PROGETTO DI LINGUAGGI E COMPILATORI

MARCO ROSA

MATRICOLA 1014425

Summary

INTRODUCTION

The project I developed is part of my degree thesis, which deals with implementing mandatory access control for third-party apps in AOSP.

In particular, in the first part of this document I show how an AndoidManifest file should be converted in a proper SELinux policy, according to the rules expressed in the paper: *"AppPolicyModule: Mandatory Access Control for Third-Party Apps"*, i.e. generating a file .txt with a proper structure, and in the second part I show how a policy should be compiled to check requirement errors, according to the rules described in the same paper.

COMPILER #1: FROM AndroidManifest TO AppPolicyModule

The first compiler I developed is used to convert the AndroidManifest.xml file that every developer must write in order to share his app. This compiler reads the whole manifest and extracts all information needed to write an AppPolicyModule file.
These information are:

- the name of the package
- the version of the app
- the permissions

Both the name and the version can be found in the tag *<manifest>*, whereas the permissions are written in their own tag *<uses-permission>*

A simple example of AndroidManifest.xml is shown in the picture below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.droid1"
    android:versionCode="1"
    android:versionName="1.0">
    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity
            android:name=".DroidActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action
                    android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk
        android:minSdkVersion="7" />
</manifest>
```

AppPolicyModule FILE

Every appPolicyModule file is in this format:

- Module: it is the header of the file. Here the name of the package is used as name of the policy, and the version as version of the policy
- Head: the code is surrounded by the keyword *require* and curly brackets. Here the name of the apps are preceded by the keyword “type”, whereas the name of the domains are preceded by the keyword “attribute”. Every attribute used in the body has to be declared here.
- Body: there are declarations of *type* rules, *typebounds* rules (the boundaries of the privileges of a type), and *typeattribute* rules (associate a type and an attribute).

There are rules which follow a common pattern:

- Type *untrusted_app* and domains *domain* and *appdomain* are always declared in the head.
- The package is used to declare the first type of the body (name of the package plus “_app”).
- Every type (except *untrusted_app*) must have the type *untrusted_app* as typebounds.
- Every type derived from the name of the package must have typeattribute *domain* and *appdomain*.

Example of an appPolicyModule file:

```
module com.examples.myapp 1.0;
require {
type untrusted_app;
attribute appdomain;
attribute domain;
attribute bluetoothdomain;
}
type com.examples.myapp_app;
typebounds com.examples.myapp_app untrusted_app;
typeattribute com.examples.myapp_app appdomain;
typeattribute com.examples.myapp_app domain;
typeattribute com.examples.myapp_app bluetoothdomain;
```

LEXER

I defined four fragments for the lexer: "*LETTER*" and "*NUMBER*" which do not need any explanation, "*SYMBOL*" which contains all the symbols that can be found in the name of the package (most of all "*.*" and "*_*") or in an attribute list (i.e. token *AL*, see below), and I also had to introduce a fragment "*OTHER_SYMBOLS*" to declare all the symbols which can be found in comments, but are not used in *AL*.

```
fragment LETTER: ('a'..'z'|'A'..'Z');
fragment NUMBER: ('0'..'9');
fragment SYMBOL: ('@'|'/'|'_'|'.'|'|');
fragment OTHER_SYMBOLS: ('('|')'|'"'|':'|','|';'|'-');
```

The token I declared are all the xml tags which a developer can write in a manifest.
Despite the opening or closing tags (e.g. "*TAG_ACTIVITY*" and "*CLOSE_ACTIVITY*"), there are some different tokens used to accomplish the goal of translating the whole manifest, such as *WS* (white spaces) or *SINGLELINE_COMMENT* .

```
//Using channel=HIDDEN, the parser jumps the WS, when it finds one
WS: ( ' ' | '\t' | '\r' | '\n')+ {$channel = HIDDEN;};

//attribute list
AL: ('android:')(LETTER+)('="')(LETTER|SYMBOL|NUMBER)+('"');

CLOSE_ACTIVITY: '</activity>';
CLOSE_ALIAS: '</activity-alias>';
CLOSE_APPLICATION: '</application>';
CLOSE_COMMENT: '-->';
CLOSE_INTENTFILTER: '</intent-filter>';
CLOSE_MANIFEST: '</manifest>';
CLOSE_PROVIDER: '</provider>';
CLOSE_RECEIVER: '</receiver>';
CLOSE_SERVICE: '</service>';

CLOSENORMAL: '>';

CLOSETAG: '/>';

SINGLELINE_COMMENT:   '<!--' (options {greedy=false;} : ~('\r' | '\n'))* ('\r' | '\n'
)+ {$channel=HIDDEN;};

HEADER: '<?xml version="1.0" encoding="utf-8"?>';

TAG_ACTION: '<action';
TAG_ACTIVITY: '<activity';
TAG_ALIAS: '<activity-alias>';
TAG_APPLICATION: '<application';
TAG_CATEGORY: '<category';
TAG_COMPATIBLESCREENS: '<compatible-screens';
TAG_DATA: '<data';
TAG_GRANTURI: '<grant-uri-permission>';
TAG_INSTRUMENTATION: '<instrumentation ';
TAG_INTENTFILTER: '<intent-filter>';
TAG_LIBRARY: '<uses-library>';
TAG_MANIFEST: '<manifest ';
TAG_METADATA: '<meta-data ';
TAG_PATHPERM: '<path-permission>';
TAG_PERM: '<permission';
```

```
TAG_PERMGROUP: '<permission-group';
TAG_PERMTREE: '<permission-tree';
TAG_PROVIDER: '<provider>';
TAG_RECEIVER: '<receiver>';
TAG_SERVICE: '<service>';
TAG_SUPPORTSSCREEN: '<supports-screens';
TAG_SUPPORTSGL: '<supports-gl-texture';
TAG_USESCONFIGUTATION: '<uses-configuration';
TAG_USESFEATURES: '<uses-feature';
TAG_USESSDK: '<uses-sdk';

XMLNS: 'xmlns:android="http://schemas.android.com/apk/res/android"';
```

I would like to focus attention on these tokens:

```
PACKAGE: ('package="') ((LETTER+)('.')(LETTER+)('.')(LETTER|SYMBOL|NUMBER)+('"'));

TAG_PERMISSION: '<uses-permission android:name="android.permission.';

