Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Florian Gerdes

MASTER THESIS

Sweble Security Programming Plugin

Security Rules Engine

Submitted on 02.10.2017

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

[CITY], [DATE]

License

This work is licensed under the Creative Commons Attribution 4.0 International license

(CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

_____

[CITY], [DATE]

Abstract

Within the scope of industry 4.0 and digitalization, there is a growing danger of cyber crime and security attacks, causing huge harm for digital business. Therefore, in nowadays software development, IT-security is regarded as a quality criterion, determining the success of a product or project. Nevertheless, the complexity of security attacks, vulnerabilities and software development as a whole, complicates the reliable protection against and mitigation of security attacks.

To support software engineers to develop more secure software, this thesis shows the concept and presents a prototype of a software security rules methodology called *Serum*. Serum is designed to help software developers and all other project members in creating a more secure software. A domain-specific language was designed and integrated into a global knowledge management system (*Sweble*), to allow modeling and describing software assets, associated security attacks as well as known countermeasures. A second component, using the gathered security knowledge, was implemented, focusing on the support of software architects during the creation of a threat- and risk analysis. To facilitate the consideration of security even more a custom test- and dashboard system allows developers and test architects to monitor their contribution towards a more secure system.

The thesis should provide a basis for a holistic security support during all phases of the software development life cycle.

Zusammenfassung

Cyber Kriminalität und sicherheitsrelevante Angriffe werden im Rahmen der Digitalisierung und Industrie 4.0 zur enormen Gefahr für moderne, digitale Geschäftsmodelle. Deshalb wird IT-Sicherheit in der heutigen Softwareentwicklung als ausschlaggebendes Qualitätskriterium für den Erfolg eines Software Projekts oder Produkts betrachtet. Doch die Vielzahl von sicherheitsrelevanten Angriffen und die Verwundbarkeit von verteilten Softwaresystemen sowie die Komplexität der Softwareentwicklung im Ganzen erschweren den zuverlässigen Schutz oder die Schadensminderung sicherheitsrelevanter Angriffe.

Im Rahmen dieser Arbeit wurde prototypisch eine Software Infrastruktur namens *Software SEcurity RUles Methodology (Serum)* entwickelt. *Serum* nutzt das Konzept einer Security Rules Engine. Diese soll Softwareentwickler und alle anderen Projektbeteiligten bei der Entwicklung sicherer Software unterstützen. Eine domänenspezifische Sprache zur Modellierung und Beschreibung von Software Komponenten, sicherheitsrelevanter Attacken und bekannter Gegenmaßnahmen wurde entworfen. Diese Sicherheitsentitäten werden verknüpft und in ein globales Wissensmanagementsystem, genannt *Sweetly Enabling the Web (Sweble)* integriert. Die gesammelten Sicherheitsinformationen sollen Software Architekten bei der Gefahren- und Risikoanalyse ihrer Software helfen. Ein anpassbares Test- und Dashboard vereinfacht die Durchführung und Überwachung von Sicherheitsmaßnahmen im weiteren *Software Development Lifecycle (SDLC)*. Die Arbeit soll die Basis für einen ganzheitlichen, den ganzen Softwarelebenszyklus betrachtenden Prozess der Sicherheitsgewährleistung legen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

1.1 Motivation

Die Digitalisierung verändert unsere Wirtschaft und Gesellschaft grundlegend. Unternehmen bietet die Digitalisierung gewaltige Chancen, um Innovationen voranzutreiben und durch weltweites Agieren neue Märkte zu gewinnen. Gleichzeitig ändern sich Branchenstrukturen und durch die neuen datenbasierten Geschäftsmodelle und Geschäftsfelder entsteht ein vielfältiger, agiler Wettbewerb.

Doch immer mehr Cyber Attacken gefährden die digitale Industrie und richten zunehmend größere Schäden an. Dabei spielen auch Entwicklungen wie der verstärkte Einsatz eingebetteter Systeme oder cloudbasierte Datenspeicherungen eine wichtige Rolle. Stichwörter wie „Industrie 4.0“ oder „Smart Home“ ziehen unweigerlich die Frage nach mehr IT-Security nach sich. Zudem hat der wirtschaftliche Schaden von Sicherheitslücken und Informationsverlust in den verschiedensten Anwendungen den dringenden Bedarf an Verbesserung der Security gezeigt.

Software Security ist ein eher neuer Bereich der Software Qualitätssicherung. Unter Software Security versteht man den Schutz des Software Systems vor unerwünschtem Zugriff oder Angriffen von außerhalb. Um diesen Schutz des Systems zu gewährleisten werden beispielsweise Authentifizierungs- oder Verschlüsselungsverfahren eingesetzt.

(Abolhassen, 2017) (Fischer, Steinacker, Bertram, & Steinmetz, 1998) (Graham, Howard, & Olson, 2010) (Kriha & Schmitz, 2008) (Partner, 2017) (Singer & Friedman, 2014)

1.2 Security Testen heute

In den letzten Jahren gab es bereits einige Fortschritte im Bereich der Security Assurance, wie z.B. (Black, Kass, & Fong, 2006). Besonderer Fokus lag dabei auf Methoden zur statischen Code Analyse, zu Szenario-abhängigen dynamischen Tests und zu den Architektur Evaluationsverfahren zur Sicherstellung von Software Security während bestimmter Entwicklungsphasen.

Grundsätzlich kann man die Ansätze und Methoden des Security Testens heute in die Kategorien Architektur, Code, Test und Prozess unterteilen.

In die Kategorie Architektur fallen alle Methoden, die nur auf Basis der groben Softwareanforderungen oder der Architektur einer Software arbeiten. Besonders populär sind Szenario- oder Anwendungsfallmodelle, wie in (Kazman, Klein, & Clement, 2000) oder (Pauli & Xu, 2005).

Die Kategorie Code umfasst vor allem Regelsätze und Techniken zur statischen Code Analyse. Beispiele sind die Sammlungen (Neuninger, 2008), (Svoboda, Flynn, & Snavely, 2016) und (Object Management Group, 2016).

Alle dynamischen Tests, das heißt Verfahren, die auf Basis von tatsächlich ausführbarer Software laufen, fallen in die Kategorie Test. Die Ansätze dieser Kategorie sind anwendungsspezifisch. Die Auswahl und Ausführung der Testfälle basiert oft auf Modellen aus dem Softwaredesign, wie den *Unified Modeling Language (UML)* Klassendiagrammen, oder auf formalen Sprachen. Beispiele für dynamische Sicherheitstests sind in den Arbeiten (Al-Shaer, El-Atawy, & Samak, 2009), (Bozic & Wotawa, 2014) oder (Elrakaiby, Mouelhi, & Le Traon, 2012) zu finden.

Die letzte Kategorie „Prozess“ umfasst alle Ansätze, die den Prozess der Softwareentwicklung speziell auf Sicherheitsanforderungen hin anpassen. Oft werden dabei bekannte Ansätze um neue Überlegungen aus dem Bereich der Sicherheit erweitert. Ein Beispiel ist die Integration des *Security Architecture Approaches* (*SABSA)* in die *Enterprise Architecture Methodology (TOGAF)* (Open Group TOGAF-SABSA Integration Working Group, 2011).

Trotz des großen methodischen Fortschritts in der Security Assurance haben sich in den letzten 15 Jahren die erfolgreichsten Security Angriffe aus den Sammlungen des *Open Web Application Security Project* (*OWASP)*, des *SysAdmin, Networking and Security (SANS)* Instituts und der *Common Weakness Enumeration (CWE)* kaum geändert (Schoenfield, 2015). Es bleibt weiterhin eine Herausforderung Security ganzheitlich zu betrachten und als einen natürlichen Teil in den Softwareentwicklungszyklus (SDLC) zu integrieren.

1.3 Grundlagen

Im nachfolgenden Abschnitt werden die Grundbegriffe und grundlegenden Zusammenhänge der IT-Security eingeführt und erläutert. Diese definieren die Relationen und Entitäten der Security Domäne und sind für das Verständnis der Arbeit und der Arbeitsziele grundlegend.

1.3.1 Security

Allgemein versteht man unter Security den Schutz eines Software System vor unerwünschtem Zugriff oder Angriffen von außerhalb. Eine Verletzung der IT-Security hat ein unautorisiertes Ergebnis, in Form von z.B. unerlaubtem Ressourcenzugriff, Eindringen von Malware oder korrumpierten Daten zur Folge.

Der Begriff der IT-Security lässt sich weiter in drei Aspekte unterteilen.

Confidentiality (Vertraulichkeit): Unter Vertraulichkeit versteht man, dass Informationen nur autorisierten Entitäten zur Verfügung stehen. Alle Informationen, nicht nur der Inhalt einer Nachricht, sondern auch Meta- oder Verbindungsinformation, sind privat und als solche nur nach Berechtigung einsehbar. Bekannte Maßnahmen, um die Vertraulichkeit von Information zu gewährleisten, sind Passwort-Authentifizierung und Verschlüsselung.

Integrity (Integrität): Integrität meint den Schutz von Informationen vor unberechtigter Manipulation, ob mit Absicht oder durch einen Software- oder Hardwarefehler hervorgerufen. Der Zustand der Daten muss jederzeit identisch mit dem Zustand unmittelbar nach dem letzten, autorisierten, bearbeitenden Zugriff sein. Im weiteren Sinne kann sich Integrität auch auf ein Informationssystem im Allgemeinen, also den manipulationsfreien Zustand eines Systems, nicht nur einer Information, beziehen.

Availability (Verfügbarkeit): Verfügbarkeit meint die Möglichkeit des uneingeschränkten, autorisierten Zugriffs auf Ressourcen. Ähnlich wie bei Integrität kann sich auch Verfügbarkeit auf jeglichen Aspekt einer technischen Infrastruktur beziehen und beschränkt sich nicht nur allein auf Informationen. Der wohl bekannteste Angriff auf die Verfügbarkeit eines Systems ist die *Denial of Service-Attacke*.

1.3.2 Begriffe

Um Missverständnisse zu vermeiden, folgen Definitionen zu den wichtigsten Begriffen der IT-Security. Da es sich bei denen im Folgenden beschriebenen Objekten um Objekte aus der Security Domäne handelt, werden diese in der weiteren Arbeit auch als Security Entitäten bezeichnet.

1.3.2.1 Asset (Schutzobjekt)

Im Zentrum jeder Diskussion um IT-Security steht ein Asset oder Schutzobjekt dessen IT-Security es zu bewahren gilt. Ein Asset kann, wie im vorherigen Abschnitt unter Confidentiality bereits erwähnt, im Allgemeinen eine Information, aber auch jeder andere Aspekt einer technischen Infrastruktur sein. Grob lässt sich ein Asset unterteilen in logische Entitäten, zum Beispiel Account, Prozess, Funktionalität oder Daten und physische Entitäten, zum Beispiel Computer, Komponenten, Netzwerke oder Kommunikationskanäle.

1.3.2.2 Vulnerability (Schwachstelle)

Eine Vulnerability ist ein Fehler oder Mangel im System, der ausgenutzt werden kann, um einem Asset zu schaden, beziehungsweise eine unautorisierte Aktion durchzuführen. Bei einer Passwort-Authentifizierung könnte eine Schwachstelle zum Beispiel die unverschlüsselte Weiterleitung des Passworts sein. Ein System ohne Schwachstelle gibt es nicht. Schwachstellen sind allgegenwärtig, jedoch oft nur schwierig zu finden oder mit großer Mühe auszunutzen. Oft spricht man von einer Schwachstelle als einen Fehler in der Implementierung einer Software oder Hardware Komponente. Doch eine Schwachstelle kann auch in der Konzeption, im Design oder der Konfiguration liegen.

1.3.2.3 Threat (Gefährdung)

Ein Threat ist eine Aktion, die durch die Ausnutzung einer oder mehrerer Vulnerabilities die IT-Security eines Assets verletzen könnte. Ein Threat könnte beispielsweise das Eindringen eines Schadprogramms durch einen Email Anhang oder das unautorisierte Mitlesen einer unverschlüsselten Datei sein.

1.3.2.4 Threat (Bedrohung)

In der deutschen Literatur unterscheidet man weiterhin oft zwischen Gefährdung und Bedrohung. Eine Bedrohung ist eine Aktion oder eine Entität, die die Security eines Assets verletzen könnte. Dabei ist eine Bedrohung allein für ein perfektes System noch nicht gefährlich. Erst durch das Ausnutzen einer Schwachstelle entwickelt sie sich zu einer Gefährdung. Eine Bedrohung kann absichtlich oder unabsichtlich bestehen. Beispiele für eine Bedrohung sind ein Schadprogramm oder ein unautorisierter Nutzer/Hacker. Der Zusammenhang zwischen Gefährdung und Bedrohung wird in Abbildung 1 noch einmal veranschaulicht.

1.3.2.5 Attack (Angriff)

Ein Angriff ist eine konkrete Instanz einer Gefährdung. Werden zum Beispiel die Benutzerdaten eines Onlinedienstes durch einen Hacker publik, so wurde der Service attackiert. Eine Attacke verletzt die Vertraulichkeit, Integrität oder Verfügbarkeit eines konkreten Assets, unter Ausnutzung mindestens einer konkreten, diesem Asset eigenen Schwachstelle.

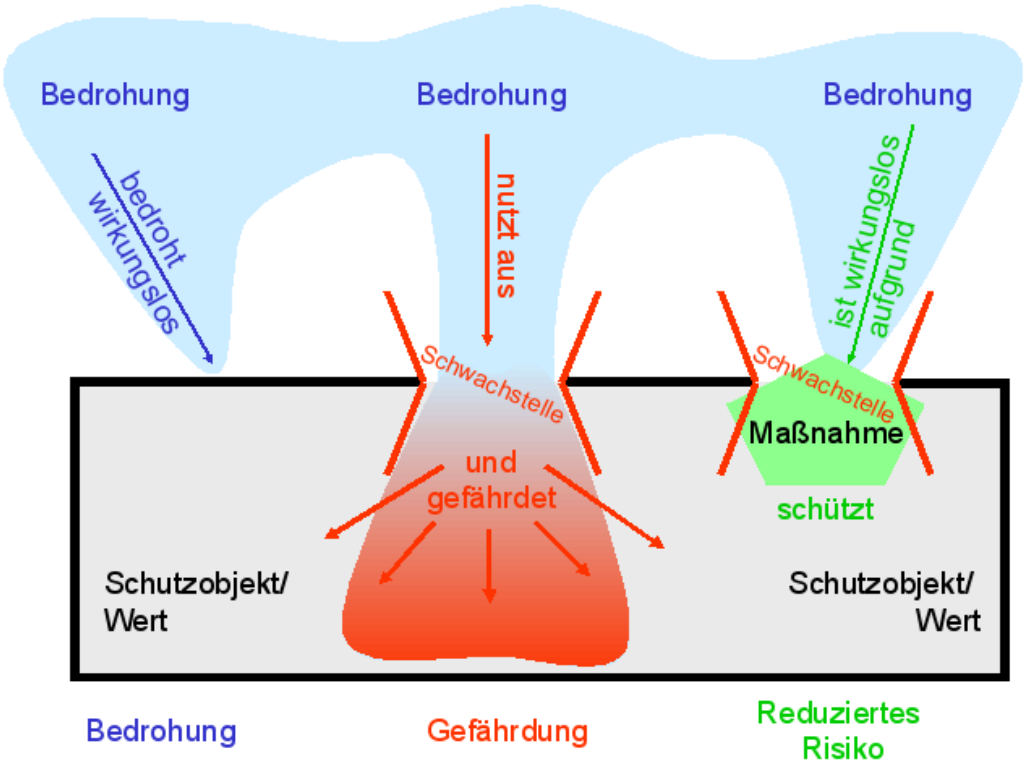

Abbildung 1 *Grafische Veranschaulichung der Security Entitäten*

(ISi-Projektgruppe, 2011)

1.3.2.6 Countermeasure (Gegenmaßnahme)

Eine Gegenmaßnahme, manchmal auch Defense genannt, ist eine Aktion zur Prävention vor Gefährdungen (Prevention), zum Schutz vor Gefährdungen (Protection) oder zur Minimierung des Schadens eines erfolgreichen Angriffs (Mitigation). Der Zeitpunkt zu der die Gegenmaßnahme eingeführt wird und der Aufwand der Gegenmaßnahme können enorm variieren.

(Feinman, Goldman, Wong, & Cooper) (Howard & Longstaff, 1998) (Lehtinen, Russell, & Gangemi Sr., 2006) (Kostopoulos, 2013) (Lee, 2013)

1.4 Herausforderungen im Bereich Security

Trotz des Fortschritts der letzten Jahre bleibt es eine Herausforderung, Security als einen natürlichen und ganzheitlichen Teil in den Softwareentwicklungsprozess zu integrieren. Die Probleme reichen von der Akzeptanz einer Securityanalyse bis hin zur technischen Umsetzung und Überwachung der Testergebnisse. Außerdem können große Softwaresysteme bisher nur partiell und auf Komponentenebene getestet werden. Die Methoden bleiben, trotz der Bemühungen, szenariospezifisch. Zudem erschwert die Unerfahrenheit von Entwicklern, Architekten oder Projektleitern die Auswahl und Evaluierung der richtigen Techniken und Prozesse.

1.4.1 Fehlende Kenntnis von Gefährdung

Die Komplexität von IT-Security bedingt auch das fehlende allgemeine und natürliche Verständnis der Problematik. Einem Software Architekten oder Entwickler ohne Erfahrung im Bereich IT-Security fehlt in der Regel das Wissen von Bedrohungen und Gefährdungen für die zu entwickelnde oder entwerfende Software. Vielleicht hat der Einzelne auch noch nie einen Security Angriff erlebt.

Die Konsequenz ist eine unsichere Software, mit Schwachstellen gegen die man hätte vorgehen können. Die Gefahr einem bekannten Angriff zum Opfer zu fallen ist groß.

1.4.2 Fehleinschätzung von Gegenmaßnahmen

Wenn der Architekt weiß, welchen Bedrohungen und Gefahren aus dem Bereich Security, sein System ausgesetzt ist, so dürfte ihm vielleicht trotzdem, aufgrund seiner geringen Erfahrung mit der Problematik, eine Auswahl und Einschätzung der möglichen Gegenmaßnahmen schwer fallen. Für ein bekanntes Angriffsmuster oder eine Software Security Vulnerability kann es keine, oder auch beliebig viele Gegenmaßnahmen geben. Kann der Architekt oder Entwickler die Gegenmaßnahmen nicht einschätzen, riskiert er großen Schaden durch einen Security Angriff oder muss enormen Zusatzaufwand für die Entwicklung aufbringen.

1.4.3 Technische Komplexität von Gegenmaßnahmen

Oft sind die Gegenmaßnahmen auch mit hohem Aufwand und Komplexität bei der technischen Umsetzung verbunden. Damit ist die Installation und Konfiguration der nötigen Programme und Tools oder die kognitive Komplexität einer Designmethode gemeint. Allein das Wissen um benötigte Gegenmaßnahmen reicht nicht aus, um selbständig die Security einer Software zu verbessern. Kapitel 1.2 bietet einen Einblick in die Komplexität der Thematik.

1.4.4 Schwierigkeiten in der Nachverfolgbarkeit

Schließlich bedingt die große Anzahl an Tools und Prozessen, die bei der Softwareentwicklung als auch beim Security Testen zum Einsatz kommen, dass die Ergebnisse individuell überprüft und verarbeitet werden müssen. Dabei kann man schnell den Überblick verlieren. Auch ist die Nachverfolgbarkeit der ursprünglichen Anforderungen und Bedrohungen oder die Dokumentation der Security Maßnahmen, bzw. die Nachverfolgbarkeit im Entwicklungsprozess zwischen den Artefakten oftmals nicht gegeben.

1.4.5 Fehlende zentrale Infrastruktur

Zusätzlich zu den obigen, den Prozess betreffenden Herausforderungen fehlt eine zentrale Infrastruktur, um Security Informationen strukturiert und formal zu sammeln. Für Security Experten gibt es bisher keine Möglichkeit ihr Wissen über Security Bedrohungen oder Gegenmaßnahmen zu teilen. Die bisherigen Versuche, wie die OWASP (The OWASP Foundation, 2013) und CWE (MITRE, 2017) Sammlungen, sind unzureichend formal oder in ihrer Struktur nicht auf ein verteiltes Konzept ausgelegt. Es fehlt eine Struktur um eine zentrale, öffentliche Sammlung der Security Informationen für Security Experten zu ermöglichen und dabei eine unternehmensweite Integrität der Daten zu ermöglichen. Das ist der Grundstein für eine weitreichende Security Assurance.

