Karlsruher Institut für Technologie

Ausarbeitung zum Proseminar "Web-Anwendungen und Serviceorientierte Architekturen"(WASA)

Bearbeiter: Julian Todt
Semester: 3. BA-Semester
Projektteam: Pen_iCC

Wintersemester 2018/2019

Cooperation & Management (C&M, Prof. Abeck)
Institut für Telematik, Fakultät für Informatik
www.cm.tm.kit.edu

Inhaltsverzeichnis

1 Einarbeitung

In diesem Kapitel soll sich mit den grundsätzlichen Ideen, Prinzipien und Technologien des Cooperation & Management (C&M)-Entwicklungsprozesses theoretisch auseinandergesetzt werden.

1.1 C&M-Entwicklungsprozess

Der Entwicklungsprozess bei C&M versucht aus den häufigsten Problemen der Softwareentwicklung in der Industrie zu lernen, welche darauf zurückzuführen sind, dass a) das falsche und b) falsch entwickelt wird. C&M verwendet daher eine Kombination aus Behaviour-Driven Development (BDD) und Domain-Driven Design (DDD), welche Technologien und Methoden bereitstellen, um die Probleme des 'das Richtige' und 'richtig' entwickeln zu lösen. [Co18b]

1.1.1 Behaviour-Driven Development (BDD)

Die Grundstruktur des Entwicklungsprozesses bei BDD kann Abbildung 1.1 entnommen werden.

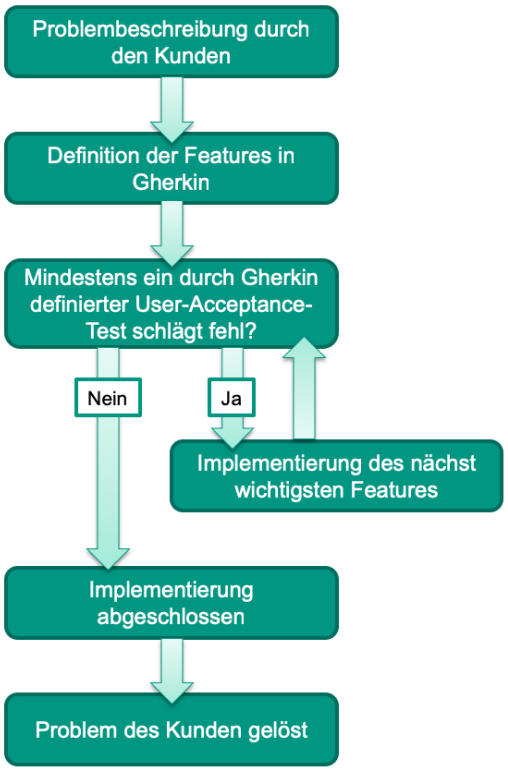


Abbildung 1.1: Entwicklungsprozess bei BDD

Dabei ähnelt das Vorgehen in der Implementierungsphase beim BDD dem des Test-Driven Development. Bei diesem werden zunächst Unit-Tests geschrieben, welche die gewünschte Funktionalität einer Klasse widerspiegelen (sollen) und daraufhin solange implementiert bis die Tests erfolgreich sind. Das potenzielle Problem hierbei liegt jedoch darin, dass die vom Implementierer geschriebenen Unit-Tests nicht unbedingt die vom Kunden gewünschten bzw. die in einem Pflichtenheft definierten Funktionalitäten voll umfassend wiedergeben. Dieses Problem löst das BDD durch die in der Analyse-Phase definierten Features (hier in Gherkin). Diese geben durch ihre Szenarien exakt die Anforderderungen des Kunden

wieder und lassen sich durch die Formalisierung in Gherkin testen, sodass sich diese User-Acceptance-Tests ideal als Tests beim Test-Driven Development eignen. [Co18b]

Die Unterteilung der Funktionalität der gewünschten Software in diese Features erlaubt bzw. erfordert außerdem ein agileres, iteratives Vorgehen, bei welchem Features nacheinander implementiert und getestet werden können, sodass nach jedem Feature eine funktionsfähige Software vorliegt, die dem Kunden gezeigt werden kann. Somit lässt sich BDD auch hervorragend zusammen mit anderen agilen Entwicklunsmethoden verwenden. [SW11]

1.1.2 Domain-Driven Design (DDD)

Bei DDD steht die Domäne im Mittelpunkt. Das bedeutet, dass, auch wenn der Entwicklungsprozess bei C&M nicht dem üblichen Wasserfallmodell folgt, Analyse und Entwurf trotzdem nicht weggelassen werden und sich in Form von UML-Klassen- und Komponentendiagrammen vor der Implementierung Gedanken über den Entwurf und die Domäne gemacht werden. Im Zusammenspiel mit BDD ergibt sich damit ein iteratives Domänenmodell, welches vor der Implementierung der einzelnen Featuers erweitert und überarbeitet wird. Dieses Vorgehen erlaubt auch die Erweiterung bereits bestehender Systeme und setzt nicht eine vollständige Neuentwicklung voraus. DDD erlaubt insbesondere die Trennung von anwendungsspezifischen und -agnostischen Softwareteilen, welche beispielsweise die C&M-Mikroservice-Architektur mit Backend und Backend for Frontend (BFF) vorsieht. [Ev03] [Co18b]

1.1.3 Toolunterstütztes Entwickeln

Ein weiterer Aspekt des Entwicklungsprozesses bei C&M ist nach [Co18b], dass alle Phasen und Bereiche der Entwicklung durch entsprechende Tools unterstützt werden, um diesen zu erleichtern und das Ergebnis zu verbessern.

- *Kommunikationstools:* Um die Koordination und Kommunikation zwischen Teammitgliedern zu verbessern, werden Plattformen wie der Teamserver (Sharepoint) und Chat Tools wie Slack genutzt.
- *Versionsverwaltung:* Um Änderungen an Code oder Latex-Dokumenten besser nachvollziehen zu können und das gleichzeitige Arbeiten am Projekt durch mehrere Teammitglieder zu unterstützen wird Git und Bitbucket bei C&M eingesetzt.
- *Entwicklungstools:* Um die Arbeit der einzelnen Teammitglieder in den einzelnen Phasen der Entwicklung zu erleichtern, werden jeweils spezielle Tools eingesetzt. Beispielsweise verschiedene Integrated Development Environments (IDEs) in der Implementierungsphase wie Eclipse und IntelliJ oder zur Erstellung von UML-Diagrammen der Enterprise Architect.

1.2 Docker

Die Idee hinter der Containerisierung von Anwendungen ist es, diese mit all ihren Abhängigkeiten zusammen in Container-Images zu packen, sodass diese portabel, skalierbar und unabhängig von der darunterliegenden Infrastruktur ausführbar sind. Der am meisten verbreitete Standard für diese Container ist *Docker* und ist ein zentraler Bestandteil von Mikroservice-Architekturen. Damit sind Container die flexiblere Alternative zu virtuellen Maschinen in Rechenzentren, da anstatt von VMs Applikationen in Form von Docker-Containers direkt als Prozesse auf Hypervisorn laufen können, was Abbildung 1.2 zeigt. Dadurch können Ressourcen gespart werden und die Verwaltung der Infrastruktur wird einfacher.

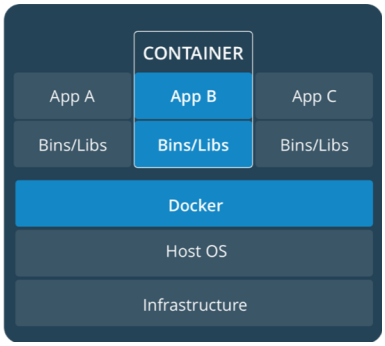


Abbildung 1.2: Aufbau eines Docker-Deployment-Umgebung [Do18]

Eine containerisierte Anwendung mit ihren Abhängigkeiten bezeichnet man im portablen Zustand als Image und im ausgeführten Zustand als Container. Docker-Images werden mithilfe sogenannter Dockerfiles gebaut, welche die genaue Bauanleitung für diese beinhalten. Dabei wird meistens von einem zugrundeliegendem Base-Image ausgegangen, welches durch das Dockerfile um eine Schicht (Layer) erweitert wird. Das spart Speicherplatz, da Docker jede Schicht nur einmal speichern muss, obwohl diese von mehreren Images verwendet werden kann. Bei der Erweiterung der Schicht können Daten in das Image reinkopiert werden (beispielsweise die kompilierte Anwendung), Befehle im Container ausgeführt werden (beispielsweise ein weiteres Paket installieren), der beim Start des Containers auszuführende Befehl gesetzt werden und weiteres. Durch den Aufruf von `docker build` wird aus dem Dockerfile das eigentliche Image gebaut und in der lokalen Registry, dem Speicherort für alle Images, abgelegt. Um dieses mit anderen zu teilen oder zu deployen, muss es daraufhin mit `docker push` zur gewählten Registry hochgeladen werden.

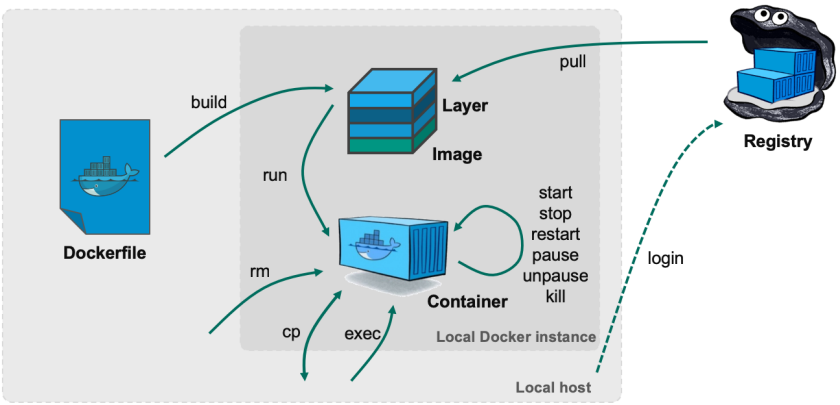


Abbildung 1.3: Überblick über Docker-Befehle

Abbildung 1.3 zeigt alle Stationen des Lifecycles eines Docker-Containers sowie die entsprechenden Kommandozeilen-Befehle. Beispielsweise kann ein gerade lokal gebauter Container mit `docker run` direkt ausgeführt werden. Da Docker-Container von ihrer Ausführungsumgebung komplett unabhängig sind, kann so schon während der Entwicklung der Anwendung die ordnungsgemäße Ausführung auf dem Zielsystem sichergestellt werden. Außerdem können Container aufgrund ihrer geringen Größe und des Schichtenmodells sowie ihrer Unabhängigkeit von anderen Systemen besonders schnell aktualisiert und skaliert werden.

Ein bekanntes Orchestrierungstool für Container ist Kubernetes. Es ermöglicht automatisches Deployment, Skalierung und Management von Anwendungen in Microservices. Außerdem stellt es Schnittstellen zum Load Balancing der Container, zum Konfigurationsmanagement sowie zur Speicher-Orchestrierung bereit.

2 Praktischer Kurs: TLM-Demo

Im Folgenden sollen die Erfahrung mit einer einfachen Beispielanwendung, dem Todolistmanagement (TLM) beschrieben werden. Die Anwendung folgt der C&M-Microservice-Architektur und besteht daher aus Frontend, BFF und Backend. Backend und BFF sind Spring-Boot-Anwendung, das Frontend ist in Angular geschrieben. Die Primärquelle für alle Texte in diesem Kapitel ist, falls nichts anderes erwähnt, [Co18a].

2.1 Installation der Entwicklungsumgebung

Entgegen der Empfehlungen der READMEs der Projekte, werde ich nicht die Spring Tool Suite, welche auf Eclipse basiert, als IDE verwenden, sondern IntelliJ, da ich mit dieser mehr Erfahrung habe und hier bereits einige hilfreiche Erweiterungen integriert sind. Die Installation des Java Development Kit (JDK) 10 war eine größere Herausforderung als erwartet, da diese von *Oracle* nach dem Release von JDK 11 als End-Of-Life gekennzeichnet wurde. Da die TLM-Demo jedoch nicht mit JDK 11 kompatibel ist, war dies keine Option. Im Gegensatz zu den normalen Java-Versionen verlangt *Oracle* für End-Of-Life Versionen einen Login, was die Installation deutlich komplizierter als erwartet gemacht hat. Schlussendlich habe ich dies jedoch geschafft und konnte wie in Abbildung 2.1 zu sehen, die Microservices des TLM erfolgreich inspizieren.

2.2 Bereitstellen und Ausführung der Anwendung

Um die Backend- und BFF-Microservices zu starten, müssen die Sping-Boot-Anwendungen zunächt mit `mvn package` gebaut werden und können dann mit `java -jar target/todo-management -0.0.1-SNAPSHOT.jar` gestartet werden. Beide Anwendungen stellen eine Weboberfläche bereit, welche die durch Swagger definierten API-Endpunkte zeigt, welche im Browser unter `localhost:{8080,8081}` betrachtet werden können, wie in Abbildung 2.2 zu sehen ist.

Das Frontend kann durch den Aufruf `ng serve` im entsprechenden Ordner gestartet werden nachdem mit `npm install` die benötigten Abhängigkeiten installiert wurden. Danach kann man wie in Abbildung 2.3 die grafische Oberfläche des TLM sehen.

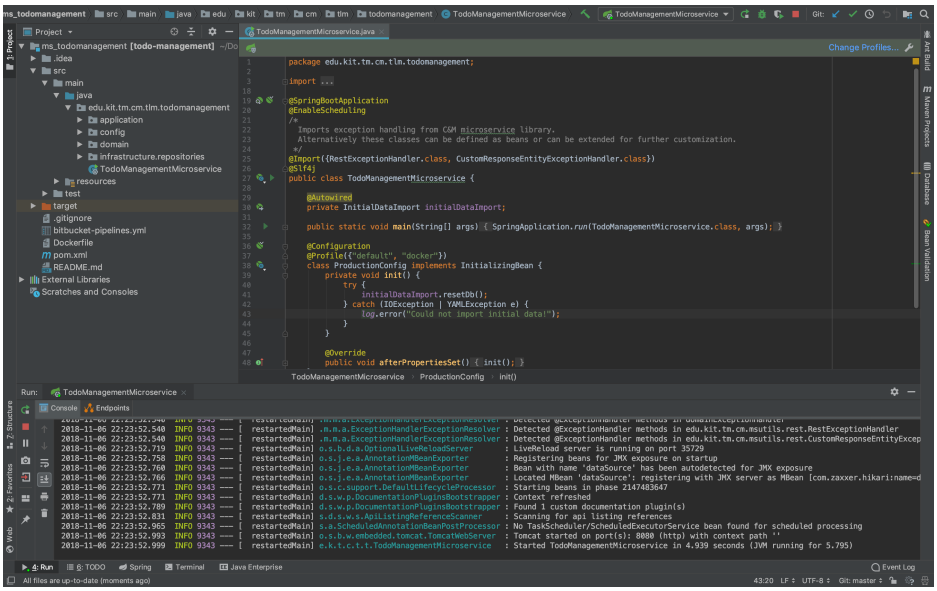


Abbildung 2.1: Das TLM Backend in IntelliJ

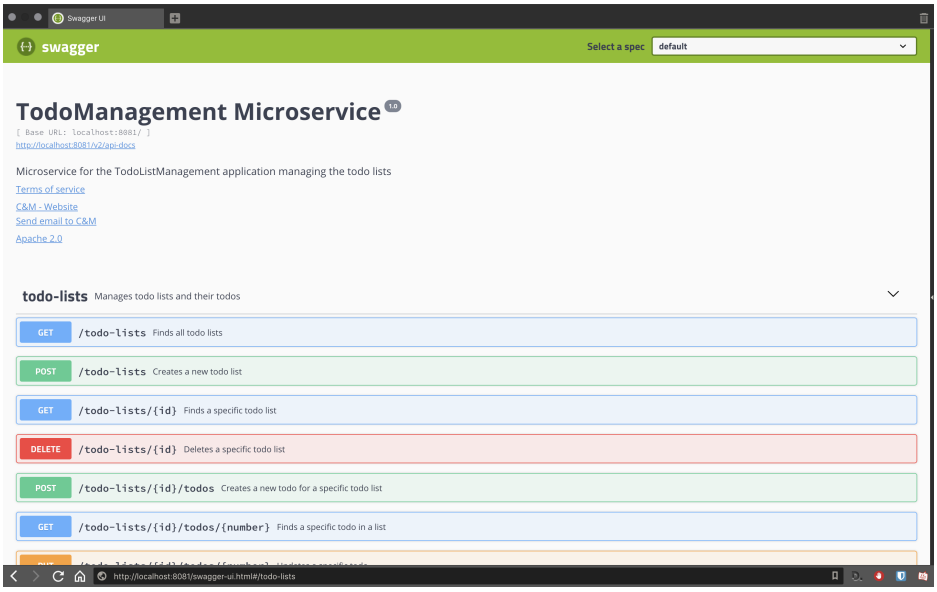


Abbildung 2.2: Die API des BFF

2.3 Bearbeitung der Aufgaben

Im Folgenden soll sich genauer mit den Technologien, der Implementierung und den Konzepten der TLM-Demo beschäftigt werden, indem zu den drei Microservicen einige Fragen beantwortet werden.