PERMESSO: 'ACCESS_CHECKIN_PROPERTIES'|
          'ACCESS_COARSE_LOCATION'|
          'ACCESS_FINE_LOCATION'|
          'ACCESS_LOCATION_EXTRA_COMMANDS'|
          'ACCESS_MOCK_LOCATION'|
             …
             ;
```

PACKAGE is the token which contains the name of the package, which is very important in order to write the appPolicyModule file (see next chapter). It is a particular case of *AL* which I decided to headlight and extract the value of the attribute in an easier way.

PERMESSO is the token which contains the name of the permission. It is not a single line or a complete attribute of *AL*, but it is the conclusion of the token "*TAG_PERMISSION*". I decided to separate it to easily convert a permission into a SELinux domain
Example of the whole tag, as it is written in an AndroidManifest file:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

PARSER

The first rule is the start rule, which contains the principal structure of the file

START

```
start
@init { init (); } :
      HEADER
      comment?
      body
      { handler.writeToFile(); }
      ;
```

`init()` is a java method declared in *members*, which is used to initialize the variables of the parser. In this case it initializes the variable *handler* of the class Handler (see next chapter for a discussion on this class).
HEADER is a token from the scanner
`comment` and `body` are parser rules described below
`{ handler.writeToFile(); }` is the Java method which must be called after parsing the document in order to write the output file

COMMENT

```
comment: SINGLELINE_COMMENT CLOSE_COMMENT ;
```

In every AndroidManifest.xml file it is possible to insert comments (e.g. *<!--this is a comment -->*). I assume that these comments can just be inserted after the header of the file (in many files there is not a hint of comments at all).

BODY

```
body: begin_manifest
      node*
      CLOSE_MANIFEST
      ;
```

The body rule is the main rule of the file.
It is composed of different rules: a begin rule (`begin_manifest`) where it is declared the opening tag in, a "middle" rule (`node`) where the content is shown in, and a final rule which is a token from the lexer (*`CLOSE_MANIFEST`*, i.e. the closing tag of the rule).

An example of the code parsed by this rule is shown in the picture below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="string"
          android:sharedUserId="string"
          android:sharedUserLabel="string resource"
          android:versionCode="integer"
          android:versionName="string"
          android:installLocation=["auto" | "internalOnly" | "preferExternal"] >
    . . .
</manifest>
```

Begin manifest

```
begin_manifest: TAG_MANIFEST
                XMLNS
                pack=PACKAGE
                (vers=AL{handler.addVersions(vers.getText());})*
                CLOSENORMAL
        {
          handler.setPack(pack.getText());
        };
```

The rule *begin_manifest* is really important for this compiler because, as it can be seen from the code, it is the rule which parses the portion of code the developer declares the package and the version in. So, with regard to get these information, two different Java methods are called: *setPack* and *addVersions* (see next chapter).

The difference between the way they have been used is that *setPack* has been used at the end of the rule, because the variable *pack* is unique and it is found as the parser recognizes the token *PACKAGE*, whereas *addVersions* must be called every time the parser recognizes a AL token, because it is not possible to forecast the position of the attribute *android:versionName* in the attribute list.

Node

The rule *node* is a complex rule, which contains all the tags that can be found inside the tag manifest:

```
node: sdk|
      permission|
      application|
      //less used:
      instrumentation|
      uses_features|
      supports_screen|
      compatible_screen|
      gltexture|
      permissiontree|
      permissiongroup|
      permission_normal
      ;
```

Except for `application`, all the other rules are in the same format:
TAG_XXX AL CLOSETAG*;
where *XXX* is the name of the tag and *TAG_XXX* is a token from the grammar:

```
sdk: TAG_USESSDK AL* CLOSETAG ;
instrumentation: TAG_INSTRUMENTATION AL* CLOSETAG;
uses_features: TAG_USESFEATURES AL* CLOSETAG ;
supports_screen: TAG_SUPPORTSSCREEN AL* CLOSETAG ;
compatible_screen: TAG_COMPATIBLESCREENS AL* CLOSETAG ;
gltexture: TAG_SUPPORTSGL AL* CLOSETAG ;
permissiontree: TAG_PERMTREE AL* CLOSETAG ;
permissiongroup: TAG_PERMGROUP AL* CLOSETAG ;
permission_normal: TAG_PERM AL* CLOSETAG ;
```

A bit different is the case of the rules `permission` and `application`, which are described in the following paragraphs.

PERMISSION

```
permission: TAG_PERMISSION
            perm=PERMESSO
            ('"')
            AL*
            CLOSETAG
            {handler.addPermission(perm.getText());};
```

Example of a permission: `<uses-permission android:name="android.permission.INTERNET"/>`

TAG_PERMISSION was defined in the lexer grammar as a tag plus the attribute *android:name* and the beginning of its value. This was done both to make more simple getting the name of the permission and because it is always used as first attribute. The attribute *android:name* (i.e. the attribute I care in this project) is not the only attribute allowed to be used with the tag *uses-permission*: there is also another attribute (*android:maxSdkVersion*), but it is rarely used and it never appears as first attribute. Java method *getText* is used to get the name of the variable *perm*, which is not a String but a Token.

APPLICATION

Application is a complex tag, but it is not critical for the purpose of this compiler because all essential information for writing the appPolicyModule file are in *manifest* and *uses-permission* tags. The format of the application rule is this:

```
application: begin_application
                 application_node+
                 CLOSE_APPLICATION
                 ;
```

It is the mirror rule of the body rule: there are a rule “begin” (`begin_application`), which declares the opening tag and the attribute list, a rule “close”, which is the token of the closing tag (*CLOSE_APPLICATION*), and, between them, another rule (`application_node`), which is a node declaring all the tags that can be contained in an activity tag.

```
begin_application: TAG_APPLICATION AL* CLOSENORMAL ;
```

Application node

```
application_node: activity_tag|
                  alias_activity|
                  service|
                  receiver|
                  provider|
                  metadata|
                  library
                  ;
```

