Mobile Security Risks Overview

Ioan ADĂSCĂLIȚEI
Bucharest University of Economic Studies
ioan.adascalitei@ie.ase.ro

Engineers give careful consideration to programming configuration so they give us a smooth and advantageous experience. Individuals readily introduce versatile applications and provide individual data, yet infrequently stop to think about the protection suggestions. As versatile application designers we should be comfortable with conceivable security hazards that a transportable application may confront. Realizing potential dangers makes it simpler to keep up a strategic distance from potential entanglements and compose increasingly secure applications. This paper presents the most important mobile security risks and proposes several counter-measures.
Keywords*: Mobile Vulnerabilities, Code Vulnerabilities, Security Risks, OWASP*
DOI: 10.24818/issn14531305/25.4.2021.06

Introduction

Cell phones have gotten more mainstream than work areas and workstations. Additionally, to the actual fact that they're anything but difficult to convey, however innovative headways have likewise empowered them to perform almost comparable capacities as work areas do. As indicated by Techjury.net, through the span of the foremost recent one-year, portable clients have expanded by quite 10 percent and almost 51 percent of the time spent by clients online within the USA is on cell phones [1]. Clients take part in pretty much all activities on mobile phones, straightforwardly from review the news to perusing messages, messaging, purchasing things on the net, and doing bank trades. Through these applications, associations can gather usable information, for instance, the region, use bits of knowledge, signal, inclinations, loathes, and other significant estimations about clients, which might help associations with picking careful decisions to enliven their organizations. Inside the occasion that the information in these phones enters improper hands, it will in general be disastrous to the client. Thusly, the prerequisite for flexible application security has gotten unavoidable. Flexible application security will be an action to defend applications from external risks like malware and other modernized fakes that hurt fundamental individual and cash related information from software engineers. Compact application security has gotten likewise critical during this current day and age. An entrance in versatile security can't simply give software engineers admittance to the client's own special life logically yet moreover uncover data like their current region, banking information, individual information, and altogether more.

OWASP mobile top 10 security risks

OWASP-(Open Web Application Security Project) is a web network of security authorities that have made uninhibitedly accessible learning materials, documentation and instruments to help work with ensuring web and versatile applications. Among others they need to arrange a rundown of 10 most regular dangers to portable applications.
NowSecure [2] tried applications on the Google Play store and Apple store and discovered that 85% of applications damage at any rate one top 10 hazard. Of these applications half have shaky information stockpiling and nearly an identical number of applications utilize unreliable correspondence.

