Poster: TaintGrep: A Static Analysis Tool for Detecting Vulnerabilities of Android Apps Supporting User-defined Rules

Ruiguo Yang
Peking University
Beijing, China
yangruiguo@pku.edu.cn

Jiajin Cai
Peking University
Beijing, China
caijiajin@pku.edu.cn

Xinhui Han
Peking University
Beijing, China
hanxinhui@pku.edu.cn

ABSTRACT

In this poster, we present TaintGrep, a novel static analysis approach to detect vulnerabilities of Android applications. This approach combines the advantages of semantic pattern matching and taint analysis to get better accuracy and be able to detect cross-function vulnerabilities. Compared with many traditional tools, TaintGrep does not require the full source code or building environment to analyze. Moreover, it supports users in defining their customized matching rules using their vulnerability mining experience, which makes this approach more flexible and scalable. In the preliminary experiment, we give a detailed analysis of the rules of two typical vulnerabilities: generic DoS and arbitrary file read/write, and have detected 77 0day vulnerabilities with these rules in 16 well-known Android applications.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Static Analysis; Vulnerability Detecting; Android Apps

ACM Reference Format:
Ruiguo Yang, Jiajin Cai, and Xinhui Han. 2022. Poster: TaintGrep: A Static Analysis Tool for Detecting Vulnerabilities of Android Apps Supporting User-defined Rules. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3548606.3563527

1 INTRODUCTION

Most vulnerabilities detected in Android applications belong to logic vulnerabilities, which generally contain two parts: the data-flow information, which tells us how unreliable data (i.e., user input) can be passed into a vulnerable function, and semantic information, which shows how the critical data is cropped, modified, and judged during the flow.

Generally, security researchers need to audit the code manually to find these possibly vulnerable points. However, because of the import of third-party libraries and various infrastructure and logic in Android application design, it will take lots of time to read the code, and the efficiency highly depends on the experience of the specific auditor. As a result, we need an automatic code audition approach to help researchers locate the code segments with logic vulnerabilities.

Existing solutions consist of dynamic analysis and static analysis methods. Dynamic methods [6] like fuzzing have relatively bad performance on code coverage and efficiency. As for static methods, Amandroid [5], DroidSafe [2], OAuthLint[4] are some popular frameworks based on data-flow analysis. These tools accurately track point-to-point data flow, so they perform well in some data-flow vulnerabilities. However, since most of them cannot catch the semantic features of the code, their ability to detect vulnerabilities with complicated semantic information is relatively weak. For instance, when auditing the local denial of service vulnerability, these frameworks would report a huge number of false positive code segments, most of which are wrapped by try-catch and so cannot be attacked. OAuthLint provide a mechanism called "anti-protocols" to catch the logic mistakes, but it's only used in OAuth vulnerabilities.

Some semantic pattern matching tools can also be used to detect vulnerabilities, like CodeQL [1] and Semgrep [3]. CodeQL extracts semantic information in the compile process and stores them in a database, whose relationship and attributes can be queried from the database to detect vulnerabilities. The shortcoming of CodeQL is that it relies on the full source code and the building environment, which is inaccessible to security researchers. Semgrep is a fast, open-source, static analysis tool for finding bugs. It has the advantages of easily understood rules and strong matching ability. However, Semgrep has some weak points. First, Semgrep only supports scanning in a single file, but most logical vulnerabilities in Android are cross-functional, which means that Semgrep is hardly up to the task of vulnerability mining. Second, Semgrep does not support the recognition of Android application components, making it impossible for users to get application components that can accept vulnerable input.

In this poster, we propose a lightweight system called TaintGrep, which has a solid ability to detect cross-function logic vulnerabilities of apps in Android. It analyzes an app's code and manifest files and constructs a call graph. According to the custom rule defined by the user, TaintGrep extracts and filters the expected function call chains using taint tracking and pattern matching. Finally, it reports the remaining call chains to users for further code audition.

In the preliminary experiment, we defined several rules for different vulnerabilities and found 77 0day vulnerabilities in 16 different well-known applications with these rules, and all of the vulnerabilities have been confirmed by related discloure processes. We would present two of the rules in Section 3, and show how they work in Taintgrep and the vulnerabilities they help us find.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3563527

