University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Computer Science

Automatic vulnerability testing in android applications

Supervisor
Prof. Eleonora Losiouk
University of Padova

Master Candidate
Matteo Todescato

StudentID
1207264

Academic Year
2023-2024

"People think that computer science is the art of geniuses, but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."
–Donald Knuth

Abstract

Scanning software for security vulnerabilities can have many applications and can be used in many contexts. Cybersecurity researchers can use it to find vulnerabilities in a software and report them to the developers or for bug bounty programs. Also in the software industry is common practice to scan software to find vulnerabilities before these are exploited by attackers, as security breaches can have serious consequences for the company both financially and in terms of reputation. Unfortunately, the tools of scanning for security vulnerabilities are not perfect, and will never be perfect. Except from few cases most of the tools widely used in the industry for scanning for security issues do a wide use of heuristics and therefore the results will contain false positives and false negatives. For what concerns false negatives, there isn't much that can be done other that trying to improve the scanning tools and their heuristics. False positives on the other hand can be mitigated by the developers manually inspecting the results and deciding if the reported vulnerabilities are real or not. This approach can fail on many levels, for example, not all the developers are security experts, and they might not be able to understand the security implications of the reported vulnerabilities. Moreover, the process of manually inspecting the results of the scanning tools is time-consuming and can be error-prone. We propose a new approach to enhance the result of the scanning tools, by combining the result of the static analysis tools with a dynamic execution of the supposed vulnerable code. Being able to execute the vulnerable code in a controlled environment allow gaining more informations about the vulnerability, and if the vulnerability allows it, set up some kind of test or probe to check it automatically. This approach can help developers and security researchers to differentiate between real vulnerabilities and false positives faster, if not automatically, and also to provide more informations about the found vulnerabilities, as being able to execute the supposed vulnerable code easily can greatly help in the understanding the security implications of the vulnerability. In this way we can speed up the process of finding the real vulnerabilities, fix them, and avoid wasting resources on false positives, which is both expensive and frustrating for the developers.

Contents

Abstract **v**

List of figures **ix**

List of tables **xi**

Listing of acronyms **xiii**

1 Introduction **1**
- 1.1 Static code analysis . . . 2
- 1.2 Static Application Security Testing . . . 4
- 1.3 Real-world usage of SAST . . . 6

2 Requirements and Design **9**
- 2.1 Requirement Analysis . . . 9
- 2.2 Design . . . 11
  - 2.2.1 Speck . . . 12
  - 2.2.2 Gaps . . . 14
  - 2.2.3 Cauldron . . . 14
  - 2.2.4 Vulnerability testing and probes . . . 15

3 Implementation **19**
- 3.1 Speck . . . 19
- 3.2 Cauldron . . . 20
  - 3.2.1 Programming language . . . 20
  - 3.2.2 Serialization and Deserialization . . . 21
  - 3.2.3 Probes execution . . . 24
- 3.3 Probes . . . 25
  - 3.3.1 Rule 1 - Show an app chooser . . . 26
  - 3.3.2 Rule 5 - Use SSL traffic . . . 27
  - 3.3.3 Rule 11 - Store only non-sensitive data in cache files . . . 29
  - 3.3.4 Rule 7 - Use WebView objects carefully . . . 31

4 In depth tools testing **33**
- 4.1 Speck . . . 34
  - 4.1.1 Preliminary Analysis . . . 34

4.2 Gaps . . . 45
4.2.1 Path reconstruction . . . 45
4.2.2 Path execution . . . 50
4.3 Probes executions . . . 51
4.3.1 Rule 1 - Show an app chooser . . . 51
4.3.2 Rule 5 - Use SSL traffic . . . 51
4.3.3 Rule 7 - Use WebView objects carefully . . . 52
4.3.4 Rule 11 - Store only non-sensitive data in cache files . . . 53

5 Mixing everything in our Cauldron **55**
5.1 Speck analysis . . . 55
5.2 Gaps analysis . . . 56
5.3 Cauldron path matching . . . 56

6 Design changes after testing **61**
6.1 New pipeline output . . . 62

7 Real-world applications **65**
7.1 Security researcher . . . 65
7.2 Enterprise . . . 66

8 Future Work and Possible Improvements **69**
8.1 Gaps Path Reconstruction . . . 69
8.2 Adding ML and AI Techniques . . . 70

9 Conclusion **71**

References **73**

Acknowledgments **77**

Listing of figures

1.1 Soundness of static analysis 4
1.2 Static Application Security Testing magic quadrant 2023 5
1.3 Secure Software Development Life Cycle 6

2.1 Design of the tool pipeline 11

4.1 Vulnerable function code 48
4.2 Vulnerable function usages 49
4.3 6676667498cb71d5fcce3170 vulnerability path 49
4.4 6676666198cb71d5fcce30dd vulnerability path 50
4.5 6676663198cb71d5fcce3091 vulnerability path 50

6.1 Vulnerabilities priority 63

Listing of tables

4.1 Path reconstruct analysis 47

5.1 Speck vulnerabilities 58
5.2 AliExpress matched vulnerabilities 59
5.3 Coinbase Wallet matched vulnerabilities 60

Listing of acronyms

SAST ..... Static Application Security Testing

SDLC ..... Software Development Life Cycle

VA ..... Vulnerability Assessment

PT ..... Penetration Testing

SSDLC ..... Secure Software Development Life Cycle

GAPS ..... Graph-based Automated Path Synthesizer

APK ..... Android Package Kit

SSL ..... Secure Socket Layer

HTML ..... HyperText Markup Language

SQL ..... Structured Query Language

SQLi ..... Structured Query Language Injection

CA ..... Certificate Authority

OS ..... Operating System

API ..... Application Programming Interface

URL ..... Uniform Resource Locator

HTTP ..... HyperText Transfer Protocol

HTTPS ..... HyperText Transfer Protocol Secure

AI ..... Artificial Intelligence

ESA ..... European Space Agency

ML ..... Machine Learning

1 Introduction

In the software industry the security of the codebase is a fundamental aspect, this is especially true for the companies that develop software that is widely used by the public and for some specific industries such as:

- safety critical systems;
- financial and banking systems;
- trust service providers;

where in these fields is mandatory to periodically check the security of the codebase. Checking for security vulnerabilities in the software is done in the enterprise world using mainly two methods:

- vulnerability assessments (VA)/ penetration testing (PT);
- vulnerability scanning;

VA/PT is a process done by a third party company that is specialized in security, that searches for vulnerabilities in the company products usually by the means of various scanning tools and or tries to exploit the vulnerabilities. These activities are quite expensive and therefore are executed periodically, the frequency of the activities depends on the company and the industry, but generally they are done

at least once a year. VA/PT are not sufficient to guarantee the security of the software, for this reason alongside with these periodic activities, companies are adopting a more continuous approach to the security of the codebase, with the use of scanning tools that are integrated in the development process. Companies are increasingly a more security oriented SDLC (Software Development Life Cycle) by implementing in their process continuous scanning activities with SAST (Static Application Security Testing) tools. Many reasons can lead a company to add a continuous scanning activities in his development process with a SAST tool, some of them are:

- the company wants to protect its users and its reputation;
- it is required by the clients, especially enterprise ones;

In the following sections we will introduce the base concepts and the state of the art about static code analysis and SAST tools, and we will show how they are used in the Enterprise world.

1.1 Static code analysis

Static code analysis (also called static program analysis or static analysis or static simulation) is a method of analysis of the codebase that is done without executing the code, in contrast with dynamic code analysis that is done by executing the code. The analysis aims to prove certain properties in the code, such as security vulnerabilities or the absence of memory leaks. Apart from trivial properties that are a minority, the properties that are interesting to us such as:

- memory leaks;
- buffer overflows;
- array out of bounds;
- use of uninitialized variables;

are non-trivial and therefore for the Rice's theorem [1] they are undecidable. For this reason it is not possible to build a tool that can prove all the properties exactly,

and therefore some kind of approximation is needed. In the past 60 years the academic community has developed many techniques to approximate non-computable properties [2–8], and many tools have been developed to apply these techniques. A strong commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code [9]. For example, the following industries have identified the use of static code analysis as a means of improving the quality of increasingly sophisticated and complex software, and some of them event made it mandatory in order to meet regulations:

- medical software [10];
- automotive and machines [11];
- nuclear software [12];
- aviation and aerospace software [13];

Static analysis tools are categorized based on the concept of soundness and completeness, creating an opposition between heuristics and formal methods. The concept of soundness is fundamental in static analysis, it defines how the analysis is going to approximate the properties and what kind of warranties we can have on the results. The soundness is related to the way the analyzer is going to cover the spectrum of the results, as we can see in figure 1.1 which take as example a non-computable property such as termination analysis. We can have three possible type of analysis:

- complete: a complete analysis is also sound as it perfectly covers the spectrum of the results, but it is not computable;
- sound: it covers the spectrum of the results partially by being conservative, with the warranty that no false negatives will be present;
- unsound: it covers the spectrum of the results partially without any warranties;

Both sound and unsound tools have their own use cases, as can be seen by the success in the industry of unsound tools such as Coverity [14], also sound tools such as Astree are widely used in safety-critical systems like Airbus, ESA, Bosch [15].

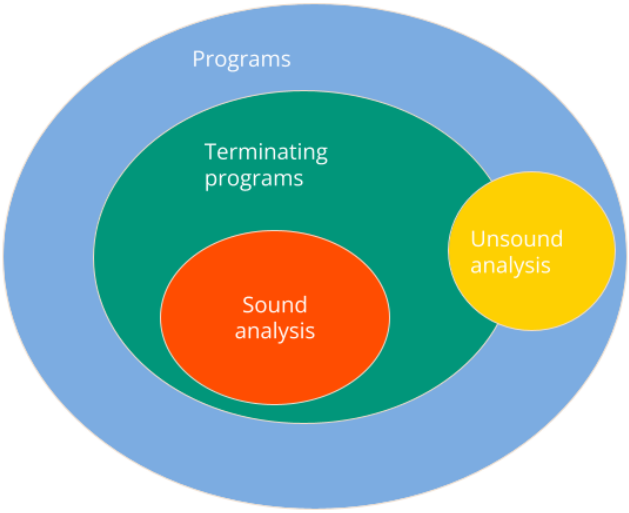

Figure 1.1: Soundness of static analysis

The choice of the type of tool is dictated by the needs of the company and its operation field. For non safety-critical systems the use unsound tools is the most adopted option as they are generally faster, easier to use and less strict allowing the developer to use a wide range of libraries, frameworks and languages with their full set of features. On the other hand safety-critical systems use sound tools as they are stricter, and they can guarantee that the code is free from certain kind of vulnerabilities, which is critical to meet the certification requirements at which safety-critical systems are subject to. In order to have this kind of warranties the sound tools are generally more complex, slower and require more expertise to use, as they are generally based on formal methods, and they require the developer to write the code in a certain way in order to be able to have the best results.

1.2 Static Application Security Testing

The concept of SAST is a special case of a static analysis tool, as in fact it is a static analysis tool that is ”specialized” in finding security vulnerabilities, and does not include any form of dynamic testing of the vulnerabilities. SAST is defined by Gartner [16] as

> ”Static application security testing (SAST) is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities. SAST solutions analyze an application from the “inside out” in a non-running state.”

