FB.SIC\_THREADLOCAL\_DEADLY\_EMBRACE

Этот класс является внутренним классом, но, вероятно, должен быть статическим внутренним классом. В действительности существует серьезная опасность смертельного столкновения между внутренним классом и потоком, локальным во внешнем классе. Поскольку внутренний класс не является статическим, он сохраняет ссылку на внешний класс. Если локальный поток содержит ссылку на экземпляр внутреннего класса, внутренний и внешний экземпляры будут доступны и не подлежат сборке мусора.

Таблица №1 – Основные характеристики детектора FB.SIC\_THREADLOCAL\_DEADLY\_EMBRACE

|  |  |
| --- | --- |
| Категория критичности | Minor |
| Надежность | Unknown |
| Источник | SpotBugs |
| Тип ошибки | Deadlock/Leak |
| CWE | --- |
| CVE | --- |
| Возможные последствия | Утечка памяти, снижение производительности, отказ службы |
| Комментарий | Рекомендуется преобразовывать внутренние классы в статические, если они используются с ThreadLocal |

Таблица №2 – Примеры срабатывания детектора и устранения уязвимости

|  |  |
| --- | --- |
| Потенциально небезопасное | Корректная конструкция |
| public static void dangerousUsage() {  OuterClass outer = new OuterClass();  outer.initDangerous();  // Потенциальная утечка памяти и deadlock  } | public static void safeUsage() {  // Вариант 1: Статический внутренний класс  OuterClass.StaticInner safe1 = new OuterClass.StaticInner();    // Вариант 2: WeakReference  ThreadLocal<WeakReference<OuterClass.StaticInner>> safeThreadLocal =  new ThreadLocal<>();  safeThreadLocal.set(new WeakReference<>(safe1));    // Вариант 3: Отдельный класс  ThreadLocal<IndependentClass> safest = new ThreadLocal<>();  safest.set(new IndependentClass());  } |

Методы устранения уязвимости при срабатывании детектора.

1. Использование статических внутренних классов;
2. Очистка ThreadLocal;
3. Использование WeakReference;
4. Переход на альтернативные подходы.