Journal of Computers Vol. 33 No. 2, April 2022, pp. 189-200
doi: 10.53106/199115992022043302017

FASSFuzzer—An Automated Vulnerability Detection System for Android System Services

Le Weng[1], Chao Feng[1*], Zhi-Yuan Shi[1], Ying-Min Zhang[2], Lian-Fen Huang[2]

[1] School of Electronic Science and Engineering, School of Informatics, Xiamen University, Fujian, China
23120191150201@stu.xmu.edu.cn, chaof@xmu.edu.cn, zyshi@xmu.edu.cn
[2] School of Informatics, Xiamen University, Fujian, China
23320181154355@stu.xmu.edu.cn, lfhuang@xmu.edu.cn

Received: 11 January 2022; Revised: 4 February 2022; Accepted: 4 March 2022

Abstract. As the core component of Android framework, Android system services provide a large number of basic and core function services for Android system. It has a lot of resources and very high system permissions. And for the Android system, it is a very important attack surface. Attackers can use Android system service vulnerabilities to steal user privacy, cause Android applications or Android system denial of service, remote malicious code execution and other malicious behaviors, which will seriously affect the security of Android users. Based on fuzzy testing technology, this paper designed and implemented a vulnerability mining system for Android system services, optimized and improved the fuzzy testing method, so as to improve the speed and effectiveness of vulnerability mining, and timely submitted the discovered vulnerabilities to the corresponding manufacturers and security agencies, to help Android manufacturers repair the vulnerabilities in time. The main work of this paper is as follows: Aiming at the null pointer reference vulnerability of Android system services, we designed and implemented an automatic fast mining system FASSFuzzer. FASSFuzzer uses ADB to quickly detect null pointer reference vulnerabilities in Android services. At the same time, FASSFuzzer added automatic design to automatically perceive the generation of vulnerabilities and ensure the full automation of the whole vulnerability mining process, and automatically generate a vulnerability mining report after the completion of vulnerability mining.

Keywords: Android system service, vulnerability detection, fuzzing

1 Introduction

With the rapid development of mobile Internet, mobile smart phones have been widely used. At the same time, the open and diverse characteristics of Android operating system also attract a large number of developers and consumers, so in recent years, a large number of Android device manufacturers have emerged in China and abroad to meet the needs of users, including Xiaomi, Huawei, Meizu and OnePlus mobile phone manufacturers. As Android occupies the majority of the mobile operating system market share, coupled with its open source characteristics, the Android platform has attracted many malicious attackers to exploit Android vulnerabilities for damage or profit.

The number of CVE Vulnerabilities per year for the Android platform since 2009, when Android joined the Common Vulnerabilities and Exposures to the CVE project, and since 2019. As shown, the number of vulnerabilities began to increase sharply from 2015 and reached 843 in 2017, which was mainly because Google and various Android manufacturers began to pay more attention to the security of Android system at this stage, and Google set up a special Project Zero [1]. Each Android manufacturer also established their own Security Response Center (SRC) to maintain system Security, and provided platforms for Vulnerability authentication and reward for Baike, such as Xiaomi SRC and Huawei SRC. Therefore, vulnerabilities grew rapidly in this stage.

Facing the situation that Android system vulnerabilities is complicated, the research has gradually flourished, that security researchers have focused on the research of Android system vulnerabilities mining. The most classic of intent fuzzers is IntentFuzzer [2], which fuzzes the components of Android applications by constructing empty data and random serialized data.

R. Sasnauskas et al. [3] used static analysis and random test sample generation to construct more effective Intent information, and fuzzed multiple applications in Google Mall, aiming to discover denial of service vulnerabilities that caused soft restarts of applications or systems. M. Zhou et al. [4] proposed DroidRVMS, which first obtains component information through reverse engineering, then obtains the information of the Intent object through static data flow analysis and constructs the control flow graph of the component to expand the coverage

* Corresponding Author

of test examples, and implements vulnerability mining through Accessibility technology automation. Aiming at the vulnerability mining of the standard library in the native library layer of the Android system, MFFA [5] fuzzed the libstagefright multimedia library by constructing a correctly structured but malformed multimedia file, and dug multiple CVE vulnerabilities. Android-afl [6] ported AFL [7] to Android and discovered two denial of service vulnerabilities in media server. K. Ispoglou et al. proposed FuzzGen [8], which infers the interface and dependency graph of the library through the analysis of the entire system, and generates the stub code of LibFuzzer [9] through the dependency graph to realize automatic vulnerability mining. FuzzGen selected 7 libraries in Debian and AOSP, and found 17 previously unpatched vulnerabilities, of which 6 vulnerabilities obtained CVE certification. FuzzGen selected 7 libraries in Debian and AOSP, and found 17 previously unpatched vulnerabilities, of which 6 vulnerabilities obtained CVE certification. Syzkaller [10] is an unsupervised coverage-based bootstrap Linux kernel fuzzing tool. Syzkaller can effectively obtain kernel code coverage information and detect memory vulnerabilities that occur when the code is running.

