

Hochschule Rhein-Waal
Rhine-Waal University of Applied Science
Faculty of Communication and Environment

Projektbericht

Secure Coding Statische Code Analyse

Projektbericht WS 2022/23

Fälligkeitsdatum: 03.02.2023

Modul: Technischer Datenschutz und Mediensicherheit

Dozent: Prof. Dr. Ulrich Greveler

Sebastian Chromik (28963)

Abweichung vom Onepager	3
Projektvorstellung	3
Dynamische Code Analyse	4
Statische Code Analyse	7
Ursprung	7
Abstract Syntax Tree	8
ESLint	10
Implementation einer eigenen Regel	10
Semgrep	14
Einbindung in Github CI-Pipeline	16
ESLint	16
Semgrep	17
Vor- und Nachteile	18
Fazit und Reflektion	19
Github	19

Abweichung vom Onepager

Die ursprüngliche Idee war es, mittels Codeanalyse Tools, vorrangig GitLab SAST und DAST, Tests für Webanwendungen zu erstellen und durchzuführen. Hieraus sollte hervorgehen, inwieweit sich für Webapplikationen durch automatisierte Sicherheitstests die Sicherheit erhöhen bzw. ein gewisser Sicherheitsstandard gewährleisten lässt.

Im Verlauf des Projekts taten sich allerdings viele Probleme und Unklarheiten im Bereich der dynamischen Code-Analyse auf. Aus diesem Grund wollte ich den Schwerpunkt des Projektes weg von der Anwendung mehr auf das Verständnis und die Funktionsweise statischer Codeanalyse verlagern. Grundlegend ist die dynamische Code Analyse ein mächtiges Tool, allerdings erfordert diese ebenfalls ein tiefes Verständnis von Programmiersprachen, Softwarearchitektur und Systemdesign. Eine zusätzliche Herausforderungen der dynamischen Codeanalyse ist die große Anzahl der Variablen und Eingaben, die berücksichtigt werden müssen.

Projektvorstellung

Diese Projektarbeit handelt im Allgemeinen über statische Code-Analyse. Im genaueren werden verschiedene Tools betrachtet und welche Funktionsweise dahinter steht. Zusätzlich wird dies konkret am Beispiel von ESLint und Semgrep dargestellt. Mittels dieser Tools wird gezeigt, wie diese eingesetzt werden können und welche Möglichkeiten bestehen. Ebenfalls wurde ein Github Repository erstellt, in welchen die hier gezeigten Beispiele implementiert wurden, zusätzlich wird die Installation und Konfiguration dort nochmal genauer erläutert.

Dynamische Code Analyse

Bei der dynamischen Code Analyse wird ein geschlossenes und laufendes System als eine Art Black Box betrachtet, es erfolgt also keine Analyse des Codes im direkten Sinne. Bei dieser Vorgehensweise wird also ein alleinstehendes System von außen angegriffen, wie es auch in der realen Welt, im Sinne eines Hackerangriffs, der Fall wäre. Es kann ebenfalls als Pentest angesehen werden. Bei der dynamischen Code Analyse steht die Sicherheit im Vordergrund, Performanceprobleme oder Bugs können ebenfalls ausfindig gemacht werden, sind aber meist nicht die Hauptaufgabe eines solchen Tests.

Für dynamische Code Analyse gibt es eine Vielzahl an Frameworks und Open Source Tools, wie beispielsweise den OWASP Zed Attack Proxy (OWASP ZAP), welcher unter anderem auch in diesem Projekt im Rahmen von GitLab DAST (Dynamic Application Security Testing) genutzt wurde. Im Bezug auf eine Webanwendung, erstellt ZAP eine Übersicht über die Seiten der Webanwendungen, sowie der Ressourcen, welche zum Rendern dieser Seiten verwendet werden. Anfragen und Antworten werden aufgezeichnet und gegebenenfalls Warnmeldungen erstellt, wenn mit einer Antwort möglicherweise etwas nicht stimmt. Diese Meldungen, sowie Anfragen und Antworten können im Falle von GitLab DAST anschließend in einer GUI eingesehen und beispielsweise nach Schweregrad gefiltert werden. Die Anfragen zielen dabei auf die Sicherheitslücken der OWASP-Top-Ten ab. [1]

WebGoat wurde als zu testende Webanwendung verwendet. Dabei handelt es sich um eine absichtlich unsichere Webanwendung, welche ebenfalls von der OWASP Foundation entwickelt wurde. Diese ermöglicht es, Schwachstellen zu testen, die häufig in Java-basierten Anwendungen gefunden werden, welche beliebte und gängige Open-Source-Komponenten verwenden.[2] Der Test wurde mittels GitLab DAST in einer CI-Pipeline durchgeführt. Zu den gefundenen Sicherheitslücken gehören beispielsweise reflektiertes Cross-Site-Scripting, die Verwendung unsicherer Bibliotheken oder SQL Injections. Letzteres lässt sich anhand der gefundenen Sicherheitslücke gut erläutern.

SQL Injection

×

Status: **Detected**

Description: SQL injection may be possible.

