

11-2013

Mobile SCALe: Rules and Analysis for Secure Java and Android Coding

Lujo Bauer

Carnegie Mellon University, lbauer@cmu.edu

Lori Flynn

Carnegie Mellon University, lflynn@cert.org

Limin Jia

Carnegie Mellon University

Will Klieber

Carnegie Mellon University, weklieber@cert.org

Fred Long

Aberystwyth University,

See next page for additional authors

Follow this and additional works at: <http://repository.cmu.edu/sei>



Part of the [Software Engineering Commons](#)

Authors

Lujo Bauer, Lori Flynn, Limin Jia, Will Klieber, Fred Long, Dean F. Sutherland, and David Svoboda

Mobile SCALE: Rules and Analysis for Secure Java and Android Coding

Lujo Bauer, Carnegie Mellon University, Department of Electrical and Computer Engineering
Lori Flynn, Software Engineering Institute
Limin Jia, Carnegie Mellon University, Department of Electrical and Computer Engineering
Will Klieber, Software Engineering Institute
Fred Long, Aberystwyth University, Department of Computer Science
Dean F. Sutherland, Software Engineering Institute
David Svoboda, Software Engineering Institute

November 2013

TECHNICAL REPORT
CMU/SEI-2013-TR-015
ESC-TR-2013-015

CERT® Division

<http://www.sei.cmu.edu>



Copyright 2013 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution except as restricted below.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

Carnegie Mellon® and CERT® are registered marks of Carnegie Mellon University.

DM-0000726

Table of Contents

Acknowledgments	vii
Abstract	ix
1 Introduction	1
2 The Java Coding Guidelines Book	2
3 Android Secure Coding Rules	3
4 Android App Analysis Tool	5
4.1 Design and Implementation	5
4.2 Limitations and Future Work	7
5 Summary and Future Work	8
References	9

List of Figures

Figure 1:	Java Coding Guidelines Book Cover	2
Figure 2:	Introductory Section of an Android Rule	3

List of Tables

Table 1:	Count of Java Guidelines, as Applicable to Android	2
Table 2:	Guidelines Applicability Definitions	2
Table 3:	Count of Java Rules, as Applicable to Android	4
Table 4:	Rule Applicability Definitions	4

Acknowledgments

We are grateful for the line funding award from the Software Engineering Institute, which enabled this work to be done. Many thanks go to JPCERT for the major contributions of Masaki Kubo, Hiroshi Kumagai, and Yozo Toda toward Android rule creation and analysis of applicability of rules and guidelines. We also thank professional editor Carol Lallier for her improvements to this paper.

Abstract

This report describes Android secure coding rules, guidelines, and static analysis that were developed as part of the Mobile Source Code Analysis Laboratory (SCALE) project. The project aims to create a set of rules that can be checked (and potentially enforced) and to develop checkers for these rules. These efforts are intended to increase confidence in continued safe and secure operation of mobile devices and the networks on which they operate. The focus for this phase of the project is the Android platform for mobile devices. Work described in this report involved three activities: (1) preparing the Java Coding Guidelines book for publication, (2) developing Android secure coding rules for the Android section of the CERT Oracle Secure Coding Standard for Java wiki, and (3) developing software that does static analysis of a set of Android apps for data flows between them so that security leaks can be detected.

1 Introduction

This report describes Android secure coding rules, guidelines, and static analysis that were developed as part of the Mobile SCALe project. The project aims to create a set of rules that can be checked (and potentially enforced) and to develop checkers for these rules. These efforts are intended to increase confidence in continued safe and secure operation of mobile devices and the networks on which they operate. The current phase of the project focuses on the Android platform for mobile devices.

Mobile SCALe project work discussed in this report is a continuation of related work done by the CERT® Secure Coding Initiative, part of Carnegie Mellon University's Software Engineering Institute, particularly the Initiative's previous work on secure coding for the Java language. That work includes the development of the CERT Oracle Secure Coding Standard for Java wiki and publication of the eponymous book [Long 2011]. The CERT Oracle Secure Coding Standard for Java contains a set of normative rules, meaning that it should be possible to determine whether a piece of code conforms to these rules.

Recommended practices, even though they are not normative, can help with the production of more secure and reliable code. Section 2 of this report tells about new work that developed these recommended practices into a set of *guidelines* for publication as a second book.

