Research on Android Application Security Configuration Rules Based on Fuzzing Test

Zi Nan [1, a*], Xiaojian Liu [2, b]
[1]School of Computer Science and Technology, Xi'an University of Science and Technology
Xi'an, China
[2]School of Computer Science and Technology, Xi'an University of Science and Technology
Xi'an, China
* Corresponding author: [a]925255442@qq.com
[b]e-mail: 780209965@qq.com

Abstract

In the official Android documentation, although it has described various configuration elements in the configuration file, some descriptions do not give accurate semantic definitions, and some descriptions do not show accurate relationships between related configuration elements, namely At present, there are uncertainties and ambiguities in the description of natural language in the document, which will cause some differences in the understanding of developers, and may cause developers to improperly configure the App to produce some security vulnerabilities. Therefore, this article mainly adopts the fuzzing-based security configuration rules research framework (ARMG) based on reading official documents, that is, for the existence of some configuration files with fuzzy semantics, the Fuzzing test method of multi-dimensional strategy is used to construct test cases. Analyze the test results to further clarify the rule definitions of related configuration items, and then use MobSF to statically analyze the sample set, and observe the compliance of the samples and the effectiveness of the rules by parsing, outputting logs, and comparing them. The final rules are used by designers or developers to improve development efficiency and better avoid software security vulnerabilities.

Keywords-Security configuration; Fuzzing; Rule mining; Security breach

1. Introduction

With the rapid development of science and technology, smart mobile devices have become more and more important in people's daily lives due to their rich functions, portability, and ease of operation. Different mobile devices carry different operating systems, and Android has become one of the most popular mobile operating systems in recent years. Since its release, the Android system has continued to improve, but this does not mean that the Android system is absolutely safe and reliable [1-2]. There are always criminals taking advantage of the vulnerabilities of the Android system itself to carry out information attacks and steal user information. Android security issues [3-4] appear to be particularly important.

First, Android developers do not always have the knowledge needed to understand the source code they are dealing with. When the code lacks documentation and comments [5], in order to make up for the lack of knowledge, developers usually refer to official Android documentation, team members, and other sources of information on the Internet. Information obtained through the Internet etc. does not always help answer the question of what a particular source code is doing. Current work has solved this problem through automatic summarization methods [6-7] and by creating abstract or abstract summaries [8]. But when developers want to understand the specific usage and effects of the configuration parameters in the code from the official documentation, it is not satisfactory. At present, there are uncertainties and ambiguities in the description of natural language in official documents, which will cause some differences in the understanding of developers, may cause developers to improperly configure App permissions and other security vulnerabilities.

Next, there is a lot of work in current Android application testing [9-11]. Including stochastic technology, such as Dynadroid with feedback [12], model-based technologies such as MobiGuitar [9] and SwiftHand, and search-based technologies such as EvoDroid and Sapienz [11], based on generated [13] fuzzing technology And the fuzzing technique based on mutation [14]. However, during the construction of test cases for fuzz testing, there is a single, single-dimensional problem in the current general test framework.

In response to the above two issues, this paper proposes a research framework for security configuration rules based on fuzzing testing. Test some Android security configuration items to obtain accurate configuration rules, and then use the MobSF automated testing tool to statically analyze the data sample application, and observe the compliance of the sample and the validity of the rules through analysis, log output and comparison.

2. Materials and Methods

2.1 Fuzzing

Fuzzing was first proposed by Professor Barton Miller of Wisconsin-Madison University in 1989 to test the robustness of UNIX systems. According to Miller et al., we believe that a fuzzy input may be an unexpected input, so fuzzing is a method of finding bugs [15] by providing unexpected input and monitoring abnormal results. It is a way to use fuzzing Modern software testing technology. Fuzzy testing is generally divided into 6 stages as shown in Fig. 1.

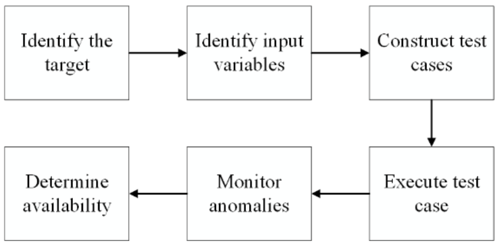


Figure 1. Fuzzing workflow

Test case generation is a key step in fuzz testing. The generation strategy can generally be divided into random generation, generation-based strategy, and mutation-based strategy.