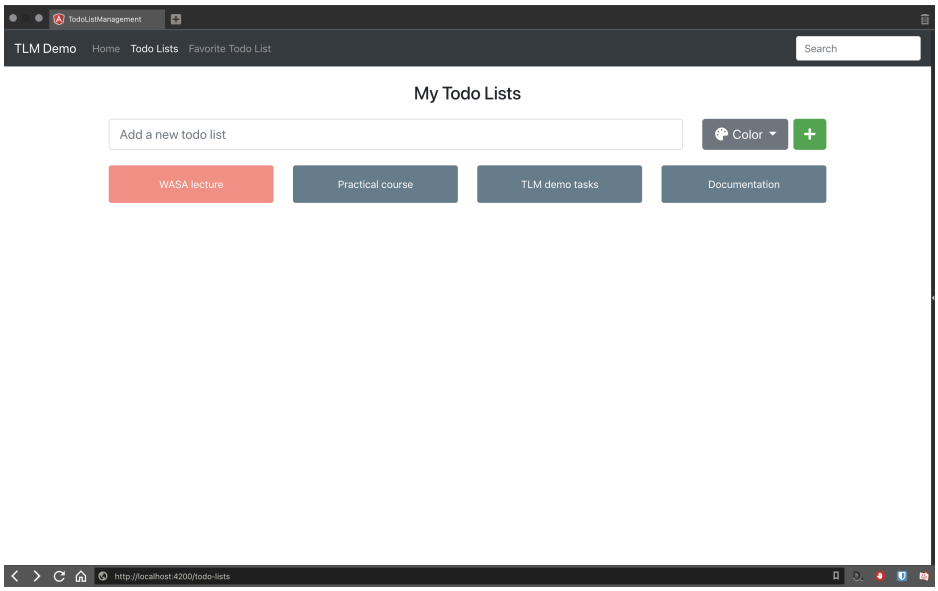


Abbildung 2.3: Todolistmanagement

2.3.1 Aufgaben zum Backend

Die Domänenlogik der TLM-Demo steckt im Backend, welches damit die Basis für die anderen Microservices bereitstellt. Deshalb wollen wir uns genauer anschauen, wie es aufgebaut ist, um zu verstehen, wie es erweitert werden kann.

2.3.1.1 Backend-Struktur

Frage: *Wie ist das Backend strukturiert und was sind Vorteile dieser Implementierung?*
Das Backend ist so strukturiert, dass Datenspeicherung, Domainmodellierung und API-Logik getrennt sind, wodurch viele Klassen anwendungsagnostisch sind und in ähnlichen Microservices wiederverwendet werden können. Die Ordnerstruktur im `edu.kit.tm.cm.tlm.todomanagement`-Paket sieht wie folgt aus:

- `application`: Beinhaltet die Logik zur Darstellung der REST-API des Microservices. Insbesondere das `TodoListApi`-Interface unter `controllers/api`, sowie die Klassen zur Strukturierung der API-Requests.
- `config`: Beinhaltet Konfigurationsfiles für den Microservice, insbesondere die `SwaggerConfig`.
- `domain`: Beinhaltet alle Klassen des Domänenmodells, sowie deren Attribute und Assoziationen. Beispielsweise die `TodoList` und `Todo`-Klassen unter `models`.
- `infrastructure`: Beinhaltet die Logik zum Speichern der Daten, beispielsweise der Todolisten in `TodoListRepository`.

2.3.1.2 API

Frage: *Welche Schritte sind notwendig, um die API zu erzeugen? Welche Klassen sind dafür verantwortlich und wie wird die API dargestellt?*
Um die API zu erzeugen, musste zunächst ein Domänen-Modell für die TLM-Demo erstellt werden aus welchem dann die Domänen-Objekte herausgezogen wurden. Diese bilden die Models im Backend. Daraufhin wurde die API definiert, indem für jedes Model die CRUD-Operationen gewählt wurden, also Erzeugen (Create), Auslesen (Read), Aktualisieren (Update) und Löschen (Delete). Die wichtigste Klasse der API ist das `TodoListApi`-Interface unter `application/controllers/api`. Unter Verwendung der Annotationen von *Swagger* beschreibt es die implementierten API-Endpunkte des Microservices. Außerdem gibt es Informationen über die Fehlercodes, welche von den Endpunkten zurückgegeben werden können, sowie die Logik, mit welcher die Antworten für die API-Anfragen generiert werden. In Abbildung 2.4 sieht man beispielsweise die Logik für den `PUT /ID/todos/NR`-Endpunkt zum Bearbeiten des Todos `NR` in der Todoliste `ID`.
Außerdem werden noch einige andere Klassen zur Erzeugung der API genutzt, insbesondere die-

```
@PutMapping("/{id}/todos/{number}")
@ApiOperation(value = "Updates a specific todo")
@ApiResponses({
        @ApiResponse(code = 400, message = "Todo was not valid"),
        @ApiResponse(code = 404, message = "Todo list or todo not found"),
        @ApiResponse(code = 409, message = "Todo with the same content already exists in this list")
})
TodoResponse updateTodo(@PathVariable Long id,
                        @PathVariable Integer number,
                        @Valid @RequestBody UpdateTodoRequest updatedTodo
);
```

Abbildung 2.4: API-Endpunkt zum Bearbeiten von Todos

jenigen unterhalb des `application`-Ordners, beispielsweise die Request, Response und Service Datentransferobjekt-Klassen, welche die jeweiligen Vorgänge im API-Aufrufs-Prozess modellieren und validieren. Abbildung 2.5 zeigt beispielsweise die Klasse für einen Request zum Bearbeiten eines Todos, an welcher man sieht, welche Daten einem solchen Aufruf mitgegeben werden müssen, damit dieser erfolgreich ausgeführt werden kann.
Diese Klassen verweisen wiederrum auf die Klassen des Domänenmodells aus dem `domain`-Ordner,

```
@Data
@ApiModel("Todo-PatchRequest")
public class PatchTodoRequest {
    @ApiModelProperty(position = 1)
    Integer position;

    @ApiModelProperty(position = 2)
    String content;

    @ApiModelProperty(position = 3)
    Boolean done;

    @ApiModelProperty(position = 4)
    Optional<String> description;
}
```

Abbildung 2.5: Request-DTO zum Bearbeiten von Todos

welche die Basis für die jeweiligen API-Operationen sind.

2.3.1.3 Kardinalität

Frage: *Was ist die Kardinalität der Beziehung zwischen TodoListen und den enthaltenen Todos? Wie ist diese implementiert?*
Eine TodoListe hat 0 bis n Todos, ein Todo ist Mitglied genau einer TodoListe. Außerdem sollen zugehörige Todos gelöscht werden, wenn die entsprechende TodoListe gelöscht wird. Die Implementierung dieser Beziehung erfolgt durch Annotationen im Code und kann man unter `domain/models/` sehen, siehe Abbildung 2.6.

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "todoList", orphanRemoval = true)
@OrderBy("position asc")
private List<Todo> todos;

@ManyToOne
private User user;
```

Abbildung 2.6: Implementierung der Beziehung Todo-TodoList

2.3.1.4 IAM

Frage: *Angenommen, Identity und Accessmanagement würde zum TLM hinzugefügt werden. Jetzt sollen persönliche Listen eines Users nur noch für diesen gespeichert und sichtbar sein. Welche Schritte sind notwendig, um diese Funktionlität hinzuzufügen? Wie wirkt sich der IAM Kontext auf das Domänenmodell aus? Welche Klassen würden hinzugefügt oder bearbeitet werden und warum?*
Die TLM-Demo hat bereits eine User-Klasse im Domänenmodell (siehe `domain/models/User.java`), welche jedoch aktuell noch keine Benutzung findet.
Grundsätzlich würde man einen weiteren Microservice einsetzen, der für Identity und Access Management zuständig ist, sodass die gesamte Logik bezüglich dieser Domäne getrennt von der Domäne des TodoListManagements stattfinden kann. Das Backend des TodoListManagement würde sich für jede TodoListe nur, wie bereits durch das aktuelle Modell angedeutet ein weiteres Attribut, welches den Benutzer eindeutig identifiziert, speichern und diesen bei den API-Aufrufen zur TodoListe mitgeben. Die Zugriffssteuerung, sodass jeder User nur auf seine eigenen TodoListen zugreifen kann, würde im BFF hinzugefügt werden, da dies bereits anwendungsspezifische Logik ist.

2.3.2 Aufgaben zum BFF

Oft bestehen Anwendungen nur aus Backend und Frontend. Warum ein BFF sinnvoll ist und wie es aufgebaut ist, wird in diesem Abschnitt behandelt.

2.3.2.1 BFF Weiterleitung

Frage: *Die meisten Anfragen an die TLM-Demo werden nur vom BFF weitergeleitet. Nichtsdestotrotz wurde das BFF implementiert - warum? Was sind Vor- und Nachteile?*
Das BFF ist dazu da, Domänen- und Applikationslogik zu trennen. Dabei übernehmen die Microservices die Rolle der Domänen und beinhalten nur die Logik, die aus einem UML-Klassendiagramm der spezifischen Domäne stammt. Eine Applikation kann jedoch aus mehreren Microservicen (/Backends) bestehen. Diese werden durch das BFF orchestriert, welches so die Zusammenarbeit mehrerer Domänen für eine Anwendung ermöglicht. Das BFF enthält außerdem die Applikationslogik, also den Code der anwendungsspezifisch ist und nur von einem speziellen Frontend genutzt wird.
Im Falle des TLM ist das BFF nicht besonders umfangreich, da nur sehr wenig anwendungsspezifischer Code enthalten ist (nur das Färben von WASA-TodoListen) und nur ein einziger Microservice. Dies ist auch direkt ein Nachteil des BFF: Bei nicht so umfangreichen Anwendungen entsteht durch ein BFF sehr viel unnötiger Code, das System wird stark verlangsamt und benötigt deutlich mehr Ressourcen. Bei umfangreicheren Anwendungen (insbesondere bei mehreren Microservicen) ergeben sich durch ein BFF jedoch auch einige Vorteile, wie die Wiederverwendbarkeit von Microservicen und die bessere Skalierbarkeit durch mehr und/oder mehrerer Microservices.

2.3.2.2 Unterschiede Backend BFF

Frage: *Was sind die hauptsächlichen Unterschiede zwischen dem TLM Microservice und dem BFF?*
Die API der beiden Services unterscheidet sich dadurch, dass der TLM-Microservice die CRUD-Operationen seiner Domänenobjekte anbietet, während das BFF auf das Frontend abgestimmte API-Endpunkte enthält. Beispielsweise ein GET für eine TodoListe mit all ihren Todos zusammen, eine Operation, die der TLM-Microservice so nicht anbietet. Außerdem beinhaltet das BFF noch die Applikationslogik, welche auch auf das Frontend angepasst ist, hier beispielsweise das Färben der WASA-Todolisten. Außerdem beinhaltet das Backend keinerlei Applikationslogik, sondern nur die spezifische Domänenlogik. Im Gegensatz dazu beinhaltet das BFF die Applikationslogik, welche in komplexeren Szenarien auch mehrere Domänen umfassen kann. Die Microservices für die verwendeten Domänen werden außerdem vom BFF orchestriert, da das BFF auf deren Domänenlogik aufbaut.

2.3.2.3 IAM

Frage: *Die Aufgaben beim Backend fügten einen IAM Service hinzu. Wie würde dieser das BFF beinflussen?*
In diesem Fall würde das BFF nun zwei Microservices orchestrieren statt nur einen. Beispielsweise hätte das Frontend noch Funktionien zum Ein- und Ausloggen eines Benutzers, Operationen,

die vom BFF direkt an den IAM-Microservice weitergeleitet werden würden, nicht an den TLM-Microservice. Außerdem ergibt sich eine neue Applikationslogik, in welcher die Berechtigungen zum Sehen und Speichern von Todos und Todolisten im TLM-Microservice über den IAM-Microservice überprüft werden würden. Beispielsweise wird aktuell der Frontend Request für alle TodoListen nur an den TLM-Microservice weitergeleitet. Mit IAM würde das BFF jedoch die TodoListen des TLM-Microservices anhand des aktuell angemeldeten Benutzers aus dem IAM-Microservice filtern. Dieser Request benötigt also beide Backend Microservices sowie Applikationslogik im BFF statt wie zuvor reiner Domänenlogik.

2.3.3 Aufgaben zum Frontend

Das Frontend macht das TodoListManagemant erst richtig benutzbar und ist das, womit der Benutzer interagiert. In diesem Abschnitt geht es um die im Frontend verwendeten Technologien und deren Implementierungen.

2.3.3.1 Technologien

Frage: *Was sind die Beziehungen zwischen den Technologien (Werkzeuge, Frameworks, Programmiersprachen), die in der Implementierung des Frontends genutzt werden?*
In Abbildung 2.7 kann man die im Frontend verwendeten Programmiersprachen sehen und wie diese zusammenarbeiten, um die dem Nutzer sichtbare Weboberfläche zu erstellen beziehungsweise für welche Teile der Webseite die einzelnen Technologien verantwortlich sind.
Bei diesem auf das Framework *Angular* aufbauende Frontend steht dabei TypeScript, die auf JavaScript aufbauende Programmiersprache, welche im Browser des Benutzers ausgeführt wird, im Mittelpunkt. Insbesondere übernimmt es das Routing (siehe `src/app/app-routing.module.ts`) auf die verschiedenen definierten Komponenten des Frontneds. Diese repräsentieren die unterschiedlichen zu sehenden Seitentypen, beispielsweise das Dashboard, die TodoListen-Ansicht und die Todo-Detailansicht. Außerdem ist der TypeScript-Teil des Frontends dafür verantwortlich, die für die aufgerufenen Seiten benötigten Daten via der REST-API des BFF aus dem Backend zu laden.
HTML und CSS sind für die tatsächliche Darstellung der Webseite verantwortlich und werden von dem vom Benutzer verwendeten Browser interpretiert. Bei der TLM-Demo wird dabei auf Bootstrap, ein CSS Framework gesetzt, welche eine einfach umzusetzende, einheitliche Designsprache ermöglicht.

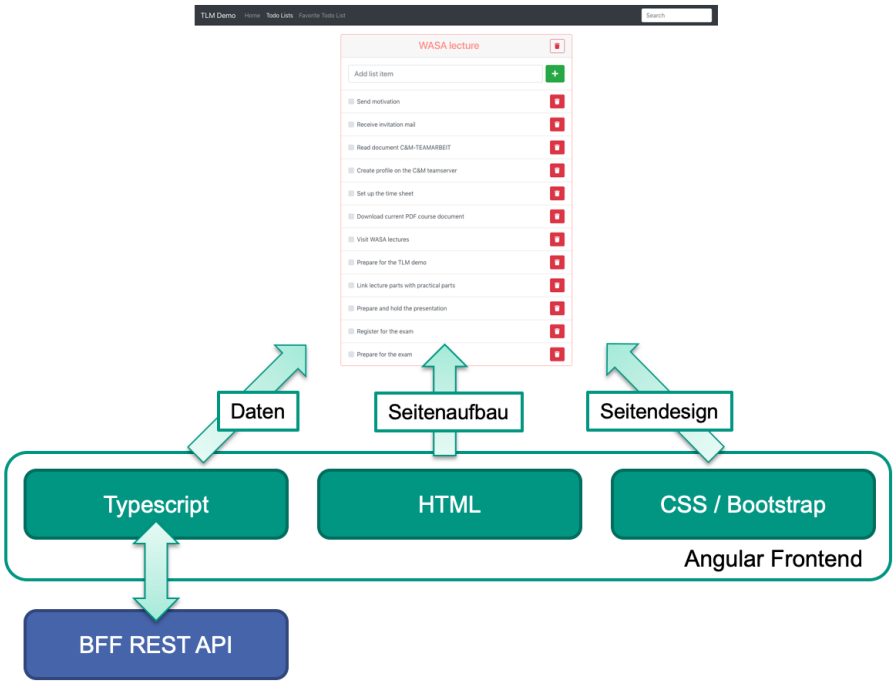


Abbildung 2.7: Technologien im TLM Frontend

2.3.3.2 Service Requests

Frage: *Wähle eine der Service-Requests und beschreibe seine Nutzung im Service und dem Aufruf der Komponenten.*

Wir betrachten den Service-Request zum Hinzufügen eines Todos zu einer bestehenden Liste, was hier bedeutet, dass wir einfach ein neues Todo-Objekt mit den richtigen Attributen erstellen. Den Code dieses Service Requests ist in Abbildung 2.8 zu sehen.

Wir sehen, dass der `POST /ID/todos` Endpunkt der REST-API des BFFs genutzt wird, `ID` steht