The rule `application_node` has the same format of the previous rule `node` (cf. Body paragraph): a list of other rules which can be single line rules (e.g. `library`) or more structured rules (e.g. `activity_tag`). So are the rules contained in `application_node`, in such a recursive way.

```
alias_activity: TAG_ALIAS AL* CLOSENORMAL internal_tag* CLOSE_ALIAS ;
service: TAG_SERVICE AL* CLOSENORMAL internal_tag* CLOSE_SERVICE ;
receiver: TAG_RECEIVER AL* CLOSENORMAL internal_tag* CLOSE_RECEIVER ;
library: TAG_LIBRARY AL* CLOSETAG WS*;
metadata: TAG_METADATA AL* CLOSETAG WS*;

//Activity tag:
activity_tag: begin_activity
              internal_tag*
              CLOSE_ACTIVITY
              ;

begin_activity: TAG_ACTIVITY AL* CLOSENORMAL ;


internal_tag: intent_filter_tag|
              metadata
              ;

//Intent filter:
intent_filter_tag: begin_intent_filter
                   intent_filter_node*
                   CLOSE_INTENTFILTER
                   ;

//intent filter can begin with or without space before '>'
begin_intent_filter: TAG_INTENTFILTER|'<intent-filter >' ;

intent_filter_node: action_tag|
                    category_tag|
                    data_tag
                    ;

action_tag: TAG_ACTION AL* CLOSETAG ;

category_tag: TAG_CATEGORY AL* CLOSETAG ;

data_tag: TAG_DATA AL* CLOSETAG ;


//Provider:
provider: begin_provider
          provider_node*
          CLOSE_PROVIDER
          ;

begin_provider: TAG_PROVIDER AL* CLOSENORMAL ;

provider_node: uri_permission|
               metadata|
               path_permission
               ;

uri_permission: TAG_GRANTURI AL* CLOSETAG ;

path_permission: TAG_PATHPERM AL* CLOSETAG ;
```

After writing the parser, the file *myGrammar.g* is completed and can be compiled. It automatically generates the package *myCompiler* which contains two classes: *myGrammarLexer* (the class of the lexer) and *myGrammarParser* (the class of the parser). These classes are used in the main class to scan and parse the input file.

CLASS HANDLER

Class Handler was created to support the grammar file with some methods used to extract the name of the package, the version and the permissions, and to write the output file.

This class has five fields:

```
/** The package. */
private String pack;

/** The version. */
private String version;

/** The permissions. */
private ArrayList<String> perms;

/** The domains. */
private HashSet<String> domini;

/** The output file. */
private String fileOut = ".\\resources\\appPolicyModule.txt";
```

and a constructor which initializes these fields (except *fileOut* which is a constant):

```
public Handler(){
	pack="";
	version="";
	perms=new ArrayList<String>();
	domini=new HashSet<String>();
}
```

Handler class also has some set and get methods, besides add methods to add elements in arrays and sets:

```
/**
 * Set the name of the package.
 *
 * @param p the name of the package
 */
public void setPack(String p){
	String temp=extractPackage(p);
	if(temp!=null) pack=temp;
}

/**
 * Add a permission.
 *
 * @param permission the name of the permission
 */
public void addPermission(String permission){
	perms.add(permission);
}

/**
 * Add the number of version.
 *
 * @param version the version of the manifest
 */
public void addVersions(String version){
```

```
            String temp=extractVersion(version);
            if(temp!=null) this.version=temp;
      }

      /**
       * Get the name of the package.
       *
       * @return the package
       */
      public String getPack(){
            return pack;
      }

      /**
       * Get all the permissions.
       *
       * @return an array containing the names of the permissions
       */
      public ArrayList<String> getPermission(){
            return perms;
      }
```

As it can be seen from *setPack* and *addVersions*, there are two private methods used to extract the name of the package and the version (already mentioned in the previous chapter):

SetPack

```
      /**
       * Extract the package from the attribute "package" (tag <manifest>) of
AndroidManifest
       * Ex: package="com.examples.myapp"
       *
       * @param p the attribute "package" from AndroidManifest
       * @return a String containing the name of the package
       */
      private String extractPackage(String p){
            String[]s=p.split("=");
            if(s[0].equalsIgnoreCase("package"))
                  return s[1].substring(1, s[1].length()-1);
            else return "";
      }
```

This method was written because the argument *p* is a String in the format `package="com.package.name"`
For this reason, I firstly split *p* at the character *"="* obtaining two strings: the first one is the name of the attribute (`package`) and the second is the name of the package, surrounded by quotation marks (`"com.package.name"`). So, after a check that the first string is equal to "package" (i.e. the attribute is "package"), the latter is returned with a substring to exclude the quotation marks (i.e. the value of the attribute is returned).

ExtractVersion

```
      /**
       * Extract the package from the attribute "versionName" (tag <manifest>) of
AndroidManifest
       * Ex: android:versionName="1.0"
       *
       * @param s the attribute "versionName" from AndroidManifest
```

```
	 * @return a String containing the version
	 */
	private String extractVersion(String s){
		String st[]=s.split("=");
		if(st[0].equalsIgnoreCase("android:versionName"))
			return st[1].substring(1, st[1].length()-1);
		else return "";
	}
```

This method is the dual of the previous one. The argument *s* is in the format `android:versionName="1.0"`

Handler class contains two other methods: a private one to convert permissions into domains and a public one to write the output file.

FillDomini

```
	/**
	 * Convert all permissions found inside the tag uses-permissions of an AndroidManifest
	 * into the corresponding SELinux domain.
	 */
	private void fillDomini(){
		PermissionHashtable ph=new PermissionHashtable();
		//domains "domain" and "appdomain" must be included in every policy
		domini.add("domain");
		domini.add("appdomain");
		//conversion android permission -> selinux domain
		for(String p:perms)
			domini.add(ph.getDomain(p));
	}
```

This method is used to convert the permissions into SELinux domains. According to paper "*AppPolicyModules: Mandatory Access Control for Third-Party Apps*", which this project is based on, every appPolicyModule file must include *domain* and *appdomain* domains (this is why there is a handmade addition of these two domains). See next chapter (PermissionHashtable class) for further information on conversion.

WriteToFile

Finally, the method *writeToFile* is used to write an output file and show results on console:

```
	/**
	 * Write the policy module on a file .
	 */
	public void writeToFile(){
		if(pack.length()==0){
			System.err.println("No name for the policy detected");
			return;
		}
		if(version.length()==0) version="1.0";
		//the name of the package is used for the name of the policy and for the name of the type of the app
		fillDomini();

		//////////////////////PRINT POLICY ON CONSOLE://///////////////////////
		System.out.println("module "+pack+" "+version+";");
		//HEAD
```

```
		System.out.println("require {");
		System.out.println("type untrusted_app;");
		//conversion from Android permission to SELinux domain
		for(String s:domini)
			System.out.println("attribute "+s+";");
		System.out.println("}");
		//BODY
		String type_app=pack+"_app";
		System.out.println("type "+type_app+";");
		System.out.println("typebounds "+type_app+" untrusted_app;");
		for(String s:domini)
			System.out.println("typeattribute "+type_app+" "+s+";");

		////////////////////////////////////////////////////////7//
		//WRITING ON A FILE:
		try {
			FileWriter fOut = new FileWriter (fileOut);
			fOut.append("module "+pack+" "+version+";\n");
			//HEAD
			fOut.append("require {\n");
			fOut.append("type untrusted_app;\n");
			//conversion from Android permission to SELinux domain
			for(String s:domini)
				fOut.append("attribute "+ s +";\n");
			fOut.append("}\n");
			//BODY
			//String type_app=type+"_app";
			fOut.append("type "+type_app+";\n");
			fOut.append("typebounds "+type_app+" untrusted_app;\n");
			for(String s:domini)
				fOut.append("attribute "+ type_app + s +";\n");
			fOut.close();

		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
```

The format of the output file was explained in chapter “appPolicyModule FILE”.