Random generation refers to the use of random functions to generate pseudo-random data as input variables. As shown in Fig. 2(a), fields 2 and 4 are generated using malformed data, and other fields are generated using normal data. This method is a non-intelligent strategy, and its test efficiency is very low, and it is basically abandoned at present. The strategy based on generation refers to constructing malformed or special input data by analyzing the information of the measured target input variable, as shown in Fig. 2(b), performing bit or byte mutation operations on fields 3 and 4 of the sample. The test cases constructed using this method have relatively high code coverage, but it takes a certain amount of time to analyze a large amount of specification documents and manually generate test cases in the early stage. Familiarity with the target input will greatly affect the effectiveness of the constructed test cases. A mutation-based strategy refers to the use of random mutation or heuristic methods to modify certain input fields of the test sample based on the collection of known samples, and test cases can be generated without knowing the relevant knowledge of the tested target. The advantage is that it does not need to understand and analyze the format, specifications and other information of the tested target, and does not require a lot of preliminary work. It only needs to provide one or more variant sample.

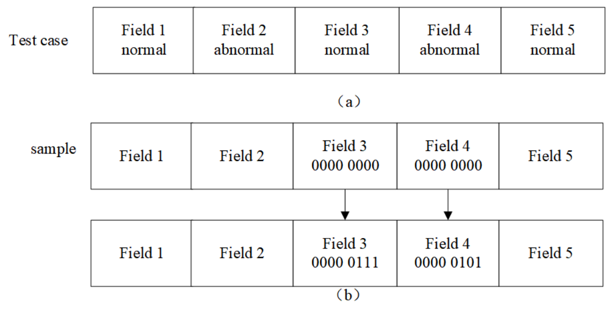
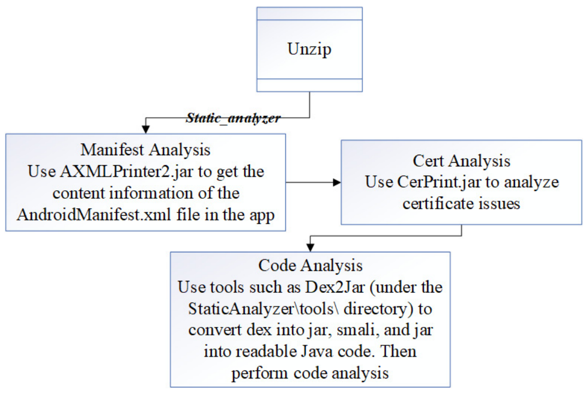


Figure 2. Construction strategy

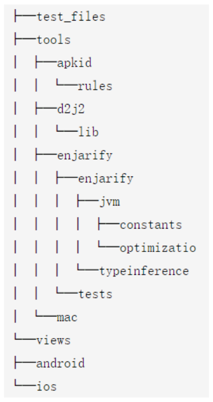
2.2 MobSF test tool

MobSF (Mobile-Security-Framework) is an open source mobile application (Android/ios/Windows) automatic testing tool, which can perform static and dynamic analysis and output reports on the web.

Static analysis includes Manifest Analysis, Cert Analysis, Code Analysis, and the process is shown in Fig. 3.


Unzip
Static_analyzer
Manifest Analysis
Use AXMLPrinter2.jar to get the content information of the AndroidManifest.xml file in the app
Cert Analysis
Use CerPrint.jar to analyze certificate issues
Code Analysis
Use tools such as Dex2Jar (under the StaticAnalyzer\tools\ directory) to convert dex into jar, smali, and jar into readable Java code. Then perform code analysis
Figure 3. MobSF static analysis process