The use of SAST tools is getting more and more common in the software industry, and companies have many options to choose from as we can see in Gartner quadrant for 2023 1.2. Most of the tools in the quadrant use heuristics to find the

Figure 1.2: Static Application Security Testing magic quadrant 2023

vulnerabilities, the reason behind this is a pragmatic one, the use of heuristics allow extending and improve the tool more early allowing the tool to stay up to date with the latest vulnerabilities and technologies, this kind of flexibility is not possible with sound tools that are based on formal methods. The case of Synopsys with his tools Coverity showed that an unsound approach can do a good job in finding bugs and security vulnerabilities [14], even with the use of heuristics.

1.3 Real-world usage of SAST

Implementing a process of SAST in an Enterprise scenario is a complex task that involves many aspects, from the choice of the tool to the integration in the development pipeline, to the management of the results and the policies that regulate the process. For the scope of this work we will focus on the tools and the integration in the development pipeline, as policies and other process aspects will apply in the same way to our approach. In the enterprise world the use of SAST tools is generally done in conjunction with the application of a SSDLC (Secure Software Development Life Cycle) which comprehends the integration of security as part of the development process, this kind of development process is vastly used and documented in the industry as a best practice [17–21]. An example schema of a process of SSDLC is shown in figure 1.3, showing the integration of the security related activities in the development process. The SAST tool can be run sporadically run

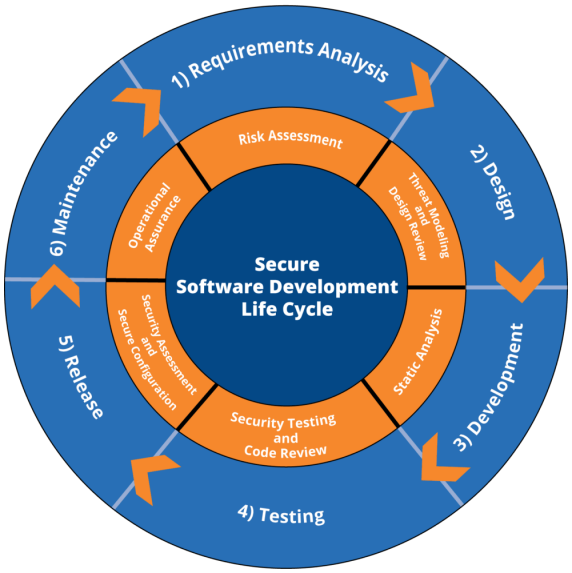

Figure 1.3: Secure Software Development Life Cycle

manually on the codebase but is not recommended and also defies the purpose of having a continuous analysis of the codebase. The most common way to integrate the tool in the process is to run it automatically based on a chosen policy. We can choose between various policies such as:

- on every commit;

- on every merge request;
- on a schedule;

and the choice of the policy will depend on the company and the development process, with no one-size-fits-all solution. Fixing the vulnerabilities is also a part of the process which has two possibilities, for each vulnerability the company can:

- Fix it;
- accept the risk;

The fist option require the vulnerability to be assigned to a team or the developer responsible for the product that will have to manually check the vulnerability and fix it. The process of fixing the vulnerabilities is the most time-consuming part of the process, as the list of vulnerabilities might be long and not all of them are real issues as SAST tool being unsound can have many false positives, so the developers have to manually check each one of them. The process of manually fixing the vulnerabilities has many issues:

- error-prone: generally developers aren't security experts, and they might not be able to understand the real impact of a vulnerability, and might miss some real issues;
- time-consuming: the list of vulnerabilities might be long and the developers have to manually check each one of them, this might take a lot of time. Which is a cost for the company that need to allocate the resources to do this job and not to develop new features;
- frustrating: the developer will generally be searching from a needle in a haystack, as for some tools the ratio of false positives to real vulnerabilities is very high. In the first place at some point the developer will start to check with less care the vulnerabilities and might miss some real issues. In the worst cases might lead to the developer ignoring the tool.

For this reasons having a tool that can help the developer in this process is very important, as it might reduce significantly the time needed to fix the vulnerabilities and the risk of missing some real issues, but most importantly reducing the risk of the developers stopping to care about the tool results.

2

Requirements and Design

We have shown in the previous chapter that most of SAST tools are unsound, and they can generate false positives, for this reason we choose design a tool that can be used to test the vulnerabilities found by the SAST tool. We propose a tool that starting from the vulnerabilities found by the SAST tool can execute a set of probes that can be used to test the vulnerabilities and gain more information about them in order to make easier the job of finding and fixing the real vulnerabilities. In this work we are going to focus on the Android platform, but this concept can theoretically be applied to other platforms as well. This chapter will have two main sections, the first one will be about the requirements of the tool, and the second one will be about the design of the tool.

2.1 Requirement Analysis

The requirements for the tool are the following:

- **SAST tool independence:** The tool should be able to work with any SAST tool that can output results in a machine-readable format.
- **Dynamic execution of vulnerable code:** The tool should be able to execute the vulnerable code in a controlled environment.

- **Automated vulnerability probing:** The tool should be able to automatically probe the vulnerable code for vulnerabilities.
- **Automated vulnerability testing:** The tool should be able to automatically test the vulnerable code for vulnerabilities.
- **Results reporting and bundling:** The tool should be able to report the results and bundle them in a convenient format.

SAST tool independence

The tool should be able to work with any SAST tool that can output results in a machine-readable format, or at least be easy to extend to work with a new SAST tool without the need of rewriting a new tool from scratch, but rather just adding a new module to the existing tool not requiring structural changes to the existing code. This requirement is relevant as a company might already be using one or more SAST tools to find vulnerabilities in their code, and requiring them to change the tools that they are using would require a lot of effort from both the company and the developers.

Dynamic execution of vulnerable code

The tool should be able starting from a vulnerability found by a SAST tool to generate some sort of input that can be used to trigger the vulnerability in a controlled environment.

Automated vulnerability probing

The tool should provide a set of probes that can be used during the dynamic execution phase in order collect data about the behavior of the application and the vulnerability found by the SAST tool, and therefore gain more information about the vulnerability.

Automated vulnerability testing

The tool can provide a set of test probes that can be used to automatically check if the vulnerability found by the SAST tool is real or not. This requirement is

desirable as we now that for some of the vulnerabilities it is not possible to write a generic and automated test.

Results reporting and bundling

The tool should be able to report the results of the testing phase and bundle them in a convenient format, in order to make easier the job of the developer that has to fix the vulnerabilities found by the SAST tool. This can be done by providing some kind of report file or multiple reports one for each vulnerability as opposed to a SAST tool that only need to display minimal information about the vulnerability found, in our case we possibly need to provide more information about the vulnerability and the path that leads to it.

2.2 Design

In order to satisfy the requirements we have identified in the previous section and the principle of not reinventing the wheel, a possible solution is based on a chain of tools that can be used alongside any existing SAST tool. We propose a tool that allow us to orchestrate a process or a pipeline allowing it to be SAST independent, working as shown in figure 2.1. The analysis process is divided in four main steps:

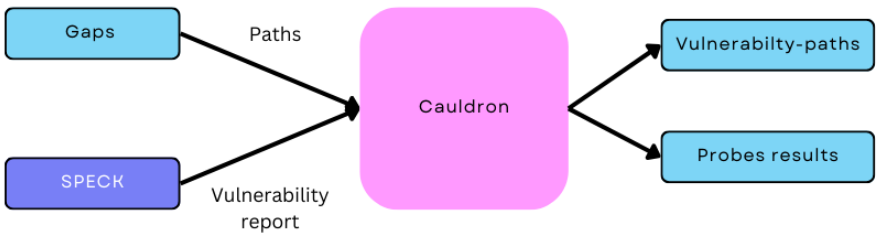

Figure 2.1: Design of the tool pipeline

- scanning: In this step any SAST tool can be used, the only requirement is that the tool should be able to generate a list of potential vulnerabilities; In our case we use Speck 4.1;

- path reconstruction: in this step GAPS 4.2 reconstruct the paths to the vulnerabilities found in the scanning phase, in order to be able to execute them;
- Vulnerability testing: in this phase we use a set of probes or test cases that can be used to test the vulnerabilities found in the scanning phase and executing the reconstructed paths.
- Composing the results: in this phase the results of the scanning phase are composed with the results of the vulnerability testing phase, and execute the path with the probes in order ti produce a list of real vulnerabilities or auxiliary information that can be used to understand the real impact of the vulnerabilities.

2.2.1 Speck

Speck is a static analysis tool designed to search for several bad coding practices on an Android application. It works based on rules extracted from the Android documentation, that developers should follow in order to improve the security of their apps. In particular, for each violated rule, Speck shows the developer the specific line of code where the vulnerability has been detected, thus prompting him to fix the issue. There are 32 rules in total, and they are divided in 6 categories:

- PART 1: https://developer.android.com/topic/security/best-practices
  - Rule1 : Show an App Chooser
  - Rule2 : Control Access to yours Content Providers
  - Rule3 : Provide the right permissions
  - Rule4 : Use intents to defer permissions
  - Rule5 : Use SSL Traffic
  - Rule6 : Use HTML message channels
  - Rule7 : Use WebView objects carefully
  - Rule8 : Store private data within internal storage
  - Rule9 : Share data securely across apps
  - Rule10 : Use scoped directory access
  - Rule11 : Store only non-sensitive data in cache files

Rule12 : Use SharedPreferences in private mode

Rule13 : Keep services and dependencies up-to-date

Rule14 : Check validity of data

- PART 2: https://developer.android.com/training/articles/security-tips

  Rule15 : Avoid custom dangerous permission

  Rule16 : Erase data in webview cache

  Rule17 : Avoid SQL injections

  Rule18 : Prefer explicit intents

  Rule19 : Use IP Networking

  Rule20 : Use services

  Rule21 : Use telephony networking

  Rule22 : Use cryptography

  Rule23 : Use broadcast receivers

  Rule24 : Dynamically load code

- PART 3: https://developer.android.com/training/articles/security-ssl

  Rule25 : Hostname verification

  Rule26 : SSLSocket

- PART 4: https://developer.android.com/training/articles/security-config

  Rule27 : Configure CAs for debugging

  Rule28 : Opt out of clear text traffic

- PART 5: https://developer.android.com/guide/topics/security/cryptography

  Rule29 : Choose a recommended algorithm

  Rule30 : Deprecated cryptographic functionality

- PART 6: https://developer.android.com/training/articles/direct-boot

  Rule31 : Migrate existing data

  Rule32 : Access device encrypted storage

Speck has been chosen as scanning tool because we wanted both to put it at the test and also because if needed we could modify it to better fit our needs in the case some limitations were found. One point of attention on Speck is that its

analysis don't have a context of the application, in fact it only scans the code of the application, and it doesn't take into account the paths that are reachable from the entry point of the application. This generates a good amount of issues on library code even if that code is not reachable from the entry point of the application, as it is not used by the application.

2.2.2 Gaps

Gaps is a tool designed to provide a reliable method for achieving automated interaction with mobile applications. This property is useful for our purpose as we have many target methods which we want to execute in order to test the vulnerabilities found by Speck. Gaps have two separate components, a static and a dynamic one. The static component starting form an APK is able to extract the paths that leads to a specific method, if the method is reachable from the entry point of the application. The static component will output a list of paths and actions that can be used to reach a specific method, allowing us to execute the code without the need of a manual path reconstruction and interaction with the application. The dynamic component is an executor of the output of the static component, and is able to execute the paths extracted by the static component, and check if the path has been reached correctly.