CLASS PERMISSIONHASHTABLE

The class PermissionHashtable was created as a Hashtable with a mapping between all Android permissions and SELinux domains. This class has an only field:

```
Hashtable<String,String> permissions
```

Which is initialized in the constructor of this class:

```
public PermissionHashtable(){
        permissions=new Hashtable<String,String>();

        //fill the table with all android permissions and their respective selinux
        modules
        permissions.put("ACCESS_CHECKIN_PROPERTIES", "appdomain");
        permissions.put("ACCESS_COARSE_LOCATION", "netdomain");
        permissions.put("ACCESS_FINE_LOCATION", "netdomain");
        permissions.put("ACCESS_LOCATION_EXTRA_COMMANDS", "appdomain");
        permissions.put("ACCESS_MOCK_LOCATION", "appdomain");
        permissions.put("ACCESS_NETWORK_STATE", "netdomain");
        …
}
```

The keys of this Hashtable are all the Android permissions (they are 151, listed at *http://developer.android.com/reference/android/Manifest.permission.html*), and their value is the name of the correspondent SELinux domain.

Since it does not exist an official mapping between permissions and domains while writing this document, this was done in a simple way analyzing the repository at *https://android.googlesource.com/platform/external/sepolicy*

As written in paper *"AppPolicyModule: Mandatory Access Control for Third-Party Apps" (chapter 7)* it would be very useful having a correspondence 1:1 between Android permissions and SELinux domains.

Finally, this class also has a method used to apply the conversion:

```
/**
 * Given an Android permission, return the proper selinux domain.
 *
 * @param s a String containing the permission
 * @return the domain
 */
public String getDomain(String s){
        return permissions.get(s);
}
```

Every time this method is called, argument *s* is a String containing the name of an Android permission, and the return String is the correspondent SELinux domain. For this purpose, I took advantage of the method "get" of Hashtable class.

MAIN CLASS

The main class is “myTester” class. It has a main method, where four variables are declared:

```
String fileIn = "…\path\AndroidManifest.xml";
myGrammarLexer lexer;
myGrammarParser parser;
CommonTokenStream tokens;
```

fileIn is the path of the file manifest .
myGrammarLexer and myGrammarParser are the auto-generated classes for the lexer and the parser.
CommonTokenStream is a class which filters token streams to tokens on a particular channel.

These variables are used in this sequence:

```
lexer = new myGrammarLexer(new ANTLRReaderStream(new
FileReader(fileIn)));
tokens=new CommonTokenStream(lexer);
parser= new myGrammarParser(tokens);
```

and then the starting rule is thrown:

```
parser.start();
```

These lines have to be surrounded by a try-catch clause.

EXAMPLE

I show an example of the functioning of this compiler creating the appPolicyModule file of the AndroidManifest.xml for YouTube Android Player API app, which can be found at *https://developers.google.com/youtube/android/player/downloads/*.

I edited the comment of the app from the original:

```
<!--
  Copyright 2012 Google Inc. All Rights Reserved.

  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
  See the License for the specific language governing permissions and
  limitations under the License.
-->
```

To:

```
<!--
  Copyright 2012 Google Inc. All Rights Reserved.
-->
```

Because this compiler does not manage a multi-line comment.

AndroidManifest.xml (edited) for YouTube Android Player API app:

```
<?xml version="1.0" encoding="utf-8"?>

<!--
  Copyright 2012 Google Inc. All Rights Reserved.
-->

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.examples.youtubeapidemo"
    android:versionCode="1"
    android:versionName="1.0" >

  <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="16"/>

  <uses-permission android:name="android.permission.INTERNET"/>

  <application android:label="@string/youtube_api_demo">

    <activity
        android:label="@string/youtube_api_demo"
        android:name=".YouTubeAPIDemoActivity">
      <intent-filter >
        <action android:name="android.intent.action.MAIN" />
```

```
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
  <meta-data android:name="@string/isLaunchableActivity" android:value="false"/>
</activity>

<activity
    android:label="@string/videowall_demo_name"
    android:name=".VideoWallDemoActivity"
    android:screenOrientation="landscape"
    android:theme="@style/BlackNoBarsTheme"
    android:configChanges="orientation|screenSize|keyboardHidden">
  <meta-data android:name="@string/minVersion" android:value="11"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>

<activity
    android:label="@string/playerview_demo_name"
    android:name=".PlayerViewDemoActivity"
    android:screenOrientation="nosensor"
    android:configChanges="orientation|screenSize|keyboardHidden">
  <meta-data android:name="@string/minVersion" android:value="8"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>

<activity
    android:label="@string/fragment_demo_name"
    android:name=".FragmentDemoActivity"
    android:screenOrientation="nosensor"
    android:configChanges="orientation|screenSize|keyboardHidden">
  <meta-data android:name="@string/minVersion" android:value="11"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>

<activity
    android:label="@string/player_controls_demo_name"
    android:name=".PlayerControlsDemoActivity"
    android:windowSoftInputMode="stateHidden">
  <meta-data android:name="@string/minVersion" android:value="8"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>

<activity
    android:label="@string/fullscreen_demo_name"
    android:name=".FullscreenDemoActivity"
    android:screenOrientation="sensor"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:theme="@style/BlackNoTitleBarTheme">
  <meta-data android:name="@string/minVersion" android:value="8"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>

<activity
    android:label="@string/action_bar_demo_name"
    android:name=".ActionBarDemoActivity"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:theme="@style/OverlayActionBarTheme"
    android:screenOrientation="sensorLandscape">
  <meta-data android:name="@string/minVersion" android:value="11"/>
  <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
</activity>
```

```
    <activity
        android:label="@string/standalone_player_demo_name"
        android:name=".StandalonePlayerDemoActivity">
      <meta-data android:name="@string/minVersion" android:value="8"/>
      <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
    </activity>

    <activity
        android:label="@string/intents_demo_name"
        android:name=".IntentsDemoActivity">
      <meta-data android:name="@string/minVersion" android:value="8"/>
      <meta-data android:name="@string/isLaunchableActivity" android:value="true"/>
    </activity>

  </application>

</manifest>
```