Principle analysis: Print out the Static Analyzer directory tree structure (as shown in Fig. 4), you can understand that migrations are migration files, test_files are files used to test static tests, tools are tools for decompilation, etc., views are what we want to analyze Source code.


```
├─test_files
├─tools
|  ├─apkid
|  |  └─rules
|  ├─d2j2
|  |  └─lib
|  ├─enjarify
|  |  ├─enjarify
|  |  |  ├─jvm
|  |  |  |  ├─constants
|  |  |  |  └─optimizatio
|  |  |  └─typeinference
|  |  └─tests
|  └─mac
└─views
├─android
└─ios
```
Figure 4. Static Analyzer directory tree structure

2.3 Research Framework of Security Configuration Rules Based on Fuzzing Test

Before introducing this framework, in order to further improve the test efficiency, this article modified the fuzzing test process, added the investigation of historical security vulnerabilities, and through sufficient understanding, it is used to assist in the construction of test cases and test the security-related configuration items. As shown in Fig. 5.

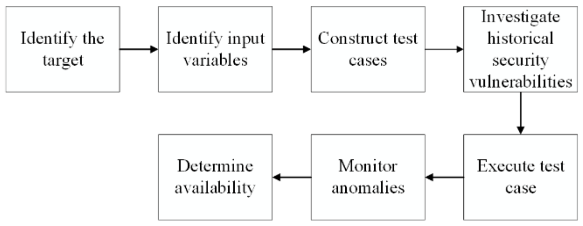


Figure 5. Fuzzing test process in this article

According to the Fuzzing test flow chart, this paper designs a complete rule mining framework, a security configuration rule test framework ARMG based on fuzzing, as shown in Fig. 6. Target investigation refers to the investigation of security configuration through the investigation of official Android documentation, familiarity with input variables, parameters and other information, in preparation for testing. Vulnerability research refers to obtaining information about security vulnerabilities related to target configuration through authoritative vulnerability databases such as China's National Information Security Vulnerability Database, as one of the basis for constructing test cases. Through the first two steps and the experience of deformed data construction, a deformed sample library is constructed for the field construction of test cases. The test case strategy is a construction strategy based on the combination of generation and mutation. Among them, the strategy based on generation needs to generate test cases based on test experience, etc. Based on mutation, the method of field mutation is used to construct test cases on the basis of normal and vulnerable samples. Then, according to the test results, analyze and derive the relevant security configuration rules. Finally, the MobSF tool is used to statically detect malicious and benign samples in the CLCMalDroid 2020 data set, output logs, and compare and analyze the rules obtained to obtain the compliance of the samples and the availability of the rules.

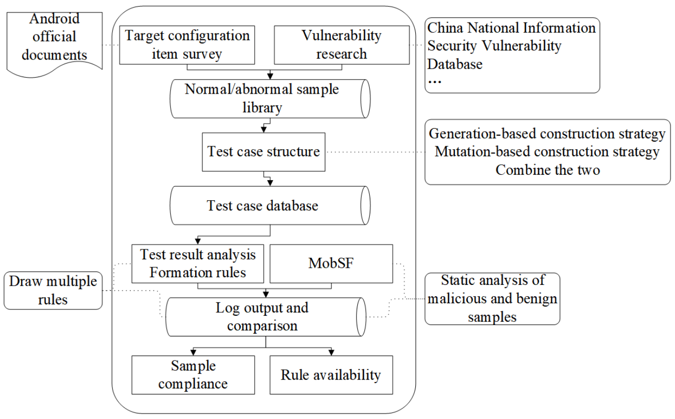

Android official documents
Target configuration item survey
Vulnerability research
China National Information Security Vulnerability Database ...
Normal/abnormal sample library
Test case structure
Generation-based construction strategy
Mutation-based construction strategy
Combine the two
Test case database
Test result analysis Formation rules
MobSF
Draw multiple rules
Static analysis of malicious and benign samples
Log output and comparison
Sample compliance
Rule availability

Figure 6. Security configuration rule testing framework based on fuzzing

2.4 Test case construction algorithm

Based on the generated strategy, test cases need to be generated based on test experience and so on. However, according to the above test framework, a mutation-based test case construction algorithm R_{mut} is proposed. Among them, use Q to represent the mutation matrix,$Q=\left(q_{ij}\right)_{n*m}$,$q_{ij}=0 or 1$,$1<i<n$, $1<j<m$. When q_{ij} is 1, it represents the f_i field of random variation sample s_j, and 0 represents no variation. R indicates that the test case is constructed according to the mutation matrix Q. For the mutated field, the random function $rand$ is used to change the field data. R_{mut} uses pseudo-code to indicate as shown in Table 1, where the $select\_data$ function is to select a data packet from the $sample\ set$, and the $variation$ function is to mutate the sample data packet according to the q_{ij} value.

