Java Sicherheitsanalyse mit Pattern-Detection-Tools

Mark Kreitz · Andrea Baumann

Universität der Bundeswehr München
Institut für Software Engineering
{mark.kreitz | andrea.baumann}@unibw.de

Zusammenfassung

Pattern-Detection-Tools werden zum Auffinden von Entwurfsmustern (Design-Patterns) in Softwaresystemen genutzt. In diesem Beitrag zeigen wir, dass sich diese Tools auch für das Erkennen von sicherheitsrelevanten Schwachstellen oder Designfehlern eignen. Dazu stellen wir unser Tool vor, das Java Quelltexte sowohl statisch als auch dynamisch analysiert und gehen auf verwandte Arbeiten ein. Um die Sicherheitslücken in einem Pattern-Detection-Tool zu erkennen, definieren wir diese als Muster, die im Nachfolgenden als Flaw-Pattern bezeichnet werden. Neben den Flaw-Patterns betrachten wir auch Entwurfsmuster, die die Programmsicherheit erhöhen können, die sogenannten Security-Patterns. Anhand konkreter Erweiterung validieren wir den von uns gewählten Ansatz. Darüber hinaus diskutieren wir, wie durch die Verwendung von Annotationen die Analyse verbessert werden kann. Abschließend wird die Integration der Tool-gestützten Sicherheitsanalyse in den Software-Engineering-Prozess dargelegt.

1 Motivation

Im Software-Engineering-Prozess fehlt häufig eine kontinuierliche Beachtung der Security. Dies liegt daran, dass die Sicherheit zwar als nicht-funktionale Anforderung formuliert, aber während der Implementierung meistens nicht betrachtet wird. Heutzutage erwarten Kunden jedoch ein sicheres System, sogar ohne diese Anforderung explizit zu stellen [Paul11]. Selbst bei gut strukturierter Software und immer größer werdenden Softwarearchitekturen ist es im Nachhinein nicht mehr offensichtlich, ob Sicherheitslücken vorhanden sind. Diese werden erst in der Code-Review oder zum Teil gar nicht erkannt und benötigen gegebenenfalls aufwendig zu realisierende Sicherheitspatches.

Idealerweise sollten die Sicherheitsanforderungen in jedem Prozessschritt beachtet werden [HoLi06]. Das hat den Vorteil, dass von Beginn an ein sicheres System entsteht und nachfolgende Sicherheitspatches unwahrscheinlicher werden. Im Rahmen dieses Dokuments fokussieren wir uns dabei auf den Prozessschritt der Implementierung von Java-Applikationen und wie man diesen auch ohne Kenntnis der nicht-funktionalen Anforderungen im Kontext der Security unterstützen kann.

Jeder Programmierer sollte sich an folgendem Leitfaden orientieren: *„Prefer to have obviously no flaws rather than having no obvious flaws“* [Orac15]. Programmcode, der offensichtlich keine Fehler enthält, soll einem Programmcode bevorzugt werden, der keine offensichtlichen Fehler beinhaltet. Dies kann z.B. durch Code-Reviews erreicht werden. Besser wäre es allerdings, wenn man den Programmierern bei der Erstellung eines sicheren Softwaresystems Ana-

P. Schartner · A. Baumann (Hrsg.) · D•A•CH Security 2017 · syssec (2017) 362-372.

lysewerkzeuge zur Verfügung stellt, die vorhandene Schwachstellen auffinden können. Solche Tools existieren bereits um Fehler im Quelltext zu finden. Beispiele für Java sind FindBugs [HoPu04] und PMD [pmd17] (siehe Abschnitt 3). Beide Tools fokussieren sich jedoch nicht auf das Finden von Sicherheitslücken, sondern auf das Aufzeigen von Programmierfehlern. Dadurch decken sie den Security-Aspekt nur teilweise ab.

Infolgedessen wird ein Tool benötigt, welches die Sicherheit von Programmen durch das Auffinden von Sicherheitslücken bewerten kann. Ein solches Tool kann dann sowohl für die Entwicklung von Softwaresystemen, als auch zur Analyse von fremden Programmen oder Bibliotheken benutzt werden. Bei fremden Quelltexten ist eine Überprüfung besonders wichtig, falls der Hersteller nicht vertrauenswürdig ist oder Sicherheit im Herstellungsprozess keine oder nur eine untergeordnete Rolle gespielt hat.

Typische (Security-)Fehler, die von Entwicklern vermieden werden sollten, hat Oracle in den Secure Coding Guidelines (SCG), vgl. [Orac15], beschrieben. Einige dieser Richtlinien lassen sich als Muster ausdrücken. Daher ist es ein naheliegender Ansatz diese Muster für ein Pattern-Detection-Tool zu definieren und dieses zur Sicherheitsanalyse des Programms zu nutzen. Entwickler können bereits während der Implementierung mit diesem Werkzeug Software auf mögliche Schwachstellen überprüfen und entsprechende Anpassungen vornehmen. Ein Vorteil der Nutzung eines Pattern-Detection-Tools ist, dass dieses sich nicht nur für die Suche nach potentiellen Sicherheitslücken, sondern gleichzeitig auch für seine eigentliche Bestimmung, der Suche nach Entwurfsmustern, nutzen lässt. Gefundene Patterns lassen auf eine strukturierte und durchdachte Programmierung schließen und machen den Quelltext vertrauenswürdiger. Gleichzeitig vereinfachen die Informationen über die Patterns die Code-Review. Bei der Suche nach Entwurfsmustern lassen sich außerdem Security-Patterns finden. Diese bezeichnen Standardlösungen für sicherheitskritische Probleme.