Using a main class like the one described in the previous chapter, the start rule of the parser initializes the variable *handler*, parses the header token, finds out the comment and then moves to the body rule.
At this point, the parser follows the rule *begin_manifest* and at the code

```
pack=PACKAGE
(vers=AL{handler.addVersions(vers.getText());})*
```

it saves the token *PACKAGE* in a variable named *pack*, and executes the method *addVersions* at every attribute it finds. This method calls a private method, which saves in the variable *version* of the class Handler only the right value (in this example: "1.0").
At the end of the *begin_manifest* rule, another Java method of the Handler class is called:

```
handler.setPack(pack.getText());
```

This method is built like *addVersions*, described above: it calls another private method named *extractPackage*, which does the job of saving in the variable *pack* of the class Handler the right value.

At the end of this rule, the parser passes to the node rule. At first the parser finds a tag "uses-sdk" in the manifest, so it executes the *sdk* rule.

Then, it finds a "uses-permission" tag, so it follows the *permission* rule:

```
permission: TAG_PERMISSION
            perm=PERMESSO
            ('"')
            AL*
            CLOSETAG
            {handler.addPermission(perm.getText());};
```

The parser recognizes the token *TAG_PERMISSION* and saves the following one, *PERMESSO*, in a variable (*perm*). At the end of the rule this variable is used to save the name of the permission in the array *perms* of the class Handler.

There is a tag "application" afterwards, so the parser continues its job executing the proper rules until it finds the token CLOSE_MANIFEST, which closes the body rule.

At the end, a final method is executed:

```
{ handler.writeToFile(); }
```

which has the assignment of printing both on console and on file the final policy. This method was described in the previous chapter, and the result is this file:

```
module com.examples.youtubeapidemo 1.0;
require {
type untrusted_app;
attribute appdomain;
attribute domain;
attribute netdomain;
}
type com.examples.youtubeapidemo_app;
typebounds com.examples.youtubeapidemo_app untrusted_app;
attribute com.examples.youtubeapidemo_app appdomain;
attribute com.examples.youtubeapidemo_app domain;
attribute com.examples.youtubeapidemo_app netdomain;
```

COMPILER #2: REQUIREMENTS INSPECTOR

The second compiler I developed is used to check two requirements that every policy has to satisfy, both described in paper *"AppPolicyModules: Mandatory Access Control for Third-Party Apps" (Chapter 5)*.

The first requirement (i.e.: *No impact on the system policy*) says that *"[…] the provided appPolicyModule must not be able to have an impact on privileges where source and target are system types"*. This means that every time there is an *allow* rule in the appPolicyModule file, both source and target types must be previously declared in the policy. This was done also for attributes in this compiler: it is not allowed declare a *typeattribute* rule if the type was not declared before.

The second requirement (i.e.: *No escalation*) says that *"the app cannot specify a policy that provides to its types more privileges than those available to untrusted_app"*. This means that every time a *typebounds* rule is declared, the boundary domain must be *"untrusted_app"*.

LEXER

As done for the first compiler, I declared three fragments: *LETTER*, *NUMBER* and *SYMBOL*.

```
fragment LETTER: ('a'..'z'|'A'..'Z');
fragment NUMBER: ('0'..'9');
fragment SYMBOL: ('_'|'.');
```

These are used for the last two tokens:

```
VERSION: NUMBER+'.'NUMBER+;
ID: LETTER(LETTER|SYMBOL|NUMBER)*;
```

VERSION is the token which identifies the version of the policy module, whereas *ID* is the token of the name of the types. This must be put in the last position of the lexer because ANTLR's parser is greedy (it consumes as much input as it can match). This is the reason why, even though I declared the tokens in alphabetical order, *TYPE* is declared after *TYPEATTRIBUTE* and *TYPEBOUNDS*.

The other tokens are:

```
WS: ( ' ' | '\t' | '\r' | '\n')+ {$channel = HIDDEN;};

ALLOW: 'allow' ;
ATTRIBUTE: 'attribute' ;
CLOSE_REQUIRE: '}';
END: ';';
MODULE : 'module' ;
NEVERALLOW: 'neverallow' ;
OPEN_REQUIRE: 'require' WS* '{';
SINGLELINE_COMMENT: '#' (options {greedy=false;} : ~('\r' | '\n'))* ('\r' | '\n' )+
{$channel=HIDDEN;};
TYPEATTRIBUTE: 'typeattribute' ;
TYPEBOUNDS: 'typebounds' ;
TYPE: 'type' ;
```

PARSER

The parser declares all the rules an appPolicyModule must follow. They are written in the same file of the lexer's tokens: *Grammatica.g*.
Compiling this file two classes are automatically generated: *GrammaticaLexer* and *GrammaticaParser*, included in *myCompiler* package.
These classes are used in the main class (cf. *Tester Class* chapter).

The parser begins with a start rule, which is the most important rule:

START

```
start
      @init { init (); }:
      beginning*
      header
      require
      body+
      {handler.checkWarnings();}
      {handler.verifyPolicy();}
      ;
```

This rule begins with the Java function `init ()` which is declared in field `@members` of the compiler:

```
@members{
  Handler handler;

  void init(){
    handler=new Handler();
    }

}
```

The aim of this function is initializing the variable *handler* of class Handler (cf. next chapter).

After executing this function, a `beginning` rule is found.

At the end of the start rule, two Java methods are executed: the former checks the warnings, the latter verifies the policy.

BEGINNING

```
beginning: WS|SINGLELINE_COMMENT;
```

This rule is used to manage possible white spaces or single line comments at the beginning of the file. There also can be no white spaces or no comments, so the multiplicity of this rule is 0:N (as it can be seen from the *).

Example of a comment:

```
#this is a comment
```

HEADER

```
header: MODULE ID VERSION END ;
```

The header of a policy module is the real beginning of the file. This rule is composed of four tokens: *MODULE* is the token containing the word "*module*", *ID* is the name of the policy, *VERSION* is the number of the version, *END* is the *;* at the end of the row.

