

Applying Assurance Techniques to a Mobile Phone Application

Padmanabhan Krishnan
Sergej Hafner
Andreas Zeiser

Centre for Software Assurance
Bond University
Gold Coast, QLD 4229, Australia
Email: pkrishna@staff.bond.edu.au

Abstract

As users download applications to their mobile phones, security is a critical issue. In this paper we present a process for the security assurance of applications. It uses well identified vulnerability databases and application development guidelines to identify potential security issues. The identified issues are then validated using black-box testing, unit testing code inspection and static analysis. This process is illustrated using an application for the Android platform.

1. Motivation

Modern mobile phones are not very different from desktop computers in terms of the range of applications they can support. Various development platforms allow anyone to develop an application for the mobile device. However this raises the issue of potentially dangerous (perhaps malicious) applications. As noted [14] viruses for mobile phones are a major issue as mobile phones do not have standard virus checkers. These viruses typically exploit multimedia messages, instant messaging, IP connections and web-browsing. The damage caused can be as simple as draining the battery to actual phone-jacking where the malicious software can make any call.

Typical commercial platforms require some form of “certification” for all applications. For instance, Symbian and Apple require the application level code to be signed by a trusted authority. The trusted authority will perform the necessary checks to ensure that the applications do not have any dangerous behaviour. Android, which is the Open Handset Alliance’s platform, requires the application to declare the resources

it will use in its execution. When the application is installed, the user can examine the list of resources and then decide on whether to install the application. The underlying operating system will prevent any unauthorised access to resources. A detailed study of the security features in Android is described in [11]. A detailed comparison between various environments is given in [5].

Although the Android platform provides a sandbox, Symbian and Apple use certification to prevent malicious impact on other applications, it is important to develop an assurance framework that developers (or certifiers) can use to determine the application satisfies the security requirements. It is this issue that we address in this paper.

The importance of certification has been argued [13]. However, they also discuss the limitations of current approaches (including certifying the actual code, limitations of formal approaches, and the need for domain specific techniques).

There are many published guidelines that pertain to the security at the application level. The Build Security in Software Assurance Initiative (BSI <https://buldsecurityin.us-cert.gov/bsi/home.html>) provides a general framework that addresses all aspects of software development. Repositories like the CWE (<http://cwe.mitre.org>) provide a set of identified vulnerabilities in web applications. They also discuss mitigation strategies. OWASP and OSVDB (<http://osvdb.org/>) are similar and provide reports for web based systems and open source system respectively. The Common Attack Pattern Enumeration and Classification (CAPEC) has been defined to assist in enhancing security throughout the software development life-cycle, and to support the needs of developers, testers and educators. This is sponsored by the Department of

Homeland Security as part of the Software Assurance strategic initiative of the National Cyber Security Division. The objective of this effort is to provide a publicly available catalogue of attack patterns along with a comprehensive schema and classification taxonomy. As mobile devices have access to the Internet, on device databases etc., the usual application level threats identified in CAPEC such as SQL injection, cross-site scripting etc. have to be considered.

There are also language specific guidelines. For example the Cert Oracle Secure Coding standard has been published for Java <https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java>.

These can be further specialised for specific platforms. The Android developer's guide (<http://developer.android.com/guide/index.html>) provides a framework for constructing an application for the Android platform while organisations such as iSEC Partners provide guidelines for application security for Android applications.

As with any assurance process a systematic testing procedure is necessary. It is also important to document the approach used. In this paper we report our experience of carrying out an assurance process for an application on the Android platform. We are not the authors of the system so our exercise can be viewed as a process of independent assurance. We expect such assurance processes to replace simple certification processes especially for applications that really need to be trusted. Before we describe the details of our experience, we review some of the relevant assurance techniques in the next section. In Section 3 the application and the key steps of the assurance process are described.