2 Pattern-Detection-Tool

Das an unserem Institut entwickelte Pattern-Detection-Tool verwendet sowohl statische als auch dynamische Code-Analyse. Die dynamische Betrachtung stellt einen wesentlichen Unterschied im Vergleich zu anderen Tools, wie zum Beispiel PINOT [ShOl06] oder HEDGEHOG [BlBS05], dar, die den Code nur statisch analysieren.

Unser Tool verwendet einerseits die Reflection API, die auf den class- und jar-Dateien operiert, und andererseits die Compiler Tree API, welche die java-Dateien benötigt. Beide APIs werden von Oracle zur Verfügung gestellt. Die Analyse erfolgt datengetriebenen. Nacheinander werden die einzelnen Klassen geladen. Die resultierenden Class-Objekte werden dann an die patternspezifischen Filter übergeben. Die Aufgabe eines Filters ist es, das Class-Objekt zu untersuchen und, falls das im Filter definierte Pattern zutrifft, dieses an die Senke zu berichten. Die programmatische Umsetzung der Filter ist mit der in Abschnitt 4 exemplarisch vorgestellten, regelbasierten Beschreibungsform einfach, wie wir anhand konkreter Beispiele zeigen.

Der Schwerpunkt in unserem Tool liegt auf der Verwendung der Reflection API, da wir mit dieser sowohl statische als auch dynamische Informationen gewinnen können. Zudem ist der Zugriff auf den Aufbau der Klassen über die Reflection API gegenüber der Compiler Tree API wesentlich einfacher. Somit können wir die statischen Eigenschaften der Klasse, ihrer Attribute, ihrer Konstruktoren und Methoden mit deren Parametern ermitteln.

Darüber hinaus können wir durch die dynamische Code-Analyse der Reflection API Informatio-

nen erlangen, die erst zur Laufzeit verfügbar sind. Beispielsweise benötigen wir diese Fähigkeit für einen Filter in Abschnitt 4.4, in dem wir das referenzierte Objekt eines Attributes zur Laufzeit auslesen und den konkreten Typen bestimmen müssen. Mit Hilfe der Reflection API ist es jedoch nicht möglich Aussagen über den Programmablauf innerhalb einer Methode zu treffen.

Daher analysieren wir die Methodeninhalte mit der Compiler Tree API, die für jede Datei einen abstrakten Syntaxbaum (AST) aus dem Programmcode erstellt. Dieser enthält alle Klassen und deren Attribute sowie Methoden die in dieser Datei vorhanden sind. Nativ gibt es keine Möglichkeit, von den Reflection-Objekten, wie Klassen oder Methoden, an den zugehörigen AST zu gelangen. Daher stellt unser Tool Funktionalität zur Verfügung, die es erlaubt, von einem Reflection-Objekt zu dem richtigen Teilbaum eines AST, beispielsweise dem einer bestimmten Methode, zu navigieren. Diese ermöglicht dem Filter, von der zunächst auf Reflections basierenden Analyse, den gezielten Zugriff auf den zutreffenden Teilbaum im AST. Die Kombination zwischen Reflections und Compiler Tree API wird von uns beispielsweise in Abschnitt 4.1 und 4.2 verwendet, um festzustellen welche Methoden innerhalb einer Methode aufgerufen werden.

Bevor wir im Abschnitt 4 die regelbasierte Beschreibungsform an Beispielen einführen, mit der wir Security- und Flaw-Pattern definieren, werden nachfolgend einige Analysewerkzeuge zum Auffinden von Schwachstellen vorgestellt.

3 Verwandte Analysewerkzeuge

Im Folgenden werden vergleichbare Tools vorgestellt und die Unterschiede zu unserem Verfahren erläutert.

Xanitizer von Rigs-IT [Rigs17] ist ein Sicherheits-Analysewerkzeug, welches speziell für Java-Web Anwendungen konzipiert ist. Es bietet unter anderem Taint-Checking und prüft ob Angriffe über Injections oder Cross-Site Scripting möglich sind. Da unser Tool sich nicht auf Web-Anwendungen fokussiert, wird dieses Werkzeug nicht weiter betrachtet.

FindBugs ist ein Open-Source Tool der University of Maryland [HoPu04]. Seine Aufgabe besteht darin, wie der Name schon sagt, Fehler im Code zu finden. Dabei handelt es sich vermehrt um typische Programmierfehler als um Fehler in der Security. Der Ansatz zum Finden der Bugs ist dem von uns Gewähltem sehr ähnlich. Für jeden Fehler existiert ein Bug-Pattern, nach dem gesucht wird.