Project: 28963 / webgoat-test

Method: POST

URL: <http://webgoat:8080/WebGoat/crypto/hashing>

Request: POST <http://webgoat:8080/WebGoat/crypto/hashing>
Accept: */*
Accept-Language: en-US
Content-Length: 39
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Cookie: JSESSIONID=*****
Host: webgoat:8080
Origin: <http://webgoat:8080>
Proxy-Connection: keep-alive
Referer: <http://webgoat:8080/WebGoat/start.mvc>
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)

Actual Response:

200 OK
Connection: keep-alive
Content-Type: application/json
Date: Thu, 10 Nov 2022 03:15:32 GMT
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block

<Message body is not provided>

Evidence: The page results were successfully manipulated using the boolean conditions [AND 1=1 --] and [OR 1=1 --] The parameter value being modified was stripped from the HTML output for the purposes of the comparison Data was NOT returned for the original parameter. The vulnerability was detected by successfully retrieving more data than originally returned, by manipulating the parameter

Identifiers: [SQL Injection](#), [CWE-89](#)

Severity: ◆ High

Tool: DAST

Scanner Provider: OWASP Zed Attack Proxy (ZAP) and Browserker

Links: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Abbildung 1: GitLab DAST

Wie bereits erwähnt, werden Anfrage und Antwort gespeichert, darüber hinaus ebenfalls die angewandte Methode und der Beweis bzw. Nachweis (siehe "Evidence" in Abbildung 1) der gefundenen Sicherheitslücke. Unter "Evidence" ist nun genau nachzuvollziehen, welcher Input zu dieser Sicherheitslücke geführt hat. In diesem Fall war es ein SQL boolean Statement, mit welchem sich zusätzliche Informationen aus der Datenbank abrufen ließen. Da ZAP natürlich hier nicht weiß inwiefern es sich um Informationen handelt, welche nicht bestimmt sind für den Nutzer, wird hier lediglich gemeldet, dass der Output mit diesem SQL-Statement modifiziert werden konnte.

Employee Name:

Authentication TAN:



37648	John	Smith	Marketing	64350	3SL99A
-------	------	-------	-----------	-------	--------

Employee Name:

Authentication TAN:



USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY	AUTH_TAN
32147	Paulina	Travers	Accounting	46000	P45JSI
34477	Abraham	Holman	Development	50000	UU2ALK
37648	John	Smith	Marketing	64350	3SL99A
89762	Tobi	Barnett	Development	77000	TA9LL1
96134	Bob	Franco	Marketing	83700	LO9S2V

Abbildung 2: Webgoat SQL Injection

Die Abbildung 2 zeigt den modifizierten Output mittels dem boolean SQL-Statement.

Statische Code Analyse

Bei der statischen Codeanalyse wird der Programmcode auf potenzielle Probleme oder Fehler untersucht, ohne dass der Code tatsächlich ausgeführt bzw. kompiliert wird. Es handelt sich dabei unter anderem auch um eine Methode zur Bewertung der Codequalität. Mithilfe von verschiedenen CI/CD-Tools können solche Analysen auch automatisiert ausgeführt werden. Zu den häufigen Problemen, die mit Hilfe der statischen Codeanalyse identifiziert werden können, gehören Syntaxfehler, Sicherheitsschwachstellen, Verstöße gegen Compliance und Leistungsprobleme. Durch die frühzeitige Erkennung dieser Probleme und dem geringem Zeitaufwand im Entwicklungsprozess sind statische Code Analysen eine gute und weit verbreitete Möglichkeit, Sicherheit und Standards durchzusetzen, vereinfacht formuliert, zu einer höheren Qualität der Software beizutragen.

Ursprung

Die statische Code Analyse nimmt ihren Ursprung in den späten siebziger Jahren, wo das erste Tool Namens "Lint", von den Bell Labs entwickelt, grundsätzlich dazu dienen sollte die damaligen Schwächen eines Compilers auszugleichen, da diese nur sehr rudimentäre Prüfungen vornahmen. Stephen Johnson, damals bei den Bell Laboratories, entwickelte ein Werkzeug zur Untersuchung von C-Quellprogrammen, um Fehler zu finden, welche dem Compiler entgangen waren. Der Begriff "Lint" (englisch für "Fussel") bzw. "Linter" bezieht sich dabei auf seine Analogie zur Fusselrolle. Ähnlich wie diese über die Kleidung gerollt wird, sollte ein Linting Programm über einen gegebenen Code laufen und "ungewollte Stücke" entfernen bzw. melden.[3]

Statische Analyse Tools stellen eine von vielen Möglichkeiten dar, Fehler in einem Programm zu reduzieren. Beispielsweise sind Code-Reviews ebenfalls eine weit verbreitete Methodik, um die Qualität von Software zu verbessern. Allerdings braucht es viel Zeit und auch Übung, wenn eine Gruppe von Leuten unbekannten Code studieren muss, um mögliche Fehler zu finden. Hinzu kommt, dass es gerade bei Problemen, seien es Bugs oder Sicherheitslücken, oftmals von großer Bedeutung ist, wann diese gefunden werden.

Am besten wäre es also Fehler zu erkennen in dem Moment wo man sie macht. Die meisten Fehler fallen in bekannte Kategorien, da Menschen dazu neigen, immer wieder in die gleichen Fallen zu tappen. Genau aus dieser Vorhersehbarkeit entsteht der Nutzen für Werkzeuge wie Lint. [4]

Abstract Syntax Tree

Abstract Syntax Tree

Die Funktionsweise statischer Codeanalyse basiert auf einem Abstract Syntax Tree, kurz AST. Ein Abstract Syntax Tree ist eine baumartige Darstellung der Struktur von Programmcode. Er wird erstellt, indem die Syntax des Quellcodes in "Tokens" unterteilt wird, bzw. in dem der Code in seine einzelnen Komponenten, wie Variablen, Funktionen und Steueranweisungen, zerlegt wird. Anschließend werden diese Tokens in größere Einheiten gruppiert, welche Knoten (Nodes) genannt werden und die verschiedenen Elemente des Codes darstellen. Die Knoten des AST sind in einer Baumstruktur organisiert, wobei jeder Knoten ein bestimmtes Element des Codes darstellt und seine untergeordneten Knoten die Komponenten repräsentieren, aus denen dieses Element besteht. Ein Funktionsdeklarationsknoten kann zum Beispiel Unterknoten haben, die den Funktionsnamen, die Argumente und den Körper der Funktion darstellen.