2 Ziel der Arbeit

Ziel der Arbeit ist eine holistische Methodik und eine prototypische Infrastruktur zur Unterstützung bei der Entwicklung sicherer Software, im Folgenden *Sofware SEcurity RUles Methodology (Serum)* genannt, zu erstellen. *Serum* ist in Kooperation mit der zentralen Forschungsabteilung der *Siemens AG* entwickelt worden und soll die Arbeit mit einer Zielsoftware im gesamten Software Lebenszyklus (SDLC) unterstützen. Dafür ist eine Security Rules Language definiert worden. Diese soll die Definition und assoziative Modellierung von Security Threats, Projekt Assets und Security Countermeasures ermöglichen. Zwei weitere Komponenten *Serums* sollen die prototypische Unterstützung bei der Erstellung einer Threat- and Risk Analysis (TRA), der Anwendung der Security Regeln auf eine bestehende Software Komponente, sowie der Analyse der Testergebnisse, zeigen.

Serum ist in das Infrastrukturprojekt um *Sweetly Enabling the Web (Sweble)* eingebettet worden. *Sweble* ist ein Open Source Projekt der *Open Source Research Group (OSR Group)* der Friedrich-Alexander-Universität Erlangen-Nürnberg, zur Verwaltung strukturierter Daten (Kapitel 3.2). Es stellt die Infrastruktur zur Sammlung von Security Informationen bereit. Im Rahmen dieser Arbeit soll die Nutzung von *Sweble* gezeigt und erweitert werden. (Dohrn & Riehle, 2011)

Besonderer Fokus bei der Entwicklung und Evaluierung von *Serum* soll, neben der prototypischen Herangehensweise, auf der Erweiterbarkeit der Arbeit liegen. Die Arbeit wird als solches nicht nur als konzeptuelle Grundlage (Prototyp), sondern auch als tatsächliche Arbeitsgrundlage, in der *Sweble* Infrastruktur betrachtet.

2.1 Vision

Die Vision dieser Arbeit ist die Unterstützung bei der Herstellung und Wartung von sicherer Software in allen Domänen. Unter Berücksichtigung der Schwierigkeiten im Bereich der Security Assurance soll es möglich werden, das Wissen um Bedrohungen, Gefahren, Verwundbarkeiten und Gegenmaßnahmen strukturiert zu sammeln und zu formalisieren. Das erworbene Wissen soll dann verwendet werden, um ein Security Level, ähnlich einer Gewährleistung von Sicherheit von Softwareprojekten oder -produkten, zu definieren. Das Security Level entspricht also dem gewünschten Security Qualitätslevel der Zielsoftware. Mithilfe der in dieser Arbeit entstandenen Software soll die Security Gewährleistung, einem „Serum“ ähnlich, auf die Zielsoftware übertragen werden. Abhängig allein vom Security Level und den Assets der Zielsoftware soll eine definierte Bandbreite von Gegenmaßnahmen vorgeschlagen und ein Security Standard angewendet werden.

2.2 Verwandte Arbeiten und Abgrenzung

Diese Arbeit soll explizit keine weitere Technik vorstellen, die sich lediglich auf eine bestimmte Phase des Software Lebenszyklus oder auf ein spezielles Szenario bezieht. Stattdessen soll sie die Entwicklung von Software möglichst holistisch betrachten und die Security Assurance durch ein benutzernahes, ganzheitliches Konzept verbessern. Dabei ist es wichtig, alle in den Prozess der Security Assurance involvierten Stakeholder intensiv einzubinden und nicht durch neue technische Hindernisse abzuschrecken.

Verwandte Arbeiten beziehen sich daher meist nur auf Komponenten, die denen von *Serum* entsprechen. Zur domänenspezifischen Sprache und zum Wissensmanagement *Serums* verwandte Arbeiten sind die *Common Weakness Enumeration* (MITRE, 2017) und die Listen der *OWASP Foundation* (The OWASP Foundation, 2013). Beide haben ein semi-formales Domänenmodell als Grundlage und haben über die letzten Jahre enorm zur Sammlung von Security Informationen beigetragen. Die *CWE* ist in Struktur und Aufbau auf Gegenmaßnahmen konzentriert. Die *OWASP* Sammlung beinhaltet ein formales und spezifisches *Risk-Modeling*, im Konzept den Security Typen ähnlich.

Verwandte Arbeiten zur Unterstützung der TRA sind die Tools *SD Elements* von *Security Compass* (Security Compass, 2017) und *IriusRisk – Threat Modeling Tool* von *Continuum Security* (Continuum Security, 2017). *IriusRisk* konzentriert sich auf die Unterstützung von Entwicklung und Test. *SD Elements* arbeitet unter anderem mit Daten aus den *CWE* und *OWASP* Sammlungen und erlaubt die teilautomatische Ausführung verschiedener Testing

Tools. Der Schwerpunkt von *SD Elements* liegt auf der Application-Management Integration. *SD Elements* wurde von der *Siemens AG* erprobt, jedoch aufgrund der inflexiblen Definition neuer Angriffe und Assets abgelehnt. Hieraus ergibt sich die Anforderung an *Serum* eine flexiblere Definition der Security Entitäten zu ermöglichen.

2.3 Stakeholder

Die Grundlage zur Verbesserung der Security einer Software ist das Wissen um eventuelle Schwachstellen und Bedrohungen und deren Gegenmaßnahmen. Security Experten haben dieses Wissen und sind somit eine wichtige Stakeholdergruppe. Der Erfolg *Serums* steht oder fällt mit deren Einbindung in jede Phase des Softwareentwicklungsprozesses. *Serum* kann nur durch die zuverlässige und regelmäßige Nutzung von Security Experten dabei helfen, die Security in der Softwareentwicklung zu verbessern.

Um die sichere Softwareerstellung schon möglichst früh und verstärkt im Lebenszyklus zu unterstützen, richtet sich die Arbeit an Software Architekten. Wird die Security einer Software zu spät in der Entwicklung bedacht, kann dies enorme Kosten oder mangelnde Security nach sich ziehen. Software Architekten haben zudem meist Einfluss auf jede Phase bei der Erstellung und Wartung von Software und können so zu jedem Zeitpunkt Einfluss auf die Security eines Softwareprodukts oder Projekts ausüben.

Software Entwickler haben durch ihre Arbeit meist einen besonderen Blick für die Auswirkungen von Qualitätsmaßnahmen auf die tatsächliche Software und sind am meisten durch statische Codeanalysen oder zusätzlichen Aufwand bei Modultests betroffen. Damit bilden Software Entwickler eine dritte wichtige Stakeholdergruppe, deren Bedürfnisse bei der Entwicklung von sicherer Software unbedingt bedacht werden müssen.

Zuletzt richtet sich *Serum* an Testarchitekten und Tester. Ein Großteil der Gegenmaßnahmen wird erst in den jeweiligen Testphasen durchgeführt und beeinflusst damit direkt die Arbeit der Tester. Die Testarchitekten sollen bei der technischen Umsetzung, bei der Evaluierung und der Beobachtung von Gegenmaßnahmen auf die Zielsoftware unterstützt werden.

Prinzipiell richtet sich *Serum* an alle, die an dem Prozess der Security Assurance beteiligt sind. Stakeholder aus dem Bereich der *Sweble* Infrastruktur und der Zusammenarbeit von *Serum* und *Sweble* wurden explizit aus obiger Aufzählung ausgeschlossen.

2.4 Anforderungen

Die einzelnen und detaillierten Anforderungen an *Serum* orientieren sich direkt an den in Abschnitt 1.4 aufgezählten Security Herausforderungen und der Auswahl der oben genannten Stakeholder. Weitere Anforderungen in Bezug auf die allgemeine Vorgehensweise, die *Sweble* Infrastruktur und wichtige Qualitätskriterien sind dem ersten Abschnitt aus Kapitel 2 zu entnehmen. Der Appendix enthält weitere, detaillierte Information zu den Anforderungen und deren Erarbeitung (z.B.: Anwendungsfalldiagramme).

Vorab ist es wichtig zu erwähnen, dass sich die Arbeit und die Anforderungen primär auf die Erstellung einer Infrastruktur beziehen. Das heißt, dass das erstellte System den Rahmen für die Spezifikation und den Zugriff auf Security Wissen bieten soll. Die Füllung des Systems mit den oben genannten Daten, in Form von Bedrohungen, Gegenmaßnahmen u.Ä. ist explizit nicht Teil dieser Arbeit. Das gleiche gilt für die Spezifikation und technische Beschreibung von konkreten Gegenmaßnahmen (z.B. Toolvorschläge) und die Füllung des Dashboards (siehe Abschnitt 1.4.4) mit konkreten Analyseergebnissen. Um trotzdem ein Gesamtbild der Infrastruktur, Beispiele in der schriftlichen Arbeit und Analyseergebnisse in der praktischen Arbeit zeigen zu können, wurden exemplarische Security Daten generiert.

2.4.1 Wissensmanagement von Gefährdungen

Epic #01: Als Security Experte und Nutzer von *Serum* habe ich die Möglichkeit mein Wissen um Security Threats strukturiert zu spezifizieren, zu manipulieren, mit Software System Assets zu assoziieren und mit den beteiligten Software Architekten zu teilen.

Epic #02: Als Software Architekt und Nutzer von *Serum* habe ich Zugriff auf strukturiertes Wissen über Bedrohungen und Gefährdungen (meiner Zielsoftware).

Durch die *Serum* Infrastruktur muss sichergestellt werden, dass das gesammelte Wissen um Bedrohungen und Gefährdungen möglichst vollständig, integer und aktuell ist. Die systemeigenen Funktionalitäten sollen unterstützend tätig sein diese Informationseigenschaften zu gewährleisten. Ein Anspruch auf absolute Vollständigkeit/Integrität/Aktualität kann natürlich nicht erhoben werden. Um ein einfaches Verständnis der Daten zu erreichen und eine Weiterverarbeitung zu begünstigen, soll das Wissen in strukturierter Form, einem klaren Threat-Model entsprechend, vorliegen/abgerufen/spezifiziert/manipuliert werden können. Der Zugriff soll wahlfrei, bedrohungsspezifisch oder im Rahmen einer Threat- and Risk Analysis der Zielsoftware möglich sein. Um eine Analyse der Threat-Daten entsprechend einer vorliegenden Zielsoftware

zu erlauben, muss es den Security Experten möglich sein, Threats und Assets zu verknüpfen. Die Weitergabe, bzw. das „Teilen“ des Wissens mit Software Architekten aus Epic #01, bezieht sich auf Epic #02 und soll gemäß dieser Anforderung möglich sein.

2.4.2 Wissensmanagement von Gegenmaßnahmen

Epic #03: Als Security Experte und Nutzer von *Serum* habe ich die Möglichkeit mein Wissen um Security Countermeasures strukturiert zu spezifizieren, zu manipulieren, mit Security Threats zu assoziieren und mit Software Architekten zu teilen.

Epic #04: Als Nutzer von *Serum* habe ich Zugriff auf strukturiertes Wissen über Countermeasures (für die Threats meiner Zielsoftware).

Die Daten über Countermeasures sollen in den Eigenschaften ähnlich wie in den Anforderungen in 2.4.1 beschrieben gestaltet werden. Auch Zugriff und Form der Daten, Threat-Model, wahlfreier-/TRA- Zugriff soll logisch denen aus 2.4.1 entsprechen. Für die TRA-Funktionalität soll es den Security Experten möglich sein, die Countermeasures den entsprechenden Threats zuzuordnen.

Epic #05: Als Nutzer von Serum habe ich die Möglichkeit durch Spezifikation der Assets, eines Security Levels und mithilfe der von *Serum* gesammelten Daten eine automatische Threat- and Risk Anaylsis der Zielsoftware durchzuführen.

Die durch Security Experten vermerkten Assoziationen, Threats und Countermeasures sollen speziell die Software Architekten bei der Threat- and Risk Analysis ihrer Zielsoftware unterstützen. Durch Auswahl von Software Assets sollen die Security Threats der Zielsoftware und Countermeasures angezeigt werden. Das Security Level soll die Auswahl und Filterung der Threats und Countermeasures vorgeben. Dabei sollen unter Auswahl der gleichen Assets, der gleichbleibenden *Serum* Daten und dem gleichem Security Level auch die gleichen Gefährdungen und Gegenmaßnahmen vorgeschlagen werden.

2.4.3 Unterstützung bei technischer Umsetzung von Gegenmaßnahmen

Epic #06: Als Nutzer von *Serum* habe ich die Möglichkeit mein Wissen um die Durchführung, Vor- und Nachbereitung von Gegenmaßnahmen strukturiert zu spezifizieren, zu manipulieren, mit Security Coutnermeasures zu assoziieren und mit anderen, projektbeteiligten Nutzern zu teilen.

Epic #07: Als Nutzer von *Serum* habe ich Zugriff auf strukturiertes Wissen über Durchführung, Vor- und Nachbereitung von Gegenmaßnahmen.

Auch für Epic #06 und Epic #07 gelten die Eigenschaften, Zugriff und Form der Daten wie für Epic #01 und Epic #02. Die Durchführungsdetails von Gegenmaßnahmen sollen in einer TRA einsehbar sein. Beispiele für Informationen solcher Art wären Toolempfehlungen, Installations- und Integrationsanleitungen oder dergleichen.

Außerdem soll der technische Rahmen für die Integration von teil-/automatischer Umsetzung/Durchführung der Gegenmaßnahmen geschaffen werden.

2.4.4 Nachverfolgbarkeit

Epic #08: Als Nutzer von *Serum* habe ich die Möglichkeit die Resultate meiner Security Countermeasures zentralisiert einzusehen.

Unter Spezifikation von technischen Details, soll ein zentrales Dashboard erstellt werden, das Informationen über die durchgeführten Gegenmaßnahmen enthält. Diese Informationen können Erfolg/Misserfolg, aber auch detaillierte Daten, wie Code Abdeckung oder Durchführungsdauer sein.

2.4.5 *Sweble* Integration

Epic #09: *Serum* zeigt die Nutzung von *Sweble* als Grundlage für die Sammlung, Verwaltung, Versionierung und anwendungsspezifischen Anwendung strukturierter Daten.

Serum soll als Ganzes die Nutzung von *Sweble* prototypisch zeigen und fördern. Dabei soll *Sweble* als zentraler Knotenpunkt für technische Information und Programmierung positioniert werden. Besonders bei den Anforderungen Epic #01, Epic #03, Epic #06 und Epic #08 soll auf die Integration von *Serum* und *Sweble* und eine zukünftige infrastrukturelle Zusammenarbeit geachtet werden.

3 Architektur und Design

3.1 Übersicht

Die Funktionsweise und die Komponenten der *Serum* Infrastruktur lassen sich grob in vier Schritte unterteilen wie in der nachfolgenden Abbildung dargestellt. Auf der rechten Seite der Abbildung sieht man die betreffenden Stakeholder.

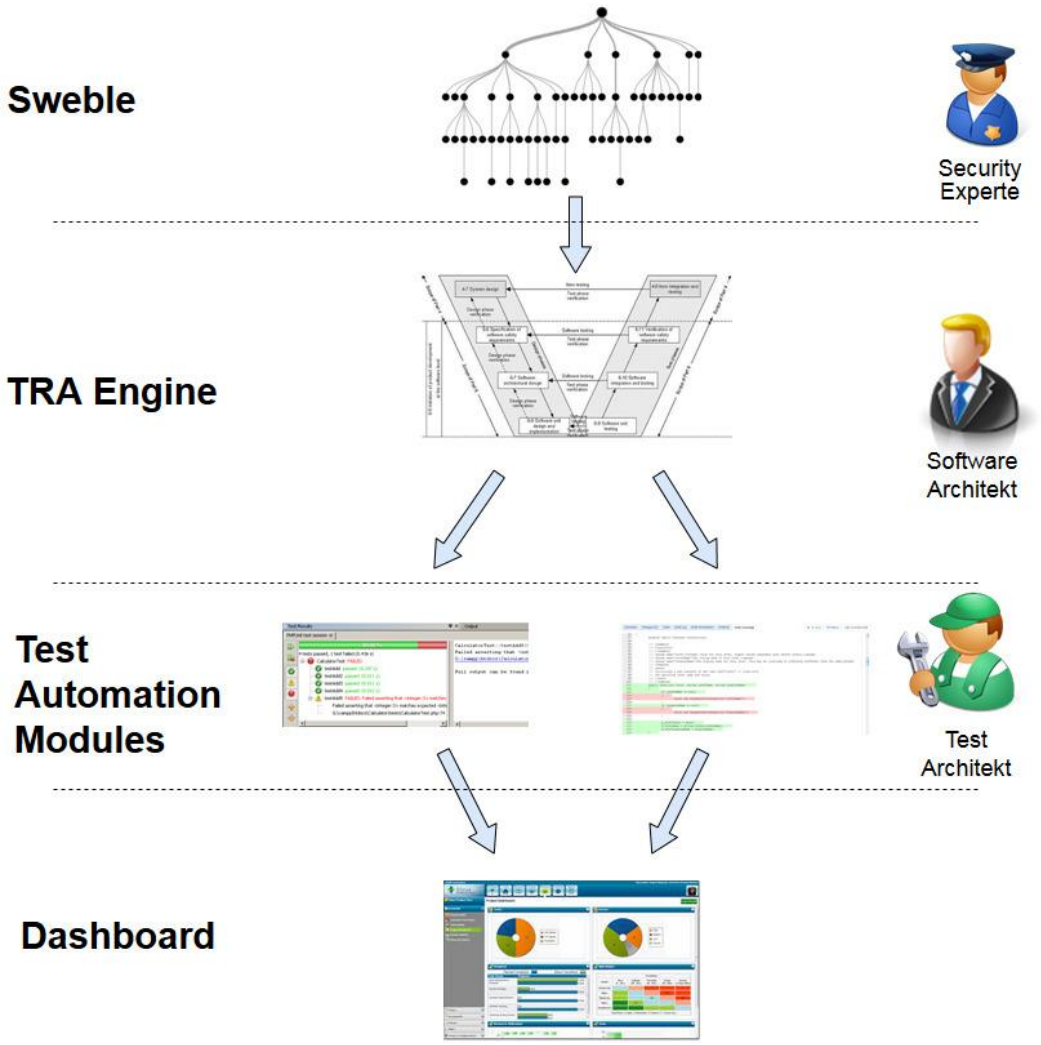

Abbildung 2 *Serum Skizze*

Abbildung 3 zeigt die Komponenten *Serums* in einem statischen Komponentendiagramm. In Abbildung 4 sieht man hingegen die Komponenten *Serum*s im Software Development Lifecycle (SDLC).