2.2.3 Cauldron

The responsibility of this component is to compose the results of all the previous steps in order to provide a simple and clear output to the developer. Cauldron has many tasks:

- serve as adapter between the different tools, as they might have different input and output formats: The aim is to support the input from different SAST tools which is of primary importance, as it is really likely that someone might want to search for vulnerabilities using a different tool and rules; This make also possible to replace Gaps with another tool that can do the same job, but with a different input and output format, but I consider this to be less likely as is more complex to find a tool that can do the same job as Gaps;
- filter the paths that comes from Gaps in order to remove the ones that are not interesting: This let us clear out the vulnerabilities found by Speck in the library code, as in many cases that code is not used.

- Generate the input to provide to the Gaps dynamic component: split it by the different vulnerabilities found by the SAST tool, and then provide the input to the Gaps dynamic component in order to test the vulnerabilities one by one;
- execute the correct probe for each vulnerability: This is the core of the tool, as it is the one that will test the vulnerabilities found by the SAST tool; It will execute the correct probe for each vulnerability and try to reach the path using the Gaps dynamic component;

The output should be a list of report containing all the information about the vulnerability bundled with the information about the path that leads to the vulnerability, and the result of the test.

2.2.4 Vulnerability testing and probes

The structure of this component is really simple, like with unit tests we have a set of probes that can be used to test the vulnerabilities found by the SAST tool. While designing the probes to test the rules we have to take into account two main aspects:

- The probes should be as generic as possible, in order to be able to test all the vulnerabilities of a specific rule with a single probe, if possible otherwise we would like to use a little amount of probes as possible;
- Computability of the rules properties: as some rules have properties too complex to be checked automatically;

Writing generic exploits

Writing generic probes is not mandatory, but is advisable as it will reduce the amount of work needed to be compared to write a lot of custom probes for each vulnerability. Even if having multiple probes for each rule is supported, still the exploit need to be generic enough to be able to test all the vulnerabilities of that type, otherwise the probes will become a test for the specific issue defying the purpose of the tool. Let's take as example an SQL injection which is a really common vulnerability and has many ways to be exploited:

- Error-based SQLi
- Union-based SQLi
- Blind SQLi boolean-based
- Blind SQLi time-based

Each one of this type of SQL injection has its own way to be exploited, but even if we choose to write an exploit for each of the types, still the exploit don't easily work on all the vulnerabilities of that type, this happens as it is related with the structure and the content of the query, and the way the query is executed. As an example the blind SQLi time-based is a really complex vulnerability which can't be tested with an exploit that works for every query. For this reason we need to identify the vulnerabilities that can't be generically tested in our case of the rule used by Speck following rules do not allow for generic exploits:

- Rule 4 - Use intents to defer permissions
- Rule 6 - Do not use WebView JavascriptInterface
- Rule 10 - Use scoped directory access
- Rule 13 - Keep services and dependencies up-to-date
- Rule 14 - Check validity of external storage data
- Rule 16 - Erase data in WebView cache
- Rule 17 - Avoid SQL injections
- Rule 20 - Do not export unprotected services
- Rule 22 - Use secure random number generators for cryptographic keys
- Rule 23 - Protect exported Broadcast Receivers
- Rule 24 - Do not load code dynamically
- Rule 29 - Choose a recommended cryptographic algorithm
- Rule 30 - Do not use deprecated cryptographic functionality

For those vulnerabilities we can only provide a way of executing the path to the vulnerability, but we can't provide a generic exploit.

Approximation of non-computable properties

In order to be able to write the probes we need to understand that this is tightly related to the computability of the rules properties.
In the case of Speck the rules are defined from the android best practices, and a majority of them are not computable, and therefore we are each rule needs to be individually analyzed in order to understand the best type of probe to analyze the vulnerability.

3 Implementation

For what concerns the implementation of the pipeline, the work can be divided in three main parts:

- adapting Speck's output content;
- Cauldron implementation;
- Probes implementation;

Of these components the most challenging and interesting part is the probes' implementation, as it requires to write an exploit or an attacker app that is able to exploit the vulnerabilities that we are testing and be reusable for multiple tests. In the following sections we will show the implementation of the pipeline and the tools that are part of it.

3.1 Speck

The implementation work on Speck is quite limited as it is a complete tool, for the purposes of this work the only adaptation needed was regarding the output of Speck. The first reason is that as described in section 4.1, Speck return the line of code where it finds the rule violation, this precision is useful but can carry over a degree

of imprecision, especially when collaborating with people that run the analysis with different configurations, especially having the same jadx is really important to obtain the same results. The second reason is that Gaps has no concept of line of code, but it works at method level. Given these two reason, and that adding the class and method to the Speck output was not really a complex task it has been adapted in order to output the informations needed by Gaps, but still keeping the line number. This because for some vulnerabilities Speck can find a violation in a file which is not a java class or a Kotlin class, for example a manifest.xml issue, in this case the output of Speck will mark undefined the method name and the class name, and line number will be the only information that we have. For the scope of this work this limitation is not relevant as our aim is to execute the vulnerable code and this can't be located in the manifest file or in a resource file.

3.2 Cauldron

The responsibility of Cauldron is to manage the pipeline for the analysis and on its own is not a complex tool, a little of complexity is added by the fact that the ability to adapt to the different outputs of the tools that are used in the pipeline, but this can be easily achieved with the right design and technology choice. Cauldron serves four main tasks:

- parse the output of Speck and Gaps;
- generate the input for Gaps dynamic component;
- execute the probes;
- generate the report;

In the next subsections we will show the implementation of the internal components of Cauldron, how they interact with each other in order to accomplish these tasks, and the technology choices that have been made.

3.2.1 Programming language

The first choice that we had to make was the language to use for the implementation of Cauldron. Seen its simple nature of parsing, mixing data and trigger scripts, the

first choice was to use Python as it is a script language that is easy to use and handle pretty well this kind of scripting tasks and also Gaps and Speck are written in python and for that reason it would be reasonable to use the same language. During the first development phases the prototype tool was written in python, but as the development went on the output of the tools was changing and the parsing of the new inputs required to rewrite the parsing code and on top of this the fact that we wanted this tool to be easy to support other SAST tools pushed us search for a different solution. As always when choosing programming languages possibilities are many, we needed a language that with top parsing capabilities and that can work with other languages or at least with python easily. Given the importance of the parsing capabilities we evaluated the possibilities in using a functional languages as they generally have good parsing capabilities, compared with C for example. The main problem with using a functional language is that generally they are running on a virtual machine e.g. Haskell, Erlang, Elixir making them unsuitable for running have the kind of system interaction needed and the ones that support native code such as OCaml are a little short in libraries and support, making them not the best choice for this kind of project. After taking out of the game functional languages, the choice in the imperative languages was easy, Rust gives the capabilities to easily run scripts and to work with python efficiently, and for what concerns parsing capabilities Rust has Serde that allow great parsing capabilities, as we will see in more detail in the serialization and deserialization subsection 3.2.2. Rust is also known to be a fast language, often benchmarked against top performers like C and C++, offer a significant performance improvement over python in compute intensive tasks, an example of performance gap can be seen here [22], this is not a big concern for the pipeline as the most time-consuming part is the execution of the probes, but still it is a nice to have feature.

3.2.2 Serialization and Deserialization

The serialization and deserialization of the data is a key feature of the pipeline, as we need to parse the data coming from Speck and Gaps, generate the input for the dynamic component of Gaps and generate the report. As said in the previous subsection Serde has great parsing and decoding/encoding capabilities, as this library allow defining Rust types such as the one see in the listing 3.1 which are the data structures

for decoding and encoding the data coming from Gaps. After defining the data structures these can be used by Serde to parse the data into the object that defined by calling the function `serde_json::from_str::<GapsReport>(&gaps_output)` this will return the object with the type previously defined.

Listing 3.1: Parsing data struct

```
1  use serde::{Deserialize, Serialize};
2  use std::collections::HashMap;
3
4  use super::path_data::PathData;
5
6  #[derive(Serialize, Deserialize, Debug)]
7  #[serde(transparent)]
8  pub struct GapsReport {
9      path_classes: HashMap<String, PathClass>,
10 }
11
12 impl GapsReport {
13     pub fn new(path_classes: HashMap<String, PathClass>) -> Self {
14         GapsReport { path_classes }
15     }
16
17     pub fn get_path_classes(&self) -> &HashMap<String, PathClass> {
18         &self.path_classes
19     }
20 }
21
22 #[derive(Serialize, Deserialize, Debug)]
23 #[serde(transparent)]
24 pub struct PathClass {
25     paths: HashMap<String, PathData>,
26 }
27
28 impl PathClass {
```

```
29      pub fn new(paths: HashMap<String, PathData>) -> Self {
30        PathClass { paths }
31      }
32
33      pub fn get_paths(&self) -> &HashMap<String, PathData> {
34        &self.paths
35      }
36    }
```

For what concerns the serialization the library can be used in the same way to serialize the data into a file or string. For example, we need to assign to each Speck issue the paths that are related to that issue, this can be done by grouping the paths by the issue and then serializing the data into a file, like we can see in listing 3.2 generating the files is easy as is only needed to call the group_gaps_paths_by_speck_issue function that will output a vector of tuples with the filename and the GapsReport object that contains the paths, and then in order to create the executable file we can use the serde_json::to_writer_pretty function that will serialize the object into a file directly.

Listing 3.2: Speck issues and Gaps path group by

```
1  use std::collections::HashMap;
2  use crate::io_types::gaps_report::GapsReport;
3  use crate::io_types::path_data::PathData;
4  use crate::io_types::speck_report::SpeckReport;
5  use crate::io_types::gaps_report::PathClass;
6  use crate::io_types::speck_report::Issue;
7
8  pub fn group_gaps_paths_by_speck_issue(
9    speck: SpeckReport,
10   gaps: GapsReport)
11   -> Vec<(String, GapsReport)> {
12   let issues = speck.get_issues();
13   let mut result: Vec<(String, GapsReport)> = Vec::new();
14   for issue in issues.iter() {
15     let target_vulnerable_method =
```

```
16          encode_vulnerable_method(issue.clone());
17      let target_method_triggers =
18          extract_method_triggers(&target_vulnerable_method, &gaps);
19      if target_method_triggers.is_empty() == false {
20          let save_filename = format!("{}-rule{}.json",
21          issue.get_object_id().get_oid(),
22          issue.get_rule());
23          result.push((save_filename,
24                       GapsReport::new(target_method_triggers)));
25          }
26      }
27    result
28 }
```

3.2.3 Probes execution

The implementation of the probes' execution component is trivial, the example of the implementation of a probe executor can be seen in the listing 3.3. The executor is made of two parts, a rule selector that given an input file select the correct probe to execute, and the probe executor that actually execute the probe. This way of executing the probes is quite simple and allows to easily add new probes to the pipeline, as this can be done only by adding a new match arm to the match statement in the run_trigger function. This solution also provide the feature of allowing to have more than one probe for a single rule, as the probe runner can be adapted to run all the probes that might be needed.

Listing 3.3: Probes executor

```
1  use std::{path::PathBuf, process::Command};
2
3  pub fn run_trigger(filename: PathBuf) -> () {
4      let basename = filename.file_stem().unwrap().to_str().unwrap();
5      let (oid, rule) = basename.split_once("-").unwrap();
6      match rule {
7          "rule5" => {
```

```
8                 rule5(oid.to_string(), filename);
9             }
10            _ => {
11                println!("No probe for rule, skipped {}", rule);
12            }
13        }
14    }
15
16    fn rule5(oid: String, filename: PathBuf) -> () {
17        println!("IssuedId: {}", oid);
18        println!("Running rule5 on {}", filename.to_str().unwrap());
19        let result = Command::new("python")
20            .arg("probes/rule5/rule_tester.py")
21            .arg("--gaps-input")
22            .arg(filename.to_str().unwrap())
23            .arg("--output")
24            .arg(format!(
25                "./reports/{}.txt",
26                filename.file_stem().unwrap().to_str().unwrap()
27            ))
28            .output()
29            .expect("failed to execute process");
30        println!("status: {:?}", result);
31    }
```

3.3 Probes

With no doubt writing the probes is the most challenging part of the pipeline, not necessarily for the complexity of the code, but for the fact that the code needs to be generic enough in order to not need to write a test for each vulnerability. The probes are written in python, for two main reason, firstly writing the probes is a pure scripting task and secondly the majority of the tools that we needed to use and interact with are in python such as:

- Gaps
- Jadx [23]
- Frida [24]
- Androguard [25]
- etc.