```
163   /**
164    * Adds a @param todo to the list @param id
165    * The todo must contain a color and the content
166    * @param id - the id of the todo list
167    * @param todo - the todo that should be added
168    */
169   addTodoToList(id: number, todo: Todo): Observable<Todo> {
170     const url = `${this.todoListUrl}/${id}/todos`;
171     return this.http.post<Todo>(url, todo, httpOptions).pipe(
172       tap( next: (todo: Todo) =>
173         this.log(`added todo with number=${todo.number} to todo list id=${id}`)
174       ),
175       catchError(this.handleError<Todo>( operation: 'addTodoToList'))
176     );
177   }
```

Abbildung 2.8: TLM-Service Request zum Erstellen von Todos

dabei für die ID der TodoListe zu der das neue Todo hinzugefügt wird. Dieser Service Request wird

nur einmal im Frontend verwendet und zwar im TodoList-Komponenten, also in der Weboberfläche auf der TodoListen-Übersichtsseite auf der auch alle bereits bestehenden Todos einer Liste angezeigt werden. In der Oberfläche ist hier eine Textbox, in welcher der Text eines neuen Todos eingetragen werden soll, und ein grüner Haken als Button zu sehen, der das neue Todo anlegen soll. Wird dieser Button ausgelöst, wird in der TodoList-Komponente die addTodo()-Methode aufgerufen, welche nach einer Validierung die Daten des neuen Todos an den beschriebenen Service Request schickt, welcher diesen an das BFF schickt und damit das Todo auch im Backend anlegt.

2.3.3.3 Fehlendes

Frage: *Was fehlt dem Frontend?*
Es sind einige Funktionen des Frontends entweder in der Oberfläche oder im Code angedeutet, jedoch nicht implementiert, beispielsweise gibt es bereits Andeutungen eines Identity und Access Managements, erkennbar an der `login`-Komponente und dem `user`-Model. Jedoch wird die Komponente noch nirgends eingebunden und das Model nicht verwendet. Zusätzlich gibt es noch keinen entsprechenden Service und auch das BFF unterstützt dies nicht. Außerdem fehlt dem Frontend noch die Möglichkeit TodoListen nach Erstellung zu bearbeiten, inbesondere Titel und Farbe. Zugleich ist unklar wie die favorisierte TodoListe verwaltet wird, da es keine Möglichkeit gibt, eine andere TodoListe als Favorit zu markieren.

2.3.3.4 Zusatz

Aufgabe: *Füge weitere UI Elemente auf Basis bereits bestehender Funktionalität hinzu. Das BFF sollte die Funktionalität bereits beinhalten. Beispielsweise dem Benutzer erlauben, bereits auf der TodoListen-Übersicht TodoListen zu löschen.*
Wir fügen die vorgeschlagenen Funktionalität hinzu, das Löschen von TodoListen auf der TodoListen-Übersicht. Zunächst schauen wir uns den Code an, mit dem TodoListen in der TodoListen-Komponente gelöscht werden können, siehe Abbildung 2.1.

```
1 ## HTML:
2 <button type="button" class="btn btn-outline-danger float-right" (click
      )="deleteList()">
3     <i class="fas fa-trash"></i>
4 </button>
5
6 ## TypeScript:
7 deleteList(): void {
```

```
8       this.todoService.deleteTodoListSynchron(this.todoList.id).then(()
            => {
9         this.router.navigate(['/todo-lists']);
10      });
11    }
```

Quelltext 2.1: Löschen von TodoListen auf der TodoListen-Seite

Also fügen wir entsprechende Codezeilen zur Dashboard-Komponente hinzu, siehe Abbildung 2.9. Dabei zeigt der im HTML angelegte neue Button den von der TodoListen-Seite bekannten Mülleimer

```
diff --git a/src/app/dashboard/dashboard.component.html b/src/app/dashboard/dashboard.component.html
index e8da030..ffe8e34 100644
--- a/src/app/dashboard/dashboard.component.html
+++ b/src/app/dashboard/dashboard.component.html
@@ -31,6 +31,9 @@
         <button type="button" class="btn todo py-3" routerLink="/todo-lists/{{todoList.id}}"
             [ngStyle]="{'background-color': todoList.color, 'border-color': todoList.color}">
             {{todoList.title}}
+            <button type="button" class="btn btn-outline-danger float-right" (click)="deleteList(todoList.id, $event)">
+                <i class="fas fa-trash"></i>
+            </button>
         </button>
     </div>
 </div>
diff --git a/src/app/dashboard/dashboard.component.ts b/src/app/dashboard/dashboard.component.ts
index e3b5295..13a3006 100644
--- a/src/app/dashboard/dashboard.component.ts
+++ b/src/app/dashboard/dashboard.component.ts
@@ -51,4 +51,9 @@ export class DashboardComponent implements OnInit {
   setColor(color: Color) {
     this.color = color.code;
   }
+
+  deleteList(id, event): void {
+    this.todoService.deleteTodoList(id).subscribe( todo => (this.getTodoLists()));
+    event.stopPropagation();
+  }
 }
```

Abbildung 2.9: Änderungen an der Dashboard-Komponente

und ruft die `deleteList(id, event)` Methode aus dem TypeScript-Teil auf. `$event` ist eine Variable von Angular, welche Informationen über das Klick-Event auf den Button enthält und welches wir nutzen können, um zu verhindern, dass uns der TodoListen-Button (in welchem sich unser kleinerer Button befindet) uns zusätzlich zur Löschaktion noch auf die Seite der TodoListe weiterleitet. Um die Liste zu löschen, müssen wir nur noch den Todo-Service entsprechend aufrufen. Darauf aktualisieren wir noch die gezeigten TodoListen, damit unsere gelöschte Liste nicht mehr angezeigt wird.

2.4 TLM Workshop

Im TLM Workshop ging es darum, die uns mittlerweile recht vertraute TLM Demo zu erweitern. Die erste Aufgabe dabei war, das Löschen von TodoListen auf dem Dashboard zu ermöglichen. Da dies jedoch bereits als Beispiel für eine Erweiterung im Aufgabendokument stand, war dies bereits

erledigt.
Wir schauen uns daher die zweite Aufgabe an, bei der es um das Erweitern von Todos um ein weiteres Attribut geht, hier ein Deadline String. Dieses Attribut ist, so wie die Beschreibung von Todos, eine Zeichenkette, die auch leer sein darf. Daher können einfach alle Microservices nach *description* durchsucht werden und die entsprechenden Teile kopiert werden, wobei *description* durch *deadline* ersetzt wird. Das klappt in fast allen Fällen, jedoch wollen wir im Frontend ein Textfeld statt eines Textbereichs für unser neues Attribut. Einige, jedoch nicht alle, der nötigen Änderungen an den drei Microservices kann man in Listing 2.2 sehen, das Ergebnis in Abbildung 2.10.