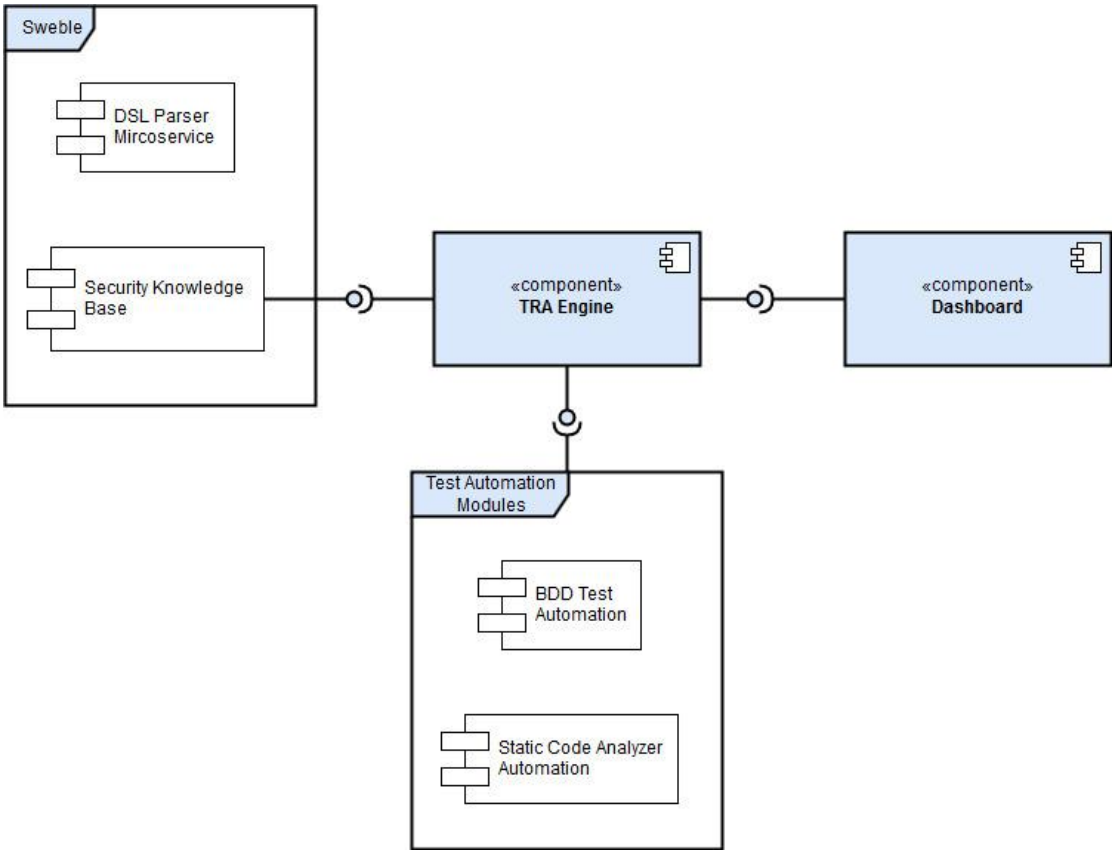

Abbildung 3 *Serum Komponenten Übersicht*

Die vier Komponenten von Serum lassen sich relativ eindeutig den Anforderungen aus dem vorhergehenden Kapitel zuordnen. *Sweble*, Abbildung 2 oben, Abbildung 3 links und Abbildung 4 oben, kann als zentraler Knotenpunkt betrachtet werden, der es ermöglicht technische, strukturelle Informationen zu sammeln, zu versionieren und zu verwalten. Ein DSL Parser Plugin erlaubt die Verarbeitung und Verwaltung der Security Informationen. *Sweble* ist die einzige Komponente *Serum*s, die losgelöst vom SDLC verwendet wird. Somit können Security Experten jederzeit neue Attack Pattern oder Countermeasures mithilfe einer domänenspezifischen Sprache in *Sweble* einspeisen. Die *TRA Engine* greift später auf die verwalteten Daten zu.

Die *TRA Engine* kommt in der Software Architektur Phase zum Einsatz. Sie soll vorwiegend durch Software Architekten oder Testarchitekten genutzt werden. Ziel der *TRA Engine* ist die Automatisierung einer TRA und der damit verbundenen Darstellung bedrohungsspezifischer Angriffe und Gegenmaßnahmen. Mithilfe einer Architekturbeschreibung durch Assets und der Security Informationen aus *Sweble* wird die automatische TRA durchgeführt. Wichtigstes Kriterium zur Auswahl der relevanten Attack Pattern, neben den zu-

vor ausgewählten Assets, ist das Security Level, also das gewünschte Security Qualitätslevel der Software. Das Security Level ist durch die Art und die Anforderungen der Zielsoftware im Entwicklungsprozess vorgegeben. Die Ergebnisse der automatischen TRA sind eine Auswahl von vorgeschlagenen Gegenmaßnahmen und Recommended Approaches. Die Recommended Approaches sind Hinweise und Tipps zur genauen Umsetzung der Gegenmaßnahmen in allen Phasen des SDLCs. In Abbildung 4 sind die Recommended Approaches zur besseren Übersicht nur einmal für die Architekturphase dargestellt.

Die *TRA Engine* ist durch Testautomatisierungsmodule erweiterbar. Die Testautomatisierungsmodule dienen der automatischen oder teilautomatischen Integration von externen Tools in den Prozess der Software Security Assurance. Die Testautomatisierungsmodule werden schwerpunktmäßig in den Phasen der Implementierung und der Modultests eingesetzt. Es können aber auch Tests zu allen anderen Phasen des SDLC automatisch durchgeführt werden. Auch der Support durch Testautomatisierungsmodule ist in Abbildung 4 zur besseren Übersicht nur für die Implementierungs- und Modultestphase dargestellt.

Die Ergebnisse der ausgeführter Security Maßnahmen werden auf ein einheitliches Format standardisiert und schließlich in einem gemeinsamen Dashboard angezeigt. Software Architekten und Testarchitekten sowie alle Projektmitarbeiter erhalten so Feedback über den aktuellen Status der Security Assurance im Bezug auf das von ihnen gewählte Security Level.

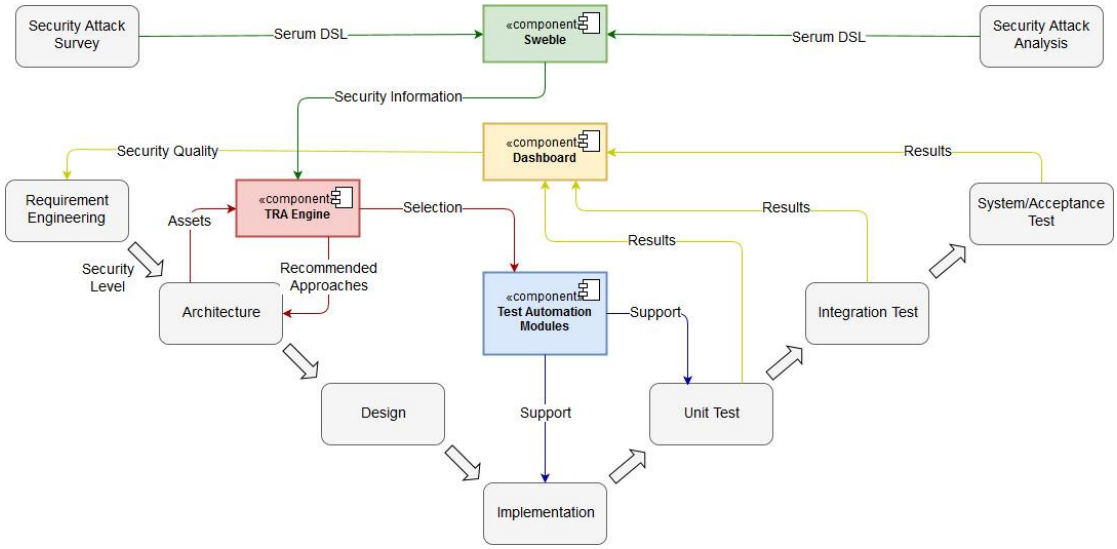

Abbildung 4 *Serum Komponenten im SDLC*

3.2 *Sweble*

Die Verwendung von *Sweble* und die Entwicklung der Submodule ergibt sich in erster Linie durch die Anforderungen Epic #01, Epic #03, Epic #06 und Epic #09.

Sweble ist ein Open Source Projekt der *OSR Group* der Friedrich-Alexander-Universität Erlangen-Nürnberg zur Versionsverwaltung strukturierter Daten. In seiner Funktionsweise ähnelt es anderen Versionskontrollsystemen wie *Git*. *Sweble* konzentriert sich aber auf die Verwaltung strukturierter Daten im Allgemeinen und ist nicht allein auf den Source-Code beschränkt. Daher eignet sich *Sweble* besonders für diese Arbeit. Intern basiert *Sweble* auf baumbasierten Algorithmen.

Sweble erlaubt es, ähnlich wie *Git*, ein Repository anzulegen. Da es sich bei einem *Sweble* Repository im Grunde um eine verteilte Datenbank handelt, wird ein solches Repository im Folgenden Knowledge-Base genannt.

Im Rahmen von *Serum* ist die Aufgabe von *Sweble* die Verwaltung und Verarbeitung der Security Informationen.

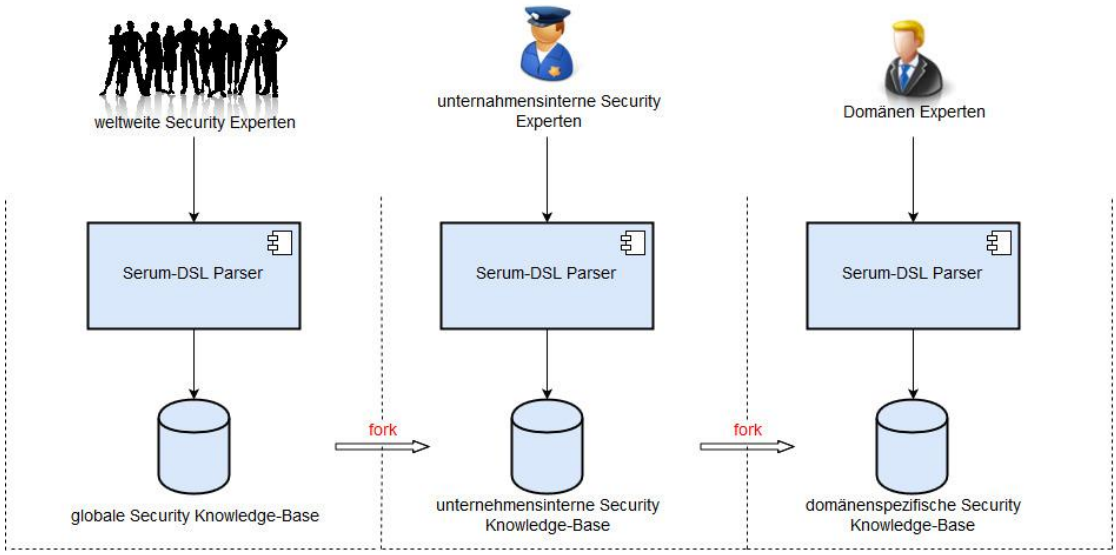

Abbildung 5 *Sweble Verwendung*

Um eine möglichst große Vollständigkeit und Aktualität der Security Information zu gewährleisten, bietet es sich an eine globale, frei zugängliche Security Knowledge-Base zu erstellen. Analog zu bereits bestehenden Sammlungen, wie *OWASP* oder *CWE*, kann diese Security Knowledge-Base von Security Experten weltweit manipuliert und aktualisiert werden (siehe Abbildung 5 links). Um die Integrität der Knowledge-Base unternehmensweit zu wahren, kann man einen Fork der globalen Knowledge-Base erstellen. Wie bei bekannten Versionskontrollsystemen sind so lokal sichtbare Manipulationen und die Aktualität durch Merges oder Rebases möglich (Abbildung 5 Mitte). Durch wei-

tere unternehmensinterne Forks kann die Knowledge-Base mit domänenspezifischen Security Informationen angereichert werden (Abbildung 5 rechts). (Dohrn & Riehle, 2011)

3.3 *Serum*-DSL

Sweble alleine arbeitet derzeit mit allgemein strukturierten Daten.

Zur Eingabe, Spezifikation und Manipulation von Security Daten ist eine domänenspezifische Sprache und ein entsprechendes Parser Plugin, der *Serum-DSL Parser*, entwickelt worden. Im Rahmen der Arbeit macht diese Sprache es den Security Experten möglich mithilfe bekannter Begriffe, Threats, Countermeasures und Assets zu beschreiben und miteinander zu verknüpfen. Trotzdem ist eine einfache Lesbarkeit gewährleistet, die es auch Laien erlaubt die zugrunde liegenden Sachverhalte zu verstehen. Die Struktur der Sprache orientiert sich am objektorientierten Konzept von Klassen und Vererbungen.

Die Semantik der Sprache basiert auf einem klaren Threat-Model (siehe Abbildung 6). Grundlegendes zu den Begriffen ist in Kapitel 1.3.2 zu finden.

Den Mittelpunkt der Domänenwelt bildet das Attack Pattern (Angriffsmuster). Der Begriff wurde anstatt des Threats eingeführt, um ein eindeutiges Verständnis zu garantieren. Im Gegensatz zu einem Threat, welcher sich sowohl auf die grundsätzliche Bedrohung als auch auf die konkrete Gefährdung beziehen kann, gibt es bei der Interpretation eines Attack Pattern keine begriffliche Unsicherheit.

Verpflichtende Attribute eines Attack Patterns sind die Auftrittswahrscheinlichkeit (Likelihood) und die operationale Auswirkung (operational Impact). Die Auftrittswahrscheinlichkeit (Likelihood) entspricht der Wahrscheinlichkeit, dass ein beobachteter Angriff diesem Angriffsmuster folgt. Die operationale Auswirkung (operational Impact) eines Attack Patterns sind die, bei Erfolg eines Angriffs diesen Typs, verletzten Security Eigenschaften (genaueres dazu in Abschnitt 1.3.1). Aufgrund der Domänenstruktur bieten sich Vererbungshierarchien besonders gut zur Veranschaulichung mancher Assoziationen zwischen den verschiedenen Attack Pattern an.

Die Abbildung 7 zeigt die Vererbungshierarchie zwischen dem Attack Pattern *Session Hijacking* und *Session Fixation*.

Session Hijacking: Sessions werden in TCP Verbindungen eines Clients zu einem Server verwendet. Bei jeder Anfrage des Clients schickt dieser eine eindeutige Session ID mit, um sich gegenüber dem Server zu authentifizieren. Ziel einer *Session Hijacking-Attacke* ist die Übernahme einer Session und den damit verbundenen Rechten durch einen unautorisierten, zweiten Client. (The OWASP Foundation, 2013)

Session Fixation: Bei einem *Session Fixation-Angriff* handelt es sich um einen speziellen *Session Hijacking-Angriff*. Dem angreifenden Client muss es gelingen die Session ID des angegriffenen Clients vor dessen Kommunikation mit dem Server, auf einen, ihm bekannten Wert festzulegen. Gelingt dies, kann der Angreifer sich gegenüber dem Server mithilfe der bekannten Session ID als der autorisierte, angegriffene Client ausgeben. (MITRE, 2017)

Außer möglicher Vererbungsbeziehungen können einem Attack Pattern beliebig viele Countermeasures zugeordnet werden. Jede Countermeasure ist mindestens einer Phase des Softwareentwicklungslebenszyklus zuzuordnen, in der diese in die Software eingearbeitet werden muss. Außerdem soll jede Countermeasure einen Aufwand (Effort Attribut) haben, der angibt wie viel zusätzlicher Aufwand durch den Einsatz dieser Gegenmaßnahme entsteht.

Das Wissen um die Durchführung, Vor- und Nachbereitung einer Gegenmaßnahme soll in assoziierten Recommended Approaches festgehalten werden.

Weiterhin kann eine Attacke eines bestimmten Attack Patterns nur dann ausgeführt werden, wenn die entsprechend richtige Kombination an Assets (AssetCombination), Komponenten und Funktionalitäten in der Zielsoftware vorliegt. Ein *Session Hijacking-Angriff* kann beispielsweise nur bei sessionbasierter Authentifizierung gegenüber einem Service durchgeführt werden. Das heißt für den *Session Hijacking-Angriff* muss sowohl ein Service als auch eine sessionbasierte Authentifizierung vorliegen. Theoretisch soll es möglich sein, einem Attack Pattern beliebig viele, aber mindestens eine Asset Kombination (AssetCombination) zuzuordnen. Dabei kann das gleiche Asset in beliebig vielen dieser Asset Kombinationen (AssetCombinations) nötig sein. Assets innerhalb einer Asset Kombination (AssetCombination) sollen auch komplexer kombiniert werden können, um Zusammenhänge realistisch darzustellen, Containment, Extension, Connection, Implementation usw.

Abbildung 6 zeigt das detaillierte Design der domänenspezifischen Sprache. Eingebettet in *Sweble,* soll diese die Verwaltung der Security Informationen möglich machen.

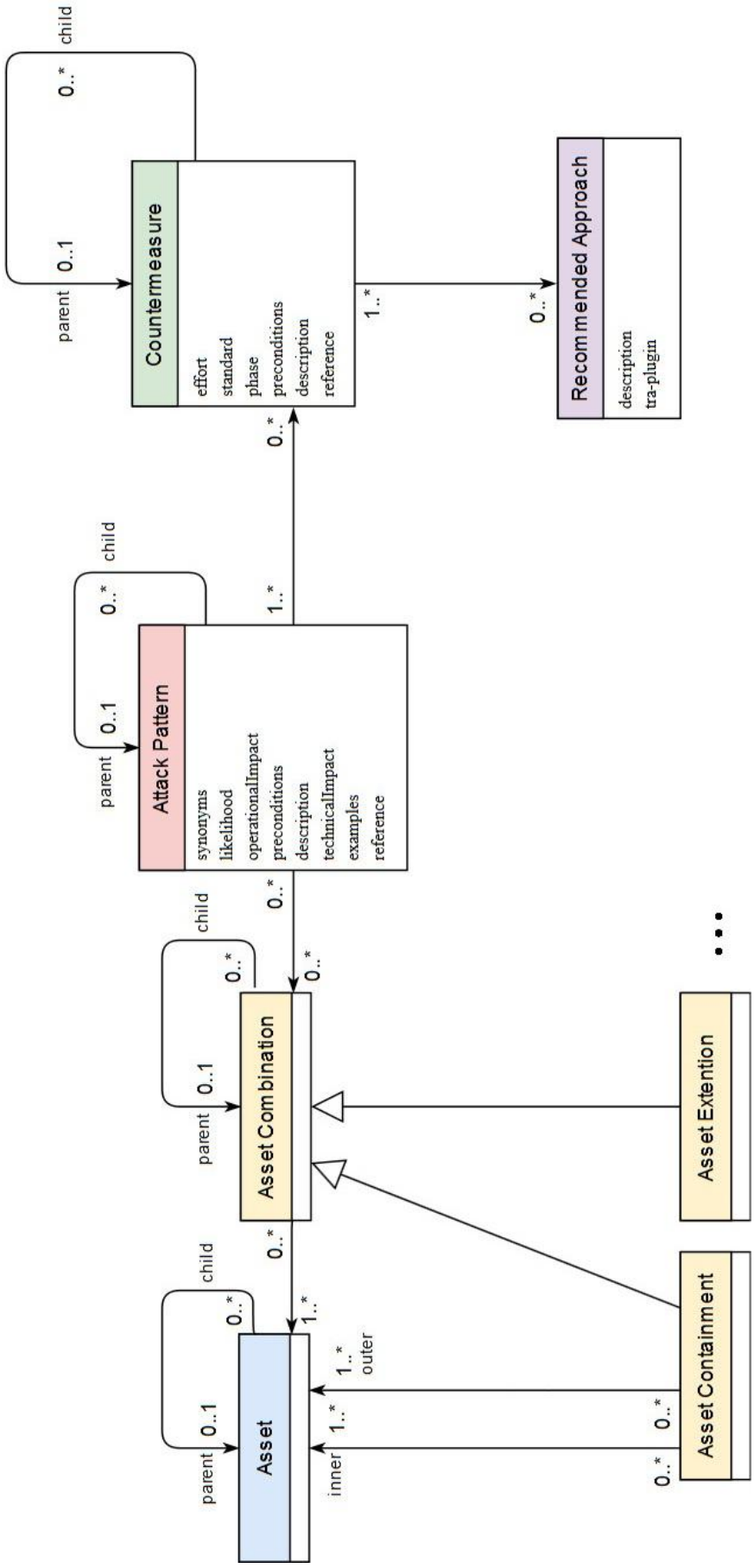

Abbildung 6 *Detailliertes Domänenmodell*

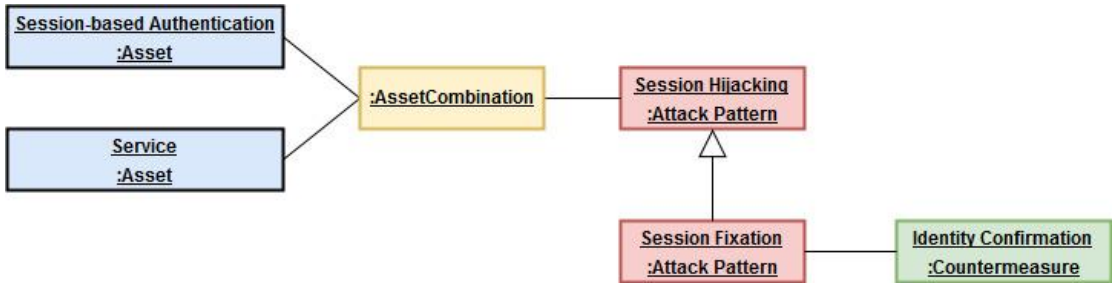

Abbildung 7 *Objektdiagramm Domänenmodell*

3.4 Architektur der TRA Engine

Die zweite Komponente von *Serum* ist die *TRA* (Threat- and Risk Analysis) *Engine*. Diese richtet sich primär an Software Architekten und deckt große Teile der Anforderungen Epic #02, Epic #04, Epic #05 und Epic #07 ab. Ziel der TRA Engine ist die automatisierte TRA und der anwendungs- und bedrohungsspezifische Zugriff auf Attack Pattern, Countermeasures und Recommended Approaches. Außerdem hat die *TRA Engine* Zugriff auf eine *Sweble* Security Knowledge-Base. Dieser Knowledge-Base entnimmt die *TRA Engine* die nötigen Security Informationen, die zuvor durch die DSL formalisiert wurden.