Almost all Android applications need to use system services - a core part of the Android framework. Android system services are packages of software and hardware services provided by the Android platform for developers. Common system services include network services, multimedia services, and short message services. Research on Android system service vulnerability mining is also an important part of Android system security. In 2015, K. Wang et al. [11] obtained the data flow of Java system services through IPC Hooker to build the original seed library, mutated the seed library to generate a data set for fuzzing, and detected 2 vulnerabilities in Android 5.1.0. In 2016, H. Feng et al. [12] analyzed the security boundary of the attack surface after studying more than 100 system service-related vulnerabilities, and proposed BinderCrack [13]. BinderCrack constructs the input of the target system service by recording the request data of 30 popular APPs. Model, which is a parameter-aware fuzzing test. Y. Shao et al. proposed Kratos [14], which uses static analysis to build accurate function call graphs to identify paths that allow third-party applications with insufficient permissions to access sensitive resources and violate security policies, and successfully discover 14 paths that can lead to system damage and privacy breaches. In 2017, A.K. Iannillo et al. proposed Chizpurfle [15], Chizpurfle conducts fuzz testing for system services customized by third-party Android device manufacturers, and obtains block coverage information executed by system services through dynamic binary instrumentation technology as a feedback item to enhance the efficiency of vulnerability mining. J. Wu et al. proposed ExHunter and ExCatcher for Java exceptions that may cause denial of service in the Android system. ExHunter uses Java dynamic reflection and randomly generated test samples to test Java system services, and ExCatcher avoids key system services from being killed by setting a whitelist, thereby mitigating the impact of Java exceptions. In 2018, L. Zhang et al. proposed Invetter [16]. Invetter identifies the input validation of sensitive system services through static analysis and machine learning, and combines some security rules for vulnerability detection of system services. In the end, Investter found 20 vulnerabilities by scanning 8 Android system images. In 2019, D. Cotroneo et al. [17] further improved Chizpurfle, using genetic algorithms to fuzz test system services customized by third-party Android device manufacturers using a variety of fitness functions and selection algorithms. Z. Zou et al. [18] used the reflection mechanism to obtain Java system services and their API information, and fuzzed the target system services by constructing malformed parameters, and found 5 existing denial-of-service vulnerabilities in 3 versions of Android system testing. In 2020, B. Liu et al. proposed FANS [19], which is used for generation-based fuzzing of Android's local system services. FANS consists of four components: Interface Collector, Interface Model Extractor, Dependency Inferencer, and Fuzzing Engine. FANS can identify multi-layer interfaces and generate more effective test cases through interface models, interface dependencies and variable dependencies. 30 local system service related vulnerabilities were found on the system with Android version android-9.0.0_r46, of which 20 vulnerabilities Got confirmation from Google.

2 Related Work

Android vulnerabilities seriously affect the security of the Android platform. Therefore, security researchers need to use corresponding vulnerability mining technology to timely and effectively mine the latest Android security vulnerabilities, so as to help Android manufacturers fix the vulnerabilities as soon as possible and prevent them from being exploited by malicious attackers in advance. And cause damage to the user's security and privacy. Android vulnerability mining technology is mainly divided into static vulnerability mining technology and dynamic vulnerability mining technology. This section will introduce the commonly used Android vulnerability mining technology.

2.1 Android System Service

Android system services are packages of various hardware and software services provided by Android for developers. With the continuous upgrading of the Android system version, Android system services for developers are also expanding and richer, from the initial 50 or 60 system services to hundreds of system services now, to help developers to develop applications more easily. In the Android system, many functions are provided by system services, such as network connection functions, device lock screen settings, multimedia functions, and camera functions. Each system service will provide rich interfaces so that developers can perform specific operations. For example, the system service of device lock screen settings allows developers to obtain and set the mode and password of the device lock screen.

Table 1 introduces some commonly used Android system services and their functions. Android system services are divided into two categories:

The first category is the Java system service located in the Android application framework layer, which is mainly implemented in the Java language. Most of the Android system services are Java system services. The Java system service is started by the system process system_server after the Android system is started. According to the different service objects, it is divided into two types. The first is the core platform service, which serves the Android internal platform, such as PackageManager Service, ActivityManager Service and WindowManager Service, etc. to ensure that the Android system can run normally and stably. The second is hardware services, which serve Android applications and provide applications with APIs for the underlying hardware of the operating system, such as LockSettings Service, Notification Service, and Location Service.

The second category is the local system service located in the native library layer of the Android system, which is mainly implemented in the native language C++, and helps the local system service improve the operating efficiency through the native language. Local system services are further divided into local daemons and Native system services. The local daemon has always existed since the Android system is started, helping the Android system to create a basic operating system environment, such as local daemons such as SurfaceFlinger and Mediaserver. Most of the Native system services are started by the local daemon Mediaserver after the Android system starts, such as Camera Service and AudioPolicy Service.

