UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RENAN ROCHA SOUTO DOS SANTOS

ANÁLISE DE CÓDIGO COM CODEQL PARA DESCOBERTA DE VULNERABILIDADES EM APLICAÇÕES WEB

FLORIANÓPOLIS
2021

Renan Rocha Souto dos Santos

Análise de código com CodeQL para descoberta de vulnerabilidades em aplicações web

Trabalho de conclusão de curso do Curso de Graduação em Ciência da Computação do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciência da Computação.
Orientadora: Profa. Dra. Carla Markle Westphall

Florianópolis
2021

Renan Rocha Souto dos Santos

Análise de código com CodeQL para descoberta de vulnerabilidades em aplicações web

Este trabalho de conclusão de curso foi julgado adequado para obtenção do título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo Curso de Graduação em Ciência da Computação.

Florianópolis, 24 de setembro de 2021.

Banca Examinadora:

Profa. Dra. Carla Markle Westphall
Orientadora

Prof. Dr. Jean Everson Martina
Avaliador

Prof. Dr. Alex Sandro Roschildt Pinto
Avaliador

RESUMO

Devido ao fato que os projetos de código aberto serem frequentemente mantidos em repositórios Git, os provedores Git podem adicionar uma camada a mais no processo de atualização de código para ajudar os projetos a corrigirem vulnerabilidades de segurança de maneira automatizada antes dos códigos estarem em produção. O GitHub é um dos provedores Git mais populares utilizado por grandes organizações e fornece uma solução por meio do mecanismo CodeQL, que possui uma linguagem de consulta semelhante à sintaxe SQL e realiza análise de código a partir de consultas. Assim, com o CodeQL é possível escrever uma consulta para encontrar vulnerabilidades de segurança em códigos-fonte com base nos dados providos pelo usuário e funções sensíveis que levam a alguma vulnerabilidade, conhecido como análise de contaminação. Portanto, este trabalho tem como objetivo estudar todo o suporte do CodeQL e possivelmente contribuir com o projeto do GitHub com novas consultas para descobrir vulnerabilidades em aplicações web baseadas em JavaScript, estabelecidas nos 10 principais conjuntos de vulnerabilidades publicadas pela organização OWASP.

Palavras-chave: Análise de código, Segurança da Informação, Segurança em Aplicações Web

ABSTRACT

Due to the fact that open-source projects are often maintained in Git repositories, Git providers may add an extra layer to the code update process to help projects automatically fix security vulnerabilities before the codes are in production. GitHub is one of the most popular Git providers used by large organizations and provides a solution through the CodeQL engine, which has a query language similar to SQL syntax and performs code analysis from queries. Thus, with CodeQL is possible to write a query to find security vulnerabilities in source code based on data provided by the user and sensitive functions which lead to vulnerability, known as taint analysis. Therefore, this work aims to study all the CodeQL support and possibly contribute to the GitHub project with new queries to discover vulnerabilities in JavaScript-based web applications, established in the 10 main vulnerabilities sets published by the OWASP organization.

Keywords: Code Analysis, Information Security, Web Application Security

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

OWASP	Open Web Application Security Project
HTML	HyperText Markup Language
AST	Abstract Syntax Tree
CFG	Control Flow Graph
SAST	Static application security testing
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
SGBD	Sistema de gerenciamento de banco de dados
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure

SUMÁRIO

1 INTRODUÇÃO

As análises de códigos são comumente utilizadas no processo de desenvolvimento e atualização de *softwares* para apresentar possíveis erros que podem levar a um problema de execução, ou até mesmo alguma vulnerabilidade de segurança que possa ser explorada por um usuário malicioso. Estas análises podem ser realizadas através de especialistas em análise de código ou por *softwares* que realizam este processo de forma automatizada. Nos softwares, as análises podem ser realizadas de forma estática, baseando-se somente no código-fonte ou de forma dinâmica, baseando-se na aplicação em execução [19]. O resultado das análises pode apresentar especificamente em qual área do código ocorre um erro ou apresentar partes sensíveis do código para que futuramente especialistas possam analisar e validar o problema. Isto pode ocorrer em casos complexos em que *softwares* não são capazes de distinguir se um erro realmente está ocorrendo; como as análises de segurança, que através do fluxo de execução podem ter mitigadores sendo executados antes de uma área de código vulnerável, o que resulta em um código não vulnerável.

Os *softwares* de análises de código para encontrar problemas de segurança cada vez mais estão sendo necessários devido à imensa quantidade de códigos sendo criados e atualizados. Assim, um processo automatizado ajuda a solucionar as vulnerabilidades já conhecidas e otimizam o tempo em que um especialista poderia demorar para encontrar as falhas, ou possivelmente as falhas poderiam não ser encontradas. Várias soluções são lançadas todos os anos apresentando novas funcionalidades e diferentes abordagens para melhorar a eficácia e eficiência das análises e, dentre as diversas soluções públicas e privadas existentes [18], o CodeQL é um dos mecanismos possíveis para realizar análise de código estática em códigos abertos, que está atualmente integrado nos repositórios do GitHub. Esta solução propõe a possibilidade de realizar análise de código baseado em consultas, similarmente às consultas feitas na linguagem SQL em banco de dados relacionais. Assim, é possível modelar um tipo de vulnerabilidade já conhecida e suas variantes através da escrita de uma consulta, e portanto utilizá-la para encontrar vulnerabilidades em qualquer projeto de código aberto.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo deste trabalho é realizar o estudo do mecanismo CodeQL, apresentando a maior parte do seu suporte para análise de códigos na linguagem JavaScript. E com base neste estudo, desenvolver a escrita de consultas para encontrar vulnerabilidades de segurança em aplicações web.

1.1.2 Objetivos Específicos

Os objetivos específicos para este trabalho consistem em:

- Realizar o estudo e a descrição do CodeQL baseada na documentação oficial.
- Realizar a descrição das vulnerabilidades web baseada na lista dos dez conjuntos de vulnerabilidades mais recorrentes publicadas pela organização OWASP.
- A partir dos conjuntos de vulnerabilidades mais recorrentes e mais críticos, desenvolver consultas e apresentar os resultados em bibliotecas do NPM que podem tornar as aplicações web vulneráveis, disponíveis em repositórios do GitHub.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está dividido em 9 capítulos, incluído o capítulo introdutório. O capítulo 2 apresenta a primeira parte da fundamentação teórica, abordando os conceitos sobre revisão e análise de código. O capítulo 3 apresenta a segunda parte da fundamentação teórica, descrevendo os dez conjuntos de vulnerabilidades mais recorrentes em aplicações web. O capítulo 4 apresenta a terceira parte da fundamentação teórica, abordando a descrição do CodeQL. O Capítulo 5 apresenta a quarta parte da fundamentação teórica, detalhando o suporte do CodeQL para análise de código em aplicações desenvolvidas na linguagem JavaScript. O capítulo 6 apresenta os trabalhos relacionados. O capítulo 7 apresenta a implementação e os resultados obtidos a partir das consultas desenvolvidas para o CodeQL. E por fim, o capítulo 8 apresenta a conclusão deste trabalho.

2 REVISÃO E ANÁLISE DE CÓDIGO

A revisão de código é uma das técnicas existentes para identificar falhas de segurança no início do ciclo de vida de desenvolvimento das aplicações [4]. Quando aplicada junto aos testes de intrusão automatizados e manuais, a revisão de código pode aumentar significativamente a eficácia de custo de esforços de verificação de segurança nas aplicações [4]. Entretanto, com a crescente complexidade das aplicações e novas tecnologias sendo criadas, a forma tradicional de teste pode falhar em detectar todas as falhas de segurança presentes nas aplicações, pois é preciso compreender o código da aplicação, os componentes externos e as configurações para apresentar maior probabilidade de encontrar e evitar as falhas [4]; o que em alguns casos a complexidade pode ser grande.

A MITER — corporação que cataloga todas as vulnerabilidades públicas existentes em diferentes tipos de aplicações — já catalogou cerca de 1000 tipos diferentes de vulnerabilidades de *software* no projeto *Common Weakness Enumeration* (CWE) [4], que representa a categorização das falhas baseadas em sua criticidade. Todas estas vulnerabilidades são maneiras diferentes pelas quais os desenvolvedores de *software* podem cometer erros que levam à insegurança, em que cada uma destas falhas podem ser sutis e muitas são difíceis de compreender [17, 20, 22]. Contudo, estes problemas tornaram-se muito importantes devido ao fato que a capacidade de criar novas tecnologias ultrapassa a capacidade de protegê-las [4].

Os *softwares* de análises de código estão altamente relacionados com a revisão de código. O objetivo destes *softwares* é prover de forma automatizada análises diretamente relacionadas ao código-fonte de uma aplicação, destacando possíveis vulnerabilidades em determinadas áreas do código. Embora os *scripts* de varredura de código possam ser eficientes para encontrar padrões de código inseguros, geralmente não possuem a capacidade de rastrear o fluxo de dados de forma eficiente. Esta lacuna é preenchida pelos analisadores de código estáticos — também conhecidos como SAST, que identificam os padrões de código inseguros compilando parcialmente (ou totalmente) o código-fonte. As análises estáticas de código podem ser realizadas por diferentes técnicas, tais como:

- Análise de fluxo de dados: É utilizada para coletar informações em tempo de execução (dinâmicas) sobre os dados no *software* enquanto ele está em um estado estático.
- Grafo de fluxo de controle: É uma representação em grafos da forma abstrata do *software* pelo uso de nós que representam blocos básicos. Um nó em um grafo representa um bloco e as arestas direcionadas são utilizadas para representar os caminhos de um bloco para outro.
- Análise de contaminação: Identifica variáveis que podem ser contaminadas com entradas controláveis pelo usuário (*sources*) e as rastreia para possíveis funções vulneráveis (*sinks*). Se algum dado de entrada contaminado for encaminhado para alguma função vulnerável sem primeiro ser filtrado, é sinalizado como uma vulnerabilidade.

- Análise léxica: Converte a sintaxe do código-fonte em *tokens* de informação em uma tentativa de abstrair o código-fonte e torná-lo mais fácil de manipular.

Os analisadores de código estático são opções abrangentes para complementar o processo de revisão de código, apresentando algumas vantagens:

- Redução do trabalho manual: O tipo de padrões a serem verificados permanece comum entre os aplicativos. Neste cenário, os *scanners* desempenham um grande papel na automação do processo de pesquisa de vulnerabilidades por meio de grandes bases de código.
- Encontrar todas as instâncias de uma vulnerabilidade: Os *scanners* são muito eficazes na identificação de todas as instâncias de uma vulnerabilidade em particular com sua localização exata. Isto é útil para uma base de código maior, em que o rastreamento de falhas em todos os arquivos é difícil de realizar.
- Elaboração de relatórios: Os *scanners* fornecem um relatório detalhado sobre as vulnerabilidades observadas com trechos de código exatos, classificação de risco e descrição completa das vulnerabilidades. Isto ajuda as equipes de desenvolvimento a compreenderem facilmente as falhas e implementarem os controles necessários.

3 VULNERABILIDADES EM APLICAÇÕES WEB

Os *softwares* inseguros podem causar problemas de forma direta ou indireta a infraestrutura financeira, de saúde, defesa, energia e outras infraestruturas críticas. À medida que os *softwares* estão se tornando cada vez mais críticos, complexos e interligados, a dificuldade em garantir a segurança cresce exponencialmente [12]. O ritmo acelerado dos processos modernos de desenvolvimento de *softwares* faz com que seja ainda mais crítico identificar estes riscos de forma rápida e precisa [12]. Assim, a organização OWASP é responsável por publicar as dez categorias de vulnerabilidades mais recorrentes e críticas dos últimos anos através da OWASP Top 10. Portanto, o objetivo deste capítulo é descrever sucintamente estas dez categorias de vulnerabilidades baseadas na publicação de 2017.

3.1 A1 INJEÇÃO

As falhas de injeção ocorrem quando dados não confiáveis são enviados para um interpretador como parte de um comando ou consulta. A consequência da injeção dos dados providos pelo atacante é fazer o interpretador executar comandos não pretendidos.

Dentro deste escopo existem diversos tipos de vulnerabilidades associadas. A injeção de SQL ocorre quando o atacante consegue manipular uma consulta SQL, o que implica em todos os dados contidos no banco de dados poderem ser manipulados ou extraídos. O exemplo mais comum desta falha ocorre quando as aplicações utilizam os dados providos pelo usuário diretamente na consulta a ser interpretada pelos SGBDs em forma de concatenação de *string*. O código 3.1 ilustra um exemplo de aplicação que utiliza as constantes *user* e *password* — que representam dados providos pelo usuário — para verificar se este usuário existe no banco de dados com as credenciais fornecidas. Neste exemplo é ilustrado como um atacante poderia se autenticar como um usuário que possui privilégios de administrador, realizando o fechamento das aspas simples e tornando a condição booleana de *password* sempre verdadeira através de *'1'='1'*. Neste caso, o atacante poderia se autenticar como o usuário *admin* sem ter o conhecimento real da senha, como pode ser visto na consulta que seria executada no código 3.2.

```
1 const user = "admin";
2 const password = "' or '1'='1";
3
4 execQuery("select user from users where user='" + user "' and (password='" +
    password + "')");
```

Código 3.1 – Exemplo de injeção de SQL

```
1 select user from users where user='admin' and (password='' or '1'='1');
```

Código 3.2 – Exemplo de injeção de SQL 2

Outras vulnerabilidades bastantes comuns são as injeções de comandos do sistema operacional e as injeções de código. O código 3.3 ilustra um exemplo de vulnerabilidade de injeção de comando através da biblioteca *child_process* do *runtime* Node.js. Para este exemplo, o usuário malicioso cria um arquivo nomeado por *file* no sistema operacional que está executando a aplicação vulnerável.

```
1 const { execSync } = require('child_process');
2 const user = "test;touch file";
3 console.log(execSync('id ' + user));
```

Código 3.3 – Exemplo de injeção de comandos no sistema operacional

Descrição do código 3.3:

- Linha 1: Importação da função *execSync* da biblioteca *child_process* para possibilitar a execução de comandos no sistema operacional de forma síncrona.
- Linha 2: Constante *user* representando um usuário fornecido por um usuário qualquer.
- Linha 3: Realiza a execução do comando *id* para verificar se o usuário existe no sistema operacional. Posteriormente, utiliza o comando *console.log* para imprimir o resultado na tela.