Abbildung 8 zeigt den Ablauf der TRA zur Auswahl und Anzeige von Attack Pattern.

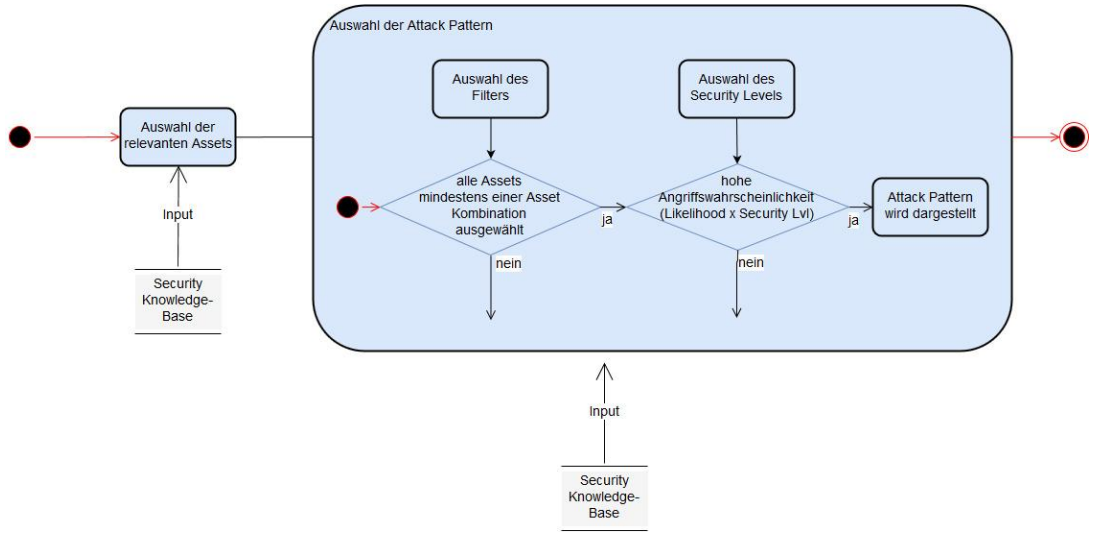

Abbildung 8 *Ablauf TRA bis Attack Pattern*

Zum Zeitpunkt der TRA ist noch kein Wissen über die tatsächliche Implementierung der Zielsoftware bekannt. Ausgangspunkt einer üblichen TRA ist die Softwarespezifikation und -architektur. Die Beschreibung der Softwarearchitektur wird im Rahmen dieser Arbeit durch die Auswahl der Assets vorgenommen. Es können natürlich nur jene Assets ausgewählt werden, die zuvor in *Sweble* definiert wurden. Idealerweise hat der Software Architekt die Mög-

lichkeit die Zielsoftware vollständig durch die Auswahl an Assets zu beschreiben.

Mithilfe der gewählten Assets und der in der Security Knowledge-Base spezifizierten Assoziationen zwischen Assets, Attack Pattern und Countermeasures, kann dann die TRA durchgeführt werden. Das Resultat ist eine Darstellung der Attack Pattern, die für einen Angriff auf die Zielsoftware infrage kommen, und der Countermeasures, die empfohlen werden, um die Wahrscheinlichkeit des Erfolgs eines solchen Angriffs zu minimieren.

Durch die Variation der Filter zur Selektion der bedrohlichen Attack Pattern und der empfohlenen Countermeasures kann die TRA weiter beeinflusst werden (siehe Abschnitt 3.4.1).

Wichtiger Bestandteil der TRA ist außerdem die Auswahl eines Security Levels. Dieser gibt die Wahrscheinlichkeit an, mit der ein Angriff, egal welchen Typs, auf die Zielsoftware erfolgt. Zusammen mit der Wahrscheinlichkeit (Likelihood) und dem Aufwand (Effort) können dann Attack Pattern und Countermeasures ausgewählt werden, die für die Gewährleistung einer zufriedenstellend sicheren Software nötig sind.

Die Auswahl der Countermeasures funktioniert analog. Ausschlaggebend für die Auswahl einer Countermeasures ist der Aufwand (Effort), welcher der Wahrscheinlichkeit (Likelihood) eines Attack Patterns entspricht.

Am Ende einer TRA stehen Recommended Approaches, also Vorschläge, Tipps, Integrations- und Installationsanweisungen zur Anwendung einer Gegenmaßnahme.

3.4.1 Auswahlstrategien

Grundsätzlich sollen bei der Auswahl der relevanten Attack Pattern unterschiedliche Assoziationsstrategien zum Einsatz kommen können. Diese beziehen sich meistens auf die Vererbungshierarchien der einzelnen Security Entitäten. So können beispielsweise Kinder eines Attack Patterns mit anderen Assets als deren Väter assoziiert werden. Um eine höhere Flexibilität zu gewährleisten, wird im Rahmen dieser Arbeit keine Auswahlstrategie festgelegt, obwohl die Semantik dieser Beziehungen zur Vollständigkeit eines klaren Domänenmodells gehört.

So soll auch in Bezug auf das Verständnis von Vererbungshierarchien im Domänenmodell die Erweiterbarkeit von *Serum* gewährleistet werden.

3.4.1.1 Flache Auswahlstrategie

Bei der flachen Auswahlstrategie werden nach Auswahl eines Assets jene Attack Pattern dargestellt, die eine direkte Verbindung zu dem ausgewählten Assets haben. Das heißt, dass diese Attack Pattern mindestens eine Asset Kombination haben, die durch die ausgewählten Assets komplett abgedeckt ist. Kinder dieser Attack Pattern werden auch automatisch angezeigt.

Abbildung 9 zeigt beispielhaft assoziierte Assets und Attack Pattern, anhand derer die Auswahlstrategien leicht nachvollziehbar sind. Die nachfolgende Tabelle zeigt die assoziierten Attack Pattern für eine jeweils unterschiedliche Auswahl an Assets nach Anwendung der flachen Auswahlstrategie.

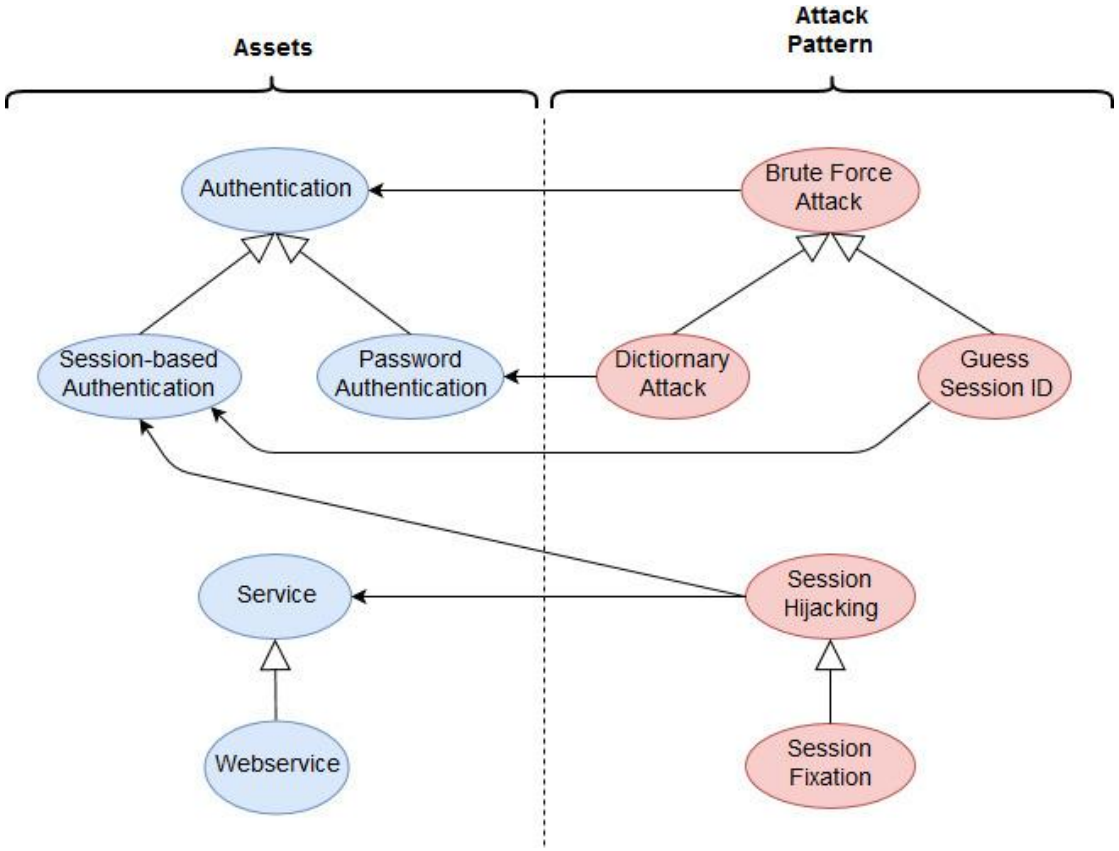

Abbildung 9 *Beispiel Domänenmodell*

Tabelle 1 *Flache Auswahlstrategie Attack Pattern Auswahl*

ID	Ausgewählte Assets	Assoziierte Attack Pattern
1	Authentication Service	Brute Force Attack Dictionary Attack Guess Session ID
2	PW Authentication Service	Dictionary Attack
3	SB Authentication Service	Guess Session ID Session Hijacking Session Fixation
4	Webservice Authentication	Brute Force Attack Dictionary Attack Guess Session ID
5	Service Authentication PW Authentication	Brute Force Attack Dictionary Attack Guess Session ID
6	Service Authentication SB Authentication	Brute Force Attack Dictionary Attack Guess Session ID Session Hijacking Session Fixation

Vorteil der flachen Auswahlstrategie ist die einfache Verständlichkeit. Außerdem eignet sich die flache Auswahlstrategie dann besonders gut, wenn ein Software Architekt in seiner Zielsoftware nur ein Authentifizierungsverfahren verwendet, beispielsweise Session-based Authentication. Alternativ sollte er zum Zeitpunkt der TRA auch alle anderen, verwendeten Authentifizierungsverfahren der Zielsoftware klar benennen können.

Ein Nachteil wird durch die Beispiele 2 und 3 in Tabelle 1, besonders deutlich. Die Auswahl eines Kindes von Authentication, hier Password-Authentication, wird nicht mehr mit dem Attack Pattern *Brute Force Attack* verknüpft. Stattdessen werden nur noch direkte Assoziationen, hier *Dictonary Attack*, angezeigt. Verknüpft man die *Brute Force Attack* jetzt seinerseits mit einer Gegenmaßnahme, würde diese im Falle der Auswahl aus Beispiel 2 nicht mehr vorgeschlagen werden. Die flache Auswahlstrategie funktioniert also nur, wenn Baum-interne Knoten, also Entitäten mit mindestens einem Kind, keine für die TRA wichtigen Assoziationen oder Eigenschaften besitzen.

3.4.1.2 Auswahlstrategie mit hierarischem Ausschluss

Bei der Auswahlstrategie mit hierarischem Ausschluss wird ein gegenseitiger Ausschluss der Attack Pattern einer Ebene angenommen. Ist ein Attack Pattern direkt assoziiert, werden alle Vorgänger dieses Attack Patterns und alle Nachfolger automatisch mit selektiert. Attack Pattern mit dem gleichen Vaterknoten werden allerdings nur dann eingeschlossen, wenn für deren direkte Assoziation keine nicht ausgewählten Assets nötig sind. Logisch ergibt sich dies unter der Annahme, dass auch bauminternen Attack Pattern und Countermeasures Eigenschaften und Assoziationen zugeordnet werden können. Nicht ausgewählte Assets in einer TRA sollen auf jeden Fall ausgeschlossen werden. Außerdem vereinfacht die Auswahlstrategie mit hierarischem Ausschluss die *Serum*-DSL. Kinder eines Attack Patterns erben hier tatsächlich die Asset-Assoziationen ihrer Väter und müssen diese somit nicht explizit definieren.

Die nachfolgende Tabelle zeigt die Auswahlstrategie mit hierarchischem Ausschluss zur Abbildung 9.

Tabelle 2 *Auswahlstrategie mit hierarchischem Ausschluss Attack Pattern Auswahl*

ID	Ausgewählte Assets	Assoziierte Attack Pattern
1	Authentication Service	Brute Force Attack Dictionary Attack Guess Session ID
2	PW Authentication Service	Brute Force Attack Dictionary Attack
3	SB Authentication Service	Brute Force Attack Guess Session ID Session Hijacking Session Fixation
4	Webservice Authentication	Brute Force Attack Dictionary Attack Guess Session ID
5	Service Authentication PW Authentication	Brute Force Attack Dictionary Attack Guess Session ID
6	Service Authentication SB Authentication	Brute Force Attack Dictionary Attack Guess Session ID Session Hijacking Session Fixation

3.4.2 Security Level

Mit Bezug auf Epic #05 und Abschnitt 2.1 soll ein Security Level den erwünschten Grad an Sicherheit der Zielsoftware definieren. Abhängig von der Bereitschaft Geld und Zeit in die Sicherheit der Software zu investieren, wählt der Architekt während der TRA ein Security Level. Dabei sollte natürlich auch die Angriffswahrscheinlichkeit der Software bedacht werden.

Zusammen mit der Auswahlstrategie und den ausgewählten Assets bestimmt das Security Level die Auswahl der relevanten Attack Pattern und Countermeasures. Ist ein Security Level gewählt, wird die Auswahl der Attack Pattern und Countermeasures eingeschränkt. Bei einem hohen Security Level werden alle systemrelevanten Attack Pattern und Countermeasures angezeigt und sollen verpflichtend durchgeführt werden. Bei einem niedrigen Security

Level werden nur jene Attack Pattern mit hoher Auftrittswahrscheinlichkeit (hohem Wert des Likelihood Attributs) angezeigt. Die Auswahl der Countermeasures misst sich an der Höhe des Aufwands (Effort Attribut).

3.5 Testautomatisierungsmodule

Am Ende einer automatisierten TRA stehen die Recommended Approaches, diese richten sich primär an die Anforderung Epic #07. Die Recommended Approaches bieten außer der Möglichkeit Hinweise, Tipps und Anleitungen zu geben, auch die Angabe einer *tra-plugin Klasse* (siehe Abbildung 6 rechts unten). Die *tra-plugin Klasse* ist ein Verweis auf ein Testautomatisierungsmodul. Testautomatisierungsmodule werden als Plugins für die *TRA Engine* implementiert. Ziel dieser Plugins ist die automatische oder teilautomatische Ausführung eines externen Programms zur Durchführung oder Analyse einer Countermeasure oder zum Test auf Software Vulnerabilities. Zusammen mit den sonstigen Hinweisen aus den Recommended Approches soll so die technische Komplexität von Toolunterstützung reduziert werden. Primärer Stakeholder der Recommended Approaches und der Testautomatisierungsmodule ist der Testarchitekt. Besonders geeignet für die Testautomatisierung sind zum Beispiel statische Codeanalysen oder Systemtest für Webapplikationen. Auf Anfrage des Software Architekten, bzw. des *Serum* Nutzers, der die TRA durchführt, sollen die angegebenen Testautomatisierungsmodule zur Ausführung kommen. Beispielhaft wurden im Rahmen dieser Arbeit ein Plugin für *BDD-Security* (Continuum Security, 2017) und für Xanitizer (Rigs IT, 2017) entwickelt. Details der Komponente sind in Abschnitt 4.3 zu finden.

3.6 Dashboard

Das *Serum-Dashboard* ist eine eigenständige Komponente *Serums* zur zentralisierten Darstellung der Security Testergebnisse. Es ergibt sich aus den Anforderungen Epic #08 und Epic #09.

Aufgrund der Vielzahl unterschiedlicher Werkzeuge und Methoden, die bei der Security Assurance zum Einsatz kommen können, liegen die Security Testergebnisse verstreut und nicht einheitlich vor. Um Software Architekten und Testarchitekten trotzdem einen Überblick über die Gesamtheit der Testergebnisse bieten zu können, wurde das *Serum-Dashboard* und die darunter liegende Serverkomponente entwickelt.

Um vom *Serum-Dashboard* dargestellt werden zu können, müssen die Security Testergebnisse in einem ersten Schritt manuell oder von den Testautomatisierungsmodulen in ein einheitliches Format (siehe Abbildung 20) gebracht werden. Dieses wird von der zentralen Serverkomponente des *Serum-Dashboards*

eingelesen, mit zusätzlichen Security Informationen aus der TRA annotiert und schließlich persistiert.

Um für verteilte Stakeholder gut nutzbar zu sein, wurde das *Serum-Dashboard* Frontend selber als Webapplikation entwickelt. Dieses soll die Testergebnisse, abhängig von Projekt, eingesetztem Werkzeug, Security Level und Zeitpunkt des Testdurchlaufs darstellen können. Die Daten für das Dashboard werden bedarfsgerecht von der zentralen Serverkomponente geladen. Der Ablauf ist in Abbildung 23 grafisch dargestellt. Bei der Entwicklung des *Serum-Dashboards* und der Serverkomponente wurde besonders auf eine flexible Implementierung und Architektur geachtet.

Die Schritte und die Zusammenarbeit der Testautomatisierungsmodule mit dem Dashboard sowie eine gesamte Übersicht über die Komponenten von *Serum* sind in Abbildung 10 zur besseren Übersicht noch einmal dargestellt.

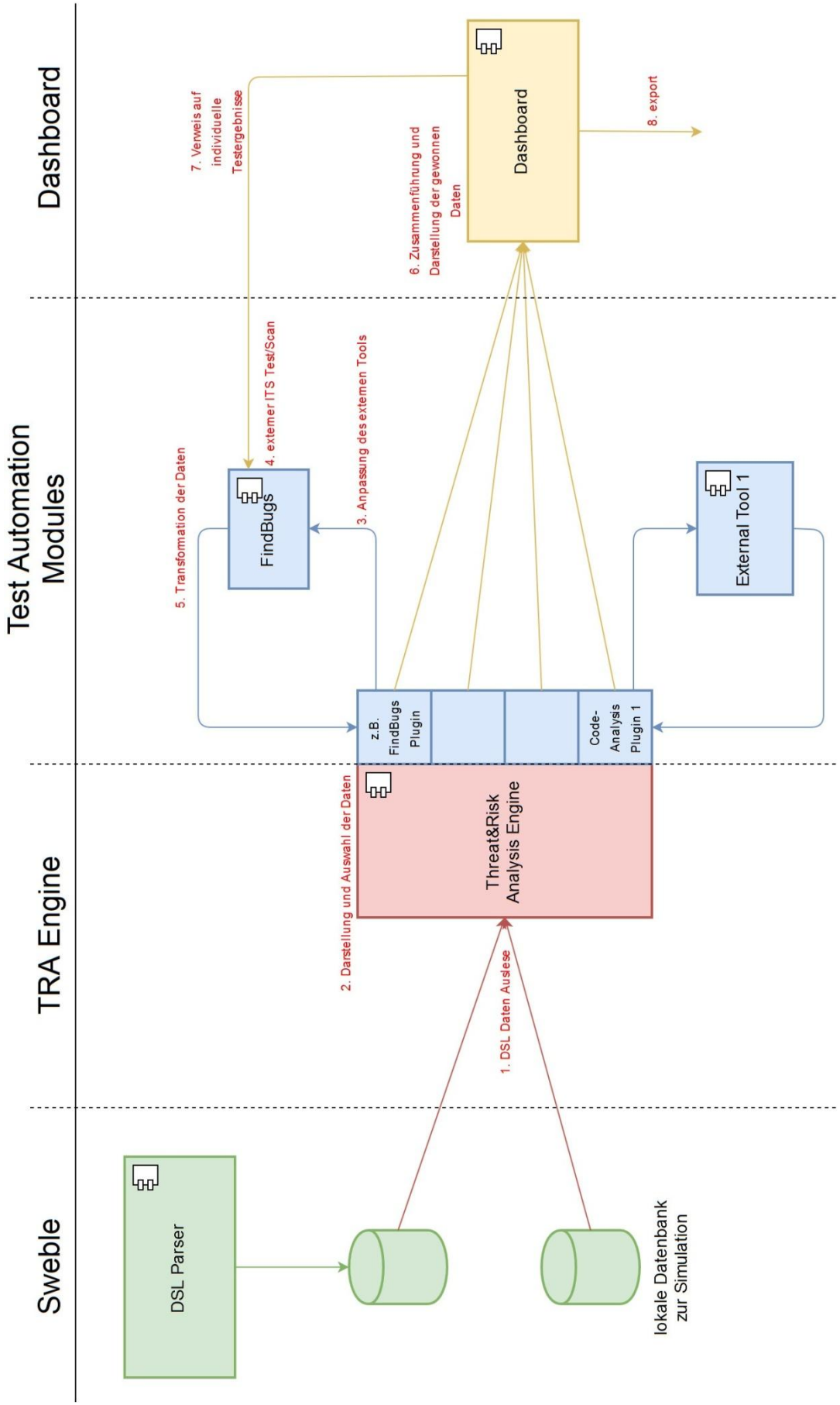

Abbildung 10 *Komponentendiagramm Serum gesamt*

4 Implementierung

Im Folgenden wird die detaillierte Implementierung der bereits vorgestellten und im Rahmen dieser Arbeit entwickelten Komponenten beschrieben.