Example:

```
module com.examples.name 1.0;
```

REQUIRE

```
require: OPEN_REQUIRE node_require* CLOSE_REQUIRE ;
```

`require` is the first complex rule. It begins with the token *OPEN_REQUIRE* and closes with the *CLOSE_REQUIRE* one. In the middle there can be a node rule called `node_require` which contains a `type` and an `attribute` rule:

```
node_require: type|
              attribute
              ;
```

Node require

The type rule is dealt with separately in the next paragraph, whereas the attribute rule is a simple rule used to declare an attribute (i.e. a SELinux domain):

```
attribute: ATTRIBUTE ID END;
```

Example:

```
require {
type name_of_the_type;
attribute attr1;
attribute attr2;
…
}
```

TYPE

```
type: TYPE t=ID END
  {
    handler.addType(t.getText(),t.getLine());
  }
  ;
```

A type rule can be declared both in the header and in the body of a policy. Every time the parser founds the *TYPE* token and recognizes this rule, it saves the token *ID* in a variable called *t*, and a Java method is executed by the variable *handler* at the end of the rule (cf. next chapter).

Example:

```
type name_type;
```

BODY

```
body: type|
      typebounds|
      typeattribute|
      allow|
      neverallow
      ;
```

The `body` rule includes the declarations of all the possible rules of the body of a appPolicyModule file, i.e. type, typebounds, typeattribute, allow and neverallow. For a better discussion of rules that must appear in every appPolicyModule file see chapter "*AppPolicyModule FILE*" in section "*COMPILER#1: FROM AndroidManifest TO appPolicyModule*".

The `type` rule was described in the previous paragraph.

Neverallow

The `neverallow` rule is simple and not critical for this compiler because it never breaks requirements:

```
neverallow: NEVERALLOW ID ID END;
```

Example:

```
neverallow app_name domain_name ;
```

The other rules are dealt with in the following chapters.

TYPEBOUNDS

```
typebounds: TYPEBOUNDS t=ID bound=ID END
            {
              handler.addTypebounds(t.getText(),t.getLine());
              //check requirements 2
              handler.checkRequirements2(t.getText(),bound.getText());
            }
            ;
            catch [Requirement2Exception e]
            {
              handler.addError(new ErrorToken(e.getMessage(),t.getLine()));
            }
```

This is the rule that declares the boundaries of the privileges for a type. It begins with the token *TYPEBOUNDS* ("*typebounds*") and finishes with the *END* (";") one.
The first *ID* contains the name of a type, whereas the second one is the name of the boundary domain. As described before, the name of the boundary must be "*untrusted_app*" so this rule could break the requirement number 2.

Two methods are executed in the final Java code (cf. next chapter) to check the requirement and throw an exception if it is broken. The exception is handled adding a message in a list.

Example:

```
typebounds myapp untrusted_app;  ← OK
typebounds myapp domain_X;       ← NO
```

TYPEATTRIBUTE

```
typeattribute: TYPEATTRIBUTE t=ID ID END
               {
                  //check requirements 1
                  handler.checkRequirements1(t.getText());
               }
               ;
               catch [Requirement1Exception e]
               {
                  handler.addError(new ErrorToken(e.getMessage(),t.getLine()));
               }
```

This rule assigns a domain (the second *ID*) to a type (the first *ID*). The first *ID* token is saved in a variable *t*, which is used to check the first requirement. In fact, as said before, every attribute must be assigned to a type which has to be previously declared. A Java method is used to check this requirement, and if this is broken an exception is thrown. The exception is handled adding a message in a list.

Example:

```
typeattribute myapp domain;
```

ALLOW

```
allow: ALLOW source=ID target=ID END
       {
          //check requirements 1:
          handler.checkRequirements1(source.getText(),target.getText());
       }
       ;
       catch [Requirement1Exception e]
       {
          handler.addError(new ErrorToken(e.getMessage(),source.getLine()));
       }
```

Every time the parser finds an `allow` rule, it must check the first requirement. Both the *ID* tokens are saved in a variable, *source* the former and *target* the latter, which are used as arguments in a Java method. If the requirement is broken, an exception is thrown. The exception is handled adding a message in a list.

Example:

```
allow source_type target_type ;
```

CLASS HANDLER

The class "Handler" is a class created to check the requirements. For this purpose, the class has two sets: the first contains the names of the types from *type* rules, the latter contains the names of the types from *typebounds* rules. Furthermore this class contains two lists, used for error handling: the first one is the error list, the latter one is the warning list.

Therefore, as said, there are four fields:

```
private Hashtable<String,Integer> avTypes;
private Hashtable<String,Integer> typebounds;
private ArrayList<ErrorToken> errorList;
private ArrayList<ErrorToken> warningList;
```

which are initialized in the class constructor:

```
public Handler(){
	avTypes=new Hashtable<>();
	typebounds=new Hashtable<>();
	errorList=new ArrayList<>();
	warningList=new ArrayList<>();
}
```

As it is visible from the code, `avTypes` (the set containing the types from *type* rules) and `typebounds` (the set containing the types from *typebounds* rules) are Hashtables. This was done in order to keep the name of the type (a String) and the line the type is declared in (an Integer). Thanks to this, it is more simple inserting the wrong line in error messages.

There are some methods to add elements in these Hashtables:

```
/**
 * Adds a type to the list.
 * In this list there are the types declared in type rules
 *
 * @param type the name of the type
 * @param line the line of the rule
 */
public void addType(String type, int line){
	if(!avTypes.containsKey(type))
		avTypes.put(type,new Integer(line));
}
```

This method adds a type to the list. The type is declared in a *type* rule and it is only inserted if its name does not already exist. This was done to prevent duplicates: the developer could write the same type rule in different lines.

```
/**
 * Adds a type to the list.
 * In this list there are the types declared in typebounds rules
 *
 * @param type the name of the type
 * @param line the line of the rule
 */
public void addTypebounds(String type, int line){
	if(!typebounds.containsKey(type))
	typebounds.put(type,new Integer(line));
}
```

This method is the dual of the previous one. It inserts a type declared in a *typebounds* rule to the list, i.e. the name of the type the bounds are applied to.

There are two methods to add messages to the error list, too:

```
/**
 * Adds an error message.
 *
 * @param et the Error Token containing the message and the line of the error
 */
public void addError(ErrorToken et){
	errorList.add(et);
}

/**
 * Adds an error message.
 *
 * @param line the line of the error
 * @param msg the message
 */
public void addError(int line, String msg){
	errorList.add(new ErrorToken(msg, line));
}
```

`errorList` (and `warningList`) are ArrayList of ErrorToken. ErrorToken is a class containing a message and the line of the error, so these are equivalent methods. See next chapter for further information on ErrorToken class.