Dabei würde die im folgenden Bild zu sehende JavaScript Funktion den links daneben stehenden Tree erzeugen. Der Code wird in diesem Fall in sieben Nodes unterteilt, die wiederum aus 14 Tokens bestehen.

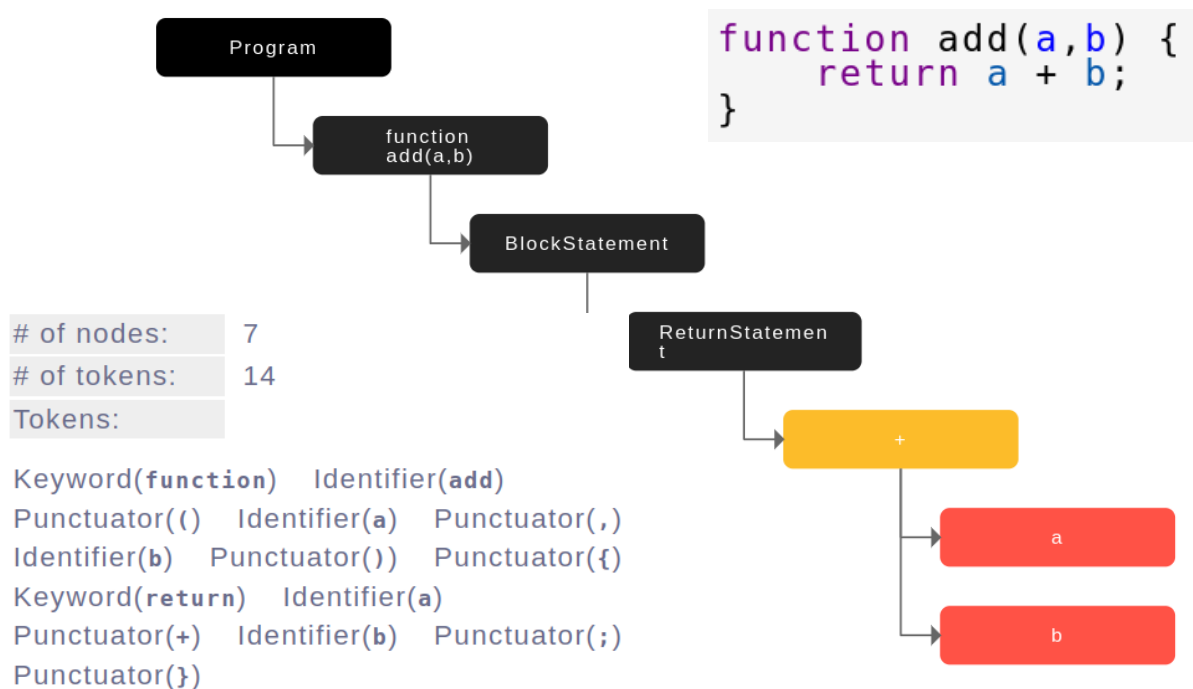


Abbildung 3: Darstellung AST



Abbildung 4: Quellcode zu AST

Abbildung 4 visualisiert den Vorgang vom Source Code bis zum AST. Zunächst erfolgt die Zuweisung der Tokens, dies ist auch Teil der lexikalischen Analyse. Ein Parser führt nun auf Basis dieser Vorverarbeitung eine syntaktische Analyse durch, sprich überprüft, ob die formale Grammatik der Programmiersprache eingehalten wird und erstellt einen Syntaxbaum, welcher die syntaktischen Zusammenhänge repräsentiert.

Ebenfalls spielen ASTs eine wichtige Rolle für Compiler. Dort werden sie als Zwischendarstellung (auch Zwischencode genannt) genutzt und sind meistens das Resultat der Syntaxanalysephase eines Compilers. Programmiersprachen sind von Natur aus oft mehrdeutig. Um diese Mehrdeutigkeit zu vermeiden, werden sie oft in Form einer kontextfreien Grammatik (CFG) spezifiziert. Es gibt jedoch häufig Aspekte von Programmiersprachen, die eine CFG nicht ausdrücken kann, die aber trotzdem Teil der Sprache sind. Dabei handelt es sich um Details, die einen Kontext erfordern, um ihre Gültigkeit und ihr Verhalten zu bestimmen. Wenn eine Programmiersprache beispielsweise die Deklaration neuer Typen erlaubt, kann eine CFG weder die Namen solcher Typen noch die Art und Weise ihrer Verwendung vorhersagen. Der Sinn eines Abstract Syntax Tree ist es also, eben genau diesen Kontext herzustellen. Der Entwurf eines ASTs hängt oft mit dem Entwurf des Compilers eng zusammen.[5]

ESLint

ESLint gehört mit über 29 Millionen wöchentlichen Downloads zu den meistgenutzten Analysetools für Javascript. Die Aufgabe von ESLint ist allgemein formuliert die Herstellung von Konsistenz, sowie die Vermeidung von Bugs. Dafür bringt ESLint bereits ein vorgefertigtes Regelset mit, dies unterteilt sich in "mögliche Probleme", "Empfehlungen" und "Layout & Formatting". Für die Regeln kann definiert werden, ob diese einen "Error" werfen oder lediglich als "warning" auftauchen, natürlich ist es auch möglich sie vollständig zu deaktivieren.