```
1  ## Frontend (src/app/todo-detail/todo-detail.component.html):
2  <div class="form-group row">
3      <label for="todoDeadline" class="col-sm-2 col-form-label">Deadline<
           /label>
4      <div class="col-sm-10">
5          <input type="text" class="form-control" id="todoDeadline" [(
               ngModel)]="todo.deadline" name="todoDeadline" placeholder="
               Enter todo deadline">
6      </div>
7  </div>
8
9  ## Frontend (src/app/todo.ts):
10 deadline: string;
11
12 ## Backend (bspw. /src/main/java/edu/kit/tm/cm/tlm/todomanagement/
       application/dtos/request/PatchTodoRequest.java):
13 @ApiModelProperty(position = 5)
14 Optional<String> deadline;
```

Quelltext 2.2: Ergänzen eines Deadline-Attributs für Todos

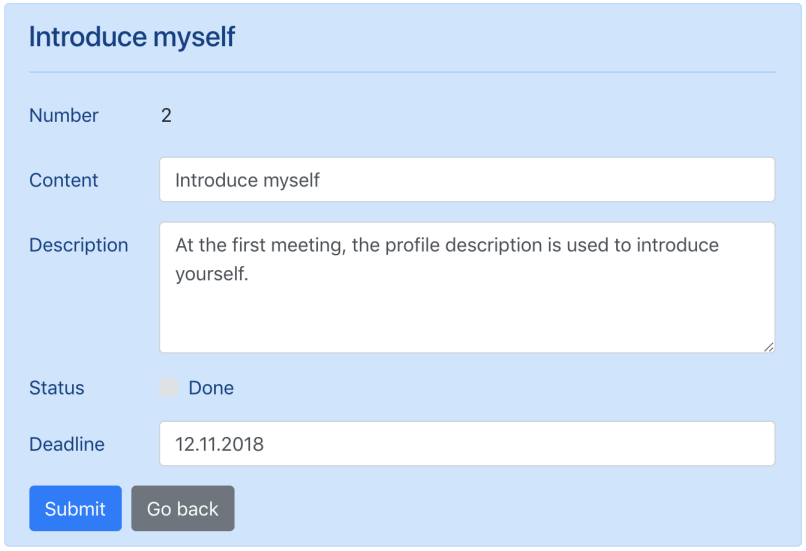


Abbildung 2.10: Todos mit Deadline Attribut

3 Projektspezifische Aufgabe

Im Projektteam *Pen_iCC* geht es darum, Web-Anwendungen automatisch mit aus neuronalen Netzen generierten Penetration-Tests verlässlich auf Sicherheitslücken zu testen. Im Folgenden wollen wir uns zunächst in die verwendeten Konzepte und Technologien einarbeiten, bevor wir diese in einer Build-Pipeline zusammenführen und orchestrieren.

3.1 Grundlagen

Zunächst schauen wir uns einige grundlegende Konzepte an, bevor wir die bereits bestehenden Systeme genauer inspizieren.

3.1.1 IT-Sicherheit

Um in IT-Systemen von Sicherheit sprechen zu können, müssen die in [Mö18a] aufgelisteten Schutzziele Confidentiality, Integrity und Availability, die sogenannte *CIA-Triade*, erfüllt sein. Confidentiality beschreibt dabei den Schutz von Daten vor dem Zugriff durch Dritte. Wobei Integrity den Schutz vor der unbefugten Änderung von Daten durch Dritte meint. Availability meint den Schutz vor Ausfällen des IT-Systems.
Unternehmen sind sehr daran interessiert, ihre Systeme zu schützen, da erfolgreiche Cyberangriffe großen Schaden anrichten können. Beispielsweise im Jahr 2014 über 800 Millionen US-Dollar nach [PB16] im Jahr. Eine wirksame Methode zum Schutz der Systeme kann unter anderem Penetration-Testing sein.

3.1.2 Penetration-Testing

Penetration-Testing beschreibt den Vorgang bei dem versucht wird, Sicherheitsvorkehrungen von IT-Systemen zu umgehen mit dem Ziel Sicherheitslücken zu finden. Dabei werden meistens verschiedene Attacken auf ein System simuliert und ähneln dem Vorgehen von Angreifern. Dieser kontrollierte und organisierte Weg Sicherheitsprobleme aufzudecken ist meist recht teuer für Unternehmen, da die Tests von hochqualifizierten Sicherheitsexperten durchgeführt werden. [PB16]
Da das Vorgehen beim Penetrationen-Testing zunächst nicht von einem illegalen Angriff zu unterscheiden ist, ist es wichtig, dass sich der Penetration-Tester mit seinem Auftraggeber Vorgehen und

Umfang der Tests vorher schriftlich festhält, um sich nicht strafbar zu machen. [PB16] empfielt, dass diese "Rules Of Engagement" unter anderem eine "Permission to Hack" und eine Verschwiegenheitserklärung enthalten sollen. Außerdem sollen die erlaubten Vorgehensweisen (bspw. Denial-of-Service erlaubt?), die Zeitdauer, die Ziele und die Verantwortlichkeiten vorher dokumentiert werden.

3.1.2.1 Arten des Penetration-Testing

Zunächst muss zwischen White-Box, Gray-Box und Black-Box-Testing unterschieden werden. Diese Klassifizierung gibt darüber Auskunft, wie viel der Penetration-Tester über das zu testende System weiß. Beim White-Box-Testing hat der Tester Zugriff auf das gesamte System, inklusive Quellcode und Logs. Gray-Box-Testing schränkt den Zugriff auf nur noch die Logs ein und beim Black-Box-Testing hat der Tester keinerlei Informationen über das System. Obwohl White-Box-Testing durch Code-Kenntnis ausgeklügeltere Angriffe erlaubt, wird meist Black-Box-Testing verwendet, da es dem Vorgehen echter Angreifer am nächstem kommt. [Mö18a]
Eine weitere Unterscheidung von Penetration-Tests kann man anhand dessen machen, ob die Tests manuell vom Penetration-Tester ausgeführt werden oder automatische Test-Skripte verwendet werden. Automatische Tests erlauben es, deutlich mehr potenzielle Sicherheitslücken in kurzer Zeit zu überprüfen und können so die schwerwiegensten Sicherheitsprobleme aufdecken. Außerdem haben diese den Vorteil, dass sich die Tests einfach wiederholen lassen und meist plattformunabhängig geschrieben und aktualisiert werden. Manuelle Tests haben jedoch dafür die Chance, aufgrund der hohen Qualifikation der Tester, Sicherheitslücken zu finden, welche automatische Tests nicht abdecken würden, da die Tests angepasst werden können und die Tester kreativer sein können. Nur beim manuellen Penetration-Testing kann man Zero-Day-Exploits (also bisher noch unbekannte Sicherheitslücken) aufdecken. Für einen vollständigen Vergleich von manuellem und automatisierten Testen siehe auch Abbildung 3.1. [PB16]

	Manual	Automated
Testing Process	Manual, non-standard process; Labor and capital intensive; High cost of customization;	Fast, standard process; Easily repeatable tests;
Vulnerability/ Attack Database Management	Maintenance of database is manual; Need to rely on public database; Need re-write attack code for functioning across different platforms;	Attack database is maintained and updated; Attack codes are written for a variety of platforms;
Reporting	Requires collecting the data manually;	Reports are automated and customized;
Cleanup	The tester has to manually undo the changes to the system every time vulnerabilities found;	Automated testing products offer clean-up solutions;
Training	Testers need to learn non-standard ways of testing; Training can be customized and is time consuming.	Training for automated tools is easier than manual testing.

Abbildung 3.1: Vergleich von manuellem und automatisiertem Penetration-Testing [PB16]

3.1.2.2 Angriffsmöglichkeiten bei Web-Anwendungen

Bevor wir selber Penetration-Tests für Web-Anwendungen generieren, schauen wir uns an, welche Möglichkeiten wir haben, um Angriffe durchzuführen. Dabei wollen wir auf die Web-Anwendung in einer Weise einwirken, in welcher sie von den Entwicklern nicht vorhergesehen wurde.
Dabei können folgende Daten der HTTP-Anfragen an die Web-Anwendung manipuliert werden (nach [Mö18a]):

- HTTP-Methode (GET, POST, PUT, PATCH oder DELETE)
- Cookies
- weitere HTTP-Header
- Formulardaten

Insbesondere bei Cookies und Formulardaten besteht die Gefahr, dass Eingaben des Benutzers nicht ordentlich maskiert werden und dadurch enthaltener Code auf dem Server (Cookie- und SQL-Injection) oder auf anderen Clients (Cross-Site-Scripting (XSS)) ausgeführt wird.
Eine weitere Angriffsmöglichkeit besteht in Denial-of-Service (DoS) Angriffen. Bei diesen werden unverhältnismäßig viele Anfragen an die Anwendungen schickt, welche diese überlasten. Diese Art des Angriffs richtet sich meist gegen das Schutzziel der Availability.

3.1.3 Machine Learning

Im Mittelpunkt unserer Anwendung sind die neuronalen Netze, der Neural Payload Generator und der Neural Evaluator. Diese sollen bei uns weitestgehend die Rolle des Penetrationtesters übernehmen und so die Automatisierung ermöglichen. Daher schauen wir uns kurz einige Grundlagen zum maschinellen Lernen an.
Neuronale Netze sollen das menschliche Gehirn mit vielen schwachen Recheneinheiten (Neuronen) nachempfinden. Die Gewichtungen der Verbindungen zwischen den Neuronen kann dabei trainiert werden, was das eigentliche "Lernen"des machinellen Lernens darstellt. Um ein neuronales Netz zu trainieren, werden ihm Eingabe- und Ausgabewerte gegeben. Umso mehr Daten, desto besser wird das neuronale Netz. Nach dem Training kann zu einem Eingabewert ein vermuteter Ausgabewert vom neuronalen Netz angegeben werden.
Wir nutzen Machine Learning, um automatisierte Penetrationtest zu generieren und zu evaluieren. Dazu haben wir zwei neuronale Netze, einen Payload Generator und einen Evaluator. Der Payload Generator ist dabei dem Black-Box-Testing zuzuordnen. Er nutzt die Daten aus sehr vielen erfolgreichen und erfolglosen Angriffen, um neue Angriffe zu generieren, welche eine hohe Erfolgschance haben. So können Vorteile des manuellen und automatisierten Testen kombiniert werden, da die Tests

ohne großen Aufwand eines Penetration-Testers ausgeführt werden können, jedoch durch die "Kreativität"des neuronalen Netzes auch neue Sicherheitslücken gefunden werden können. Der Evaluator ist dem White-Box-Testing zuzuordnen, da er Zugriff auf Quellcode und Logs des Zielsystems hat. Seine Aufgabe ist es, zu bestimmen, ob vom Payload Generator generierte Angriffe erfolgreich waren und, falls möglich, die exakte Sicherheitslücke im Quelltext zu finden. Der Evaluator ist damit das System, welches das Feedback für den Payload Generator als auch für den System-Entwickler über erfolgreiche Angriffe auf das System bereitstellt.

3.1.4 Continuous Integration (CI) / Continuous Delivery (CD) mit Jenkins

CI/CD steht für Continuous Integration und Continuous Delivery, also kontinuierliches Testen und Verteilen. Das Ziel dahinter ist es, dass Softwareentwicklung dynamischer wird und die Zeitdauer zwischen Anforderungen und Ausrollen von neuen Features nur noch wenige Wochen dauert. Damit das möglich ist, wird versucht, einzelne Teile des Entwicklungsprozesses zu automatisieren. Außerdem kann es beim Bauen und Testen von Projekten zu Problemen kommen, wenn mehrere Personen daran arbeiten. Hier kommt CI/CD ins Spiel, welches sich oft direkt an das Versionsverwaltungssystem (wie git) hängt und nach jedem gepushten Commit (oder alternativ immer nachts, sogenannte Nightlys) die Unit-Tests ausführt und falls diese erfolgreich sind, das Projekt baut. So erhalten die Entwickler möglichst schnell Feedback zu ihren Änderungen und können Bugs gegebenenfalls korrigieren. Außerdem steht so im Idealfall täglich eine neue Version bereit, die ausgeliefert werden könnte. Eine Applikation, welche CI/CD-Pipelines orchestrieren und ausführen kann, ist Jenkins. Jenkins kann beispielsweise lokal mit `docker run -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts` gestartet werden, siehe Abbildung 3.2.

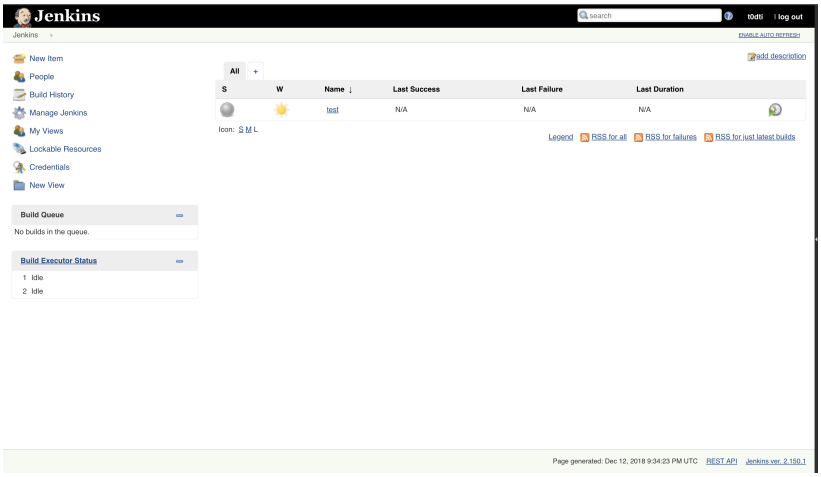


Abbildung 3.2: Jenkins Dashboard

3.1.5 ZAP und Selenium

ZAP (Zed Attack Proxy) und Selenium sind zwei Open-Source-Tools, die wir in unserer Build-Pipeline verwenden.
ZAP ist ein Sicherheitsscanner für Webseiten, der unterschiedlichste Funktionen hat insbesondere einen passiven und einen aktiven Scan. Beim passiven Scan funktioniert ZAP als Proxy und loggt den Traffic zwischen der Web-Anwendung und dem Benutzer. Diese Logs können daraufhin auf mögliche Sicherheitslücken entweder automatisch oder manuell durch einen Penetrationtester analysiert werden. Beim aktiven Scan geht ZAP ein Skript durch und checkt damit die Anwendung automatisch auf übliche Sicherheitslücken und nutzt auch eine Spider, die die Webseite crawlt. Da die Skripte des aktiven Scans nur primitive Sicherheitslücken finden, werden wir diesen in unserer Build-Pipeline nicht nutzen. Unsere Tests werden dann von neuronalen Netzen generiert und wir nutzen nur den passiven Scan von ZAP um unsere Tests zu dokumentieren und die Logs von neuronalen Netzen auswerten zu lassen.
Selenium ist ein Tool, um Browser zu automatisieren. Damit können beispielsweise Frontends getestet werden, indem Selenium-Skripte geschrieben werden. Wir benutzen Selenium in unserer Build-Pipeline, um die von den neuronalen Netz generierten Angriffe, in der Form von Payloads, auf die verschiedenen Seiten der Web-Anwendung auszuführen.

3.2 Projekt

In unserem Projekt geht es darum, vorhandene Tools und Anwendungen zusammenzubringen und zu automatisieren. Damit sollen in einer autonomen CI/CD-Pipeline von neuronalen Netzen generierte Sicherheitstests durchgeführt und evaluiert werden.

3.2.1 Feature

Um die Funktionalitäten des neuen Systems zu definieren, formulieren wir Gherkin-Features, welche später als User-Acceptance-Tests genutzt werden können. Dies entspricht dem Behaviour-Driven Development (BDD), welches im C&M-Entwicklungsprozess praktiziert wird. Die Features basieren dabei auf einem Language-Sketch (Abbildung 3.3), welcher die Capabilities und Use-Cases des Systems darstellt.

Im Language-Sketch sind die verschiedenen Personen, Systeme und Artefakte zu sehen, die in unserem System vorkommen. Insbesondere ist aber auch dargestellt, wie diese miteinander interagieren und welche Beziehung zwischen diesen bestehen. Im Mittelpunkt steht die CI/CD-Pipeline, die automatisch die Anwendung baut und mittels eines Test Executors testet, sowohl die Unit-Tests,

die von den Developern geschrieben wurden, als auch die Penetration Tests, die vom Penetration Tester generiert wurden. Eine weitere wichtige Rolle spielt das Notification System, dass auf die Test Results der Pipeline zurückgreift und Product Owner, Developer und Penetration Tester über erfolgreiche und fehlgeschlagene Tests informiert. Für alle Beziehungen und Interaktionen siehe Abbildung 3.3.

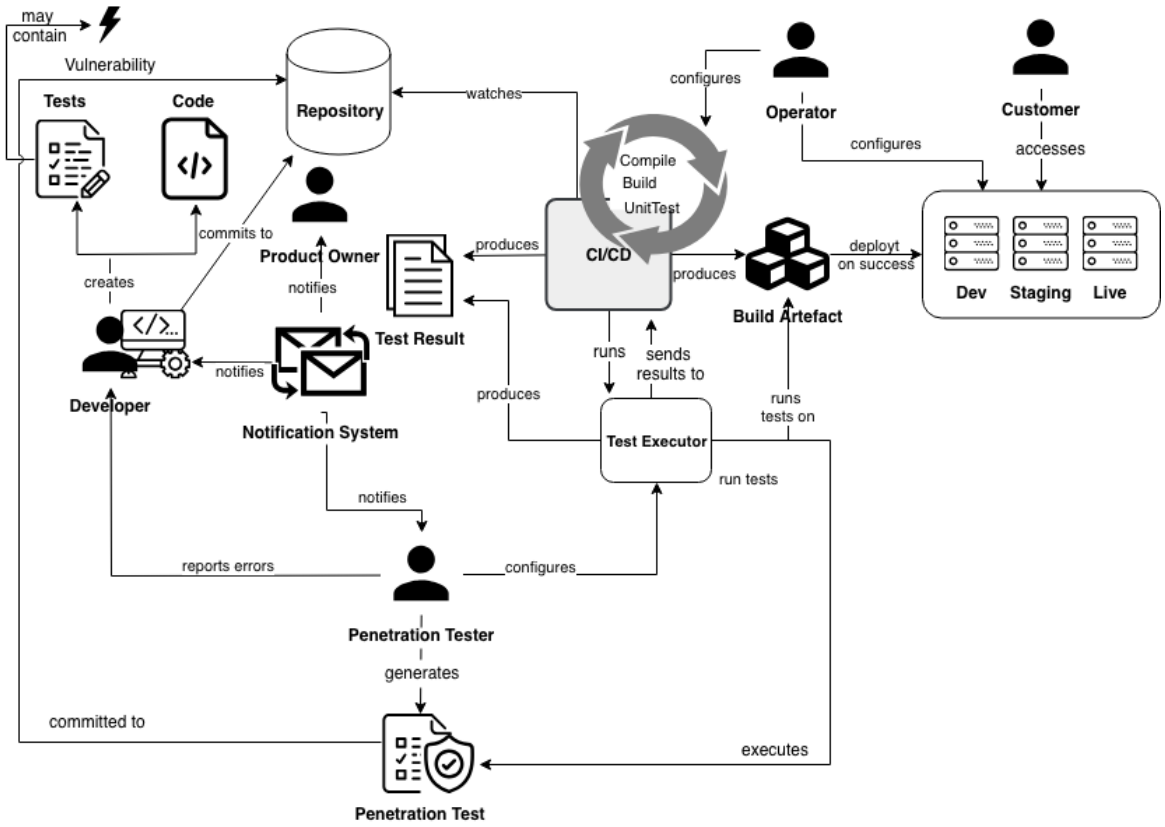


Abbildung 3.3: Language-Sketch

Ein mögliches Feature, dass die Funktionalität des Systems aus Sicht eines Entwicklers darstellt, wäre in Listing 3.1 zu sehen.

```
1 Feature: Automatic Notification About Test Results
2
3 As a Developer
4 I want my Code to be tested for functionality and security regularly
      and I get notified about problems
5 So that I can fix Bugs and Security Issues as soon as possible
6
7 Scenario: Automatic Test execution
8 Given I have modified the Code of a project
9   And I commited the Code changes to my local Repository
```