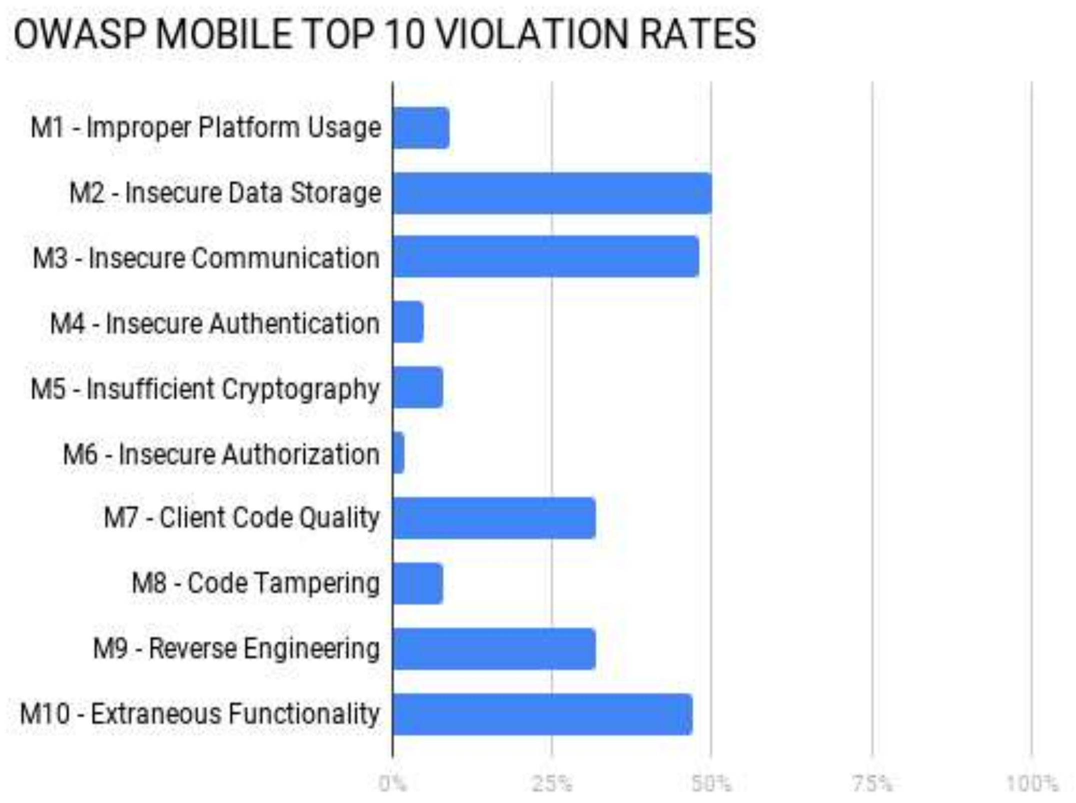


Fig. 1. OWASP TOP 10 Security Risks [2]

MSR1 Improper platform usage
Inability to utilize stage security controls or misuse of a platform feature:

- Android intents
- Misuse of TouchID
- Misuse the Keychain
- Platform permissions
- Misuse of other security controls

Example: on Citrix app you could bypass login through TouchID
It was found, that it had been conceivable to sidestep Touch ID for Citrix Worx applications by [3]:

1. Reboot the iPhone
2. Open Citrix Worx app (any app from them)
3. Start authentication, and then cancel it when the TocuhID popup appear
4. Close the application and open it again and you were connected in the app

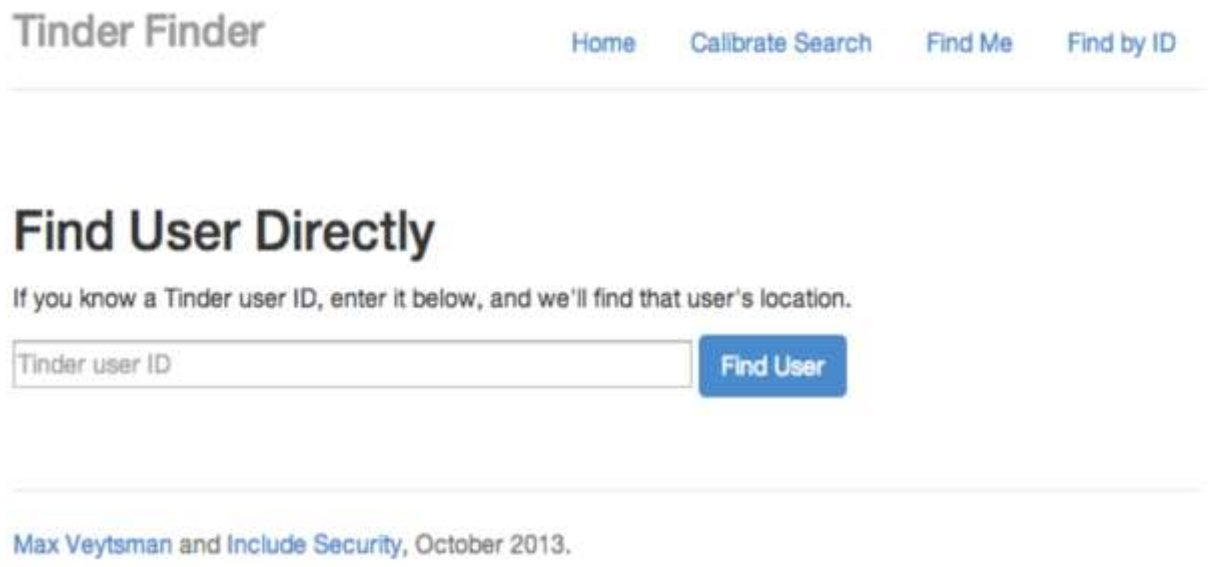
The issue looked as if it would be that the mystery that was recovered by passing Touch ID was put away mistakenly. Henceforth the application expected that the client was effectively confirmed when the confirmation procedure was dropped and application restarted.

MSR2 Insecure data storage
Spreads uncertain information stockpiling and unintended information spillage.
Might include:

- insufficient file data protection (ex. NSFileProtectionComplete vs NSFileProtectionNone)
- wrong keychain accessibility option, (ex. kSecAttrAccessibleAlways vs. kSecAttrAccessibleWhenUnlocked)
- access to protection assets when utilizing this information inaccurately.

Example: Tinder issue
Tinder presented a part that indicated individuals signed on near you. Issue: the precise area of every individual nearby was sent to the gadget. First fix was to provide separation just, however it had been conceivable to parody the world and to utilize triangulation. The next fix was to send this information without accuracy [4]. To show how awful the found vulnerability was an

application was made by clients to point out tinder clients with accurate area.