O ponto principal na injeção deste tipo de exploração acontece devido ao fato que o atacante pode realizar a concatenação de comandos. Assim, o conteúdo de *id* do código 3.3 mostra a concatenação realizada através do símbolo *;* e posteriormente a execução do comando *touch*, que realiza a criação de um arquivo. Contudo, neste cenário o atacante poderia inserir qualquer tipo de comando do sistema operacional, tornando o sistema completamente vulnerável.

A vulnerabilidade de injeção de código acontece similarmente à vulnerabilidade de injeção de comando, entretanto ao invés de executar um comando diretamente no sistema operacional, um código é interpretado. O exemplo de código 3.4 ilustra a constante *data* sendo executada como argumento da chamada da função *console.log*, implicando na impressão do conteúdo da constante *data* na tela. Nota-se que este comportamento está sendo executado pela função *eval*, que possibilita executar código JavaScript através de uma *string*.

```
1 const data = "test";
2 eval('console.log("' + data + '")');
```

Código 3.4 – Exemplo de injeção de código

Neste exemplo, supondo que o atacante tem acesso a esta constante *data*, é possível realizar também uma concatenação de comando, mas neste caso o atacante teria somente a possibilidade de executar códigos JavaScript a princípio. Entretanto, em alguns casos é possível

importar a biblioteca *child_process* diretamente e executar comandos no sistema operacional, conforme ilustrado no código 3.5.

```
1 const data = 'test"); require("child_process").execSync("touch file") //';
2 eval('console.log("' + data + '")');
```

Código 3.5 – Exemplo de injeção de código 2

3.2 A2 QUEBRA DE AUTENTICAÇÃO

Esta categoria é composta por vulnerabilidades que podem variar dependendo da aplicação. Quando uma aplicação possui algum processo de autenticação, os problemas podem ocorrer de diversas maneiras, pois a quebra de autenticação ocorre por algum erro de implementação ou regra de negócio. Um dos ataques comuns neste contexto é o ataque de força bruta [12], em que o atacante possui dicionários de senhas previamente coletadas geralmente através de vazamentos de dados públicos, e então realiza tentativas exaustivas para se autenticar. Neste caso, as aplicações comumente implementam algumas políticas para bloquear IPs que realizam várias tentativas em um curto período de tempo, criando assim uma lista de IPs para a qual deve-se impedir a autenticação. Atualmente também são muito utilizados os serviços de CAPTCHA [4], que obrigam os usuários a completar tarefas dinâmicas para que o usuário possa provar que não é um *software* automatizado.

3.3 A3 EXPOSIÇÃO DE DADOS SENSÍVEIS

As exposições de dados sensíveis geralmente ocorrem através de más configurações [12]. Muitos ataques ocorrem por meio de atacantes interceptando ou tentando realizar algum tipo de exploração relacionado a criptografia, em que são utilizados modelos criptográficos não eficientes contra ataques de força bruta. Este tipo de exploração pode ocorrer quando um usuário acessa uma página HTTP, em que os dados trafegam na rede de forma não cifrada. Assim, um atacante que tenha acesso à mesma rede, como os casos comuns de WIFIs públicos, pode realizar a captura dos pacotes de rede e ter acesso a todo o conteúdo trafegado pelos usuários em que não há criptografia envolvida. Portanto, quando algum usuário realiza algum tipo de autenticação em algum *site* que não possui HTTPS, o atacante possui acesso às credenciais de forma explícita.

3.4 A4 ENTIDADES EXTERNAS DE XML (XXE)

Esta vulnerabilidade ocorre quando os atacantes podem abusar de processadores XML vulneráveis [12]. Em aplicações vulneráveis que permitem enviar arquivos XML e serem processados posteriormente no lado servidor, os atacantes podem incluir algum conteúdo

malicioso no arquivo XML. Em alguns casos, o impacto desta vulnerabilidade pode levar a tipos de explorações críticas, como a execução remota de código no servidor. A exploração mais comum é quando o atacante pode ler conteúdos arbitrários do servidor em que está hospedando a aplicação, através da definição de uma *entity*, como ilustrado no código 3.6. Neste exemplo, supondo que o conteúdo de *content* é mostrado na interface de uma aplicação web, o atacante poderia definir que o conteúdo seria justamente o conteúdo do arquivo */etc/passwd*. Consequentemente, o atacante poderia ler o conteúdo de qualquer arquivo em que o usuário utilizado pelo servidor tem acesso, incluindo o código-fonte da aplicação.

```
1 <!--?xml version="1.0" ?-->
2 <!DOCTYPE replace [<!ENTITY example "file:///etc/passwd"> ]>
3  <test>
4   <content>&example;</content>
5  </test>
```

Código 3.6 – Exemplo de arquivo XML para exploração de XXE

3.5 A5 QUEBRA DE CONTROLE DE ACESSOS

A quebra de controle de acessos ocorre quando um atacante pode executar ações em que um usuário comum não deveria ter acesso [12] . Esta falha pode ocorrer através de APIs que não implementam corretamente o controle de acesso. Da mesma maneira, pode ocorrer através dos *cookies* de autenticação, quando estão vulneráveis e podem sofrer modificações nos metadados, como o caso das vulnerabilidades associadas ao JSON Web Token (JWT).

3.6 A6 CONFIGURAÇÕES DE SEGURANÇA INCORRETAS

Esta vulnerabilidade ocorre quando são realizadas as configurações dos serviços que compõem o servidor de forma incorreta, em termos de segurança [12]. Em aplicações, os servidores comumente são utilizados com configurações padrões e podem levar a algum tipo de exploração. Por exemplo, deixar habilitado a listagem de diretórios. Quando um usuário qualquer acessa um *endpoint* que não possui um arquivo *index.html*, automaticamente são indexados todos os arquivos que pertencem a aquele *endpoint*. Caso o *endpoint* possua algum arquivo sensível, os usuários possuem acesso facilitado. Da mesma forma ocorre quando os desenvolvedores esquecem de excluir arquivos de testes que possuem alguma informação sensível e estão expostos aos usuários.

3.7 A7 CROSS-SITE SCRIPTING (XSS)

Esta vulnerabilidade ocorre quando um atacante é capaz de inserir alguma *tag* HTML ou *tag script* na página web vulnerável, levando à execução de códigos JavaScript [12]. Se o atacante possuir acesso a este tipo de vulnerabilidade, pode-se executar ações como se fosse a própria vítima ou até obter informações sigilosas do usuário. Em alguns casos pode-se obter os *cookies* de sessão quando o *httponly* não está habilitado. Na primeira linha do código 3.7 é ilustrado uma tag script, que ao ser executada por uma página web, executaria códigos JavaScript. Na segunda linha a execução ocorre através de um evento de erro. Como a *tag img* não possui uma URL definida em *src* para ser carregada, automaticamente é gerado um erro e, portanto, é executado o conteúdo JavaScript que estiver definido em *onerror*.

```
1 <script>...</script>
2 <img src onerror="..." />
```

Código 3.7 – Exemplos de exploração para XSS

Esta vulnerabilidade pode ser categorizada em alguns tipos, tais como:

- Reflected XSS: A exploração depende da vítima acessar uma URL específica, em que a exploração não persiste na página vulnerável.
- Stored XSS: A exploração persiste na página. Por exemplo, quando um atacante realiza um comentário em um blog e este comentário pode conter *tags* que executam JavaScript. Assim, quando qualquer usuário acessa a página que contém o comentário malicioso, automaticamente ela está vulnerável.
- DOM XSS: Ocorre quando uma aplicação utiliza funções como *innerHTML* baseado em um conteúdo provido por um usuário. Portanto, as *tags* HTMLs são renderizadas.

3.8 A8 DESSERIALIZAÇÃO INSEGURA

Os objetos serializáveis podem ser utilizados em aplicações web para converter a representação de um objeto em uma linguagem de programação qualquer, e assim enviá-lo através de requisições HTTP, ou até mesmo guardar o conteúdo em um arquivo. A desserialização insegura ocorre quando as aplicações web utilizam de alguma forma objetos serializáveis fornecidos pelo usuário de forma insegura [12]. Quando o atacante possui acesso ao envio de um objeto serializado para a aplicação ou de alguma forma o atacante consegue manipular um objeto serializado, pode ser possível realizar a exploração. Quando um objeto serializado é enviado, a aplicação deve executar uma função *unserialize* para converter o objeto serializado em um objeto da linguagem utilizada. O exemplo mais comum desta exploração na linguagem PHP é quando a aplicação utiliza algum dos métodos mágicos [10], como *__destruct*, que é um método

executado automaticamente para realizar a destruição de um objeto serializado. Assim, quando o atacante pode manipular o objeto serializado antes de enviar para a aplicação, pode-se definir qual será o conteúdo executado por *__destruct*. E como *__destruct* é executado automaticamente no fim da execução, esta vulnerabilidade pode levar à execução remota de código.

3.9 A9 UTILIZAÇÃO DE COMPONENTES VULNERÁVEIS

Esta categoria representa as vulnerabilidades publicamente divulgadas que não são corrigidas pelas aplicações web em produção. Estas vulnerabilidades ocorrem quando uma aplicação web utiliza alguma biblioteca ou algum serviço de código aberto que possui alguma vulnerabilidade em alguma determinada versão. Quando estas vulnerabilidades ocorrem, todas as aplicações devem ser atualizadas para receber a versão mais recente do serviço com as devidas correções. Estas vulnerabilidades são comumente alertadas através de *softwares* de análise de vulnerabilidades que utilizam como base as vulnerabilidades catalogadas através das CVEs, como o *software* OpenVAS, Nessus, entre outros. Portanto, quando as aplicações web possuem em seu processo de desenvolvimento algum destes *softwares* automatizados de forma integrada, as aplicações podem ser alertadas e receberem as correções o mais cedo possível.

3.10 A10 REGISTRO E MONITORAÇÃO INSUFICIENTE

Esta categoria descreve o registro e a monitoração insuficiente pelas aplicações web. Em muitos casos, a exploração de vulnerabilidades associadas ao servidor podem ser identificadas previamente através de registro e monitoração. A monitoração pode ocorrer através de registros de acesso às requisições HTTP realizadas pelos usuários, registros de acesso à APIs, e etc. Portanto, a falta de registro e monitoração não implica em uma vulnerabilidade específica, mas a falta deste mecanismo facilita a exploração das vulnerabilidades, caso elas ocorram.

4 CODEQL

O CodeQL é um projeto de código aberto coordenado pelo GitHub que atua como um mecanismo para automatizar verificações de segurança por pesquisadores para realizar análises de variantes de vulnerabilidades existentes via linguagem de consulta [1]. No CodeQL as vulnerabilidades de segurança são modeladas como consultas que podem ser executadas em bancos de dados extraídos do código-fonte [1]. O mecanismo é composto pela funcionalidade de gerar os bancos de dados a partir de códigos e por um conjunto de bibliotecas que tornam possíveis as análises de variantes através de consultas feitas com o suporte da linguagem QL. Assim, o fluxo de uma análise acontece da seguinte forma:

1. Criação do banco de dados baseado no código.
2. Execução de consultas no banco de dados.
3. Interpretação dos resultados da consulta.

O quaro 1 apresenta as linguagens suportadas pelo CodeQL.

Quadro 1 – Linguagens suportadas

Linguagem	**Variantes**	**Compiladores**	**Extensões**
C/C++	C89, C99, C11, C18, C++98, C++03, C++11, C++14, C++17	Clang (and clang-cl [1]) extensions (up to Clang 9.0), GNU extensions (up to GCC 9.2), Microsoft extensions (up to VS 2019), Arm Compiler 5 [2]	.cpp, .c++, .cxx, .hpp, .hh, .h++, .hxx, .c, .cc, .h
C#	C# up to 8.0	Microsoft Visual Studio up to 2019 with .NET up to 4.8, .NET Core up to 3.1	.sln, .csproj, .cs, .cshtml, .xaml
Go (aka Golang)	Go up to 1.15	Go 1.11 or more recent	.go
Java	Java 7 to 15	javac (OpenJDK and Oracle JDK), Eclipse compiler for Java (ECJ) [4]	.java
Javascript	ECMAScript 2019 or lower	Not applicable	.js, .jsx, .mjs, .es, .es6, .htm, .html, .xhm, .xhtml, .vue, .json, .yaml, .yml, .raml, .xml
Python	2.7, 3.5, 3.6, 3.7, 3.8	Not applicable	.py
TypeScript	2.6-3.7	Standard TypeScript compiler	.ts, .tsx

4.1 BANCO DE DADOS CODEQL

As análises de código com CodeQL através das consultas feitas em linguagem QL necessitam de bancos de dados gerados a partir dos códigos de origem. Portanto, é extraída primeiro uma única representação relacional de cada arquivo de origem na base de código [1]. Para linguagens compiladas, a extração funciona monitorando o processo normal de construção. Cada vez que um compilador é chamado para processar um arquivo, é feita uma cópia desse arquivo e todas as informações relevantes sobre o código-fonte são coletadas [1]. Isto inclui dados sintáticos sobre a árvore de sintaxe abstrata, e dados semânticos sobre a vinculação de nomes e informações de tipagem [1]. E para linguagens interpretadas, o extrator é executado diretamente no código-fonte, resolvendo as dependências para fornecer uma representação precisa da base de código.

No processo de extração cada linguagem suportada pelo CodeQL possui seu extrator específico para garantir que o processo seja o mais preciso possível. Para bases de código em várias linguagens de programação, os bancos de dados são gerados em uma linguagem por vez [1]. Ao final da extração é criado um único diretório que contém todos os arquivos necessários para futuras análises, e este diretório é referenciado como banco de dados CodeQL [1]. Portanto, o diretório irá conter os dados relacionais, os arquivos de origem copiados e um esquema que especifica as relações mútuas nos dados específicos da linguagem de programação.

