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

Suspending functions should not be called on a different dispatcher

Analyze your code

Code Smell Major coroutines clumsy

By convention, calls to suspending functions should not block the thread, eliminating the requirement of calling suspending functions on background threads. Any long-running blocking operations should be moved to background threads within the suspending function that performs these operations. If suspending functions do block, this is breaking the Kotlin coroutines conventions (also see {rule:kotlin:S6307}).

As suspending functions can generally be called directly within other suspending functions, there is no need to move such a call to a background thread. This does not bring much benefit while adding complexity and making the code harder to understand.

In fact, various libraries explicitly provide suspending APIs for otherwise long-running blocking operations. The complexity of moving the appropriate tasks to background threads is already taken care of within the library and does not have to be considered when calling the library's suspending API.

Note also, however, that suspending functions do not block by convention. This behavior is not enforced on a technical level, leaving it to the developer to ensure the conventions are actually followed. If a suspending library function not under the control of a developer is actually blocking, then calling it in a different thread can work around the issues caused by the poorly written suspending function. It should be preferred to fix the callee, however, and not make such workarounds common practice.

This rule raises an issue when a suspending function is called in a way that executes the call in a new thread.

Noncompliant Code Example

```
suspend fun caller() = coroutineScope {
    async(Dispatchers.IO) {
        callee()
    }
}

suspend fun callee() {
    // Do some work
}
```

Compliant Solution

```
suspend fun caller() = coroutineScope {
    async {
        callee()
    }
}

suspend fun callee() {
    // Do some work
}
```

See

- Composing suspending functions in the Kotlin docs

"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

- {rule:kotlin:S6307}: Suspending functions should be main-safe

Available In:

sonarlint sonarcloud sonarqube