```
10 When  I push the changes to the remote Repository
11 Then  my changes get tested automatically
12
13 Scenario: Unit-Tests fail
14 Given I have pushed Code changes to the remote Repository
15   And the Code contains a bug that causes unit Tests to fail
16 When  the continuous integration runs the tests
17 Then  the bug gets discovered
18   And I am notified about the issue
19
20 Scenario: Security issue gets discovered
21 Given I am responsible for a project that contains an unknown Security
       Issue
22   And automatic penetration testing is enabled for the project
23 When  the automatic penetration testing runs
24   And the neural network runs a new attack that is successfull because
         of a Security Issue
25 Then  I am notified about it
```

Quelltext 3.1: BDD-Feature Developer

3.3 Neural Evaluator

Die beiden neuronalen Netze existieren bereits aus vorangegangen Semestern, insbesondere aus der Bachelorarbeit von Cedric Mössner [Mö18a]. Da wir diese nun verbessern, erweitern und in eine Build-Pipeline einbinden wollen, schauen wir diese bestehenden Systeme genauer an. Ich habe mich insbesondere mit dem Neural Evaluator beschäftigt. Quelle in diesem Kapitel ist, wenn nichts anderes angegeben, [Mö18a].

3.3.1 Ziele

Abbildung 3.4: Ablauf des Neural Evaluator

Damit Penetrationstests Sinn ergeben, muss natürlich nach einem Angriff auf das System auch überprüft werden, ob die Angriffe erfolgreich waren. Dies macht normalerweise, auch bei automatisierten Penetrationtests, der Penetrationtester manuell. Da in unserer Anwendung die Penetrationtests als Teil

einer Build-Pipeline komplett autonom laufen sollen, brauchen wir ein System, was diese Aufgabe übernehmen kann, dies ist der Neural Evaluator. Als Teil der automatischen Build-Pipeline, die nach von Entwicklern durchgeführten Änderungen durchläuft, soll neben beispielsweise Unit-Tests auch die Penetration-Tests laufen. In dem diese ausgewertet werden, kann dem Entwickler auch darüber Feedback gegeben werden, ob seine Änderungen eine neue Sicherheitslücke hinzugefügt haben, sodass ein automatisches Deployment der neuen Version noch gestoppt werden kann. Außerdem kann der Neural Evaluator als Feedback für den Payload Generator dienen, welcher immer neue Payloads generiert, mit dem Ziel erfolgreiche Angriffe auszuführen. Indem wir ihm Feedback geben, welche Payloads erfolgreich und welche nicht erfolgreich waren, können wir diesen weiter trainieren, um zukünftig noch bessere Payloads zu generieren.

Der Neural Evaluator ist der letzte Teil der Build-Pipeline und wertet die Ergebnisse der vorangegenen Phasen aus, siehe Abbildung 3.4. Dazu erhält das neuronale Netz zum einen die Antwort der Webseite, wenn kein Payload angegeben wird und daher auch kein Angriff vorlag. Sowie das vom Neural Payload Generator erstellte Payload, welches an die Webseite geschickt wurde und die darauffolgende Antwort der Webseite. Damit erhält das Netz alle beim Blackbox-Testing vorhandenen Daten. Mit diesen Daten berechnet das neuronale Netz dann eine Wahrscheinlichkeit, ob der der Angriff auf die Webseite erfolgreich war und gibt das wahrscheinlichere Ergebnis zurück.

3.3.2 Funktionsweise

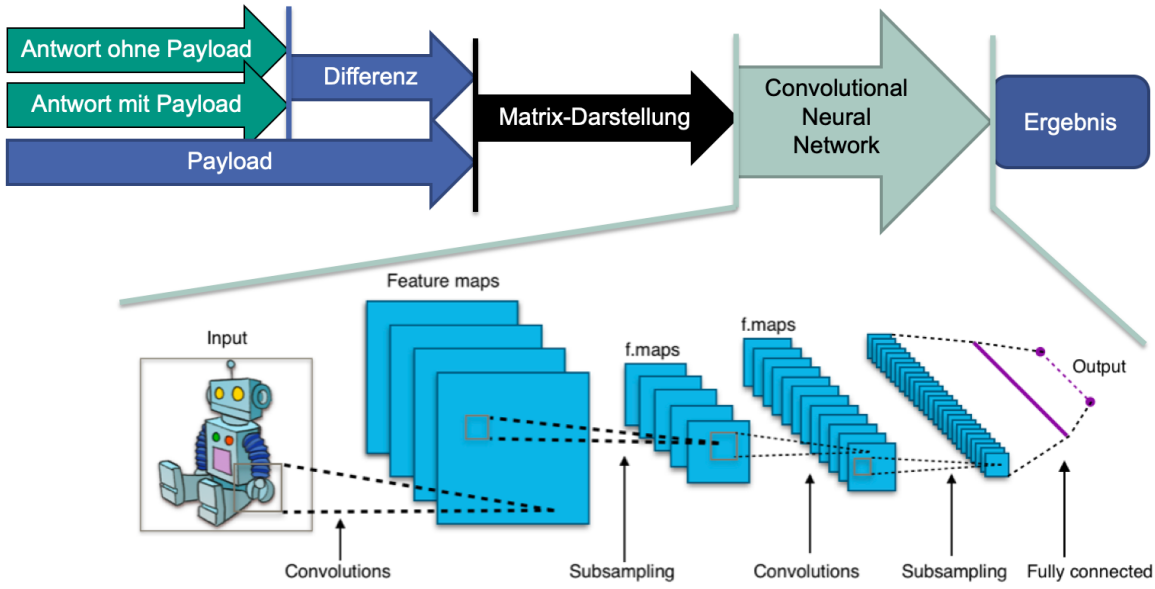


Abbildung 3.5: Funktionsweise des Neural Evaluator; CNN-Teil aus [Ap15]

Die Funktionsweise, wie die Wahrscheinlichkeit des Erfolgs des Angriffs berechnet wird, ist in Abbildung 3.5 zu sehen. Anstatt im neuronalen Netz jeweils die gesamten Antworten der Webseite (mit und ohne Payload) zu analysieren (welche unter Umständen sehr groß sein können), wird zunächst die Differenz zwischen diesen berechnet. Die gerade berechnete Differenz und der verwendete Payload werden als Matrix dargestellt, in dem einzelne Buchstaben als Vektoren und die "Wörter" als Liste der Buchstaben-Vektoren.

Daraufhin wird ein Convolutional Neural Network mit der Matrix als Eingabe gestartet, welches dann die Wahrscheinlichkeit eines erfolgreichen Angriffs als Zahl zwischen 0 und 1 als Ausgabe zurückgibt. CNNs bestehen aus mehreren Layern und werden häufig in der Bilderkennung eingesetzt. Die Hauptidee dabei ist es, besondere Merkmale bzw. Features zu erkennen und anhand dieser Schlüsse über das Gesamtbild zu ziehen. Dabei werden zunächst einzelne Teile der Matrix (Convolution) in Feature-Maps übertragen, bevor diese vereinfacht werden (Pooling). Dieser Schritt wird mehrfach wiederholt, sodass man große Menge an relativ kleinen Feature-Maps erhält, welche dann jeweils an ein Neuron weitergegeben werden. Diese Neuronen sind dann im Fully-Connected-Layer alle miteinander verbunden und bilden letztendlich das Ergebnis des Netzes.

3.3.3 API

Zur Benutzung des neuronalen Netzes stellt dieses eine API bereit, über die all seine Funktionen aufgerufen werden können. Da dies elementar beim Aufbau der Build-Pipeline ist, schauen wir uns diese genauer an. Quelle hierfür ist der Code des Neural Evaluator unter [Mö18b].

- `/sets/` Wird genutzt zur Verwaltung der Datensets zum Training des neuronalen Netz. Da wir in unserem Anwendungsfall immer von einem bereits trainierten Netz ausgehen, sind diese Endpunkte in der Pipeline irrelevant.
  - `/sets/addset/` Zum Hinzufügen eines Datensets
  - `/sets/getsets/` Listet alle aktuellen Datensätze
  - `/sets/deleteset/<filename>` Löscht den Datensatz <filename> von der Liste der Datensätze
- `/retrain/` Retrainiert das neuronale Netz mit den Datensätzen unter `/sets/`. Auch dieser Endpunkt wird in der Pipeline nicht benötigt.
- `/query/` Stellt den Endpunkt zur Evaluation eines Angriffes bereit. Eine Anfrage an diesen Endpunkt enthält entsprechend der Funktionsweise des Netzes:
  - `raw_website` Die Headers und der Body der Webseite ohne Angriff

- `attacked_webseite` Die Headers und der Body der angegriffenen Webseite
- `payload` Der verwendete Angriffsvektor

Und gibt zurück, ob der Angriff erfolgreich war, oder nicht. Dieser Endpunkt wird dabei in unserer Build-Pipeline genutzt, um die Angriffe zu evaluieren.

3.4 Pipeline

In der Pipeline wird nun unser gesammeltes Wissen genutzt, um die vorgestellten Tools zu verwenden und zu verbinden. Diese Pipeline soll später nach jedem Push einer Änderung an einer bestimmten Web-Anwendung diese automatisch auf Sicherheitslücken testen.

3.4.1 Aufbau

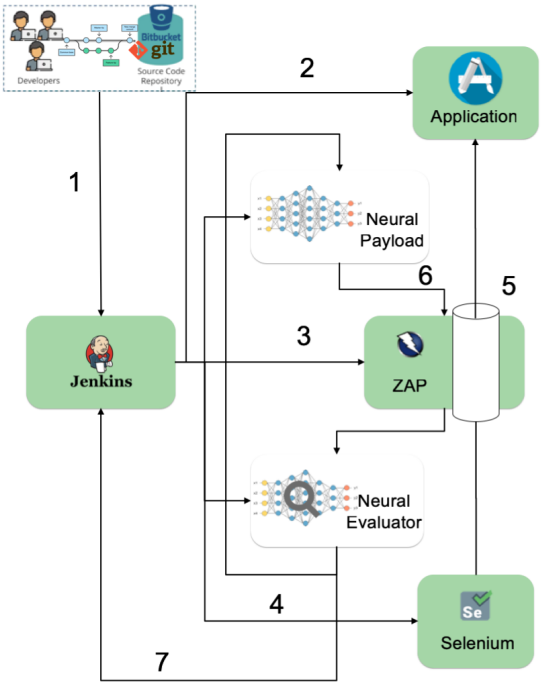


Abbildung 3.6: Aufbau der Build-Pipeline

Siehe Abbildung 3.6

1. Ein Push in das Repository der zu testenden Web-Anwendung stößt ein neue Pipeline in Jenkins an.

2. Jenkins baut daraufhin die neueste Version der Web-Anwendung in einen Docker-Container, sodass diese aufrufbar ist.
3. Zed Attack Proxy wird gestartet, um alle Aufrufe und Antworten der Web-Anwendung zu protokollieren.
4. Selenium wird gestartet und die Web-Anwendung darin geöffnet.
5. Der Traffic von Selenium zur Web-Anwendung geht über ZAP als Proxy.
6. Der Neural Payload Generator generiert Angriffe auf die Web-Anwendung.
7. Der Neural Evaluator wertet die Angriffe anhand der Reports von ZAP aus und schickt das Feedback zurück an Jenkins.

3.4.2 Infrastruktur

Da viele verschiedene Anwendung gleichzeitig laufen bietet sich natürlich ein Aufbau mit Mikroservices in Docker-Containern an ganz im Sinne der WASA-Vorlesung.
Dies erlaubt es uns einzelne Anwendungen der Pipeline beispielsweise zum Updaten austauschen oder auch automatisch und schnell die Anwendungen zu skalieren, um beispielsweise darauf zu reagieren, wie viele Pipelines gerade laufen. Außerdem können so auch leicht weitere Anwendungen zur Pipeline hinzugefügt werden, denkbar wären beispielsweise Unit-Tests oder Code-Inspections. Wenn man sich den Aufbau der Pipeline anschaut, sollte einem bewusst sein, dass es einige Abhängigkeiten zwischen den unterschiedlichen Anwendungen gibt, beispielsweise hängt alles von Jenkins ab, die neuronalen Netze von ZAP und Selenium und mehr. Wenn man nun auch noch auf die Abhängigkeiten der einzelnen Anwendugen achten müsste, beispielsweise unterschiedliche benötigte Java-Versionen und verschiedene Bibliotheken würde die Pipeline deutlich komplizierter werden. Da die Docker-Container alle ihre Abhängkeiten selbst beinhalten ist das jedoch kein Problem für uns. Außerdem erlauben Container die Umsetzung dieser Pipelines in allen möglichen Infrastrukturen. Sowohl in Cloud-Umgebugen als auch eigenen Rechenzentren. Aber auch zu Testzwecken auf dem eigenen Laptop.

Zukünftig wollen wir zur Orchestrierung der Pipeline Orchestrierungstools wie Docker Swarm oder Kubernetes nutzen. Diese übernehmen unter anderem die Ressourcenverwaltung und die automatische Skalierung der Container. Diese Tools erlauben es außerdem den gewünschten Stand der Container als Infrastrukturcode zu definieren und erlauben es so, unglaublich schnell komplette Umgebungen auf- und abzubauen und sind ein elementarer Bestandteil von Product-As-A-Service.

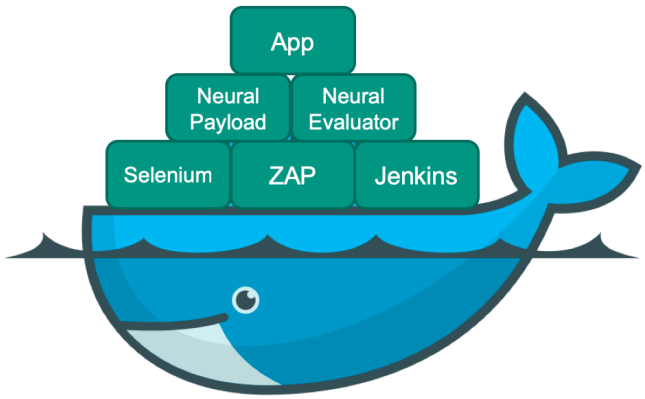


Abbildung 3.7: Die verschiedenen Anwendung der Pipeline als Docker-Container; Logo [DW18]

4 Zusammenfassung und Ausblick

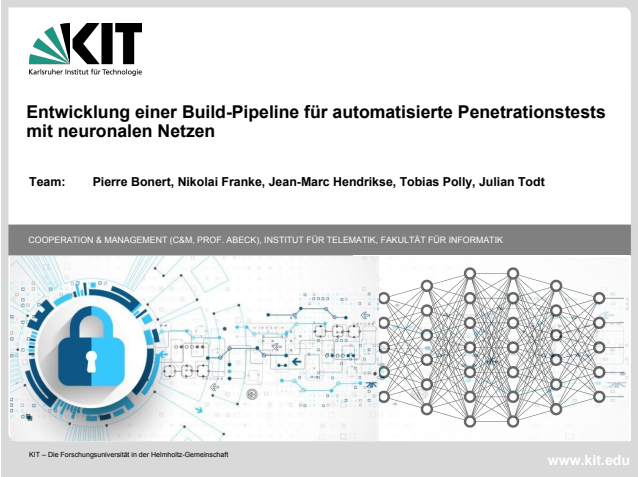
Während des Proseminars haben wir uns zunächst mit den Entwicklungstechniken bei C&M befasst, insbesondere BDD und DDD. Diese Techniken haben wir dann in der TLM-Demo in Aktion gesehen, ein Beispiel für eine Mikroservice-Webanwendung mit Backend, BFF und Frontend.

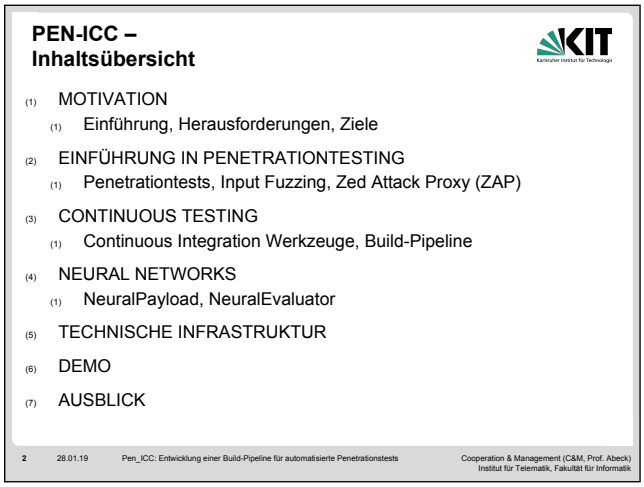
Im projektspezifischen Teil haben wir uns mit automatischen Penetrationtests mithilfe von neuronalen Netzen auseinandergesetzt und eine autonome Build-Pipeline dafür gebaut. Dazu haben wir uns zuerst theoretisch IT-Sicherheit und Penetrationtesting angeschaut bevor wir verschiedene Tools zum Durchführen von Penetrationetests getestet haben. Anschließend haben wir uns anhand eines Gherkin-Features überlegt, welche Funktionen das System von uns unter anderem schlussendlich haben soll. Für unsere Pipeline sind insbesondere auch die neuronalen Netze interessant, mit deren Funktionsweise und API wir uns danach beschäftigt haben. Die vorliegende Proseminararbeit hat sich überwiegend mit dem Neural Evaluator befasst, der auswertet, ob Angriffe auf Webanwendung erfolgreich waren und damit Sicherheitslücken in dieser vorhanden sind. Schlussendlich haben wir alle Anwendungen bis auf die neuronalen Netze zusammen in eine Build-Pipeline zusammengeführt, sodass dies alles automatisch ausgeführt werden kann.