PMD ist ein Analysewerkzeug für verschiedene Programmiersprachen. Es handelt sich dabei überwiegend um einen Style-Checker. Dieser ist auch dazu geeignet Fehler und Schwachstellen zu finden. Von den in den SCG genannten Richtlinien wurden jedoch laut eigener Dokumentation nur zwei umgesetzt [pmd17]. Dafür unterstützt PMD das Prüfen auf viele andere Regeln. Darunter befindet sich zum Beispiel die Regel, dass Attribute am Anfang der Klasse deklariert werden sollen. Darüber hinaus unterscheidet sich PMD von unserem Tool dadurch, dass hierbei lediglich auf statische Source-Code Analyse gesetzt wird und keine dynamischen Prozesse betrachtet werden.

4 Security- und Flaw-Pattern

Dieser Abschnitt betrachtet einige sicherheitsrelevanten Richtlinien aus den SCG im Detail. Zusätzlich werden Elemente aus [Bloc08, GHJV95] und [SFBHB+06] behandelt. Die resultierenden Muster werden in die Kategorien *Security-Pattern* und *Flaw-Pattern* eingeordnet.

Die **Security-Patterns** sind Design-Patterns und werden bei der Entwicklung eingesetzt um die Sicherheit der Software zu verbessern. Ihre Anwesenheit deutet auf ein durchdachtes Programmdesign und kann das Vertrauen in den vorliegenden Code stärken, was insbesondere bei fremden Quelltexten relevant ist. Unter den Security-Patterns werden auch Muster für den Aufbau ganzer Systeme gefasst, wie sie unter anderem in [SFBHB+06] behandelt werden. Diese System-Patterns stehen jedoch nicht im Fokus dieser Arbeit. Es werden daher nur die Software-Muster weiter betrachtet.

Beispiele für die Security Patterns sind das Proxy-Pattern ([GHJV95] S. 207ff), das Interpreter-Pattern ([GHJV95] S. 243ff) oder das Builder-Pattern ([Bloc08] S. 11ff). Das Proxy-Pattern wird in 4.1 genauer behandelt. Das Interpreter-Pattern ist ein Verhaltensmuster. Klassischer Weise wird es zur Interpretation einer formalen Sprache verwendet. Im Bezug auf die Sicherheit von Software bietet es jedoch auch die Möglichkeit Eingaben und Parameter (hierbei in der Regel Strings) zu überprüfen, bevor diese weiterverarbeitet werden. Durch eine Validierung der Eingabe wird die Robustheit des Programms verstärkt. Sie ist ein wichtiger Sicherheitsmechanismus, wenn externe Eingaben verarbeitet oder Objekte aus fremdem Quelltext angenommen werden (vergleiche [Orac15] Abschnitt 5). Das Builder-Pattern von Joshua Bloch ermöglicht ebenfalls eine Überprüfung der Eingaben, hier jedoch vor der Erstellung von Objekten. Es stellt sicher, dass ein Objekt nur erzeugt wird, wenn alle notwendigen Daten vorliegen und das Objekt korrekt erzeugt werden kann. Dies verhindert das Entstehen von nicht-initialisierten Objekten im Speicher und beugt finalizer-Angriffen vor.

Die **Flaw-Patterns** beschreiben Sicherheitslücken oder potentiell bösartigen Programmcode. Sie bilden den Hauptteil der betrachteten Muster. Die Flaw-Pattern werden im Nachfolgenden weiter differenziert in Sicherheitslücken und Warnungen. Die Sicherheitslücken sind kritisch und müssen von einem Entwickler behoben werden. Die Warnungen stellen im wesentlichen Empfehlungen dar, welche die Sicherheit erhöhen können, es aber nicht zwingend notwendig ist, diese zu beachten. Des Weiteren fallen unter die Warnungen auch nicht eindeutig identifizierbare Sicherheitslücken, die zu vielen (ggf. falschen) Ausgaben führen können. Dies betrifft in der Regel Patterns, welche eine sehr allgemeine Definition haben und daher häufig fälschlich erkannt werden könnten.

Im Folgenden wird ein Repräsentant der Security- und verschiedene Flaw-Patterns genauer vorgestellt. Dabei wird auf die verschiedenen Regeln eingegangen und beschrieben, wie die Regeln in den Filtern umgesetzt werden können.

4.1 Security-Pattern: Das Proxy Pattern

Das Proxy Pattern (siehe Abbildung 1) wird in der Literatur als Struktur-Pattern aufgeführt. Es bietet einen regulierten Zugriff auf Methoden eines konkreten Objektes über ein Stellvertreterobjekt [GHJV95]. Das Muster existiert in Variationen für verschiedene Anwendungsszenarien. Der Protection Proxy ist eine dieser Varianten. Er stellt dem (Client-)Objekt eine reduzierte Auswahl der Methoden aus *Subjekt* über das *KonkretesSubjekt* zur Verfügung. Somit erfolgt eine Art der Zugriffskontrolle, weshalb dieses Pattern ein Security-Pattern ist.