Probe scripts can have various forms and complexity based on the rule that we are trying to exploit, in the following subsections we will show the implementation of probes covering the various types of probes types and the expected output.

3.3.1 Rule 1 - Show an app chooser

This rule is from the Google Android security best practices [26], and it states:

> If an implicit intent can launch at least two possible apps on a user's device, explicitly show an app chooser. This interaction strategy allows users to transfer sensitive information to an app that they trust.

The probe for this rule is quite simple and is entirely contained in a single android application. The probing application catches the implicit intent that the vulnerable app sent without the use of the app chooser, and print its content in the log. For this probe is also possible to have an automatic check for the result as our attacker app can predictably output a log with the intent content that it received, which is enough to understand if the probe was successful or not. The data contained in the received intent might not be sensitive and this shall be checked by hand, but since the rule doesn't require the data to be sensitive for the vulnerability to be considered real, but that the app chooser shall be used. This vulnerability has been mitigated by the Android OS itself, as now the OS will show an app chooser if the intent can be handled by more than one app as we will see in section 4, and only applicable to older versions of Android. For this reason only a partial implementation with toy apps has been done, as the probe in not really useful anymore and not working on the test version with Android API 31.

3.3.2 Rule 5 - Use SSL traffic

This rule is from the Google Android security best practices [26], and it states:

> If your app communicates with a web server that has a certificate issued by a well-known, trusted CA, the HTTPS request should be used.

For this probe instead of providing a full exploit like we have showed for rule 1 we have opted for a different approach. In this case we provided a mean to intercept the traffic between the app and the server, and to check if the app is sending clear HTTP traffic, but the way this is implemented don't allow to be used in a real attack scenario. The probe consist of a proxy that allow to intercept the traffic that enters and exits the Android emulator allowing us to capture the traffic and check if the app is sending clear HTTP traffic, as shown in 3.4 the probe is quite simple, and it is based on the mitmproxy [27].

Listing 3.4: Rule 5 probe

```
1  import argparse
2  import subprocess
3  import time
4  import os
5
6  def run_gaps_with_input(gaps_input, apk_path):
7      print('Running␣GAPS␣with␣input:␣' + gaps_input)
8      result = subprocess.run(['python', './probes/gaps_run_old.py',
9      '-i', apk_path, '-instr', gaps_input],
10                         capture_output = True,
11                         text = True)
12     print(result.stdout)
13     print(result.stderr)
14
15
16
17 if __name__ == "__main__":
18     parser = argparse.ArgumentParser()
19     parser.add_argument("--gaps-input", help="gaps␣input")
```

```
20    parser.add_argument("--output", help="output filename")
21    args = parser.parse_args()
22
23    subprocess.run(['adb', 'wait-for-device'])
24    print('Emulator ready')
25
26    # Start proxy
27    f = open(args.output, "w")
28    proc = subprocess.Popen(['mitmdump',
29       '-s',
30       './probes/rule5/proxy_addon.py'], stdout=f)
31
32    run_gaps_with_input(args.gaps_input)
33
34
35    # Stop proxy
36    time.sleep(15)
37    proc.terminate()
```

Similarly to the rule 1 probe, the probe provide an automatic check for the Speck issue, but in the same way we would like to know if also the data that is sent is sensitive or not, and this shall be checked by hand. In order to allow this probe to work properly the device need to have the mitmproxy certificate installed, this allows the application to work properly in the parts where HTTPS is used, otherwise execution is likely compromised by certificate errors in the requests using SSL. In this way also the HTTPS traffic can be seen by the probe, to exclude from the output SSL communications as is a real attack scenario we wrote a mitmproxy proxy add-on, this allows to filter the traffic based on the properties that we are interested in, as we can see in the listing 3.5 the script is quite simple and will only filter out HTTPS connections.

Listing 3.5: Rule 5 proxy addon

```
1  import logging
2
3  class Counter:
```

```
4       def request(self, flow):
5           if flow.request.scheme == "http":
6               logging.info(flow.request.pretty_host)
7               if flow.request.pretty_host == "10.0.2.2":
8                   flow.request.host = "localhost"
9
10      def response(self, flow):
11          if flow.request.scheme == "http":
12              logging.info(flow)
13              logging.info(flow.response)
14              logging.info(flow.response.content)
15
16  addons = [Counter()]
```

The proxy also provide a way to output all the data allowing to bundle together all the connections data and allowing to easily check if the app is sending clear HTTP traffic or not, it is enough to check if the output file is empty, this allows to provide the automatic check for the probe. In order to allow further analysis it also provides the full content of the request and the response allowing a manual inspection of the data that is sent and received to understand if sensitive data is sent in clear HTTP traffic.

3.3.3 Rule 11 - Store only non-sensitive data in cache files

This rule is from the Google Android security best practices [26], and it states:

> To provide quicker access to non-sensitive app data, store it in the device's cache. For caches larger than 1 MB in size, use getExternalCacheDir(); otherwise, use getCacheDir(). Each method provides you with the File object that contains your app's cached data.

This probe is quite similar to the one for the rule 1, as it can be exploited writing an attacker app that is able to read the cache of the victim app. This process is quite simple as the attacker app only need to have the permission to read the cache

files, this can be obtained by adding the permissions to the manifest file as we can see in the listing 3.6.

Listing 3.6: Rule 11 permissions required

```
1 <uses-permission android:name=
2    "android.permission.MANAGE_EXTERNAL_STORAGE"
3    tools:ignore="ScopedStorage" />
4 <uses-permission android:name=
5    "android.permission.WRITE_EXTERNAL_STORAGE" />
6 <uses-permission android:name=
7    "android.permission.READ_EXTERNAL_STORAGE" />
```

Since Android API 24 the permissions to read the cache files are not granted anymore, so this probe is not working on the test version with Android API 31. Informations about the cache files can be found in the official Android documentation [28]. Apart from the mitigations put in place not allowing us to actually test this probe, is fascinating for the computability limitations compared to some other rules. For the previous two rules the requirement was pointing to a practice like "HTTPS traffic shall be used" that has really distinct signals if violated, as this can be easily checked by searching if there are some HTTP packets in the traffic. For this rule the requirement is the non-sensitivity of the data, sensitive data may have infinite forms and encoding making this rule not computable, as data sensitivity is a really complex concept that can't be checked by a program in a definitive way. With this purpose heuristics, artificial intelligence and machine learning can be used in order to try to understand if the data is sensitive or not, but still would be an unsound approximation of the property and therefore will need to be manually checked. In this situation we have two options:

- count as error the fact that the cache file are used: which is a really strict approach and might be appropriate if our scope were a safety critical system as this choice really limits the coding opportunities, and if cache file have been provided by Android it means that they are meant to be used;;
- bundle the fetched cache content and check it by hand: It requires manual work but is more precise and faster that finding the content of the cache by reading the application code;

Given the scope of this work we implemented the probe using the second option, but we ended up not testing it as the mitigations put in place by Android OS don't allow us to test it on the test version with Android API 31.

3.3.4 Rule 7 - Use WebView objects carefully

This rule is from the Google Android security best practices [26], and it states:

> Whenever possible, load only whitelisted content in WebView objects. In other words, the WebView objects in your app shouldn't allow users to navigate to sites that are outside your control.

This Speck rule is violated when the app uses a WebView object to load content that is not whitelisted, this can be exploited by injecting malicious code in the WebView object and execute it. This can also be automatically checked by injecting a simple alert that print to console log, and check if the alert is shown. For this probe instead of starting from a toy example as for the other probes, in this case building a toy application that is vulnerable to this kind of attack is quite complex, for this reason we decided to start from a real vulnerable application. Unfortunately we were unable to find the paths to this kind of vulnerabilities in the test applications, this is described in more detail in the section 4.3.3, for this reason we have not been able to implement this probe.

4

In depth tools testing

During All the design and implementation phases we tested the tools that compose the pipeline extensively in order to understand the best way to use and combining them, and also to be able to understand the quality of the results that we can expect from the composition of the tools in the pipeline. In order to test all the tools and their composition inside the pipeline we choose to analyze some real application from the play store [29], as both Speck and Gaps were not much tested on real world applications but on toy applications. Our goal was to have a deep and detail analysis, for this reason we initially choose to analyze only one application, the AliExpress app [30]. Later on for test purposes we needed to analyze another application as the AliExpress app was missing some classes of vulnerabilities that we wanted to test, so we choose to analyze also the Coinbase Wallet app [31]. In this chapter we will show the results of the testing and analysis on the pipeline components:

- Speck;
- Gaps;
- Probes;

For what concerns Cauldron we have done some testing to ensure that it works properly, but those are not reported as the results are not interesting, this because

differently from the other tools Cauldron is a pretty classical piece of software that only need to respect its specifications and has not gray areas on his results like Speck false positives or Gaps that might miss paths.

4.1 Speck

The testing on Speck is primarily focused on understanding the quality of the results and the effort required to understand the context of the issues found by the tool in order to be able to fix it or to understand if the issue is a false positive. For the testing we proceed as follows, we choose a set of 10 random vulnerabilities found by Speck in the AliExpress app, and we manually analyzed them to understand the context of the issue which for our purposes is the following set of information:

- the method and class that contains the vulnerable code;
- checking the code if the issue real and exploitable;
- check if the method is used by the application;

We only analyzed 10 issues because the effort required to understand the context of the issue is quite high, since the purpose of this work was to try to put in place some kind of automation to reduce this effort, we didn't want to spend too much time on this, but it was important to have some insight on the quality of the results of Speck. In this preliminary analysis we found 2 false positives as the code is not used, which in 10 issues is a 20% false positive rate. For an application like AliExpress that counted 600 vulnerabilities means that 120 of them are false positives. These two conditions proved the reason behind the usage of also a dynamic analysis tool and the development of the probes, to reduce the effort required to understand the context of the issue and to reduce the false positive rate.

4.1.1 Preliminary Analysis

The vulnerabilities are identified by Speck using the issue ID in the form of an oid, in this instance we will substitute the oid with a name that is more human-readable. In this section we will show the results in detail of the preliminary analysis of the 10 vulnerabilities found by Speck in the AliExpress app.