Ein Grund für die weite Verbreitung von ESLint ist die Möglichkeit, vollständig eigene Regeln zu schreiben. Dies bringt viele neue Optionen, die Konsistenz im Code weiter zu verbessern, so können beispielsweise auch sehr spezifische und komplexere Compliance-Regeln durchgesetzt werden. Zusätzlich besteht die Möglichkeit, ESLint über Plugins in IDE's wie VSCode zu integrieren. Die Implementation von Regeln kann dabei ebenfalls in Javascript erfolgen. Bezüglich des Vorgehens empfiehlt es sich hier, den Code, für den die Regel geschrieben werden soll, in einem AST Format anzusehen. Dafür kann beispielsweise das Online Tool <https://astexplorer.net/> verwendet werden. Diese Schritte werden aber noch in dem nachfolgenden Kapitel erläutert.

Implementation einer eigenen Regel

Im folgenden Beispiel wird für eine bestehende Problematik eine benutzerdefinierte Regel geschrieben.

Oft wird in JavaScript für die Fehlerbehandlung die "try-catch-finally" Funktion verwendet. Dabei wird ein Fehler, welcher möglicherweise im Try-Block entsteht, im Catch-Block "aufgefangen". Der Code im finally-Block wird dabei immer ausgeführt, unabhängig vom Ergebnis des Try-Blocks. Eine Problematik, die hier entstehen könnte, ist dass ein Return-Statements im Try oder Catch-Block ignoriert würde, wenn ein Return-Statement im Finally-Block steht. Folgender Code (Abbildung 5) würde also immer "2" zurückgeben. Für

```
function test(){
  try{
    return 0;
  }catch(err) {
    return 1;
  }finally{
    return 2;
  }
}
```

diesen Fall kann mit ESLint nun eine Regel geschrieben werden, welche den Code daraufhin überprüft. Wie bereits erwähnt, macht es Sinn, sich die Struktur vorher in einem AST anzusehen, welche in Abbildung 6 dargestellt ist.

Abbildung 5: JavaScript Code, Return-Problematik

Hier wird nun ersichtlich (Abbildung 6), dass sich unter dem Knoten "TryStatement" drei weitere Knoten, nämlich Block, Handler und Finalizer befinden. Diese wiederum besitzen jeweils einen body in dem sich das return statement befindet.

```
- TryStatement {
  type: "TryStatement"
  start: 21
  end: 117
  - block: BlockStatement {
    type: "BlockStatement"
    start: 24
    end: 49
    - body: [
      + ReturnStatement {type, start, end, argument}
    ]
  }
  - handler: CatchClause {
    type: "CatchClause"
    start: 49
    end: 85
    + param: Identifier {type, start, end, name}
    - body: BlockStatement {
      type: "BlockStatement"
      start: 60
      end: 85
      - body: [
        + ReturnStatement {type, start, end, argument}
      ]
    }
  }
  - finalizer: BlockStatement {
    type: "BlockStatement"
    start: 92
    end: 117
    - body: [
      + ReturnStatement {type, start, end, argument}
    ]
  }
}
```

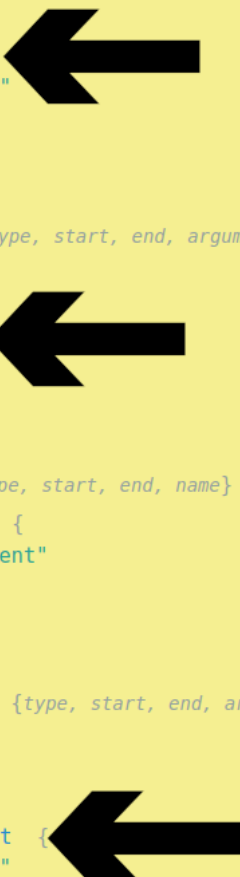


Abbildung 6: Quellcode im AST

Die Implementation der Regel sieht dabei wie folgt aus (Abbildung 7). Das Context Objekt stellt die grundlegende Funktionalität zur Verfügung. Innerhalb des Return-Statements wird ein Objekt mit Methoden zurückgegeben, welche genutzt werden, um den AST zu durchlaufen.

```
module.exports = {
  meta: {
    messages: {
      noReturnTryCatch: 'found return statement inside finally block',
    },
  },
  create(context) {
    return {
      TryStatement(node) {
        //Funktion aufrufen und Node übergeben
        if (checkTryCatchFinallyForReturn(node)) {
          context.report({ node: node, messageId: 'noReturnTryCatch' });
        }
      }
    }
  }
};
```

Abbildung 7: Regeldefinition

In diesem Fall wird die Funktion “checkTryCatchFinallyForReturn()” mit der jeweiligen Node des Try-Statements aufgerufen. In dieser Funktion werden dann die einzelnen Blöcke (try/catch/finally) durchlaufen und es wird geprüft, ob sich ein Return-Statement darin befindet.

```
function checkTryCatchFinallyForReturn(node) {
  let hasTryReturn = false;
  let hasCatchReturn = false;
  let hasFinallyReturn = false;

  //try block iterieren und auf return statement prüfen
  for(statement of node.block.body){
    if (statement.type === 'ReturnStatement') {
      hasTryReturn = true;
    }
  }

  //catch block iterieren und auf return statement prüfen
  for(statement of node.handler.body.body){
    if (statement.type === 'ReturnStatement') {
      hasCatchReturn = true;
    }
  }

  //falls finally block existent -> iterieren und auf return statement prüfen
  if(node.finalizer !== null){
    for(statement of node.finalizer.body){
      if (statement.type === 'ReturnStatement') {
        hasFinallyReturn = true;
      }
    }
  }

  //Ist im finally block ein return statement, darf sich keins im try oder catch block befinden
  if(hasFinallyReturn && hasTryReturn || hasFinallyReturn && hasCatchReturn){
    return true;
  }else{
    return false;
  }
}
```

Abbildung 8:
Regel Funktion

In der Funktion wird also jeweils durch das “body” Element des Blocks iteriert und geprüft, ob sich dort ein Return-Statement befindet. Wird die Regel verletzt, also ein true zurückgegeben, kann mittels “context.report” eine Fehlermeldung ausgegeben werden, welche in dem “meta” Objekt unter dem Attribut “messages” spezifiziert werden kann.

