




 Secrets


 ABAP


 Apex


 C


 C++


 CloudFormation


 COBOL


 C#


 CSS


 Flex


 Go


 HTML


 **Java**


 JavaScript


 Kotlin


 Objective C


 PHP


 PL/I


 PL/SQL


 Python


 RPG


 Ruby


 Scala


 Swift


 Terraform


 Text


 TypeScript

 T-SQL

 VB.NET

 VB6

 XML



## Java static code analysis

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

All rules632

Vulnerability53

Bug154


Security Hotspot36

Code Smell389


Quick Fix42


Tags ▾


Search by name... 🔍


 Vulnerability


Server hostnames should be verified during SSL/TLS connections





 Insecure temporary file creation methods should not be used





 Passwords should not be stored in plain-text or with a fast hashing algorithm





 Server certificates should be verified during SSL/TLS connections




 Persistent entities should not be used as arguments of "@RequestMapping" methods



 "HttpSecurity" URL patterns should be correctly ordered



 LDAP connections should be authenticated



 Cryptographic keys should be robust



 Weak SSL/TLS protocols should not be used



 "SecureRandom" seeds should not be predictable






 Cipher Block Chaining IVs should be unpredictable



### Double-checked locking should not be used

Analyze your code

 Bug  Blocker  cwe multi-threading cert

Double-checked locking is the practice of checking a lazy-initialized object's state both before and after a synchronized block is entered to determine whether or not to initialize the object.

It does not work reliably in a platform-independent manner without additional synchronization for mutable instances of anything other than `float` or `int`. Using double-checked locking for the lazy initialization of any other type of primitive or mutable object risks a second thread using an uninitialized or partially initialized member while the first thread is still creating it, and crashing the program.

There are multiple ways to fix this. The simplest one is to simply not use double checked locking at all, and synchronize the whole method instead. With early versions of the JVM, synchronizing the whole method was generally advised against for performance reasons. But synchronized performance has improved a lot in newer JVMs, so this is now a preferred solution. If you prefer to avoid using synchronized altogether, you can use an inner static `class` to hold the reference instead. Inner static classes are guaranteed to load lazily.

#### Noncompliant Code Example

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }

    static class Resource {
    }
}
```





#### Compliant Solution

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public static synchronized Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

https://rules.sonarsource.com/java/RSPEC-2168

1/2

Basic authentication should not be used	 Vulnerability
Regular expressions should not be vulnerable to Denial of Service attacks	 Vulnerability
"HttpServletRequest.getRequestSession" should not be used	 Vulnerability
Hashes should include an unpredictable salt	 Vulnerability

```
static class Resource {  
    }  
}
```

With inner static holder:

```
@ThreadSafe  
public class ResourceFactory {  
    private static class ResourceHolder {  
        public static Resource resource = new Resource();  
    }  
  
    public static Resource getResource() {  
        return ResourceHolder.resource;  
    }  
  
    static class Resource {  
    }  
}
```

Using "volatile":

```
class ResourceFactory {  
    private volatile Resource resource;  
  
    public Resource getResource() {  
        Resource localResource = resource;  
        if (localResource == null) {  
            synchronized (this) {  
                localResource = resource;  
                if (localResource == null) {  
                    resource = localResource = new Resource();  
                }  
            }  
        }  
        return localResource;  
    }  
  
    static class Resource {  
    }  
}
```

See

- [The "Double-Checked Locking is Broken" Declaration](#)
- [CERT, LCK10-J](#). - Use a correct form of the double-checked locking idiom
- [MITRE, CWE-609](#) - Double-checked locking
- [JLS 12.4](#) - Initialization of Classes and Interfaces
- Wikipedia: [Double-checked locking](#)

Available In:

