Vol.2013-DPS-154 No.4
Vol.2013-CSEC-60 No.4
2013/3/14

# AndroidにおけるWebViewの脆弱性を利用した攻撃を防止するアクセス制御方式の提案

于 競[1,a)　山内 利宏[1]

**概要**：Android は，WebView を利用することで，Web ページと連携できる．これにより，Web ページ内の JavaScript は，Android 端末のメソッドを実行できる．しかし，実行されたメソッドにより，JavaScript から Android 端末内の個人情報の奪取，データの改ざんなどの不正操作ができる問題がある．そこで，JavaScript から Android 端末の資源を操作する Java オブジェクトにアクセス制御機構を追加することで，Android 端末への攻撃を防止する方式を提案する．提案方式は，Java オブジェクトの登録時に，Java オブジェクトが実行できる処理を解析し，危険な API リストと比較する．これにより，Java オブジェクトの利用により生じる潜在する脅威を検出し，利用者に提示する．利用者は，提示された情報をもとに，Java オブジェクトを無効化することで，Android 端末への攻撃を防止できる．

## 1. はじめに

近年，Google 社が提唱・開発した携帯端末用プラットフォームである Android[1] が注目されている．Android は，オープンソースで広く開発が行われており，Android を搭載した端末が急激に普及している．

Android は，Android アプリケーション（以降，Android アプリと呼ぶ）の開発をサポートするために，豊富なライブラリを提供している．特に，WebKit が提供している WebView[2] を利用することで，Android アプリに簡易的なブラウザ機能を実現できる．利用者は，デフォルトのブラウザを使わずに，Android アプリ内で Web ページをレンダリングできる．また，Android アプリと JavaScript の連携により，Web ページ内の JavaScript は，Android 端末の機能や情報へのアクセスができる．このように，Android アプリの開発者は，機能の豊富なアプリを簡単に開発できる．

一方，WebView の使い方をうまく把握できないと，脆弱性が生まれる [3]．この脆弱性が悪用された場合，個人情報の奪取，データの改ざんなどのセキュリティ上の問題が発生する．WebView の脆弱性を利用した攻撃について，文献 [4] と文献 [5] で報告されている．文献 [4] では，WebView の脆弱性を利用した攻撃は，Web ページから Android 端末への攻撃と Android 端末から Web ページへの攻撃の 2 種類あると述べている．文献 [5] は，WebView により，Android パーミッション (以降，パーミッションと呼ぶ) を悪用する JavaScript の脅威を紹介している．また，その脅威を推定する静的解析手法を提案している．しかし，WebView の脆弱性を利用した攻撃への対策は述べられていない．

本稿では，WebView の脆弱性を利用した攻撃を防止するアクセス制御方式を提案する．提案方式は，Android アプリが JavaScript と連携するときに利用する Java オブジェクトにアクセス制御機構を追加する．また，addJavascriptInterface API による Java オブジェクトの登録処理を行う際に，Android アプリを逆アセンブルすることで，Java オブジェクトが実行できる処理を解析し，危険な API リストと比較する．これにより，Java オブジェクトの利用により生じる潜在する脅威を検出できる．さらに，利用者に Java オブジェクトの登録を許可するか否かを尋ねる．利用者は，提示された情報をもとに，Java オブジェクトを無効化することで，Android 端末への攻撃を防止できる．

## 2. Androidの基本機構とWebViewの問題点

### 2.1 基本機構

#### 2.1.1 Dalvik 仮想マシン

Android は，Dalvik 仮想マシンを利用している．Dalvik 仮想マシンは，端末ごとのスペックの違いを吸収することに加えて，サンドボックスの役割を持つ．Android アプリは，個別の UID を持ち，Dalvik 仮想マシン上で動作している．これにより，安全性を高める．