Alpha

Severity	critical
APK	com.alibaba.aliexpresshd
File	BaseCollectionDetailPresenter.java
Line	85
Rule	1

The vulnerable method identified by Speck is the method N0 of the class BaseCollectionDetailPresenter, the method is called only by the method X of the same class. The Method create an implicit intent of type android.intent.action.SEND containing a string as app/user provided data.

```
1  public final void N0(String str) {
2    Intent intent = new Intent("android.intent.action.SEND");
3    intent.setType("text/*");
4    intent.putExtra("android.intent.extra.SUBJECT", this.f10692a);
5    String a2 = UGCURLGenerator.a(this.f10686a, f0(), this.f47916a);
6    if (getHostActivity() != null && !getHostActivity().isFinishing())
7     {
8      intent.putExtra("android.intent.extra.TEXT", str);
9    }
10   ModulesManager.d().b().b(this.f10688a.getActivity(),
11                            intent, a2, this.c);
12   CollectionTrack.l(getPageName(), this.f10686a);
13 }
```

This behavior is compatible with the rule 1 as the application don't use the createChooser method to show the intent to the user. The X function has only two direct callers, but they don't directly disclose what kind of data the string might contain, given the fact that the two callers have more than 50 callers each, the effort required to understand the context of the issue is too high to be worth the effort in this stage of the analysis. Since the content of the intent is a string is not known if the data is sensitive or not, so the issue is marked as a true positive and therefore a

potential vulnerability, but further analysis is needed to understand if vulnerability is truly exploitable.

Beta

Severity	critical
APK	com.alibaba.aliexpresshd
File	UGCPostDetailActivity.java
Line	1651
Rule	1

The application create an intent of the type android.intent.action.SEND in the openShare method without taking any parameters.

```
1  public void openShare() {
2  if (this.f47942k && String.valueOf(this.f10718a)
3                                  .equalsIgnoreCase(this.f47937e)) {
4    AAFToast.c(R.string.in_black_list_tip);
5    return;
6  }
7  if (Constants.SOURCE_LEGACY_ALIEXPRESS.equalsIgnoreCase(f47935h)) {
8    x();
9    return;
10 }
11 String str = this.d;
12 String string = getString(R.string.itao_share_from_hint);
13 String a2 = UrlGenerator.a(String.valueOf(this.f10718a));
14 String format = MessageFormat.format("({1})\n{0}", a2, string);
15 y();
16 Intent intent = new Intent("android.intent.action.SEND");
17 intent.setType("image/*");
18 intent.putExtra("android.intent.extra.SUBJECT", str);
19 intent.putExtra("android.intent.extra.TEXT", format);
20 Uri uri = this.f10719a;
```

```
21 if (uri != null) {
22   intent.putExtra("android.intent.extra.STREAM", uri);
23 }
24 if (Constants.SOURCE_ITAO.equalsIgnoreCase(f47935h)) {
25   ModulesManager.d().e().b(this, intent, a2,
26                            PostDetailHelper.b(this.f10721a));
27   try {
28       ModulesManager.d().e().i(getPage());
29   } catch (Exception e2) {
30     Log.d("UGCPostDetailActivity", e2);
31     }
32 }
33 }
```

The intent has type set to image/* and to it are added three items:

- android.intent.extra.SUBJECT: Takes the value from the member variable d, which is initialized inside the onCreate method using the data from a received intent `this.d = intent.getStringExtra(EXTRA_POST_DESC);`
- android.intent.extra.TEXT:Contains a URL generated via an URLGenerator class which is able from a variable of type Long to generate a URL or in this case a base part of if since the URL is composed by two parts:

  ```
  String string = getString(R.string.itao_share_from_hint);
  String a2 = UrlGenerator.a(String.valueOf(this.f10718a));
  String format = MessageFormat.format("({1})\n{0}", a2, string);
  intent.putExtra("android.intent.extra.TEXT", format);
  ```

- android.intent.extra.STREAM: Takes its value from a member variable with URI type, the name of the variable was not decompiled successfully. The URI is created from a local file containing some king of image.

The intent is sent without using the createChooser method and therefore can possibly leak information to other applications, and more specifically the intent leaks the value of the unique identifier ANDROID_ID. Similarly to Alpha the vulnerability is real but the data that is leaked is not known, so the issue is marked as a true positive and therefore a potential vulnerability, but further analysis is needed to understand if vulnerability is truly exploitable.

Charlie

Severity	critical
APK	com.alibaba.aliexpresshd
File	UgcBasePostFragment.java
Line	250
Rule	1

The vulnerable method q6 crating the implicit intent is called only by one class, UgcVideoPostFragment which call this method inside the onClick() method The intent string contains the title field from the NPDetail class which is supposed to handle the information about posts inside the platform.

```
1  public final void q6(String str, Function0<String> function0) {
2      String str2;
3      NPDetail nPDetail = this.f47872a;
4      if (nPDetail == null) {
5          return;
6      }
7      Intent intent = new Intent("android.intent.action.SEND");
8      intent.setType("text/*");
9      NPDetail f47872a = getF47872a();
10     String str3 = "";
11     if (f47872a != null && (str2 = f47872a.title) != null) {
12         str3 = str2;
13     }
14     intent.putExtra("android.intent.extra.SUBJECT", str3);
15     String a2 = UGCURLGenerator.a(nPDetail.postId, nPDetail.apptype,
16                                   nPDetail.detailStyle);
17     FragmentActivity activity = getActivity();
18     boolean z = false;
19     if (activity != null && !activity.isFinishing()) {
20         z = true;
21     }
```

```
22      if (z) {
23          intent.putExtra("android.intent.extra.TEXT", str);
24      }
25      String invoke = function0.invoke();
26      if (getActivity() == null ||
27          !com.ugc.aaf.base.util.StringUtil.c(invoke)) {
28          return;
29      }
30      AEProxy b = ModulesManager.d().b();
31      FragmentActivity activity2 = getActivity();
32      Intrinsics.checkNotNull(activity2);
33      b.b(activity2, intent, a2, invoke);
34      CollectionTrack.m(getPage(), nPDetail.postId, nPDetail.apptype);
35  }
```

Even thought I'm not familiar with the way posts are made in AliExpress I would say they are public or at worst shared with customer/seller relationship. Given the type of goods sold inside the platform I would say that the title of the post is really unlikely to have interesting information. For these reason the issue is marked a positive as the intent chooser is not used, but it is unlikely exploitable as the data that is leaked are not sensitive.

Delta

Severity	critical
APK	com.alibaba.aliexpresshd
File	webview/export/internal/android/l.java
Line	22
Rule	1

The code containing this issue is part of Chrome or one of his libraries. In fact can be seen that the sole purpose of the method is to create an intent to be sent via a createChooser.

```
1  public final Intent createIntent() {
```

```
2     Intent intent = new Intent("android.intent.action.GET_CONTENT");
3     intent.addCategory("android.intent.category.OPENABLE");
4     if (Build.VERSION.SDK_INT >= 16) {
5           intent.setTypeAndNormalize(getAcceptTypes()[0]);
6     } else {
7           intent.setType(getAcceptTypes()[0]);
8     }
9     return intent;
10 }
```

The sole usage of this method in fact create an intent chooser as it should, so the issue is marked as a false positive.

Echo, Foxtrot and Golf

Severity	critical
APK	com.alibaba.aliexpresshd
File	LinkShareDelegate.java
Line	34
Rule	1

Severity	critical
APK	com.alibaba.aliexpresshd
File	MediaShareDelegate.java
Line	49
Rule	1

Severity	critical
APK	com.alibaba.aliexpresshd
File	TextShareDelegate.java
Line	25
Rule	1

These three vulnerabilities are pretty much identical, method identified by Speck as vulnerable does not use createChoose as defined by the rule. The application calls a custom-made method called ShareContentBuilder.a to choose the right activity as can be seen from the code snippet below.

```
1  public static void a(Activity activity, ShareMessage shareMessage,
2                       ShareContext shareContext, IShareCallback iShareCallback)
3  if (Yp.v(
4     new Object[]{activity, shareMessage, shareContext, iShareCallback},
5     null, "43186", Void.TYPE).y
6     || ParamChecker.b(activity, shareMessage, iShareCallback)) {
7     return;
8  }
9  if (!(shareMessage.getMediaContent() instanceof LinkContent)) {
10   if (iShareCallback != null) {
11       iShareCallback.onShareFailed(null, shareMessage, "-1", null);
12   }
13 } else {
14   if (TextUtils.isEmpty(((LinkContent) shareMessage.getMediaContent())
15     .etLinkUrl())) {
16       if (iShareCallback != null) {
17         iShareCallback.onShareFailed(null, shareMessage, "-1", null);
18         return;
19       }
20       return;
21   }
22   String shortUrl = shareMessage.getShortUrl();
23   if (TextUtils.isEmpty(shortUrl)) {
24       shortUrl = shareMessage.getContentUrl();
25   }
26   String a2 = ShareContentBuilder.a(shortUrl, shareMessage);
27   Intent intent = new Intent("android.intent.action.SEND");
28   intent.setType("text/plain");
29   intent.putExtra("android.intent.extra.SUBJECT", a2);
30   intent.putExtra("android.intent.extra.TEXT", a2);
```

```
31      ShareServiceHelperInner.startShareIntent(activity,
32      ShareUnitManager.b(activity, intent, true),
33      shareMessage, shareContext, iShareCallback);
34  }
35  }
```

For this reason unless the custom-made method is flawed the issue is marked as a false positive.

HOTEL

Severity	critical
APK	com.alibaba.aliexpresshd
File	f/a/i/d/b.java
Line	224
Rule	5

The method create a URL starting form a string value passed as parameter to the method. There is a check if the URL is HTTPS but whether this check fails the call is executed, allowing the use of unprotected HTTP request.

```
1  if (url.getProtocol().equals("https")) {
2    ((HttpsURLConnection) httpURLConnection)
3          .setHostnameVerifier(f37401a);
4    ((HttpsURLConnection) httpURLConnection).
5    setSSLSocketFactory(new TlsSniSocketFactory(DispatchConstants.a()));
6  }
7  if (ALog.g(1)) {
8    ALog.c("awcn.DispatchCore", "amdc␣request.", str2,
9  eaders",␣httpURLConnection.getRequestProperties().toString());
10 }
11 httpURLConnection.getOutputStream()
12 ␣␣.write(Utils.b(map,␣OConstant.UTF_8).getBytes());
13 int␣responseCode␣=␣httpURLConnection.getResponseCode();
```

The f method that send the HTTP request is called only by the method g of the same class, the data used to create the URLs for the request seems to come from a map provided as parameter. The exact way URL are created and request executed further investigation is needed to understand if the vulnerability is actually usable and the data that we might capture useful.

India

Severity	critical
APK	com.alibaba.aliexpresshd
File	com/google/android/gms/measurement/internal/zzeb.java
Line	57
Rule	5

The method identifier vulnerable by Speck clearly shows that is using clear text HTTP as can be seen from the code snippet below.

```
1  public final HttpURLConnection u(URL url) throws IOException {
2  URLConnection openConnection = url.openConnection();
3  if (openConnection instanceof HttpURLConnection) {
4    SSLSocketFactory sSLSocketFactory = this.f65650a;
5    if (sSLSocketFactory != null &&
6        (openConnection instanceof HttpsURLConnection)) {
7         ((HttpsURLConnection) openConnection)
8            .setSSLSocketFactory(sSLSocketFactory);
9    }
10   HttpURLConnection httpURLConnection =
11    (HttpURLConnection) openConnection;
12   httpURLConnection.setDefaultUseCaches(false);
13   httpURLConnection.setConnectTimeout(60000);
14   httpURLConnection.setReadTimeout(61000);
15   httpURLConnection.setInstanceFollowRedirects(false);
16   httpURLConnection.setDoInput(true);
17   return httpURLConnection;
```

```
18 }
19 throw new IOException("Failed to obtain HTTP connection");
20 }
```

The function containing the vulnerable code is only used by a single method which is a Java runnable, but that method is never called.