As bibliotecas CodeQL definem para cada linguagem as classes para fornecerem uma camada de abstração sobre as tabelas de banco de dados. Isso fornece uma visão dos dados orientados a objetos, o que torna mais fácil escrever consultas. Por exemplo, em um banco de dados CodeQL para um programa em linguagem JavaScript, as duas tabelas principais são:

- A tabela de expressões contendo uma linha para cada expressão no código-fonte que foi analisada durante o processo de construção.
- A tabela de instruções contendo uma linha para cada instrução no código-fonte que foi analisada durante o processo de construção.

4.2 VISÃO GERAL SOBRE A LINGUAGEM QL

QL é uma linguagem de programação lógica orientada a objetos declarativa baseada no modelo semântico do Datalog, com o objetivo principal de realizar consultas complexas, sendo potencialmente estruturas de dados recursivas codificadas em um modelo de dados relacional [14]. Além de ser uma linguagem de propósito geral, ela possui um grande suporte a predicados recursivos e agregadores, o que a torna particularmente adequada para a implementação de análise estática de código, consultas de código e métricas de *software* [2]. Por exemplo, considerando um banco de dados contendo relacionamentos pai-filho para pessoas e o objetivo é encontrar o número de descendentes de uma pessoa, o algoritmo poderia ser da seguinte forma:

- Encontrar um descendente de uma determinada pessoa, ou seja, um filho ou descendente de uma criança.
- Contar o número de descendentes encontrados usando a etapa anterior.

O algoritmo em linguagem QL usa a recursão para encontrar todos os descendentes de uma determinada pessoa e um agregador para contar o número de descendentes. A tradução dessas etapas é possível devido à natureza declarativa da linguagem, como apresentado no código 4.1. Descrição do código:

- *getADescendant*: Obtém um descendente baseado em uma determinada pessoa.
- *getNumberOfDescendants*: Conta a quantidade de descendentes baseado em cada descendente encontrado pela função *getADescendant*.

```
1 Person getADescendant(Person p) {
2   result = p.getAChild() or
3   result = getADescendant(p.getAChild())
4 }
5
6 int getNumberOfDescendants(Person p) {
7   result = count(getADescendant(p))
8 }
```

Código 4.1 – Algoritmo descendentes

Apesar do código 4.1 não ser suficiente para demonstrar todo o suporte ao paradigma de orientação a objetos, a linguagem QL oferece todos os benefícios (reutilização de código, por exemplo) sem comprometer sua base lógica, definindo um modelo de objeto simples em que as classes são modeladas como predicados e as heranças como implicação [2]. Portanto, as bibliotecas disponibilizadas para todas as linguagens suportadas podem fazer o uso extensivo de classes e heranças.

4.2.1 Predicados

Os predicados são utilizados para descrever as relações lógicas que constituem um programa QL, sendo representado por um conjunto de tuplas [21].

```
1 predicate isCountry(string country) {
2   country = "Brazil" or country = "India" or country = "Russia"
3 }
```

Código 4.2 – Predicado sem retorno

O código 4.2 define um predicado que verifica se a partir do nome de um país informado no parâmetro *country* é igual a *Brazil*, *India* ou *Russia*. Consequentemente, o predicado isCountry é o conjunto de uma tupla ("*Brazil*"), ("*India*"), ("*Russia*"). Este exemplo ilustra a definição de um predicado que não retorna um resultado, entretanto, é possível definir um predicado que retorna algum tipo de resultado.

```
1 int getSuccessor(int i) {
2   i in [1 .. 5] and
3   result = i + 1
4 }
```

Código 4.3 – Predicado com retorno

O código 4.3 representa o predicado que retorna o número sucessor entre 1 e 5 baseado no parâmetro informado. Também é possível expressar a relação entre o resultado e outras variáveis. O código 4.4 representa o mesmo algoritmo, porém o resultado escrito por extenso.

```
1  string getSuccessor(int i) {
2    i in [1 .. 5] and result = getString(i + 1) or
3    result = "Out of range"
4  }
5
6  string getString(int i) {
7    i == 1 and result = "One" or
8    i == 2 and result = "Two" or
9    i == 3 and result = "Three" or
10   i == 4 and result = "Four" or
11   i == 5 and result = "Five"
12 }
```

Código 4.4 – Resultado relacionado com outras variáveis

Como apresentado no início desta seção, também há um tipo de predicado recursivo. Este é o tipo de predicado que possibilita consultas complexas com CodeQL e será explicado nas próximas subseções.

4.2.2 Tipos de variáveis e classes

As variáveis podem ter objetivos diferentes. Algumas variáveis são livres e seus valores afetam diretamente o valor de uma expressão que as usa ou uma determinada fórmula [14, 21]. Outras variáveis são chamadas de variáveis ligadas, sendo restritas a conjuntos de valores específicos [14, 21]. No exemplo de código 4.5 a primeira linha representa a variável f como

variável ligada e na segunda linha a variável *i* como variável livre.

```
1 min(float f | f in [-3 .. 3]);
2 (i + 7) * 3;
```

Código 4.5 – Variável livre e variável ligada

Existem cinco tipos de variáveis primitivas. O tipo *boolean* para armazenar valores lógicos *true* ou *false*; o tipo *string* para armazenar textos; o tipo *int* para números inteiros; o tipo *float* para números racionais; e por fim, o tipo *date* para armazenar datas [21]. Apesar das linguagens de programação poderem ter sintaxes diferentes para declaração de variáveis e funções, quando gerado o banco de dados CodeQL as variáveis podem ser convertidas para uma representação universal através da palavra-chave *@variable* [21].

É possível definir novos tipos de dados algébricos através da palavra-chave *newtype*, o que permite a criação de estruturas de dados ao definir um tipo de dado e suas propriedades [21]. O exemplo de código 4.6 ilustra a criação de um tipo de dado que representa a chamada de um predicado ou de outro predicado, o que classifica como um tipo de dado com duas ramificações. Logo, o predicado *call2* só será executado caso o predicado *call1* não seja executado.

```
1 newtype OptionCall = call1(Call c) or call2()
```

Código 4.6 – Criação de tipo de dado algébrico

De acordo com o suporte à implementação de códigos orientados a objetos na linguagem QL, é possível definir novos tipos de objetos através de novas classes. Apesar do conceito de classe ser similar a outras linguagens, em QL não há a criação de fato de um objeto a partir de uma classe, e sim a criação de uma propriedade lógica [21]. O exemplo de código 4.7 ilustra a definição de uma classe chamada *OneTwoThree*, que possui o predicado característico — conhecido como método construtor em outras linguagens — para a definição do comportamento do tipo a ser criado. Esta classe possui a palavra-chave *extends*, o que ilustra outro conceito chamado de Herança (única ou múltipla) utilizado no paradigma de orientação a objetos. Entretanto, na linguagem QL toda definição de nova classe deve obrigatoriamente ser herdada de outro tipo existente [21]. Contudo, o código 4.7 define que o tipo do objeto a ser criado deve ser igual a 1, 2, ou 3; e que herda todos os predicados definidos pela classe *int*.

```
1 class OneTwoThree extends int {
2   OneTwoThree() { this = 1 or this = 2 or this = 3 }
3 }
```

Código 4.7 – Criação de classe

As classes abstratas são definidas em QL como lógica de união de classes, podendo ser criadas classes abstratas para suprir a necessidade de agrupar diferentes classes em uma única, respeitando todos os predicados característicos da sua classe e das subclasses [14, 21]. Também é possível a criação de predicados que interagem com várias subclasses que compõem a mesma ou adicionar novos predicados em classes existentes. Contudo, as classes abstratas são definidas como o exemplo de código 4.8.

```
1 abstract class NewClass extends OldClass
```

Código 4.8 – Classe abstrata

Além do modo de definição de uma classe apresentado anteriormente, é possível usar a palavra-chave *newtype* para criar um novo tipo de dado para definir qual será o conteúdo de uma classe específica, sendo esta forma de implementação chamada de União de Tipos [21]. O tipo de dado que implementa o conteúdo da classe deve conter argumentos separados pelo operador lógico *or*. No exemplo 4.9, *Definition1* e *Definition2* são os possíveis conteúdos para a classe *DefiniteInitialization* e, dentro das mesmas, existe um operador *exists* que irá retornar se a definição em questão irá ou não ser o conteúdo da classe.

```
1 newtype InitialValueSource =
2   Definition1() { exists(...)} or
3   Definition2() { exists(...) }
4
5 class DefiniteInitialization = Definition1 or Definition2;
```

Código 4.9 – União de tipos

4.2.3 Expressões gerais

Uma expressão de intervalo denota um intervalo de valores ordenados entre duas expressões que representam valores de tipos de variáveis primitivas [7]. Consiste em duas expressões separadas como [a .. b]. Exemplo, [1 .. 5] é uma expressão que representa quaisquer números inteiros entre 1 e 5 (incluindo 1 e 5). Da mesma forma que a expressão de intervalo é possível definir um conjunto de valores expressos explicitamente sobre quais valores fazem parte do conjunto. Exemplo, [1, 3, 5] representa três valores explicitamente e a expressão anterior representa cinco valores implicitamente.

Devido ao conceito de herança definido pelo paradigma de orientação a objetos, na linguagem QL existe a expressão *super* que pode ser utilizada pelas subclasses para acessar propriedades da classe base [14, p.4–5]. Ainda neste contexto, é possível sobrescrever os predicados herdados das classes através da palavra-chave *override*. O exemplo de código 4.10

ilustra a situação onde as classes A e B são herdadas da classe *int* e que, por sua vez, a classe C é herdada das classes A e B conjuntamente. Como ambas as classes A e B possuem o predicado *getANumber* e a classe C herda os predicados de ambas, a diretiva *override* deve ser definida obrigatoriamente para enfatizar qual das duas implementações irá ser utilizada para não haver ambiguidade. Logo, o predicado C define que a implementação do predicado *getANumber* a ser utilizado é o predicado herdado pela classe B, definido através da expressão *super*.

```
1  class A extends int {
2    int getANumber() { result = 1 }
3  }
4
5  class B extends int {
6    int getANumber() { result = 2 }
7  }
8
9  class C extends A, B {
10   override int getANumber() {
11     result = B.super.getANumber()
12   }
13 }
```

Código 4.10 – Herança

Existem algumas operações e expressões complementares que podem ser utilizadas, como as operações unárias, operações binárias, expressão *don't care* e as expressões *casts* [7]. As Operações unárias são operações definidas através do sinal - ou + antes de uma expressão; o que implica na conversão de sinal de todos os elementos que fazem parte da expressão, sendo somente aplicável aos tipos *int* e *float*. As operações binárias são definidas através das operações de adição, multiplicação, divisão, subtração e módulo; e são aplicadas sobre uma expressão. A expressão *don't care* é escrita como um único sublinhado, com o objetivo de representar qualquer valor dentre os valores possíveis. As expressões *casts* consiste na conversão para restringir o tipo de uma expressão, semelhante aos *Casts* da linguagem Java; definido antes de uma expressão entre parênteses [7, 13].

4.2.4 Expressões de agregação e do tipo any

A agregação é um mapeamento que calcula um valor de resultado baseado em um conjunto de valores de entrada que são especificados por uma fórmula, sendo representada pela sintaxe do código 4.11. Na primeira parte da agregação são definidas as variáveis necessárias; na segunda parte uma fórmula que representa a lógica do que fará parte da agregação; e na terceira parte uma expressão que representa a ordem dos dados que fazem parte da agregação

[7, 13].

```
1 <aggregate>(<variable declarations> | <formula> | <expression>)
```

Código 4.11 – Sintaxe da agregação

Os tipos gerais de agregação são:

- count: Retorna a quantidade de elementos que fazem parte da fórmula definida.
- min: Retorna o elemento de menor valor dentre os elementos que fazem parte da fórmula definida.
- max: Retorna o elemento de maior valor dentre os elementos que fazem parte da fórmula definida.
- avg: Retorna o valor médio dentre os elementos que fazem parte da fórmula definida.
- sum: Retorna a soma total dentre os elementos que fazem parte da fórmula definida.
- unique: Retorna valores únicos dentre os elementos que satisfazem a fórmula definida. Portanto, se os valores possíveis forem 1,1,2,2,3,3 o retorno será igual a 1,2,3.
- rank: Classifica os elementos que fazem parte da fórmula definida e retorna um valor em uma determinada posição dentre os valores classificados. Neste caso, é necessário informar qual posição deve ser retornada entre colchetes. Exemplo: rank[...].
- concat: Retorna a concatenação de todos elementos que fazem parte da fórmula definida, sendo somente possível aplicar em variáveis do tipo string. Nesta agregação também é possível adicionar uma segunda expressão, o que implicará em uma agregação também.
- strictconcat, strictcount, strictsum: Estes três agregadores funcionam similarmente aos agregadores concat, count e sum, respectivamente. São utilizados de maneira mais restrita quando não há um valor que satisfaça a fórmula, retornando um valor vazio.

Os agregadores monotônicos são um tipo de agregação que possibilita a agregação de diferentes resultados de agregadores combinados através de tuplas de todos os possíveis valores, e quando combinados com o agregador de avaliação através da palavra-chave *exists*, é possível obter um único valor como resultado [7]. O exemplo de código 4.12 ilustra a agregação de dois sub-agregadores que resulta na soma dos valores possíveis para a variável *i*. Para os dois sub-agregadores é gerado uma tupla composta pelo primeiro valor representado pela variável *i* e o segundo pela variável *j*. Como o predicado *charAt* retorna um caractere baseado em uma posição informada, os possíveis valores para a variável *i* são 0, 1, 2, 3, 4 e para variável *j* são 0, 1, 2, 3, 4, 5. Portanto, é gerado todas as combinações entre os dois agregadores, resultando em

30 combinações. Assim, para cada valor de *i* da tupla o agregador *sum* irá realizar a soma.

```
sum(int i, int j |
    exists(string s | s = "hello".charAt(i)) and
    exists(string s | s = "world!".charAt(j))
| i)
```

Código 4.12 – Agregadores monotônicos

Os agregadores monotônicos também podem ser implementados com recursividade, definidos na parte da expressão [7]. O exemplo de código 4.13 ilustra um algoritmo para calcular a distância de um nó em um grafo a partir do resultado das folhas. Portanto, em cada nível de profundidade são analisados os nós que possuem suas folhas já calculadas.

```
int depth(Node n) {
  if not exists(n.getAChild())
  then result = 0
  else result = 1 + max(Node child | child = n.getAChild() | depth(child))
}
```

Código 4.13 – Agregador monotônico recursivo

A descrição a seguir apresenta como seria o fluxo de execução para a sequência de nós ilustrado pela figura 1.