An Android extension of the original CERT Oracle Secure Coding Standard for Java was needed to give better guidance for secure coding of Java applications (apps) for Android mobile devices. Section 3 of the report details the new Android section that we added to the CERT Oracle Secure Coding Standard for Java wiki.

With the development of secure coding rules for Android, the Source Code Analysis Laboratory (SCALe) tools needed to be extended so that they could cope with the Android architecture. Section 4 of this report describes a new static analysis tool we developed for Android apps.

Section 5 of this report summarizes the latest work completed and tells about future work planned.

® CERT® is a registered mark owned by Carnegie Mellon University.

2 The Java Coding Guidelines Book

Some members of the Mobile SCALE project team helped prepare the book *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* [Long 2013] for publication (Figure 1). The work for the book was originally developed on a private wiki. The wiki pages were then exported as documents in a format used for offline prepublication work, edited, and sent to the publisher in May 2013. Proofs were received from the publisher and corrections and edits made to the proofs, which were then returned to the publisher. Final proofs were received from the publisher and further minor corrections made and submitted in July. The book was published on September 9, 2013.

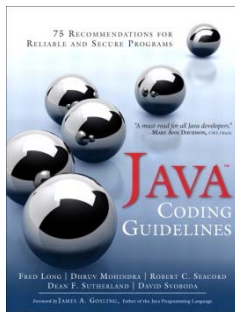


Figure 1: Java Coding Guidelines Book Cover

The Java language is the basis for the Android platform used in many mobile devices. However, the Android architecture differs somewhat from that used in normal Java applications, and the security model differs a lot. Not all of the newly developed Java coding guidelines apply to Android. Analysis was done to determine the applicability of each of the Java coding guidelines to Android. The book's appendix describes the applicability of the guidelines in the book to developing Java apps for the Android platform, as summarized in Table 1 using definitions listed in Table 2.

Table 1: Count of Java Guidelines, as Applicable to Android

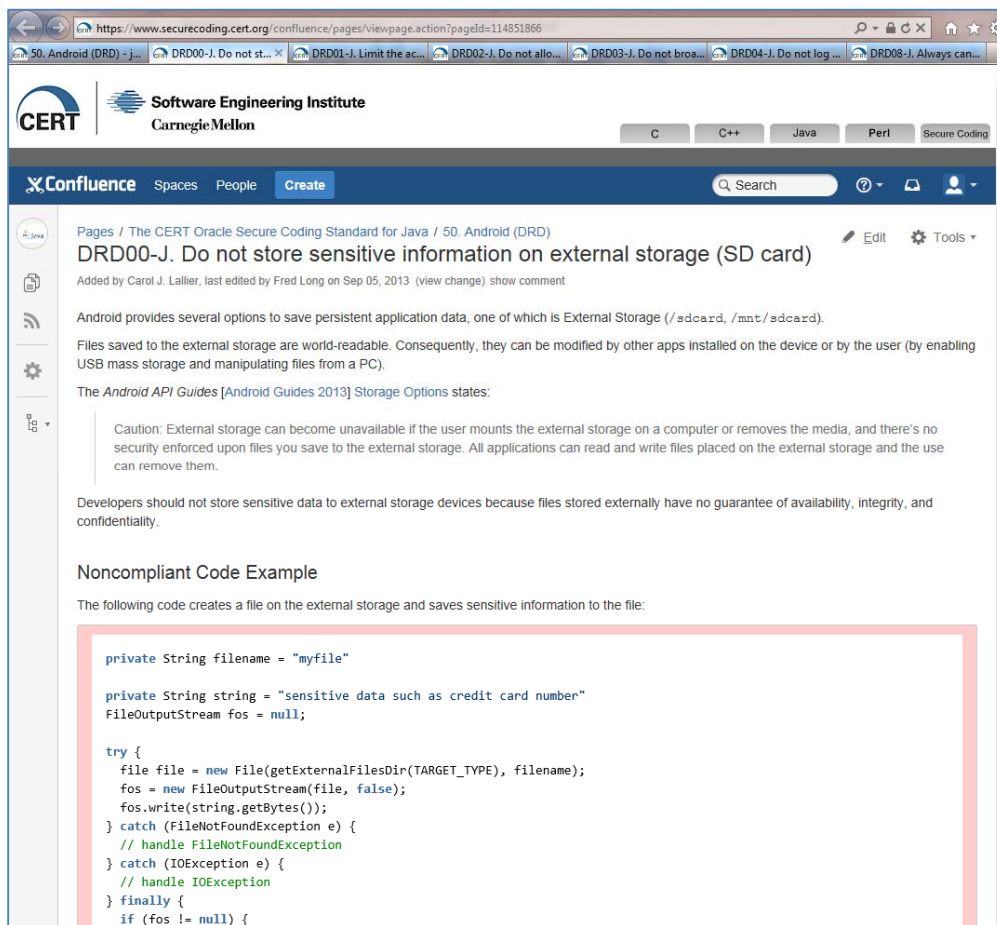
	Android-only	Definitely Applicable	Probably Applicable	Applicable in Principle	Not Applicable	Unknown (not judged <i>Probably Applicable</i>)
Guidelines	0	58	0	9	6	2