2. Background Work

While we have presented some of the practical guidelines used for assurance, in this section we review some of the related research. Square [8] uses a risk based process for the identification of security properties which are related to business goals. Their principal focus is on relating these issues back to the requirements. The methodology has nine key steps. Their main contribution is a process to include security considerations at the requirements phase in the software development life-cycle. CLASP [18] describes a process where the various security related resources are identified. It addresses all the phases of the software development life cycle (e.g., project management, pre-release testing). Attack surfaces [6] are one way of identifying potential vulnerabilities in an application. This is related to

the identification of resources in CLASP. Manadhata et al [7] describe a metric for measuring an attack surface. As the paper focuses on the FTP daemon it uses entry points, exit points, channels and untrusted data in its measurement. They identify 14 attack classes in Linux. They also use the damage potential to give a weightage to the various resources.

Since we are interested in application security, management standards such as ISO17799 are not directly relevant. The software engineering processes we have identified earlier or standards such as NASA-STD-8739.8 are more relevant. As certification of large systems beyond EAL3 using the common criteria is very expensive [12], the lightweight certification process for mobile phones described by Enck et al [4] is more suitable. Although the certification they describe is applied only at the install stage the process can be generalised to address many classes of security vulnerabilities.

In this paper we focus testing against known vulnerabilities. Although we try to be process agnostic, we adapt the key aspects of the CLASP process. This is mainly because it is directly relevant to our work in identifying the resources and their interactions. The identification of resources is important to describe the attack surface [6]. We have extended the resources identified in [7] as part of the attack surface to suit our application domain. Enck et al [4] suggest the process that identifies the assets, the requirements, security goals and threats, develop the asset's security requirements and finally determine the security mechanism's limitations. The key difference is that Enck et al [4] define rules that can be used to determine quickly if an application is malware while our process is a detailed technique to assure a given application. So we do not use any of rules used in their certification process.

In this paper we rely on published repositories to prioritise the potential vulnerabilities. So we have not relied on any business case of our application to calculate the actual risk to operations.

3. Case Study

The application we consider is one that controls the communications of the user. That is, the applications does not allow anyone to make or receive a call to/from any user. This application is representative of other applications (e.g., mobile phone based banking) which have stringent security concerns.

There are two subsystems in this application. The first aspect is the security manager while the second aspect is the secure dialer. Only the administrator invoke the security manager. The role of the security manager is to manage the policy for accessing the phone set-

ting related to sending/receiving SMSs and calls. This is done via white-lists associated with each user. Each user has a list of permitted numbers that can be called or messaged. Similarly there is a list of permitted numbers from whom calls or messages can be received. The administrator can create or delete an account, change the password of all users. The administrator can also add or delete numbers from a user’s white-list. The secure dialer is a simple application which allows the user to call or send message to any number. If a number is not on the white-list the operation terminates abnormally. Similarly, if an incoming call is not on the white-list the call is not completed.

In this article we focus mainly on the security manager but also the service aspects of the secure dialer. A key observation is that almost all misuse of the dialer requires some tampering with the information managed by the security manager.

The implementation has databases (based on SQLite provided by Android) to store the policy for the users, the white-lists for each user and the common information. The implementation also has various files for authentication of the administrator and users. Like all Android applications the system has the manifest file which describes the resources used.

The assurance process we will use is:

- Use the design to understand the system and assurance process
- Identify the key functionality and the resources necessary in the implementation
- Identify the attack surface based on the above step (but we do not attach any numeric measure to this)
- Using the published guidelines map attacks to the identified surface
- Validate (via testing, verification or code-inspection) to determine if the system is resilient to the attacks.

3.1. Architecture

The architecture of the application is shown in Figure 1. This architecture has been derived from the source code of the system. It shows some of the high level operations and the resources that they are likely to use. In this design the main resources are databases and files. The key database items per user are `whitelist.db`, `ssid.db` and `userpolicy.db`. There is another file called `admin.dat` where the user name passwords are stored. There is also the standard `AndroidManifest.xml` where the resources used

by the application are specified. In the assurance process we divide the functionality into those that can be used in an attack.

We consider the following behaviours that are potentially unsafe. These have been identified via traces that can access the resources.