CheckRequirement1

In addition to this, there are some methods to check the requirements, actually the real scope of Handler class:

```
/**
 * Check requirement 1 described in paper
 * "AppPolicyModules: Mandatory Access Control for Third-Party Apps"
 * Chapter 5.
 *
 * @param type a String containing the type to be checked
 * @throws Requirement1Exception if the rule is broken
 */
public void checkRequirements1(String type)throws Requirement1Exception{
	if(!avTypes.containsKey(type))
		throw new Requirement1Exception("(Req1) "+type+" undefined");
}
```

This method is used to check the first requirement: whenever a type is found in a *typeattribute* rule, if it has not been declared yet, this method throws an exception, which adds this message to *errorList*:

```
line number_of_the_line: (Req1) type_name undefined
```

```
/**
 * Check requirement 1 described in paper
 * "AppPolicyModules: Mandatory Access Control for Third-Party Apps"
 * Chapter 5.
 *
 * @param source a String containing a type to be checked
```

```
	 * @param target a String containing a type to be checked
	 * @throws Requirement1Exception if the rule is broken
	 */
	public void checkRequirements1(String source, String target)throws
Requirement1Exception{
		if(!avTypes.containsKey(source))
			throw new Requirement1Exception("(Req1) "+source+" undefined");
		if(!avTypes.containsKey(target))
			throw new Requirement1Exception("(Req1) "+target+" undefined");
	}
```

This method does the same job of the previous one, yet for the *allow* rules. Every time an *allow* rules is parsed, the Handler checks that both the source and the target have already been declared. If one of them is still unknown, an exception is thrown, with an error message added to *errorList*:
`line number_of_the_line: (Req1) type_name undefined`

CheckRequirement2

```
	/**
	 * Check requirement 2 described in paper
	 * "AppPolicyModules: Mandatory Access Control for Third-Party Apps"
	 * Chapter 5.
	 *
	 * @param type a String containing a type to be checked
	 * @param bound a String containing a type to be checked. This type must be
"untrusted_app"
	 * @throws Requirement2Exception if the rule is broken
	 */
	public void checkRequirements2(String type, String bound) throws
Requirement2Exception{
		if(!bound.equalsIgnoreCase("untrusted_app")) throw new
			Requirement2Exception("(Req2) The type "+type+" has not
typebounds=untrusted_app but "+bound);
	}
```

This method is used to check the second requirement. Every time a *typebounds* rule is parsed, the boundary must be controlled (i.e. the second *ID* token, cf. TYPEBOUNDS paragraph in PARSER chapter). If this is not "*untrusted_app*", an exception is thrown, adding a message of this kind to *errorList*:
`line n: (Req2) The type type_name has not typebounds=untrusted_app but xxx_app`

The first argument of this method (String type) is just used for the message. In fact it is useful to let the user understand what is the type which has a wrong boundary.

CheckWarnings

According to the paper, the first requirement is broken if an allow rule or a typeattribute rule use a non-declared type, whereas the second requirement is broken if a typebounds rule has not *untrusted_app* as boundary type. There are cases which are not dealt with in the paper, so I used another method to check them, at the end of the parsing:

```
	/**
	 * A final check of properties deducted from requirements described in paper
	 * "AppPolicyModules: Mandatory Access Control for Third-Party Apps"
	 * Chapter 5
```

```
	 * First check: every type defined in a typebounds sentence must be previously
declared
	 * Second check: every defined type must have a typebounds
	 * Every line breaking these rules generates a warning.
	 */
	public void checkWarnings(){
		for(String b:typebounds.keySet()){
			if(!avTypes.containsKey(b)){
			warningList.add(new ErrorToken(new String("The type "+b+" appears in a"
					+ " typebounds sentence without ever being defined"),
						typebounds.get(b).intValue()));
			}
		}
		for(String t:avTypes.keySet()){
		if(!(t.equalsIgnoreCase("untrusted_app") | typebounds.containsKey(t))) {
				warningList.add(new ErrorToken(new String("The type "+t+
						" has not a defined typebounds"),
avTypes.get(t).intValue()));
			}

		}
	}
```

As it can be seen from the code, there are two for-cycles which could signal two kind of warnings. The first warning is signaled if a non-declared type appears in a *typebounds* rule. A message of this kind is added to *warningList*:

```
line n: The type type_name appears in a typebounds sentence without ever being defined
```

The second warning is signaled id a declared type does not have a bounds, that is if it does not appear in a *typebounds* rule, and a message is added to *warningList*:

```
line n: The type type_name has not a defined typebounds
```

VerifyPolicy

Finally, there is a method for verifying the policy:

```
	/**
	 * Do a verification of the policy:
	 * print all the warnings and the errors (contained in two different lists)
	 * If there aren't errors, tip off the user that the policy is OK.
	 */
	public void verifyPolicy(){
		if(warningList.size()!=0){
			System.err.println("WARNINGS:");
			Collections.sort(warningList);  //order the warnings according to
their line
		for(ErrorToken et:warningList)
			System.err.println(et.toString());
		}

		if(errorList.size()==0)
			System.out.println("Correct policy");
		else{
				System.err.println("ERRORS:");
				for(ErrorToken et:errorList)
					System.err.println(et.toString());
				System.out.println("Incorrect policy");
		}

	}
```

It prints a list of the warnings, after ordering them by their line, and then the errors, already ordered because they are added to the list in sequential order as they are found. If there are no errors the policy is correct.
I assume the policy is correct also if there are warnings but there are no errors. The reason for this is that there are no hints of this situation in the paper *"AppPolicyModules: Mandatory Access Control for Third-Party Apps"*. Perhaps, future tests on devices will solve this doubt.