Table 1. Introduction to common Android system services

Android system_services	Categorize	Function introduction
PackageManager Service	Java system_services	Responsible for the management of application package related information
ActivityManager Service	Java system_services	Responsible for the management of the four major components of Android
WindowManager Service	Java system_services	Responsible for Android window management
LockSettings Service	Java system_services	Responsible for phone lock screen settings
Notification Service	Java system_services	Responsible for notification of system events
Location Service	Java system_services	Responsible for reading and setting location information
SurfaceFlinger	Local system_services	Responsible for the display of user interface graphics
Mediaserver	Local system_services	Responsible for starting and initializing other multimedia services
Camera Service	Local system_services	Responsible for taking pictures and videos of the camera
AudioPolicy Service	Local system_services	Responsible for developing system audio strategy

In addition to the system services in the Android source code, various Android device manufacturers will also add or customize the Android system services in order to enhance their market competitiveness. As shown in Fig. 1, Xiaomi has expanded 16 system services on the MIUI 12 system based on Android 10 to provide users with more function choices. At the same time, some manufacturers will also remove some authority verification and input verification in system services to improve the overall running speed and fluency of the system, and modify system services to provide support for the specific hardware of their products, but these changes also greatly increase the code risk.

```
zym@ubuntu:~$ adb shell service list | grep miui
6       miui.sedc: [com.xiaomi.security.devicecredential.ISecurityDeviceCredenti
alManager.v1]
7       miui.mqsas.MQSService: [miui.mqsas.IMQSService]
8       miui.face.FaceService: []
9       miui.contentcatcher.ContentCatcherService: [miui.contentcatcher.IContent
CatcherService]
12      miui.whetstone.power: [miui.whetstone.power]
13      miui.whetstone.klo: [miui.whetstone.klo]
14      miui.whetstone.mcd: [miui.whetstone.mcd]
58      whetstone.activity: [com.miui.whetstone.server.IWhetstoneActivityManager
]
59      ProcessManager: [miui.IProcessManager]
62      MiuiBackup: [miui.app.backup.IBackupManager]
63      MiuiInit: [miui.os.IMiuiInit]
64      security: [miui.security.ISecurityManager]
169     miui.shell: [miui.IShellService]
179     miuiboosterservice: [com.miui.performance.IMiuiBoosterManager]
187     miui.fdpp: []
194     miui.mqsas.IMQSNative: []
```

Fig. 1. List of system services added by MIUI 12

2.2 Static Vulnerability Mining Technology

The static mining technology of Android vulnerabilities does not need to run the target program, but analyzes the lexical, syntax and semantics of the program through the control flow graph, program dependency graph and data flow graph of the target program, and then uses type deduction and security rule checking. Vulnerability mining using methods such as model checking [20-21]. Commonly used static mining technologies for Android vulnerabilities include static taint propagation analysis technology and reachable path analysis technology.

The static taint propagation analysis technology mainly detects data-related vulnerabilities, and is used to track and analyze the flow of taint data in the target program. The technology first selects a pollution source as the starting point of propagation, and marks the input data of the pollution source as tainted data. During the propagation process, if other data has a data dependency with the tainted data, it will be infected and become tainted data, and then according to the tainted analysis rules The data is tracked, and at some key points of the target program, it is detected whether the tainted data will cause damage to the normal operation of the program. [22-23] In the specific vulnerability mining, the static taint propagation analysis technology will first parse to obtain the intermediate representation of the target program code, and then obtain the call graph and program dependency graph of the target program and other information, and use the taint analysis rules to identify the target in the intermediate representation of the code. The pollution source of the program and the key points that need to be detected can finally be analyzed based on the data flow information or the dependency information to analyze the spread of the taint, so as to check whether there is a vulnerability that can be destroyed by the taint in the target program.

Reachable path analysis technology is mainly used to mine confusing proxy vulnerabilities. This type of vulnerability is because there may be unexpected reachable paths in the program, allowing malicious attackers to directly bypass the protection mechanism to reach the target interface through this path [24]. Static taint propagation analysis technology pays more attention to the propagation of data flow, while reachable path analysis technology pays more attention to analyzing the control flow of the target program. Constraint solving, and then observe whether there is an interface that can directly call sensitive APIs on the control flow, so as to detect whether there is an obfuscated proxy vulnerability in the target program.

2.3 Dynamic Vulnerability Mining Technology

With these sizes, the interline distance should be set so that some 45 lines occur on a full-text page. Many vulnerability triggering conditions depend on the context provided by the target program when it is actually running. The dynamic mining technology of Android vulnerabilities is to test during the running process of the program. Compared with static mining technology, it can more intuitively see the triggering and triggering of vulnerabilities. its trigger consequences. Commonly used Android dynamic vulnerability mining technologies include symbolic execution technology and fuzzing testing technology.

Symbolic execution technology is used to explore the execution path of a program, use symbolic expressions to simulate the execution process of the program, express the output of the program as an expression composed of

these symbols, and then use a constraint solver to solve it for semantic analysis [25-27]. In the specific vulnerability mining, because the technology needs to analyze the execution path of the program, firstly, the intermediate representation of the target program code is obtained by parsing, and then the call graph and control flow graph of the target program are obtained. Next, the calculation expressions composed of constants and symbolic values are used to express the values of variables in the program, and the constraints of symbolic values are set through vulnerability analysis rules and graph information, including path conditions and conditions for program vulnerabilities. The execution path is selected by the path condition, and whether the program is vulnerable is determined by the condition of the vulnerability of the program. However, the efficiency of symbolic execution technology is also greatly limited. If there are too many program branching conditions, the path will explode. If there are a large number of complex constraints, the efficiency of the constraint solver will also be greatly reduced.