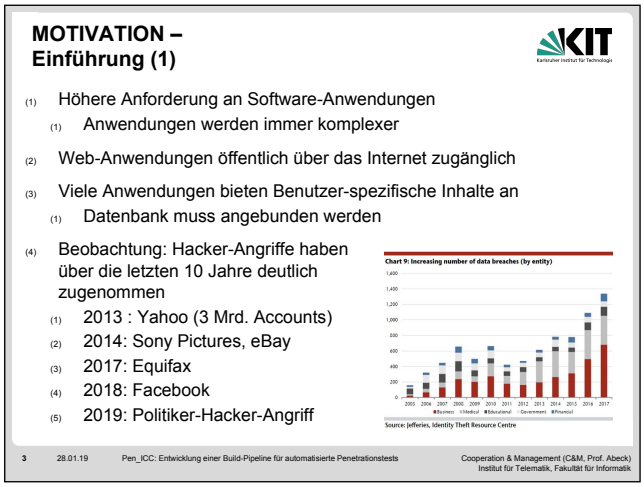
Zukünftig müssten die neuronalen Netze noch in die Build-Pipeline eingefügt werden. Dazu haben wir aber mit der Basis-Pipeline sowie der Herausarbeitung der API der neuronalen Netze gute Vorarbeit geleistet. Außerdem sollten die neuronalen Netze noch weiter optimiert und trainiert werden, um zukünftig bessere Ergebnisse beim Penetration-Testing zu erreichen, insbesondere neue Sicherheitslücken zu finden.

5 Anhang

5.1 Seiten der Abschlusspräsentation




PEN-ICC –
Inhaltsübersicht
(1) MOTIVATION
(1) Einführung, Herausforderungen, Ziele
(2) EINFÜHRUNG IN PENETRATIONTESTING
(1) Penetrationtests, Input Fuzzing, Zed Attack Proxy (ZAP)
(3) CONTINUOUS TESTING
(1) Continuous Integration Werkzeuge, Build-Pipeline
(4) NEURAL NETWORKS
(1) NeuralPayload, NeuralEvaluator
(5) TECHNISCHE INFRASTRUKTUR
(6) DEMO
(7) AUSBLICK
2 28.01.19 Pen_ICC: Entwicklung einer Build-Pipeline für automatisierte Penetrationstests
Cooperation & Management (C&M, Prof. Abeck)
Institut für Telematik, Fakultät für Informatik

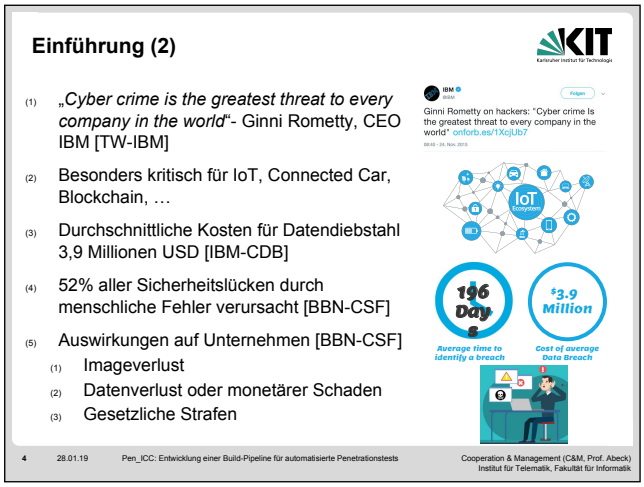


(1) Im Gegensatz zu den frühen Softwaresystemen mit nur einer sehr geringen Nutzerzahl sind die Systeme deutlich komplexer geworden. Wir bewegen uns heute mit kleinen kompakten Hochleistungs-Computern wie Tablets, Smartphones oder Smartwatches täglich rund um die Uhr online im Netz des digitalen Zeitalters und zeichnen dabei Unmengen von Daten auf.

(2) Da Webanwendungen in der Regel über eine Anbindung an das öffentliche Netz, dem World Wide Web (WWW), verfügen, sind diese besonders anfällig für Angriffe. Diese Angriffe versuchen unautorisiert über die Webanwendung auf dahinterliegende Systeme Zugriff zu erlangen und Schaden anzurichten. Aus diesem Grund gilt es Webanwendungen besonders vor unautorisierter Zugriffskontrolle zu schützen und Schwachstellen in der Software zu vermeiden, die das Risiko für einen Angriff erhöhen.

(3) Viele Web-Anwendungen bieten für ihre Benutzer speziell angepasste Inhalte, damit sie langfristig einen Mehrwert in der Anwendung entdecken. Dabei generieren sowohl Kunden als auch Unternehmen jede Menge Daten, die auf Datenbanken abgespeichert werden.

(4) Besonders in den letzten 10 Jahren hat die Anzahl an Angriffen und Sicherheitslücken deutlich zugenommen. Dabei sind nicht nur kleinere Unternehmen betroffen, sondern besonders durch das bekannt werden in den Medien wird das Bewusstsein der Benutzer für die Bedrohung deutlich.
(4.1) Im Jahr 2013 fand ein schwerer Datendiebstahl auf den Servern der Firma Yahoo statt, bei dem 3 Mrd. Accounts betroffen waren.
(4.2) 2014 stürzte eines der größten amerikanischen Filmkonzerne in seine schwerste Krise. Dabei ist der Datenklau ein gutes Beispiel dafür, was geschehen kann, wenn es ein Unternehmen im digitalen Zeitalter das Thema Datensicherheit nicht ernst nimmt. E-Mails mit sensiblen Daten wurden unverschlüsselt versendet, Passwörter im Klartext abgespeichert und Projekte wurde unverschlüsselt auf Servern abgelegt.
(4.3) Bei dem Angriff auf den amerikanischen Finanzdiesntleister Equifax waren bis zu 143 Millionen US-Bürger betroffen. Bei diesem Angriff wurden Sozialversicherungsnummern und Kreditkartennummern von den Angreifern erbeutet.
(4.4) Bei dem jüngsten Hackerangriff auf das Unternehmen Facebook wurden mehr als 50 Millionen Accounts, mit teils sehr sensiblen Daten, wie Orte, an denen sich die Benutzer in den vergangenen Wochen befanden, und Verknüpfungen auf Bildern, erbeutet.
(4.5) Der für Deutschland jüngste bekannte Angriff (Januar 2019) war ein gezielter Angriff auf Kontodaten, private Fotos, Briefe von Bundestagsabgeordneten und Privatadressen von Prominenten und Politiker.



(1) Der Satz von Ginni Rmetty, CEO von IBM, „Cyber crime is the greatest threat to every company in the world" zeigt, welche Bedeutung IT-Sicherheit zukommt. Dabei lässt sich dieser Satz weiter ausdehnen auf jede Person, jeden Ort und jedes „Ding", das am Internet hängt.

(2) Besonders kritisch sind Sicherheitsbedrohungen im Hinblick auf Internet of Things (IoT), Connected Car oder bei Smart Contracts der Blockchain-Technologie. Existiert eine Sicherheitslücke, ist diese nur sehr schwer im Nachgang zu schließen und Angriffe können verheerende Folgen nach sich ziehen.

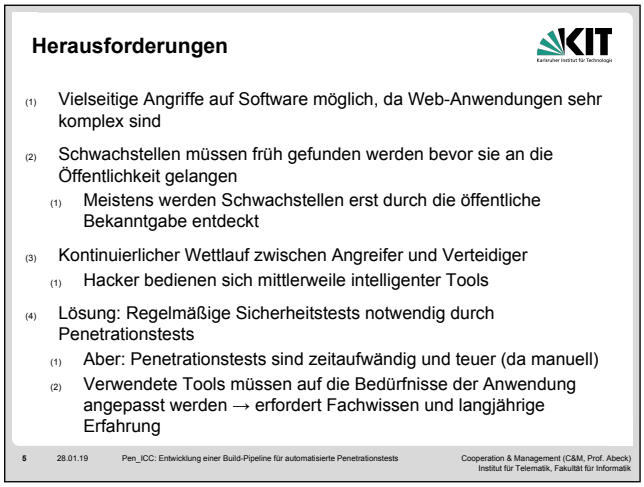
(3) Einer Studie zur Folge [IBM-CDB] sollen die Durchschnittskosten für einen Datenangriff 3,9 Millionen USD bezeichnen.

(4) Laut einem Bericht von [BBN-CSF] werden 48% der Sicherheitslücken durch böswillige Absichten verursacht. 52% der Sicherheitsbedrohungen entstehen durch menschliche Fehler oder Versagen der Systeme.

(5) Jede einzelne Sicherheitslücke in der Software erhöht sonst das Risiko für jeden Benutzer Opfer eines Angriffs zu werden. Ein aufgedeckter Angriff oder eine entdeckte Datenbanklücke kann zu einem existenziellen Schaden für das Unternehmen führen und hat in den meisten Fällen wirtschaftliche Folgen. Zusätzlich leidet das positive Image des Unternehmens, sodass das Vertrauen der Benutzer verloren gehen kann. Darüber hinaus gibt es bei nicht Einhaltung von Sicherheitsrichtlinien Sanktionen vom Gesetzgeber.

During the next five years, cybercrime might become the greatest threat to every person, place and thing in the world. With evolving technology comes evolving hackers, and we are behind in security. Understanding the cyber terminology, threats and opportunities is critical for every person in every business across all industries. By providing advanced cyber training and education solutions in all departments of your business, from marketing and sales to IT and InfoSec, you are investing in your company's protection against cyber threats.

[TW-IBM] Twitter IBM, https://twitter.com/ibm/status/669193863429509120, Stand 2019-01-22.
[IBM-CDB] IBM, Cost of a Data Breach Study, https://www.ibm.com/security/data-breach, Stand 2019-01-22.
[BBN-CSF] BBN-Times, Cyber Security Facts 2018,
https://www.bbntimes.com/en/technology/cyber-security-facts-2018, Stand 2019-01-22.



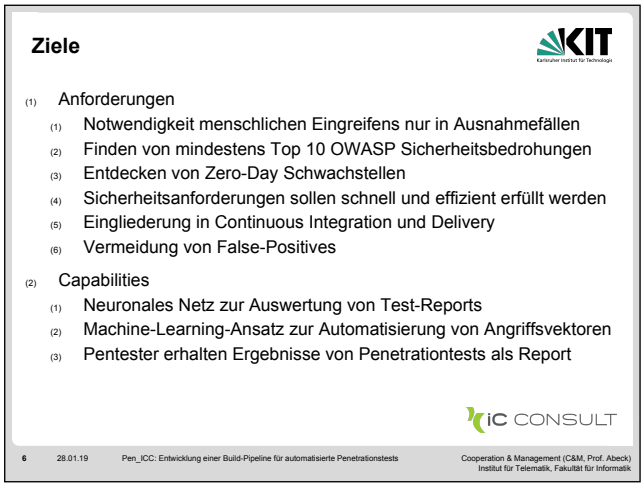
(1) Da Web-Anwendungen immer komplexer werden und vielseitigen Input entgegennehmen können, sind dementsprechend auch die Angriffe vielfältig und komplex. So werden immer wieder neue Sicherheitslücken entdeckt [HS–SFH] [SO-AFF] .

(2) In den meisten Fällen werden Schwachstellen erst im eigenen Unternehmen erkannt, wenn es meistens schon zu spät ist, da Angreifer Sicherheitslücken ausgenutzt und Schaden verursacht haben. Das Beheben der Schwachstellen in der Software ist dann in aller Regel mit sehr hohen Kosten verbunden. Somit wird deutlich erkennbar, dass die Betrachtung von Sicherheit den gesamten Softwareentwicklungsprozess kontinuierlich durchlaufen muss und nicht erst am Ende bei der Auslieferung der Software, damit Sicherheit in der Anwendung zu einem hohen Maße garantiert werden kann [Mc06].
Jede einzelne Sicherheitslücke in der Software erhöht sonst das Risiko für jeden Benutzer Opfer eines Angriffs zu werden.

(3) Auch Angreifer haben den Mehrwert von künstlicher Intelligenz erkannt und wenden neue mächtigere Werkzeuge als bisher an, um in Systeme von Unternehmen zu gelangen. Damit wird der Angriff auf Computersysteme auf eine neue höhere Ebene gehoben. Es kommt zu einem Wettstreit zwischen Angreifer und Verteidiger.

(4) Im laufenden Prozess werden spezielle Informationssicherheitsexperten beauftragt, die IT-Systeme und Software aus Sicht eines Angreifers zu testen. Sogenannte Penetration-Tester versuchen dabei aktiv Sicherheitsbarrieren zu umgehen, um die Verwundbarkeit der Systeme oder Webanwendungen aufzuzeigen. Dabei verwendet der Penetration-Tester Werkzeuge, die ihn beim Eindringen unterstützen
sollen.
(4.1) Da traditionelles Penetration-Testing mit einem hohen Kosten- und Zeitaufwand verbunden ist, versucht man viele Prozesse im Penetration-Testing zu automatisieren. Diese Vorgehensweise wird zentral in diesem Projektthema betrachtet.

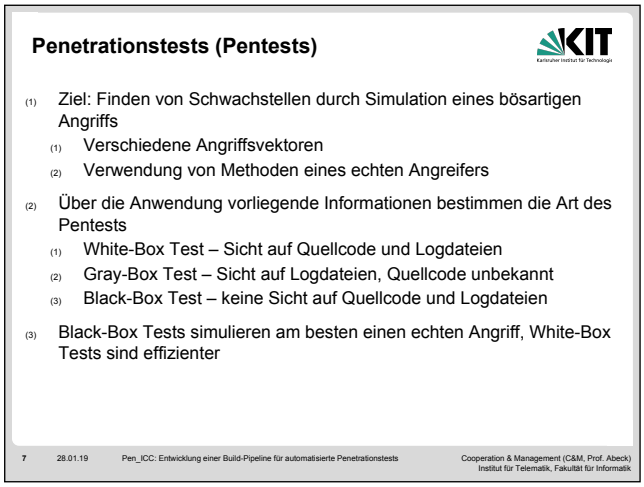
[HS–SFH], HeiseSecurtity, So funktioniert der Heartbleed-Exploit,
https://www.heise.de/security/artikel/So-funktioniert-der-Heartbleed-Exploit-2168010.html, Stand 2019-01-22
[SO-AFF] Spiegel-Online, Apples furchtbarer Fehler,
http://www.spiegel.de/netzwelt/web/goto-fail-apples-furchtbarer-fehler-a-955154.html, Stand 2019-01-22
[Mc06] MCGRAW, Gary: Software security: building security in. Bd. 1. Addison-Wesley Professional, 2006



(1) Da, wie bereits erwähnt, das manuelle Testen von Sicherheitslücken sehr zeitaufwändig und somit teuer ist, bietet sich hierfür eine automatisierte Lösung mithilfe von Machine-Learning-Verfahren an.
Somit ist die Kernanforderung, dass die Notwendigkeit für ein manuelles Eingreifen des Pentesters nur in Ausnahmefällen dank automatisierter und intelligenter Services basierend auf Neuronalen Netzen für Penetration-Testing notwendig sein soll.
Im Nachfolgenden sind die Business Goals aufgeführt, die es für die erfolgreiche Umsetzung des Continuous Penetration Testings zu erfüllen gilt:
(1.1) Reduzierung manueller Eingriffe durch Penetration Tester. Einschränkung auf besondere Fälle, die besonders komplexes Fachwissen benötigen.
(1.2) Finden von aktuellen Top 10 OWASP-Sicherheitsbedrohungen.
(1.3) Entdecken neuer/weiterer Sicherheitslücken.
(1.4) Sicherheitsanforderungen effizient und schnell erfüllen.
(1.5) Das Penetration-Testing soll sich in eine bestehende Pipeline aus Continuous Integration und Delivery (CICD) eingliedern.
(1.6) Da Standard-Penetrationsprogramme häufig viele False-Positives generieren sollen diese erkannt und aussortiert werden.

(2) Capabilities bieten dem Benutzer die Möglichkeit Aufgaben und Aktionen mithilfe der Software durchführen zu können. Darüberhinaus sollen sie zur Umsetzung der Business Goals dienen. Im Kontext unseres Continuous Penetration Testings ergeben sich die folgenden Capabilities:

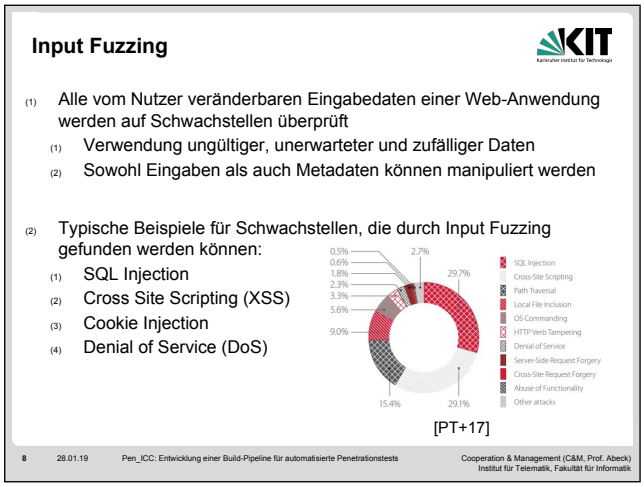
(2.1) Testfälle sollen vollständig automatisch generiert, durchgeführt und ausgewertet werden. Ein neuronales Netz soll dabei die Arbeit des Penetrationtesters übernehmen.