JULIETT

Severity	critical
APK	com.alibaba.aliexpresshd
File	com/google/android/gms/measurement/internal/zzay.java
Line	57
Rule	5

The method identifier vulnerable by Speck clearly shows that is using clear text HTTP as can be seen from the code snippet below.

```
1  public final HttpURLConnection w(URL url) throws IOException {
2    URLConnection openConnection = url.openConnection();
3    if (openConnection instanceof HttpURLConnection) {
4      SSLSocketFactory sSLSocketFactory = this.f65568a;
5      if (sSLSocketFactory != null &&
6          (openConnection instanceof HttpsURLConnection)) {
7        ((HttpsURLConnection) openConnection)
8        .setSSLSocketFactory(sSLSocketFactory);
9      }
10     HttpURLConnection httpURLConnection =
11       (HttpURLConnection) openConnection;
12     httpURLConnection.setDefaultUseCaches(false);
13     httpURLConnection.setConnectTimeout(60000);
14     httpURLConnection.setReadTimeout(61000);
15     httpURLConnection.setInstanceFollowRedirects(false);
16     httpURLConnection.setDoInput(true);
17     return httpURLConnection;
```

```
18     }
19     throw new IOException("Failed to obtain HTTP connection");
20 }
```

The function w that contains the vulnerable code is never used.

4.2 Gaps

The quality of the results of Gaps is critical for the functioning of the pipeline as if Gaps is not able to reconstruct the path to the vulnerabilities we stop with only the static analysis result. Gaps have been tested on a pool of standard application with good results as we can see here [32] in the reconstructions results is able to reconstruct over the 65% of the applications paths. However, these test are based on a set of not really complex applications and with only the percentage of reconstructed paths as a metric, which in our case is not really as we are not interested in the path in general but only in the paths that lead to the vulnerabilities. For example Gaps might even find 80% of the paths for example but those missing 20% might be the paths that we are interested in. For what concerns path execution Gaps showed to be able to execute the paths, the success rate is more bound with the quality of the paths than the quality of the execution. In the following subsections we will show in detail the results of the testing on the Path reconstruction and Path execution components of Gaps.

4.2.1 Path reconstruction

This in depth analysis with a manual path reconstruction was not planned for the development of this project, but was made necessary by the fact that we were not able to find Gaps paths leading to the vulnerabilities that we were interested in. In this specific case we were interested in the WebView vulnerabilities, as we were testing the Coinbase Wallet app, and we were finding any path leading to WebView vulnerabilities. In fact, we were only able to match 4 vulnerabilities to the paths using Cauldron:

- 66696b45bea0b3be9f8a9640 - rule type n.5;
- 66696b45bea0b3be9f8a9647 - rule type n.5;

- 66696b7ebea0b3be9f8a96b5 - rule type n.18;
- 66696b7ebea0b3be9f8a96b6 - rule type n.18;

Which was somewhat disappointing as we added to the testing app the Coinbase Wallet to specifically target WebView vulnerabilities which is rule n.7 of the Speck rules that were missing in the AliExpress app. We identified 3 possible reasons for why we were not hitting our target methods:

- the methods is never used, and therefore Gaps can't generate a path to it;
- the method is used, but the path is not generated by Gaps;
- Cauldron in failing to match the paths to the vulnerabilities;

For what concerns the last point we manually checked that the paths to the target methods were not present in the Gaps output, so we can exclude this option. We know that Gaps is not able to generate all the paths, but since Gaps in this case was missing all the target paths we decided that a further manual analysis was needed, to have a better understanding on the issue and the time required to manually find the paths. For this test our objective was to understand how hard it is to find a path manually that is not also be found by Gaps, given this scope we choose a set of 20 vulnerabilities and tried to manually reconstruct the paths to the target methods. The results of the manual analysis are shown in the table 4.1, where we can immediately see that existing paths not found by Gaps are not many, and on top of that is not easy to find by hand. More in detail the following 4 examples represent the 4 possible structure of the paths that can be found during a manual reconstruction:

- 6676663198cb71d5fcce3091: complete path;
- 6676666198cb71d5fcce30dd: incomplete path;
- 6676667498cb71d5fcce3170: incomplete path;
- 6676666298cb71d5fcce30ef: incomplete path;

These vulnerabilities are described with more detail in the following subsections, for the issue 6676666298cb71d5fcce30ef we will also take into consideration the code, but for the other we will skip it as the focus is on the path reconstruction, not the quality of the Speck analysis.

Table 4.1: Path reconstruct analysis

application	IssueID	rule	Gaps	Manual
Coinbase	6676666198cb71d5fcce30dd	16	not found	not found
Coinbase	6676667498cb71d5fcce3170	29	not found	not found
Coinbase	6676666298cb71d5fcce30ef	17	not found	not found
Coinbase	6676663298cb71d5fcce30a7	6	not found	not found
Coinbase	6676663298cb71d5fcce30a8	29	not found	not found
Coinbase	6676667598cb71d5fcce31e7	29	not found	not found
Coinbase	6676667598cb71d5fcce31ed	29	not found	not found
Coinbase	6676668498cb71d5fcce3212	32	not found	not found
Coinbase	6676666c98cb71d5fcce312c	22	not found	not found
Coinbase	6676663198cb71d5fcce30a0	5	not found	not found
Coinbase	6676663198cb71d5fcce30a1	5	not found	not found
Coinbase	6676663198cb71d5fcce3091	5	not found	found
Coinbase	6676666998cb71d5fcce310c	18	not found	not found
Coinbase	6676666998cb71d5fcce310d	18	not found	not found
Coinbase	6676666998cb71d5fcce310e	18	not found	not found
Coinbase	6676666998cb71d5fcce310f	18	not found	not found
Coinbase	6676666998cb71d5fcce3110	18	not found	not found
Coinbase	6676666998cb71d5fcce3111	18	not found	not found
Coinbase	6676666998cb71d5fcce3112	18	not found	not found
Coinbase	6676666998cb71d5fcce3113	18	not found	not found

6676666298cb71d5fcce30ef – Unused method

This violation found by Speck is a SQL injection vulnerability, taking a look at the code in figure 4.1 we can clearly see that the code might be vulnerable to a SQL injection attack, as the query is formed by composition of strings. The code might also not be vulnerable as the query could be formed by a constant string or strings not controlled by the user, but in a code review activity this would be flagged as a potential vulnerability and require a proof that this is the only way of doing this. Therefore we can assume this as a vulnerability, in this case the function is small and the codebase not that complex and for these reasons is really easy to understand that the function is never used, as we can see from the figure 4.2, this is not always as easy to see as we will see in the following subsections. Even thought the function can be vulnerable the fact that is never used makes the vulnerability not exploitable, but this can change in the future as the codebase evolves, so this is still potential

```java
@Override // android.content.ContentProvider
public Cursor query(Uri uri, String[] strArr, String str, String[] strArr2, String str2) {
    int i;
    File fileForUri = this.mStrategy.getFileForUri(uri);
    String queryParameter = uri.getQueryParameter(DISPLAYNAME_FIELD);
    if (strArr == null) {
        strArr = COLUMNS;
    }
    String[] strArr3 = new String[strArr.length];
    Object[] objArr = new Object[strArr.length];
    int i2 = 0;
    for (String str3 : strArr) {
        if ("_display_name".equals(str3)) {
            strArr3[i2] = "_display_name";
            i = i2 + 1;
            objArr[i2] = queryParameter == null ? fileForUri.getName() : queryParameter;
        } else if ("_size".equals(str3)) {
            strArr3[i2] = "_size";
            i = i2 + 1;
            objArr[i2] = Long.valueOf(fileForUri.length());
        }
        i2 = i;
    }
    String[] copyOf = copyOf(strArr3, i2);
    Object[] copyOf2 = copyOf(objArr, i2);
    MatrixCursor matrixCursor = new MatrixCursor(copyOf, 1);
    matrixCursor.addRow(copyOf2);
    return matrixCursor;
}
```

Figure 4.1: Vulnerable function code

vulnerability that should be fixed. For this type of cases Gaps correctly does not generate a path allowing us to exclude this kind of vulnerabilities from the analysis.

6676667498cb71d5fcce3170 – Linear incomplete path

Some methods might be used buy some other methods but still not have an executable path, this is the case of the issue 6676667498cb71d5fcce3170, as we can see in figure 4.3 the method is used by other internal methods, but not from the main activity. In this case a manual check of the path is somewhat easy as we only need to follow a linear call stack and at some point we will reach a method that is never called, unfortunately this is not always the case as we will see in the next subsection.

6676666198cb71d5fcce30dd – Branched incomplete path

In some cases the path might be more complex and the method might be used by more than one method, this is the case of the issue 6676666198cb71d5fcce30dd, as we can see in figure 4.4 the method is used by more than one method, and the path is not linear. So we need to follow all the possible branches to understand if the

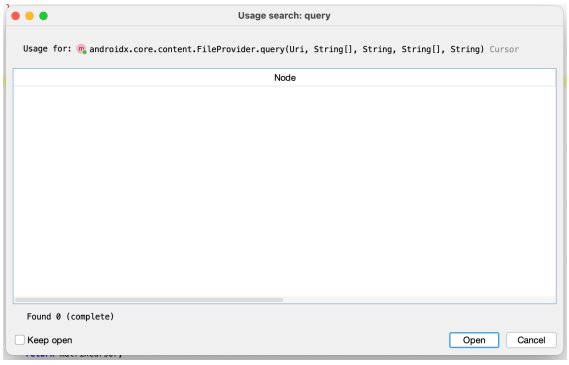

Figure 4.2: Vulnerable function usages

```
com.reactlibrary.auth.crypto.keyStore.CipherGenerator.generateEncryptionCipher(SecretKey) Cipher
com.reactlibrary.auth.crypto.keyStore.Encryptor.encrypt(SecretKey, byte[]) String
com.reactlibrary.auth.crypto.keyhandler.PinKeyHandler.encryptSynchronous(String, String) String
com.reactlibrary.auth.crypto.keyhandler.PinKeyHandler.m685encrypt$lambda0(PinKeyHandler, String, String) String
com.reactlibrary.auth.crypto.keyhandler.PinKeyHandler.encrypt(String, String) Single<String>
com.reactlibrary.CBNativeAuthModule.revertMnemonicMigrationUsingPIN(String, String, Promise) void
```

Figure 4.3: 6676667498cb71d5fcce3170 vulnerability path

method is used or not, in this case the method is not used, but is really easy to get lost in the branches and miss some of them. These are the cases where Gaps make a huge difference in terms of time saved to analyze an issue, as not having to search manually for the path is a huge time saver.

6676663198cb71d5fcce3091 – Complete path

In some cases the path might be complete, this is the case of the issue 6676663198cb71d5fcce3091, as we can see in figure 4.5 the method is used by the main activity, so the path is complete. Here the point is not how easy it is to follow the path but how many paths we need to analyze before being lucky enough to find the one that we are interested in, also this path is not found by Gaps. Even if Gaps lose some paths, we can't compare the time needed to find the path manually to the time needed to find the path with Gaps, as the time needed to find the path manually is not bounded and is based on pure luck, especially if you don't know the application you are working on.