Semgrep

Semgrep steht für “semantic grep” und wurde 2009 von Facebook entwickelt. Es ist ebenfalls ein statisches Analysetool, im Vergleich zu ESLint fokussiert es sich aber mehr auf die Sicherheit. Es nutzt ebenfalls ASTs, verfolgt aber eine andere Logik bei der Regeldefinition. Semgrep arbeitet mit einer benutzerdefinierten Sprache namens “Semgrep Patterns”, welche eine Art reguläre Ausdrücke darstellen. Dabei verfügt Semgrep über eine eingebaute Logik, die erkennt, welche Codemuster äquivalent sind, so dass es nicht nötig ist, alle Kombinationsmöglichkeiten zu spezifizieren, die wir benötigen würden, wenn wir **nur** auf AST-Ebene arbeiten würden. Hierauf bezieht sich auch der erste Teil des Namens “Sem” also semantisch, Semgrep kennt also die semantischen Eigenschaften der Programmiersprache. Bei einer Suche nach dem String “make_super_user” würde also im folgenden Beispiel “msu” als äquivalent angesehen werden.

```
import make_super_user as msu

msu("admin", mutate=True)
```

Abbildung 9: Python Code

Wie bereits erwähnt baut sich der AST hauptsächlich auf den syntaktischen Zusammenhängen der Programmiersprache auf, Semgrep ergänzt dies nun und zieht die semantischen Eigenschaften ebenfalls mit in Betracht. Der Gedanke hinter Semgrep ist also, die Einfachheit der grep-Syntax mit der Robustheit und Präzision von ASTs zu verbinden.

Installiert werden kann Semgrep einfach und schnell mit dem Befehl “python3 -m pip install semgrep”. Die Regeln werden dabei im yaml-Format geschrieben. Ein möglichst einfaches Beispiel wäre hier die Suche nach der Funktion “eval()”. Dabei wird unter “patterns” eben genau diese Funktion angegeben, die drei Punkte definieren dabei, dass hier noch Null bis viele Zeichen folgen können.

```
rules:
- id: find_eval
  message: Use of eval could lead to code Injection
  languages: [javascript]
  severity: ERROR
  pattern: eval(...)
```

Abbildung 10: Semgrep Regel

Zusätzlich gibt es die Parameter "id", welches den Namen der Regel festlegt, "message" definiert die Fehlermeldung und "severity" den Schweregrad, welcher "info", "warning" und "error" annehmen kann. Semgrep steht des Weiteren für eine Vielzahl an Programmiersprachen zur Verfügung, was es ebenfalls zu einem weit verbreiteten Tool macht.

Eigene Regeln können einfach mit "semgrep -f meineRegeln.yml" angewendet werden. Die Möglichkeiten sind natürlich deutlich weitreichender als im Beispiel zu sehen. Nachfolgend zeigt die Abbildung 11 die Implementation derselben Regel, welche unter dem Punkt "Implementation einer eigenen Regel" für ESLint beschrieben wurde.

```
rules:
- id: return-in-init
  pattern-either:
  - pattern:
    try{
      ... return ...
    }
    catch($ERR){
      ...
    }
    finally{
      ... return ...
    }
  - pattern:
    try{
      ...
    }
    catch($ERR){
      ... return ...
    }
    finally{
      ... return ...
    }
  - pattern:
    try{
      ... return ...
    }
    catch($ERR){
      ... return ...
    }
    finally{
      ... return ...
    }
  message: Do not include a return statement inside a try or catch
  block if there is a return statement in the finally block
  languages:
  - js
  severity: ERROR
```

Abbildung 11: Semgrep für try/catch Problematik

Mittels "pattern-either" können mehrere Muster logisch OR-Verknüpft werden. In diesem Fall gibt es also drei Pattern, die zu einem Fehler führen sollen. Das "\$" Zeichen wird dabei genutzt, um genau eine Variable bzw. genau ein Wort zu erlauben.

Einbindung in Github CI-Pipeline

Da statische Codeanalyse ein leichtgewichtiger und schneller Prozess ist, kann diese gut in eine CI-Pipeline eingebunden werden, so dass die Tests immer wieder bei neuen Commits ausgeführt werden. Im Falle von Github nennt sich die CI/CD Lösung "Actions" bzw. die Pipelines werden als "Workflows" bezeichnet. Ein neuer Workflow kann unter dem Reiter "Actions" dann "New workflow" und anschließend "set up a workflow yourself" eingerichtet werden. Dies ist nichts anderes als ein Pipeline-Skript im yaml-Format.

ESLint

```
name: ESLint

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  eslint:
    name: Run eslint scanning
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Install ESLint
        run: |
          npm install eslint@8.10.0

      - name: Run ESLint
        run: npx eslint ./thisCodeGetsTested
          --config .eslintrc.js
          --ext .js,.jsx,.ts,.tsx
        continue-on-error: false
```

Abbildung 12: ESLint Pipeline-Skript

So lässt sich mit dem obigen Skript ESLint mit der Konfiguration, wie sie in der ".eslintrc.js" angelegt wurde, ausführen (siehe Github). "continue-on-error" sorgt dabei dafür, dass die Pipeline abbricht, sobald eine Regel verletzt wurde.