- Nível 0: *b*, *d*, *e*. O nó *c* depende do nó *b* e do nó *d*. O nó *a* depende do nó *b* e do nó *c*.
- Nível 1: c. O nó *a* depende do nó *c*.
- Nível 2: *a*.

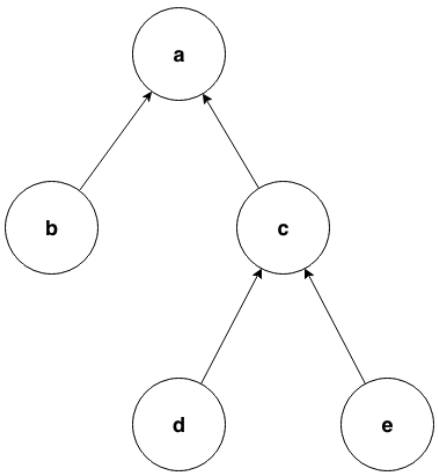

Figura 1 – Exemplo de grafo para agregador monotônico recursivo

A expressão do tipo *any* consiste na mesma sintaxe da expressão de agregação, entretanto com o objetivo de encapsular qualquer tipo de dado [7]. Assim, é possível adicionar expressões para selecionar valores satisfeitos pela fórmula. A primeira linha do exemplo de código 4.14 apresenta o conjunto dos nomes de todos os elementos. A segunda linha apresenta uma expressão de multiplicação para cada valor satisfeito pela fórmula.

```
1 any(Element e | e.getName());
2 any(int i | i = [0 .. 3] | i * i);
```

Código 4.14 – Exemplos de agregadores do tipo any

4.2.5 Fórmulas

Como apresentado na subseção anterior sobre os agregadores, as expressões são essenciais para definir o comportamento de um agregador e podem ser necessárias para outras situações. As fórmulas podem ser definidas para comparações lógicas, verificação de tipos, verificação de alcance, chamadas para predicados, fórmulas quantificadas e conectivos lógicos [8]. A sintaxe da definição de uma fórmula é expresso da seguinte forma:

```
1 <expression> <operator> <expression>
```

Código 4.15 – Sintaxe das fórmulas

As comparações lógicas são representadas e aplicadas similarmente as outras linguagens de programação. Os comparadores de ordem e de igualdade são expressos como: <, >, <=. >=, ==, !=; representando o valor semântico de menor, maior, menor ou igual, maior ou igual, igual e diferente; respectivamente [7].

A verificação de tipos é constantemente utilizada na análise de código, pois ela verifica a instância de um dado através da sintaxe ilustrada na primeira linha do código 4.16. Na segunda linha, é apresentada uma verificação se a variável *x* é do tipo *Person*.

O verificador de alcance verifica se dentro de um conjunto existe um determinado elemento, similarmente ao conceito de contido na teoria de conjuntos [8]. Na terceira linha do código 4.16 apresenta a sintaxe, e na quarta linha se a variável *x* está contida no conjunto.

```
1 <expression> instanceof <type>
2 x instanceof Person
3 <expression> in <range>
4 x in [1, 2, 3, 4, 5]
```

Código 4.16 – Exemplo de verificação de tipos

Uma chamada é uma fórmula ou expressão que consiste em uma referência a um predicado e vários argumentos, como apresentados em exemplos de códigos anteriores [8]. Por exemplo, *isThree(x)* pode ser uma chamada para um predicado que se mantém se o argumento *x* for igual a 3. Entretanto, uma chamada para um predicado também pode conter um operador de fechamento; como será apresentado na seção sobre recursividade.

As fórmulas quantificadas introduzem variáveis temporárias e as usa em fórmulas em seu corpo. Esta é uma forma de criar novas fórmulas a partir das já existentes [8]. Assim, as fórmulas quantificadas são representadas por:

- *exists*: Esta fórmula é válida se houver pelo menos um conjunto de valores que as variáveis podem usar para tornar a fórmula verdadeira, como apresentado o seu uso na seção Tipos de variáveis e classes.
- *forall*: Introduz algumas novas variáveis e normalmente tem duas fórmulas em seu corpo. Portanto, todos os valores retornados deverão satisfazer as duas fórmulas simultaneamente.
- *forex*: É a combinação das fórmulas quantificadas anteriores ligadas através do operador lógico *and*. Portanto, esta fórmula garante que haverá pelo menos um valor que satisfaz a fórmula 1.

A sintaxe das fórmulas quantificadas *exists*, *forall* e *forex* podem ser vistas respectivamente nas linhas do código 4.17.

```
1 exists(<variable declarations> | <formula>)
2 forall(<variable declarations> | <formula 1> | <formula 2>)
3 forex(<variable declarations> | <formula 1> | <formula 2>)
```

Código 4.17 – Sintaxe das fórmulas quantificadas

Os conectivos lógicos permitem combinar fórmulas existentes em fórmulas mais longas e complexas [8]. Além disso, os conectivos lógicos podem ser colocados entre parênteses para explicitar precedência entre os conectivos envolvidos. Os tipos de conectivos são:

- Negação: Inverte o sentido lógico da fórmula, ou seja, se a fórmula seleciona um determinado conjunto, a negação irá selecionar elementos que estão fora desse conjunto.
- Condicional: Segue a mesma implementação das linguagens de programação. Tem como objetivo executar ações baseadas em uma condição definida.
- Conjunção: Representa o *e* lógico.
- Disjunção: Representa o *ou* lógico.

- Implicação: Se uma fórmula é verdadeira, ela pode encadear outras fórmulas tornarem verdadeiras também.

Cada linha do código 4.18 ilustra um exemplo para cada conectivo apresentado, respectivamente.

```
1 not x = 1
2 if A then B else C
3 x = 1 and y = 1
4 x = 1 or y = 1
5 x = 1 implies y = 1
```

Código 4.18 – Sintaxe dos conectivos lógicos

4.2.6 Anotações

Uma anotação é uma palavra que pode ser escrita diretamente antes da declaração de uma entidade ou nome, como pode ser visto no código 4.19 com o exemplo para a definição de uma anotação para um módulo privado. O objetivo das anotações é definir algum tipo de comportamento [3], e a linguagem QL fornece os seguintes tipos:

- *Private*: Evita que nomes sejam exportados. Se um nome tiver a anotação *private* e se for acessado por meio de uma instrução de importação, somente é possível se referir a este nome a partir do *namespace* do módulo atual.

- *Abstract*: Define uma entidade abstrata. Predicados abstratos são predicados membros que não possuem corpo. Eles definem que as classes que herdam as classes abstratas terão obrigatoriamente que implementar os predicados abstratos definidos por elas.

- *Cached*: Indica que uma entidade deve ser avaliada em sua totalidade e armazenada na cache de avaliação. Todas as referências posteriores a esta entidade usarão os dados já calculados e armazenados. Isto afeta as referências de outras consultas, bem como da consulta atual. Por exemplo, pode ser útil armazenar em cache um predicado que leva muito tempo para ser avaliado e é reutilizado em muitos lugares. Em contrapartida, a quantidade de dados a serem armazenados na cache não pode ser grande, pois a memória é limitada e isto implicará em dificuldades para a otimização do compilador.

- *Deprecated*: É aplicada aos nomes que estão desatualizados e programados para remoção em uma versão futura do QL.

- *External*: É aplicada em predicados para definir um módulo como externo, ou seja, que pode ser importado por qualquer outro arquivo.

- *Transient*: É aplicada em predicados não membros que também são anotados com externo, para indicar que eles não devem ser armazenados em *cache* durante a avaliação.
- *Final*: É aplicada em entidades que não podem ser substituídas ou estendidas, ou seja, uma classe do tipo *final* não pode atuar como um tipo base para nenhum outro tipo. Da mesma forma, um predicado ou campo *final* não pode ser substituído em uma subclasse.
- *Library*: É aplicada a nomes aos quais podem ser referenciados em um arquivo *.qll*.
- *Query*: É utilizada para transformar um predicado (ou alias de predicado) em uma consulta. Isto significa que é parte da saída em um programa em linguagem QL.
- *Bindingset*: É utilizado para declarar explicitamente os conjuntos de ligação para um predicado. Um conjunto de ligação é um subconjunto dos argumentos do predicado de modo que, se estes argumentos forem restritos a um conjunto finito de valores, então o próprio predicado é finito. A anotação *bindingset* leva uma lista separada por vírgulas de variáveis e cada variável deve ser um argumento do predicado.
- *language[monotonicAggregates]*: Permite utilizar agregadores monotônicos ao invés dos agregadores padrão da linguagem.

```
1 private module M { ... }
```

Código 4.19 – Sintaxe das anotações

Além das anotações apresentadas, pode-se definir *pragmas* para informar ao compilador a realizar determinadas ações. O objetivo principal é definir diretivas para que o otimizador possa gerar códigos e consultas mais otimizadas em termos de desempenho. Porém, essas diretivas devem ser utilizadas somente em casos extremos em que um determinado código está sofrendo problemas de desempenho [3]. Os tipos são:

- *pragma[inline]*: Informa ao otimizador QL para sempre embutir o predicado de anotação nos locais onde é chamado [3]. Isso pode ser útil quando o corpo do predicado necessita de uma computação maior, pois garante que o predicado seja avaliado com as outras informações contextuais nos locais onde é chamado.
- *pragma[noinline]*: É utilizado para evitar que um predicado seja embutido no lugar onde é chamado [3]. Esta anotação é útil quando já foi agrupado certas variáveis em um predicado pelo usuário, assim garante que a relação seja avaliada inteiramente e não modificada pelo otimizador.
- *pragma[nomagic]*: É utilizado para evitar que o otimizador QL execute a otimização de “conjuntos mágicos” em um predicado [3]. Este tipo de otimização envolve obter as informações do contexto de uma chamada de predicado e colocá-las no corpo de

um predicado. Porém, a definição de *nomagic* implica consequentemente na anotação *noinline*.

- *pragma[noopt]*: É utilizado para evitar que o otimizador QL otimize um predicado, exceto quando for absolutamente necessário para que a compilação e avaliação funcionem. Esta anotação é raramente utilizada, pois implica em algumas complicações [3]. A primeira complicação é que o otimizador QL ordena automaticamente os conjuntos de uma fórmula complexa de maneira eficiente, e utilizando o predicado *noopt* os conjuntos são avaliados exatamente na ordem em que é escrito. A segunda complicação é que o otimizador QL cria automaticamente conjuntos intermediários para interpretar certas fórmulas em uma conjunção de fórmulas mais simples, e com o uso do predicado *noopt* deve-se escrever estas conjunções explicitamente. E a terceira complicação é que não é possível encadear chamadas de predicado ou chamar predicados em um *cast*.

4.2.7 Informações complementares

Em QL também existem outros componentes que auxiliam no processo de desenvolvimento e extensão da linguagem. A palavra-chave *alias* é uma alternativa para definir outros nomes (apelidos) a entidades existentes e podem ser definidos para classes, predicados e módulos.

Os módulos são uma possível maneira de agrupar códigos de forma organizada e proporcionar a escritas de bibliotecas para serem importadas através da palavra-chave *import* [11]. Os módulos em arquivos diferentes são gravados em arquivos com extensão *.qll*, diferentemente dos arquivos de consulta que são escritos em arquivos com extensão *.ql* [11]. Os módulos também podem ser definidos no mesmo arquivo e dentro de estruturas, como: predicados, tipo de dado criado pelo usuário, alias, e cláusulas *select* em consultas. O código 4.20 apresenta a definição do módulo *M*, representando que a classe *OneTwo* pode ser importada por outros arquivos.

```
1 module M {
2   class OneTwo extends OneTwoThree {
3     OneTwo() {
4       this = 1 or this = 2
5     }
6   }
7 }
```

Código 4.20 – Definição de módulo

4.2.8 Consultas

As consultas são a saída de um programa QL que avaliam conjuntos de resultados. Existem dois tipos de consultas [15]. Para um determinado módulo de consulta, as consultas neste módulo são:

1. A cláusula de seleção, se houver, será definida neste módulo.
2. Quaisquer predicados de consulta no *namespace* de predicado desse módulo. Ou seja, podem ser definidos no próprio módulo ou importados de um módulo diferente.

A cláusula de seleção serve para escrever um módulo de consulta através da palavra-chave *select* (geralmente no final do arquivo). No código 4.21 é apresentada a estrutura de um módulo de consulta. As palavras-chave *from* e *where* são opcionais, porém elas podem ser necessárias para selecionar um determinado conjunto baseado em predicados e classes. A cláusula *from* determina os itens que irão ser necessários para a cláusula *where* e *select*. A cláusula *where* define a lógica para os valores a serem selecionados. E por fim, a cláusula *select* define o que irá ser retornado como resposta da consulta a respeito dos valores satisfeitos pela cláusula *where*.

```
1 from int x, int y
2 where x = 3 and y in [0 .. 2]
3 select x, y, x * y as product
```

Código 4.21 – Estrutura de uma consulta

O código 4.21 demonstra a seleção dos números definidos pela cláusula *where*. Portanto, na cláusula *from* são definidos as variáveis x e y que são usadas nas cláusulas *where* e *select*. A cláusula *where* define que o valor de x será igual a 3 e y será os valores 0, 1 e 2. E a cláusula *select* define o formato da resposta separados por vírgula. Neste exemplo é definido que o formato de resposta de *x * y* será mostrado como a *string product* através da palavra-chave *as* e a saída serão os valores do quadro 2.