Figure 4.4: 6676666198cb71d5fcce30dd vulnerability path

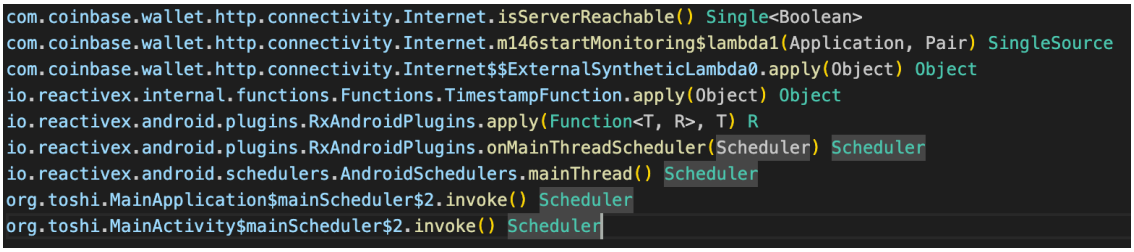

Figure 4.5: 6676663198cb71d5fcce3091 vulnerability path

4.2.2 Path execution

Regarding the Gaps component that execute the path we didn't really investigate on it mainly because apart from some corner cases the paths were executed correctly. For the scope of this project there are only two points of attention regarding the path execution:

- paths that require to be logged in;
- paths in edge execution cases, like direct boot;

The first point is not really a problem as we can log in before the execution of the paths that requires it. It might happen that the login is a part itself of the

path, in this case we can't execute the path as we can't log in before the execution of the path and Gaps can't log in for us, it is more likely that the interesting vulnerabilities are after the login, not the login itself, but should be noted that this limitation exists. The second point is more a limitation in the scope of action of Gaps, as It is designed to interact with the application like a user would do, so it can't trigger the paths that require direct boot mode, in our rule set for example we have the rule 31 and 32 that requires the application to be executed in a direct boot scenario to be triggered, this is not a big limitation as those two rule generate warnings and not critical vulnerabilities, but should be noted that this limitation exists.

4.3 Probes executions

4.3.1 Rule 1 - Show an app chooser

This vulnerability has been mitigated by the Android OS itself, as the intent chooser is always shown to the user, we can see from the android documentation since API 21 the chooser is always shown to the user [33]. Since we are targeting Android API 31 we can not test properly this vulnerability. Anyway since we had already implemented the probe we tested it on a toy application used during the development of the probe, and we can confirm that the probe is working as expected. Still we can't test it on a real application, as is not really relevant as Android 5.0 is not really used anymore.

4.3.2 Rule 5 - Use SSL traffic

We tested this probe on the AliExpress app with the following Speck vulnerabilities:

- 65e5820a0269d04de7e7eb99 - rule5
- 65e5820a0269d04de7e7eb9c - rule5
- 65e5820a0269d04de7e7eb9e - rule5
- 65e5820a0269d04de7e7ebaa - rule5
- 65e5820a0269d04de7e7ebac - rule5

- 65e5820a0269d04de7e7ebb0 - rule5

From which we obtained a report file for each vulnerability in order to check if the data captured are sensitive or not. We will not show the report files as they are too long, and not really interesting to see, as the data captured were not sensitive. But we can say that the probe is working and able to generate a report file with the data captured, these data can be easily read and a developer con understand if the data captured are sensitive or not without a lot of effort and with a good degree of confidence.

4.3.3 Rule 7 - Use WebView objects carefully

This types of vulnerabilities are the reason why we tested also the Coinbase Wallet app, as for the AliExpress app we were not able to find any WebView related vulnerabilities matching the path generated by Gaps. For this reason we searched for new applications to use as a target for this type of vulnerabilities, we initially identify 4 possible applications:

- Quvideo-xiaojun;
- Quvideo-slideplus;
- Mymail
- Coinbase Wallet;

From a preliminary test to understand what application was the most interesting to analyze we found that the Quvideo-xiaojun and Mymail where repeatedly crashing during the execution on the emulator. The Quvideo-slideplus was not crashing but analyzing and app in Chinese is not really easy, if you don't know the language, so we choose to analyze the Coinbase Wallet app as it seemed to be more interesting, and it was easier to understand its functionalities as it is a wallet app. Unfortunately even if we changed the target app to seek for WebView vulnerabilities we were not able to find any matching the paths generated by Gaps, as it was unable to generate the paths leading to the vulnerabilities.

4.3.4 Rule 11 - Store only non-sensitive data in cache files

As written in the implementation section this vulnerability have been mitigated by android itself, as the cache directory is not accessible by other applications, both with a permission schema that only allow the application that created the cache to access it and also with a randomization of the cache directory name. This makes these kinds of vulnerabilities not exploitable, we tested this on a toy application but on the test system we were not able to access the cache directory of the application, so we can't test it on a real application.

5

Mixing everything in our Cauldron

In the previous chapters we showed the results, issues and limitations of the tools, in this chapter we will analyze the results of the composition of the tools. For this purpose we will use the two test applications, the AliExpress and the Coinbase Wallet app. We are not going to show specific example vulnerabilities as we have already shown them in the previous chapter as we want to put the emphasis on the results of the composition of the tools, and give a more general overview of the results and capabilities of the entire pipeline and understand if it meets the requirements that we have set in the introduction.

5.1 Speck analysis

As we can see from the table 5.1 the Speck tool has found a considerable amount of issues in both applications. The AliExpress app has a total of 620 issues and the Coinbase Wallet app has a total of 377 issues. The amount of issues found by Speck is compliant with the expectations, as is common to find a considerable amount of issues when scanning an application with a new static analysis tool.

5.2 Gaps analysis

Expressing the number of paths found by Gaps is not trivial as the paths are grouped by the type of the path, so ideally we should count them separately. For our purpose the class of the path is not relevant and also Cauldron don't distinguish between the type of the path, for these reasons we can just count the total number of paths found by Gaps without grouping them up. The AliExpress app has a total of 47 complete paths and for what concerns the Coinbase Wallet app we have a total of 36 complete paths. Summing up the total number of paths found by Gaps we have a total of 83 paths, which is considerably less than the total number of issues found by Speck.

5.3 Cauldron path matching

During our testing we started the matching phase of our two test applications, the AliExpress and the Coinbase Wallet app with almost a thousand issues found by Speck and less than a hundred paths found by Gaps. Given the prior analysis and testing of Speck is safe to expect that a good majority of the issues will not be matched on the paths found by Gaps, and this is normal with the design and configuration of the tools. It should be noted that starting from an X number of paths doesn't mean that X is the maximum number of vulnerabilities that we can match, as a path can match with one o more vulnerabilities and can also not match with any vulnerability. This is because the matching strategy doesn't match only the landing point of the path, but we will match it even if a vulnerable method is called along the path. Even thought we have many ways of matching the issues with the paths during the test we only matched a very low percentage of the total issues found by Speck, as we can see for AliExpress in table 5.2 we have a total of 29 issues matched and for the Coinbase Wallet app in table 5.3 we have a total of 4 issues matched of which none of the vulnerabilities of the rule 7 that we were interested in.

The test run of the pipeline resulted in a 3.6% issues matched, which is a really low percentage even with the fact that Speck match library code that is not used. This low percentage is generally a bad result, unless we can search for the paths with a tool that find all the paths in the application, in that case we know that

the 3.6% of the issues are the only real issues in the application, and everything else is a false positive. As Gaps don't provide such warranties, as we have seen in the previous chapter, this results suggest a shift in target users and in the way the pipeline should be integrated in a SSDLC process, this will be discussed in the next chapter 6.

Table 5.1: Speck vulnerabilities

Rule	# of issues AliExpress	# of issues Coinbase wallet
rule1	12	11
rule2	0	0
rule3	0	0
rule4	6	3
rule5	56	25
rule6	17	16
rule7	30	16
rule8	0	0
rule9	0	0
rule10	2	2
rule11	35	8
rule12	0	0
rule13	1	0
rule14	7	0
rule15	0	0
rule16	33	18
rule17	9	12
rule18	160	41
rule19	4	1
rule20	25	1
rule21	0	0
rule22	3	4
rule23	11	2
rule24	4	2
rule25	3	2
rule26	0	37
rule27	0	0
rule28	0	1
rule29	184	168
rule30	1	6
rule31	2	0
rule32	15	1

Table 5.2: AliExpress matched vulnerabilities

Rule	# speck issues	# of issues matched
rule1	12	0
rule4	6	0
rule5	56	6
rule6	17	0
rule7	30	0
rule10	2	0
rule11	35	5
rule13	1	0
rule14	7	0
rule16	33	0
rule17	9	0
rule18	160	15
rule19	4	0
rule20	25	0
rule22	3	0
rule23	11	0
rule24	4	0
rule25	3	1
rule29	184	0
rule30	1	0
rule31	2	0
rule32	15	2

Table 5.3: Coinbase Wallet matched vulnerabilities

Rule	# of Speck issues	# of issues matched
rule1	11	0
rule4	3	0
rule5	25	2
rule6	16	0
rule7	16	0
rule10	2	0
rule11	8	0
rule16	18	0
rule17	12	0
rule18	41	2
rule19	1	0
rule20	1	0
rule22	4	0
rule23	2	0
rule24	2	0
rule25	2	0
rule26	37	0
rule28	1	0
rule29	168	0
rule30	6	0
rule32	1	0

6

Design changes after testing

Following the prototype testing is clear that the original design of the pipeline is not feasible, we will discuss the main problems and the new design of the pipeline output.

- Not all the probes needed for the pipeline can be created, and a vast majority of the probes are not automatically testable;
- Gaps matched paths can't be trusted as the sole possible vulnerabilities;

The first point basically undermines the ability of the pipeline to automatically test the vulnerabilities, which would be a really important feature for the pipeline. This was a kind of utopian idea, but it was worth trying to see how many of the probes could automatically check the vulnerabilities. Even without automatic test for the vulnerabilities, still the probes and the paths can be used to gain insights on the vulnerabilities. The second point is more like a strategy problem, because we can't rely on the Gaps generated paths as the only possible vulnerabilities, we still need to check all the vulnerabilities found by Speck. These two points put together, made us reconsider what we wanted to output from the pipeline.

6.1 New pipeline output

Originally the output was a reduction of the output of Speck removing the false positives, given the path misses this filtering is not possible as it would lead to a loss of real vulnerabilities. Whoever is using SAST tool should be used to vulnerabilities categorizations, as tools usually have a way of categorizing the vulnerabilities based on the risk score, the impact of the vulnerability, the ease of exploitation, etc. For this reason we choose to move the scope of the pipeline and use its output to enrich the categorization of the vulnerabilities found by Speck based on the likelihood of the vulnerability to be exploited. We choose to use the pipeline to gain more insight on the vulnerabilities found by Speck, and not to filter them. We propose then to fit the vulnerabilities into a hierarchy of classes in order to quantify the insight about a vulnerability and allow us to prioritize the vulnerabilities based on this as we can see in figure 6.1, the structure as the form of a pyramid as the likelihood of finding automatically testable vulnerability is a lot lower the finding a path for executing a vulnerability. This allows us to generate a report that classifies the vulnerabilities and prioritize them. As Speck doesn't really have a way of classifying the vulnerabilities we didn't try to create a composed risk score, but since most static analysis tools have a way of classifying the vulnerabilities this can be used in conjunction with the pipeline to generate a better risk score, we will discuss the opportunities about this in the possible improvements' section 8.