4.1 *Serum*-DSL Parser

4.1.1 Aufbau

Zur Entwicklung des Parsers selber wurde *Xtext* benutzt. *Xtext* ist ein Open-Source-Framework für die Entwicklung von domänenspezifischen Sprachen. Es ist Teil des *Eclipse-Modeling-Frameworks* und arbeitet als solches mit dem Resourcenkonzept von *Eclipse*. *Xtext* generiert einen LL-Parser.

Abbildung 11 zeigt einen Ausschnitt der *Xtext Grammar* Beschreibung.

```
Model:
    ((pattern+=Pattern)|(assets+=Asset)|(countermeasures+=Countermeasure))*;

Pattern:
    'attack:' name=ID ('extends' superType=[Pattern|ID])?'{'
        ('synonyms:{' (synonyms+=ID)'}')?
        ('likelihood:' likelihood=Probability)
        ('operationalImpact:{' operationalImpact+=ImpactCategory+'}')?
        ('preconditions:{' (preconditions+=STRING)+'}')?

        (('assets:{'  (assetCombinations+=AssetCombination)+ '}')|
        ('assets:{'  (assetInCombinations+=AssetInCombination)+ '}')|
        ('assets:{'  (assetConnectCombinations+=AssetConnectCombination)+ '}')|
        ('assets:{'  (assetApplyCombinations+=AssetApplyCombination)+ '}')|
        ('assets:{'  (assetExtendsCombinations+=AssetExtendsCombination)+ '}')|
        ('assets:{'  (assetImplCombinations+=AssetImplCombination)+ '}')|
        ('assets:{'  (assetReadsCombinations+=AssetReadsCombination)+ '}')|
        ('assets:{'  (assetWritesCombinations+=AssetWritesCombination)+ '}')|
        ('assets:{'  (assetUsesCombinations+=AssetUsesCombination)+ '}')|
        ('assets:{'  (assetCallsCombinations+=AssetCallsCombination)+ '}'))?
        ('countermeasures:{' (countermeasures+=[Countermeasure|ID])+ '}')?

        ('description:' description=STRING)?
        ('technicalImpact:' technicalImpact=STRING)?
        ('examples:' examples=STRING)?
        ('reference:' reference=STRING)?
    '}'
;

AssetCombination:
    '('
        (assets+=[Asset|ID])+
    ')'
;
```

Abbildung 11 *Serum-DSL Definition Xtext*

Ganz oben in Abbildung 11 ist die Definition des Models zu sehen. Es bildet die Wurzelentität eines jeden *Serum*-DSL Dokuments. Das Model enthält beliebig viele Assets, Attack Pattern und Countermeasures in beliebiger Ordnung und kann selber nicht explizit definiert werden. Auch hat das Model keine Bedeutung im Security Domänenmodell.

Abbildung 11 zeigt außerdem die Definition von Attack Pattern (Pattern) und Asset Kombinationen (AssetCombination). Jede Art der Asset Kombination wurde als eigene Entität, ähnlich der normalen AssetCombination, definiert (siehe Attack-Asset-Assoziation: *AssetInCombination, AssetApplyCombination* usw.). Dies ist entscheidend für die Assoziation zwischen Asset und Attack Pattern (siehe dazu auch 3.3 *Serum*-DSL). Die separaten Entitäten sorgen für eine gute Erweiterbarkeit der Sprache. Es können einfach neue Asset Kombinationen hinzugefügt, als auch bereits vorhandene abgeändert werden. Außerdem könnte man in einer Asset Kombination zusätzliche Eigenschaften definieren. Die Lesbarkeit der DSL selber wird nicht eingeschränkt. Zuletzt unterstützt diese Art der AssetCombination-Definition die Weiterverarbeitung der Daten in der *TRA-Engine*.

```
asset: SessionBasedAuthentication extends Authentication

attack: SessionHijacking {
    likelihood: HIGH
    operationalImpact:{ CONFIDENTIALITY}
    assets:{ (Service SessionBasedAuthentication)  }
    description: "The Session Hijacking attack consists of the exploitation of the web session
                  control mechanism, which is normally managed for a session token. ... "
    technicalImpact: "Gaining unauthorized access to the Web Server"
    examples: "The attacker can compromise the session token by using malicious code or programs
               running at the client-side. ..."
    reference: "https://www.owasp.org/index.php/Session_hijacking_attack"
}

countermeasure: CompareSessionAndCookieValue {
    effort: LOW
    phase: DESIGN
}
```

Abbildung 12 *Serum-DSL Beispiel*

Abbildung 12 zeigt ein Beispiel einer *Serum*-DSL Eingabemenge. Eigene Keywords der Sprache sind in lila angezeigt.

4.1.2 Einpassung in *Sweble*

Sweble arbeitet mit einer *REST-API* und zahlreichen Microservices. Um die Verwaltung von Security Informationen in *Sweble* zu ermöglichen, müssen die entsprechenden Daten in Form eines Document Object Models vorliegen und werden so an einen Microservice zur Persistierung und Verwaltung weitergegeben.

Um der Schnittstelle des *Sweble* Microservices zu entsprechen, wurde der von *Xtext* generierte Parser in einem eigens implementierten Parser eingebettet. Das entsprechende Klassendiagramm und die daraus resultierende Schnittstel-

le des Parser Microservices (*ParserImpl*) sind in Abbildung 13 dargestellt. Weitere Diagramme zur Interaktion und zum Arbeitsablauf der Komponenten sind im Anhang verfügbar.

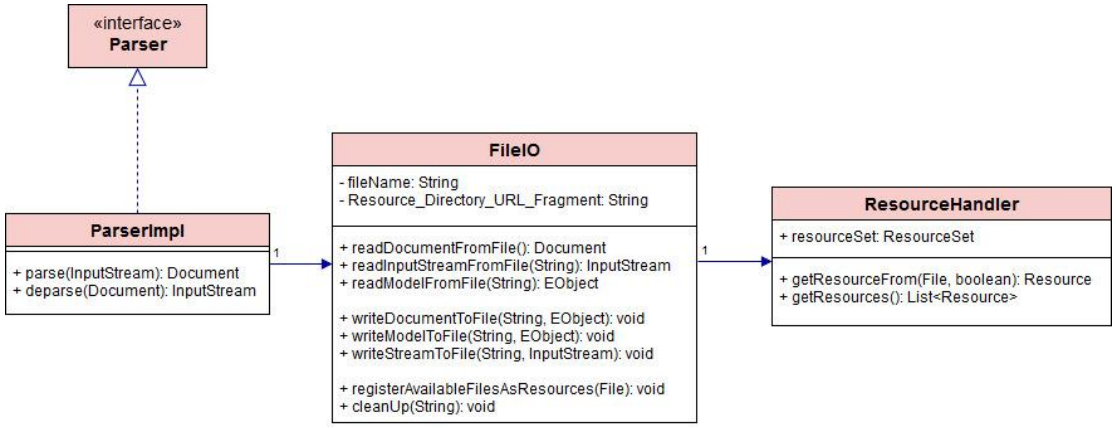

Abbildung 13 *Klassendiagramm Serum-DSL Parser*

4.1.3 Semantische vs. syntaktische Validierung

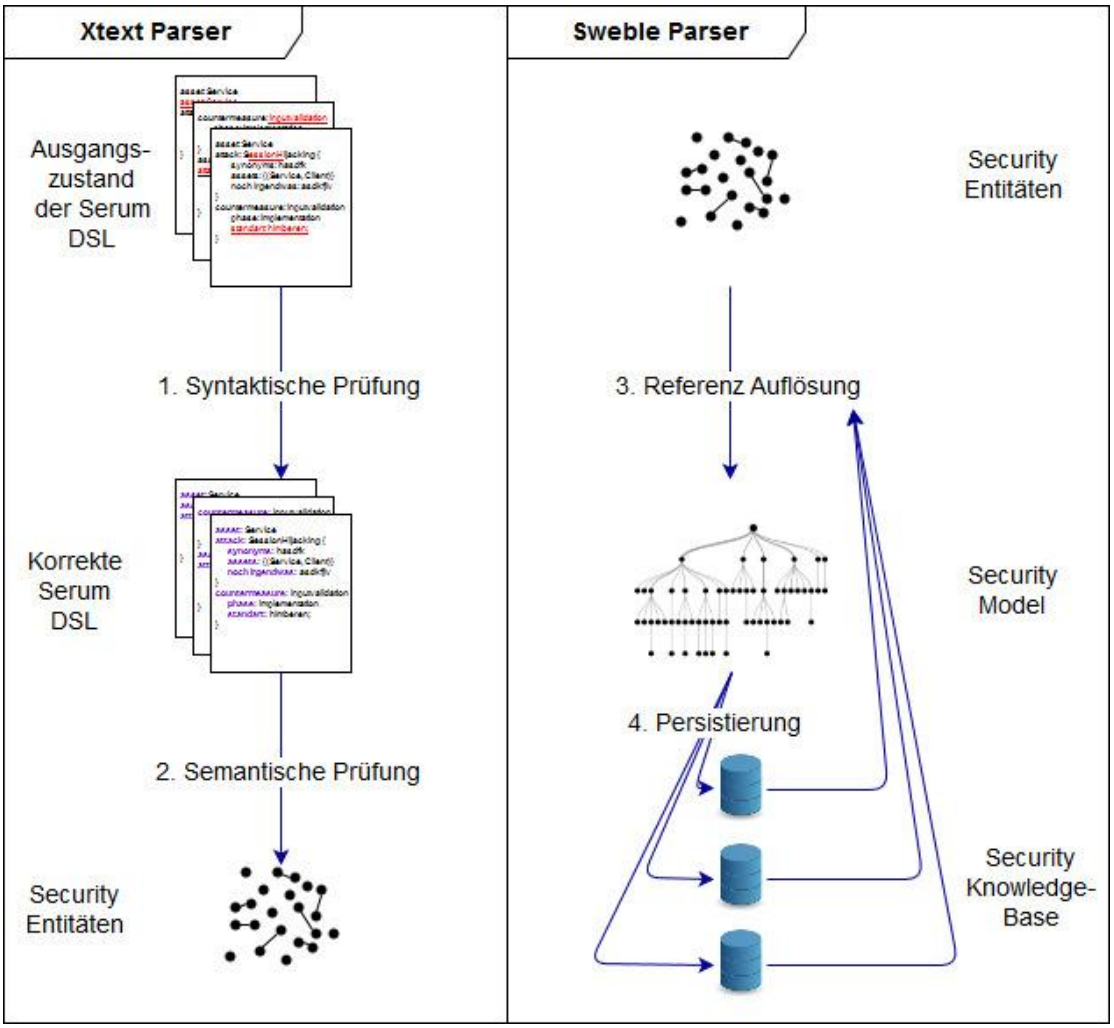

Abbildung 14 *Validierung der Serum-DSL*

Abbildung 14 zeigt die Schritte der beiden Parser. Ausgangspunkt ist die eine ungeprüfte *Serum*-DSL Eingabemenge. Die syntaktische Validierung der *Serum*-DSL findet ausschließlich im *Xtext* Parser statt. Diese ist durch die *Xtend* Programmiersprache einfach zu verändern oder zu ergänzen. Der syntaktische *Serum*-DSL Code wird anschließend in ein Modell übersetzt. Die semantische Analyse, also die Prüfung der logischen Korrektheit der *Serum*-DSL, kann vor allem aufgrund der Entitätsreferenzen nur teilweise durch den generierten Parser abgedeckt werden. Ergebnis von Schritt 2 aus Abbildung 14 sind die Security Entitäten aus Abbildung 6. Bis hierher sind nur Referenzen innerhalb der gleichen *Serum*-DSL Eingabemenge aufgelöst.

Zur Auflösung der Cross-Referenzen, also jener Referenzen, die auf Entitäten außerhalb der gleichen *Serum*-DSL Eingabemenge deuten, werden die benötigten Daten aus der Security Knowledge-Base geladen. Das Ergebnis ist ein vollständig referenziertes Modell. Anschließend werden die Entitäten, nach Typ geordnet, persistiert. Die Vor- und Nachteilen der getrennten Persistierung werden im Abschnitt 5.1.3 Validierung der DSL erörtert.

4.2 Implementierung der TRA Engine

Die *TRA Engine* wurde als native Java Applikation entwickelt. Für eine gute Wartbarkeit, Lesbarkeit und Erweiterbarkeit und um eine einfache Portierung als Webapplikation zu ermöglichen, orientiert sie sich am *Model-View-Controller Architectural Pattern*.

4.2.1 Anbindung an *Sweble*

Um an die nötigen Security Informationen für eine TRA zu gelangen, ist die TRA Engine auf eine Instanz einer *Sweble* Knowledge-Base angewiesen. Da das *Sweble* Projekt zum Zeitpunkt dieser Arbeit noch nicht fertiggestellt ist, wurde die Anbindung der TRA-Engine an *Sweble* durch eine lokale OO-Datenbank simuliert. Die Schnittstelle zur Abfrage der Daten aus der Knowledge-Base ist klar definiert, so dass für den Moment der *Sweble* Fertigstellung eine einfache Umstellung möglich ist. Die logische Abfrage der Daten aus der Knowledge-Base findet in *Sweble*, bzw. auf Serverseite der Schnittstelle statt (siehe Abbildung 15). Die Anfragen, im Fall der Simulation in den Klassen *ComplexQueryBuilder* und *BasicQueryBuilder*, sind durch das Persistierungsmodell der Daten geprägt. Die Daten des *Model Packages* werden zur Weiterverarbeitung an die Businesslogik-Schicht weitergereicht.

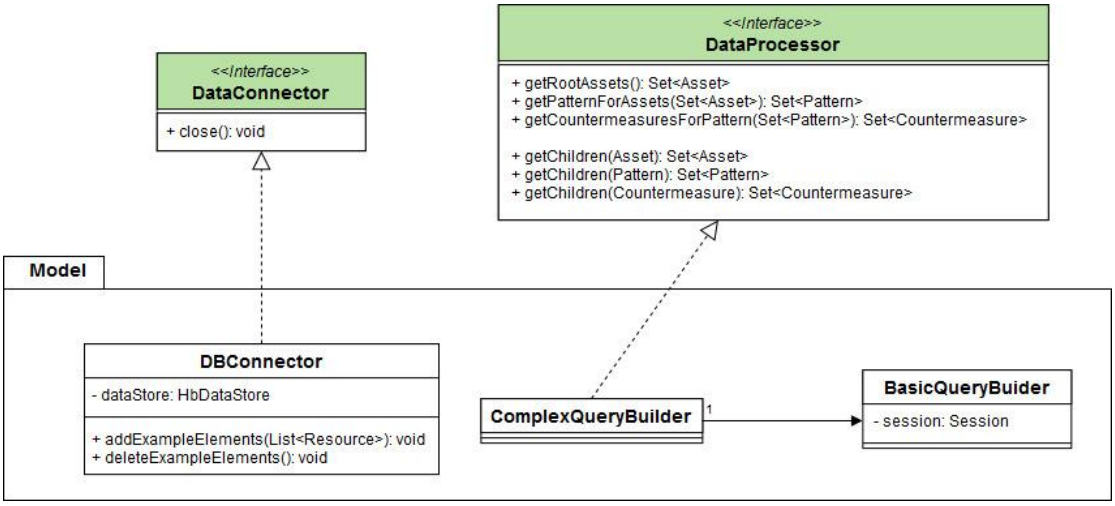

Abbildung 15 *Klassendiagramm TRA Engine Model*

4.2.2 Businesslogik

Die Schnittstelle der Businesslogik ist ähnlich der Schnittstelle des *DataProcessors* aus Abbildung 15. Inhaltlich ist die Businesslogik aber für die Manipulation und Filterung der Daten zuständig, unabhängig von deren Persistierungsmodell. Die Businesslogik arbeitet auf dem Domänenmodell der Daten und findet komplett auf Client Seite statt, also nicht mehr in *Sweble*.

Kernelemente der Businesslogik sind die zwei Strategiemuster *getPattern* und *getCountermeasure*. Dabei handelt es sich jeweils um Strategien zur Auflösung und Abfrage der Assoziationen zwischen Asset und Attack Pattern, bzw. zwischen Attack Pattern und Countermeasure. Sie tragen wesentlich zur TRA bei. Logisch entsprechen sie der Umsetzung des Designs aus Kapitel 3.4.1Auswahlstrategien.

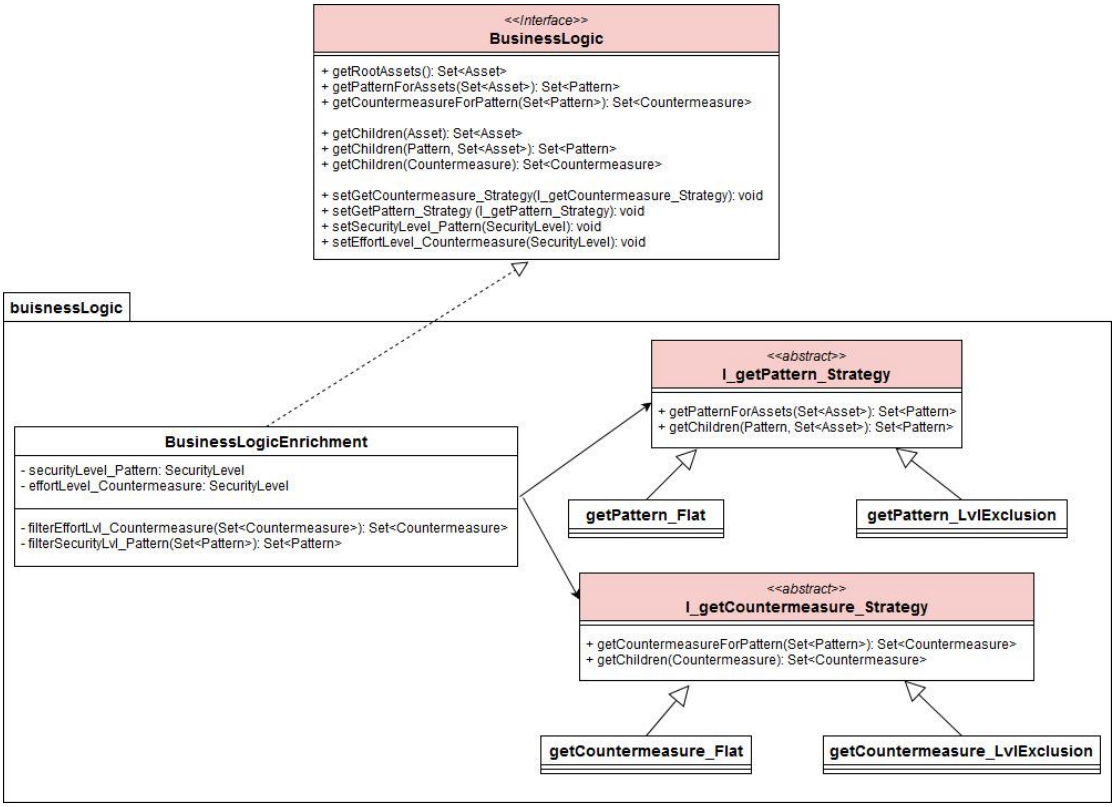

Abbildung 16 *Klassendiagramm TRA Engine Businesslogik*

4.2.3 Controller

Für die TRA stehen dem Architekten baumbasierte Ansichten für alle Assets, Attack Pattern und Countermeasures zur Verfügung. Die Controller und View Schichten der Applikation sind für die Darstellung und Verwaltung der UI Elemente zuständig, insbesondere der Baumansichten. Abbildung 17 zeigt das vereinfachte Klassendiagramm der Controller Schicht. Im Appendix unter Abbildung 24 ist nochmal eine Übersicht über alle Schichten der *TRA Engine* zu sehen.

