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

"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

Operator "is" should be used instead of "isInstance()"

Analyze your code

Code Smell Major

The `is` construction is a preferred way to check whether a variable can be cast to some type statically because a compile-time error will occur in case of incompatible types. The `isInstance()` functions from `kotlin.reflect.KClass` and `java.lang.Class` work differently and type check at runtime only. Incompatible types will therefore not be detected as early during development, potentially resulting in dead code. `isInstance()` function calls should only be used in dynamic cases when the `is` operator can't be used.

This rule raises an issue when `isInstance()` is used and could be replaced with an `is` check.

Noncompliant Code Example

```
fun f(o: Any): Int {
    if (String::class.isInstance(o)) {  // Noncompliant
        return 42
    }
    return 0
}

fun f(n: Number): Int {
    if (String::class.isInstance(n)) {  // Noncompliant
        return 42
    }
    return 0
}
```

Compliant Solution

```
fun f(o: Any): Int {
    if (o is String) {  // Compliant
        return 42
    }
    return 0
}

fun f(n: Number): Int {
    if (n is String) {  // Compile-time error
        return 42
    }
    return 0
}

fun f(o: Any, c: String): Boolean {
    return Class.forName(c).isInstance(o) // Compliant,
}
```

Available In:

sonarlint sonarcloud sonarqube