Dalvik 仮想マシンが実行可能なファイルのフォーマッ

[1] 岡山大学 大学院自然科学研究科
[a)] yu@swlab.cs.okayama-u.ac.jp

トは，dex である．dex ファイルには，コンパイルした Android アプリのソースコードが含まれている．

#### 2.1.2 パーミッション機構

Android は，Android アプリがアクセスできる機能や情報をパーミッション機構で制御している．Android アプリが利用できるパーミッションは，Android アプリのインストール時に利用者に提示され，利用者が承認した上でインストール処理を行う．なお，要求されたパーミッションの一部のみを許可して，その他のパーミッションを否定することはできない．このため，利用者は，要求されたすべてのパーミッションを許可する必要がある．また，インストールされた Android アプリのパーミッションを変更することはできない．

### 2.2 WebView

WebView は，ブラウザエンジンである WebKit が提供するクラスである．WebView は，Web ページを Android アプリにロードし，レンダリングすることができる．また，WebView には様々な API が提供されている．これらの API を利用することで，Android アプリと Web ページの連携ができ，Android アプリに簡易的なブラウザ機能を実現できる．これにより，Web ページを読み込む際に，デフォルトブラウザへ切り替えることなく，Android アプリ内で Web ページをレンダリングできる．WebView の利用により，Android アプリの開発者は，機能が豊富で，利便性の高いアプリを開発できる．

Android アプリと Web ページの連携に，よく利用する 3 つの API を以下に説明する．

(1) setJavaScriptEnabled API
JavaScript を有効化するための API である．JavaScript を有効化しないと，Web ページを正常に表示できない可能性がある．このため，WebView では，setJavaScriptEnabled API により JavaScript を有効化する．

(2) addJavascriptInterface API
JavaScript から Android 端末に定義されているメソッドを実行できるようにする仕組みである．addJavascriptInterface API を利用することで，Java オブジェクトを WebView に登録する．登録された Java オブジェクトの利用により，JavaScript は Java クラス内に定義されているメソッドを実行できる．

(3) loadUrl API
Web ページをロードするために利用する API である．以下に，Google のホームページをロードする例を示す．

```
WebView webview = new WebView(this);
webview.loadUrl("http://www.google.com/");
```

また，loadUrl API により，Web ページに JavaScript コードを送り込むことができる．以下に，Google のホームページに「Hello WebView」のアラートを出力する例を示す．

```
webview.loadUrl("http://www.google.com/
        javascript:alert("Hello WebView")");
```

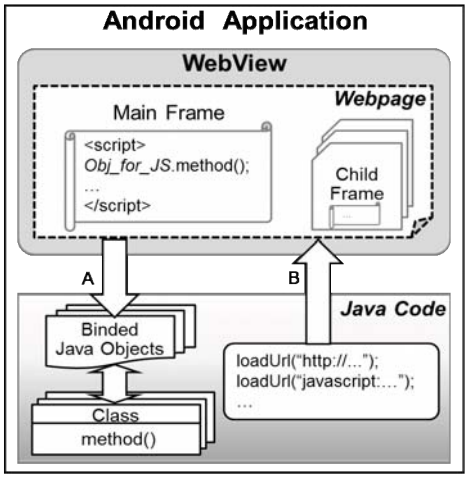


**図 1** WebView を利用した Android アプリの構成

### 2.3 WebView の問題点

2.2 節で述べたように，WebView を利用することで，Android アプリは Web ページと連携できる．これにより，Web ページ内の JavaScript は，Android アプリが要求したパーミッションの範囲で，Android 端末を操作することができる．この機能は，悪意のある JavaScript に利用された場合，Android 端末に脅威となる．

### 2.4 WebView を利用した攻撃

#### 2.4.1 攻撃モデル

**図 1** に WebView を利用した Android アプリの構成を示す．WebView を利用した攻撃には，以下の 2 種類がある [4]．

(1) Web ページから Android 端末への攻撃
JavaScript から Android 端末内の個人情報の奪取，データの改ざんなどの不正操作を行う攻撃である．攻撃経路を図 1 の A に示す．addJavascriptInterface API により，Java オブジェクトを WebView に登録すると，WebView にロードされるすべての Web ページは，オリジン (ドメイン名，プロトコル，ポートを指す) を問わず，Java オブジェクトが実行できるメソッドを呼び出すことができる．

(2) Android 端末から Web ページへの攻撃
loadUrl API を利用して，Android アプリから WebView にロードされた Web ページに悪意のある JavaScript を送り込むことで，JavaScript インジェクションを行う攻撃である．攻撃経路を図 1 の B に示す．

本研究は，(1) の Web ページから Android 端末への攻撃

```
1:  WebView webview = new WebView(this);
2:  webview.getSettings().setJavaScriptEnabled(true);
3:  webview.addJavascriptInterface(new JavaObject(), “Obj_for_JS”);
4:  webview.loadUrl(“http://target.com”);
…
5: public class JavaObject () {
6:      public void method_1() {
7:          //実行したい処理
8:      }
9:      public boolean method_2(String data) {
10:         //実行したい処理
11:     }
 …
12:     public string method_n() {
13:         //getLine1Number() APIにより電話番号を取得
14:         phoneNum = telephonyManager.getLine1Number();
            …
15:     }
16: }
```

**図 2** Android 端末側の実装コードの例

```
1: <script>
2:    var secData;
3:    secData = Obj_for_JS.method_n();
      …
4: </script>
```

**図 3** Web ページ側の実装コードの例

への対策を提案する．

#### 2.4.2 攻撃例

WebView を利用した様々な攻撃例が公開されている [6]．本稿では，JavaScript から Android 端末の電話番号の情報を奪取する例を用いて，Web ページから Android 端末への攻撃を説明する．Android アプリは，Web ページと連携するための INTERNET パーミッションと，電話番号を取得するための READ_PHONE_STATE パーミッションを保持するものとする．

Android アプリと Web ページの連携を実現するには，Android 端末側と Web ページ側の両方の実装が必要である．**図 2** に Android 端末側の実装コードの例を，**図 3** に Web ページ側の実装コードの例を示す．

図 2 の 1 行目から 4 行目までは，JavaScript から Android 端末のメソッドを呼び出すための必要な要素である．各要素の説明を以下に示す．

(1) `WebView` のインスタンスの生成
(2) `JavaScript` の有効化
(3) `JavaScript` が利用できる `Java` オブジェクトの登録
(4) ロードする `Web` ページの `URL` の指定

上記の 4 つの必要な要素が全部揃わないと，JavaScript から Android 端末のメソッドを呼び出すことができず，Android 端末に脅威とならない．このため，これらの 4 つの要素は脅威となるのに必要な要素といえる．

図 2 の 3 行目では，addJavascriptInterface API により，JavaObject クラスと対応している Obj_for_JS という Java オブジェクトが登録される．これにより，JavaScript は Obj_for_JS を利用することで，JavaObject クラスに定義されている処理を実行できる．図 3 の 3 行目では，JavaScript は，Obj_for_JS を利用して Android 端末の method_n() を呼び出す．method_n() は，さらに TelephonyManger クラスの getLine1Number() を呼び出している．これにより，Web ページ内の JavaScript から Android 端末の電話番号の情報を取得できる．

#### 2.4.3 考察

JavaScript から Android 端末のメソッドの呼び出しは，addJavascriptInterface API により登録された Java オブジェクトを利用している．また，Java オブジェクトは，Java クラスのインスタンスであり，JavaScript から呼び出し可能なメソッドは Java クラスに定義されている．このメソッドに，個人情報へのアクセス，データの変更のような危険性のある API(以降，危険な API と呼ぶ) が含まれる場合，JavaScript から個人情報の奪取，データの改ざんなどの不正操作が可能である．上記により，危険な API は脅威の原因となることがわかった．このため，脅威を防止するには，危険な API を特定して制御する必要がある．

