depends on:

- the `libnetcipher` library module, for NetCipher
- the `netcipher-hurl` library module, for some code sharing with the `HttpURLConnection` implementation
- OkHttp3 itself (`com.squareup.okhttp3:okhttp`)

`StrongOkHttpClientBuilder` extends from the `StrongBuilderBase` described above:

```
package info.guardianproject.netcipher.okhttp3;

import android.content.Context;
import android.content.Intent;
import info.guardianproject.netcipher.hurl.StrongBuilderBase;
import okhttp3.OkHttpClient;

/**
 * Creates an OkHttpClient using NetCipher configuration. Use
 * build() if you have no other OkHttpClient configuration
 * that you need to perform. Or, use applyTo() to augment an
 * existing OkHttpClient.Builder with NetCipher.
 */
public class StrongOkHttpClientBuilder extends
  StrongBuilderBase<StrongOkHttpClientBuilder, OkHttpClient> {
  /**
   * Creates a StrongOkHttpClientBuilder using the strongest set
   * of options for security. Use this if the strongest set of
   * options is what you want; otherwise, create a
   * builder via the constructor and configure it as you see fit.
   *
   * @param ctxt any Context will do
   * @return a configured StrongOkHttpClientBuilder
   * @throws Exception
   */
  static public StrongOkHttpClientBuilder forMaxSecurity(Context ctxt)
    throws Exception {
    return(new StrongOkHttpClientBuilder(ctxt)
      .withDefaultKeystore()
      .withBestProxy());
  }

  /**
   * Creates a builder instance.
   *
   * @param ctxt any Context will do; builder will hold onto
   *             Application context
   */
  public StrongOkHttpClientBuilder(Context ctxt) {
    super(ctxt);
  }
```

```
  /**
   * Copy constructor.
   *
   * @param original builder to clone
   */
  public StrongOkHttpClientBuilder(StrongOkHttpClientBuilder original) {
    super(original);
  }

  /**
   * OkHttp3 does not support SOCKS proxies:
   * https://github.com/square/okhttp/issues/2315
   *
   * @return false
   */
  @Override
  public boolean supportsSocksProxy() {
    return(false);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public OkHttpClient build(Intent status) {
    return(applyTo(new OkHttpClient.Builder(), status).build());
  }

  /**
   * Adds NetCipher configuration to an existing OkHttpClient.Builder,
   * in case you have additional configuration that you wish to
   * perform.
   *
   * @param builder a new or partially-configured OkHttpClient.Builder
   * @return the same builder
   */
  public OkHttpClient.Builder applyTo(OkHttpClient.Builder builder, Intent status) {
    return(builder
      .sslSocketFactory(buildSocketFactory())
      .proxy(buildProxy(status)));
  }
}
```

(from Internet/HTTPStacks/netcipher-okhttp3/src/main/java/info/guardianproject/netcipher/okhttp3/StrongOkHttpClientBuilder.java)

Note that since OkHttp3 does not support SOCKS proxies, `supportsSocksProxy()` is overridden to return `false`.

`StrongOkHttpClientBuilder` adds just two builder methods:

- `build()` returns an `OkHttpClient` using a stock `OkHttpClient.Builder`. Use this if you do not need to configure anything else on OkHttp3.
- `applyTo()` returns an `OkHttpClient` by adding configuration to a supplied `OkHttpClient.Builder`. Specifically, we use `sslSocketFactory()` to use the

SSLContext for our root certificates, and we use proxy() to set up the Orbot proxy.

Volley

The netcipher-volley library module depends on:

- the libnetcipher library module, for NetCipher
- the netcipher-hurl library module, for some code sharing with the HttpURLConnection implementation
- the officialy packaged version of Volley, as Google still is not shipping this themselves (com.mcxiaoke.volley:library)

The Volley code is set up a bit differently than the other two. Volley already has the notion of separating out its HTTP implementation. By default, Volley will use its HurlStack class on Android 2.3+ and its HttpStack class on older devices. Those, in turn, use HttpURLConnection and Android's built-in HttpClient implementation, respectively. However, the newRequestQueue() method on the Volley class has a version that takes a stack implementation as a parameter, so you can substitute in your own implementation.