- Any user can attempt to register or login as administrator (Login).
- Only an authenticated administrator can create or delete an account (Create/Delete).
- The administrator can change a user’s password (Change Password)
- Only an authenticated administrator can change the policy setting of any user (Change Policy)
- Only an authenticated administrator can launch the secure dialer via the security manager.

Many of these occur when the credentials associated with the administrator have been compromised. There are also other ways of updating information (e.g., via SQL injection attacks) that are considered.

We use the taxonomy of threats in the Android framework [11] to identify the unsafe behaviours. As [11] recommends, application certification is an ideal countermeasure against malicious applications. Specifically, the abuse of costly services can occur when such a service is added to `whitelist.db` without proper authentication. Similarly, the blocking of communication can occur when a legitimate user is removed from `whitelist.db`. Based on the above behaviour we identify the relevant Java classes that could be used in the attack. A summary of the functionality, its implementation and the resources used is presented in Table 1.

3.2. Modeling

The first step we undertake is to model the Android framework where the main purpose is to understand the validation processes that might be necessary. Here we use ideas from model-based testing techniques [17] and vulnerability testing using models [10]. The specification has three components, the generic install process the generic execution model and the behavioural aspects of our system. The install process was not directly relevant as we do not have different applications with the same signature. In Android an application can use another application’s resource if the two applications have the same signature.

We use reachability analyses and model-checking to either verify the required properties or generate potential test sequences. We expressed our model using SAL

Functionality	Class(es)	Resource
Login	SecurityMan.java Login.java	admin.dat
Create/Delete	UserPolicy.java SecuritySharedData.java	userpolicy.db ssd.db
Change Password	PasswordEditor.java	ssd.db admin.dat
Change Policy	ManagePolicy.java SecuritySharedData.java	whitelist.db ssd.db