TABLE I. R_{mut} PSEUDO CODE

```
$R_{mut}$
Input: $Sample\ Set$ , $Q$
Output: $R$
Begin
  $R=\Phi$ , //initialization
  for ($count$=0; $count$ <|$Sample\ Set$|; $count$ ++ ) do
      $R_{samples}$= $select\_data$(|$Sample\ Set$| );
      for($i$ =1; $i \leq n$; $i$++) do
         for($j$ =1; $j \leq n$; $j$++) do
              $R$=$variation$($R_{samples}$, $q_{ij}$ );
         end
      end
  end
end
```

3. Results&Discussion

3.1 Experimental environment and data set

The Android Studio integrated development environment is used for development, the virtual machine version is Android 10.0 and above, the tool for interacting with virtual Android devices, Android Debug Bridge (ADB) , and the operating system is Microsoft Windows 10.

Use the CLCMalDroid 2020 public data set. The downloaded data set is in .tar.gz format and needs to be decompressed in a Linux environment. After decompression, the file suffix will be changed to .apk format. The software types included are shown in Table II.

TABLE II. TYPES OF SOFTWARE IN THE DATA SET

Type	Adware	Banking	Benign software	Risk software	SMS malware	Total
Number	1253	2100	1795	2546	3904	11598

3.2 Experimental results and analysis

3.2.1 The test experiment results in the rule types shown in Table III: There are 19 sets of experimental results (rules), here is one set for illustration.

TABLE III. TYPES OF EXPERIMENTAL RESULTS

Total type	***Basic property configuration of the application***	***Application and component process analysis***	***Research on permissions configuration of applications and components***	***LaunchMode***
Sub-type	ShareUserId attribute value	The correct format of the process	App permissions collection between shared users	Standard
	Multiple apps ShareUserId	Priority of the application and its components	Inheritance of enforced permissions	SingleTop
	ShareUserId value is the package name of another installed App	Deploy component to other component processes	Permission analysis of communication between components	SingleTask
	Application package coverage	The meaning of the exported attribute and its default value	Intent's implicit start	SingleInstanc e
	Define but not register components	Multiprocess attributes and component instantiation analysis	The permissions imposed by the Intent action	allowTask -Reparenting
	Contains multiple MainActivity			taskAffinity

Inheritance of enforced permissions. The experimental results are shown in Table IV:

TABLE IV. INHERITANCE EXPERIMENT RESULTS OF ENFORCED PERMISSIONS

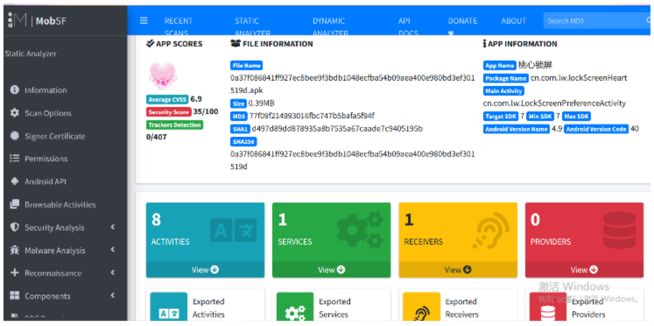
ID	App1. <uses-permission>	App2. permission	A21. permission	operate	result	RAM /MB	Running time/s
1	Null	Null	Null	Run App1, click the button "App2:FirstActivity"	Run successfully	112	5.367
2				Run App1, click the button "App2:SecondActivity"	Failed to run, App1 crashed	98	6.401
3				Run App2, click the button "SecondAct-ivity" and "ThirdActivity"	Run successfully	103	4.235
4	Null	P1	Null	Same as above	Failed to run, App1 crashed	94	3.763
5				Same as above	Failed to run, App1 crashed	102	6.358
6				Run App2	Pop-up window "No application installed"	86	5.872
7	Null	Null	P1	Same as above	Run successfully	115	6.825
8				Same as above	Failed to run, App1 crashed	98	5.932
9				Run App2, click the button "SecondAct-ivity" and "ThirdActivity"	Run successfully	119	7.384
10	P1	Null	P1	Same as above	Run successfully	121	8.324
11				Same as above	Failed to run, App1 crashed	94	5.365
12				Add exported="true" in A22, run App2, click button "Second- Activity" and "ThirdActivity"	Run successfully	115	7.623
13	P1	Null	{P1，P2}	Add two permissions in A22	Crash directly	93	4.356
14	P1	P1	P2	Same as the 3rd case	Run successfully	101	5.457
15				Same as the 3rd case	Failed to run, App1 crashed	95	3.872
16				Same as the 3rd case	Run successfully	91	5.356