So, we have StrongHurlStack, which extends HurlStack and does the same sort of work as we did with HttpURLConnection back in StrongConnectionBuilder:

```
package info.guardianproject.netcipher.volley;

import com.android.volley.toolbox.HurlStack;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.Proxy;
import java.net.URL;
import javax.net.ssl.SSLSocketFactory;

/**
 * Volley HurlStack subclass that adds in NetCipher protections.
 * It is simplest to create one through StrongVolleyQueueBuilder.
 */
public class StrongHurlStack extends HurlStack {
  private final Proxy proxy;

  StrongHurlStack(SSLSocketFactory sslSocketFactory, Proxy proxy) {
    super(null, sslSocketFactory);

    this.proxy=proxy;
  }

  @Override
  protected HttpURLConnection createConnection(URL url)
    throws IOException {
```

```
    HttpURLConnection result;

    if (proxy==null) {
      result=(HttpURLConnection)url.openConnection();
    }
    else {
      result=(HttpURLConnection)url.openConnection(proxy);
    }

    // following from original HurlStack
    // Workaround for the M release HttpURLConnection not observing the
    // HttpURLConnection.setFollowRedirects() property.
    // https://code.google.com/p/android/issues/detail?id=194495
    result.setInstanceFollowRedirects(HttpURLConnection.getFollowRedirects());

    return(result);
  }
}
```

(from Internet/HTTPStacks/netcipher-volley/src/main/java/info/guardianproject/netcipher/volley/StrongHurlStack.java)

`StrongVolleyQueueBuilder` then uses `newRequestQueue()` to opt into using `StrongHurlStack`:

```
package info.guardianproject.netcipher.volley;

import android.content.Context;
import android.content.Intent;
import com.android.volley.RequestQueue;
import com.android.volley.toolbox.Volley;
import info.guardianproject.netcipher.hurl.StrongBuilderBase;

/**
 * Builds an HttpUrlConnection that connects via Tor through
 * Orbot.
 */
public class StrongVolleyQueueBuilder extends
  StrongBuilderBase<StrongVolleyQueueBuilder, RequestQueue> {
  /**
   * Creates a StrongVolleyQueueBuilder using the strongest set
   * of options for security. Use this if the strongest set of
   * options is what you want; otherwise, create a
   * builder via the constructor and configure it as you see fit.
   *
   * @param ctxt any Context will do
   * @return a configured StrongVolleyQueueBuilder
   * @throws Exception
   */
  static public StrongVolleyQueueBuilder forMaxSecurity(Context ctxt)
    throws Exception {
    return(new StrongVolleyQueueBuilder(ctxt)
      .withDefaultKeystore()
      .withBestProxy());
  }

  /**
   * Creates a builder instance.
   *
```

```
   * @param ctxt any Context will do; builder will hold onto
   *             Application context
   */
  public StrongVolleyQueueBuilder(Context ctxt) {
    super(ctxt);
  }

  /**
   * Copy constructor.
   *
   * @param original builder to clone
   */
  public StrongVolleyQueueBuilder(StrongVolleyQueueBuilder original) {
    super(original);
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public RequestQueue build(Intent status) {
    return(Volley.newRequestQueue(ctxt,
      new StrongHurlStack(buildSocketFactory(), buildProxy(status))));
  }
}
```

(from Internet/HTTPStacks/netcipher-volley/src/main/java/info/guardianproject/netcipher/volley/StrongVolleyQueueBuilder.java)

HttpClient

`StrongHttpClientBuilder` is more complicated:

- Partly because the HttpClient API is extremely verbose
- Partly because `StrongHttpClientBuilder` extends `HttpClientBuilder`, adding in `StrongBuilder` support via delegation
- Partly because it is based on existing NetCipher code of indeterminate utility

```
package info.guardianproject.netcipher.httpclient;

import android.content.Context;
import android.content.Intent;
import java.io.IOException;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;
import cz.msebera.android.httpclient.HttpHost;
import cz.msebera.android.httpclient.client.HttpClient;
import cz.msebera.android.httpclient.config.Registry;
import cz.msebera.android.httpclient.config.RegistryBuilder;
import cz.msebera.android.httpclient.conn.HttpClientConnectionManager;
import cz.msebera.android.httpclient.conn.socket.ConnectionSocketFactory;
```

```
import cz.msebera.android.httpclient.conn.socket.PlainConnectionSocketFactory;
import cz.msebera.android.httpclient.impl.client.CloseableHttpClient;
import cz.msebera.android.httpclient.impl.client.HttpClientBuilder;
import cz.msebera.android.httpclient.impl.conn.PoolingHttpClientConnectionManager;
import info.guardianproject.netcipher.hurl.OrbotInitializer;
import info.guardianproject.netcipher.hurl.StrongBuilder;
import info.guardianproject.netcipher.hurl.StrongBuilderBase;

/**
 * Subclass of HttpClientBuilder that adds configuration
 * options and defaults for NetCipher, improving the security
 * of socket connections.
 */
public class StrongHttpClientBuilder extends HttpClientBuilder implements
  StrongBuilder<StrongHttpClientBuilder, HttpClient> {
  final static String PROXY_HOST="127.0.0.1";
  private Simple netCipher;
  private final Context ctxt;

  /**
   * Creates a StrongHttpClientBuilder using the strongest set
   * of options for security. Use this if the strongest set of
   * options is what you want; otherwise, create a
   * builder via the constructor and configure it as you see fit.
   *
   * @param ctxt any Context will do
   * @return a configured StrongHttpClientBuilder
   * @throws Exception
   */
  static public StrongHttpClientBuilder forMaxSecurity(Context ctxt)
    throws Exception {
    return(new StrongHttpClientBuilder(ctxt)
      .withDefaultKeystore());
  }

  /**
   * Standard constructor
   *
   * @param ctxt any Context will do; we hold onto the Application
   *             singleton
   */
  public StrongHttpClientBuilder(Context ctxt) {
    this.ctxt=ctxt.getApplicationContext();
    netCipher=new Simple(ctxt);
  }

  /**
   * Copy constructor.
   *
   * @param original builder to clone
   */
  public StrongHttpClientBuilder(StrongHttpClientBuilder original) {
    this.netCipher=new Simple(original.netCipher);
    this.ctxt=original.ctxt;
  }

  @Override
  public CloseableHttpClient build() {
    throw new IllegalStateException(
      "Use a one-parameter build() method please");
```

```
}

/**
 * {@inheritDoc}
 */
@Override
public HttpClient build(Intent status) throws IOException {
  init(status);

  return(super.build());
}

@Override
public void build(final Callback<HttpClient> callback) {
  OrbotInitializer.get(ctxt).addStatusCallback(
    new OrbotInitializer.SimpleStatusCallback() {
      @Override
      public void onEnabled(Intent statusIntent) {
        OrbotInitializer.get(ctxt).removeStatusCallback(this);
        try {
          callback.onConnected(build(statusIntent));
        }
        catch (IOException e) {
          callback.onConnectionException(e);
        }
      }

      @Override
      public void onStatusTimeout() {
        OrbotInitializer.get(ctxt).removeStatusCallback(this);
        callback.onTimeout();
      }
    });
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withBestProxy() {
  netCipher.withBestProxy();

  return(this);
}

/**
 * {@inheritDoc}
 */
@Override
public boolean supportsHttpProxy() {
  return(true);
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withHttpProxy() {
  netCipher.withHttpProxy();
```

```
  return(this);
}

/**
 * {@inheritDoc}
 */
@Override
public boolean supportsSocksProxy() {
  return(true);
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withSocksProxy() {
  netCipher.withSocksProxy();

  return(this);
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withDefaultKeystore()
  throws CertificateException, NoSuchAlgorithmException,
  KeyStoreException, IOException, UnrecoverableKeyException,
  KeyManagementException {
  netCipher.withDefaultKeystore();

  return(this);
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withKeystore(KeyStore keystore)
  throws KeyStoreException, NoSuchAlgorithmException,
  IOException, CertificateException,
  UnrecoverableKeyException, KeyManagementException {
  netCipher.withKeystore(keystore);

  return(this);
}

/**
 * {@inheritDoc}
 */
@Override
public StrongHttpClientBuilder withWeakCiphers() {
  netCipher.withWeakCiphers();

  return(this);
}

protected void init(Intent status) {
  StrongSSLSocketFactory sFactory;
  int socksPort=netCipher.getSocksPort(status);
```

```
    if (socksPort==-1) {
      int httpPort=netCipher.getHttpPort(status);

      if (httpPort!=-1) {
        setProxy(new HttpHost(PROXY_HOST, httpPort));
      }

      sFactory=
        new StrongSSLSocketFactory(netCipher.getSSLContext());
    }
    else {
      sFactory=
        new StrongSSLSocketFactory(netCipher.getSSLContext(),
          socksPort);
    }

    setSSLSocketFactory(sFactory);

    Registry<ConnectionSocketFactory> registry=
      RegistryBuilder.<ConnectionSocketFactory>create()
        .register("http", PlainConnectionSocketFactory.getSocketFactory())
        .register("https", sFactory)
        .build();

    HttpClientConnectionManager ccm=
      new PoolingHttpClientConnectionManager(registry);

    setConnectionManager(ccm);
  }

  private static class Simple extends StrongBuilderBase<Simple, HttpClient> {
    public Simple(Context ctxt) {
      super(ctxt);
    }

    public Simple(
      StrongBuilderBase original) {
      super(original);
    }

    @Override
    public HttpClient build(Intent status) throws IOException {
      throw new IllegalStateException("Um, don't use this, m'kay?");
    }
  }
}
```

(from Internet/HTTPStacks/netcipher-httpclient/src/main/java/info/guardianproject/netcipher/httpclient/StrongHttpClientBuilder.java)