## 3. 提案方式

### 3.1 目的

提案方式は，悪意のある JavaScript から危険な API へのアクセスを防止することを目的とする．利用者は，Android アプリをインストールする際に，要求された Android 端末の機能や資源へのアクセスのパーミッションを許可する．しかし，利用者は，Android アプリと Web ページの連携により，Web ページ内の JavaScript が許可されたパーミッションを利用できることを意識していない．Android アプリが悪意のある Web ページを読み込んだ場合，悪意のある JavaScript が Java オブジェクトを利用し，危険な API を実行することが可能になる．このため，JavaScript から危険な API へのアクセスを制御する必要がある．

### 3.2 考え方

2.4.2 項で述べたように，Java オブジェクトは，脅威となる 4 つの必要な要素の 1 つである．Java オブジェクトを制御することで，JavaScript からの攻撃を防止できる．また，2.4.3 項の考察より，JavaScript からの脅威の原因は，危険な API である．このため，提案方式の目的を実現するために，危険な API を Java オブジェクト単位で特定して制御する．

Java オブジェクトが実行できるメソッドに危険な API が含まれる場合，この Java オブジェクトを制御対象とする (以降，制御対象となる Java オブジェクトと呼ぶ)．提案方式の目的を実現するための要件を以下に示す．

**(要件 1)** 制御対象となる Java オブジェクトを判別できること

**(要件 2)** 制御対象となる Java オブジェクトへのアクセスを制御できること

addJavascriptInterface API により登録された Java オブジェクトは，制御対象となる Java オブジェクトであるか否か判別する必要がある．また，制御対象となる Java オブジェクトと判別された場合，悪意のある JavaScript からのアクセスを防止する必要がある．

### 3.3 課題

提案方式の課題を以下に示す．

**(課題 1)** Java オブジェクトが実行できる処理を把握すること

**(課題 2)** 潜在する脅威を Java オブジェクト単位で特定すること

**(課題 3)** 潜在する脅威が特定された場合に，利用者に警告を提示し，判断結果により制御すること

(要件 1) を満たすために，(課題 1) と (課題 2) がある．(要件 2) を満たすために (課題 3) がある．

### 3.4 対処

#### 3.4.1 (課題 1) への対処

Java オブジェクトが実行できる処理を把握するために，対応する Java クラスで定義されている処理のソースコードを参照する必要がある．しかし，Android アプリはコンパイルされており，パッケージの形式になっている．このため，Android アプリのソースコードを直接参照できない．そこで，Android アプリを静的解析することで，各 Java クラスが実行できる処理を把握する．

Android アプリを解析するツールとして，Dedexer[7] や dex2jar[8] などがある．Dedexer は，Android アプリの実行ファイルである dex ファイルを逆アセンブルすることで，dex ファーマットをアセンブリコードに変更できる．dex2jar は，dex ファイルを class ファイルを含む jar ファイルに変更できる．さらに，JD-GUI[9] を利用することで，jar ファイルを Java コードの形式で表示できる．しかし，上記のツールは Windows 上で利用するため，Android 上では直接利用できない．

提案方式では，Android アプリに潜在する脅威を動的に検出して制御するため，Android 上で静的解析を行う必要がある．そこで，提案方式は，Android が提供する逆アセンブルツールである dexdump を利用する．dexdump により，生成されたアセンブリコードを解析することで，各 Java クラスの処理を把握でき，(課題 1) に対処する．

#### 3.4.2 (課題 2) への対処

dexdump により生成されたアセンブリコードを解析することで，API の呼び出し元はどの Java クラスにあるかを把握できる．これにより，危険な API を Java オブジェクト単位で特定できる．なお，危険な API は，個人情報を扱う API や外部と通信する API など**表 1** で定義したものとする．