Semgrep

Semgrep bietet für die CI-Integration eine GUI über die Website semgrep.dev. Diese kann leicht eingerichtet werden und erstellt automatisch das Pipeline-Skript. Zusätzlich können über die GUI die Regeln sowie die Ergebnisse der Tests verwaltet werden.

```
name: Semgrep

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  semgrep:
    name: Scan
    runs-on: ubuntu-latest
    env:
      SEMGREP_APP_TOKEN: ${ secrets.SEMGREP_APP_TOKEN }
    container:
      image: returntocorp/semgrep
    steps:
      - uses: actions/checkout@v3
      - run: semgrep ci
#   - run: semgrep --error ./thisCodeGetsTested -f semgrepRules.yml
#Will man das Dashboard (www.semgrep.dev) nicht nutzen, kann man letzteren Befehl verwenden
#um die Regeln manuell einzubinden. Die error flag bedingt das die Pipeline bei einem Fund abbricht.
```

Abbildung 13: Semgrep Pipeline-Skript

Das Skript ist dabei sehr simple, es führt im Prinzip lediglich den Befehl “semgrep ci” aus, welcher sich die nötigen Informationen, sprich die Regeln, von semgrep.dev holt. Will man die GUI nicht verwenden, kann der darunter stehende Befehl genutzt werden, um die Regeln “manuell” anzuwenden. Die “error” Flag in dem Befehl sorgt dabei wieder dafür, dass die Pipeline abbricht, sollte eine Regel verletzt werden.

Eine genaue Anleitung zur Einrichtung über semgrep.dev, befindet sich auf Github.

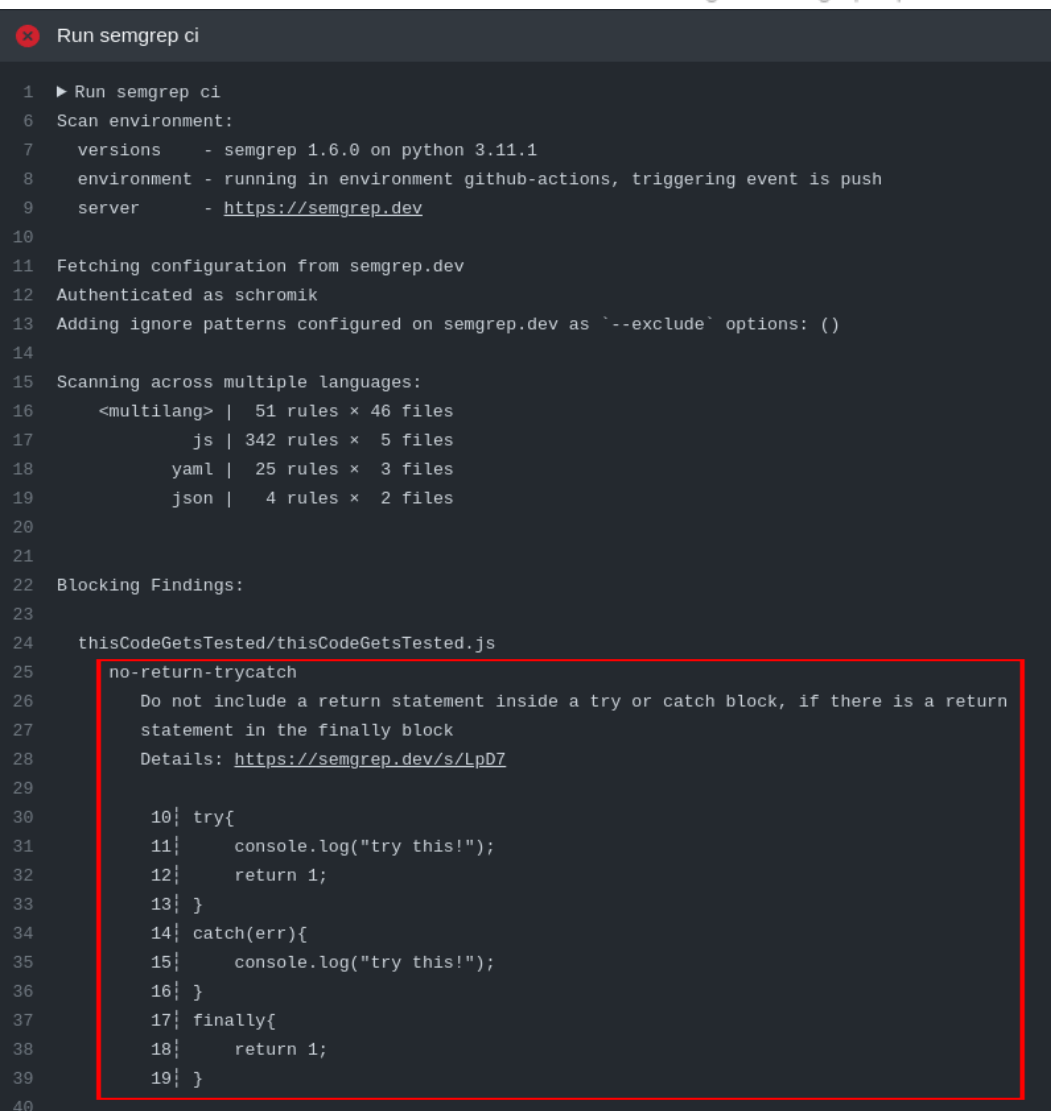
Wird nun ein fehlerhafter Code committed, welcher eine oder mehrere Regeln verletzt, bricht die Pipeline an dieser Stelle ab. Beispielsweise müsste also bei dem Commit des folgenden Codes, Semgrep, sowie ESLint einen Fehler finden, da im Try-, wie auch im Finally-Block ein Return-Statement zu finden ist.

```
function testA(){
  try{
    console.log("try this!");
    return 1;
  }
  catch(err){
    console.log("try this!");
  }
  finally{
    return 1;
  }
}
```