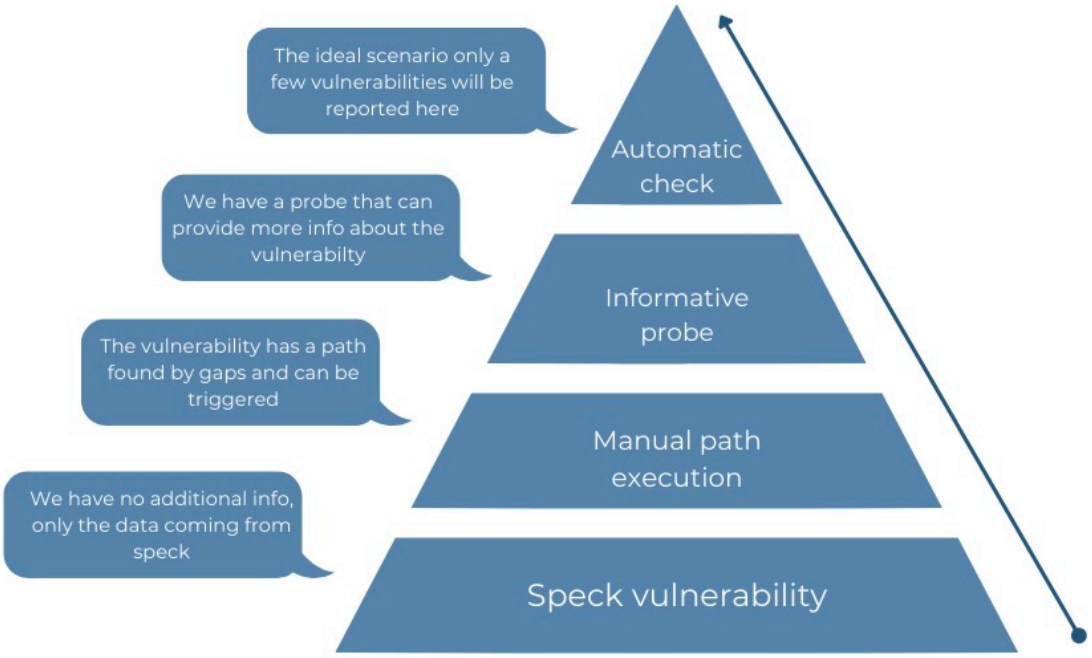

Figure 6.1: Vulnerabilities priority

7 Real-world applications

Given the change in the pipeline objective and output, in this chapter we will discuss the possible applications and target users for the pipeline in the real world. We propose two main target users for the pipeline with their respective applications:

- security researcher;
- Enterprise;

In the following sections we will present the possibilities in applications of the pipeline in these two scenarios.

7.1 Security researcher

The work of the security researcher has many tasks that might benefit from the use of the pipeline. Nowadays, Bug bounty programs [34] are quite popular, the reason behind popularity of these programs is that the companies find really advantageous to have a lot of people looking for bugs in their codebase, and the costing of the program is really low compared to the cost of having a team of security researchers looking for bugs in the codebase. A researcher can claim a bounty by finding a security vulnerability in the codebase of the company that is offering the bounty, the bounty can be quite considerable, for instance Microsoft has a bug bounty

program that can pay up to $250,000 for a single bug [35]. Searching for bugs in very big codebases can be a really demanding task, as the bug can be hidden in a lot of different palaces and if we take as example Google Chrome which has around 32 million lines of code [36], can be easily seen that only searching by patiently reading the code line by line is not a viable option. For these reasons using a scanning tool can give a hit from where to start looking for bugs, but generally the tools are unsound and produce a lot of false positives, this would require to spend a lot of time to analyze the issues found by the tool. The pipeline can be used to filter the issues found by the tool, as in this use case is not relevant to find all the vulnerabilities and not lose any, but to find a vulnerability that can be claimed for a bounty. For this use case the pipeline can offer various advantages:

- false positives are reduced; as the vulnerabilities that are matched on a path are more likely to be real vulnerabilities;
- provides a way to execute the vulnerable code in a convenient way;
- provides probes that can be used to gain more information about the vulnerability;

For these reasons the pipeline can be a really useful tool that can allow the researcher to save time and focus on the most relevant issue allowing to analyze more codebases giving the possibility to find more bugs and claim more bounties.

7.2 Enterprise

Differently from the security researcher perspective, the enterprise perspective is more focused on the general quality and security of the codebase, for this reason SAST and static analysis tools are widely used in the industry. In the enterprise view the pipeline is not used to filter the vulnerabilities as is of primary importance to not lose any vulnerability, but it can be used to gain more insight on the vulnerabilities found by the SAST tool and allow to prioritize the vulnerabilities based on the likelihood of the vulnerability to be exploited. This can allow to fasten the process of choosing which vulnerabilities to fix first and having more information about the vulnerability in conjunction with the ability to execute the vulnerable code can allow to fix the vulnerability faster. For these reasons the pipeline can provide

the mean to the company to reach is goal of making the codebase more secure and of higher quality faster and with less effort. On top of making the process of fixing the vulnerabilities faster and cheaper to the company the pipeline can also be beneficial for the developers that work on the codebase, as the feature provided by our solution can allow them to have a simpler way to execute, analyzed and test the vulnerabilities found by the SAST tool, and also to have less false positives to deal with.

8

Future Work and Possible Improvements

During the evaluation of the system, several possible improvements and future work were identified such as:

- improve Gaps path reconstruction;
- adding machine learning and artificial intelligence techniques;

These improvements are the one that should bring the most significant impact to the system and will be discussed with more details in the following sections.

8.1 Gaps Path Reconstruction

The improvements that could be made to the Gaps' path reconstruction are two. The first one is to use the path reconstruction engine of Gaps to search for specific paths instead of default searching for every possible path, this will both speed up the search and might allow applying more complex rules to the path reconstruction. Secondly, an improvement to the overall percentage of discovered paths would greatly improve the effectiveness of the system as the amount of information that can be gathered from the probes is directly proportional to the amount of paths that are discovered.

8.2 Adding ML and AI Techniques

Applying ML and AI techniques can bring significant benefits to the system especially in the analysis of the results of the probes. The probes are a very powerful tool to gather information about the vulnerabilities of a system, but as we have seen in the previous chapters there are some limitations as some rules are not computable. The way non-computable rules are probed is by generating an output that can be used to help the analyst to understand if the rule is satisfied or not, but this is a time-consuming process. These outputs can be quite verbose for some types of vulnerabilities such as the network related ones, where the network traffic is captured and needs to be analyzed, for this kind of task ML and AI techniques can be very useful as they can be trained to recognize patterns in the network traffic in order to recognize or better approximate the satisfaction of the rule.

9 Conclusion

In this Master thesis, I present a methodology to improve the results of SAST tools and to make them more effective in detecting true vulnerabilities and ease the manual analysis needed to assess and fix the vulnerabilities in Android applications. The solution employs a combination of static and dynamic analysis to gather more information compared to the sole unsound static analysis. We propose Cauldron that allows to compose the results of Speck with the path reconstruction and the path's guided automatic interaction with the application from Gaps. This composition provides a mean to execute the vulnerable code in a controlled environment and set up a system of probes and tests that can allow gathering more information about the vulnerabilities found by static analysis tools. This approach helps the developer in the manual analysis of the vulnerabilities in order to filter out false positives and speed up the process of fixing the vulnerabilities, which is of primary importance as scanning the code is only the first step in the process of securing an application. Ultimately Cauldron was tested against two test applications the AliExpress and Coinbase Wallet applications, the results showed that the approach can be effective but also showed the fragility of the system which dependent on the quality of the results of many tools. In fact is really important to start from a SAST tool with rules and vulnerabilities that are well-defined and not too generic, in order to ease the probe and test creation. It is also very determinant the quality of the path reconstruction, in fact if the paths to the vulnerabilities can't be recon-

structed no automatic interaction with the vulnerable code is possible and therefore no probe can be run, making this approach ineffective. These point emerged from the tests are points that shall be addressed in the future work in order to improve this prototype and make it more reliable and effective and possibly allow it to be used in a production environment.

References

[1] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953. [Online]. Available: http://www.jstor.org/stable/1990888

[2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[3] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *J. ACM*, vol. 23, no. 1, p. 158–171, jan 1976. [Online]. Available: https://doi.org/10.1145/321921.321938

[4] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 194–206. [Online]. Available: https://doi.org/10.1145/512927.512945

[5] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 1990, pp. 126–129.

[6] R. DeMilli and A. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[7] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '79. New York, NY,

USA: Association for Computing Machinery, 1979, p. 269–282. [Online]. Available: https://doi.org/10.1145/567752.567778

[8] A. Miné, "The octagon abstract domain," *Higher-Order Symb Computation*, vol. 19, pp. 31–100, 2006.

[9] B. Livshits. Improving software security with precise static and runtime analysis. [Online]. Available: https://web.archive.org/web/20110605125310/http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/thesis.pdf

[10] *IEC 62304:2006 Medical device software - Software life cycle processes*, IEC Std., may. 2006.

[11] *ISO 26262-1:2018*, ISO Std., dec. 2018.

[12] J. Lahtinen, M. Johansson, J. Ranta, H. Harju, and R. Nevalainen, "Comparison between iec 60880 and iec 61508 for certification purposes in the nuclear domain," in *Computer Safety, Reliability, and Security*, E. Schoitsch, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 55–67.

[13] *DO-178C Software Considerations in Airborne Systems and Equipment Certification*, RTCA, EUROCAE Std., jan. 2012.

[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, feb 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374

[15] astree. Fast and sound static analysis. [Online]. Available: https://www.absint.com/astree/index.htm

[16] Gartner. Sast (static application security testing). [Online]. Available: https://www.gartner.com/en/information-technology/glossary/static-application-security-testing-sast

[17] redhat. Security in the software development lifecycle. [Online]. Available: https://www.redhat.com/en/topics/security/software-development-lifecycle-security

[18] microsoft. Security development lifecycle (sdl) practices. [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl/practices

[19] synopsys. Secure sdlc 101. [Online]. Available: https://www.synopsys.com/blogs/software-security/secure-sdlc.html

[20] owasp. Secure development and integration. [Online]. Available: https://owasp.org/www-project-developer-guide/draft/foundations/secure_development/

[21] D. D. Murugiah Souppaya (NIST), Karen Scarfone (Scarfone Cybersecurity). Secure software development framework (ssdf) version 1.1: Recommendations for mitigating the risk of software vulnerabilities. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf

[22] B. Brace. Speed test : Python vs rust. [Online]. Available: https://dev.to/bekbrace/speed-test-python-vs-rust-1mk4

[23] skylot. jadx - dex to java decompiler. [Online]. Available: https://github.com/skylot/jadx

[24] Frida. Frida. [Online]. Available: https://frida.re/

[25] Androguard. Androguard. [Online]. Available: https://github.com/androguard/androguard

[26] Google. Google android security best practices. [Online]. Available: https://developer.android.com/topic/security/best-practices

[27] mitmproxy. mitmproxy. [Online]. Available: https://mitmproxy.org/

[28] Android-Developers. Data and file storage overview. [Online]. Available: https://developer.android.com/training/data-storage#filesInternal

[29] Google. Google play store. [Online]. Available: https://play.google.com/store/games?hl=en

[30] Alibaba. Aliexpress. [Online]. Available: https://play.google.com/store/search?q=aliexpress&c=apps&hl=en

[31] Coinbase. Coinbase wallet: Nft & crypto. [Online]. Available: https://play.google.com/store/search?q=coinbase+wallet&c=apps&hl=en

[32] S. Doria. Control flow graph-based path reconstruction in android applications. [Online]. Available: https://thesis.unipd.it/handle/20.500.12608/52254?mode=simple

[33] Android-Developers. Intents and intent filters. [Online]. Available: https://developer.android.com/guide/components/intents-filters

[34] wikipedia. Bug bounty program. [Online]. Available: https://en.wikipedia.org/wiki/Bug_bounty_program

[35] Microsoft. Microsoft bug bounty program. [Online]. Available: https://www.microsoft.com/en-us/msrc/bounty

[36] openhub. Chromium (google chrome). [Online]. Available: https://openhub.net/p/chrome/analyses/latest/languages_summary

Acknowledgments