Draw the following rules:

Rule 1: Enforced permission: If the enforced permission value of an activity does not exist, the value is derived from the enforced permission value of its application; otherwise, its value will override the value of its application.

Rule 2: If the value of <activity android:exported> is not defined, the value of exported adopts the default value. The default value depends on the existence of intFilters. When the filter does not exist, the default value is false, and the activity is exclusively for internal use; when there is at least one filter, it means that the activity is exclusively for external use. There is a security risk of privacy leakage.

3.2.2 MobSF tool analysis results of samples are shown in Fig. 7.



Average CVSS 6.9
Security Score 35/100
Trackers Detection 0/407

File Name 0a37f086841ff927ec8bee9f3bdb1048ecfba54b09aea400e980bd3ef301519d.apk
Size 0.39MB
MD5 77f09f214993016fbc747b5bafa5f94f
SHA1 d497d89dd878935a8b7535a67caade7c9405195b
SHA256 0a37f086841ff927ec8bee9f3bdb1048ecfba54b09aea400e980bd3ef301519d

App Name 桃心锁屏
Package Name cn.com.lw.lockScreenHeart
Main Activity cn.com.lw.LockScreenPreferenceActivity
Target SDK 7 Min SDK 7 Max SDK
Android Version Name 4.9 Android Version Code 40

8 ACTIVITIES
1 SERVICES
1 RECEIVERS
0 PROVIDERS

Figure 7. Malware analysis results

3.2.3 The ARMG framework proposed in this paper is compared with the existing related engineering frameworks SPIKE, Peach, Sulley and AFL, as shown in Table V: ARMG uses a multidimensional strategy based on generation and mutation to improve the detection rate of component configuration to a certain extent, uses manual analysis and testing to make up for the shortcomings of Fuzzing technology, uses methods such as analyzing vulnerability knowledge to improve the effectiveness of test cases, and uses log output And debugging methods improve the degree of automation of the test.

TABLE V. COMPARISON OF ARMG AND RELATED FRAMEWORKS

Tool	Construction strategy	Dimension	Monitoring and debugging	Intervention	Manual test	Vulnerability knowledge
SPIKE	Generation	Single dimension	none	high	none	none
Peach	Mutation	Single dimension	Log analysis, no debugging	high	none	none
Sulley	Generation	Single dimension	Ping mode, debugging	middle	none	none
AFL	Mutation	Single dimension	Adjust seed selection based on feedback	middle	none	none
ARMG	Generation and mutation	Multidimen -sional	Monitoring package and log output, With debugging	middle	have	have

3.2.4 Divided according to the type of application, and compared with log analysis and rules, the comprehensive compliance degree of different types of applications is counted, as shown in Fig. 8: The more compliant the software, the higher the security. And it can be seen that the degree of compliance can determine whether a piece of software is benign or malicious, which reflects the effectiveness of the rules obtained in this article.