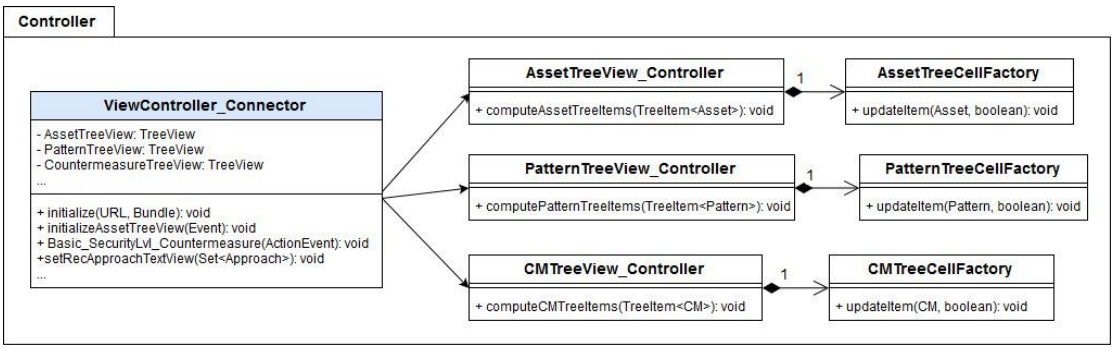

Abbildung 17 *Klassendiagramm TRA Engine Controller*

4.3 Testautomatisierungsmodule

Nachdem der Architekt in der *TRA Engine* Countermeasures ausgewählt hat, werden ihm die Recommended Approaches angezeigt. Diese enthalten Informationen und Hilfestellungen zur Durchführung der Gegenmaßnahmen, welche auch für Software Entwickler, Tester oder Testarchitekten interessant sind. Sind die Recommended Approaches sauber verwaltet und gepflegt, kann das die nachfolgenden Testarbeiten vereinfachen und die technische Komplexität des Testens reduzieren.

Für einige Tools oder Methoden bietet sich außerdem eine automatische oder teilautomatische Durchführung an. Die automatische Durchführung eines Recommended Approaches erfolgt durch ein Testautomatisierungsmodul. Dieses muss als Java-Plugin im *plugins* Ordner der *TRA Engine* liegen und durch das Attribut tra-pugin des Recommended Approaches mit einem eindeutigen Namen spezifiziert werden.

Zur Ausführung implementiert ein Testautomatisierungsmodul das Interface *TestAutomation* (siehe Abbildung 18). Nach Ausführung des Tests können die Ergebnisse vom Testautomatisierungsmodul in Form einer XML Datei exportiert und für die Darstellung in externen Dashboards, wie *Sonarqube*, genutzt werden. Außerdem werden sie in das Datenmodell des *Serum-Dashboard* transformiert und im *Serum-Dashboard* Format exportiert.

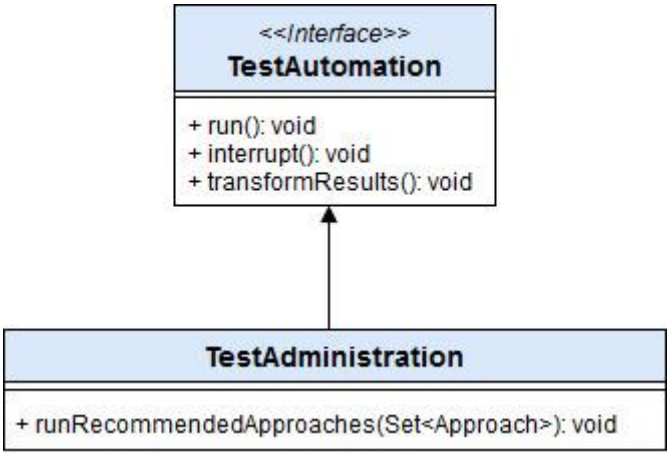

Abbildung 18 *Klassendiagramm Testautomatisierung*

Im Rahmen dieser Arbeit wurde ein Testautomatisierungsmodul für das Werkzeug *BDD-Security* (Continuum Security, 2017) entwickelt. Das Werkzeug dient dem Testen von Webapplikation. Die Ergebnisse von *BDD-Security* liegen nach Ausführung als eine Vielzahl von HTML Dateien vor. Diese werden vom Testautomatisierungsmodul traversiert und die fehlgeschlagenen oder ignorierten Testfälle extrahiert. Die Daten können sowohl im *Serum-Dashboard* Format als auch im *Sonarqube UnitTest* Format exportiert werden.

4.4 Dashboard

Das *Serum-Dashboard* wurde als eigenständige Webapplikation entwickelt. Das Frontend der Applikation ist allein für die Darstellung der Daten verantwortlich und wurde vorwiegend in JavaScript implementiert. Das Backend, auch Model Ebene oder zentrale Serverkomponente des Dashboard genannt, besteht aus einer MySQL-Datenbank und einer Vielzahl an Funktionen zum Laden und Persistieren der Daten in Java. Diese können durch Java-Servlets auch vom Frontend des Dashboard genutzt werden.

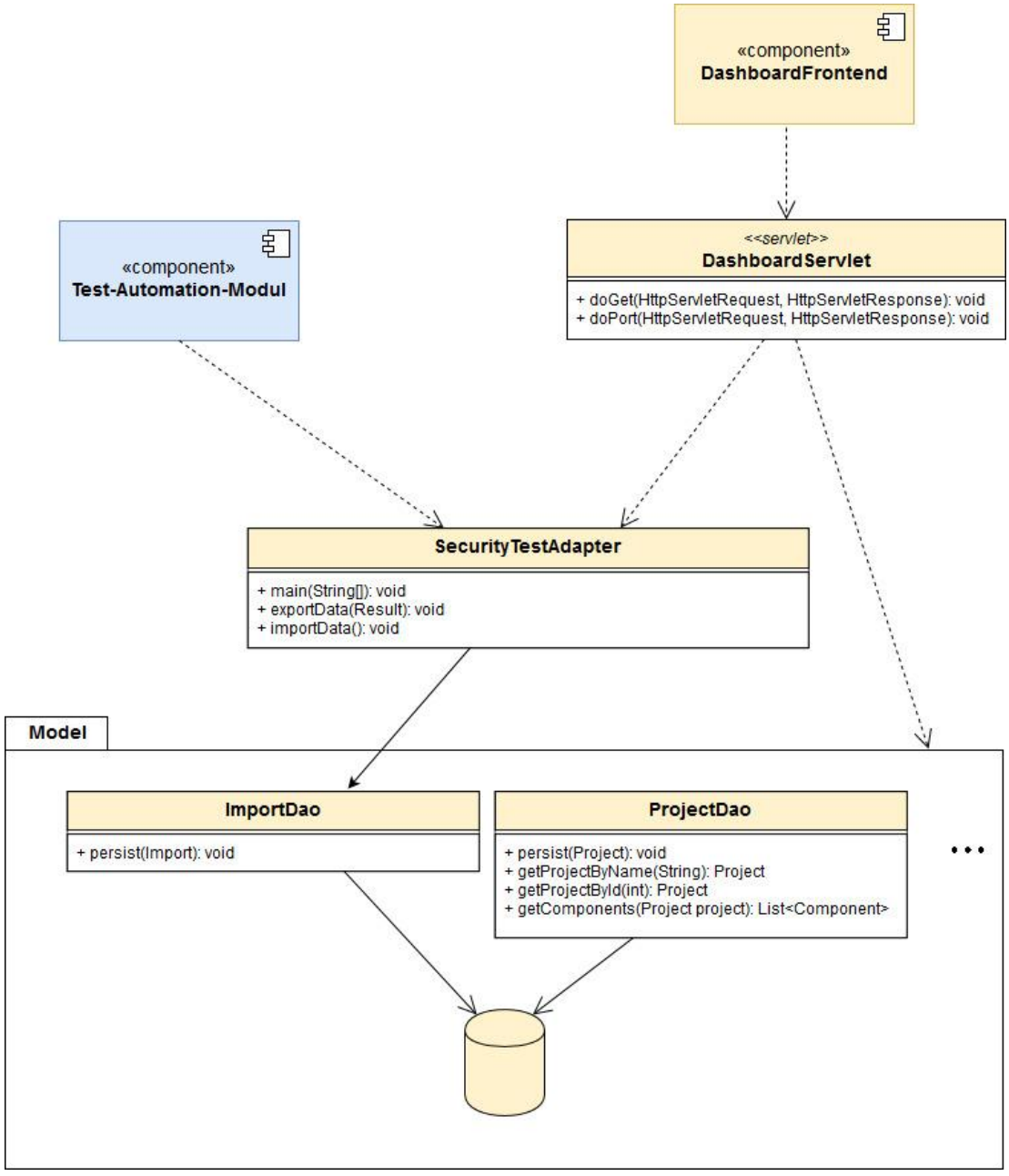

Abbildung 19 *Klassendiagramm Dashboard*

Bei der individuellen Ausführung der Serverkomponente werden alle Dateien eingelesen, die im Ordner *import* liegen und dem *Serum-Dashboard* Format (Abbildung 20) entsprechen. Anschließend werden sie mit Informationen aus der TRA annotiert und persistiert. Das *Serum-Dashboard* Format entspricht bereits den Klassen zur Persistierung in der Datenbank.

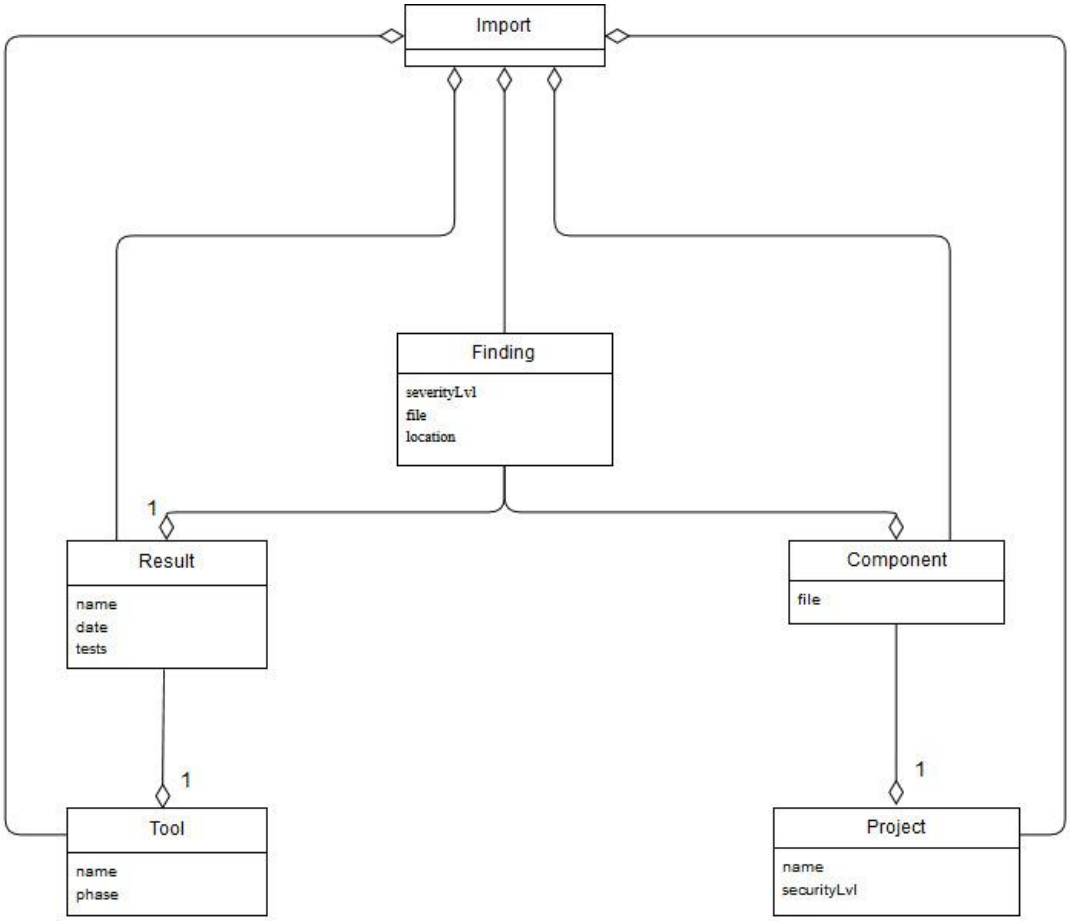

Abbildung 20 *Klassen des Serum-Dashboard Format*

Die persistierten Daten werden bei der Ausführung der Dashboard Webapplikation über die JavaServlets geladen und dargestellt.

5 Evaluierung der Artefakte

5.1 *Serum*-DSL

5.1.1 Verwendung einer DSL

Die *Serum*-DSL dient als Spezifikationsschicht für Security Informationen. Eine DSL ist ein formales Medium und als solches besonders geeignet für die Automatisierung von Prozessen. Im Rahmen dieser Arbeit bildet sie die ideale Grundlage, um ein allgemeines Verständnis der Security Entitäten zu fördern und die formalisierten Daten zur Automatisierung der TRA zu nutzen.

5.1.2 Formalisierung des Domänenmodells

In der Arbeit ist bereits die Abbildung verschiedener realer Domänenbeispiele auf die DSL gelungen. Alle wichtigen Security Entitäten sind durch entsprechende Entitäten der *Serum*-DSL abgedeckt. Ausnahme bildet hier der Security Threat. Dieser wurde explizit in Epic #01 erwähnt. Der Threat wurde aufgrund der begrifflichen Unklarheit durch das Attack Pattern ersetzt. Der Begriff Attack Pattern bietet ein intuitives Verständnis und fügt sich besonders gut in die Automatisierung der TRA ein. Man bedenke, dass sich ein Attack Pattern meistens auf mehrere, in bestimmter Weise verknüpfte Assets bezieht. Das erleichtert die Filterung und Selektion der relevanten Attack Pattern während der TRA. Auch ist es intuitiver und formal verständlicher einem Attack Pattern, statt einem Threat, eine Wahrscheinlichkeit (Likelihood Eigenschaft) zuzuordnen. Dies hat sich durch informelle Interviews mit Security Experten der Siemens AG bestätigt.

Die Verknüpfung zwischen Threat und Countermeasure, beziehungsweise die Benennung von Countermeasures für Threats, ist eindeutiger als die Verknüpfung zwischen Attack Pattern und Countermeasures. Da ein Attack Pattern mehrere Schwachstellen eines Systems ausnutzen kann, kann es auch unterschiedlich viele Countermeasures gegen ein Attack Pattern geben. Gegen ein Threat oder eine Schwachstelle des Systems gibt es hingegen meistens genau eine Gegenmaßnahme. Um sich beispielsweise vor einem *Brute Force-Angriff*

zu schützen, kann man sowohl eine *Password Policy* einführen, als auch einen *Throttling Mechanismus* in die Authentifizierung einbauen. Beide Gegenmaßnahmen richten sich an unterschiedliche, zugrunde liegende Verwundbarkeiten des Systems. Eine *Password Policy* verringert die Wahrscheinlichkeit, dass Passwörter erraten werden. Ein *Throttling Mechanismus* verhindert, dass ein Angreifer beliebig viele Passwörter ausprobieren kann.

Folgearbeiten könnten sich damit beschäftigen beide Begriffe für die Formalisierung der Security Informationen zu nutzen.

In verwandten Arbeiten findet sowohl der Begriff des Threat als auch der Angriff bzw. das Angriffsmuster getrennt Gebrauch. (The OWASP Foundation, 2013) (MITRE, 2017)

Der Begriff und die Formalisierung der Countermeasures haben sich als gut aber nicht ideal herausgestellt. In Bezug auf Kapitel 1.3.2.6 wurde ein Countermeasure explizit als Maßnahme zur Vermeidung von Gefährdungen (Prevention), zum Schutz vor Gefährdungen (Protection) oder zur Minimierung des Schadens eines erfolgreichen Angriffs (Mitigation) definiert. Im Laufe der Entwicklung haben die Bedeutungen Prevention und Mitigation an Bedeutung verloren. Die ursprüngliche Begriffsdefinition, die entsprechende Anforderung Epic #03 oder die Definition der DSL müssten diesbezüglich überarbeitet werden.

5.1.3 Validierung der DSL

Die Validierung der DSL wurde in Kapitel 4.1.3 ausführlich behandelt. Grundlage dieser Art der Validierung ist das Konzept der getrennten Persistierung der Security Entitäten. Das heißt, dass die Security Entitäten nach Typ getrennt unabhängig von der Eingabemenge persistiert werden.

Vorteil der getrennten Persistierung ist ein sauberes Validierungsverfahren, welches leicht erweiterbar ist. Nachteil ist, dass die Persistierung keinen Rückschluss mehr auf die initiale Eingabemenge erlaubt. Das heißt nachträglich kann nur jede, bereits einmal persistierte Entität getrennt geladen und getrennt bearbeitet werden. Dies sollte die Arbeit der Security Experten jedoch kaum behindern.

Um trotzdem zu verhindern, dass der Security Experte jedes Asset oder jedes Pattern einzeln bearbeiten muss, sind Erweiterungen der Ladelogik denkbar. Details dazu sind in Kapitel 6.1 beschrieben.

5.1.4 Test

Für den *Serum*-DSL Parser wurden ausführliche UnitTests geschrieben. Es wurde eine Verzweigungsüberdeckung von über 90% erreicht. Die Testdaten und Tests sind in der Versionsverwaltung enthalten und Teil der Abgabe.

5.2 Evaluierung der TRA Engine

Die *TRA Engine* ermöglicht eine einfache automatisierte TRA. Die Bedienung ist intuitiv und ohne Domänenwissen nachvollziehbar. Eine Auswahl und Filterung nach Security Level ist enthalten. Die *TRA Engine* erfüllt die Anforderungen aus Epic #05.

5.2.1 Zugriff auf Security Informationen

Im Rahmen dieser Arbeit war auch der Zugriff auf das strukturierte Security Wissen eine der zentralen Forderungen (Epic #02 und Epic #04). Dieser ist teilweise durch die *TRA Engine* und teilweise durch *Sweble* und den *Serum*-DSL Parser abgedeckt. Der *Serum*-DSL Parser enthält neben einer *parse* auch eine *deparse (pretty print) Methode*. Diese kann dazu genutzt werden, die Daten aus einer Security Knowledge-Base darzustellen. Die *TRA Engine* zeigt nach Auswahl der Assets alle relevanten Attack Pattern und Countermeasures an. Mit Doppelklick ist eine Detailansicht verfügbar. Ein wahlfreier Zugriff auf Assets, Attack Pattern oder Countermeasures ist in der *TRA Engine* nicht möglich.

5.2.2 Filterung in der TRA

Die Auswahlstrategie und Filterung der Attack Pattern und Countermeasures wurde im Kapitel 3.4.1 ausführlich erklärt. Um die Semantik der TRA so offen und erweiterbar wie möglich zu gestalten, wurde im Rahmen der Arbeit die Auswahlstrategie der Attack Pattern nicht festgelegt. Den Software Architekten steht sowohl eine flache Auswahlstrategie, als auch eine Auswahlstrategie mit hierarischem Ausschluss zur Verfügung. Durch die Erweiterbarkeit und Auswahlmöglichkeit der Auswahlstrategien ist es außerdem gelungen prototypisch verschiedene Konzepte zu zeigen und zu evaluieren.

Im Nachhinein wird klar, dass sich die Auswahlstrategie auch auf Verständnis und Logik der DSL auswirkt.

Die flache Auswahlstrategie setzt voraus, dass die Vererbungshierarchie des Domänenmodells nur als Struktur und zum leichteren Verständnis dient. Sie sieht nicht vor, dass innere Knoten der Vererbungsbäume, also Security Entitäten mit Kindern, ihrerseits für die TRA relevante Eigenschaften besitzen. Außerdem werden Asset Kombinationen bei der flachen Auswahlstrategie

nicht vererbt. Das heißt, ein Attack Pattern Kind kann sich auf komplett andere Assets beziehen als dessen Vater. Auch diese Überlegung müsste in die Modellierung der Domäne einbezogen werden.