Fuzzing technology is an automatic or semi-automatic vulnerability mining technology based on defect injection, which constructs a large number of random and unexpected semi-valid input data for the target program, and then monitors whether there is an exception during the execution of the program. Exceptions are analyzed to find vulnerabilities in programs [28-29]. The key to fuzzing technology is the construction of semi-effective input data. The test data needs to conform to the input syntax to improve code coverage, and malformed data is used to detect program security [30]. Input data can be constructed in two ways: generation-based and mutation-based [31]. Generation-based fuzzing is to generate test samples that are more in line with the input grammar on the basis of known target protocols or sample formats. Mutation-based fuzzing is Data mutation rules are applied to the original seed library to generate more random and unexpected test cases, so if the two methods are combined, semi-valid input data will be better generated, thereby improving the efficiency of fuzzing.

Fuzzing testing technology has been widely used in academia and industry, and it is the most effective method for vulnerability mining. Therefore, this paper selects fuzzing testing technology as the mining technology for Android system service vulnerabilities. At the same time, fuzzing technology is also continuously optimized and developed in the direction of automation and intelligence. In terms of automation, security researchers need to design semi-automatic or fully-automated solutions according to the type of exploited vulnerabilities and the impact on the operating system, thereby reducing the human testing input. The FASSFuzzer designed in this paper is a fully automated vulnerability mining system. In terms of intelligence, currently common intelligent solutions include feedback on the quality of test cases based on code coverage to optimize data generation strategies, combined with symbolic execution to explore program execution paths to improve test code coverage, program-based input format and syntax to generate high-quality test cases. Since the code coverage information of Android system service execution cannot be obtained at present, the IASSFuzzer designed in this paper will focus on the third scheme, which improves the efficiency of vulnerability mining by improving the quality of input data.

3 Experiment

3.1 Fundamental

There have been many null pointer reference vulnerabilities in Android system services, such as CVE-2020-28345 [32] and CVE-2019-9279 [33] for Wi-Fi service, CVE-2019-9430 [34] and CVE-2019- 9400 [35] for Bluetooth services, CVE-2016-3821 [36] for media services. Because the code implementation of these system services does not check the input parameters for null pointer references. When malicious attackers pass in empty serialized data as input parameters to the vulnerability interfaces of these system services, it will cause the Android system to crash and Restart appears.

Aiming at the situation of Android system denial of service caused by the null pointer reference vulnerability of Android system services, this paper designs and implements an automatic fast mining system--FASSFuzzer. The main feature of FASSFuzzer is fast detection and automatic mining, which helps testers to quickly check for potential null pointer reference vulnerabilities in Android system services. At the same time, without manual intervention, the potential vulnerability information can be directly obtained through the vulnerability mining report generated by FASSFuzzer. And reduced manual testing costs.

In order to quickly scan and test all the interfaces of Android system services, FASSFuzzer will not consider the semantic information as the interface input data, but directly use the empty serialized data as the input data for each interface, so as to check whether the interface has been properly verified for empty data. At the same time, FASSFuzzer has added an automated method. Compared with other operating systems, such as using AFL to mine memory vulnerabilities in a package on Linux, if a vulnerability that do not affect the Linux system is discovered, but such vulnerabilities will lead to The Android system restarts. In order to ensure that the vulnerability mining

process is not disturbed by the status of the Android system, FASSFuzzer will be deployed on the PC side and use adb (Android Debug Bridge) to communicate with Android system services. At the same time, FASSFuzzer will track the status of the Android system in real time, automatically record the vulnerability information that causes Android to restart, and automatically generate a vulnerability mining report after the vulnerability mining is completed.

3.2 Automation Design

FASSFuzzer will automatically identify and record every denial of service vulnerability that causes the Android system to restart. At the same time, the entire process does not require manual intervention, and can automatically complete the testing of all system services. FASSFuzzer treats each system service as a test unit, Fig. 1 is automated testing process. In the process of FASSFuzzer's vulnerability mining, if the Android device restarts, the reason is that the null pointer's reference exception causes the system_server process to be killed, so that all system services will also be killed, and eventually the entire system will crash and restart. Therefore, FASSFuzzer will preliminarily determine the occurrence of the vulnerability through the restarting phenomenon of the Android system.