X	**Y**	**Product**
3	0	0
3	1	3
3	2	6

Quadro 2 – Resultado da consulta

O predicado de consulta é um predicado não membro com uma anotação do tipo *query*, retornando todas as tuplas avaliadas pelo predicado. O código 4.22 ilustra o mesmo algoritmo anterior, porém retorna somente o valor da resposta ao invés de um formato específico de resposta. Portanto, no quadro de resultado será alterada a palavra *product* para *result*.

```
query int getProduct(int x, int y) {
  x = 3 and
  y in [0 .. 2] and
  result = x * y
}
```

Código 4.22 – Anotação query

O benefício de escrever um predicado de consulta ao invés de uma cláusula de seleção, é que pode-se chamar o predicado em outras partes do código. Em contraste, a cláusula select é como um predicado anônimo, então pode-se chamar mais tarde [15].

4.2.9 Recursividade

Um predicado em linguagem QL é considerado recursivo se depende, direta ou indiretamente, de si mesmo. Para avaliar um predicado recursivo, o compilador QL encontra a parte menos fixa de um predicado recursivo e posteriormente aplica a invocação do predicado quantas vezes forem necessárias para obter todo o conjunto de valores resultantes. Em particular, ele começa com o conjunto vazio de valores e encontra novos valores aplicando repetidamente o predicado até que o conjunto de valores não seja mais alterado. Este conjunto é o ponto menos fixo e, portanto, o resultado da avaliação. Da mesma forma, o resultado de uma consulta QL é o ponto menos fixo dos predicados referenciados na consulta. Em certos casos, também pode-se usar agregações recursivamente.

O código 4.23 apresenta uma consulta de contagem através do predicado getANumber, que lista todos os inteiros de 0 a 100 (inclusive). Logo, o predicado avalia o conjunto contendo 0 e quaisquer inteiros que sejam um a mais do que um número já no conjunto.

```
int getANumber() {
  result = 0
  or
  result <= 100 and result = getANumber() + 1
}

select getANumber()
```

Código 4.23 – Exemplo de contagem recursiva

A recursão mútua têm um ciclo de predicados que dependem uns dos outros. O código 4.24 apresenta uma consulta que conta até 100 usando somente os números pares. E para o mesmo exemplo, se na cláusula select for alterado de getAnEven para getAnOdd, irá retornar os números ímpares de 0 a 100.

```
int getAnEven() {
  result = 0 or
  result <= 100 and result = getAnOdd() + 1
}

int getAnOdd() {
  result = getAnEven() + 1
}

select getAnEven()
```

Código 4.24 – Exemplo de contagem recursiva com números pares

O fechamento transitivo é um predicado recursivo cujos resultados são obtidos aplicando repetidamente o predicado original [14, 16]. Em particular, o predicado original deve ter dois argumentos com tipos compatíveis (possivelmente incluindo um valor *this* ou *result*).

O fechamento transitivo do tipo + serve para aplicar um predicado uma ou mais vezes, adicionando + ao nome do predicado [14, 16]. Se uma classe *Person* possui um predicado de membro *getAParent*, então *p.getAParent* irá retornar quaisquer pais de *p*. O fechamento transitivo *p.getAParent*+ retorna os pais de *p*, pais de pais de p e assim por diante. No código 4.25, o predicado *getAnAncestor()* representa o mesmo valor semântico que *getAParent+()*.

```
Person getAnAncestor() {
  result = this.getAParent() or
  result = this.getAParent().getAnAncestor()
}
```

Código 4.25 – Exemplo de fechamento transitivo +

O fechamento transitivo do tipo * é semelhante ao operador de fechamento transitivo do tipo +, exceto que pode-se aplicar um predicado a si mesmo nulo ou mais vezes [16]. O resultado de *p.getAParent*()* é um ancestral de *p* ou do próprio *p*. No código 4.26, *getAnAncestor2()* representa o mesmo valor semântico que *getAParent*()*.

```
Person getAnAncestor2() {
  result = this or result = this.getAParent().getAnAncestor2()
}
```

Código 4.26 – Exemplo de fechamento transitivo *

5 CODEQL PARA JAVASCRIPT

Em CodeQL existe uma extensa biblioteca para analisar códigos escritos na linguagem JavaScript, podendo apresentar informações sobre o código-fonte em diferentes níveis. As classes nesta biblioteca apresentam os dados de um banco de dados CodeQL em uma forma orientada a objetos, fornecendo abstrações e predicados [5]. A biblioteca é implementada como um conjunto de módulos QL, no qual a maioria podem ser importados diretamente através do módulo *javascript.qll*. Contudo, o objetivo desta seção é descrever o suporte do CodeQL para a linguagem JavaScript.

5.1 NÍVEL TEXTUAL

O objetivo deste nível é possibilitar as classes para representarem o código-fonte como arquivos de texto não estruturados. Em seu nível mais básico, uma base de código JavaScript pode ser vista como uma coleção de arquivos organizados em pastas, em que cada arquivo é composto de zero ou mais linhas de texto [5]. Entretanto, a representação textual do *software* não é incluída no banco de dados CodeQL por padrão.

Os arquivos são representados como entidades da classe *File* e as pastas como entidades da classe *Folder*. Ambas são subclasses da classe *Container* e possuem os seguintes predicados:

- *getParentContainer*: Retorna a pasta pai de um arquivo ou diretório.
- *getAFile*: Retorna um arquivo dentro do diretório.
- *getAFolder*: Retorna um diretório aninhado dentro de um diretório.
- *getAbsolutePath*: Retorna um caminho absoluto do sistema de arquivos.
- *getRelativePath*: Retorna um caminho relativo ao local de origem no banco de dados CodeQL.
- *getBaseName*: Retorna o nome base de um arquivo ou diretório, incluindo sua extensão.
- *getStem*: É semelhante ao *getBaseName*, mas não inclui a extensão do arquivo.
- *getExtension*: Retorna a extensão do arquivo, sem incluir o ponto.

```
1 import javascript
2
3 from Folder d
4 select d.getRelativePath(), count(File f | f = d.getAFile() and
     f.getExtension() = "js")
```

Código 5.1 – Nível textual

O código 5.1 ilustra a seleção a partir de cada pasta encontrada, mostra a quantidade de arquivos JavaScript contidos nela.

A maioria das entidades em um banco de dados CodeQL tem um local de origem associado. Os locais são identificados por quatro tipos de informações: um arquivo, uma linha inicial, uma coluna inicial, uma linha final e uma coluna final [5]. As contagens de linha e coluna são baseadas em 1, portanto, o primeiro caractere de um arquivo está na linha 1 e coluna 1. Todas as entidades associadas a um local de origem pertencem à classe *Locatable*. O local é modelado pela classe *Location* e pode ser acessado por meio do predicado *getLocation*. Assim, a classe *Location* fornece os seguintes predicados:

- *getFile*: Retorna o arquivo que este local está associado.
- *getStartLine, getStartColumn, getEndLine e getEndColumn*: Retorna a linha inicial, coluna inicial, linha final e coluna final; respectivamente.
- *getNumLines*: Retorna o número de linhas (inteiras ou parciais).
- *startsBefore e endsAfter*: Determina se um local começa antes ou termina depois de outro local. Para ambos, deve-se informar um local como parâmetro.
- *contains*: Indica se um local contém completamente outro local. Deve-se informar um local como parâmetro.

5.2 NÍVEL LÉXICO

Este nível é uma representação mais estruturada de um *software* JavaScript sendo fornecida pelas classes *Token* e *Comment*, que representam símbolos e comentários respectivamente [5]. A classe *Token* possui algumas subclasses para símbolos específicos e possui alguns predicados, dentre os principais:

- *getValue*: Retorna o texto do símbolo.
- *getIndex*: Retorna o índice do símbolo baseado no seu escopo.
- *getNextToken e getPreviousToken*: Baseado na sequência de símbolos que um arquivo pode conter, retorna o próximo símbolo ou o símbolo anterior; respetivamente.

```
1 import javascript
2
3 from PunctuatorToken comma
4 where comma.getValue() = ","
5 select comma
```

Código 5.2 – Nível léxico - Símbolos

O código 5.2 ilustra uma consulta realizada inteiramente no nível léxico, que encontra todos os símbolos de vírgula no código. Descrição:

- Linha 1: Importação da biblioteca JavaScript.
- Linha 3: Cláusula *from* para definir um elemento da subclasse *PunctuatorToken*, que representa operadores e outros símbolos de pontuação.
- Linha 4: Cláusula *where* para selecionar os símbolos que forem representados por vírgula.
- Linha 5: Cláusula *select* para mostrar o símbolo encontrado.

A classe *Comment* e suas subclasses representam os diferentes tipos de comentários que podem ocorrer em códigos JavaScript [5]. Como em aplicações WEB a linguagem JavaScript pode ser utilizada no lado servidor com o *runtime* Node.JS e também no lado cliente através dos navegadores, as classes representam tanto os comentários no formato JavaScript quanto no formato HTML. Assim, os principais predicados são representados similarmente aos predicados da classe *Token*, porém o texto do comentário é fornecido pelo predicado *getText*.

```
1 import javascript
2
3 from HtmlLineComment c
4 select c
```

Código 5.3 – Nível léxico - Comentários

O código 5.3 representa uma consulta realizada apenas no nível léxico para localizar os comentários no formato HTML <!– –>. Descrição:

- Linha 1: Importação da biblioteca JavaScript.
- Linha 3: Cláusula *from* para definir um elemento da subclasse HtmlLineComment, que representa comentários no formato HTML.
- Linha 4: Cláusula *select* para mostrar o elemento encontrado.

5.3 NÍVEL SINTÁTICO

A maioria das classes na biblioteca JavaScript tem o objetivo de representar um *software* como uma coleção de árvores de sintaxe abstrata (ASTs) [5]. Estas árvores representam a estrutura sintática abstrata do código-fonte, em que cada nó da árvore denota uma construção que ocorre no código-fonte [5]. A representação genérica da árvore ocorre através da classe AST-Node, que contém todas as entidades que representam os nós nas ASTs e define os predicados genéricos de travessia da árvore:

- *getChild(i)*: Retorna o i-ésimo filho deste nó AST.
- *getAChild(i)*: Retorna qualquer filho deste nó AST.
- *getParent()*: Retorna o pai deste nó AST.

As subseções a seguir representam formas de utilizar o CodeQL para obter diferentes tipos de informações das ASTs em diversos formatos.

5.3.1 Níveis superiores

Do ponto de vista sintático, cada código JavaScript é composto por um ou mais blocos de código de nível superior, que são blocos de código JavaScript que não pertencem a um bloco de código maior [5]. Os níveis superiores são representados pela classe *TopLevel* e suas subclasses:

- *Script*: Um arquivo autônomo ou elemento HTML com a tag <script>.
- *CodeInAttribute*: Um bloco de código originado de um valor de atributo HTML. Exemplo: Uma imagem pode conter o atributo *onerror*, que por sua vez, executa um método definido se houver algum erro no carregamento da imagem. Portanto, este atributo representa um código JavaScript.
- *Externs*: Um arquivo JavaScript contendo definições externas.

Toda classe *TopLevel* está contida em uma classe *File*, mas um único *File* pode conter mais de um *TopLevel*. Para ir de um *TopLevel* para o seu arquivo, pode ser invocado o predicado *getFile*; inversamente para um *File*, o predicado *getATopLevel* retorna um nível superior. Para cada nó AST, o predicado *ASTNode.getTopLevel()* pode ser invocado para encontrar o nível superior ao qual pertence.

5.3.2 Declarações e expressões

Além da classe *TopLevel*, as subclasses mais importantes da classe *ASTNode* são *Stmt* e *Expr*. Estas subclasses representam instruções e expressões, respectivamente. Esta subseção apresenta brevemente algumas classes e predicados mais importantes [5], entretanto com uma quantidade maior devido ao fato que estas subclasses são utilizadas intensivamente no desenvolvimento de consultas.

Em *Stmt*, o predicado *getcontainer* acessa a função mais interna ou nível superior em que a instrução está contida. As subclasses mais importantes são:

- *ControlStmt*: Representa uma declaração que controla a execução de outras declarações, ou seja, declarações do tipo *if*, *for*, *with*, *switch* e *try*. Esta classe possui subclasses para representar cada tipo de declaração e possui predicados para acessar as partes específicas do seu escopo. Também é possível utilizar o predicado *Controlstmt.GetAcontrolledstmt* para acessar as instruções que as controlam.

- *BlockStmt*: Representa um bloco de declarações. O predicado *BlockStmt.getStmt(int)* pode ser utilizado para acessar instruções individuais dentro do bloco, através de um número informado como argumento.

- *ExprStmt*: Representa uma declaração de expressão. O predicado *ExprStmt.getExpr()* pode ser utilizado para acessar a própria expressão.

- *JumpStmt*: Representa uma declaração que altera o fluxo de controle estruturado, ou seja, quando ocorre a execução de um *break*, *continue*, *return* ou *throw*. Para cada tipo de alteração que pode ocorrer no fluxo, há uma subclasse associada para representá-la. O predicado *JumpstMT.GetTarget()* pode ser utilizado para determinar o destino do fluxo de controle.

- *FunctionDeclStmt e ClassDeclStmt*: Representa uma declaração de função e classe, respectivamente.

- *DeclStmt*: Representa uma declaração genérica contendo uma ou mais declaradores que podem ser acessadas através do predicado *Declstmt.GetDeclarator(int)*, baseado na posição informada como argumento. Esta classe possui subclasses que representam tipos de diferentes declarações em JavaScript, como: *var*, *const* e *let*.

Para a classe *Expr*, pode-se utilizar o predicado *Expr.getEnclosingStmt* para obter a instrução mais interna em que uma expressão pertence. As subclasses mais importantes são:

- *ThisExpr, SuperExpr, ArrayExpr e ObjectExpr*: Representam uma expressão do tipo *this*, *super*, *array* e *object*; respectivamente.

- *FunctionExpr e ArrowFunctionExpr*: Representam uma expressão do tipo função de acordo com o modo que foram definidas, ou seja, uma função definida no formato padrão ou no formato de flechas; respectivamente.

- *ClassExpr, ParExpr, SeqExpr, ConditionalExpr*: Representam uma expressão do tipo *class*, expressão entre parênteses, sequência de expressões separadas por vírgula e condicional ternária; respectivamente.