Tinder Finder

Home Calibrate Search Find Me Find by ID

Find User Directly

If you know a Tinder user ID, enter it below, and we'll find that user's location.

Max Veytsman and Include Security, October 2013.

Fig. 2. Web page that made it possible to track down Tinder users

MSR3 Insecure communication
Can include:
- HTTP instead of HTTPS
- incorrect SSL versions
- cleartext communication of sensitive assets
- poor handshaking/weak negotiation (ex. lack of certificate pinning)

Example: Misafe smart watches
Communication was not encrypted and not effectively confirmed.
According to [6], attackers could:
- make an incognito single direction sound call, keeping an eye on the child
- receive real-time GPS coordinates of the kids' watches
- retrieve a photograph of the kid, in addition to their name, gender, weight, date of birth and stature
- send sound messages to the youngster on the watch, bypassing the endorsed guest list
- call the kid on their watch.

MSR4 Insecure authentication
Issues verifying the end client or awful meeting administration.
Might include:
- weaknesses in session management
- neglecting to recognize the client at all when that ought to be required
- inability to keep up the client's personality when it is required

Example: Grab Android app [7]
The security analyst had the option to sidestep 2FA (two factor authentication) by using brute force on the four-digit code. There was no limitation of how often a 4-digit code could be entered.
Issue: access account with data on rides, installment techniques, orders.

MSR5 Insufficient cryptography
Cryptography was endeavored, however inadequate somehow or another. For instance, designers may have utilized an obsolete cryptographic calculation or composed a custom helpless calculation.
Example: Ola app
Appknox filtered the Ola application and located significant shortcomings in how cryptographic keys were utilized. They found that the cryptographic key utilized was "PRODKEYPRODKEY12". An identical key was likewise accustomed to scramble passwords which suggests that clients' different records where they were reusing passwords may be in peril too. The specialists had the choice to capture demands between the applying and also the server, counterfeit solicitation for cash and find the cash [8].

MSR6 Insecure authorization

Can include:

- failures in authorization (e.g., forced browsing, authorization decisions in the client side, etc.)
- able to execute over-privileged functionality

Example: Viper smart start

A security specialist found that the Viper savvy start neglected to accurately approve clients. After you check in to the server it absolutely was conceivable to vary the id number of the vehicle and obtain entrance additionally to other things in the vehicles area. It absolutely was additionally conceivable to vary information about the vehicle and open the vehicle remotely [9].

MSR7 Client code quality

Catch-all code-level execution issues in the mobile customer.

Might include:

- format string vulnerabilities
- buffer overflows
- many other mistakes regarding the code where the answer is to rewrite some code that's running on the mobile device.

Example: WhatsApp

WhatsApp engineers found that it had been conceivable to form a cushion flood by sending an uncommonly created arrangement of parcels to WhatsApp when making a call. For this to figure the decision should not be replied and therefore the foe can run self-assertive code. It was found that the weakness was utilized to introduce spyware on the device. This service was sold by the Israeli organization NSO Group [10].

MSR8 Code tampering

Might include:

- local resource modification
- binary patching
- dynamic memory modification
- method hooking and swizzling

Example: Pokémon GO

Fans discovered the appliance, took care of wrong geolocation information and time to get uncommon Pokémon and make eggs bring about quicker. A site was made that demonstrated the realm of every Pokémon on a guide, which changed the sport elements a substantial amount. This hack probably won't appear as perilous as the ones above yet it or how Niantic took care of it despite everything cost the organization notoriety and clients.

MSR9 Reverse engineering

Might incorporate investigation of the binary file to decide its

- libraries
- source code
- algorithms and assets

Reverse engineering makes it easy to abuse all kinds of vulnerabilities within the application. It can uncover multiple data about backend servers, cryptographic constants and figures, and licensed innovation.

Example: Many of the examples above used reverse engineering

MSR10 Extraneous functionality

It can include hidden backdoor functionality or other inside improvement security controls not expected for production environment. For example, Wifi File Transfer App opens port on Android gadget to permit associations from the PC.

Expected use: move records, photographs, anything put away on SD card.

Issue: there was no validation like a secret key, anybody could associate with the gadget and have full access.