ERROR HANDLING

Errors are inserted in two lists in class Handler: one list is for violation of requirements (*errorList*) and the other one is for warnings (*warningList*).
These lists are arrays of *ErrorToken* class:

CLASS ERRORTOKEN

This class contains a message and the line which causes the error or the warning.
The message and the line are set in the constructor:

```
/** The message. */
String message;
/** The line. */
int line;

/**
 * Instantiates a new error token.
 *
 * @param msg the message of the error
 * @param line the line of the error
 */
public ErrorToken(String msg, int line){
	this.message=msg;
	this.line=line;
}
```

There is a method for getting the line:

```
/**
 * Gets the line of the error.
 *
 * @return the line
 */
public int getLine(){
	return line;
}
```

And a method toString:

```
public String toString(){
	return "line "+line+": "+message;
}
```

Finally, *errorList* contains already ordered by line errors (because errors are added to the list as they are found in the input document), whereas *warningList* keeps errors without order. For this reason class *ErrorToken* must implement the interface *Comparable<ErrorToken>* and override the *compareTo* method:

```
@Override
public int compareTo(ErrorToken et2) {
	return line-et2.getLine();
}
```

Thanks to this method it is possible to use the method
`Collections.sort(warningList);`
to order the list before printing it (method verifyPolicy() of class Handler).

CLASS REQUIREMENT1EXCEPTION

This class defines the exception thrown whenever the first requirement is broken.
It is a very simple class:

```
public class Requirement1Exception extends Exception {

	/** The Constant serialVersionUID. */
	private static final long serialVersionUID = -6281258440203635134L;

	/**
	 * Instantiates a new requirement1 exception.
	 *
	 * @param msg the message
	 */
	public Requirement1Exception(String msg){
		super(msg);
	}
}
```

As shown in chapter PARSER, it can be caught in *typeattribute* and *allow* rules.
In these catch clauses the error is added to the list:

```
catch [Requirement1Exception e]
{
   handler.addError(new ErrorToken(e.getMessage(),source.getLine()));
}
```

CLASS REQUIREMENT2EXCEPTION

This class is equivalent to the class *Requirement1Exception*, yet for the second requirement.
The code is almost the same:

```
public class Requirement2Exception extends Exception {

	/** The Constant serialVersionUID. */
	private static final long serialVersionUID = -7579993016650540969L;

	/**
	 * Instantiates a new requirement2 exception.
	 *
	 * @param msg the message
	 */
	public Requirement2Exception(String msg){
		super(msg);
	}
}
```

And as shown in chapter PARSER, it can be caught in *typebounds* rules.
In their catch clause the error is added to the list:

```
catch [Requirement2Exception e]
{
  handler.addError(new ErrorToken(e.getMessage(),t.getLine()));
}
```

MAIN CLASS

The main class is called Tester.
It has four fields:

```
String fileIn = ".\\resources\\appPolicyModule.txt";
GrammaticaLexer lexer;
GrammaticaParser parser;
CommonTokenStream tokens;
```

The input file, which is a constant, the objects for the lexer and the parser, and the token stream.

These are used in the main method as done for the main class of the first compiler:

```
lexer = new GrammaticaLexer(new ANTLRReaderStream(new FileReader(fileIn)));
tokens=new CommonTokenStream(lexer);
parser= new GrammaticaParser(tokens);
parser.start();
```

Executing these lines, the compiler reads the input file and prints on console if it is correct. If the input policy is not correct it prints the error list (and the warning list, too).

EXAMPLES

EXAMPLE 1: A CORRECT POLICY

This is the input appPolicyModule file:

```
module com.examples.youtubeapidemo 1.0;
require {
type untrusted_app;
attribute domain;
attribute appdomain;
attribute internet_domain;
}
type com.examples.youtubeapidemo_app;
typebounds com.examples.youtubeapidemo_app untrusted_app;
typeattribute com.examples.youtubeapidemo_app domain;
typeattribute com.examples.youtubeapidemo_app appdomain;
typeattribute com.examples.youtubeapidemo_app internet_domain;
```

as it can be seen this is a correct input file: the only declared type (`com.examples.youtubeapidemo_app`) has correct bounds (`untrusted_app`), and all the typeattribute rules use this type.

Parsing it, the first found token is "module", so the *header* rule is executed:

```
header: MODULE ID VERSION END ;
```

the second line begins with "require", so the *require* rule is executed:

```
require: OPEN_REQUIRE node_require* CLOSE_REQUIRE ;
```

the *node_require* rule recognizes the type and attribute rules before the closing shape parenthesis. In particular, the *type* rule adds *untrusted_app* to *avTypes* in *Handler* class.

Afterwards the body rule is executed. This recognizes the *type* rule and adds `com.examples.youtubeapidemo_app` to avTypes.
The *typebounds* rule let the handler check the second requirement. The boundary is "untrusted_app", so no exceptions are thrown.
Finally, all the typeattribute rules declare `com.examples.youtubeapidemo_app` as type, so neither the first requirement is broken.

After parsing, the two final java methods are executed. At this moment, the set *avTypes* contains [untrusted_app, com.examples.youtubeapidemo] and the set *typebounds* contains [com.examples.youtubeapidemo] therefore there are no warnings.

The output on console is

```
Correct policy
```

EXAMPLE 2: A WRONG POLICY

The input appPolicyModule file is the same of the previous paragraph. I appended two lines to make it rise errors:

```
module com.examples.youtubeapidemo 1.0;
require {
type untrusted_app;
attribute domain;
attribute appdomain;
attribute internet_domain;
}
type com.examples.youtubeapidemo_app;
typebounds com.examples.youtubeapidemo_app untrusted_app;
typeattribute com.examples.youtubeapidemo_app domain;
typeattribute com.examples.youtubeapidemo_app appdomain;
typeattribute com.examples.youtubeapidemo_app internet_domain;
allow com.examples.youtubeapidemo_app myapp;
typebounds myapp untrusted_app;
```

All the considerations made in the previous paragraph are still valid. The parser acts at the same way until the allow rule.
At the allow token, the proper rule is executed:

```
allow: ALLOW source=ID target=ID END
       {
          //check requirements 1:
          handler.checkRequirements1(source.getText(),target.getText());
       }
       ;
       catch [Requirement1Exception e]
       {
          handler.addError(new ErrorToken(e.getMessage(),source.getLine()));
       }
```

Method *checkRequirements1* finds out that *myapp* has never been declared (i.e. *avTypes* does not contain it) so it throws an exception. This is caught in the catch clause and an error is added to the list.
The last line does not raise errors because the bounds are “untrusted_app”.

Yet, when executing the method *checkWarning* after parsing, the sets contain:
avTypes=[untrusted_app, com.examples.youtubeapidemo]
typebounds=[com.examples.youtubeapidemo, myapp]

The method notices “myapp” is in *typebounds* set but it is not in the *avTypes* one. So a warning is added to the list.

Therefore, there are errors and the console shows this output:

```
WARNINGS:
line 14: The type myapp appears in a typebounds sentence without ever being defined
ERRORS:
line 13: (Req1) myapp undefined
Incorrect policy
```