



ABAP

Apex Apex

C C

C++

CloudFormation

COBOL COBOL

C# C#

E CSS

X Flex

GO Go

₹ HTML

🐇 Java

JavaScript

Kotlin

Kubernetes

Objective C

PHP

PL/I

PL/SQL PL/SQL

Python

RPG RPG

Ruby

Scala

Swift

Terraform

■ Text

TS TypeScript

T-SQL

VB VB.NET

VB6 VB6

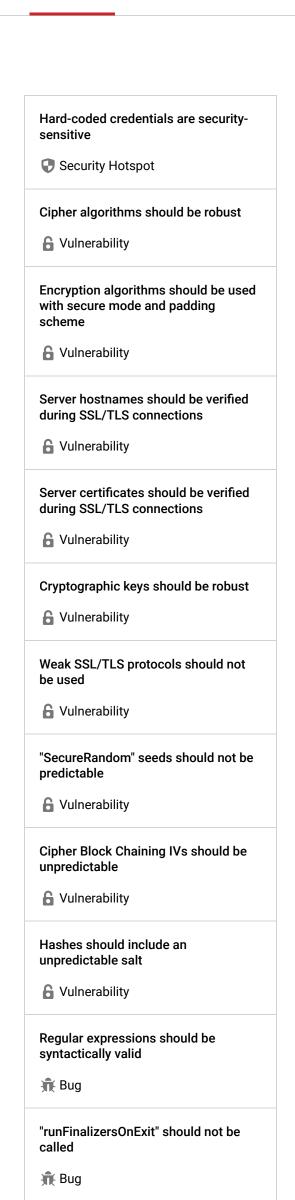
XML XML

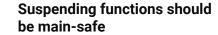


Kotlin static code analysis

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

Tags





Analyze your code

☼ Code Smell
♠ Major ② coroutines performance bad-practice pitfall

Search by name...

Generally speaking, main threads should be available to allow user-facing parts of an application to remain responsive. Long-running blocking operations can significantly reduce threads' availability and are best executed on a designated thread pool.

As a consequence, suspending functions should not block main threads and instead move any long-running blocking tasks off the main thread. This can be done conveniently by using withContext with an appropriate dispatcher.

Alternatively, coroutine builders such as launch and async accept an optional CoroutineContext. An appropriate dispatcher could be Dispatchers. IO for long-running blocking IO operations, which can create and shutdown threads on demand.

For some blocking tasks and APIs there may already be suspending alternatives available. When available, these alternatives should be used instead of their blocking counterparts.

This rule raises an issue when the call of a long-running blocking function is detected within a suspending function without the use of an appropriate dispatcher. If non-blocking alternatives to the called function are known, they may be suggested (e.g. use delay (...) instead of Thread.sleep (...)).

Noncompliant Code Example

Executing long-running blocking IO operations on the main thread pool:

```
class workerClass {
    suspend fun worker(): String {
        val client = HttpClient.newHttpClient()
        val request = HttpRequest.newBuilder(URI("https:/
        return coroutineScope {
            client.send(request, HttpResponse.BodyHandlen
        }
    }
}
```

Using inappropriate blocking APIs:

```
suspend fun example() {
    ...
    Thread.sleep(1000) // Noncompliant
    ...
}
```

Compliant Solution

Executing long-running blocking IO operations in an appropriate thread pool using Dispatcher.IO:

```
class workerClass(
    private val ioDispatcher: CoroutineDispatcher = Dispatcher
) {
    suspend fun worker(): String {
      val client = HttpClient.newHttpClient()
      val request = HttpRequest.newBuilder(URI("https://www.equality.com/patcher)
```

"ScheduledThreadPoolExecutor" should not have 0 core threads Rug Bug Jump statements should not occur in "finally" blocks Rug Bug Using clear-text protocols is securitysensitive Security Hotspot Accessing Android external storage is security-sensitive Security Hotspot Receiving intents is security-sensitive Security Hotspot Broadcasting intents is securitysensitive Security Hotspot Using weak hashing algorithms is security-sensitive Security Hotspot Using pseudorandom number generators (PRNGs) is securitysensitive 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

```
return withContext(ioDispatcher) {
      client.send(request, HttpResponse.BodyHandler
    }
}
```

Using appropriate non-blocking APIs:

```
suspend fun example() {
    ...
    delay(1000) // Compliant
    ...
}
```

See

- Coroutine context and dispatchers
- <u>Suspend functions should be safe to call from the main thread</u> (Android coroutines best practices)
- IO CoroutineDispatcher
- Default CoroutineDispatcher

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