**表 1** 危険な API リスト

| API 名 | 操作する情報 |
|---|---|
| getCellLocation | 端末の現在地 |
| getDeviceId | 端末 ID 番号 |
| getNetworkOperator | MCC, MNC 番号 |
| getPhoneType | 端末の電話形式 |
| getSubscriberId | 加入者 ID |
| getLine1Number | 電話番号 |
| getSimSerialNumber | SIM のシリアル番号 |
| getVoiceMailAlphaTag | ボイスメール番号の識別子 |
| getVoiceMailNumber | ボイスメール番号 |
| sendDataMessage | SMS でのデータ送信 |
| sendMultipartTextMessage | マルチパートテキストでの送信 |
| sendTextMessage | テキストデータの送信 |
| getAllProviders | ロケーションプロバイダ名 |
| getBestProvider | 最も沿うプロバイダ名 |
| getGpsStatus | GPS の状態 |
| getLastKnownLocation | プロバイダから得た最終位置情報 |
| clearPassword | パスワードの削除 |
| editProperties | 認証設定の変更 |
| getAccounts | 端末のアカウントリスト |
| getAuthToken | 認証トークン |
| getPassword | アカウントに関するパスワード |
| getUserData | アカウントに関するユーザデータ |
| peekAuthToken | キャッシュ内の認証トークン |
| removeAccount | アカウントの削除 |
| setPassword | パスワードの設定，削除 |
| getName | Bluetooth アダプタ名 |
| getProfileConnectionState | プロファイルの接続状況 |
| getProfileProxy | プロファイルプロキシ |
| getParams | クライアントの状態 |
| getUngzippedContent | 応答実体の InputStream |
| getCertificate | オブジェクトに関する SSL 認証 |
| clearHistory | ブックマーク，履歴の削除 |
| clearSearches | 検索データベースの削除 |
| getAllBookmarks | ブックマークのリスト |
| getAllVisitedUrls | ブラウジングした URL のリスト |

#### 3.4.3 (課題 3) への対処

addJavascriptInterface API により Java オブジェクトの登録処理が行われた場合，Web ページの JavaScript から Android 端末に定義されたメソッドを実行できるようになる．そこで，提案方式は，addJavascriptInterface API をフックすることで，Java オブジェクトの登録処理が行われる前に，静的解析により脅威が潜在するかチェックする．Java オブジェクトが制御対象となる Java オブジェクトと判断された場合，利用者に警告を提示する．利用者は，提示された情報をもとに，制御対象となる Java オブジェクトを無効化することで，JavaScript から Android 端末への攻撃を防止できる．

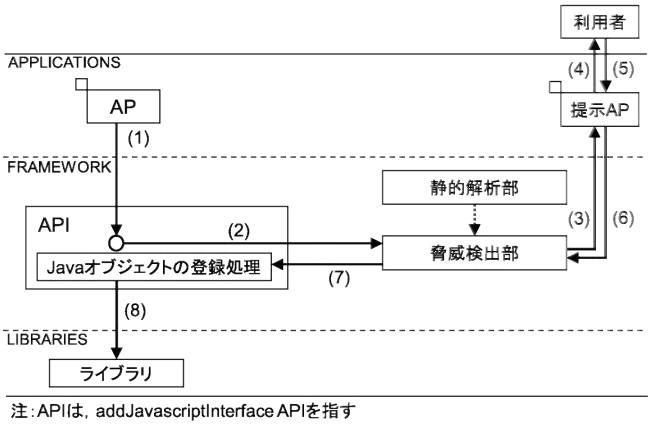


**図 4** 提案方式の全体像

### 3.5 基本構成

#### 3.5.1 全体像

**図 4** に提案方式の全体像を示す．提案方式は，フレームワーク層で Java オブジェクトの制御を行う．主に，静的解析部，脅威検出部，および提示 AP の 3 つからなる．提案方式は，addJavascriptInterface API をフックすることで，登録されようとしている Java オブジェクトの利用により生じる潜在する脅威があるかチェックし，利用者に Java オブジェクトの登録の可否を尋ねる．利用者の判断結果により，制御対象となる Java オブジェクトを無効化するか対処を行う．潜在する脅威があるか否かのチェックは，危険な API と Java クラスの対応関係を示した API_Class マッチングリストをもとに，脅威検出部が行う．API_Class マッチングリストは，静的解析部が生成する．図 4 の静的解析部から脅威検出部への点線の矢印は，脅威検出部からの API_Class マッチングリストの参照を表す．