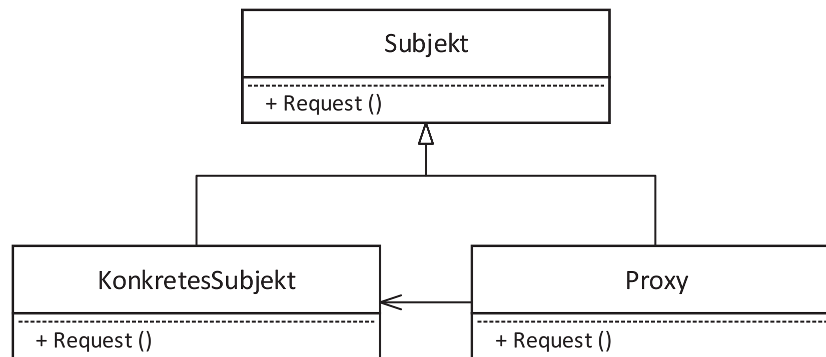

Abb. 1: Klassendiagramm des Proxy Patterns

Zur Erkennung des Patterns werden die in Tabelle 1 aufgestellten Regeln verwendet.

Tab. 1: Regeln des Proxy Patterns

Regel	Beschreibung	Abhängigkeit
Proxy.1	Untersuchte Klasse (*Proxy*) ist eine instanziierbare Klasse	
Proxy.2	*Proxy* erbt von einer abstrakten Klasse oder einem Interface (*Subjekt*)	Proxy.1
Proxy.3	*Proxy* hat ein Attribut eines Untertypen von *Subjekt* (*Konkretes-Subjekt*)	Proxy.1 und Proxy.2
Proxy.4	*Proxy* überschreibt Methoden von *Subjekt*	Proxy.1 und Proxy.2
Proxy.5	In mindestens einer dieser Methoden wird der Aufruf an *Konkre-tesSubjekt* weitergeleitet	Proxy.2 und Proxy.3

Die benötigten Informationen zur Überprüfung der Regeln Proxy.1 bis Proxy.4 können leicht über die Reflection API gewonnen werden, da der Filter z.B. über das Class-Object die Vererbungshierarchie (Proxy.2) und die Art der Klassen (Proxy.1, Proxy.2) untersuchen kann. Das Gleiche gilt für Proxy.3 und Proxy.4 bei dem die Attribute und Methoden der zu untersuchenden Klasse analysiert werden. Im Gegensatz dazu muss die Regel Proxy.5 über den zur Methode gehörigen AST verifiziert werden. Dafür kann die direkte Zugriffsmöglichkeit auf den zutreffenden Teilbaum des ASTs durch unser Tools genutzt werden.

4.2 Flaw-Pattern: Überschreibbare Methodenaufrufe im Konstruktor

Wird eine überschreibbare Methode in einem Konstruktor aufgerufen, so bietet diese einem Angreifer die Möglichkeit eine Referenz auf das erstellte Objekt zu erhalten, bevor das Objekt initialisiert ist ([Orac15] 7-4). Dies ist auch möglich, wenn der Angreifer nicht die Berechtigung zur Erstellung des Objektes besitzt, aber ein Objekt der überschreibenden Klasse erzeugt wird. Beispielhaft wird in Listing 1 die fehlerhafte Implementierung eines Zählers gezeigt. In diesem ruft der Konstruktor die Methode *setCounter* auf, welche überschreibbar ist. In der Klasse *MaliciousCounter* (Listing 2) wird diese Methode überschrieben und in Zeile 12 wird die Referenz des Objektes an eine statische Methode weitergereicht, bevor sogar der Konstruktor der Oberklasse beendet wurde. Um die Sicherheitslücke zu vermeiden sollte z.B. die Klasse *Counter* oder die Methode *setCounter* auf *final* gesetzt werden.

Listing 1: Überscheibbarer Methodenaufruf im Konstruktor

```
1  public class Counter {
2    private final int MAXIMUM;
3    private int currentValue;
4
5    public Counter(
6        int maxValue, int startValue) {
7      MAXIMUM = maxValue;
8      setCounter(startValue);
9    }
10
11   public void setCounter(int toSet) {
12     currentValue = toSet > MAXIMUM
13       ? 0 : toSet;
14   }
15   //weitere Funktionalität...
16 }
```

Listing 2: Verwendung der Sicherheitslücke zur Weitergabe der Objektreferenz

```
1  public class MaliciousCounter
2      extends Counter {
3
4
5    public MaliciousCounter(
6        int maxValue, int startValue) {
7      super(maxValue, startValue);
8    }
9
10   @Override
11   public void setCounter(int toSet) {
12     SomeClass.someStaticMethod(this);
13     super.setCounter(toSet);
14   }
15
16 }
```

Diese Sicherheitslücke lässt sich durch ein Flaw-Pattern erkennen, das durch die Regeln in Tabelle 2 definiert ist.

