Secrets
ABAP
Apex
C
C++
CloudFormation
COBOL
C#
CSS
Flex
Go
HTML
Java
JavaScript
Kotlin
Kubernetes
Objective C
PHP
PL/I
PL/SQL
Python
RPG
Ruby
Scala
Swift
Terraform
Text
TypeScript
T-SQL
VB.NET
VB6
XML

Kotlin static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your KOTLIN code

All rules 98 | Vulnerability 10 | Bug 17 | Security Hotspot 15 | Code Smell 56

Tags

Search by name...

Hard-coded credentials are security-sensitive
Security Hotspot

Cipher algorithms should be robust
Vulnerability

Encryption algorithms should be used with secure mode and padding scheme
Vulnerability

Server hostnames should be verified during SSL/TLS connections
Vulnerability

Server certificates should be verified during SSL/TLS connections
Vulnerability

Cryptographic keys should be robust
Vulnerability

Weak SSL/TLS protocols should not be used
Vulnerability

"SecureRandom" seeds should not be predictable
Vulnerability

Cipher Block Chaining IVs should be unpredictable
Vulnerability

Hashes should include an unpredictable salt
Vulnerability

Regular expressions should be syntactically valid
Bug

"runFinalizersOnExit" should not be called
Bug

Mobile database encryption keys should not be disclosed

Analyze your code

Vulnerability   Major   cwe owasp android

Storing data locally is a common task for mobile applications. There are many convenient solutions that allow storing data persistently, for example SQLiteDatabase and Realm. These systems can be initialized with a secret key in order to store the data encrypted.

The encryption key is meant to stay secret and should not be hard-coded in the application as it would mean that:

- All user would use the same encryption key.
- The encryption key would be known by anyone who as access to the source code or the application binary code.
- Data stored encrypted in the database would not be protected.

There are different approaches how the key can be provided to encrypt and decrypt the database. One of the most convinient way to is to rely on `EncryptedSharedPreferences` to store encryption keys. It can also be provided dynamically by the user of the application or fetched from a remote server.

Noncompliant Code Example

SQLCipher

```
val key = "gb09ym9ydoolp3w886d0tciczj6ve9kszqd65u7d12604
val db = SQLiteDatabase.openOrCreateDatabase("test.db",
```

Realm

```
val key = "gb09ym9ydoolp3w886d0tciczj6ve9kszqd65u7d12604
val config = RealmConfiguration.Builder()
    .encryptionKey(key.toByteArray()) // Noncompliant
    .build()
val realm = Realm.getInstance(config)
```

Compliant Solution

SQLCipher

```
val db = SQLiteDatabase.openOrCreateDatabase("test.db",
```

Realm

```
val config = RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build()
val realm = Realm.getInstance(config)
```

See

- OWASP Top 10 2021 Category A2 - Cryptographic Failures
- OWASP Top 10 2021 Category A4 - Insecure Design
- Mobile AppSec Verification Standard - Data Storage and Privacy Requirements
- OWASP Mobile Top 10 2016 Category M2 - Insecure Data Storage

"ScheduledThreadPoolExecutor" should not have 0 core threads

Bug

Jump statements should not occur in "finally" blocks

Bug

Using clear-text protocols is security-sensitive

Security Hotspot

Accessing Android external storage is security-sensitive

Security Hotspot

Receiving intents is security-sensitive

Security Hotspot

Broadcasting intents is security-sensitive

Security Hotspot

Using weak hashing algorithms is security-sensitive

Security Hotspot

Using pseudorandom number generators (PRNGs) is security-sensitive

Security Hotspot

Empty lines should not be tested with regex MULTILINE flag

Code Smell

Cognitive Complexity of functions should not be too high

Code Smell

- OWASP Top 10 2017 Category A3 - Sensitive Data Exposure
- OWASP Top 10 2017 Category A6 - Security Misconfiguration
- MITRE, CWE-311 - Missing Encryption of Sensitive Data
- MITRE, CWE-321 - Use of Hard-coded Cryptographic Key

Available In:

sonarlint | sonarcloud | sonarqube

Privacy Policy