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

Functions returning Flow/Channel should not be suspending

Analyze your code

Code Smell | Major | coroutines bad-practice

There are two ways to define asynchronous functions in Kotlin:

- using the modifier `suspend` in the function declaration
- creating an extension function on `CoroutineScope`

The `suspend` modifier is generally used for functions that might take some time to complete. The caller coroutine might potentially be suspended.

Functions that start a coroutine in the background and return before said coroutine has completed running should be extension functions on `CoroutineScope`. This helps to clarify the intention of such a function. Further, such functions should not be suspending, as suspending functions should only return once all the work they are designed to perform is complete.

Functions returning `Flow` or `Channel` should return the result immediately and may start a new coroutine in the background. As a consequence, such functions should not be suspending and if they launch a coroutine in the background, they should be declared as extension functions on `CoroutineScope`.

Noncompliant Code Example

```
suspend fun f(): Flow<Int> {
    val flow = flow {
        emit(1)
    }
    delay(500L)
    return flow
}
```

```
suspend fun f(): Channel<Int> {
    val ch = Channel<Int>()
    ch.send(1)
    return ch
}
```

Compliant Solution

```
fun f(): Flow<Int> = flow {
    emit(1)
}
```

```
fun CoroutineScope.f(): Channel<Int> {
    val ch = Channel<Int>()
    launch {
        ch.send(1)
    }
    return ch
}
```

See

"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

- Coroutine Context and Scope

Available In:

sonarlint | sonarcloud | sonarqube

© 2008-2022 SonarSource S.A., Switzerland. All content is copyright protected. SONAR, SONARSOURCE, SONARLINT, SONARQUBE and SONARCLOUD are trademarks of SonarSource S.A. All other trademarks and copyrights are the property of their respective owners. All rights are expressly reserved.
Privacy Policy