Tab. 2: Regeln des Patterns zu überschreibbaren Methodenaufrufen im Konstruktor

Regel	Beschreibung	Abhängigkeit
OMiC.1	Die zu untersuchende Klasse ist nicht final oder effektiv final	
OMiC.2	Sie besitzt einen Konstruktor welcher eine *unsichere Methode* aufruft	OMiC.3 oder OMiC.4
OMiC.3	Eine *unsichere Methode* ist eine *überschreibbare Methode* oder	OMiC.5
OMiC.4	Eine *unsichere Methode* ist eine Methode, welche eine *unsichere Methode* aufruft	OMiC.3 oder OMiC.4
OMiC.5	Eine *überschreibbare Methode* ist nicht *private* und nicht *final*	

Die Eigenschaft der Klasse final bzw. effektiv final zu sein, lässt sich über die Reflection API erhalten (OMiC.1). In den Regeln OMiC.2 bis OMiC.4 werden jeweils die Methodenaufrufe in den Konstruktoren bzw. Methoden über den zugehörigen AST untersucht. Zusätzlich zu der bereits bekannten Zuordnung auf die Teilbäume des AST, unterstützt unser Tool ebenfalls, zu einem Methodenaufruf im AST das zugehörige Reflection-Objekt zu erlangen. So können die Modifikatoren für Regel OMiC.5 wieder über die Reflections abgefragt werden.

4.3 Flaw-Pattern: Öffentliche statische Variablen

Öffentliche Klassenattribute (*public static*) sind aus jedem Kontext adressierbar. Dabei ist es problematisch, wenn diese Attribute nicht *final* sind, sondern Variablen darstellen. In diesem Fall hat jeder Programmcode die Möglichkeit, den Wert des Attributes zu ändern. Das kann Einfluss auf den Programmablauf haben. Daher sind öffentliche Klassenattribute, welche keine Konstanten darstellen ein Sicherheitsrisiko ([Orac15] 6-9). Mit der in Tabelle 3 dargestellten Regel ist es sehr einfach, die betroffenen Attribute reflexiv zu finden.

Tab. 3: Regeln des Patterns zu beschreibbaren öffentlichen Klassenattributen

Regel	Beschreibung	Abhängigkeit
ÖsV.1	Ein Attribut (der untersuchten Klasse) ist *public* und *static* aber nicht *final*	

4.4 Flaw-Pattern: Veränderliche Konstanten

Im vorigen Unterabschntt 4.3 wurde dargestellt, warum es wichtig ist, dass öffentliche Klassenattribute Konstanten sind. Es muss jedoch auch sichergestellt sein, dass eine Konstante unveränderlich ist. Wird beispielsweise ein Array als Konstante gespeichert, so ist zwar die Referenz auf das Array vor dem Überschreiben geschützt, dennoch bleiben die Inhalte des Arrays austauschbar. Daher muss sichergestellt sein, dass eine Konstante bzw. ihr Inhalt unveränderlich ist ([Orac15] 6-10).

Die Regeln für dieses Flaw-Pattern sind in Tabelle 4 aufgezählt. Die Umsetzung des Filters ist auf Basis der Reflections möglich. Für Regel VäKo.2 sind die dynamischen Analysemöglichkeiten der Reflections essentiell, da der konkrete Typ des Objektes bestimmt werden muss, der in der Konstante gespeichert ist. Es wäre ein naiver Ansatz, an dieser Stelle den Typen der Konstanten selbst zu betrachten, anstelle den des referenzierten Objektes. Beispielsweise könnte der Typ des Attributes ein Interface sein, während das referenzierte Objekt eine Instanz einer konkreten Unterklasse ist. Über die Reflections können wir auf das in der Konstanten gespeicherte Objekt zugreifen und dieses auf Veränderlichkeit überprüfen.

Tab. 4: Regeln des Patterns zu veränderlichen Konstanten

Regel	Beschreibung	Abhängigkeit
VäKo.1	Die zu untersuchende Klasse beinhaltet eine *veränderliche Konstante*	VäKo.2
VäKo.2	Eine *veränderliche Konstante* ist ein Attribut mit den Modifikatoren *public static* und *final* und beinhaltet ein *veränderliches Objekt* (keinen primitiven Datentypen oder Enums)	VäKo.3
VäKo.3	Ein *veränderliches Objekt* beinhaltet mindestens ein *veränderliches Attribut*	VäKo.4 oder VäKo.5 oder VäKo.6
VäKo.4	Ein *veränderliches Attribut* ist nicht *final* oder	
VäKo.5	Ein *veränderliches Attribut* ist ein Array oder	
VäKo.6	Ein *veränderliches Attribut* referenziert ein *veränderliches Objekt*	VäKo.3

Regel VäKo.4 kann dahingehend erweitert werden, dass ein Attribut auch nicht *effektiv final* sein darf. Dies bedeutet, dass es den Modifikator *private* trägt und von keiner Methode verändert oder zurückgegeben wird. Diese Betrachtung kann die false-positive Rate für dieses Pattern reduzieren. Zugunsten der Einfachheit wurde hier jedoch auf die Implementierung verzichtet, um nicht alle Variablenzuweisungen im abstrakten Syntaxbaum überprüfen zu müssen und das Pattern rein reflexiv erkennen zu können.