Table 1. Mapping of Functionality to Resources

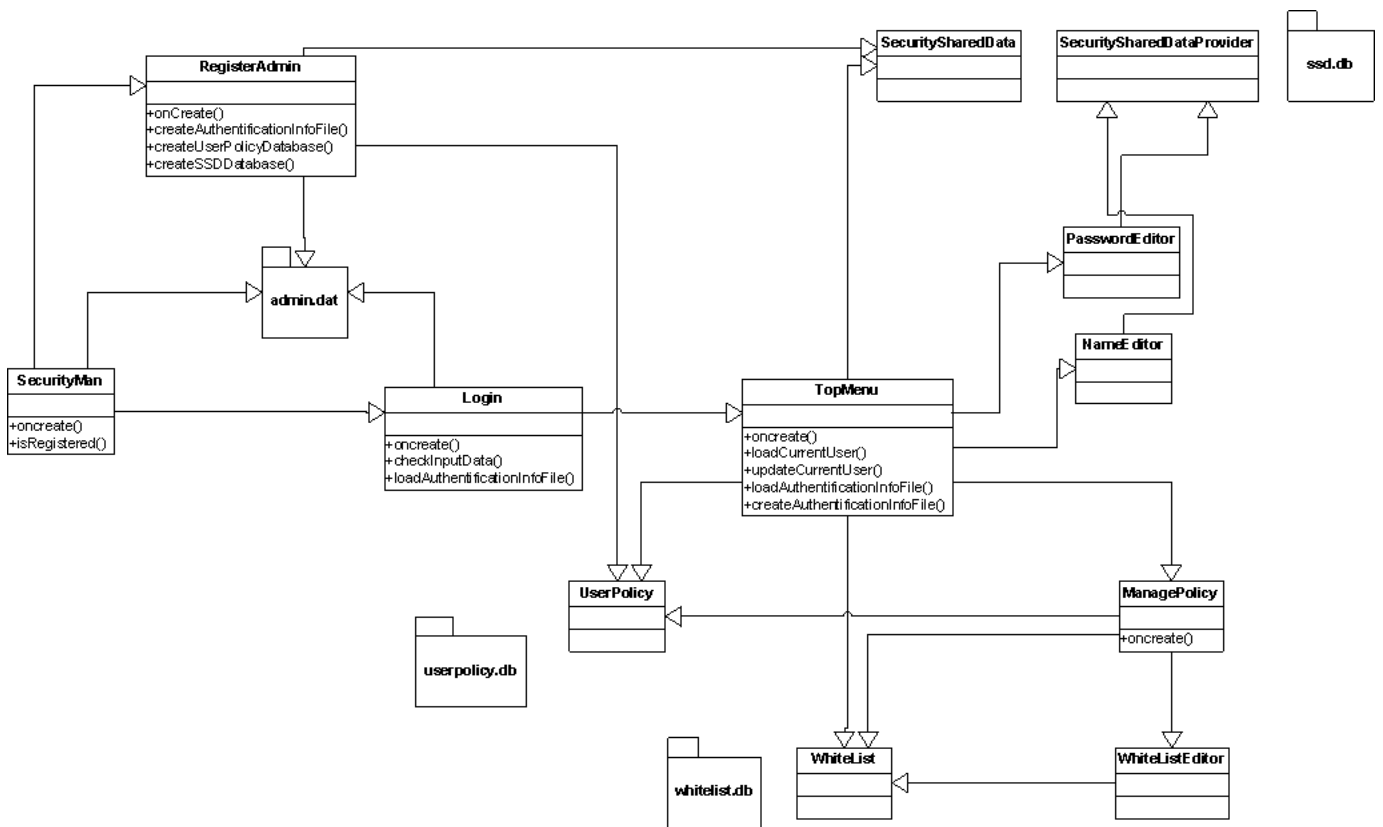


Figure 1. Architecture of System

[3] which allowed us to use `sal-atg` to automatically generate test sequences.

The formal model has characteristics to the Coq model related to Android [15, 16] but we did not conduct any formal proofs on our model. As our model was fairly abstract, the test sequences generated are not directly applicable at the system level. The adaptor necessary to facilitate this was too complex. Thus model-based testing did not seem to be directly usable

here.

However the reachability analyses were useful in identifying behaviours (paths in the specification) that can access a resource. This helped in the identification of the attack surface which is describe in Section 3.3.

3.3. Attack Surface and Vulnerabilities

In this section we describe the mapping of the behaviours to the Java classes and hence the resources. We also link this to the various published security guidelines. This then describes the attack surface we have identified. This can be done in two ways. The first is to identify the functionality and assign all the guidelines that are applicable. However, this resulted in numerous items being assigned to the functionality. The second approach is to identify a guideline and list all the functionalities against it. As we are using the guidelines to identify the top few vulnerabilities we adopt the second approach. A subset of our attack surface map is presented in Table 2.

However items such as CWE-805 are not directly applicable as CWE also specifies that Java application platforms do not have buffer overflows. This is because in the Android case the Dalvik VM handles memory management for the developer. Thus we can remove certain vulnerabilities from our initial list.

There are also other general coding guidelines that apply to the entire code and not just the high level functions. Some of these are shown in Table 3.

The resources were then classified into the following to assist in the validation process.

- Databases: where the attacker changes the information to either mount a denial of service attack (e.g., remove numbers from the white-lists) or abuse costly services (e.g., add numbers to the white-lists).
- Other file access: in this case the file `admin.dat`
- Cryptography: the algorithms used in the code.
- Interaction between systems: this is Android specific where the file `AndroidManifest.xml` is used
- Other programming related issues.

3.4. Validation of Potential Attack

The next step in the assurance process is to check if the application is actually vulnerable to any of the potential attacks identified earlier. We use code inspection, static analysis and testing to determine if the system is vulnerable. Each technique is suitable for checking a class of potential vulnerabilities. We present a subset of our validation steps and findings in Tables 4, 5 and 6. This is based on commercial assurance reports [2]. While the summary descriptions are self-explanatory we explain some of the details.