提案方式の処理の流れを以下に示す．

(1) `Android` アプリが `addJavascriptInterface API` を呼び出す．
(2) `addJavascriptInterface API` をフックし，`Java` オブジェクトの情報を脅威検出部に送信する．
(3) 脅威検出部により，脅威があると判断された場合，提示 `AP` を呼び出す．なお，脅威がないと判断された場合，(7) の処理を行う．
(4) 利用者に警告を提示する．
(5) 利用者は，提示された情報を参考に，可否判断を行う．
(6) 提示 `AP` は，判断結果を脅威検出部に返却する．
(7) 脅威検出部は，判断結果を `addJavascriptInterface API` に返却する．
(8) 脅威がない，または利用者が `Java` オブジェクトを有効化した場合，`Java` オブジェクトの登録処理を行う．なお，利用者により `Java` オブジェクトを無効化した場合，`Java` オブジェクトの登録処理を行わず，`addJavascriptInterface API` の処理を終了させる．

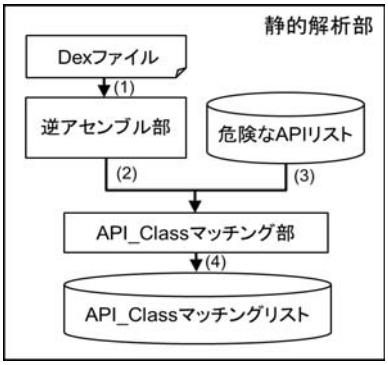


**図 5** 静的解析部

#### 3.5.2 静的解析部

**図 5** に静的解析部の構成を示す．静的解析部は，逆アセンブル部と API_Class マッチング部からなる．静的解析部の処理の流れを以下に示す．

(1) 逆アセンブル部は，Android アプリの apk ファイルから実行ファイルである dex ファイルを取得する．
(2) dexdump を利用して逆アセンブルする．提案方式は，逆アセンブル処理のオーバヘッドを抑えるために，静的解析に必要のない情報をダンプしないように dexdump のソースコードの修正を行った．
(3) API_Class マッチング部は，逆アセンブル部により生成されたアセンブリコードを解析し，各 Java クラスに含まれている API をあらかじめ作成した危険な API リストと比較する．
(4) 危険な API リストに一致する API 情報は，各 Java クラス毎に API_Class マッチングリストに格納する．これにより，危険な API と Java クラスの対応関係を把握する．

提案方式は，潜在する脅威をチェックするために，Android アプリを一回静的解析する．ただし，Android アプリが更新された場合，改めて静的解析する必要がある．静的解析部がどのようなタイミングで実行されるかは，システムの性能に影響を与える．静的解析部の実行タイミングについて，Android アプリのインストール時と addJavascriptInterface API 実行時の 2 つの方法が考えられる．

静的解析部の実行タイミングをインストール時にした場合，Android アプリをインストールするとともに，静的解析を行う．しかし，インストール時では，Android アプリは WebView を利用しているか否か判断できないため，すべての Android アプリに対して静的解析を行う．このため，WebView を利用していない Android アプリの解析が無駄になる．

静的解析部の実行タイミングを addJavascriptInterface API 実行時にした場合，検知対象となる Android アプリのみ静的解析を行う．しかし，毎回の addJavascriptInterface API 実行時に静的解析を行うと，オーバヘッドが大きい．


**図 6** 提示 AP の表示例

このため，静的解析が行われていない Android アプリであるか否か，また Android アプリの更新により再解析する必要があるか否か判断する仕組みが必要である．