Um die false-positive Rate weiter zu senken, können Annotationen verwendet werden. Wie wir sie dafür und für weitere Überprüfungen einsetzen können, wird im nächsten Abschnitt gezeigt.

5 Annotationen

Annotationen ermöglichen es Metadaten in den Quellcode und das Programm einzubinden. Für die Erkennung von Patterns lassen sich die Annotationen vielfältig verwenden. Einerseits können Warnungen vor Schwachstellen unterdrückt werden, sofern diese „by design" sind. Andererseits lassen sich Eigenschaften im Quelltext markieren, die vom Programmierer eingehalten werden müssen. Im Folgenden zeigen wir anhand verschiedener Szenarien das Potential der Verwendung von Annotationen.

In Abschnitt 4.4 musste der Filter veränderliche Konstanten erkennen. Nehmen wir an, dass eine Konstante ein Objekt vom Typ *UnmodifiableCollection* referenziert. In diesem Fall würde eine Sicherheitslücke erkannt werden, da sich die Implementierung dieser intern auf eine modifizierbare Collection abstützt. Ist bekannt, dass die interne, modifizierbare Collection nicht von anderen Programmteilen referenziert wird und nur unveränderliche Objekte beinhaltet, kann das Attribut mit einer Annotation *@SuppressWarnings("mutable")* gekennzeichnet werden.

Darüber hinaus kann eine Annotation *@Immutable* eingeführt werden, die unveränderliche Klassen direkt kennzeichnet. So kann einerseits eine Regel diese Klassen als Unveränderliche erkennen, ohne sie näher zu analysieren. In Abschnitt 4.4 kann dadurch beispielsweise früher erkannt werden, ob ein Objekt von einem unveränderlichen Typen ist. Andererseits ist mit Hilfe dieser Annotation eine gezielte Überprüfung der markierten Klasse auf Veränderlichkeit möglich. Über den gesamten Entwicklungsprozess kann dann sichergestellt werden, dass die Unveränderlichkeit der Klasse trotz Weiterentwicklung erhalten bleibt.

Durch Annotationen lässt sich nicht nur die Genauigkeit von Filtern erhöhen, es lassen sich auch weitere Schwachstellen zur selben Richtlinie erfassen. Nehmen wir als Beispiel die Richtlinien 2-1 und 2-2 [Orac15]. Sie warnen vor dem Auftauchen sensibler Daten in Exceptions oder in Logs. Welche Daten sensibel sind lässt sich mit Ausnahme bestimmter Exception, wie z.B. der *FileNotFoundException*, nicht ohne Weiteres erkennen. Werden sensible Daten mit *@Sensitive* annotiert, dann kann das Tool unter Umständen überprüfen, ob diese Daten im Rahmen einer Exception- oder Logging-Nachricht verwendet werden.

Über die Präzisierung und Erweiterung bestehender Filter hinaus können durch die Annotationen neue Schwachstellen gefunden werden. Dies betrifft unter anderem Schwachstellen aus den SCG, welche den Umgang mit Parametern und Rückgabewerten von Methoden beschreiben (vgl. Richtlinien 5-1, 5-2, 6-6, 6-7 in [Orac15]). Annotationen erlauben hier zum Beispiel eine Markierung, welche Parameter oder Rückgabewerte von Methoden wie überprüft werden müssen.

Ein weiteres Beispiel für einen Filter, der ohne die Annotationen nicht möglich ist, beschreibt die SCG 8-1: Objekte sensibler Klassen dürfen nicht serialisierbar sein. Durch Annotation der sensiblen Klasse lässt sich für diese und mögliche Unterklassen feststellen, ob sie *Serializable* implementieren und damit gegen die Richtlinie verstoßen.

6 Auswertung

Zur Überprüfung unseres Tools haben wir dieses auf zwei Arten getestet. Einerseits wurden für jeden Filter mehrere Tests geschrieben, um sicherzustellen, dass unser Tool verschiedene Implementierungsmöglichkeiten der Schwachstelle erkennt, aber ähnlich wirkende Umsetzungen ignoriert. Um die Genauigkeit des Tools zu ermitteln wurde andererseits das Open-Source

Projekt Apache Airavata mit den in Abschnitt 4 vorgestellten Filtern untersucht.

Airavata ist ein Software Framework, das einem ermöglicht groß skalierbare Applikationen und Arbeitsabläufe auf verteilten Rechenressourcen zusammenzustellen, zu managen, auszuführen und zu überwachen [Apac]. Die von uns getestete Version 0.16 beinhaltet 3796 Klassen. Die Filter der Flaw-Pattern erzielten auf dem gesamten Projekt 81 Treffer, vgl. Tabelle 5. In der Spalte Präzision ist der jeweilige Anteil der Treffer angegeben, der dem tatsächlichen Flaw entspricht. Um die Präzision zu bestimmen wurden die Ergebnisse durch ein Review des untersuchten Codes überprüft.