Ways of early security risks detection

Basically, there are variety of tools and platforms for Static Code Analysis. However, this text aims to introduce a number of them, which are utilized in Android programming as follows:

Programming Mistake Detector (PMD)

PMD may be an open-source code examination apparatus. It observes normal programming defects like unused factors, void catch blocks, superfluous article creation, etc. PMD incorporates worked in rules, and supports the adaptability to record down custom standards besides. It upholds type of dialects especially Java. Also, it includes

CPD, the copy-paste-detector. In a word, CPD finds duplicated code in some languages like Java. the whole features of PMD tool are mentioned as below:

- Possible bugs: Empty switch blocks and try, catch, finally.
- Dead code: All variables, parameters and functions that are left without being used anymore.
- If and while statements left without conditions
- Complicated expressions
- If statements that are useless in for loops that might be while loops.
- Code that is sub-optimal: Incorrect usage for String and StringBuffer classes.
- Classes which have a high level of cyclomatic complexity.
- Code which is duplicated: Copy-paste code can also mean copied-pasted bugs, and reduces maintainability.

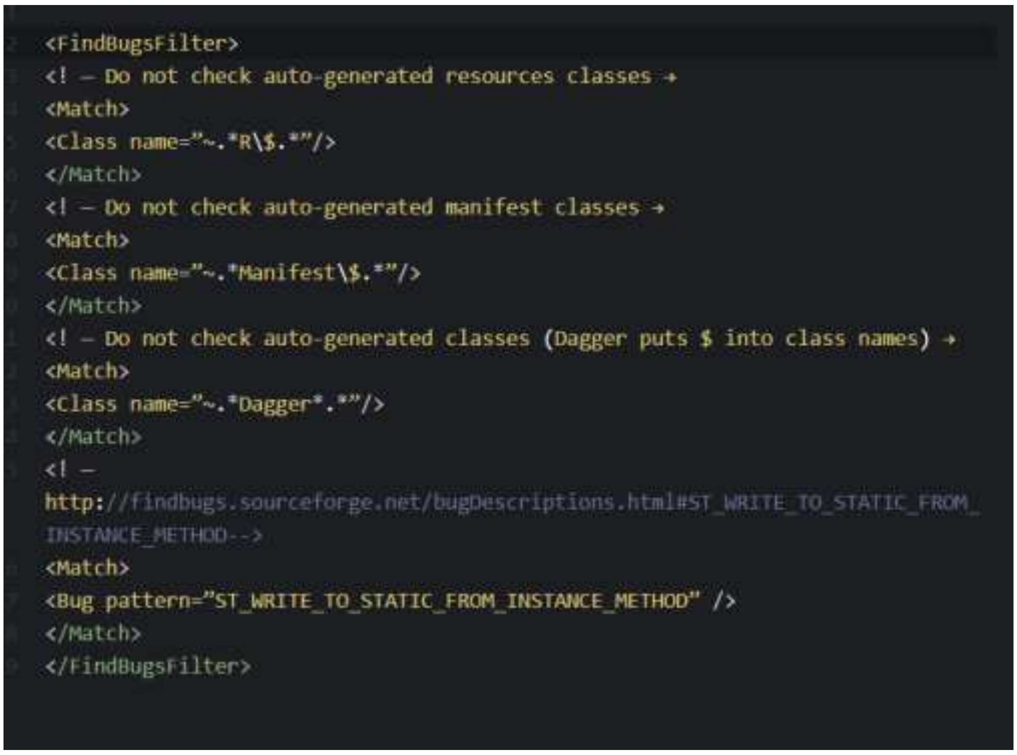
SpotBugs (a fork of FindBugs)

FindBugs is a static code analysis tool, available at https://spotbugs.github.io/, which is also open source, apparatus that examines Java byte-code, and can detect a large number of bugs and issues in an early phase. Some of them are:

- Empty test cases classes.
- Block of codes that are empty need to be deleted
- Synchronize and null check on the same field.
- Class which doesn't proper define some methods like equals() without hashCode().
- Objects created without being used.
- Inconsistent method names.
- Impossible downcast.
- Fields should be package protected.
- Repeated conditional tests.

FindBugs as an open-source project means that it is available for anyone to contribute or screen the advancement of the ASCII text record on GitHub.