Die Auswahlstrategie mit hierarchischem Ausschluss orientiert sich wesentlich stärker an dem Vererbungsprinzip, welches aus der objektorientierten Welt bekannt ist. Hier werden die Asset Kombinationen eines Attack Patterns auf dessen Kinder übertragen. Bei der Auswahl der relevanten Attack Pattern wird, bei der Auswahlstrategie mit hierarischem Ausschluss, immer auf die relevanten Assets geachtet. Dabei spielt die Ebene des Attack Patterns in der Vererbungshierarchie keine Rolle.

Zusammenfassend lässt sich sagen, dass beide Auswahlstrategien erheblichen Einfluss auf das Verständnis der Domänenstruktur haben. Um eine klare Abbildung auf die DSL zu ermöglichen, ist ein eindeutiges Verständnis der Domänenstruktur unumgänglich. Die Wahl der Auswahlstrategie widerspricht damit den Anforderungen Epic #01 und Epic #03. Zur besseren Evaluierung der Konzepte ist die Wahl der Auswahlstrategie aber sehr vorteilhaft.

5.3 Dashboard

Das Dashboard wurde flexibel und erweiterbar gestaltet. Durch die Möglichkeit Daten über XML exportieren und importieren zu können (siehe dazu auch Abbildung 23), entspricht es den Anforderungen aus Epic #08 und Epic #09. Die Darstellung der Daten ist in der bisherigen, prototypischen Implementierung jedoch unzureichend. Die Daten werden nur teilweise in Bezug zu den Sicherheitsanforderungen einer TRA gebracht. Die dafür notwendigen Daten, wie das Security Level eines Projekts, sind aber bereits im Datenmodell des Dashboard enthalten. Diese Arbeit legt damit den Grundstein für ein detailliertes Security Dashboard. Die weitere Bearbeitung der Daten kann teil zukünftiger Arbeiten sein.

6 Ausblick und zukünftige Arbeit

Der praktische Teil dieser Arbeit ist als „Proof of Concept“ und Prototyp zu betrachten. Es wurde eine gesamte Infrastruktur entwickelt, die die Arbeitsweise und Vor- und Nachteile einer Security Rules Engine zeigt. Dabei sind verschiedene Herausforderungen und Probleme aufgefallen, die im Rahmen einer zukünftigen Arbeit aufzugreifen sind. Unabhängig davon legt die Arbeit den Grundstein für eine größere Vision (siehe dazu Kapitel 2.1). Im Folgenden sollen einige der Herausforderungen und Funktionalitäten von *Serum* beschrieben werden.

6.1 *Serum*-DSL und *Sweble*

Die Aussagekraft einer durch *Serum* unterstützten TRA steht und fällt mit der Menge der vorhandenen Security Informationen in der Security Knowledge-Base. Durch die Transformation von Daten aus verwandten Arbeiten, wie der *OWASP* Security Angriffssammlung oder der *CWE*, könnte ein anfängliches Mindestmaß an Security Informationen zur Verfügung gestellt werden. Ob eine solche Transformation aufgrund der unterschiedlichen Domänenstruktur möglich ist, war in dieser Arbeit nicht zu untersuchen.

Um die Verwaltung und Manipulation großer Mengen an Daten zu erleichtern, wäre eine Erweiterung der DSL Ladelogik denkbar. Beim Laden von Attack Pattern könnten zum Beispiel auch direkt assoziierte Assets oder Countermeasures geladen werden. Es wäre auch denkbar beim Laden eines Assets, die Kind-Assets oder Vater-Assets zusätzlich zuladen. Um die Ladelogik zu erweitern bedarf es Extrafunktionalität im Deparse-Prozess des *Serum*-DSL *Sweble* Plugins.

Während die DSL ein ideales Mittel für die Formalisierung und für das Verständnis der Security Informationen ist, ist sie wahrscheinlich keine langfristige Lösung. Grafische Lösungen sind zwar oft aufwändig in der Entwicklung und schwieriger in ein formales Modell umzuwandeln, bieten aber langfristig mehr Übersicht bei der Manipulation der Daten. Gerade die komplexen Zusammenhänge der Asset-Kombinationen, wie sie durch die DSL nur oberfläch-

lich modellierbar sind, werden erst durch eine grafische Modellierungssprache in übersichtlicher Weise erkennbar.

Unabhängig davon, ob zur Bearbeitung der Security Informationen eine grafische Oberfläche zur Verfügung steht, kann und muss die semantische Validierung der Eingabe verbessert werden. Um eine stabile und saubere Security Knowledge-Base zu gewährleisten, sind weitere semantische Regeln unabdingbar.

6.2 TRA Engine

Während auch für die TRA eine grafische Darstellung der Security Informationen langfristig sinnvoll scheint, ist eine Portierung der *TRA Engine* Java Applikation auf einen webbasierten Service noch wichtiger. Um *Serum* als natürlichen Teil in den Software Lebenszyklus zu integrieren, sind die Verfügbarkeit und Benutzerfreundlichkeit der *TRA Engine* oberstes Gebot. Im Zuge der Portierung könnte man das Dashboard, beziehungsweise die Auswertung der Security Tests, in den Webservice integrieren. Damit würden zwei der Komponenten *Serum*s vereint werden.

Im Hinblick auf die Vision der Arbeit wäre außerdem eine Erweiterung und Verfeinerung des Security Levels wünschenswert. In der bisherigen Implementierung (Kapitel 3.4.2), ist das Security Level an keinen Standard gebunden. Das heißt, unabhängig von weiteren externen Anforderungen, filtert das Security Level ausschließlich aufgrund der Wahrscheinlichkeit (Likelihood) von Attack Pattern. Stattdessen sollte die Auswahl anhand von Standards, wie den *Common Criteria for Information Technology Security Evaluation* (ISO/IEC, 2012) getroffen werden.

6.3 Dashboard

Langfristig sollen die Ergebnisse einer TRA auch in *Sweble* angezeigt werden können. Damit sind sowohl Security Informationen aus der direkten TRA, als auch Security Testergebnisse gemeint. Die Dashboard Komponente *Serum*s ist auf diese Erweiterung schon explizit ausgelegt. Die Daten, die bisher in einer lokalen Datenbank verwaltet werden, können als XML Format exportiert und so zur Anzeige in anderen Tools verwendet werden. Auch könnten die Ergebnisse der Testautomatisierungsmodule direkt importiert werden.

7 Fazit

Serum wurde als Prototyp einer Software Security Rules Methodology entwickelt und im Rahmen einer Kooperation mit der zentralen Forschungsabteilung der *Siemens AG* evaluiert. Bei der Entwicklung von *Serum* ist eine gesamte Infrastruktur entstanden, die bereits einen Großteil der Phasen des SDLC abdeckt. Das Konzept einer ganzheitlichen, den ganzen SDLC betreffenden Security Rules Methodology wurde sehr positiv aufgenommen. Daher wird das Projekt von Seiten der *Siemens AG* weitergeführt. Die Evaluation der *Serum Sweble* Integration bleibt bis zur Fertigstellung einer ersten *Sweble* Version abzuwarten.

Allgemein sind die Ergebnisse des *Serum* Prototypen und die neuen Erkenntnisse aus der Evaluation sehr positiv und wegweisend für weitere Arbeiten im Bereich der ganzheitlichen Security Assurance. Die Ergebnisse dieser Arbeit werden auch auf dem Imbus-QS Tag vorgestellt.

Danksagung

Zuerst gebührt mein Dank Prof. Dirk Riehle und Hannes Dohrn von der Friedrich-Alexander Universität Erlangen-Nürnberg, die die Entstehung der Arbeit durch ihre Betreuung ermöglicht haben.

Ebenfalls möchte ich mich bei den Mitarbeitern der *Siemens* IT Security Abteilung Tobias Limmer, Dirk Kroeselberg und Christoph Fischer und allen anderen beteiligten Mitarbeitern der Siemens AG für die Hilfe bei der Evaluation der Arbeit bedanken.

Ein besonderer Dank gilt meinem fachlichen Betreuer der *Siemens AG*, Helmut Götz. Seine Unterstützung war für die Anfertigung der Arbeit und die kooperative Zusammenarbeit der *Siemens AG* und der Friedrich-Alexander Universität Erlangen-Nürnberg unerlässlich. Er stand mir jederzeit beratend zur Seite.

Zuletzt danke ich allen Freunden und Verwandten, die mich bestärkten und bei der Erstellung der Arbeit unterstützen.

Abkürzungsverzeichnis

CWE	*Common Weakness Enumeration*
OSR Group	*Open Source Research Group*
OWASP	*Open Web Application Security Project*
SABSA	*Security Architecture Approaches*
SANS	*SysAdmin, Networking and Security*
SDLC	*Software Development Lifecycle*
Serum	*Software SEcurity RUles Methodology*
Sweble	*Sweetly Enabling the Web*
TOGAF	*Enterprise Architecture Methodology*
TRA	*Threat- and Riskanalysis*
UML	*Unified Modeling Language*

Glossar

Nutzer: Theoretisch alle Nutzer von *Serum*. Die größten Nutzergruppen sind in Kapitel 2.3 aufgezählt.

Software System: Miteinander kommunizierende und zusammenwirkende Hardware- oder Softwarebausteine, die zusammen ein Computersystem bilden (Waldo, 2006).

***Serum*:** (Software Security Rules Methodology) Das in dieser Arbeit prototypisch erstellte Softwaresystem. Den genauen Aufbau und die Beschreibung *Serum*s sind in den vorhergehenden Kapiteln ausführlich beschrieben. Obwohl *Sweble* ein eigenständiges System ist, wird es begrifflich in *Serum* inkludiert.

Threat: Im Rahmen der Anforderungen ist ein Threat die Gesamtheit eines wiederkehrenden Bedrohungs- und Gefährdungsmuster. Kapitel 1.3.2.3 enthält weitere Begriffserklärungen.

Asset: Eine Systemfunktionalität oder Komponente, dessen IT-Sicherheit es zu wahren gilt. Hier wird auch eine Systemfunktionalität oder Komponente ohne Schutzanspruch als Asset bezeichnet. Eine allgemeine Definition des Begriffs Asset ist in Kapitel 1.3.2.1 zu finden.

Countermeasure: Eine Maßnahme zur Prävention von Threats, zum Schutz vor Threats oder zur Minimierung des Schadens eines erfolgreichen Angriffs. Weiteres zum Begriff Countermeasure in Kapitel 1.3.2.6.

Recommended Approach: Wissen über die Durchführung, bzw. die Vor- oder Nachbereitung einer Countermeasure. Details in Kapitel 2.4.3.

TRA-Plugin: Eine Software die die TRA-Plugin Schnittstelle implementiert und nach Ausführung ein Ergebnis im *Serum-Dashboard*-Format liefert (Testautomatisierungsmodul). Mehr Information und die Definitionen der Schnittstellen sind in den Kapiteln 3.5 und 4.3 zu finden.

Security Entität: Ein Threat, Asset, Countermeasure oder Recommended Approach.

Knowledge-Base: Eine *Sweble* Datenbank Instanz. Der Aufbau von *Sweble* ist in Kapitel 3.2 dargestellt.

DOM: Ein Dokument Objekt Model ist eine Schnittstellendefinition zum Zugriff auf HTML- und XML Formate. (W3C, 2004)

***Serum-Dashboard* Format:** Ein Format zur einheitlichen Darstellung von Security Testergebnissen.

Definieren: Eine Tätigkeit möglichst eindeutiger und vollständiger Beschreibung.

Assoziieren: Eine Verknüpfung herstellen

In Beziehung setzen: einen räumlichen oder funktionalen Zusammenhangs mehrerer Objekte bestimmen

Threats, deren Assoziationsbedingung erfüllt ist: Die Assoziationsbedingungen eines Threat sind allgemein erfüllt, wenn alle Assets mindestens einer Asset Kombination ausgewählt sind. Das Auswahlverfahren ist im Kapitel 3.4.1 genauer erklärt.

Countermeasures, deren Assoziationsbedingung erfüllt ist: Die Assoziationsbedingungen eines Countermeasure sind allgemein erfüllt, wenn alle assoziierten Threats ausgewählt sind. Auch zu diesem Auswahlverfahren sind weitere Informationen in Kapitel 3.4.1 zu finden.

Recommended Approaches, deren Assoziationsbedingung erfüllt ist: Die Assoziationsbedingungen eines Recommended Approach sind erfüllt, wenn alle assoziierten Countermeasures ausgewählt sind.

TRA-Plugin, deren Assoziationsbedingung erfüllt ist: Die Assoziationsbedingungen eines TRA-Plugin sind erfüllt, wenn alle assoziierten Recommended Approaches ausgewählt sind.

Threats, deren Wahrscheinlichkeit (Likelihood Eigenschaft) eine Darstellung rechtfertigt: Jeder Threat hat bei dessen Definition eine Auftrittswahrscheinlichkeit (Likelihood Eigenschaftswert) erhalten. Ob ein Threat schließlich dargestellt wird, hängt von diesem Wert und von dem vom Nutzer festgelegten Security Level ab.

Countermeasure, deren Aufwand (Effort Eigenschaft) eine Darstellung rechtfertigt: Jeder Countermeasure hat bei dessen Definition einen Aufwand (Effort Eigenschaftswert) erhalten. Ob ein Countermeasure schließlich dargestellt wird, hängt von diesem Wert und von dem vom Nutzer festgelegten Security Level ab.

Funktionale Anforderungen

Die in Tabelle 3 detaillierte Aufzählung funktionaler Anforderungen bezieht sich nicht auf Epic #09 und die in Abschnitt 2 genannten zusätzlichen Anforderungen. Diese sind beide mehr als Qualitätskriterien, also als nicht funktionale Anforderungen, zu betrachten.

Die Formulierung der funktionalen Anforderungen entspricht dem Muster:

Zielsystem + Priorität + Systemaktivität + Ergänzungen + Funktionalität + Bedingungen

Priorität: muss (hohe Priorität), soll (mittlere Priorität), wird (niedrige Priorität)

Systemaktivität: - (selbstständige Systemaktivität), <wem> die Möglichkeit bieten (Benutzerinteraktion), fähig sein (Schnittstellenanforderung)

(Ludwig-Maximilians-Universität München, 2009)

Tabelle 3 *Detaillierte Anforderungen*

Req ID	Ref. Epic ID	Beschreibung
Definition		
1.1	01, 02	*Serum* muss Nutzern die Möglichkeit bieten, Threats zu definieren.
1.2	01, 02, 05	*Serum* muss Nutzern die Möglichkeit bieten, Assets zu definieren.
1.3	03, 04	*Serum* muss Nutzern die Möglichkeit bieten, Countermeasures zu definieren.
1.4	06, 07	*Serum* muss Nutzern die Möglichkeit bieten, Recommended Approaches zu definieren.
1.5	08	*Serum* soll Nutzern die Möglichkeit bieten, TRA-Plugins zu definieren.

Vererbung		
2.1	01, 02	*Serum* muss Nutzern die Möglichkeit bieten, ein Threat mit einem anderen Threat zu assoziieren. Jeder Threat soll höchstens einen anderen Threat assoziieren, darf aber seinerseits beliebig oft assoziiert werden.
2.2	01, 02, 05	*Serum* wird Nutzern die Möglichkeit bieten, ein Asset mit einem anderen Asset zu assoziieren. Jeder Asset soll höchstens einen anderen Asset assoziieren, darf aber seinerseits beliebig oft assoziiert werden.
2.3	03, 04	*Serum* muss Nutzern die Möglichkeit bieten, ein Countermeasure mit einem anderen Countermeasure zu assoziieren. Jeder Countermeasure soll höchstens einen anderen Countermeasure assoziieren, darf aber seinerseits beliebig oft assoziiert werden.
2.4	06, 07	*Serum* wird Nutzern die Möglichkeit bieten, ein Recommended Approach mit einem anderen Recommended Approach zu assoziieren. Jeder Recommended Approach soll höchstens einen anderen Recommended Approach assoziieren, darf aber seinerseits beliebig oft assoziiert werden.
Assoziation		
3.1	01, 02, 05	*Serum* muss Nutzern die Möglichkeit bieten, ein Threat mit beliebig vielen, bereits definierten Assets zu assoziieren.
3.2	02, 05	*Serum* soll Nutzern die Möglichkeit bieten, bei der Assoziation von Threats und Assets, letztere miteinander in Beziehung zu setzen.
3.3	03, 04	*Serum* muss Nutzern die Möglichkeit bieten, ein Threat mit beliebig vielen, bereits definierten Countermeasures zu assoziieren.
3.4	06, 07	*Serum* muss Nutzern die Möglichkeit bieten, ein Countermeasure mit beliebig vielen, bereits definierten Recommended Approaches zu assoziieren.
3.5	08	*Serum* soll Nutzern die Möglichkeit bieten, ein Recommended Approach mit einem TRA-Plugins zu assoziieren.

Manipulation		
4.1	01-07, 09	*Serum* muss Nutzern die Möglichkeit bieten, definierte Security Entitäten, sowie deren Assoziationen in einer Knowledge-Base zu persistieren, falls in der Knowledge-Base noch keine Security Entität gleichen Namens und gleichen Typs vorhanden ist.
4.2	02, 04, 07, 09	*Serum* muss Nutzern die Möglichkeit bieten, Security Entitäten, sowie deren Assoziationen einer Knowledge-Base gezielt, lesbar und vollständig einzusehen.
4.3	01, 03, 06, 09	*Serum* muss Nutzern die Möglichkeit bieten, persistierte Security Entitäten, sowie deren Assoziationen zu manipulieren.
4.4	09	*Serum* muss fähig sein die Security Entitäten und deren Assoziationen in einen DOM zu transformieren.
Threat- and Risk Analysis		
5.1	05	*Serum* muss Nutzern alle Assets einer Knowledge-Base, darstellen.
5.2	05	*Serum* soll Nutzern die Möglichkeit geben, ein Security Level und ein Effort Level auszuwählen.
5.3	05	*Serum* muss Nutzern die Möglichkeit bieten, aus den dargestellten Assets, Assets auszuwählen.
5.4	05	*Serum* muss Nutzern alle Threats einer Knowledge-Base darstellen, deren Assoziationsbedingung erfüllt ist und deren Wahrscheinlichkeit eine Darstellung rechtfertigt.
5.5	05	*Serum* wird Nutzern die Möglichkeit bieten, aus den dargestellten Threats Threats auszuwählen.
5.6	05	*Serum* muss Nutzern alle Countermeasures einer Knowledge-Base darstellen, deren Assoziationsbedingung erfüllt ist und deren Aufwand eine Darstellung rechtfertigt.
5.7	05	*Serum* wird Nutzern die Möglichkeit bieten, aus den dargestellten Countermeasures Countermeasures auszuwählen.
5.8	05	*Serum* muss Nutzern alle Recommended Approaches einer Knowledge-Base darstellen, deren Assoziationsbedingung erfüllt ist.
5.9	08	*Serum* soll Nutzern die Möglichkeit bieten, TRA-Plugins auszuführen, deren Assoziationsbedingung erfüllt ist und die der *Serum* Instanz lokal vorliegen.

Dashboard		
6.1	08	*Serum* wird Ergebnisse von TRA-Plugin Ausführungen persistieren.
6.2	08	*Serum* soll Ergebnisse von TRA-Plugin Ausführungen in Relation zu einem Security Level darstellen.
6.3	09	*Serum* wird fähig sein, die Ergebnisse von Testwerkzeugen in einheitlichem *Serum-Dashboard*-Format zu exportieren.