現段階の提案方式の実装は，インストール時に静的解析部を実行している．今後の課題として，この 2 つの方法の性能を評価することがある．

#### 3.5.3 脅威検出部

脅威検出部は，addJavascriptInterface API のフックにより得られた Java オブジェクトの情報，静的解析部により得られた API_Class マッチングリストをもとに，Java オブジェクトの利用により生じる潜在する脅威があるか否かチェックする．脅威があると判断された場合，提示 AP を呼び出す．また，利用者の判断結果を受け取って，addJavascriptInterface API に返却する．脅威がないと判断された場合，提示 AP を呼び出さず，その結果を addJavascriptInterface API に返却する．addJavascriptInterface API は，通常の Java オブジェクトの登録処理を行う．

#### 3.5.4 提示 AP

提示 AP は，利用者に警告を提示する．また，利用者の判断結果を脅威検出部に返却する．詳細は，**図 6** に示す警告の表示例を用いて説明する．表示画面には，Android アプリ名，WebView がロードする Web ページの URL 情報，Web ページから Android 端末の資源を操作する Java オブジェクトの名前，および検出された危険な API の情報を表示する．利用者は，表示された情報を参考に，JavaScript に提供する Java オブジェクトを有効化するか，または無効化するかを判断する．

**表 2** ホスト OS の測定環境

| | |
|---|---|
| ディストリビューション | Windows 7 Home Premium |
| CPU | Inter(R) Core(TM) i7-3517U 1.90GHz |
| メモリ | 8 GB |
| 仮想化ソフトウェア | VMware Player 4.0.4 |

**表 3** ゲスト OS の測定環境

| | |
|---|---|
| ディストリビューション | Ubuntu 10.04 LTS |
| カーネル | Linux 2.6.32-44-generic |
| 仮想 CPU 数 | 2 |
| メモリ | 4 GB |

**表 4** 静的解析処理時間の測定結果

| Android アプリ | dex ファイルのサイズ | 平均処理時間 |
|---|---|---|
| HelloWebView | 5.6 KB | 182 ms |
| LivingSocial | 781.8 KB | 12,324 ms |

### 3.6 期待される効果

提案方式の実現により以下の効果が期待できる．

(1) WebView を利用したすべての Android アプリにおいて脅威を漏れなく検出できる
addJavascriptInterface API をフックし，API_Class マッチングリストを参照することで，すべての Java オブジェクトの利用により生じる潜在する脅威を検出できる．

(2) 利用者の判断を支援できる
提示する警告に Web ページの URL を表示することで，利用者は，Web からの脅威への意識を高めることができる．また，利用者が可否判断を行う際に，Web ページの URL の情報と危険な API の情報を参考できる．また，利用者は，うまく判断できない場合，Web ページの URL を WebView の代わりに，デフォルトブラウザにロードさせ，デフォルトブラウザで Web ページの安全性を確認した上で可否判断を行うこともできる．

(3) 脅威が検出された場合，Java オブジェクトを無効化して，Web ページを参照できる
提案方式は，脅威を Java オブジェクト単位で特定し，制御できる．このため，Android アプリの動作を止めることなく，脅威となる Java オブジェクトのみを無効化することで，脅威を防止できる．これにより，WebView で Web ページを安全に参照できる．

## 4. 性能評価

提案方式の性能を把握するために，静的解析の処理時間を測定した．測定用の Android アプリは，自作した HelloWebView と文献 [4] で紹介されている WebView を利用した Android アプリの代表例である LivingSocial[11] を用いた．測定環境を**表 2** と**表 3** に示す．ホスト OS は，VMware Player 4.0.4 を利用してゲスト OS を起動させた後，ゲスト OS 上で提案方式を実装した Android 4.0.3 を動作させ，