Table 2: Guidelines Applicability Definitions

Term	Definition
Android-only	The guideline is relevant only to Android platforms
Definitely applicable	The guideline can be applied to general Java platforms, including Android
Probably applicable	One reviewer found the guideline to be applicable to the Android platform, but we are waiting for applicability to be verified
Applicable in principle	The guideline can be applied to Android, but the code examples shown in the guideline are not relevant to Android
Not applicable	The guideline cannot be applied to Android platforms

3 Android Secure Coding Rules

We added an Android section to the CERT Oracle Secure Coding Standard for Java wiki. The rules in this section are specific to Android apps. However, they have the same structure as rules in the other sections (see Figure 2). That is, each rule contains an introduction followed by one or more sets of noncompliant code examples, illustrating how violation of the rule may lead to vulnerability in the code, and a compliant solution, showing how to avoid the vulnerability. These coding examples are followed by a risk assessment that indicates the severity of any vulnerability likely to arise from violating the rule, the likelihood that such vulnerability could be exploited, and the cost of remediation if the violation of the rule is found in existing code. Each of these three attributes is given a score in the range 1 to 3. The three scores are then multiplied to give a *priority* for ranking the rule. The rule is also assigned a *level* indicating the rule's criticality for secure coding, with level 1 being the most critical and level 3 being the least critical. The risk assessment may be followed by sections on automated detection of the rule and on related vulnerabilities and related guidelines. Each rule has a bibliography providing links to further information.



The screenshot shows a web browser displaying the CERT Oracle Secure Coding Standard for Java wiki page for rule DRD00-J. The page title is "DRD00-J. Do not store sensitive information on external storage (SD card)". The page is part of the "The CERT Oracle Secure Coding Standard for Java / 50. Android (DRD)" section. The page content includes an introduction about Android's external storage options, a caution about security, and a noncompliant code example. The code example shows a Java snippet that writes sensitive data to a file on external storage.

```
private String filename = "myfile"

private String string = "sensitive data such as credit card number"
FileOutputStream fos = null;

try {
    file file = new File(getExternalFilesDir(TARGET_TYPE), filename);
    fos = new FileOutputStream(file, false);
    fos.write(string.getBytes());
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
} finally {
    if (fos != null) {
```

Figure 2: Introductory Section of an Android Rule

The Android rules were developed in collaboration with members of the Japan Computer Emergency Response Team Coordination Center (JPCERT). JPCERT was the first Computer Security Incident Response Team (CSIRT) established in Japan. It acts as a coordinating center for other CSIRTs in the Japanese community. Members of JPCERT have been very active in identifying vulnerabilities in Android code.

The Android section of the CERT Oracle Secure Coding Standard for Java wiki currently contains nine rules. Six of them were provided by members of JPCERT, and they have been formatted and edited by members of the Mobile SCALE team. The remaining three were suggested by a member of the Mobile SCALE team, but they are awaiting completion.

Two of the nine rules are Android-specific instances of more general Java rules, one concerning the logging of sensitive data and the other concerning canonicalization of file path names.

Four of the remaining seven rules are concerned with the handling of sensitive data by Android apps. They discuss aspects of Android programming that could lead unwary programmers to release sensitive data by misusing features of the Android architecture.

The remaining three rules concern very specific aspects of Android programming that require particular care to avoid security problems: granting uniform resource identifier (URI) permissions, dealing with malicious intents, and protecting exported services.

All except one of the nine rules were allocated level 1 in the three-level priority model used by the CERT secure coding standards. The remaining rule was allocated level 2.

Also as part of this activity within the Android Project, an analysis has been made of the CERT Oracle Secure Coding Standard for Java rules that are applicable to Android coding, and a page was added to the Android section of the wiki showing the outcome of that analysis. Table 3 summarizes the count of Java rules, with applicability to Android, as defined in Table 4.