Abbildungen

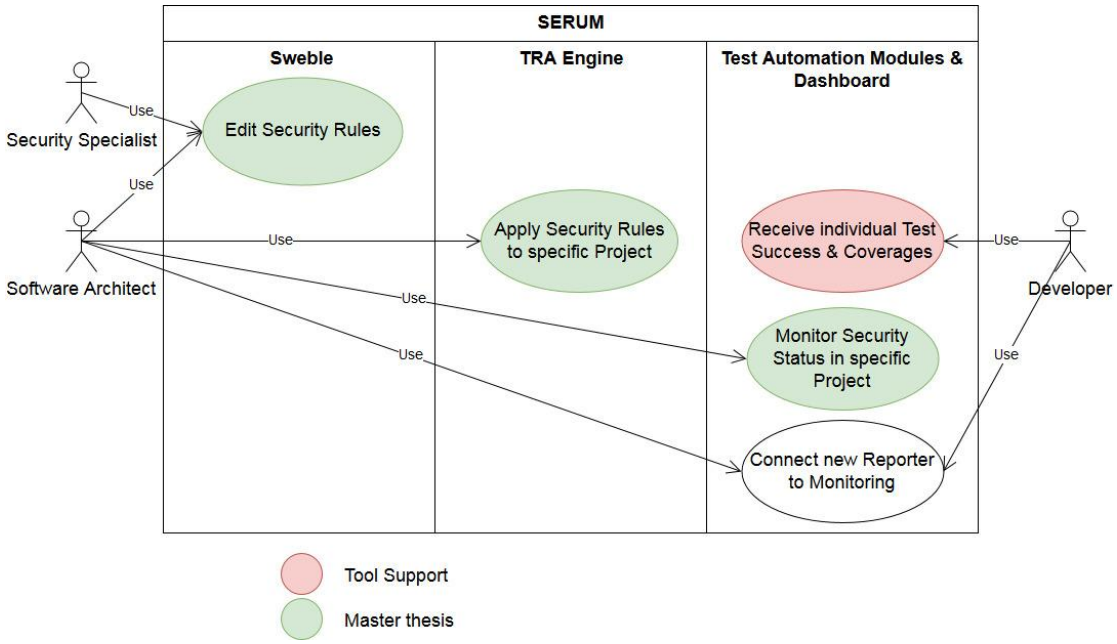

Abbildung 21 *Anwendungsfalldiagramm Überblick*

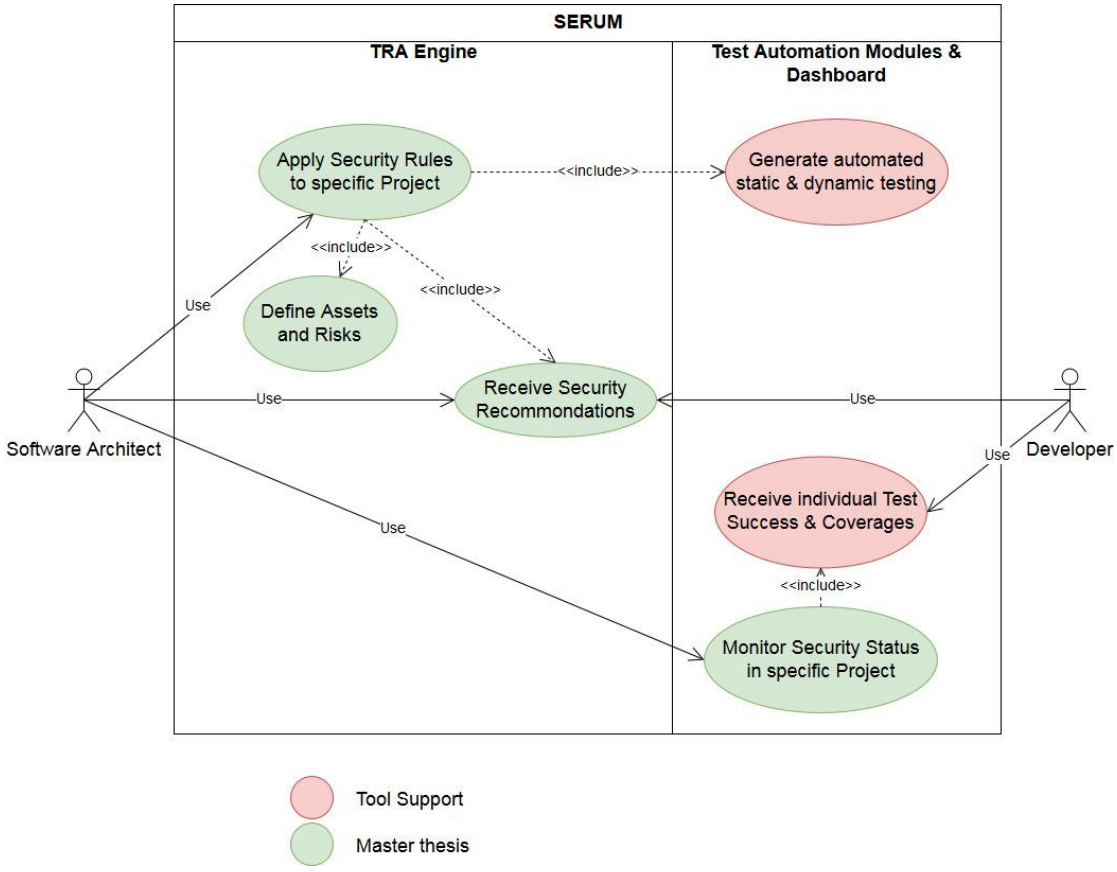

Abbildung 22 *Anwendungsfalldiagramm Detail*

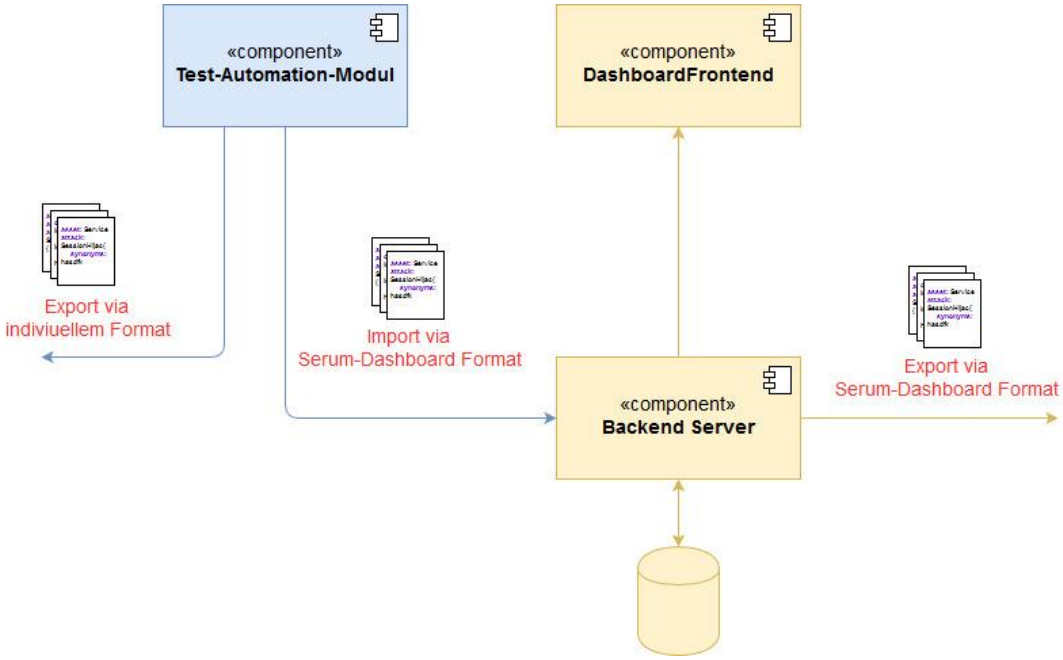

Abbildung 23 *Komponentendiagramm Dashboard*

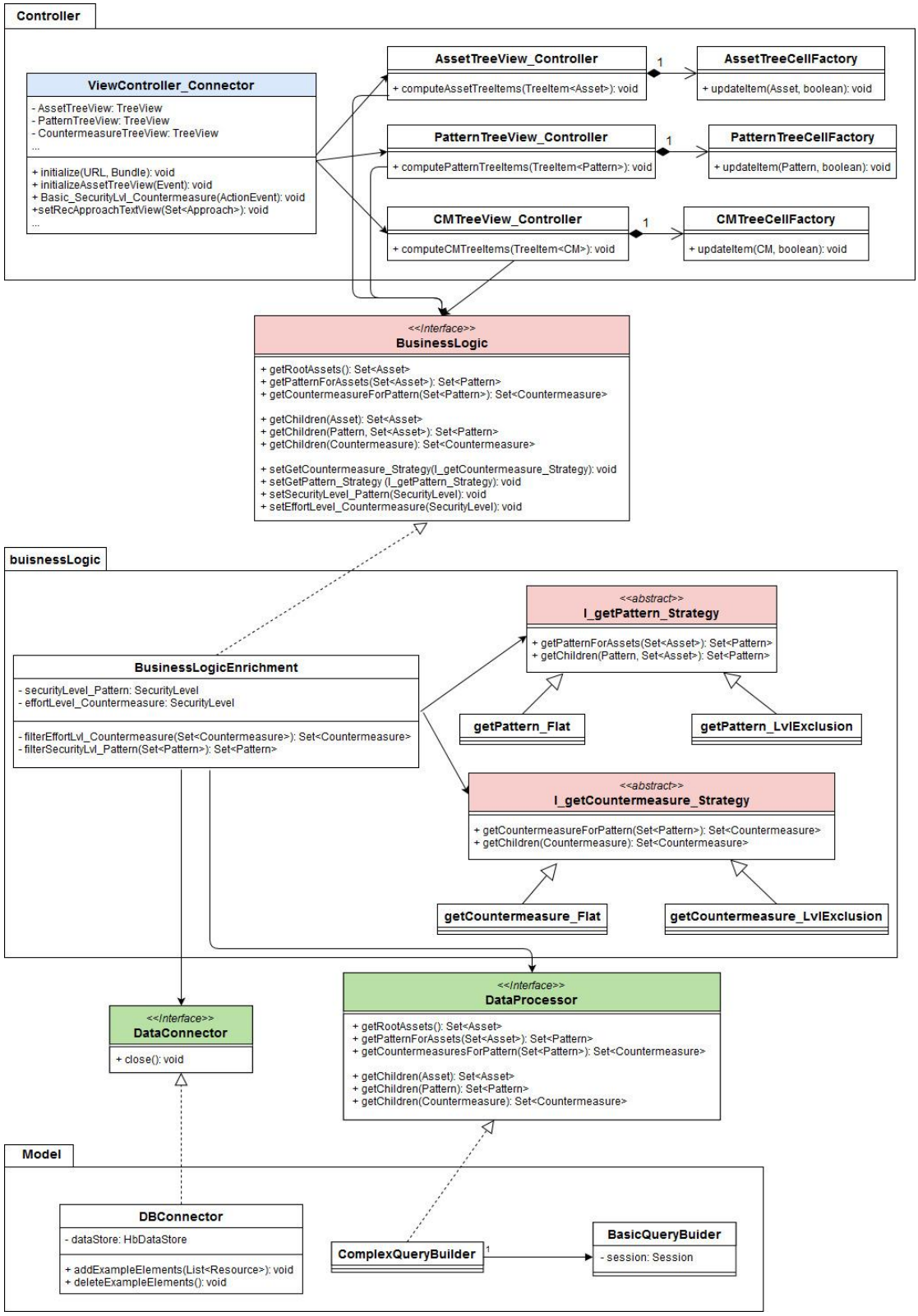

Abbildung 24 *Klassendiagramm TRA Engine gesamt*

Bill of Materials

Serum ist intern in sieben Projekte geteilt. Die Projekt Abhängigkeiten sind als erste Elemente im Bill of Material gelistet. Die Abhängigkeiten sind transitiv.

Tabelle 4 *Bill of Material TRA_Engine_FX*

GroupId	Name	Version	License
Api DBInitializer			
com.github.johnrengelman.shadow	shadow	2.0.1	Apache 2.0
org.sonarsource.scanner.gradle	sonarqube-gradle-plugin	2.5	LGPL 3
Undecorator	Undecorator	1.0	BSD
commons-collections	commons-collections	3.2.2	Apache 2.0
org.apache.commons	commons-lang3	3.6	Apache 2.0

Tabelle 5 *Bill of Material Api*

GroupId	Name	Version	License
junit	junit	4.12	EPL 1.0
org.apache.commons	commons-lang3	3.6	Apache 2.0
ro.fortsoft.pf4j	pf4j	1.3.0	Apache 2.0

Tabelle 6 *Bill of Material DBInitializer*

GroupId	Name	Version	License
Parser			
log4j	log4j	1.2.17	Apache 2.0
org.hibernate	hibernate-core	4.3.7	LGPL 2.1
org.eclipse.emf	org.eclipse.emf.teneo	2.1.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.teneo.annotations	2.1.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.teneo.hibernate	2.1.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.teneo.hibernate.mapper	2.1.0	EPL 1.0
org.hsqldb	hsqldb	2.4.0	BSD

Tabelle 7 *Bill of Material Parser*

GroupId	Name	Version	License
DSL			
log4j	log4j	1.2.17	Apache 2.0
junit	junit	4.12	EPL 1.0
commons-io	commons-io	2.5	Apache 2.0
org.eclipse.emf	org.eclipse.emf.ecore	2.12.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.common	2.12.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.ecore.xmi	2.12.0	EPL 1.0
org.eclipse.emf	org.eclipse.emf.mwe.utils	1.3.21	EPL 1.0
org.eclipse.xtext	org.eclipse.xtext	2.12.0	EPL 1.0
org.xmlunit	xmlunit-legacy	2.3.0	Apache 2.0 BSD 3-clause
com.google.inject	guice	4.1.0	Apache 2.0

Tabelle 8 *Bill of Material BDD-Security Plugin*

GroupId	Name	Version	License
Api SecurityTest_DB			
ro.fortsoft.pf4j	pf4j	1.3.0	Apache 2.0
junit	junit	4.12	EPL 1.0
org.apache.commons	commons-lang3	3.6	Apache 2.0
log4j	log4j	1.2.17	Apache 2.0
dom4j	dom4j	1.6.1	EPL 1.0
commons-io	commons-io	2.5	Apache 2.0
xerces	xercesImpl	2.11.0	Apache 2.0

Tabelle 9 *Bill of Material Xanitizer Plugin*

GroupId	Name	Version	License
Api SecurityTest_DB			
ro.fortsoft.pf4j	pf4j	1.3.0	Apache 2.0
junit	junit	4.12	EPL 1.0
org.apache.commons	commons-lang3	3.6	Apache 2.0
log4j	log4j	1.2.17	Apache 2.0
dom4j	dom4j	1.6.1	EPL 1.0
commons-io	commons-io	2.5	Apache 2.0
xerces	xercesImpl	2.11.0	Apache 2.0

Tabelle 10 *Bill of Material SecurityTest_DB*

GroupId	Name	Version	License
com.github.johnrengelman.shadow	shadow	2.0.1	Apache 2.0
mysql	mysql-connector-java	5.1.6	GPL 2.0
org.hibernate	hibernate-core	4.3.7	LGPL 2.1
log4j	log4j	1.2.17	Apache 2.0
xerces	xercesImpl	2.11.0	Apache 2.0

Tabelle 11 *Bill of Material Dashboard*

GroupId	Name	Version	License
SecurityTest_DB			
javax.servlet	javax.servlet-api	4.0.0	CDDL 2
org.json	json	20160810	JSON
org.jdom	jdom	2.0.2	Apache
log4j	log4j	1.2.17	Apache 2.0
xerces	xercesImpl	2.11.0	Apache 2.0
mysql	mysql-connector-java	5.1.6	GPL 2.0
dom4j	dom4j	1.6.1	EPL 1.0
org.hibernate	hibernate-core	4.3.7	LGPL 2.1
javax.xml.bind	jaxb-api	2.3.0	CDDL 1.1 GPL 2.0
org.slf4j	slf4j-api	1.7.25	MIT

Referenzen

ISi-Projektgruppe. (2011). *Internet-Sicherheit: Einführung, Grundlagen, Vorgehensweise.* Bonn: Bundesamt für Sicherheit in der Informationstechnik.

Abolhassen, F. (2017). *Security Einfach Machen.* Wiesbaden: Springer Fachmedien.

Al-Shaer, E., El-Atawy, A., & Samak, T. (April 2009). Automated Pseudo-Live Testing of Firewall Configuration Enforcement. *IEEE Journal on selected areas in communications, Vol. 27, No. 3*, S. 302-314.

Black, P. E., Kass, M., & Fong, E. (2006). Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics. (S. Workshop on Software Security Assurance Tools, Techniques, and Metrics). National Institute of Standards and Technology.

Bozic, J., & Wotawa, F. (2014). Security Testing Based on Attack Patterns. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation Workshops* (S. 4-11). Graz, Austria: ICSTW.

Continuum Security. (13. September 2017). *BDD-Security - Security Testing Framework*. Von https://www.continuumsecurity.net/bdd-security/ abgerufen

Continuum Security. (01. 09 2017). *IriusRisk - Threat Modeling Tool*. Von https://www.continuumsecurity.net/threat-modeling-tool/ abgerufen

Dohrn, H., & Riehle, D. (2011). *WOM: An object model for Wikitext.* Erlangen-Nürnberg: University of Erlangen, Dept. of Computer Science.

Elrakaiby, Y., Mouelhi, T., & Le Traon, Y. (2012). Testing Obligation Policy Enforcement using Mutation Analysis. *Fifth International Conference on Software Testing, Verification and Validation* . Montreal, QC, Canada : IEEE.

Feinman, T., Goldman, D., Wong, R., & Cooper, N. (kein Datum). *Security Basics: A Whitepaper.* PricewaterhouseCoopers LLP.

Fischer, S., Steinacker, A., Bertram, R., & Steinmetz, R. (1998). *Open Security: Von den Grundlagen zu den Anwendungen.* Berlin: Springer-Verlag.

Graham, J., Howard, R., & Olson, R. (2010). *Cyber Security Essentials.* New York: CRC Press.

Howard, J. D., & Longstaff, T. A. (1998). *A Common Language for Computer Security Incidents.* Department of Energy.

ISO/IEC. (2012). *Common Criteria for Information Technology Security Evaluation.*

Kazman, R., Klein, M., & Clement, P. (2000). *ATAM: Method for Architecture Evaluation.* Pittsbburgh, PA: Carnegie Mellon Software Engineering Institute.

Kostopoulos, G. K. (2013). *Cyberspace and Cybersecurity.* New York: CRC Press.

Kriha, W., & Schmitz, R. (2008). *Internet-Security aus Software-Sicht.* Berlin: Springer-Verlag.

Lee, R. B. (2013). *Security Basics for Computer Architects.* Morgan & Claypool Publishers.

Lehtinen, R., Russell, D., & Gangemi Sr., G. (2006). *Computer Security Basics.* O'Reilly & Associates, Inc.

Ludwig-Maximilians-Universität München. (2009). *Programmierung und Softwaretechnik* . Von http://www.pst.ifi.lmu.de/Lehre/fruhere-semester/sose-2009/seprakt/Formulierungsregel_DE_Rupp_Schablone.pdf abgerufen

MITRE. (05. Mai 2017). Von Common Weakness Enumeration - 384: Seesion Fixation: https://cwe.mitre.org/ abgerufen

Neuninger, T. (2008). *Erhöhung der Software-Sicherheit mit statischer Source-Code-Analyse in einem Qualitätsmanagementwerkzeug.* Linz: Institut für Wirtschaftsinformatik Software Engineering.

Object Management Group. (2016). *Automated Source Code Security Measure.*

Open Group TOGAF-SABSA Integration Working Group. (2011). *TOGAF and SABSA Integration.* The Open Group and the SABSA Institute.

Partner, R. &. (18. 08 2017). *Das Zeitalter der Digitalisierung*. Von http://www.roedl.de/themen/digitalisierung/ abgerufen

Pauli, J. J., & Xu, D. (2005). Misuse Case-Based Design and Analysis of Secure Software Architecture. *Proceedings of the International Conference on Information Technology: Coding and Computing.* Madison, Fargo USA: Dakota State University, North Dakota State University.

Rigs IT. (20. 09 2017). *Xanitizer*. Von https://www.rigs-it.net/ abgerufen

Schoenfield, B. S. (2015). *Securing Systems - Applied Security Architecture and Threat Models.* New York: CRC Press.

Security Compass. (01. 09 2017). *SD Elements*. Von https://www.securitycompass.com/sdelements/ abgerufen

Singer, P., & Friedman, A. (2014). *Cybersecurity and Cyberwar: What everone needs to know.* New York, USA: Oxford University Press.

Svoboda, D., Flynn, L., & Snavely, W. (2016). *Static Analysis Alert Audits; Lexicon & Rules.* Boston: CERT Division.

The OWASP Foundation. (2013). *OWASP Top 10 - 2013*.

W3C. (07. April 2004). *Document Object Model (DOM) Level 3 Core Specification*. Von https://www.w3.org/TR/DOM-Level-3-Core/Overview.html#contents abgerufen

Waldo, J. (2006). *On System Design.* Portland: Sun Microsystems Laboratories.