(2.3) Schwachstellen im Code sollen dem Penetration Tester aufgezeigt werden. Penetration Tester und/oder Entwickler aus dem Softwareentwicklungsteam werden auf Sicherhehitslücken hingewiesen und bekommen die entsprechenden Schwachstellen im Code geliefert.



(1) Bei einem Penetrationstest wird ein kontrollierter Angriff auf eine Anwendung durchgeführt um dessen Grad an Sicherheit zu testen und Schwachstellen frühzeitig zu finden.
(1.1) Um einen möglichst hohen Grad an Sicherheit gewährleisten zu können, sollten viele verschiedene Angriffsvektoren verwendet werden.

(2) Beim Pentesting wird zwischen drei verschiedenen Arten von Tests unterschieden, je nachdem wie viele Informationen der Pentester über die zu testende Anwendung erhält.
(2.1) Bei einem White Box Test stehen dem Pentester der Quellcode und die internen Details der Anwendung zur Verfügung.
(2.2) Bei einem Gray Box Test hat der Pentester keinen Zugriff auf den Quellcode. Er erhält jedoch mehr Informationen als ein gewöhnlicher Nutzer, wie z.B. Log-Nachrichten, einen groben Aufbau der Anwendung, oder verwendete Algorithmen.
(2.3) Ein Black Box Test wird aus der Sicht eines gewöhnlichen Nutzers der Anwendung durchgeführt. Der Pentester erhält also keine internen Details und muss sich alle Informationen über die Anwendung selbst beschaffen.

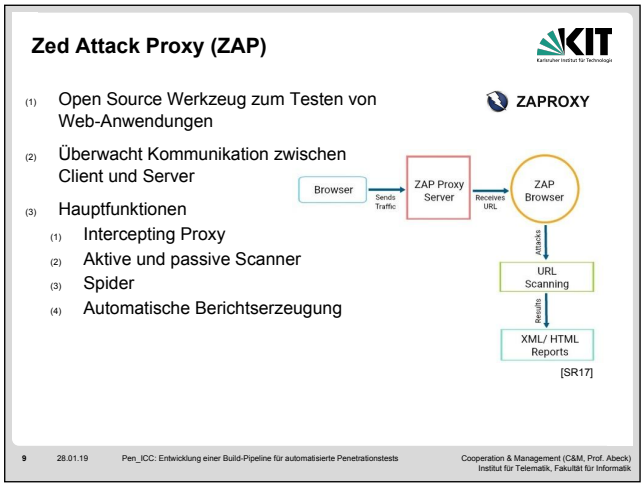
(3) Da ein Black Box Test auf Basis der gleichen Informationen, die einem potentiellen Angreifer zur Verfügung stehen, durchgeführt wird, simuliert dieser einen echten Angriff am besten.



(1) Die meisten Web-Anwendungen erlauben einem Nutzer Daten an den Server zu übergeben, zum Beispiel über Textfelder. Abhängig von der internen Struktur der Web-Anwendung und dessen verwendeten Technologien können diese Daten verwendet werden, um einen Angriff durchzuführen.

(2.1) Verwendet eine Web-Anwendung eine SQL- Datenbank, so können über nicht ausreichend überprüften Benutzereingaben SQL-Befehle eingeschleust werden. Diese werden dann von dem Server ausgeführt und ein Angreifer erhält Zugriff auf die Datenbank.
(2.2) Werden die Eingaben eines Nutzers in ein Eingabefeld nicht angemessen überprüft, kann so HTML Code in den Server eingeschleust werden. Dadurch können indirekt Skripte an den Browser anderer Nutzer gesendet werden, wo Schadcode auf der Seite des Clients ausgeführt wird.
(2.3) Cookies werden von Webanwendungen häufig verwendet um Informationen wie Login-Daten in Form von Tokens auf Nutzerseite zu speichern. Ein Angreifer kann schadhafte Cookies erzeugen oder vorhandene Cookies manipulieren um sich beispielsweise unautorisierte Administratorrechte zu verschaffen oder an sensible Informationen anderer Nutzer zu gelangen.
(2.4) Bei einem DoS-Angriff ist das Ziel, die Verfügbarkeit eines Services oder Teilen davon einzuschränken, sodass Nutzer nicht mehr darauf zugreifen können. Dies wird durch eine Überlastung des Servers erreicht, welche zum Beispiel durch Senden einer großen Anzahl von fehlerhaften HTTP-Anfragen ausgelöst werden kann.

[PT+17] Positive Technologies, „WEB APPLICATION ATTACK STATISTICS Q1 2017“, from https://www.ptsecurity.com/ww-en/analytics/web-application-attack-statistics-q1-2017/

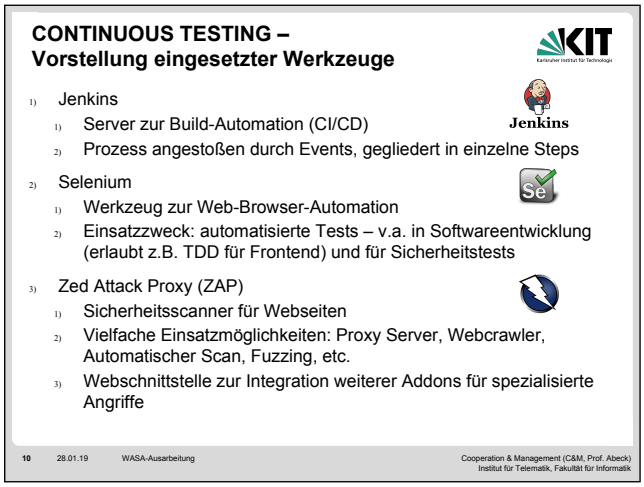


(1) Zap wird von der Open Web Application Security Project (OWASP) Foundation bereitgestellt.

(2) ZAP wird als Proxy eines Browsers eingerichtet und fängt alle Anfragen des Clients und die Antworten des Servers ab.

(3.1) Eine Intercepting Proxy fängt alle Anfragen des Browsers und die Antworten des Servers ab und dokumentiert diese. Außerdem können Anfragen manipuliert werden.
(3.2) Ein passiver Scanner beobachtet den Datenverkehr ohne selbst Anfragen an den Server zu senden. Ein aktiver Scanner versucht Schwachstellen zu finden, indem bekannte Angriffe auf das Ziel durchgeführt werden.
(3.3) Eine Spider (oder auch Web Crawler) durchsucht eine Website auf URLs und speichert diese. Danach werden rekursiv alle gefundenen Adressen aufgerufen und erneut auf URLs durchsucht.
(3.4) ZAP erzeugt automatisch Berichte über die gefundenen Sicherheitslücken und eignet sich somit gut für automatisiertes Testen.

[SR17] Srijan, „An Intro to OWASP Zed Attack Proxy“, from
https://www.srijan.net/blog/intro-owasp-zed-attack-proxy/



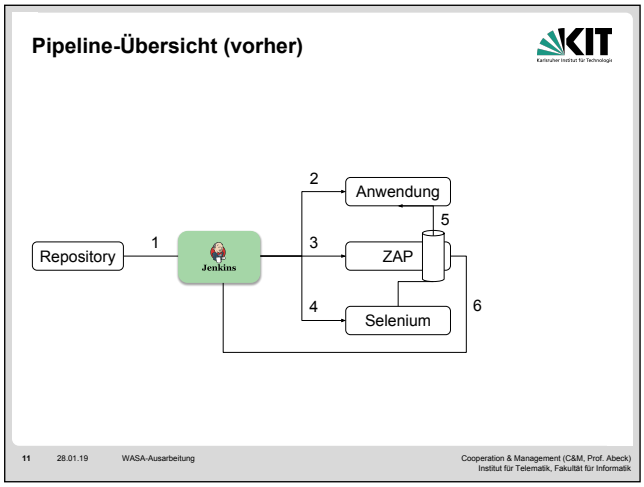
(1) Jenkins
(1.1) Weit verbreitetes System zur Build-Automatisierung – zentraler Bestandteil von Continuous Integration und Continuous Delivery
(1.2) Erlaubt Konfiguration verschiedenster Projekte, Sprachen, Frameworks und Build-Tools sowie Auslieferungsformate (z.B. Docker-Container)
(1.3) Konfiguration eines Projekts an Event geknüpft: Typischerweise sobald neuer Code im Master-Branch des Repositories ist
(1.4) Konfiguration in verschiedene Schritte (Steps) gegliedert: Bauen, Testen, Ausliefern – vergleichbar eines Skripts
(1.5) Hat eine Webschnittstelle zur Konfiguration und Überwachung

(2) Selenium
(2.1) Ist ein Werkzeug zur Browser-Interaktion: Erlaubt die programmatische Interaktion mit einem Webbrowser (fast alle gebräuchlichen Browser sind unterstützt)
(2.2) Einsatzgebiet ist automatisches Testen bei Softwareentwicklung – z.B. auch Regressionstests – sowie Sicherheitstests (es existieren dafür bereits bestehende Tests)
(2.3) Programmierung entweder durch Skripte (verschiedene Programmiersprachen und Testframeworks möglich) oder durch Aufzeichnen der eigenen Interaktion mittels Browser-Addon (Besonders gut für z.B. einen Bug-Report – Entwickler kann genaue Interaktion nachverfolgen und Fehler reproduzieren)

(3) Zed Attack Proxy (ZAP)
(3.1) Ist ein Sicherheitsscanner für Webseiten, um häufig auftretende Sicherheitsmängel automatisiert aufzufinden
(3.2) Vielfache Einsatzmöglichkeiten sind verfügbar:
(3.2.1) Proxy Server: Liest bestehende Verbindungen mit und kann Paketinhalte manipulieren, passiver Scan nach Sicherheitslücken
(3.2.2) Webcrawler: Durchsucht Webseiten und erstellt Indizes
(3.2.3) Automatischer Scan: Testskripte ausgeführt (nicht wie bei Selenium: Keine Browserinteraktion, sondern direkte Kommunikation mit Webseite)
(3.2.4) Fuzzing: Zufällige Eingaben bei Inputs – kann in mangelhaftem Code beispielsweise Buffer Overflows provozieren
(3.3) Bietet Möglichkeit zur Integration weiterer Addons für spezialisierte Angriffe (sog. Marketplace)

Lizenz der genannten Werkzeuge
Alle genannten Werkzeuge sind Open Source (Jenkins unter MIT License, Selenium und ZAP unter Apache License)

CI Continuous Integration
CD Continuous Delivery
ZAP Zed Attack Proxy



Grober Ablauf der Pipeline wie sie bereits im Einsatz ist:

(1) Neuer Code im Repository, dadurch wird in Jenkins der Prozess ausgelöst

(2) Jenkins baut die Anwendung und liefert sie auf einen Testserver aus

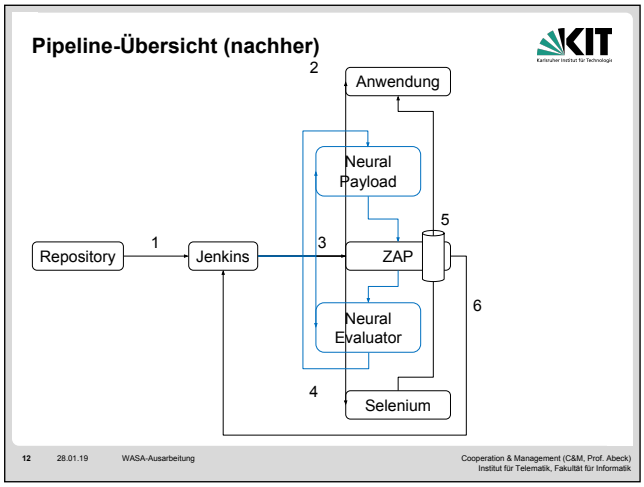
(3) Jenkins startet ZAP im Proxy-Modus

(4) Jenkins startet Selenium-Tests (vorher wurde in Selenium ZAP als Proxy konfiguriert)

(5) Selenium arbeitet Tests ab, tunnelt Verbindungen durch ZAP

(6) ZAP meldet Mängel an Jenkins

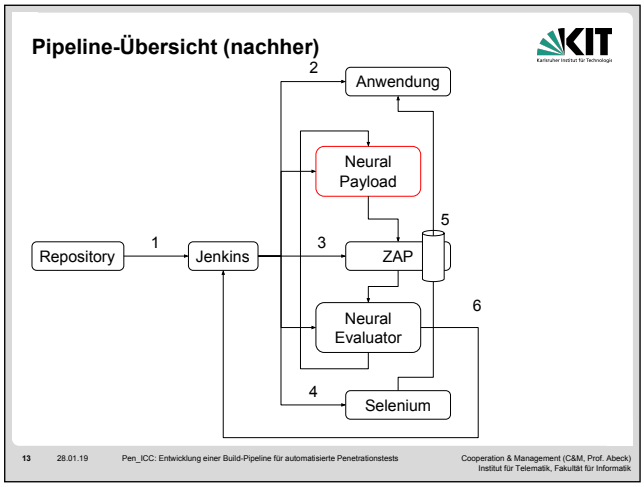
ZAP Zed Attack Proxy



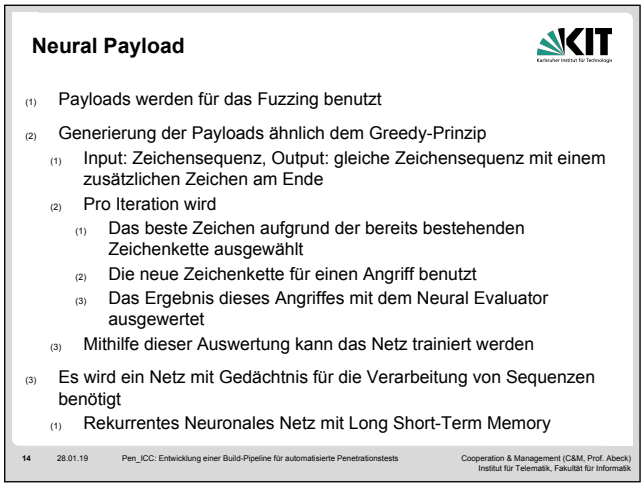
Änderungen im Rahmen des Projektes

(NeuralPayload, Neural Evaluator) Einbringen des Neural Payload und Neural Evaluator, die ZAP neue Angriffsvektoren liefern und auswerten. Bevor Selenium-Tests gestartet werden, wird Neural Payload & Evaluator gestartet

ZAP Zed Attack Proxy



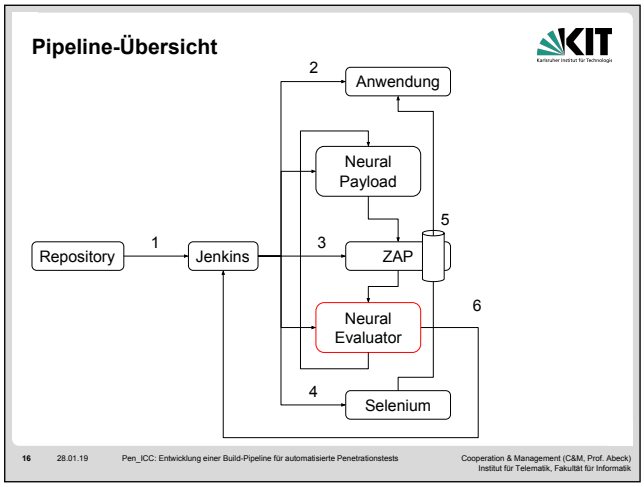
Üblicherweise werden Pentrationstests manuell ausgeführt, das heißt, dass ein oder mehrere Tester die Tests manuell vorbereiten, ausführen und auswerten muss. Unser Projekt beschäftigt sich damit diesen Vorgang zu automatisieren. Für die Automatisierung benutzen wir zwei neuronale Netze. Das erste Netz namens Neural Payload ist dafür zuständig mithilfe von Deep Learning automatisch Payloads generieren.



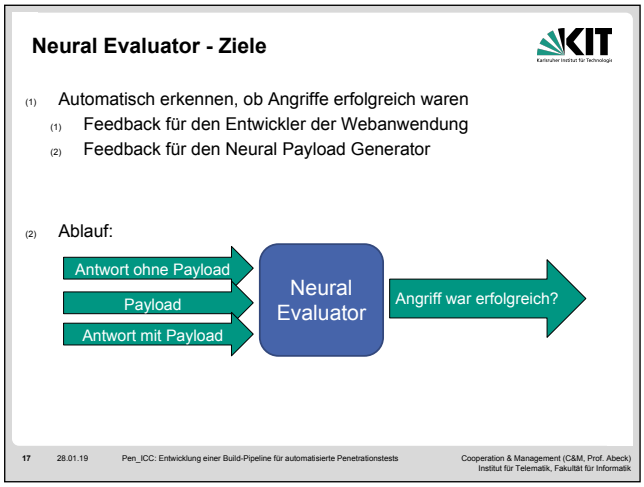
(1) Input Fuzzing ist eine Technik, welche oft bei Penetrationstests für Webapplikationen verwendet wird, um Sicherheitslücken ausfindig zu machen. Dabei benutzt man potentiell gefährliche Eingabewerte für alle vom Nutzer veränderbaren Variablen.

(2) Die Generierung der Payloads folgt dem Prinzip, wie man es aus Greedy-Algorithmen kennt: Man sucht lokal nach dem besten Wert.
(2.2.1) Am Anfang jeder Iteration wird zuerst das beste Zeichen aufgrund der bereits bestehenden Zeichenkette ausgewählt.
(2.2.2) Diese Zeichenkette wird dann für einen Angriff auf die zu testende Webapplikation verwendet.
(2.2.3) Das Ergebnis dieses Angriffes wird vom Neural Evaluator ausgewertet.
(2.3) Mithilfe der Auswertungen wird das Netz dann von Zeit zu Zeit trainiert, sodass das Generieren von Payloads sich stetig verbessert.

(3) Eine der einfachsten Formen der neuronalen Netze sind die Feed-Forward-Netze. Diese geben Informationen nur in eine Richtung weiter und können daher kein Verhältnis zwischen verschiedenen Eingabedaten herstellen. Dies ist für viele Problemstellungen nicht störend, jedoch kann es für das Verarbeiten von Sequenzen sehr nützlich sein ein solches Verhältnis herstellen zu können. Ein Beispiel, welches dies verdeutlichen kann ist die menschliche Texterkennung. Wenn wir einen einzelnen Buchstaben lesen, lässt sich dieser nur schwer klassifizieren. Lesen wir jedoch eine mehrere aufeinanderfolgende Buchstaben, so lassen sich einfach Wörter erkennen. Deswegen benutzten wir für den Neural Payload ein rekurrentes neuronales Netz mit Long Short-Term Memory. Dabei handelt es sich um ein Netz mit rückgekoppelten Kanten und speziellen Zellen, welche in der Lage sind sich Werte über beliebig lange Zeitdauer zu „merken“.

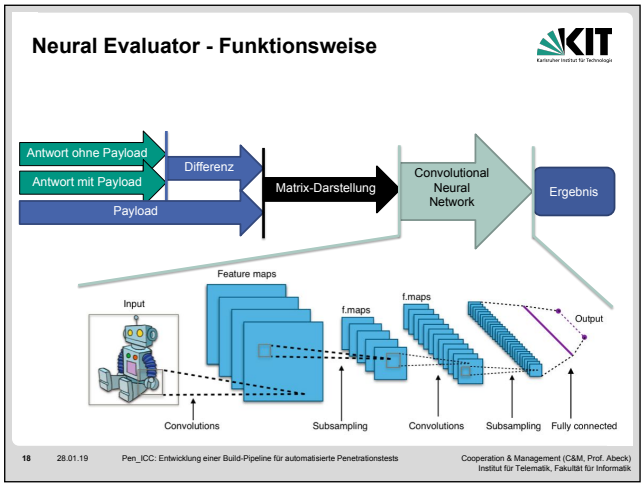


Üblicherweise werden Pentrationstests manuell ausgeführt, das heißt, dass ein Penetrationstester die Test manuell vorbereiten, ausführen und auswerten muss. Unser Projekt beschäftigt sich damit diesen Vorgang zu automatisieren. Für die Automatisierung benutzen wir zwei neuronale Netze. Das erste Netz namens Neural Payload ist dafür zuständig mithilfe von Deep Learning automatisch Payloads generieren.



(1) Damit Penetrationstests Sinn ergeben, muss natürlich nach einem Angriff auf das System auch überprüft werden, ob die Angriffe erfolgreich waren. Dies macht normalerweise, auch bei automatisierten Penetrationtests, der Penetrationtester manuell. Da in unserer Anwendung die Penetrationtests als Teil einer Build-Pipeline komplett autonom laufen sollen, brauchen wir ein System, was diese Aufgabe übernehmen kann, dies ist der Neural Evaluator.