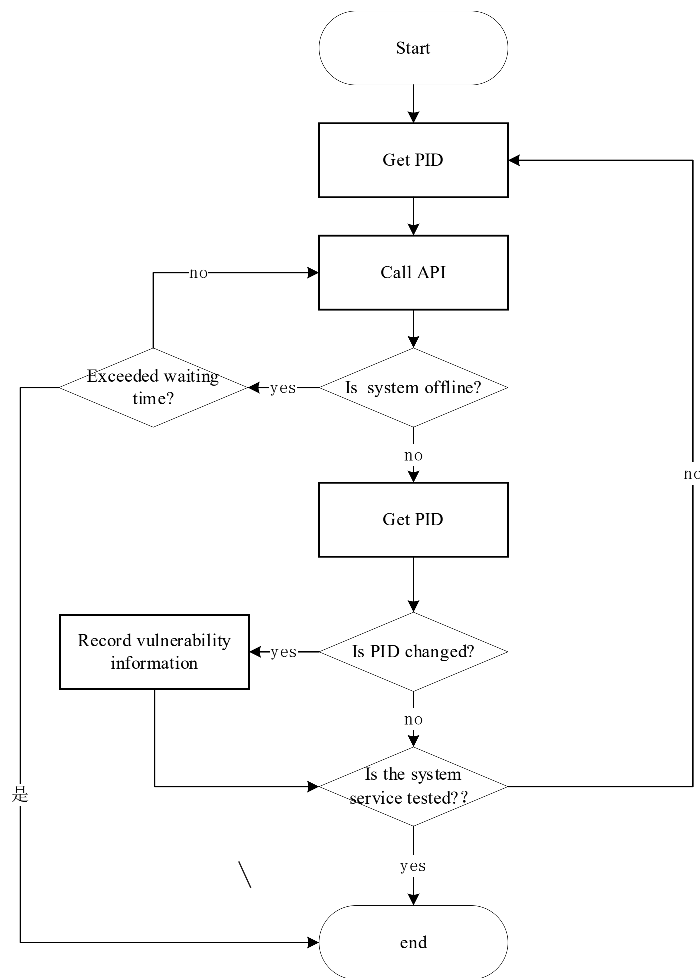

Fig. 1. Android system services automated testing process

System_server is a core process of the Android system, which runs most of the Android system services. When the Android system starts, zygote is one of the first processes created by the system, and then zygote will immediately create the system_server process as its child process, so the system_server process is the process that the Android system starts automatically. Because Process Identification (PID) will change after each system restart, so FASSFuzzer will sense the occurrence of vulnerabilities and record the vulnerability information through

whether the PID of the system_server process has changed.

In order to determine whether the Android system restarts after calling each API, FASSFuzzer will obtain the PID of the system_server process before calling the API, and then judge whether the system restarts by comparing the PID of the system_server before and after calling the API. If the PID changes, it means that the system has been restarted, and FASSFuzzer will record the system service and API method number corresponding to the potential vulnerability.

The automation of FASSFuzzer is based on adb. As a command-line tool of Client-Server architecture, adb allows testers to control Android devices on the PC side. FASSFuzzer mainly uses the relevant operation instructions (service) of Android system services in adb. The Service command is a built-in command of the Android system, which is convenient for testers to obtain and test Android system services. The operation usage of the Service command is shown in Fig. 2. The list command is used to obtain all system services in the Android device, including the system services that come with Android and those added by third-party device manufacturers. The Check command is used to detect whether the system service has changed, and check whether the Android device contains the specified system service through the return information of the command.

```
root@hammerhead:/ # service -h
Usage: service [-h|-?]
       service list
       service check SERVICE
       service call SERVICE CODE [i32 N | i64 N | f N | d N | s16 STR ] ...
Options:
   i32: Write the 32-bit integer N into the send parcel.
   i64: Write the 64-bit integer N into the send parcel.
   f:   Write the 32-bit single-precision number N into the send parcel.
   d:   Write the 64-bit double-precision number N into the send parcel.
   s16: Write the UTF-16 string STR into the send parcel.
```

Fig. 2. Service command's operational usage

Each system service contains multiple APIs, and each API is assigned a method number (Code). The code ranges from 1 to the number of APIs contained in the system service. The call instruction will call the specified method number of the specified system service, that is, to call the specific API, and the parameters can be set according to the parameter type of the API. The supported input parameter types include integer (int), floating point (float), double Precision floating point type (double) and string type, but there are many other parameter types for the API input parameters of system services, so the call instruction has certain usage restrictions and cannot be adapted to all system services according to the parameter types of the API.

The Call command will pack the input parameters into serialized data and send it to the API. Every time after calling the call commend, there will be a return value, and the execution of the API can be judged by the return value, such as the situation that the system service does not exist temporarily due to the system restart, and the situation that the system service has been traversed and the next Code has no corresponding API, and the situation where the operation permission is insufficient and the API cannot be executed.

Therefore, in order to ensure the continuity of the vulnerability mining process, after FASSFuzzer uses the call command to call the API of the system service, it will judge the current state of the Android system according to the return value of the API. If the Android system is offline, it means that the Android system may be restarting due to a null pointer reference vulnerability or other unexpected exceptions have occurred, then FASSFuzzer will enter a waiting state and sleep for a certain period of time until the system runs normally. If the system still cannot return to normal state after the timeout, FASSFuzzer will terminate the vulnerability mining in advance. Finally, if FASSFuzzer detects that the current system service has been tested, it will automatically switch to the next system service for testing.

3.3 FASSFuzzer's Vulnerability Mining Process

The vulnerability mining process of FASSFuzzer is shown in Fig. 3. FASSFuzzer will first clear the Android system log to ensure that the log content is generated during the vulnerability mining process. Then FASSFuzzer will obtain the complete list of Android system services through the service list command, and start to mine the vulnerabilities of each system service in turn.

For each system service, FASSFuzzer uses the service call command to call each of its APIs, and judges

whether the test of the system service has been completed according to the return value of the command. At the same time, FASSFuzzer will calculate the number of APIs contained in each system service and store it in a local json file.