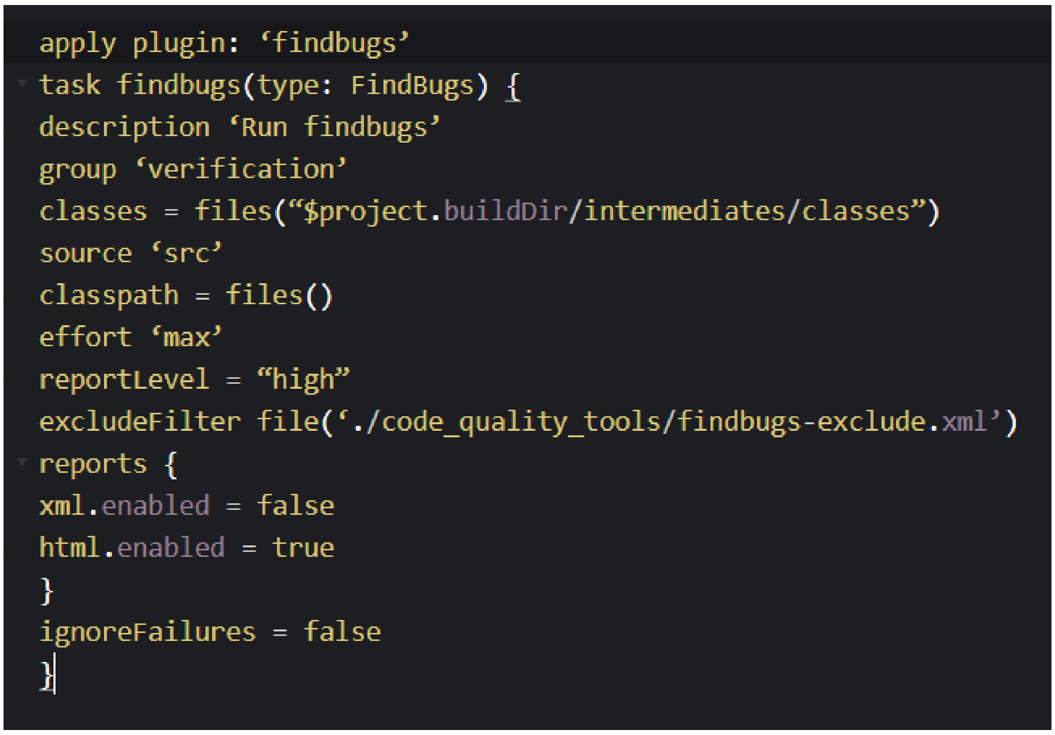
In the **findbugs-exclude.xml** record, we can prevent FindBugs from filtering a few classes (utilizing standard articulations) in projects, as autogenerated asset classes and auto-produced manifest classes. Likewise, on the off chance that Dagger is utilized, the apparatus doesn't need to check the created Dagger classes.



```
<FindBugsFilter>
<! – Do not check auto-generated resources classes →
<Match>
<Class name="~.*R\$.*"/>
</Match>
<! – Do not check auto-generated manifest classes →
<Match>
<Class name="~.*Manifest\$.*"/>
</Match>
<! – Do not check auto-generated classes (Dagger puts $ into class names) →
<Match>
<Class name="~.*Dagger*.*"/>
</Match>
<! –
http://findbugs.sourceforge.net/bugDescriptions.html#ST_WRITE_TO_STATIC_FROM_
INSTANCE_METHOD-->
<Match>
<Bug pattern="ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD" />
</Match>
</FindBugsFilter>
```

Fig. 3. findbugs-exclude.xml [5]

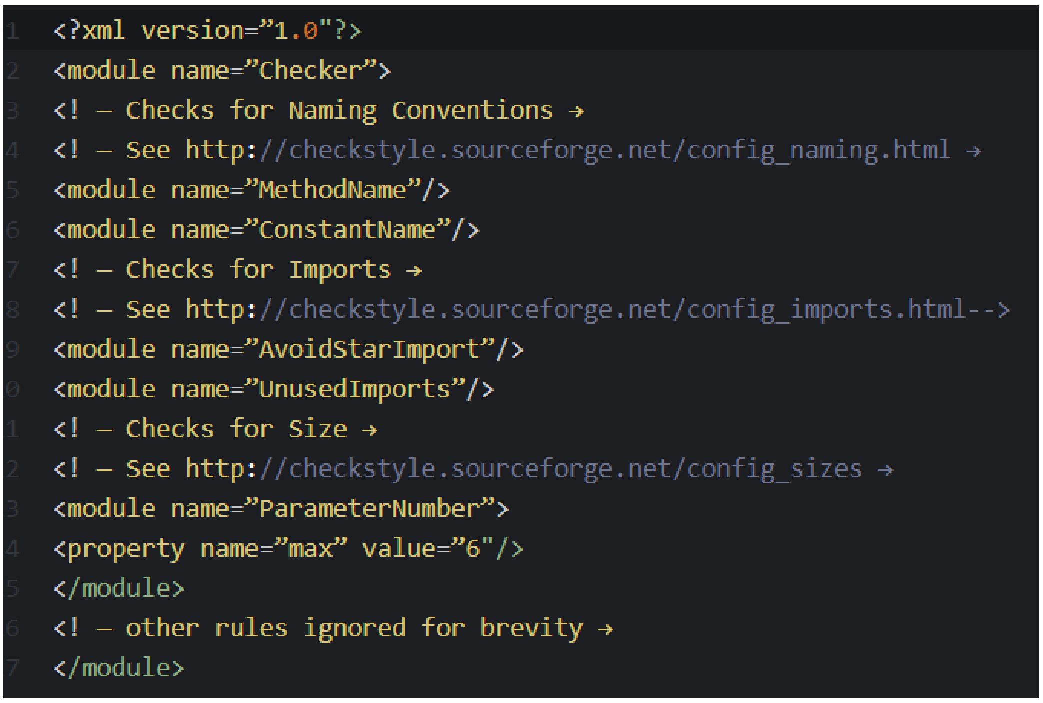
Figure 3 presents an example of findbugs-excllude.xml file. After that, we add it as a gradle plugin, like in figure 4, below.



```
apply plugin: 'findbugs'
task findbugs(type: FindBugs) {
description 'Run findbugs'
group 'verification'
classes = files("$project.buildDir/intermediates/classes")
source 'src'
classpath = files()
effort 'max'
reportLevel = "high"
excludeFilter file('./code_quality_tools/findbugs-exclude.xml')
reports {
xml.enabled = false
html.enabled = true
}
ignoreFailures = false
}
```