- *InvokeExpr*: Representa expressão de uma chamada de função ou uma instanciação de um objeto através da palavra-chave *new*. Esta classe possui as subclasses *CallExpr*, para representar uma expressão para chamada de uma função; *NewExpr*, para a invocação da palavra-chave *new*; e *MethodCallExpr*, para representar uma invocação de uma expressão.

- *PropAccess*: Representa um acesso de propriedade, ou seja, uma expressão da forma *e.f* ou uma expressão de índice da forma *e[p]*.

- *UnaryExpr, BinaryExpr, Assignment, UpdateExpr*: Representam uma expressão unária, binária, atribuição, e incremento do tipo ++ ou --; respectivamente.

Stmt e *Expr* compartilham uma superclasse comum chamada *ExprOrStmt*, que é útil para consultas que operam em declarações ou em expressões, mas não em qualquer outro nó AST [5]. Como um exemplo de como usar os nós de expressão AST, o código 5.4 representa uma consulta para obter expressões da forma $e + f \gg g$; o que implica que as expressões devem ser reescritas como $(e + f) \gg g$ para esclarecer a precedência dos operadores:

```
1 import javascript
2
3 from ShiftExpr shift, AddExpr add
4 where add = shift.getAnOperand()
5 select add
```

Código 5.4 – Nível Sintático - expressões

Descrição do código 5.4:

- Linha 1: Importação da biblioteca *JavaScript*.
- Linha 3: Cláusula *from* para definir um elemento da classe *ShiftExpr* e *AddExpr*, que representam a operação de deslocamento para direita e operação de adição; respectivamente. Ambas as classes são subclasses de *BinaryExpr*.
- Linha 4: Cláusula *where* para definir uma expressão do tipo *ShiftExpr* a partir de uma expressão do tipo *AddExpr*.
- Linha 5: Cláusula *select* para mostrar a expressão encontrada.

5.3.3 Funções

Em JavaScript existem várias maneiras de definir funções. No ECMAScript 5 foram definidas as instruções de declaração de função e expressões de função [5]. Posteriormente, o ECMAScript 2015 adicionou expressões no formato de flechas [5]. Assim, todas as classes apresentadas nas subseções anteriores para representarem os diferentes tipos de definições de função são subclasses da classe *Function*. Esta classe fornece predicados comuns para acessar os parâmetros ou o corpo da função.

O código 5.5 representa uma consulta relacionada ao corpo da função para encontrar todas as funções declaradas com a palavra-chave *function*.

```
1 import javascript
2
3 from FunctionExpr fe
4 where fe.getBody() instanceof Expr
5 select fe
```

Código 5.5 – Nível Sintático - expressões

Descrição do código 5.5:

- Linha 1: Importação da classe JavaScript.
- Linha 3: Cláusula *from* para definir um elemento da classe *FunctionExpr*, que representa uma expressão de função.
- Linha 4: Cláusula *where* para verificar se o conteúdo da função é de fato uma expressão. O predicado *getBody* retorna o corpo da função e *instanceof* verifica se dois dados são do mesmo tipo (neste exemplo, duas expressões).
- Linha 5: Cláusula *select* para mostrar as funções encontradas.

```
1 import javascript
2
3 from Function fun, Parameter p, Parameter q, int i, int j
4 where p = fun.getParameter(i) and
5     q = fun.getParameter(j) and
6     i < j and
7     p.getAVariable() = q.getAVariable()
8 select fun
```

Código 5.6 – Nível Sintático - expressões

O código5.6 representa uma consulta relacionada aos parâmetros da função para encontrar funções que possuem dois parâmetros que vinculam a mesma variável. Descrição do código:

- Importação da biblioteca JavaScript.
- Cláusula *from* para definir um elemento que representa uma função, dois parâmetros de uma função, e dois números inteiros.
- Cláusula *where* obtém um parâmetro da função na posição de ordem i, obtém um parâmetro da função na posição de ordem j, verifica que o valor de i deve ser menor que j, e verifica se os dois parâmetros representam a mesma variável.
- Cláusula *select* para mostrar as funções encontradas.

5.3.4 Classes

As classes podem ser definidas por instruções de declaração de classe, representadas pela classe *ClassDeclStmt* (uma subclasse de *Stmt*), ou por expressões de classe, representadas

pela classe *ClassExpr* (uma subclasse de *Expr*) [5]. Ambas as classes também são subclasses de *ClassDefinition*, o que fornece predicados comuns para acessar todas as informações importantes sobre as classes, como: nome da classe, obter uma superclasse (se possível), membros da classe, método construtor e etc.

As definições de método são representadas pela classe *MethodDefinition* e a definição dos campos são representados pela classe *FieldDefinition* [5]. Ainda assim, existem três classes para modelar métodos especiais: *ConstructorDefinition* modela os construtores, enquanto *GetterMethodDefinition* e *SetterMethodDefinition* modelam *getters* e *setters*, respectivamente.

5.3.5 Variáveis e padrões de vinculação

As variáveis são declaradas por instruções de declaração (classe *DeclStmt*) e possuem três tipos de representação: instruções *var* (representadas pela classe *VarDeclStmt*), instruções *const* (representadas pela classe *ConstDeclStmt*) e instruções *let* (representadas pela classe *LetStmt*) [5]. Cada instrução de declaração possui um ou mais declaradores, representados pela classe *VariableDeclarator*. Cada declarador consiste em um padrão de vinculação, retornado pelo predicado *VariableDeclarator.getBindingPattern*.

A classe *Variable* modela todas as variáveis em um código JavaScript, incluindo variáveis globais, variáveis locais e parâmetros (tanto de funções quanto de cláusulas *catch*) [5]. As variáveis e suas declarações possuem objetivos diferentes, tais como: as variáveis locais podem ter mais de uma declaração, enquanto as variáveis globais e as variáveis de argumentos locais declaradas implicitamente não precisam ter uma declaração. Os identificadores que fazem referência a uma variável, por outro lado, recebem a classe *VarAccess*.

```
1  import javascript
2
3  from DeclStmt ds, VariableDeclarator d1, VariableDeclarator d2, Variable v,
       int i, int j
4  where d1 = ds.getDecl(i) and
5      d2 = ds.getDecl(j) and
6      i < j and
7      v = d1.getBindingPattern().getAVariable() and
8      v = d2.getBindingPattern().getAVariable() and
9      not ds.getTopLevel().isMinified()
10 select v.getName()
```

Código 5.7 – Nível Sintático - padrão de vinculação

O código 5.7 representa uma consulta para encontrar todas as instruções de declaração que declaram a mesma variável mais de uma vez. Descrição do código:

- Linha 1: Importação da biblioteca JavaScript.

- Linha 3: Cláusula *from* para definir um elemento da classe *DeclStmt*, dois elementos da classe *VariableDeclarator*, um elemento da classe *Variable* e dois elementos do tipo primitivo inteiro.
- Linha 4: Cláusula *where* para obter uma declaração na posição i, dentre todas as declarações existentes no código.
- Linha 5: Continuação da cláusula *where* para obter uma declaração na posição j.
- Linha 6: Continuação da cláusula *where* para verificar se o valor de i é menor do que j. Isto é realizado para obter duas declarações distintas.
- Linha 7: Continuação da cláusula *where* para obter o padrão de vinculação através do predicado *getBindingPattern* e em seguida obtém o valor da variável. Esta linha é aplicada para a variável na posição i.
- Linha 8: Realiza os mesmos passos da linha 7, porém aplicados na variável de posição j.
- Linha 9: Continuação da cláusula *where* para verificar se a declaração da classe *DeclStmt* não é um código reduzido em seu nível superior. Os códigos reduzidos representam códigos que são difíceis de serem interpretados pelo CodeQL [5].
- Linha 10: Cláusula *select* para mostrar as variáveis encontradas.

5.3.6 Objetos e propriedades

As propriedades de um objeto são representadas pela classe *Property*, que também é uma subclasse de *ASTNode*, mas não de *Expr* nem de *Stmt* [5]. A classe *Property* possui as subclasses *ValueProperty* e *PropertyAccessor* para representar respectivamente, propriedades de valor normal e propriedades *getter* e *setter*.

```
1 import javascript
2
3 from ObjectExpr oe, Property p1, Property p2, int i, int j
4 where p1 = oe.getProperty(i) and
5     p2 = oe.getProperty(j) and
6     i < j and
7     p1.getName() = p2.getName() and
8     not oe.getTopLevel().isMinified()
9 select oe, p1.getName()
```

Código 5.8 – Nível Sintático - propriedades

O código 5.8 representa uma consulta que sinaliza expressões de objeto contendo duas propriedades nomeadas de forma idêntica. Descrição do código:

- Linha 1: Importação da biblioteca JavaScript.

- Linha 3: Cláusula *from* para definir um elemento da classe *ObjectExpr* (expressão do tipo objeto), duas propriedades da classe *Property* e dois números inteiros.
- Linha 4: Cláusula *where* para obter uma propriedade de um objeto baseado na posição i, dentre todas as propriedades existentes no objeto.
- Linha 5: Continuação da cláusula *where* para realizar a mesma função da linha 4, porém aplicado a uma propriedade na posição j.
- Linha 6: Continuação da cláusula *where* para verificar que o valor de i é menor do que j.
- Linha 7: Continuação da cláusula *where* para verificar se os nomes das propriedades da Linha 4 e 5 são iguais.
- Linha 8: Continuação da cláusula *where* para verificar se o objeto não é um código reduzido em seu nível superior.
- Linha 9: Cláusula *select* para mostrar o objeto e a propriedade encontrados.

5.3.7 Módulos

A biblioteca JavaScript tem suporte para módulos ECMAScript 2015, assim como módulos CommonJS (ainda comumente empregados por bases de código Node.js) e módulos estilo AMD [5]. As classes *ES2015Module, NodeModule e AMDModule* representam estes três tipos de módulos e todos eles estendem da superclasse Module [5]. Os predicados mais importantes definidos pela classe *Module* são:

- *getName()*: Obtém o nome base do módulo.
- *getAnImportedModule()*: Obtém outro módulo por meio de um módulo, definidos através de *import* ou *require*.
- *getAnExportedSymbol()*: Obtém o nome de um símbolo que o módulo exporta.

Há também uma classe *Import* que modela as declarações de *import* no estilo ECMAScript 2015 e invocações de *require* no estilo CommonJS/AMD. Esta classe possui o predicado *Import.getImportedModule()* para fornecer acesso ao módulo ao qual a importação se refere.

5.4 FLUXO DE CONTROLE

Uma representação de *software* diferente em termos de grafos de fluxo de controle intraprocedural (CFGs) é fornecida pelas classes da biblioteca *CFG.qll* [5]. Estes grafos são representações de todos os caminhos que podem ser percorridos por um *software* durante sua execução. A classe *ControlFlowNode* representa um único nó no grafo de fluxo de controle e

este nó pode ser representado através de uma expressão, uma instrução ou um nó de fluxo de controle sintético.

As classes *Expr* e *Stmt* não herdam de *ControlFlowNode*, embora seus tipos de entidade sejam compatíveis. Portanto, deve-se converter explicitamente de um modo de representação para o outro se for necessário mapear entre as representações de *software* baseadas em AST e baseadas em CFG [5]. Existem dois tipos de nós de fluxo de controle sintéticos. Um deles são os nós de entrada (classe *ControlFlowEntryNode*), que representam o início de um nível superior ou uma função. O outro são os nós de saída (classe *ControlFlowExitNode*), que representam o fim do fluxo.

A maioria dos níveis superiores e funções têm outro nó CFG distinto, representando o nó CFG no qual a execução começa [5]. Ao contrário do nó de entrada, que é uma construção sintética, o nó inicial corresponde a um elemento real do *software*: para níveis superiores, é o primeiro nó CFG da primeira instrução; para funções, é o nó CFG correspondente ao seu primeiro parâmetro ou primeira instrução do corpo da função.

A estrutura do gráfico de fluxo de controle é refletida nos predicados da classe *ControlFlowNode*, dentre os principais:

- *getASuccessor*: Retorna um ControlFlowNode que é um sucessor do ControlFlowNode atual no gráfico de fluxo de controle.
- *getAPredecessor*: Representa o inverso do predicado getASuccessor.

Muitas análises baseadas em fluxo de controle são formuladas em termos de blocos básicos ao invés de nós de fluxo de controle únicos, onde um bloco básico é uma sequência máxima de nós de fluxo de controle sem ramificações ou junções [5]. A classe *BasicBlock()* de *BasicBlocks.qll* representa todos estes blocos básicos. Semelhante ao *ControlFlowNode*, é fornecido os predicados *getASuccessor()* e *getAPredecessor()* para navegar no grafo de fluxo de controle no nível de blocos básicos. Os predicados *getANode(), getNode(int)*, *getFirstNode()* e *getLastNode()* são utilizados para acessar individualmente os nós do fluxo de controle dentro de um bloco básico. O predicado *Function.getEntryBB()* retorna o bloco básico de entrada em uma função e *Function.getStartBB()* fornece acesso ao bloco básico inicial que contém o nó inicial da função.

```
1 import javascript
2
3 from Function f, GlobalVariable gv
4 where gv.getAnAccess().getEnclosingFunction() = f and
5     not f.getStartBB().isLiveAtEntry(gv, _)
6 select f, "This function uses " + gv + " like a local variable."
```

Código 5.9 – Nível Sintático - fluxo de controle

O exemplo de código 5.10 representa uma consulta de análise usando blocos básicos para encontrar variáveis globais que são usadas em funções de maneira incorreta, sugerindo utilizar variáveis locais ao invés de variáveis globais. Descrição do código:

- Linha 1: Importação da biblioteca JavaScript.
- Linha 3: Cláusula *from* para definir um elemento do tipo função e um elemento para representar uma variável global.
- Linha 4: Cláusula *where* para verificar se a variável *f* é uma função e se ela possui alguma variável global em seu escopo.
- Linha 5: Continuação da cláusula *where* para obter o bloco básico inicial baseado na função da linha 4 através do predicado *getStartBB* e é verificado se este bloco básico não possui nenhuma variável local instanciada com o conteúdo de uma variável global, como: *let local = global*. O predicado do modo *isLiveAtEntry(v, u)* determina se a variável global *v* está ativa na entrada do bloco básico dado e, se assim for, liga *u* a um uso de *v* que se refere ao seu valor na entrada [5].