Abbildung 14: Fehlerhafter Code

Semgrep workflow

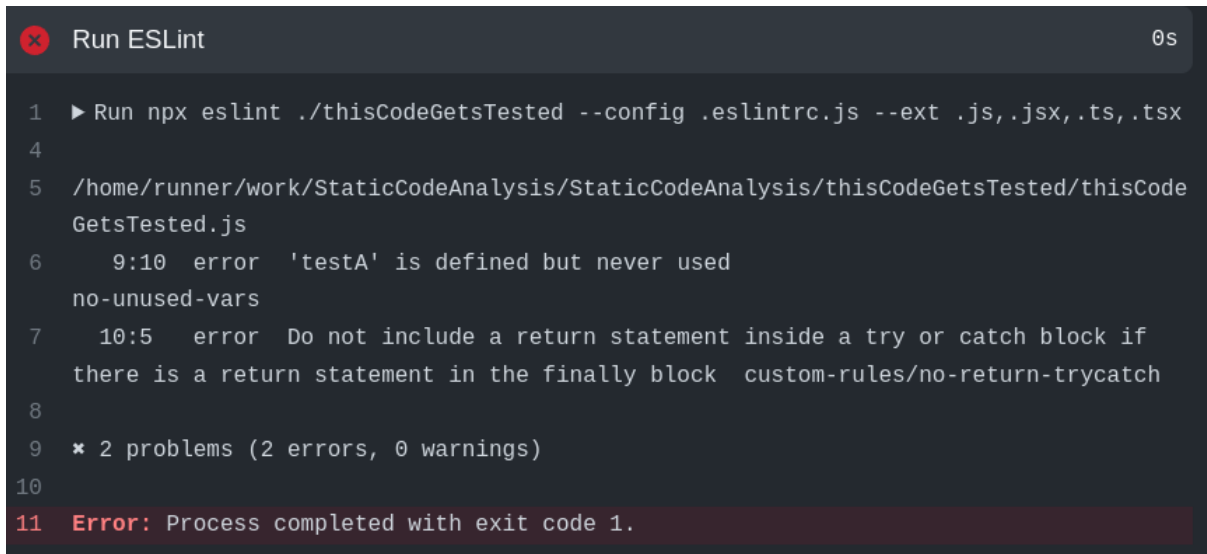
Abbildung 15: Semgrep Pipeline Resultat



```
❌ Run semgrep ci

1  ▶ Run semgrep ci
6  Scan environment:
7    versions   - semgrep 1.6.0 on python 3.11.1
8    environment - running in environment github-actions, triggering event is push
9    server      - https://semgrep.dev
10
11 Fetching configuration from semgrep.dev
12 Authenticated as schromik
13 Adding ignore patterns configured on semgrep.dev as `--exclude` options: ()
14
15 Scanning across multiple languages:
16   <multilang> | 51 rules × 46 files
17     js         | 342 rules × 5 files
18     yaml       | 25 rules × 3 files
19     json       | 4 rules × 2 files
20
21
22 Blocking Findings:
23
24 thisCodeGetsTested/thisCodeGetsTested.js
25 no-return-trycatch
26   Do not include a return statement inside a try or catch block, if there is a return
27   statement in the finally block
28   Details: https://semgrep.dev/s/LpD7
29
30   10| try{
31   11|   console.log("try this!");
32   12|   return 1;
33   13| }
34   14| catch(err){
35   15|   console.log("try this!");
36   16| }
37   17| finally{
38   18|   return 1;
39   19| }
40
```

ESLint workflow



```
1 ▶ Run npx eslint ./thisCodeGetsTested --config .eslintrc.js --ext .js,.jsx,.ts,.tsx
4
5 /home/runner/work/StaticCodeAnalysis/StaticCodeAnalysis/thisCodeGetsTested/thisCode
  GetsTested.js
6   9:10  error  'testA' is defined but never used
      no-unused-vars
7  10:5   error  Do not include a return statement inside a try or catch block if
      there is a return statement in the finally block  custom-rules/no-return-trycatch
8
9  * 2 problems (2 errors, 0 warnings)
10
11 Error: Process completed with exit code 1.
```

Abbildung 16: ESLint Pipeline Resultat

Zusätzlich gibt ESLint hier noch die Fehlermeldung raus, dass “testA” definiert ist aber nie genutzt wird, da diese Regel im Standardregelsatz implementiert ist. Der Verlauf dieser Pipeline kann ebenfalls im Github Repo nachvollzogen werden.

Vor- und Nachteile

Vorteile der statischen Codeanalyse:

- IDE unterstützung
- Probleme können im frühen Stadium des Entwicklungsprozesses identifiziert werden
- Sicherheitslücken, Bugs und Performanceprobleme können vermieden werden
- Compliance kann schnell und automatisiert durchgesetzt werden

Nachteile der statischen Codeanalyse:

- False Positives können entstehen, dies könnte gerade bei einer großen Codebase problematisch sein
- Viele Änderungen der Compliance oder der Definition von Sicherheitslücken erfordern ebenfalls ein ständiges warten der Regeln
- Die statische Codeanalyse kann nur den Code analysieren, der zum Zeitpunkt der Analyse verfügbar ist, das Laufzeitverhalten oder externe Faktoren können nicht berücksichtigt werden.