Fig. 4. gradle plugin [5]

Checkstyle
Checkstyle as an open source tool can check many aspects of your ASCII text file. It can find class design and method design problems. Furthermore, it's the aptitude to test code layouts and formatting issues. Checkstyle is extremely customizable, and offers support for most of the well-known coding standards like Google and Java Style Sun Code Conventions. In other words, it exists the possibility of mentioning your own rules in an XML file to enforce a custom behavior for the project that is in implementation phase. In fact, Checkstyle enforces those rules by analyzing your ASCII text file, and compares them with accredited standards or conventions.
Integrating Checkstyle in the code requires some configurations, as presented in figure 5.

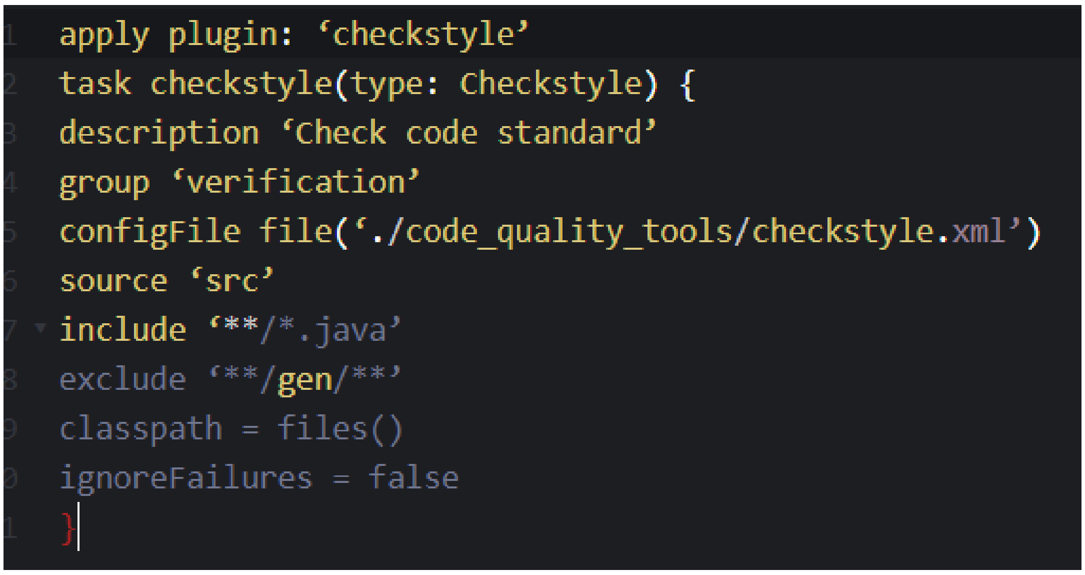


```
<?xml version=”1.0"?>
<module name=”Checker”>
<! – Checks for Naming Conventions →
<! – See http://checkstyle.sourceforge.net/config_naming.html →
<module name=”MethodName”/>
<module name=”ConstantName”/>
<! – Checks for Imports →
<! – See http://checkstyle.sourceforge.net/config_imports.html-->
<module name=”AvoidStarImport”/>
<module name=”UnusedImports”/>
<! – Checks for Size →
<! – See http://checkstyle.sourceforge.net/config_sizes →
<module name=”ParameterNumber”>
<property name=”max” value=”6"/>
</module>
<! – other rules ignored for brevity →
</module>
```

Fig. 5. Configuration example [5]

In the code presented in figure 5, are incorporated the foundations or checks that Checkstyle will validate in the ASCII text document. A very important rule is “AvoidStarImport” which, in light of the fact that the name says, checks assuming your ASCII text document incorporated an import articulation like java.util.*. (All things considered, you should expressly determine the bundle to import, for example java.util.Observable.)