5.5 FLUXO DE DADOS

5.5.1 Nós de fluxo de dados

A biblioteca *DataFlow* fornece uma representação do *software* como um grafo de fluxo de dados. Os nós são valores da classe *DataFlow::Node*, que possui a subclasse *ValueNode* [5]. Os nós envolvem uma expressão ou uma instrução que é considerada para produzir um valor de uma função ou uma instrução de declaração de classe.

Pode-se utilizar o predicado *DataFlow::valueNode* para converter uma expressão, função ou classe em seu ValueNode correspondente [5]. Também há um predicado auxiliar *DataFlow::parameterNode* que mapeia um parâmetro para o seu nó de fluxo de dados correspondente. Da mesma forma, há um predicado *ValueNode.getAstNode* para o mapeamento de *ValueNodes* para *ASTNodes*. Existe um predicado utilitário *Node.asExpr* que obtém a expressão subjacente para um *ValueNode* e é indefinido para todos os nós que não correspondem a uma expressão, porém este predicado não está definido para a função de agrupamento *ValueNodes* ou instruções de declaração de classe.

O predicado *Node.getAPredecessor* localiza outros nós de fluxo de dados a partir dos quais os valores podem fluir para este nó, e *Node.getASuccessor* para a outra direção [5]. Por exemplo, o código 5.10 apresenta uma consulta para encontrar todas as invocações de um método denominado *send* em um valor que é informado a partir de um parâmetro denominado *res*, indicando que talvez esteja enviando uma resposta de requisição HTTP.

```
1 import javascript
```

```
2
3 from SimpleParameter res, DataFlow::Node resNode, MethodCallExpr send
4 where res.getName() = "res" and
5       resNode = DataFlow::parameterNode(res) and
6       resNode.getASuccessor+() = DataFlow::valueNode(send.getReceiver()) and
7       send.getMethodName() = "send"
8 select send
```

Código 5.10 – Nível Sintático - Fluxo de dados

Descrição do código 5.10:

- Linha 1: Importação da biblioteca *JavaScript*.
- Lina 3: Cláusula *from* para definir um elemento da classe *SimpleParameter* para representar um único parâmetro; um elemento da classe *Node* para representar um nó no grafo de fluxo de dados; e um elemento da classe *MethodCallExpr* para representar uma invocação de um método.
- Linha 4: Cláusula *where* para definir que nome de um dado no grafo de fluxo de dados seja igual a *res*.
- Linha 5: Continuação da cláusula *where* para definir que um parâmetro seja baseado no nó obtido na linha 4.
- Linha 6: Continuação da cláusula *where* para definir que todos os nós sucessores ao nó obtido na linha 5 possuem em seu valor a invocação de um método qualquer.
- Linha 7: Continuação da cláusula *where* para definir que o nome do método da linha 6, se houver, seja igual a *send*.
- Linha 8: Cláusula *from* para apresentar o resultado encontrado.

5.5.2 Grafo de chamadas

A biblioteca JavaScript implementa um algoritmo de construção de grafo de chamada para aproximar estaticamente os destinos possíveis de chamadas de função e novas expressões [5]. Os algoritmos de grafo de chamadas tendem a ser incompletos, ou seja, muitas vezes não é possível resolver todos os alvos de chamadas [5]. Os algoritmos mais sofisticados podem sofrer do problema oposto de imprecisão, podendo inferir muitos alvos de chamada inválidos [5]. O grafo de chamadas é representado pelo predicado *getACallee* da classe *DataFlow::InvokeNode*, que calcula possíveis funções que podem ser chamadas em tempo de execução por uma expressão. Além disso, existem três predicados que indicam a qualidade das informações do receptor para uma invocação:

- *InvokeNode.isImprecise*: É válido para invocações em que o construtor do grafo de chamadas pode inferir destinos de chamadas imprecisas.
- *InvokeNode.isIncomplete*: É válido para chamadas em que o construtor do grafo de chamadas pode falhar em inferir possíveis destinos de chamada.
- *InvokeNode.isUncertain*: É válido se *isImprecise* ou *isUncertain* forem válidos.

5.5.3 Fluxo de dados entre funções

As análises baseadas em grafo de fluxo de dados descritas nas subseções anteriores não consideram o fluxo de argumentos de função para parâmetros de um retorno de invocação de função [5]. Assim, a biblioteca de fluxo de dados também fornece uma estrutura para a construção de análises entre funções personalizadas [5].

Para estas análises, existem uma diferença entre o fluxo de dados adequado e rastreamento de contaminação [5]. O último não apenas considera o fluxo de preservação de valor, mas casos em que um valor influencia outro sem determiná-lo inteiramente [5]. Por exemplo, em uma atribuição de *s2 = s1.substring(i)*, o valor de *s1* influencia o valor de *s2*, pois *s2* é atribuído pela invocação do método *substring* de *s1*. Em geral, *s2* não será atribuído a *s1* a si mesmo, portanto, não há fluxo de dados de *s1* para *s2*, mas *s1* ainda compromete o valor de *s2*.

Contudo, uma análise de fluxo de dados entre funções é baseada na extensão da classe *DataFlow::TrackedNode* ou *DataFlow::TrackedExpr*. O primeiro representa uma subclasse de *DataFlow::Node* e o último de *Expr*. Assim, eles garantem que os valores recém adicionados sejam rastreáveis. Desta forma, é possível utilizar o predicado *flowsTo* para descobrir para quais nós ou expressões o valor rastreado flui.

```
1  import javascript
2
3  class TrackedStringLiteral extends DataFlow::TrackedNode {
4      TrackedStringLiteral() {
5          this.asExpr() instanceof ConstantString
6      }
7  }
8
9  from TrackedStringLiteral source, DataFlow::Node sink, SsaExplicitDefinition
       def
10 where sink = DataFlow::ssaDefinitionNode(def) and source.flowsTo(sink) and
11       def.getSourceVariable().getName().toLowerCase() = "password"
12 select sink
```

Código 5.11 – Nível Sintático - Fluxo de dados entre funções

O código 5.11 representa uma consulta para encontrar senhas embutidas em um determinado código. Um possível algoritmo simples seria realizar a procura de constantes do tipo *string* fluindo para variáveis nomeadas por *"password"*. Portanto, pode-se realizar a extensão da

classe *TrackedExpr* para rastrear todas as *strings* constantes e utilizar o predicado *flowsTo* para encontrar casos em que a *string* flui para uma definição de uma variável de senha. Descrição do código:

- Linha 1: Importação da biblioteca JavaScript.
- Linha 3 a 7: Definição de uma classe que é estendida da classe *TrackedNode*, em que em seu método construtor há uma verificação da expressão representada por ela, verificando se é uma constante do tipo *string*.
- Linha 9: Cláusula *from* para definir um elemento da classe criada *TrackedStringLiteral*, um nó da classe *DataFlow*, e um elemento da classe *SsaExplicitDefinition*.
- Linha 10: Cláusula *where* para definir um nó que representa uma definição explícita no código e que o nó da constante *string* tenha um fluxo para o nó da definição explícita. A classe *SsaExplicitDefinition* é uma representação mais refinada do fluxo de dados de um *software* com base no Formulário de Atribuição Simples Estática (SSA) [5].
- Linha 11: Continuação da cláusula *where* para definir que o nome da constante *string* deve ser igual a "*password*".
- Linha 12: Cláusula *select* para apresentar o resultado encontrado.

As classes *TrackedNode* e *TrackedExpr* não restringem o conjunto de nós que possuem algum fluxo para a análise de fluxo entre funções, rastreando o fluxo em qualquer expressão para a qual eles possam fluir [5]. As análises de segurança com CodeQL para encontrar vulnerabilidades são comumente realizadas por meio do qual é especificado um fluxo de dados ou análise de contaminação em termos de *sources* (onde o fluxo começa, uma entrada de dado na aplicação pelo usuário), *sinks* (onde deve ser rastreado, um local sensível que leva à exploração da vulnerabilidade) e barreiras (onde o fluxo é interrompido e consequentemente não torna o código vulnerável).

As classes *DataFlow::Configuration* e *TaintTracking::Configuration* permitem especificar um fluxo de dados ou análise de contaminação, respectivamente, definindo os seguintes predicados:

- *isSource(DataFlow::Node nd)*: Seleciona todos os nós e de onde o rastreamento de fluxo começa, informando como parâmetro um nó base.
- *isSink(DataFlow::Node nd)*: Seleciona todos os nós para os quais o fluxo é rastreado, informando como parâmetro um nó base.
- *isBarrier(DataFlow::Node nd)*: Seleciona todos os nós que atuam como uma barreira para o fluxo de dados, informando como parâmetro um nó base. Da mesma forma, *isSanitizer* pode ser utilizado em configurações de rastreamento de contaminação.

- *isAdditionalFlowStep(DataFlow::Node src, DataFlow::Node trg)*: Permite especificar etapas de fluxo adicionais customizadas para a análise. O predicado *isAdditionalTaintStep* corresponde às configurações de rastreamento de contaminação.

```
1  class PasswordTracker extends DataFlow::Configuration {
2      PasswordTracker() {
3          // unique identifier for this configuration
4          this = "PasswordTracker"
5      }
6
7      override predicate isSource(DataFlow::Node nd) {
8          nd.asExpr() instanceof StringLiteral
9      }
10
11     predicate passwordVarAssign(Variable v, DataFlow::Node nd) {
12         exists (SsaExplicitDefinition def |
13             nd = DataFlow::ssaDefinitionNode(def) and
14             def.getSourceVariable() = v and
15             v.getName().toLowerCase() = "password"
16         )
17     }
18
19    override predicate isSink(DataFlow::Node nd) {
20         passwordVarAssign(_, nd)
21     }
22 }
23
24 from PasswordTracker pt, DataFlow::Node source, DataFlow::Node sink, Variable v
25 where pt.hasFlow(source, sink) and pt.passwordVarAssign(v, sink)
26 select sink, "Password variable " + v + " is assigned a constant string."
```

Código 5.12 – Nível Sintático - Fluxo de dados entre funções 2

O predicado *Configuration.hasFlow* realiza o rastreamento de fluxo real, começando em um *source* e procurando por fluxo *sink* que não passa por um nó de barreira [5]. Portanto, em análises de segurança resultará nas partes dos códigos vulneráveis. Para realizar as verificações de nó *source* e nó *sink*, os predicados *isSource* e *isSink* são utilizados para representar a lógica de negócio de quais nós representam tais especificidades.

Contudo, o código a 5.12 representa a configuração do fluxo de dados do código 5.11 ilustrado anteriormente. Descrição do código:

- Linha 1: Definição da classe *PasswordTracker*, que estende os predicados da classe *Data-*

Flow::Configuration. Portanto, esta classe representa a configuração de fluxo de dados.

- Linha 2 a 5: Método construtor da classe, que apenas atribui um nome para a configuração.
- Linha 7 a 9: Sobrescrita do predicado *isSource* para definir que os nós *sources* devem ser representados por uma expressão do tipo *string*.
- Linha 15 a 17: Definição do predicado *passwordVarAssign* para encontrar variáveis do tipo *string* que possuem o conteúdo *"password"* baseado em uma variável e um nó, informados como parâmetro.
- Linha 20 a 22: Sobrescrita do predicado *isSink* para definir os nós sensível que levam à vulnerabilidade (*sinks*). Portanto, é realizada a invocação do predicado *passwordVarAssign* com o primeiro parâmetro representando qualquer valor de variável e um nó a ser analisado.
- Linha 23: Cláusula *from* para definir um elemento da classe *PasswordTracker*, um elemento da classe *DataFlow::Node* para representar o *source*, outro elemento da classe *DataFlow::Node* para representar o *sink*, e um elemento da classe *Variable* para representar uma variável.
- Linha 24: Cláusula *where* para verificar se o elemento da classe *PasswordTracker* possui algum fluxo a partir do nó *source* até o nó *sink* e se a invocação do predicado *passwordVarAssign* com o parâmetro do nó *sink* é verdadeiro. A execução do predicado *hasFlow* implica na execução dos predicados *isSource* e *isSink* para realizar a verificação se um nó qualquer representa um *source* ou um *sink*, satisfazendo suas implementações. Contudo, o predicado *hasFlow* verifica se há um fluxo de dados a partir dos *sources* até os *sinks*.

6 TRABALHOS RELACIONADOS

6.1 FIX THAT FIX COMMIT: A REAL-WORLD REMEDIATION ANALYSIS OF JAVASCRIPT PROJECTS

Neste artigo de Bandara Vinuri *et al*. (2020), os autores abordam a realização de análises de código com CodeQL em projetos pertencentes ao GitHub e desenvolvidos na linguagem de programação JavaScript. Os repositórios escolhidos para análise são baseados nos 53 repositórios que mais possuem *commits* realizados. Para as análises, são utilizadas as vulnerabilidades definidas pela CWE e que possuem alguma consulta implementada e fornecida pelo repositório oficial do CodeQL [6]. Como consequência das análises, foram processados 118,023 *commits* através de 53 repositórios e foram encontradas 5,046 vulnerabilidades. No momento da publicação deste artigo, o CodeQL não era parte das análises automatizadas possíveis e, portanto, foi necessário criar um código para automatizar o processo das análises em todos os repositórios escolhidos. Contudo, este artigo mostra a efetividade das consultas existentes para o CodeQL e mostra que todas as vulnerabilidades já poderiam ter sido resolvidas muito antes da publicação do artigo. Como o CodeQL atualmente faz parte das possíveis análises de código automatizadas no GitHub, basta apenas o mantenedor do repositório ativar as análises do CodeQL. Realizando este processo, o repositório recebe uma análise a cada novo *commit* baseada nas consultas existentes no repositório oficial do CodeQL.