Because the purpose of FASSFuzzer is to quickly mine the null pointer reference vulnerability of Android system services, it is not necessary to construct specific input parameter data according to the number and type of input parameters of each API, and only need to pass empty serialized data as input parameters. Just give the API, and then observe whether the API has vulnerabilities.

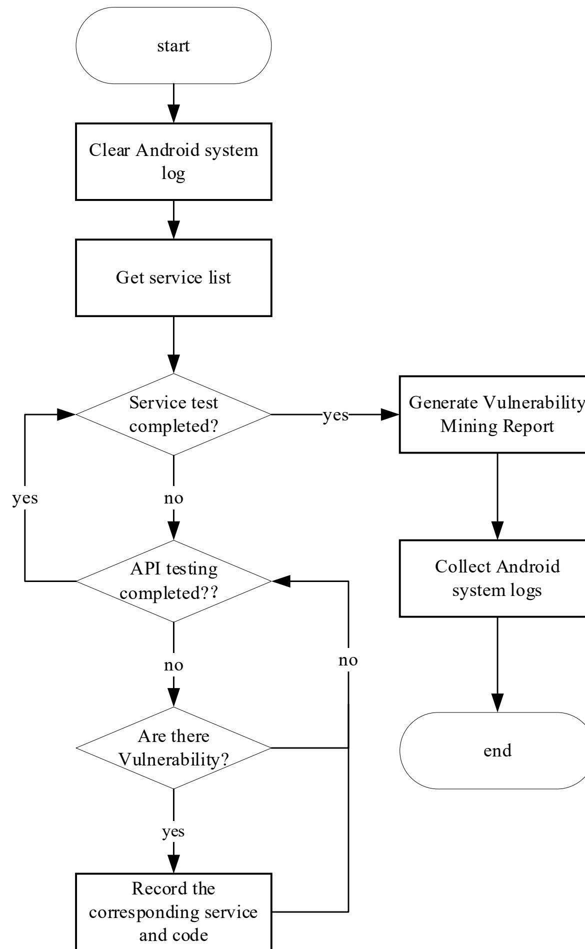

Fig. 3. FASSFuzzer vulnerability mining process

FASSFuzzer can automatically sense the occurrence of vulnerabilities. If the system restarts, FASSFuzzer will preliminarily determine that there is a vulnerability in the API, and record the corresponding system service and API method number. After all system services are checked, FASSFuzzer will automatically generate a report on this vulnerability mining, and store the system services and API method numbers corresponding to all potential vulnerabilities in a local json file for subsequent confirmation and utilization of the vulnerability.

Finally, FASSFuzzer will extract the log information generated during the vulnerability mining process through adb. In order to better analyze the log content, FASSFuzzer generates two log files based on the log information and stores them locally. One of the files records all the log information in detail, and in order to obtain the vulnerability information more directly and clearly, the other file specifically records the log information of the Error level and contains the keyword FATAL.

3.4 Log Analysis

After FASSFuzzer finishes the vulnerability mining work, in order to submit vulnerability reports to relevant security agencies or Android manufacturers, it is also necessary to reproduce and analyze the vulnerabilities according to the vulnerability mining reports. Ensure that vulnerabilities exist and can be exploited repeatedly by reproducing them. Once the Android system denial of service caused by a null pointer reference occurs, the related exception stack information will be recorded in the system log with Error level information. Therefore, the vulnerability can be analyzed in detail through log information. The Error log file generated by FASSFuzzer is shown in Fig. 4, and a fragment of the complete log information is shown in Fig. 5.

```
E/AndroidRuntime(14141): *** FATAL EXCEPTION IN SYSTEM PROCESS: AudioService
E/AndroidRuntime(14141): java.util.EmptyStackException
E/AndroidRuntime(14141):        at java.util.Stack.peek(Stack.java:57)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl.onSetRemoteControlClientBrowsedP
layer(MediaFocusControl.java:1945)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl.access$1500(MediaFocusControl.ja
va:64)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl$MediaEventHandler.handleMessage(
MediaFocusControl.java:387)
E/AndroidRuntime(14141):        at android.os.Handler.dispatchMessage(Handler.java:102)
E/AndroidRuntime(14141):        at android.os.Looper.loop(Looper.java:135)
E/AndroidRuntime(14141):        at android.media.AudioService$AudioSystemThread.run(AudioService.ja
va:3748)
I/Process (14141): Sending signal. PID: 14141 SIG: 9
```