Unterschiede

ESLint und Semgrep sind beides Werkzeuge, die zur Analyse und zum Aufspüren von Problemen im Code verwendet werden, aber sie haben einige wichtige Unterschiede:

Zweck:

ESLint ist ein Werkzeug, welches den Code mehr in Richtung der Einhaltung von Compliance und Best-Practices prüft, während Semgrep sich eher im Allgemeinen auf Sicherheitslücken spezialisiert.

Regel-Format:

ESLint verwendet JavaScript zur Implementation der Regeln, während Semgrep eine eigene Regelsyntax nutzt, welche im yaml-Format geschrieben wird.

Flexibilität der Regeln:

Da ESLint JavaScript zur Definition seiner Regeln verwendet, sind komplexere und dynamischere Regeln möglich, die aber auch schwieriger zu schreiben und zu pflegen sind. Auf der anderen Seite ist die Regelsprache von Semgrep einfacher, aber auch weniger "mächtig", so dass sich das Schreiben und Warten von Regeln oftmals einfacher gestaltet. Hier muss natürlich auch erwähnt werden, dass dies sehr abhängig vom Usecase ist.

Skalierung:

ESLint kann mit großen Codebasen umgehen, aber seine Leistung kann mit zunehmender Anzahl von Regeln und deren Komplexität abnehmen. Semgrep ist für große Codebasen ausgelegt und kann den Code meist schneller analysieren als ESLint.

Integration:

ESLint kann in andere Tools wie Texteditoren oder IDEs wie z.B. VSCode integriert werden und verfügt über eine große Anzahl von Plugins, die zusätzliche Funktionen bieten. Semgrep hingegen ist flexibler in Bezug auf die Integration und kann mit vielen Programmiersprachen unabhängig von der Entwicklungsumgebung verwendet werden.

Zusammenfassend lässt sich sagen, dass ESLint sich mehr auf Code-Stil und Konventionen konzentriert, während Semgrep auf Sicherheit und Fehlererkennung ausgerichtet ist. Je nach Anwendungsfall kann das eine Tool besser geeignet sein als das andere. Beide können zusammen verwendet werden, um sowohl den Code-Stil als auch die Sicherheit zu überprüfen.

Fazit

Zusammenfassend lässt sich sagen, dass die statische Codeanalyse ein wertvolles Werkzeug für Entwickler ist, um Sicherheitslücken, Fehler und andere Probleme in ihrem Code zu erkennen und zu vermeiden. Sie kann verwendet werden, um Compliance und die Einhaltung von Best Practices zu erzwingen. Eine der gebräuchlichsten Methoden, dies zu tun, ist der Einsatz spezialisierter Tools wie Semgrep oder ESLint. Die statische Codeanalyse ist ein wichtiger Schritt im Softwareentwicklungsprozess und ist heutzutage in vielen Entwicklungsworkflows integriert. Allgemein ist es die gängigste Methode, um automatisiert einen gewissen Standard von Codequalität und Sicherheit zu gewährleisten.

Github

Als Teil des Projektes wurde ebenfalls ein Github Repository angelegt, in welchem die Installation, Konfiguration und Implementierung der Regeln von ESLint und Semgrep, so wie sie auch hier in den Beispielen gezeigt wurden, erklärt werden.

<https://github.com/schromik/StaticCodeAnalysis>

Abbildungsverzeichnis

Abbildung 1: Gefundene Sicherheitslücke in WebGoat mittels GitLab DAST (ZAP)

Abbildung 2: SQL Injection in WebGoat

Abbildung 3: Darstellung von JavaScript Code mittels eines AST

Abbildung 4: Ablauf/Schritte vom Source Code zum Abstract Syntax Tree

Abbildung 5: JavaScript Code, Return-Problematik

Abbildung 6: Quellcode in AST-Ansicht mittels <https://astexplorer.net/>

Abbildung 7: Regeldefinition in ESLint

Abbildung 8: Funktion der implementierten Regel in ESLint

Abbildung 9: Python Code, Semgrep Äquivalenzerkennung

Abbildung 10: Einfache Semgrep Regel für eval-Funktion

Abbildung 11: Implementation der try/catch Regel in Semgrep

Abbildung 12: Pipeline Skript für ESLint

Abbildung 13: Pipeline Skript für Semgrep

Abbildung 14: Fehlerhafter Code der zum Abbruch der Pipeline führt

Abbildung 15: Durchführung des Semgrep Workflows

Abbildung 16: Durchführung des ESLint Workflows

Quellen

[1] Dokumentation ZAP, Zuletzt aufgerufen am 29.01.23 unter:

<https://www.zaproxy.org/getting-started/>

[2] WebGoat, Zuletzt aufgerufen am 29.01.23 unter: <https://owasp.org/www-project-webgoat/>

[3] Ursprung SCA, Zuletzt aufgerufen am 29.01.23 unter:

[https://de.wikipedia.org/wiki/Lint_\(Programmierwerkzeug\)](https://de.wikipedia.org/wiki/Lint_(Programmierwerkzeug))

[4] Static Code Analysis, Christof Ebert, S. 1, Zuletzt aufgerufen am 29.02.23 unter:

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1657940#>

[5] Abstract Syntax Tree: Zuletzt aufgerufen am 29.01.23 unter:

https://en.wikipedia.org/wiki/Abstract_syntax_tree