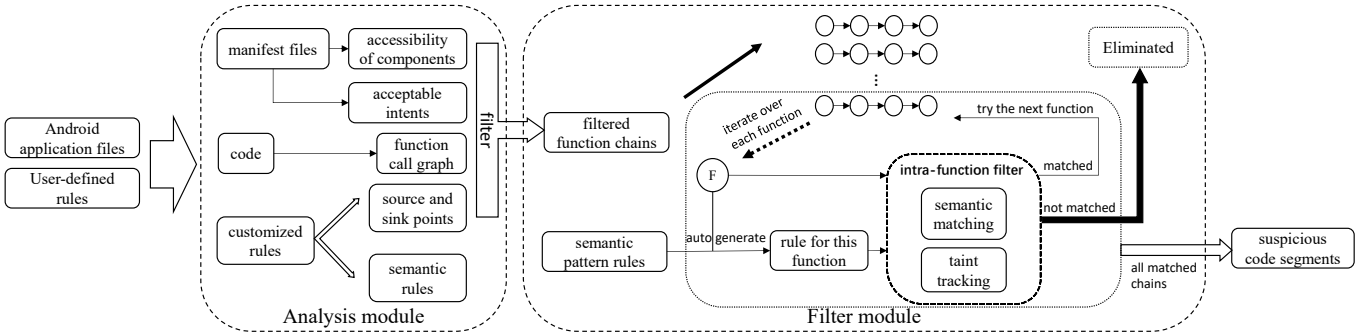

Figure 1: Framework of TaintGrep

2 METHODOLOGY

In this section, we describe the structure of TaintGrep, and briefly introduce the rule used in our system.

2.1 System Design

The framework of our system is shown in Figure 1. It accepts the application's Java code, manifest files, and user-defined rules as input and reports the suspicious code segments as output. For unpacked applications, we can simply decompile the dex files to get (part of) their source code. While for packed applications, we need to unpack the dex files first and then do the decompilation. The system consists of two modules, the analysis module, and the filter module. The two modules cooperate in this system and finally report vulnerable code segments.

The analysis module aims to extract basic information about the application. For the manifest files, the analysis module first extracts the attributes of each component, including *enabled*, *exported*, *name*, *permission* and *intent-filter*. It will analyze the acceptable intents of each component and whether each component is open to other apps based on these attributes. For the application's code, this module extracts the function call graph using jeb. With the function call graph and the accessibility of components, this module will extract multiple function chains satisfying endpoints rules using depth-first search (DFS). These function call chains are passed to the filter module to check if the vulnerability's data flow and pattern information are satisfied.

The filter module can examine whether a function satisfies data-flow information and pattern information. The rules used here are auto-generated by the analysis module from the researchers' custom rules (from cross-function rules to intra-function rules). For each function chain, the matching submodule iterates over each function and tries to match the code with pattern rules. If any function in the function chain does not meet the requirements, this function chain is eliminated. Also, when the taint tracking option is on, the taint tracking submodule will compute if the input of the source function can affect the sink function, and the function chains that do not satisfy the data-flow conditions are also eliminated. Finally, the filtered function chains are handed over to security researchers, who can manually check whether there are some vulnerabilities. In our system, we implement this module based on Semgrep.

2.2 Rules Description

Most vulnerabilities in Android apps have the following features: a source function and a sink function, among which is a function call chain; the attacker can control the input of the source function; there are some improper or incomplete parameter filters on the chain. These features are what we need to consider when we design our rule.

For the first and the second feature, we explicitly define the endpoints of the analysis path and the conditions that both should satisfy, such as their names, intents that they can accept, and whether they are open to the other apps. For the third feature, we describe the vulnerability's data-flow information and pattern information. The data-flow information is used to determine whether the input of the start function can affect the end function, and the pattern information is used to determine whether each function in the function chain from the start function to the end function satisfies the corresponding conditions. More detailed rule examples are shown in the experiment section.

3 PRELIMINARY EXPERIMENT

In the preliminary experiment, we have defined rules for several common Android application vulnerabilities, such as hard coded secret key, arbitrary file read/write/delete/reproduce, generic denial-of-service, and unsafe URL load, and put these rules in TaintGrep to detect vulnerabilities. We will introduce two of the rules in this section and show the results of the vulnerabilities we have found.

3.1 Generic Denial of Service

When an app extracts data from an intent, all data in the intent is automatically deserialized. In this way, the malicious app can send an intent to the victim app, which contains a serialized object defined only within the malicious app. Traditional taint-tracking tools can detect this type of vulnerability, but with an intolerable false positive rate since they cannot identify whether the code segment is wrapped with `try...catch`. Since most Android developers have a good habit of preventing DoS attacks and most possibly vulnerable segments are protected well, the report given by taint-tracking tools would be almost useless.

The rule of Generic DoS vulnerability is shown in Figure 2. We should monitor all possible input components, so the first rule says semi-public components[1]. The endpoint is `getExtras`, which can automatically deserialize the data and cause a crash. The semantic rule means that it does not wrap the code with `try...catch`. With the taint tracking option on, we can ensure the malicious input will affect the `getExtras`.

In total, we found 70 0day generic DoS vulnerabilities from some well-known apps (listed in Table 1), where this vulnerability can make the apps crash and lose login data.