Table 3: Count of Java Rules, as Applicable to Android

	Android-only	Definitely Applicable	Probably Applicable	Applicable in Principle	Not Applicable	Unknown (not judged <i>Probably Applicable</i>)
Rules	9	79	67	10	11	0

Table 4: Rule Applicability Definitions

Term	Definition
Android-only	The rule is relevant only to Android platforms
Definitely applicable	The rule can be applied to general Java platforms, including Android
Probably applicable	One reviewer found the rule to be applicable to the Android platform, but we are waiting for applicability to be verified
Applicable in principle	The rule can be applied to Android, but the code examples shown in the rule are not relevant to Android
Not applicable	The rule cannot be applied to Android platforms

4 Android App Analysis Tool

A main type of threat to data and application security on the Android platform is privilege escalation and illicit information-flow vulnerabilities caused by well-intentioned but buggy apps. These are caused by apps that both (1) have legitimate access to sensitive resources (e.g., the phone's microphone) or output channels (e.g., the ability to use the internet) and (2) export inadequately protected (e.g., through the use of permissions or stringent input validation) interfaces to other applications. The impact of privilege escalation is heightened by many apps requesting more permissions than they actually use [Felt 2011].

A malicious app can misuse another application's privileges or leak information. If a malicious application *A* doesn't have permission *P* but another application *B* does have permission *P*, then *A* might be able to trick *B* (or collude with *B*) to perform an operation requiring permission *P* on *A*'s behalf. Additionally, *A* might be able to receive sensitive information from *B* that *A* shouldn't have access to. For instance, if *B* has the `READ_CONTACTS` permission, and *B* is allowed to call *A*, and *A* does not have the `READ_CONTACTS` permission, then a communication flow from *B* to *A* might allow the leakage of sensitive address-book information.

To help address such threats, we designed a tool that analyzes potential communication between apps. Our static analyzer builds on previous research [Chin 2011, Fuchs 2009]. We focus exclusively on the sending and receiving of *intents*, which comprise the primary system of inter-app communication in Android. Intents are also used for *intra*-app communication between different components of a single app. Unfortunately, it is easy for a developer to mistakenly make app interfaces public when they should be private, allowing malicious apps to hijack or eavesdrop on apps that have access to sensitive information or resources.

Our static analysis tool analyzes each app individually to (1) find likely violations of secure coding rules, (2) produce a list of what kinds of intents the app registers receive, and (3) produce a list of program sites (source code or bytecode locations) that send intents, along with the action string and target class (if known). Given a set of apps, the information about individual apps can be composed to find the possible cross-application information flows within the set. For example, a security-conscious organization that distributes Android devices to its employees might test the set of apps that are preinstalled or approved for installation on its devices. By combining (1) the cross-app information flows, (2) the permissions required by each app, and (3) the sensitive information available to each app, we can detect possible information leaks and other communication-related vulnerabilities within the set of apps. The permissions required by each app are easily obtained from the app's manifest. However, there is no automatic way to determine what sensitive information will be available to each app; this requires input from the developers and/or users of the app.

4.1 Design and Implementation

We used the Soot Java analysis framework [Vallée-Rai 1999, Sable 2012, Einarsson 2008]. Soot provides a suite of static analysis facilities for inspecting Java programs. These facilities enable us to identify the method calls that send Android intents. Where possible, we identified the action string associated with the intent and the target of the intent in the case of an explicit intent. Even

though our analysis is relatively simple, we were able to precisely identify the action string and target in most cases in the apps that we analyzed.

The goal of our static analysis is, for each program site that sends an intent, to identify the intent's possible pairs of `action_string` and `target_class`. Most such program sites have only a single possible action string and target class, so we focus on this case. If more than one action string or target class is possible, we report it as *unknown*.

We represent the program state as a pair of an environment (which maps variables to memory locations) and a store (which identifies contents of memory locations):

```
prog_state: pair of (env, store)
env: mapping of var_name to mem_loc
store: mapping of mem_loc to obj_info
obj_info: mapping of field_name to value
```

If we do not know the memory location to which a variable points, we do not include the variable in the environment mapping. The store maps each memory location to an `obj_info` mapping, which describes the contents of the object stored at the memory location. In particular, for each field of the object, `obj_info` maps the field name to the value of the field. If the value of a field is unknown, the field is not included in the mapping. Our analysis is concerned with only one type of mutable object (`Intent`) and two of its fields: the action string and the target class.