各 Android アプリを 5 回静的解析した．測定結果を**表 4** に示す．

表 4 から，サイズの小さい HelloWebView の静的解析にかかる時間は 182 ミリ秒であり，短いことがわかった．一方，サイズの大きい LivingSocial は，約 12 秒程度の処理時間がかかった．ただし，提案方式では，Android アプリの静的解析は，アプリ当たり一回しか行わないため，利用者にはそれほどのオーバヘッドではないと考えられる．

## 5. 関連研究

WebView の利用により生じる脆弱性が注目されている [3],[6],[10]．文献 [4] の調査では，86%の Android アプリが WebView を利用していると報告されている．また，WebView を利用した攻撃について，Web ページから Android 端末への攻撃と Android 端末から Web ページへの攻撃の 2 種類があると述べられている．

文献 [5] は，WebView を利用した脅威を推定する静的解析手法を提案している．この静的解析手法は，Dedexer を利用し，あらかじめ作成した悪性 API リストをもとに脅威を検出する．しかし，文献 [5] は，脅威が潜在するか否かの推定を Android アプリ単位で行っている．また，推定された脅威への実行時の対処を行っていない．このため，Android アプリに脅威があると推定された場合，利用者は，その Android アプリを利用するか否かしか選択できない．

提案方式は，動作している Android アプリに WebView の利用により生じる潜在する脅威があるか否かを Java オブジェクト単位で特定し，利用者に通知する．また，利用者は，Java オブジェクトを無効化することで，悪意のある JavaScript を WebView にロードしても，Android 端末に脅威とならない．これにより，利用者は，WebView で Web ページを安全に参照できる．また，提案方式は，Java オブジェクトの登録時に脅威が潜在するかチェックするため，WebView を利用したすべての Android アプリにおいて脅威を漏れなく検出して対処できる．

## 6. おわりに

WebView の利用による Web ページから Android 端末への攻撃について述べた．この攻撃への対処として，JavaScript から Android 端末の資源を操作する Java オブジェクトにアクセス制御機構を追加することで，Android 端末への攻撃を防止する方式を提案した．提案方式では，Java オブジェクトの登録時に，Java オブジェクトの利用により生じる潜在する脅威を検出できること，検出された脅威を利用者に通知できること，利用者は Java オブジェクトを無効化することにより，Android 端末への攻撃を防止できることを示した．

今後の課題として，静的解析部の実行タイミングの検討と提案方式のオーバヘッドの評価がある．

**謝辞** 本研究の一部は，栢森情報科学振興財団平成 23 年度研究助成による．

### 参考文献

[1] Android, the world's most popular mobile platform—Android Developers, available from ⟨http://developer.android.com/about/index.html⟩
[2] WebView—Android Developers, available from ⟨http://developer.android.com/reference/android/webkit/WebView.html⟩
[3] スマートフォンアプリへのブラウザ機能の実装に潜む危険——WebView クラスの問題について，入手先 ⟨http://codezine.jp/article/detail/6618⟩ (2012.06.25).
[4] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin: Attacks on WebView in the Android System, 27th Annual Computer Security Applications Conference (ACSAC'11), pp.343-352 (2011).
[5] 川端秀明，磯原隆将，竹森敬祐，窪田 歩 : Android パーミッションを悪用する Script の脅威と静的解析，情報処理学会研究報告，vol.2011-CSEC-53，no.3，pp.1-6 (2011).
[6] lexanderA—WebView examples, available form ⟨http://lexandera.com/category/webview_examples/⟩ (2009).
[7] Dedexer, available from ⟨http://dedexer.sourceforge.net/⟩
[8] dex2jar, available form ⟨http://code.google.com/p/dex2jar/#News⟩
[9] JD-GUI—Java Decompiler, available form ⟨http://java.decompiler.free.fr/?q=jdgui⟩
[10] Android セキュリティ勉強会—WebView の脆弱性編，入手先 ⟨http://ierae.co.jp/uploads/webview.pdf⟩ (2012.10.06).
[11] LivingSocial, available form ⟨https://play.google.com/store/apps/details?id=com.livingsocial.www⟩