Tab. 5: Ergebnisse der Suche nach den vorgestellten Flaw-Pattern in Apache Airavata

Abschnit	Beschreibung	Anzahl Treffer	davon bestätigt	Präzision
4.2	Überschreibbare Methodenaufrufe im Konstruktor	24	22	91.6%
4.3	Beschreibbare öffentliche Klassenattribute	38	38	100.0%
4.4	Veränderliche Konstanten	19	17	89.5%
	Gesamt	**81**	**77**	**95.1%**

Die Fehlklassifizierungen bei *Überschreibbare Methodenaufrufe im Konstruktor* resultieren daher, dass in der aktuellen Implementierung die aufgerufenen Methoden nur an dem Methodennamen und der Anzahl der Parameter überprüft werden, jedoch nicht anhand der Parametertypen. Wird die Implementierung hier entsprechend erweitert, würde der Filter die tatsächlich aufgerufene Methode überprüfen und keine falsche Ausgabe erzeugen.

Unter den Ausgaben bei *Veränderliche Konstanten* sind zwei, deren als veränderlich erkannte Klasse aus einer externen Bibliothek eingebunden wurde. Zu diesen Klassen verfügen wir über keine Quelltexte, sodass wir deren Richtigkeit nicht bestätigen können. In der Statistik für die Präzision werden diese Ausgaben daher als False-Positives angenommen.

Insgesamt ist die Präzision mit über 95% über alle betrachteten Filter sehr hoch. Durch weitere Verbesserung der Filter ist sogar eine Steigerung zu erwarten. Dies zeigt, dass sich Pattern-Detection-Tools gut zum Auffinden von Fehlern in Programmen eignen.

7 Zusammenfassung und Ausblick

Schwachstellen in der Programmierung lassen sich durch Muster ausdrücken, die von Pattern-Detection-Tools gefunden werden. Dies haben wir an der konkreten Umsetzung einiger der Richtlinien aus den SCGs gezeigt. Am Beispiel des Frameworks Airavata konnte unser Tool im Code enthaltene Sicherheitslücken finden, wobei die false-positive Rate in einem akzeptablen Bereich liegt. Somit kann eine Bewertung der Sicherheit von fremdem Code stattfinden. Dafür ließen sich auch die Informationen über die aufgefundenen Security-Patterns nutzen.

Viele der von Oracle in den SCG aufgeführten sicherheitsrelevanten Richtlinien lassen sich in ähnlicher Weise als Muster definieren. Die Abdeckung einer Auswahl an Richtlinien für unser Tool und anderer Analysewerkzeuge können Abbildung 2 entnommen werden. Die Nummern entsprechen der Nummerierung in den SCG. Die grau gedruckten Nummern (z.B. 1-1) sind allgemeine Empfehlungen, jedoch keine auffindbaren Sicherheitslücken. Die dunkelgrau markierten Felder zeigen an, dass das entsprechende Tool diese Richtlinie abdeckt. Die blassere Markierung zeigt bei den fremden Tools an, dass diese Richtlinien nur teilweise abgedeckt werden.

Bei unserem Tool entspricht die blassere Markierung einer Warnung bzw. einem Flaw, bei dem eine geringere Präzision zu erwarten ist. Die unterstrichenen Nummern lassen sich nur mit Annotationen sinnvoll umsetzen bzw. in ihrer Präzision erhöhen. Richtlinien, die eher allgemeinen Empfehlungen entsprechen, wie beispielsweise aus dem Abschnitt *Fundamentals*, werden darin nicht berücksichtigt. Dies betrifft auch die Empfehlungen aus dem Bereich *Access-Control*. Für den Bereich *Injection and Inclusion* eignet sich die Verwendung von Tools, wie Xanitizer, besser als die eines Pattern-Detection-Tools.

Kapitel	Tool	Richtlinie											
Denial of Service	Unser Tool	1-1	1-2	1-3									
	Xanitizer	1-1	1-2	1-3									
	FindBugs	1-1	1-2	1-3									
	PMD	1-1	1-2	1-3									
Confidental Information	Unser Tool	2-1	2-2	2-3									
	Xanitizer	2-1	2-2	2-3									
	FindBugs	2-1	2-2	2-3									
	PMD	2-1	2-2	2-3									
Injection and Inclusion	Unser Tool	3-1	3-2	3-3	3-4	3-5	3-6	3-7	3-8	3-9			
	Xanitizer	3-1	3-2	3-3	3-4	3-5	3-6	3-7	3-8	3-9			
	FindBugs	3-1	3-2	3-3	3-4	3-5	3-6	3-7	3-8	3-9			
	PMD	3-1	3-2	3-3	3-4	3-5	3-6	3-7	3-8	3-9			
Accessibility and Extensibility	Unser Tool	4-1	4-2	4-3	4-4	4-5	4-6						
	Xanitizer	4-1	4-2	4-3	4-4	4-5	4-6						
	FindBugs	4-1	4-2	4-3	4-4	4-5	4-6						
	PMD	4-1	4-2	4-3	4-4	4-5	4-6						
Input Validation	Unser Tool	5-1	5-2	5-3									
	Xanitizer	5-1	5-2	5-3									
	FindBugs	5-1	5-2	5-3									
	PMD	5-1	5-2	5-3									
Mutability	Unser Tool	6-1	6-2	6-3	6-4	6-5	6-6	6-7	6-8	6-9	6-10	6-11	6-12
	Xanitizer	6-1	6-2	6-3	6-4	6-5	6-6	6-7	6-8	6-9	6-10	6-11	6-12
	FindBugs	6-1	6-2	6-3	6-4	6-5	6-6	6-7	6-8	6-9	6-10	6-11	6-12
	PMD	6-1	6-2	6-3	6-4	6-5	6-6	6-7	6-8	6-9	6-10	6-11	6-12
Object Construction	Unser Tool	7-1	7-2	7-3	7-4	7-5							
	Xanitizer	7-1	7-2	7-3	7-4	7-5							
	FindBugs	7-1	7-2	7-3	7-4	7-5							
	PMD	7-1	7-2	7-3	7-4	7-5							
Serialization and Deserialization	Unser Tool	8-1	8-2	8-3	8-4	8-5							
	Xanitizer	8-1	8-2	8-3	8-4	8-5							
	FindBugs	8-1	8-2	8-3	8-4	8-5							
	PMD	8-1	8-2	8-3	8-4	8-5							