(1.1) Als Teil der automatischen Build-Pipeline, die nach von Entwicklern durchgeführten Änderungen durchläuft, soll neben beispielsweise Unit-Tests auch die Penetration-Tests laufen. In dem diese ausgewertet werden, kann dem Entwickler auch darüber Feedback gegeben werden, ob seine Änderungen eine neue Sicherheitslücke hinzugefügt haben, sodass ein automatisches Deployment der neuen Version noch gestoppt werden kann.
(1.2) Außerdem kann der Neural Evaluator als Feedback für den Payload Generator dienen, welcher immer neue Payloads generiert, mit dem Ziel erfolgreiche Angriffe auszuführen. Indem wir ihm Feedback geben, welche Payloads erfolgreich und welche nicht erfolgreich waren, können wir diesen weiter trainieren, um zukünftig noch bessere Payloads zu generieren.

(2) Der Neural Evaluator ist der letzte Teil der Build-Pipeline und wertet die Ergebnisse der vorangegenen Phasen aus.
(2.1) Dazu erhält das neuronale Netz zum einen die Antwort der Webseite, wenn kein Payload angegeben wird und daher auch kein Angriff vorlag. Sowie das vom Neural Payload Generator erstellte Payload, welches an die Webseite geschickt wurde und die darauffolgende Antwort der Webseite. Damit erhält das Netz alle beim Blackbox-Testing vorhandenen Daten.
(2.2) Mit diesen Daten berechnet das neuronale Netz dann eine Wahrscheinlichkeit, ob der der Angriff auf die Webseite erfolgreich war und gibt das wahrscheinlichere Ergebnis zurück.

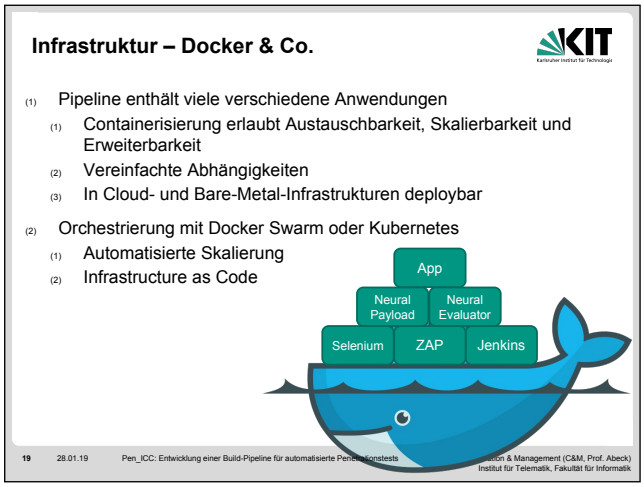


(1) Anstatt im Neuronalen Netz jeweils die gesamten Antworten der Webseite (mit und ohne Payload) zu analysieren (welche unter Umständen sehr groß sein können), wird zunächst die Differenz zwischen diesen berechnet.

(2) Die gerade berechnete Differenz und der verwendete Payload werden als Matrix dargestellt, in dem einzelne Buchstaben als Vektoren und die “Wörter“ als Liste der Buchstaben-Vektoren.

(3) Daraufhin wird ein Convolutional Neural Network mit der Matrix als Eingabe gestartet, welches dann die Wahrscheinlichkeit eines erfolgreichen Angriffs als Zahl zwischen 0 und 1 als Ausgabe zurückgibt.
(3.1) CNNs bestehen aus mehreren Layern und werden häufig in der Bilderkennung eingesetzt. Dabei werden zunächst einzelne Teile der Matrix (Convolution) in Feature-Maps übertragen, bevor diese vereinfacht werden (Pooling). Dieser Schritt wird mehrfach wiederholt, sodass man große Menge an relativ kleinen Feature-Maps erhält, welche dann jeweils an ein Neuron weitergegeben werden. Diese Neuronen sind dann im Fully-Connected-Layer alle miteinander verbunden und bilden letztendlich das Ergebnis des Netzes.

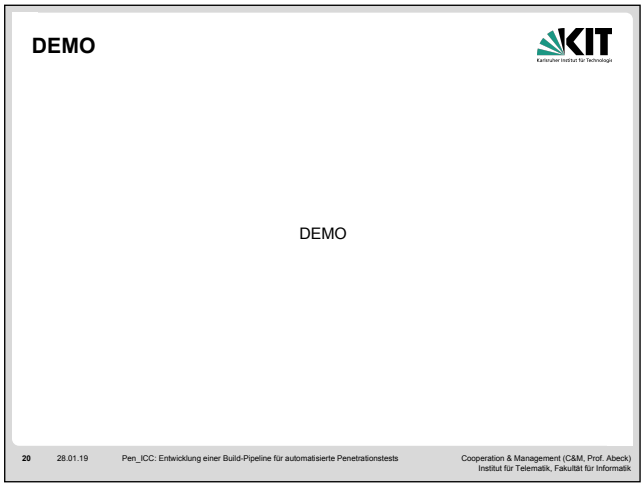
Bild: Typical CNN Architecture by Aphex34 [https://commons.wikimedia.org/wiki/File:Typical_cnn.png]

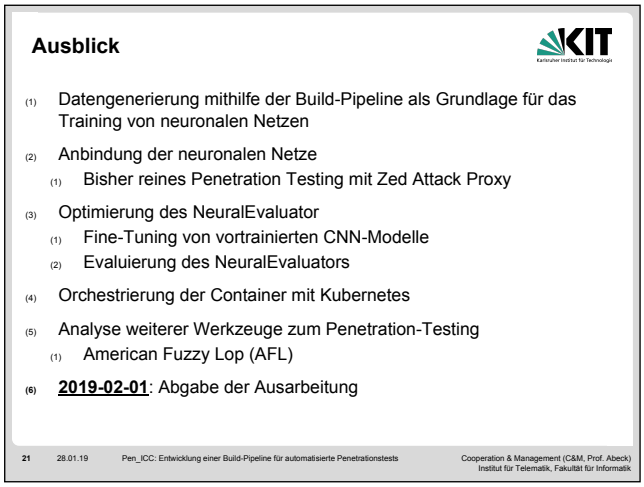


(1) Wie bereits gesehen, arbeiten in unserer Build-Pipeline, die später nach jedem Push in das Repository laufen soll, viele verschiedene Anwendungen zusammen. Diese laufen natürlich ganz im Sinne der Vorlesung als Microservices in Docker-Containern.
(1.1) Dies erlaubt es uns einzelne Anwendungen der Pipeline beispielsweise zum Updaten austauschen oder auch automatisch und schnell die Anwendungen zu skalieren, um beispielsweise darauf zu reagieren, wie viele Pipelines gerade laufen. Außerdem können so auch leicht weitere Anwendungen zur Pipeline hinzugefügt werden, denkbar wären beispielsweise Unit-Tests oder Code-Inspections.
(1.2) Wenn man an die Pipeline-Diagramme von zuvor denkt, sollte einem bewusst sein, dass es einige Abhängigkeiten zwischen den unterschiedlichen Anwendungen gibt, beispielsweise hängt alles von Jenkins ab, die Neuronalen Netze von ZAP und Selenium und mehr. Wenn man nun auch noch auf die Abhängigkeiten der einzelnen Anwendugen achten müsste, beispielsweise unterschiedliche benötigte Java-Versionen und verschiedene Bibliotheken würde die Pipeline deutlich komplizierter werden. Da die Docker-Container alle ihre Abhängkeiten selbst beinhalten ist das jedoch kein Problem für uns.
(1.3) Außerdem erlauben Container die Umsetzung dieser Pipelines in allen möglichen Infrastrukturen. Sowohl in Cloud-Umgebugen als auch eigenen Rechenzentren.

(2) Zum Aufbau unserer Pipeline nutzen wir insbesondere auch Orchestrierungstools wie Docker Swarm und Kubernetes.
(2.1) Diese übernehmen unter anderem die Ressourcenverwaltung und die automatische Skalierung der Container.
(2.2) Diese Tools erlauben es außerdem den gewünschten Stand der Container als Infrastrukturcode zu definieren und erlauben es so, unglaublich schnell komplette Umgebungen auf- und abzubauen und sind ein elementarer Bestandteil von Product-As-A-Service.

Bild: Docker Logo [https://dwglogo.com/docker-software-logo/]


DEMO
DEMO
20 28.01.19 Pen_ICC: Entwicklung einer Build-Pipeline für automatisierte Penetrationstests
Cooperation & Management (C&M, Prof. Abeck)
Institut für Telematik, Fakultät für Informatik



Ausblick

(1) Datengenerierung mithilfe der Build-Pipeline als Grundlage für das Training von neuronalen Netzen

(2) Anbindung der neuronalen Netze
  (1) Bisher reines Penetration Testing mit Zed Attack Proxy

(3) Optimierung des NeuralEvaluator
  (1) Fine-Tuning von vortrainierten CNN-Modelle
  (2) Evaluierung des NeuralEvaluators

(4) Orchestrierung der Container mit Kubernetes

(5) Analyse weiterer Werkzeuge zum Penetration-Testing
  (1) American Fuzzy Lop (AFL)

(6) **2019-02-01**: Abgabe der Ausarbeitung

21 28.01.19 Pen_ICC: Entwicklung einer Build-Pipeline für automatisierte Penetrationstests Cooperation & Management (C&M, Prof. Abeck) Institut für Telematik, Fakultät für Informatik

5.2 Wandtafel

Karlsruher Institut für Technologie

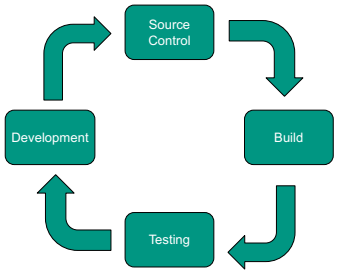
iC CONSULT

c&m IT RESEARCH

Automatisierte Penetrationstests mithilfe von Neuronalen Netzen

Pierre Bonert, Nikolai Franke, Jean-Marc Hendrikse, Tobias Polly, Julian Todt

CI/CD-Pipeline

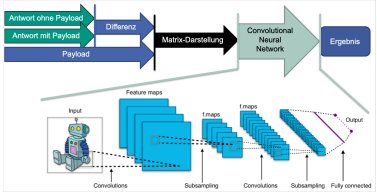


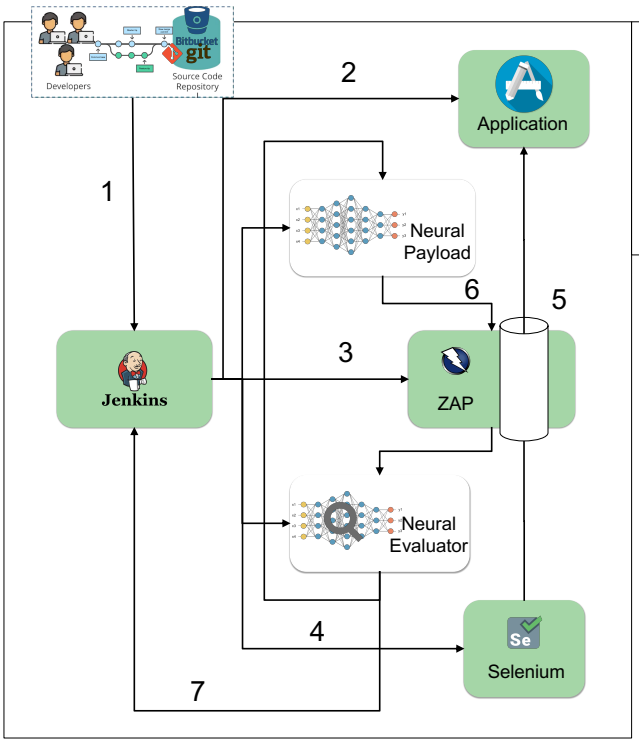
CI ist ein fortlaufender Prozess, der sich in folgende Schritte einteilen lässt:

- Ein Entwickler commited Änderungen am Code in das VCS
- Durch den Commit wird automatisch der Build-Prozess gestartet
- Unit Tests werden ausgeführt
- Ein Bericht mit den Ergebnissen wird an den Entwickler gesendet

Neural Evaluator

- Beurteilt, ob die ausgeführten Angriffe erfolgreich waren.
- Informiert den Software-Entwickler über mögliche Sicherheitslücken
- Wird auch dazu genutzt, um den Neural Payload zu trainieren
- Nutzt ein Convolutional Neural Network (bekannt aus der Bilderkennung) zur Feature-Extrahierung



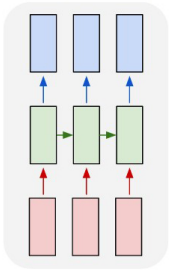


1. Neuer Code im Repository startet den Jenkins-Buildprozess.
2. Jenkins baut die Anwendung und liefert sie auf eine Testumgebung aus.
3. Jenkins startet OWASP Zed Attack Proxy (ZAP).
4. Jenkins startet Selenium-Tests.
5. Die von Selenium ausgeführten Anfragen an die Anwendung werden durch ZAP als Proxy getunnelt. ZAP agiert dabei als passiver Scanner.
6. Der Neural Payload generiert als Fuzzer Payloads und speichert die Antworten der Anwendung.
7. Der Neural Evaluator evaluiert die von ZAP und Neural Payload generierten Reports auf Sicherheitslücken in der Anwendung und sendet diese an Jenkins.

Neural Payload

- Zuständig für das Generieren von Payloads (in Form einer Zeichenkette)
- Nimmt als Input die bereits bestehende Zeichenkette entgegen und gibt eine neue Zeichenkette aus

many to many



- Das Generieren funktioniert iterativ
- Pro Iteration wird das Zeichen, welches die besten Resultate verspricht, an die Kette angehängt
- Rekurrentes neuronale Netz mit LSTM-Zellen, welche das Verarbeiten von Sequenzen ermöglichen

KIT – Die Forschungsuniversität in der Helmholtz-Gemeinschaft

www.kit.edu

5.3 Glossar

Bitbucket Webbasierter Filehoster für Softwareprojekte von *Atlassian* `https://bitbucket.com`.

Eclipse Open-source Java-IDE `https://eclipse.org`.

Enterprise Architect
Softwaremodellierungswerkzeug von *SparxSystems* `https://sparxsystems.com`.

Git Open-source Versionsverwaltungssystem `https://git-scm.com`.

IntelliJ Java-IDE von *JetBrains* `https://jetbrains.com/idea`.

Kubernetes Open-source Container-Orchestrierungssystem (ursprünglich von *Google* entwickelt) `https://kubernetes.io`.

Sharepoint Zusammenarbeits-Webanwendung von *Microsoft* `https://sharepoint.com`.

Slack Webbasierter Instant-Messanger von *Slack Technologies* `https://slack.com`.

5.4 Abkürzungsverzeichnis

BDD Behaviour-Driven Development.

BFF Backend for Frontend.

C&M Cooperation & Management.

DDD Domain-Driven Design.

IDE Integrated Development Environment.

JDK Java Development Kit.

TLM Todolistmanagement.

5.5 Literaturverzeichnis

[Ap15] APHEX34: *Typical CNN architecture*. https://commons.wikimedia.org/wiki/File:Typical_cnn.png. Version: 2015

[Co18a] COOPERATION & MANAGEMENT: *TLM Practical Course*. https://team.kit.edu/sites/cm-tm/Mitglieder/2-0.Aktuelles_Semester/18-10-17.TLM_PRACTICAL_COURSE.pdf. Version: 2018

[Co18b] COOPERATION & MANAGEMENT: *Web Application Development*. https://team.kit.edu/sites/cm-tm/Mitglieder/2-0.Aktuelles_Semester/18-10-24.WEB_APPLICATION_DEVELOPMENT.pdf. Version: 2018

[Do18] DOCKER INC.: *Docker Documentation: Get Started*. https://docs.docker.com/get-started/. Version: 2018

[DW18] DWGLOGO: *Docker Logo*. https://dwglogo.com/docker-software-logo. Version: 2018

[Ev03] EVANS, Eric: *Domain-driven design: tackling complexity in the heart of software*. Addison Wesley, 2003. – ISBN 0–321–12521–5

[Mö18a] MÖSSNER, Cedric: Entwicklung eines neuronalen Netzes zur Automatisierung von Penetrationtests. (2018)

[Mö18b] MÖSSNER, Cedric: *Neural Evaluator API*. https://github.com/TheMorpheus407/Neural-Evaluator/tree/master/RestplusAPI/api/api_endpoints/endpoints. Version: 2018

[PB16] PISKOZUB, Yaroslav Stefinko; A. ; BANAKH, Roman: Manual and Automated Penetration Testing. Benefits and Drawbacks. Modern Tendency. In: *TCSET'2016* (2016)

[SW11] SOLÍS, Carlos ; WANG, Xiaofeng: A Study of the Characteristics of Behaviour Driven Development. In: *37th EUROMICRO Conference on Software Engineering and Advanced Applications* (2011)