Fig. 4. Exception stack information in the Error log

```
E/AndroidRuntime(14141): *** FATAL EXCEPTION IN SYSTEM PROCESS: AudioService
E/AndroidRuntime(14141): java.util.EmptyStackException
E/AndroidRuntime(14141):        at java.util.Stack.peek(Stack.java:57)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl.onSetRemoteControlClientBrowsedP
layer(MediaFocusControl.java:1945)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl.access$1500(MediaFocusControl.ja
va:64)
E/AndroidRuntime(14141):        at android.media.MediaFocusControl$MediaEventHandler.handleMessage(
MediaFocusControl.java:387)
E/AndroidRuntime(14141):        at android.os.Handler.dispatchMessage(Handler.java:102)
E/AndroidRuntime(14141):        at android.os.Looper.loop(Looper.java:135)
E/AndroidRuntime(14141):        at android.media.AudioService$AudioSystemThread.run(AudioService.ja
va:3748)
I/Process (14141): Sending signal. PID: 14141 SIG: 9
I/ServiceManager(  238): service 'wifi' died
I/ServiceManager(  238): service 'wifiscanner' died
I/ServiceManager(  238): service 'rttmanager' died
I/ServiceManager(  238): service 'connectivity' died
I/ServiceManager(  238): service 'servicediscovery' died
I/ServiceManager(  238): service 'updatelock' died
I/ServiceManager(  238): service 'notification' died
I/ServiceManager(  238): service 'devicestoragemonitor' died
I/ServiceManager(  238): service 'location' died
I/ServiceManager(  238): service 'country_detector' died
I/ServiceManager(  238): service 'search' died
I/ServiceManager(  238): service 'dropbox' died
I/ServiceManager(  238): service 'wallpaper' died
W/AudioFlinger(13909): power manager service died !!!
I/ServiceManager(  238): service 'audio' died
I/ServiceManager(  238): service 'DockObserver' died
W/AudioFlinger(13909): power manager service died !!!
```

Fig. 5. Snippet of full log information

Each piece of information in the Error log file corresponds to the exception stack information generated by each vulnerability in the vulnerability mining report in turn. From the exception stack information, the type of the vulnerability and the cause of the exception can be obtained, and the code location of the vulnerability can be located according to the stack traceback information in the log. Fig. 4 corresponds to an empty stack exception (java.util.EmptyStackException) of the Audio system service. A malicious attacker passes empty serialized data as an input parameter to an API of the service, resulting in a stack data structure in the program without data. but still operate on the empty stack, so that an empty stack exception occurs. At the same time, it can be combined with another detailed log file for more specific analysis. From Fig. 5, it can be found that the system_server process with process number 14141 was killed by the signal sent by the system, and all system services accommodated by system_server also died out, eventually leading to Android System reboots.

3.5 FASSFuzzer System Demonstration

FASSFuzzer is deployed in the Ubuntu 16.04 system environment, and its operation manual is shown in Fig. 6, where id corresponds to the device number of the Android device, the device id can be obtained through the adb devices command, service corresponds to the system service to be tested, and Code corresponds to the system service to be tested The method number of the API.

FASSFuzzer supports global testing and local testing. Through the python FASSFuzzer.py --id DeviceId command, all system services of the Android system can be detected for null pointer reference vulnerabilities. After the detection is complete, FASSFuzzer will generate a vulnerability mining report.

Fig. 7 is a report from FASSFuzzer's vulnerability mining on OnePlus phones running Android 5.0. It can be known from the report that FASSFuzzer tested a total of 85 system services and 1556 APIs with a test duration of 2218 seconds, and obtained the system service and API method numbers corresponding to potential vulnerabilities in the system, the telephony.registry of the system There are 14 potential vulnerabilities in 10 system services such as bluetooth_manager and bluetooth_manager. Then testers can test the system locally based on the vulnerability mining report, and test all APIs of a system service through the python FASSFuzzer.py --id DeviceId --service ServiceName command, and pass python FASSFuzzer.py --id DeviceId -- The service ServiceName --code Code command tests the API of a system service. Through partial testing, potential vulnerabilities can be confirmed and some system services with frequent vulnerabilities can be quickly tested.

```
zym@ubuntu:~/project/code$ python FASSFuzzer.py --help
usage: FASSFuzzer.py [-h] [--id ID] [--service SERVICE] [--code CODE]

FASSFuzzer
python FASSFuzzer.py --id DeviceId
python FASSFuzzer.py --id DeviceId --service ServiceName
python FASSFuzzer.py --id DeviceId --service ServiceName --code Code

optional arguments:
  -h, --help          show this help message and exit
  --id ID             the Android Device Id
  --service SERVICE   the system service to be tested
  --code CODE         the code corresponding API to be tested
```

Fig. 6. FASSFuzzer operation manual

```
FASSFuzzer has fuzzing 85 service, 1556 APIs
The potential vulnerabilities are as follows:
telephony.registry ['25']
bluetooth_manager ['2', '4']
power ['17']
package ['78', '79']
notification ['10']
appwidget ['3']
connectivity ['56', '58']
SurfaceFlinger ['11']
trust ['4', '5']
audio ['67']
The Fuzzing time is 2218s
```

Fig. 7. FASSFuzzer's vulnerability mining report

4 Conclusion

Aiming at the denial of service vulnerability caused by the null pointer reference in the Android system service,

this paper designs and implements the FASSFuzzer system. The basic principle, automatic design, vulnerability mining process and log analysis module of FASSFuzzer are introduced in turn. Finally, the use of FASSFuzzer is comprehensively demonstrated. Through FASSFuzzer, global and local tests can be performed on null pointer reference vulnerabilities of system services, thereby helping testers to quickly check for null pointer reference vulnerabilities.

Acknowledgement

The work presented in this paper was partially supported by 2018 National Natural Science Foundation of China (Grant number 61871339), 2020 Science Technology Project of Fujian (2020H6001). and by Key Laboratory of Digital Fujian on IoT Communication, Architecture and Security Technology (Grant number 2010499).