Initial code inspection was used to determine the actual code fragment that corresponds to the credentials. The code inspection revealed that only the pass-

word was encrypted. In order to test this we used the Android Dalvik Debug Monitor Server. This is part of the Eclipse plugin and enables the tester to check the files that are actually on the device. This feature was combined with a unit test written in Robotium (<http://code.google.com/p/robotium/>). The unit test registered a new administrator with username `abc` and password `def`. After that the file `admin.dat` was pulled from the device. Here it was discovered that the value `abc` was not encrypted. This vulnerability also led us to check (although not shown in this paper) if the system was susceptible to a brute-force attack (CAPEC-112). That is, using the plain text name of the user the attacker can try various passwords. However the application locked out the user after three consecutive unsuccessful login attempts. This was achieved via a black-box test sequence.

It took us about 200 person hours to conduct the full assurance process. Of this about 100 hours was spent in understanding the details of the various guidelines. In all around 30 guidelines were used in the assurance process. For each guideline, on an average about three functionalities had to be examined. For each of these examinations an average of three validation activities were required. Thus in total 270 separate validation activities were conducted to assure the security manager of the secure dialer system. A similar number of validation activities will be required for the behaviours of the user using the dialer.

4. Conclusion

In this paper we have presented a case study of a security assurance process for an application for the Android platform. This assurance process does not examine the vulnerabilities in the Android's security enforcement environment. We have adapted a standard process to fit our example. The key finding is that the process supports the use of well established guidelines and vulnerability databases in the assurance process. That is, our work demonstrates the adoption of open guidelines in a systematic assurance process. The usefulness of a variety of validation techniques (formal models, test sequence generation, static analysis, code inspection and testing) is also demonstrated. We have also used a variety of tools/frameworks (Robotium, FindBugs) in this process. For certain tasks where we had to rely on manual inspection it might be able to encode the coding guidelines into the development environment. An important observation is that not all vulnerabilities listed in the repositories are directly relevant. We need to filter out (e.g., in this case CWE-805) those that are not relevant.

Guideline	Brief Description	Functionality
CWE-805	Buffer Overflow	All functions
Cert Java Guideline IDS00-J	Validate Input	All functions
CWE-311	Missing Encryption of data	Create User Change Password Change Policy
CWE-732	Permissions of resources	Change Password
Cert Java Guideline ENVO3-J		
Cert Java Guideline MSC01-J	Weak Crypto	All functions
Cert Java Guideline EXP03-J	Comparing Strings	Login
CAPEC-112	Brute-Force	Login

Table 2. Mapping of Functionality to guideline

Guideline	Brief Description
CWE-129	Validation of Array Index
Cert Java Guideline EXP07-J	Short circuit of conditionals
Cert Java Guideline MET03-J	Methods that perform a security check must be declared private or final
Cert Java Guideline MET04-J	Use of overridable methods

Table 3. Guidelines relevant for Coding

This project was undertaken as, a generic process and its were not clear at the onset of this work. We agree with [1] that generic assurance processes are not directly applicable in the security domain. We have presented a security specific assurance process which is more specific. In our approach we not have used any sophisticated techniques (e.g. flow analysis [9] or formal specification and analysis for assurance of critical requirements [19]). The applicability of such techniques is currently being studied. Another issue we wish to study is to encode common attack patterns so that test automation techniques (e.g., model-based testing) can be used.

References

- [1] D. Balzarotti, G. Banks, M. Cova, V. Felmetzger, R. A. Kemmerer, W. Robertson, F. Valeur, and G. Vigna. An experience in testing the security of real-world electronic voting systems. *IEEE Transactions on Software Engineering*, 36(4):453–473, 2010.
- [2] Compuware. Direct recording electronic (dre) technical security assessment report. Technical report, Ohio Secretary of State, 2003.
- [3] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
- [4] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security, CCS '09*, pages 235–245, New York, NY, USA, 2009. ACM.
- [5] T-M. Grønli, J. Hansen, and G. Ghinea. Android vs windows mobile vs java me: a comparative study of mobile development environments. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments, PETRA '10*, pages 45:1–45:8, New York, NY, USA, 2010. ACM.
- [6] M. Howard. Mitigate security risks by minimizing the code you expose to untrusted users. November 2004.
- [7] P. K. Manadhata, J. M. Wing, M. A. Flynn, and M. A. McQueen. Measuring the attack surfaces of two ftp daemons. In *ACM CCS Workshop on Quality of Protection (QoP)*, 2006.
- [8] N. R. Mead and T. Stehney. Security quality requirements engineering (square) methodology. In *SESS '05: Proceedings of the 2005 workshop on Software engineering for secure systems building trustworthy applications*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [9] D. Muthukumar, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies, SACMAT '08*, pages 155–164, New York, NY, USA, 2008. ACM.