In order to launch this check, it is necessary to create a Gradle task. In the quality.gradle file it’s needed to write a task called checkstyle, see figure 6.



```
apply plugin: ‘checkstyle’
task checkstyle(type: Checkstyle) {
description ‘Check code standard’
group ‘verification’
configFile file(‘./code_quality_tools/checkstyle.xml’)
source ‘src’
include ‘**/*.java’
exclude ‘**/gen/**’
classpath = files()
ignoreFailures = false
}
```

Fig. 6. Gradle task [5]

Notice that within the code above, first is applied the Checkstyle Gradle plugin. It started from a template and afterwards was added as a predefined Gradle bunch called confirmation.

The properties that are a must for a Checkstyle

Gradle task are [5]:

- The file used for configuration -> **configFile**
- **IgnoreFailures**: whether or to not allow the build to continue if there are warnings.
- **include**: the set of patterns.
- **exclude**: the set of excluded patterns in order to not scan unnecessary files, like the ones which are generated.

Lint

The Lint tool checks an Android project source code for potential bugs and optimization improvements for security correctness, usability, accessibility, performance, and internationalization. Lint tool offer the possibility to configure different kind of checks for various levels of the project. As an example: entire project and project module.

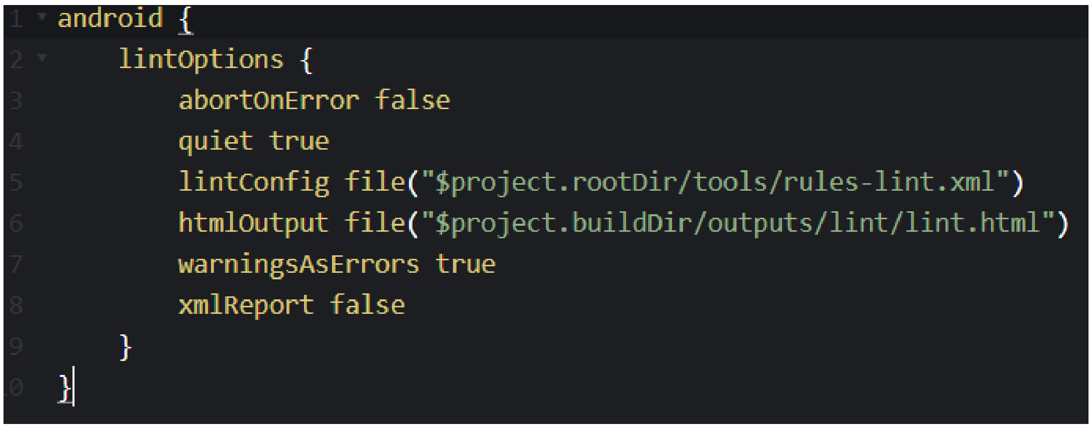
The whole process of lint is classified in three steps:

a. Creating lint.xml file
b. Selecting the ASCII text file for performing analysis by Lint (.java/.kt/XML file)
c. Checking for bugs and suggesting some improvements

The Lint checks can be customized within the lint.xml file. This is where the rules and the ignore checks can be defined. For instance, if you would like to test the unused variables and also don't want to test for naming conventions, you'll accomplish these tasks in lint.xml file (you should make this new go into the foundation directory of your Android project).

In order to configure Lint, the lintOption block is needed to be incorporated in the module-level build.gradle file, figure 7, [5]:

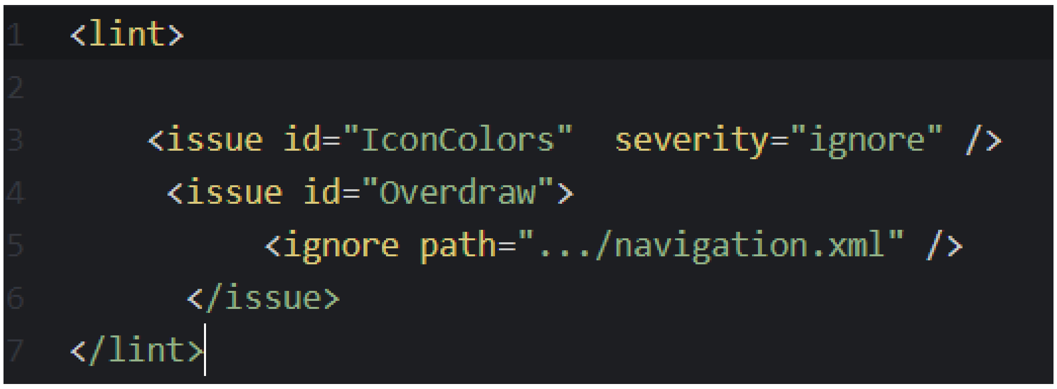
- **lintConfig**: the path to lint rule sets file where you can check issues.
- **htmlOutput**: the path where html report will be generated.
- **abortOnError**: If errors find, Lint should exit the process.
- **quiet**: used to decide if it's needed to turn off or on the analysis progress reporting.



```
android {
    lintOptions {
        abortOnError false
        quiet true
        lintConfig file("$project.rootDir/tools/rules-lint.xml")
        htmlOutput file("$project.buildDir/outputs/lint/lint.html")
        warningsAsErrors true
        xmlReport false
    }
}
```

Fig. 7. Lint configuration example [5]

The lint.xml file can include issues for Lint to ignore or modify. The below example is mentioned that Lint must ignore Icon Colors Check for whole project and ignore the path that is mentioned in the code from figure 8.



```
<lint>

    <issue id="IconColors"  severity="ignore" />
     <issue id="Overdraw">
          <ignore path=".../navigation.xml" />
      </issue>
</lint>
```

Fig. 8. Lint ignore example [5]

Conclusion

The current OWASP portable security top 10 rundown is very well refined and exhaustive. In any case, digital security scene continually changes, portable specifically. Both sides, perpetrators and developers, will generally adjust dangerously fast, and bringing issues to light of a selected issue can imply that more individuals are prepared to manage it soon. In this manner it's amazingly difficult to guess what the Mobile top 10 holds soon. We are going to, without a doubt, see the subsequent one – OWASP as of now accumulates information for it. In any case, the discharge will possibly be in 2021. Concerning 2021 the current rundown is quite significant with no guarantees. Shaky information stockpiling and correspondence also awful coding rehearses and lacking security for information very still and in travel are critical. The entirety of this is often something that we give amazingly close consideration to when creating versatile applications.

As of late, a few certify organizations utilize Static Code Analysis devices also to test and Code Review processes particularly in planning and executing an Android application as a result of the significance of value during this major. During this text the significance of this issue and furthermore a few instruments for Static Code Analysis in Android were thought of. Fundamentally, the point is to go glancing out potential weaknesses like bugs and security defects in an extremely very ASCII report in Android advancement.

Bibliography

[1] OWASP, 2021, https://owasp.org/www-community/Source_Code_Analysis_Tools.

[2] Pavithra Periyasamy, Programming Mistake Detector (PMD), 2021, Available at: https://www.mstsolutions.com/technical/programming-mistake-detector-pmd/

[3] FindBugs - Find Bugs in Java Programs, 2021, Available at: http://findbugs.sourceforge.net/

[4] Kayvan Kaseb, Static Code Analysis in Android, May, 2020, https://medium.com/kayvankaseb/static-code-analysis-in-android-10c3ef83a29a.

[5] Mgbemena, C. "Ensure High-Quality Android Code with Static Analysis Tools," May 2017, Available at: https://code.tutsplus.com/tutorials/ensure-high-quality-android-code-with-static-analysis-tools--cms-28787.

[6] Checkstyle, 2021, Available at: https://checkstyle.sourceforge.io/

[7] Harish Mohan, Achieve High-Quality Android Code with Static Code Analysis Tools, 2018, Available at: https://www.perfomatix.com/android-code-analysis-tools-android-app-development-company/

[8] Google, Improve your code with lint checks, 2021, Available at: https://developer.android.com/studio/write/lint

[9] Cristiano Calcagno et al., "Moving fast with software verification," NASA Formal Methods Symposium, pp. 3-11, 2015.

[10] Linares-Vásquez Mario, Vendome Christopher, Luo Qi, and Poshyvanyk Denys, "How developers detect and fix performance bottlenecks in android apps.," Software Maintenance and Evolution (ICSME), pp. 352-361, 2015

Ioan ADĂSCĂLIȚEI has graduated the Faculty of Cybernetics, Statistics and Economics Informatics in 2018. He holds a bachelor degree in Economics Informatics and and a master's degree in Economics Informatics. He is enrolled in a PhD program with the theme regarding mobile security area. He works as an Android Developer for about six years. He is interested in Mobile Development, including Android and iOS, Mobile Security.