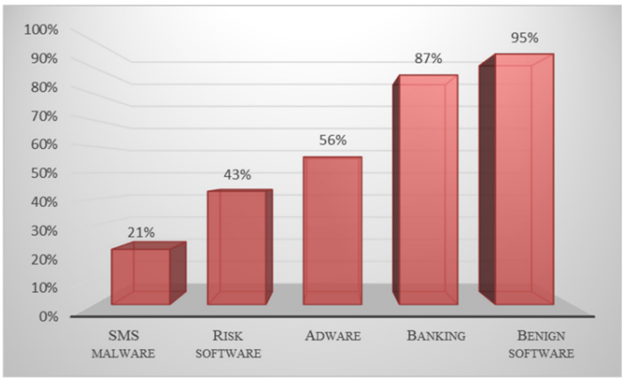


Figure 8. Comprehensive compliance statistics for each category of applications

4. CONCLUSIONS

Based on reading official documents and conducting vulnerability analysis, this paper proposes a security configuration rule research framework (ARMG) based on fuzz testing. This framework improves the detection rate of component configuration to a certain extent and improves the effectiveness of test cases, and use log output and debugging methods to improve the degree of automation of the test. After deriving the rules, the applicability of the rules is demonstrated by observing the compliance of the samples. The comprehensive compliance of the benign samples is as high as 95% and the malicious samples are only 21%. The more compliant the sample, the smaller the potential safety hazard. These rules are used as a supplement to official documents for designers and developers to improve development efficiency and better avoid software security vulnerabilities. However, when MobSF statically detects the AndroidManifest.xml file in the software, the analysis of the configuration parameters is not comprehensive enough. In the next step, we will continue to study the detection tools that detect App configuration rules, supplement and improve the detection content.

REFERENCES

[1] Z. Yue and F. D. Yang, "Research on the Security of Android Application Privacy Authority," (in Chinese) Information Security Research, vol. 7, no 3, 2021.

[2] Y. L. Fang, “Research on Secure Communication Based on Android System Components,” (in Chinese) Beijing University of Posts and Telecommunications, 2017.
[3] S. Garg and N. Baliyan, “Android Security Assessment: A Review, Taxonomy and Research Gap Study, ” Computers&Security, 2020.
[4] Z. Zhao, “Research on Security Threats and Attacks of Android,” (in Chinese) Wireless Internet Technology, 2020, vol. 17, no. 24.
[5] A. Emad, B. Gabriele, L. V. Mario and L. Michele, “Documentation of Android Apps,” IEEE Transactions on Software Engineering, 2019, pp 1-1.
[6] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A.Kraft, “Automatically documenting unit test cases,” in 2016 IEEE International Conference on Software Testing, Verification and Validation(ICST), April 2016, pp. 341–352.
[7] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata,and C. Sutton, “Autofolding for source code summarization,” IEEE T ransactions on Software Engineering, vol.43, no.12, pp.1095-1109, 2017.
[8] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, “On the use of automated text summarization techniques for summarizing source code,” in 2010 17th Working Conference on Reverse Engineering, Oct 2010, pp. 35–44.
[9] A. Domenico Amalfitano, F. A. Rita, T. Porfirio, T. B. Dzung, and M. A. M, “MobiGUITAR–A Tool for Automated Model-Based Testing of Mobile Apps”, IEEE Software, 2014.
[10] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, “Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests,” InIntl. Conf. on Software Testing, Verification and Validation (ICST). 2017, pp. 149–160.
[11] M. Ke, H. Mark and J. Yue, “Sapienz: Multi-objective Automated Testing for Android Applications,” InIntl. Symp. on Software Testing and Analysis (ISSTA), 2016, pp. 94–105.
[12] M. Aravind, T. Rohan and N. Mayur, “Dynodroid: An Input Generation System for Android Apps,” InJoint Meeting on Foundations of Software Engineering (ESEC/FSE). 2013, pp. 224–234.
[13] H. HyungSeok, C. S. Kil, “IMF: Inferred Model-based Fuzzer,” In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017, pp. 2345–2358.
[14] L. Jun, Z. B. Dong, Z. Chao, “Fuzzing: a survey,” Cybersecurity. 2018, vol. 1, no. 1, p. 6.
[15] M. V. J. M, H. HyungSeok, H. Choongwoo, C. S. Kil, E. Manuel, S. E. J, and W. Maverick, “The Art, Science, and Engineering of Fuzzing: A Survey,” IEEE Transactions on Software Engineering, 2019, vol. 47, pp. 1-1.