Table 1: Some vulnerabilities detected with TaintGrep

Type of Vulnerability	Applications	Number of Vulnerabilities	Severity[2]
Generic Denial-of-Service	Tencent Video, QQ Mail, QQ Doc, Tencent WeSing, Qzone, WeCom, WeRead, QQ Read, QQ Music, WeChat, Taobao, Tmall	70	Low
Arbitrary File Read/Write	Kugou Music, QQ Sports	2	Low
	Ele.me, Taobao, AMap	4	Medium
	Weibo SDK[3]	1	High

[2] The severity levels were identified by the security departments of corresponding applications.
[3] This SDK is not counted as an application; however, it is integrated into many applications and causes their vulnerabilities. Vulnerabilities of Kugou Music, QQ sports, and Amap shown above are caused by this SDK. Some other applications integrated this sdk are not vulnerable just because of the sdk version.

```
semi-public components ^onCreate$
arbitrary components Landroid/content/Intent;
                ->getExtras()Landroid/os/Bundle;
semantic matching on
semantic rules file rules.txt
taint tracking off
1 getIntent
```

(a) customized rules

```
NOT: |
$TYPE $START(...) {
    ...
    try {
        ...
        $END(...);
        ...
    } catch (...) {
        ...
    }
    ...
}
```

(b) rules.txt

Figure 2: Customized rules for Generic DoS

3.2 Arbitrary File Read/Write

The key point of this vulnerability is the `setResult` function, which takes an intent as the return value and the intent has a flag. When app B sets the flag inside the intent returned to app A, app A will be granted the appropriate privileges. The malicious app uses the intent with read/write permissions to launch the victim app, and the victim app returns this intent directly via `setResult`, causing the malicious app to elevate its permissions.

```
public component ^on.*
arbitrary component ^setResult$
semantic matching on
semantic rules file rules.txt
taint tracking on
2 getIntent $ARG
```

(a) customized rules

```
AND:
-NOT: (Intent $X).setData($Y);
-NOT: (Intent $X).setDataAndNormalize($Y);
-NOT: (Intent $X).setDataAndType($Y, $Z);
-NOT: (Intent $X).setDataAndTypeAndNormalize($Y, $Z);
-NOT: (Intent $X).setFlags($Y);
-NOT: (Intent $X).removeFlags($Y);
```

(b) rules.txt

Figure 3: Customized rules for Arbitrary File Read/Write

The rule of arbitrary file read/write vulnerability is shown in Figure 3. Since the malicious app should control the `data` and `flag` part of the intent, the first rule accepts public components. The semantic rule filters all system functions that may modify the `data` and `flag` fields of the intent. Taint tracking cares about the parameters of `getIntent` and start function.

With TaintGrep, we found 6 arbitrary file read/write vulnerabilities from applications, all of which can lead to arbitrary code execution. Another important discovery is the vulnerability in *Sina Weibo sdk openDefault-11.12.0.aar*. Since every application with "share Weibo" functionality is integrated with this SDK, and as long as the SDK is integrated into the app, malicious apps can read and write arbitrary files of the victim app by utilizing this vulnerability, this vulnerability is of high severity.

Overall, we have found 77 0-day Android application vulnerabilities belonging to these two types with TaintGrep (summarized in Table 1), most of which cannot be easily detected by only taint tracking or semantic pattern matching tools we mentioned in Section 1. These vulnerabilities prove the strong ability of our method in Android vulnerability detection.

4 CONCLUSION

This poster presents TaintGrep, a logic vulnerability detecting system considering both data-flow information and pattern information. By defining custom rules and scanning apps, TaintGrep extracts suspicious function call chains from the apps for the researcher to audit. In the preliminary experiment, we found 77 0day vulnerabilities, some of which are of medium/high severity. In the future, we will test our system in a broader range of applications. Moreover, since there are still some rules that fail to detect vulnerabilities, we will continue to improve the matching ability of our system and refine the design of our rules in the future.

[1]Words like "public", "semi-public" are used to describe the accessibility of components. "public" means that this component accepts arbitrary input from arbitrary applications; "semi-public" means that there are some limits for parameters passed by other applications; "private" means this component can only be accessed by components inside this application.

REFERENCES

[1] GitHub. 2022. *CodeQL*. https://codeql.github.com/
[2] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of android applications in droidsafe.. In *NDSS*, Vol. 15. 110.
[3] r2c. 2022. *Semgrep*. https://semgrep.dev/
[4] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2019. OAUTHLINT: An Empirical Study on OAuth Bugs in Android Applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 293–304. https://doi.org/10.1109/ASE.2019.00036
[5] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.
[6] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia* (Vienna, Austria) *(MoMM '13)*. Association for Computing Machinery, New York, NY, USA, 68–74. https://doi.org/10.1145/2536853.2536881