Description	The administrators credentials are saved in the file <code>admin.dat</code> . The credentials contain a username and a password. The test is to ensure that username and password are encrypted.
Validation Technique	Initial code inspection followed by white box testing. The files <code>TopMenu.java</code> and <code>RegisterAdmin.java</code> are used in the testing process.
Result	Although the password is encrypted the username is not encrypted.
Mitigation	Encrypt the username also before storing information.
Re-run validation	The modified system passes the original test.

Description	CWE recommends applications not use obsolete encryption algorithms (e.g., SHA-1) because they can be broken in hours.
Validation Technique	Manual code inspection is sufficient as one only needs to check the calls made to appropriate API.
Result	Positive as the application is using SHA-1 to encrypt credentials. This was found in the file <code>Digest.java</code> .
Mitigation	The National Institute of Standards and Technology (NIST) recommend changing the encryption from SHA-1 family to SHA-2 or SHA-5 family.

Table 4. Summary of Encryption Related Findings

- [10] P. Pari-Salas, P. Krishnan, and K. Ross. Model-based Security Vulnerability Testing. In *Australian Software Engineering Conference (ASWEC)*, pages 284–293, Melbourne, April 2007. IEEE.
- [11] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8:35–44, 2010.
- [12] K. S. Shankar and H. Kurth. Certifying open source-the linux experience. *IEEE Security and Privacy*, 02(6):28–33, 2004.
- [13] Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, December 2010.
- [14] D-H. Shih, B. Lin, H-S. Chiang, and M-H. Shih. Security aspects of mobile phone virus: a critical survey. *Industrial Management and Data Systems*, 108(4):478–494, 2008.
- [15] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. Towards formal analysis of the permission-based security model for android. In *International Conference on Wireless and Mobile Communications*, pages 87–92, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [16] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. In *Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust, 2010 IEEE International Conference on*, pages 944–951, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
- [17] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2006.
- [18] J. Viega. Building security requirements with CLASP. In *SESS '05: Proceedings of the 2005 workshop on Software engineering for secure systems building trustworthy applications*, pages 1–7, New York, NY, USA, 2005. ACM Press.
- [19] K. Weldemariam, R. A. Kemmerer, and A. Villafiorita. Formal specification and analysis of an e-voting system. In *International Conference on Availability, Reliability and Security*, pages 164–171. IEEE, 2010.

Description	Be aware of issues related to short-circuit behaviour of the conditional operators.
Validation Technique	Static Analysis followed by code inspection The static analyser FindBugs (http://findbugs.sourceforge.net/) indicates locations where this might be an issue. The code inspection checks that the execution of the conditionals has no side effect.
Result	No such issues were found.
Mitigation	No mitigation required.

Description	MET03-J and MET04-J which related to Java code concerning method declarations.
Validation Technique	Manual code inspection was used as there is easy technique to characterise functions that deal with security. It is also possible to check for overridable functions.
Result	Negative as the developer had adhered to the guidelines.
Mitigation	None required.

Table 5. Summary of Findings Related to Coding Standards

Description	The permission on the file <code>admin.dat</code> should confirm to the guidelines.
Validation Technique	Code inspection to check file permissions.
Result	The file <code>admin.dat</code> had the permission <code>MODE_WORLD_READABLE</code> . Thus every application can potentially access and read the file.
Mitigation	The Android Security Guidelines recommend using the <code>MODE_PRIVATE</code> permission, as no other application needs access to the <code>admin.dat</code> file.

Table 6. Summary of Findings Related to File Permissions