Abb. 2: Abdeckung der SCG durch verschiedene Tools

Aktuell ist unser Tool ein wissenschaftlicher Prototyp. Mit ihm ist es möglich etwa zwei Drittel der betrachteten Richtlinien aus den SCG und weitere Patterns aus anderen Quellen, z.B. [Bloc08], zu erkennen. Ziel ist es daher zunächst, den Prototypen um weitere Flaw-Patterns zu ergänzen. Zudem muss die Bedienbarkeit für den Endnutzer erhöht werden und ein Set von (austauschbaren) Annotationen von dem Tool zur Verfügung gestellt werden.

Im Rahmen des Software-Engineering-Prozess schlagen wir die Integration unseres Tools in IDEs und Build-Server vor. Die IDE-Integration ermöglicht, dass die Programmierer bereits während der Implementierung die Risiken angezeigt bekommen und beheben können. Zudem

kann der Build-Server ein Einchecken von Code, bei dem das Tool Sicherheitslücken feststellt, ablehnen.

Sicherheitsanforderungen, die der Softwarearchitekt bereits beim Entwurf berücksichtigen muss, können nun mit den von uns vorgeschlagenen Annotationen dokumentiert und später zur Überprüfung des Quelltextes durch unser Tool genutzt werden. Dies sorgt für zusätzliche Qualität in der Entwicklung. Kommt es trotzdem zu einer Sicherheitslücke, die „by design“ benötigt wird oder meldet das Tool einen false-positive, muss in einer Code-Review sichergestellt werden, dass diese Lücke nicht ausnutzbar ist oder die zu erstellende Software entsprechend angepasst werden.

Durch die oben aufgeführten Maßnahmen kann, mit wenig zusätzlichem Aufwand, die Wahrscheinlichkeit für das Auftreten von Schwachstellen im Software-Endprodukt reduziert werden, da potentielle Sicherheitsrisiken bereits während der Implementierung aufgedeckt werden und nicht erst im Nachhinein in einer Code-Review gesucht werden müssen. Die abschließende Code-Review wird zudem durch das Pattern-Detection-Tool vereinfacht, da dieses auch Entwurfsmuster finden kann.

Literatur

[Apac] Apache: Apache Airavata. Https://airavata.apache.org/index.html.

[BlBS05] A. Blewitt, A. Bundy, I. Stark: Automatic Verification of Design Patterns in Java. *In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’05, ACM, New York, NY, USA (2005), 224–232.

[Bloc08] J. Bloch: Effective Java. The Java Series, Addison Wesley, 2. Aufl. (2008).

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissidis: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing, Addison Wesley (1995).

[HoLi06] M. Howard, S. Lipner: The Security Development Lifecycle. Secure software, Microsoft (2006).

[HoPu04] D. Hovemeyer, W. Pugh: Finding Bugs is Easy. In: *SIGPLAN Not.*, 39, 12 (2004), 92–106.

[Orac15] Oracle: Secure Coding Guidelines for Java SE (2015), http://www.oracle.com/technetwork/java/seccodeguide-139067.html.

[Paul11] S. Paulus: Basiswissen Sichere Software: Aus- und Weiterbildung zum ISSECO Certified Professional for Secure Software Engineering. dpunkt.verlag (2011).

[pmd17] PMD: Don’t Shoot the Messenger (2017), https://pmd.github.io/.

[Rigs17] Rigs-IT: XANITIZER (2017), https://www.rigs-it.net/index.php/product.html.

[SFBHB+06] M. Schuhmacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerland: Security Patterns: Integrating Security and Systems Engineering. Wiley Verlag (2006).

[ShOl06] N. Shi, R. A. Olsson: Reverse Engineering of Design Patterns from Java Source Code. *In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE ’06, IEEE Computer Society, Washington, DC, USA (2006), 123–134.