References

[1] Google, Project Zero. https://googleprojectzero.blogspot.com/, (access 14.06.15) .
[2] MindMac, IntentFuzzer. https://github.com/MindMac/IntentFuzzer, (accessed 17.06.01).
[3] R. Sasnauskas, J. Regehr, Intent fuzzer: crafting intents of death, in: Proc. of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), 2014.
[4] M. Zhou, A. Zhou, L. Liu, P. Jia, C. Tan, Mining denial of service vulnerability in Android applications automatically, Journal of Computer Applications 37(11)(2017) 3288-3293.
[5] A. Blanda, MFFA. https://github.com/fuzzing/MFFA (accessed 16.04.02).
[6] ele7enxxh, android-afl. https://github.com/ele7enxxh/android-afl (accessed 17.08.02).
[7] Google, AFL. https://lcamtuf.coredump.cx/afl/ (accessed 20.03.04).
[8] K. Ispoglou, D. Austin, V. Mohan, Fuzzgen: Automatic fuzzer generation, in: Proc. 29th USENIX Security Symposium, 2020.
[9] Google, LibFuzzer. https://llvm.org/docs/LibFuzzer.html (accessed 20.03.04).
[10]Google, Syzkaller. https://github.com/google/syzkaller (accessed 20.01.04).
[11]K. Wang, Y. Zhang, Q. Liu, D. Fan, A fuzzing test for dynamic vulnerability detection on android binder mechanism, in: Proc. 2015 IEEE Conference on Communications and Network Security (CNS). IEEE, 2015.
[12]H. Feng, K.G. Shin, Understanding and defending the Binder attack surface in Android, in: Proc. of the 32nd Annual Conference on Computer Security Applications, 2016.
[13]Y. Shao, J. Ott, O.A. Chen, Z. Qian, Z.M. Mao, Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework, in: Proc. of NDSS, 2016
[14]A.K. Iannillo, R. Natella, D. Cotroneo, C. Nita-Rotaru, Chizpurfle: A gray-box android fuzzer for vendor service customizations, in: Proc. 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2017.
[15]J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, Y. Wang, Exception beyond Exception: Crashing Android System by Trapping in “Uncaught Exception”, in: Proc. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 2017.
[16]L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, M. Yang, Invetter: Locating insecure input validations in android services, in: Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.
[17]D. Cotroneo, A.K. Iannillo, R. Natella, Evolutionary Fuzzing of Android OS Vendor System Services, Empirical Software Engineering 24(6)(2019) 3630-3658.
[18]Z. Zou, A. Zhou, Research on Mining Vulnerability in Android System Services, Modern Computer (13)(2019) 90-95.
[19]B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, J. Zhuge, FANS: Fuzzing Android Native System Services via Automated Interface Analysis, in: Proc. 29th USENIX Security Symposium, 2020.
[20]Y. Xu, Z. Ma, Z. Wang, X. Niu, Y. Yang, Survey of security for Android smart terminal, Journal of Communications 37(6) (2016) 169-184.
[21]Q. Zou, T. Zhang, R. Wu, J. Ma, M. Li, C. Chen, C. Hou, From automation to intelligence: Survey of research on vulnerability discovery techniques, Journal of Tsinghua University (Science and Technology) 58(12)(2018) 1079-1094.
[22]H. Yue, Y. Zhang, W. Wang, Q. Liu, Android Static Taint Analysis of Dynamic Loading and Reflection Mechanism, Journal of Computer Research and Development 54(2)(2017) 313.
[23]Z. Yang, M. Yang, Leakminer: Detect information leakage on android with static taint analysis, in: Proc. 2012 Third World Congress on Software Engineering. IEEE, 2012.
[24]Y. Zhang, Z. Fang, K. Wang, Z. Wang, H. Yue, Q. Liu, Y. He, X. Li, G. Yang, Survey of Android Vulnerability Detection, Journal of Computer Research and Development 52(10)(2015) 2167-2177
[25]L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, P. Liu, System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation, in: Proc. of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017.
[26]N. Mirzaei, S. Malek, C.S. Păsăreanu, N. Esfahani, R. Mahmood, Testing android apps through symbolic execution,

ACM SIGSOFT Software Engineering Notes 37(6)(2012) 1-5
[27]L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, P. Liu, Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation, IEEE Transactions on Mobile Computing 19(12)(2019) 2946-2964.
[28]X. Zhang, Z. Li, Survey of Fuzz Testing Technology, Computer Science 43(5)(2016) 1-8, 26.
[29]P. Godefroid, Fuzzing: Hack, art, and science, Communications of the ACM 63(2)(2020) 70-76
[30]W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, B. Liang, Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery, in: Proc. 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.
[31]G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks, Evaluating fuzz testing, in: Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018.
[32]CVE-2020-2834. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-28345, (accessed 15.03.04).
[33]CVE-2019-9279. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9279, (accessed 19.02.08).
[34]CVE-2019-9430. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9430, (accessed 20.02.08).
[35]CVE-2019-9400. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9400, (accessed 20.02.08).
[36]CVE-2016-3821. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3821, (accessed 16.03.30).