We track three types of objects: strings, class objects, and intents. For strings (which are immutable in Java), we encode the value of the string in the name of the memory location, so we do not need an entry in the store. This practice loses precision (because we cannot distinguish between identical strings that are at different memory locations), but for our purposes, it does not matter. Class objects are handled in a similar way to strings: the name of the class is encoded in the name of the memory location.

For intents, the name of the memory location is the program site where the intent object was allocated. For example, if line 100 of `Foo.java` consists of the statement `x = new Intent()`, then the name of the memory location might be "`Foo.java:100.1`." This naming scheme implies that we cannot distinguish between two intents that were both allocated at the same program site. Typically, however, only the most recently allocated intent is of interest. In addition, we need to allow strong updates to the most recently allocated intent. So, whenever a new intent is allocated, we first forget about all intents that were previously allocated at the same allocation site. That is, we remove any environment mappings to the memory location, and we remove the memory location from the store.

Our analysis is a forward-flow analysis that uses the fixed-point framework of Soot [Vallée-Rai 1999]. At program points that have more than one inflow (e.g., the top of a loop body), the program states of the two inflows are merged as follows: For each variable name `var`, if `var` is mapped to the same memory location by both inflows, then the merged environment will also map `var` to the same memory cell. If the two inflows differ in this regard, then `var` will be omitted in the merged environment mapping. Similarly, for each memory location `loc` and each field name `field`, if `store_in1[loc][field] = store_in2[loc][field]`, then

`store_merged[loc][field]` will have the same value as in the two inflows, and otherwise `field` will not be mapped by `store_merged[loc]`.

A common mistake in Android apps (and a violation of secure coding rules) is to use implicit intents for intra-app communication. Such intents should be explicit to prevent malicious apps from accessing the communication interface. There is no definitive way to determine whether an implicit intent was really intended for intra-app communication or whether the developer contemplated the intent being sent to another app. However, most inter-app intents use a standard, predefined action string (starting with “`android.intent.action.`”). As a heuristic, if an implicit intent is sent with a nonstandard action string, we flag the intent as a possible violation.

4.2 Limitations and Future Work

In its current state, the tool has several limitations that can be addressed in the future. First, the approach to detecting information flows between applications is coarse grained, and potential flows flagged by the tool need not appear in any execution (e.g., a detected method call for inducing cross-application communication is not reachable). Finer grained static analysis could be used to reduce the incidence of such false positives.

Second, the tool currently focuses exclusively on Android intents, the main method of cross-application communication. Applications have other communication methods at their disposal, such as (1) directly querying `ContentProviders`, (2) reading from and writing to an SD card, and (3) using native code and communication channels implemented by the underlying operating system (e.g., sockets). Such means are currently beyond the scope of our analysis. Some of them could easily be accounted for by small extensions to the current analysis. Others would require more substantial effort to detect, and some (such as the use of covert channels) may not be feasible to detect automatically.

5 Summary and Future Work

This report describes Android secure coding rules, guidelines, and static analysis that were developed as part of the Mobile SCALE project. We prepared *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* [Long 2013] for publication; the book includes an appendix describing the applicability of guidelines to developing Java apps for the Android platform, and it was published September 9, 2013. We added an Android section to the CERT Oracle Secure Coding Standard for Java wiki, with new Android-only rules plus charts showing analyzed applicability of previously existing Java secure coding rules and guidelines. The new Android rules require further work to complete. The applicability analysis of previously developed rules and guidelines also requires further work to complete, although significant progress has been made: 137 were found definitely applicable, 17 were found definitely not applicable, and the remaining rules/guidelines have undergone initial review for applicability. We designed and implemented a tool that takes as input a set of Android applications and outputs a list of the cross-application information flows that can potentially exist within the set. However, information flows analyzed are coarse-grained and limited to intents. The analysis is sound (in that it will find all potential information flows of the kind that it looks for), so it is necessarily imprecise (i.e., a flagged potential information flow might never appear in any actual execution).

Currently, our static analysis tool flags all communication flows, including both legitimate and malicious flows, since the chief distinction between the two can be in the users' expectation (e.g., one app is trusted to receive sensitive information, while another is not equally trusted even though it behaves similarly as far as static analysis is concerned). Such false positives cannot be remedied through more sophisticated analyses. They can, however, largely be ruled out using input from the developers and users of the apps. An interesting problem is how to enable such input to be provided in a reasonably feasible manner. For example, work by Jia and colleagues [Jia 2013] develops such a system for labeling apps and their components to dynamically enforce information-flow properties.

Future work includes further development of a secure coding standard that can be used to develop Java Android apps and a version of SCALE that can be used to analyze these applications and determine if they conform to the standard. This effort will include completion of the analysis of applicability to Android of existing Java rules and guidelines, as well as creation of new ones. We also plan to do finer grained static and dynamic analyses of information flows between applications, to extend analyses to additional types of information flows, and possibly to instrument apps with runtime checks.

References

URLs are valid as of the publication date of this document.

[Chin 2011]

Chin, E.; Felt, A.; Greenwood, K.; & Wagner, D. “Analyzing Inter-Application Communication in Android,” 239–252. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. Bethesda, MD, June 2011. ACM, 2011.

[Einarsson 2008]

Einarsson, Á. & Nielsen, J. D. *A Survivor’s Guide to Java Program Analysis with Soot*. BRICS, Department of Computer Science, University of Aarhus, Denmark, Version 1.1, 2008. <http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf>

[Felt 2011]

Felt, A.; Chin, E.; Hanna, S.; Song, D.; & Wagner, D. “Android Permissions Demystified,” 627–638. *Proceedings of the 18th ACM Conference on Computer and Communications Security*. Chicago, IL, Oct. 2011. ACM, 2011.

[Fuchs 2009]

Fuchs, A.; Chaudhuri, A.; & Foster, J. *SCanDroid: Automated Security Certification of Android Applications* (Technical Report). University of Maryland, 2009. <http://www.cs.umd.edu/~avik/projects/scandroidascaa/paper.pdf>

[Jia 2013]

Jia, L.; Aljuraidan, J.; Fragkaki, E.; Bauer, L.; Stroucken, M.; Fukushima, K.; Kiyomoto, S.; & Miyake, Y. “Run-Time Enforcement of Information-Flow Properties on Android (extended abstract),” 775–792. *18th European Symposium on Research in Computer Security (ESORICS 2013)*. Egham, U.K., Sep. 2013. Springer-Verlag, 2013.

[Long 2011]

Long, F.; Mohindra, D.; Seacord, R. C.; Sutherland, D. F.; & Svoboda, D. *The CERT Oracle Secure Coding Standard for Java* (SEI Series in Software Engineering). Addison-Wesley Professional, 2011 (ISBN 978-0-321-80395-5).

[Long 2013]

Long, F.; Mohindra, D.; Seacord, R. C.; Sutherland, D. F.; & Svoboda, D. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (SEI Series in Software Engineering). Addison-Wesley Professional, 2013 (ISBN 978-0-321-93315-7).

[Sable 2012]

Sable Research Group. *Soot: A Java Optimization Framework*, v. 2.5.0. School of Computer Science, McGill University, 2012. <http://www.sable.mcgill.ca/soot/>

[Vallée-Rai 1999]

Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P.; & Sundaresan, V. “Soot—A Java Bytecode Optimization Framework.” *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 1999.

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE November 2013		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Mobile SCALE: Rules and Analysis for Secure Java and Android Coding			5. FUNDING NUMBERS FA8721-05-C-0003	
6. AUTHOR(S) Lujo Bauer, Lori Flynn, Limin Jia, Will Klieber, Fred Long, Dean F. Sutherland, and David Svoboda				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2013-TR-015	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2013-015	
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE	
13. ABSTRACT (MAXIMUM 200 WORDS) This report describes Android secure coding rules, guidelines, and static analysis that were developed as part of the Mobile Source Code Analysis Laboratory (SCALE) project. The project aims to create a set of rules that can be checked (and potentially enforced) and to develop checkers for these rules. These efforts are intended to increase confidence in continued safe and secure operation of mobile devices and the networks on which they operate. The focus for this phase of the project is the Android platform for mobile devices. Work described in this report involved three activities: (1) preparing the Java Coding Guidelines book for publication, (2) developing Android secure coding rules for the Android section of the CERT Oracle Secure Coding Standard for Java wiki, and (3) developing software that does static analysis of a set of Android apps for data flows between them so that security leaks can be detected.				
14. SUBJECT TERMS Android, Java, secure coding, rules, guidelines, static analysis, app analysis, data flows, security leaks, Mobile SCALE, SCALE, secure coding standard			15. NUMBER OF PAGES 23	
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	