6.2 DETECTING EXPLOITABLE VULNERABILITIES IN ANDROID APPLICATIONS

Como os *scanners* de segurança e *softwares* SAST (*software* estático de teste de segurança de aplicativos) geralmente geram muitos resultados falso positivos em relação às vulnerabilidades, elas acabam requerendo em alguns casos a verificação manual de forma exagerada pelos engenheiros de segurança, o que é um processo de análise não escalável. O objetivo desta dissertação proposta pelo autor Sankarapandian, Shivasurya (2021) foi a construção de uma estrutura genérica para verificar vulnerabilidades de segurança exploráveis em aplicativos Android. A solução proposta, nomeada como DEVAA, é composta pelo analisador de código estático CodeQL, um sistema para emular o sistema operacional Android e do Android Debug Bridge. O fluxo de execução ocorre por meio da análise de código com CodeQL e de seus resultados encontrados. Para cada possível vulnerabilidade sinalizada pelo CodeQL, é realizada a validação da vulnerabilidade baseada em *payloads* fornecidos pelo engenheiro de segurança, enviando-os para o aplicativo em modo teste. Desta maneira, a vulnerabilidade pode ser completamente verificada por meio da sinalização do CodeQL e validada no aplicativo teste em execução. Com o DEVAA é possível adicionar mais *payloads* para testar as vulnerabilidades, possibilitando abranger o máximo de técnicas existentes para as explorações. Da mesma forma, é possível adicionar outros analisadores estáticos de código para funcionar conjuntamente com o CodeQL.

6.3 RTFM! AUTOMATIC ASSUMPTION DISCOVERY AND VERIFICATION DERIVATION FROM LIBRARY DOCUMENT FOR API MISUSE DETECTION

Os autores Lv, Tao *et al* (2020) neste artigo apresentam uma arquitetura, nomeada como Advance, capaz de realizar análises de vulnerabilidades em APIs. As análises são baseadas na interpretação da documentação das APIs e geração de consultas para o CodeQL por meio de modelos baseados em processamento de linguagem natural (PLN). Assim, com as consultas geradas é possível realizar as análises no código-fonte de qualquer aplicação que utilize a API. Contudo, foram realizadas a geração de consultas baseadas nas bibliotecas OpenSSL, SQLite, libpcap, libdbus e libxml2. Como resultado desta pesquisa foram analisadas 39 aplicações que possuem algumas das bibliotecas de forma integrada. Foram descobertos 193 usos indevidos, incluindo 139 vulnerabilidades nunca relatadas antes.

7 IMPLEMENTAÇÃO

Esta seção consiste na apresentação das consultas elaboradas para identificar as vulnerabilidades em códigos de aplicações desenvolvidas em JavaScript, especificamente de bibliotecas desenvolvidas para Node.js. A abordagem para identificar as vulnerabilidades através das consultas baseia-se na análise de contaminação, que consiste na identificação das variáveis que podem ser contaminadas com entradas controláveis pelo usuário (*sources*) e as rastreia para possíveis funções vulneráveis (*sinks*). Se algum dado de entrada contaminado for encaminhado para alguma função vulnerável, é então sinalizado como uma vulnerabilidade. O modelo de consulta para a análise de contaminação pode ser visto no código 7.1, em que no predicado *isSource* são definidas as entradas controláveis pelo usuário e no predicado *isSink* são definidas as funções vulneráveis. Portanto, na cláusula *where* do código 7.1 consiste apenas na verificação se há algum fluxo de dado a partir dos *sources* até os *sinks*, através do predicado *hasFlowPath*.

```
1  override predicate isSource(Node nd) {
2    ...
3  }
4
5  override predicate isSink(Node nd) {
6    ...
7  }
8
9  from MyConfig cfg, PathNode source, PathNode sink
10 where cfg.hasFlowPath(source, sink)
11 select source, sink
```

Código 7.1 – Modelo de consulta para análise de contaminação

Nas seções sobre as consultas, são descritas primeiramente como ocorrem as vulnerabilidades nos códigos em JavaScript e posteriormente é apresentada a modelagem através do modelo de análise de contaminação apresentado no código 7.1, a partir da definição dos *sources* e *sinks* de cada vulnerabilidade. As implementações das consultas para encontrar as vulnerabilidades foram organizadas da seguinte forma:

- Injeção de comando: Consultas 1 e 2.
- Injeção de código: Consultas 3 e 4.
- Poluição de protótipo: Consulta 5.

7.1 ENTRADAS DE DADOS CONTROLÁVEIS (SOURCES)

Para todas as consultas foi definido como entrada de dados todos os argumentos de funções em que o usuário tem acesso, ou seja, os argumentos de funções exportadas pelo código a ser analisado. Assim, quando ocorre a importação de um código em JavaScript, é possível acessar somente as funções que forem explicitamente exportadas. As duas linhas do código 7.2 ilustram as duas formas de exportar uma função de um código em JavaScript.

```
1 exports.function = function () {};
2 module.exports = function () {};
```

Código 7.2 – Formas de exportação de funções em JavaScript

As bibliotecas utilizadas pelas aplicações em Node.js são gerenciadas através do *software* NPM. O CodeQL proporciona uma biblioteca com alguns predicados para obter algumas informações sobre as bibliotecas desenvolvidas para o NPM, chamada de *semmle.javascript.NPM*.

```
1  import javascript
2  import DataFlow
3  import semmle.javascript.NPM
4
5  Node moduleExport(Module m) {
6     exists(PropWrite pw |
7        pw.getFile() = m.getFile() and
8        pw.toString() = "module.exports" and
9        result = pw.getRhs().getAFunctionValue()
10    )
11 }
12
13 Node export(Module m) {
14     result = m.getAnExportedValue(_).getAFunctionValue()
15 }
16
17 from NPMPackage pkg, Module m, FunctionNode node
18 where m = pkg.getMainModule() and
19   (node = export(m) or node = moduleExport(m))
20 select node.getAFunctionValue().getAParameter()
```

Código 7.3 – Consulta para obter os argumentos das funções exportadas

O código 7.3 ilustra a consulta completa para obter as duas formas de exportação de função baseada em uma biblioteca do NPM. Descrição do código:

- Linha 1, 2 e 3: Importação da biblioteca JavaScript, importação da biblioteca que proporciona obter os fluxos de dados e a importação da biblioteca para obter informações sobre as bibliotecas do NPM.
- Linha 5 a 11: Obtém as funções exportadas através de *module.exports*:
  - Linha 5: Parâmetro *Module* representa um módulo de uma biblioteca NPM.
  - Linha 6: Obtém uma variável do tipo *PropWrite* para obter a escrita no formato apresentado no código 7.2.
  - Linha 7: Define que o arquivo a ser analisado deve ser o mesmo arquivo que contém a biblioteca.
  - Linha 8: Define que a expressão de atribuição, como uma *string*, deve ser igual a *module.exports*.
  - Linha 9: Define que o resultado será a função da parte direita da atribuição de *module.exports*.
- Linha 13 à 15: Obtém as funções exportadas através de *exports.function*.
  - Linha 13: Parâmetro *Module* representa um módulo de uma biblioteca NPM.
  - Linha 14: O resultado é definido pelo predicado *getAnExportedValue* fornecido pela biblioteca *semmle.javascript.NPM*. Assim, o predicado *getAnExportedValue* é invocado com o parâmetro *don't care*, representando qualquer valor de exportação; e posteriormente é obtido a função que este valor representa.
- Linha 17 à 20: Consulta para obter os argumentos das funções exportadas.
  - Linha 17: Cláusula *from* para definir as variáveis do tipo *NPMPackage*, representando uma biblioteca; *Module*, representando um módulo; e *FunctionNode*, representando um nó do fluxo de dados de uma função.
  - Linha 18: Cláusula *where* para obter o módulo principal exportado pela biblioteca NPM.
  - Linha 19: Continuação da cláusula *where* para definir que as funções podem ser do tipo dos predicados criados anteriormente para representar as duas formas de exportação de função.
  - Linha 20: Cláusula *select* para selecionar os argumentos das funções que satisfazem a cláusula *where*.

7.2 CONSULTA 1: INJEÇÃO DE COMANDO DE FORMA DIRETA

Esta consulta representa a vulnerabilidade de injeção de comando por meio da biblioteca *child_process* apresentada na seção 1 do capítulo 3. Como esta biblioteca apresenta diferentes maneiras de executar comandos no sistema operacional, esta consulta aborda a execução das funções *exec* (função assíncrona) e *execSync* (função síncrona), em que ambas recebem o comando a ser executado pelo primeiro argumento da função, como ilustrado no código 7.4.

```
1 const { exec, execSync } = require('child_process');
2 exec('command 1');
3 execSync('command 2');
```

Código 7.4 – Exemplo de injeção de comando

Baseado no modelo de consulta para análise de contaminação apresentado na seção 5 do capítulo 5, esta consulta define no predicado *isSource* os argumentos de funções exportadas obtidos pela consulta 7.3, como ilustrado no código 7.5.

```
override predicate isSource(Node nd) {
  exists( NPMPackage pkg , Module m, FunctionNode node |
        m = pkg.getMainModule() and (node = export(m) or node =
            moduleExport(m)) and
          nd = node.getAFunctionValue().getAParameter()
    )
}

override predicate isSink(Node nd) {
  exists(ModuleImportNode imported, CallNode func |
    (
      func = imported.getAMemberCall("exec") or
      func = imported.getAMemberCall("execSync")
    ) and nd = func.getArgument(0)
  )
}
```

Código 7.5 – Consulta injeção de comando direta

O objetivo do predicado *isSink* é obter os nós do fluxo de dados que representam o primeiro argumento das funções *exec* e *execSync* que são executadas através da biblioteca *child_process*. Descrição:

- Linha 10: Define a variável do tipo *ModuleImportNode* para obter um módulo que é

importado no código em análise e a variável do tipo *CallNode*, que representa um nó de chamada de função.

- Linha 12: Através da biblioteca importada, obtém uma função que esta biblioteca acessa, cujo o nome é *exec*.
- Linha 13: Realiza o mesmo procedimento da linha 12 para o nome *execSync*.
- Linha 14: Define que o nó *sink* é representado pelo primeiro argumento das funções obtidas nas linhas 17 ou 18.

As cláusulas *from*, *where* e *select* seguem o mesmo padrão da consulta apresentada no código 5.12. Portanto, a análise de contaminação é validada através do predicado *hasFlowPath*, em que verifica se há algum fluxo a partir do nós *source* até os nós *sink*. A consulta completa pode ser vista no apêndice A.1.

7.3 CONSULTA 2: INJEÇÃO DE COMANDO POR MEIO DA FUNÇÃO SPLIT

Esta consulta representa uma extensão da consulta anterior para obter as outras maneiras de executar comandos no sistema operacional através da biblioteca *child_process*, tais como: *execFile*, *execFileSync*, *spawn* e *spawnSync*. O comportamento destas funções ocorrem diferente das funções da consulta anterior, em que estas funções recebem no primeiro parâmetro o binário a ser executado e os argumentos para este binário no segundo argumento como *array* de *strings*, como ilustrado no código 7.6.

```
1 const { execFile, execFileSync, spawn, spawnSync } = require('child_process');
2
3 execFile('binary', ['argument 1', 'argument 2', '...']);
4 execFileSync('binary', ['argument 1', 'argument 2', '...']);
5 spawn('binary', ['argument 1', 'argument 2', '...']);
6 spawnSync('binary', ['argument 1', 'argument 2', '...']);
```

Código 7.6 – Funções para executar comando do sistema operacional com 2 argumentos

Existem diferentes formas de como uma biblioteca pode tratar os dados providos pelo usuário até efetivamente executar estas funções e tornar o código vulnerável. Esta consulta aborda a função *split* como uma possível função a ser utilizada pelas bibliotecas com o objetivo de tornar a entrada de dados do usuário, que é uma *string*, em um *array* de *strings* para representar os argumentos de um binário, como pode ser visto no código 7.7. Esta abordagem ainda necessita da verificação de qual binário será executado, pois para haver de fato a execução de comandos no sistema operacional o binário precisa fornecer algum argumento que possibilite este tipo de exploração.

```
const { execFile } = require('child_process');

function exportedFunction(userData) {
  const arguments = userData.split(' ');
  execFile('bin', arguments);
}
```

Código 7.7 – Exemplo de injeção de comando por meio da função split

Baseando-se nesta abordagem, foram definidas duas análises de contaminação para esta consulta. Uma análise para o fluxo de dados dos argumentos das funções exportadas até a função *split* e outra análise para o fluxo de dados da função *split* até as funções sensíveis da biblioteca *child_process*. O conteúdo do predicado *isSource* da primeira análise é representado pelo mesmo conteúdo de *isSource* do código 7.5 e o conteúdo de *isSink* pelo código 7.8.

```
1 override predicate isSink(Node nd) {
2     exists(CallExpr callSplit, AnalyzedVarDef split |
3         callSplit.getCalleeName() = "split" and
4         callSplit.getAnArgument().getStringValue().matches(" ") and
5         nd = callSplit.flow()
6     )
7 }
```

Código 7.8 – Predicado isSink para a primeira análise.

Descrição do código 7.8:

- Linha 2: Define as variáveis do tipo *CallExpr*, que representa uma expressão de chamada de função; e *AnalyzedVarDef*, que representa uma definição de variável em qualquer forma de definição possível em JavaScript.
- Linha 3: Define que o nome da função que está sendo chamada é igual a *split*.
- Linha 4: Define que o argumento desta função deve ser igual a um espaço em branco, para representar o delimitador da criação do *array* por espaço em branco utilizado pela função *split*.
- Linha 5: Define que o nó *sink* é representado pelo nó da expressão da chamada de função.

Para a segunda análise é utilizado para o predicado *isSource* o nó que representa a função *split*, que é representado pelo nó *sink* da análise de contaminação anterior. Portanto, o nó utilizado como parâmetro do predicado *isSource* será informado através do resultado da consulta anterior na cláusula *where*. O predicado *isSink* é similar ao código 7.5, entretanto são

utilizadas as funções da biblioteca *child_process* para este escopo, e o argumento que torna o código vulnerável é representado pelo argumento 2 (posição 1), como ilustrado no código 7.9. Na linha 9 é definido que o primeiro argumento da função sensível deve ser igual ao binário *curl*, porém em análises reais deve-se adicionar outros nomes de binários em que existe alguma possibilidade de execução de comandos do sistema operacional via argumentos.

```
1  override predicate isSink(Node nd) {
2      exists(ModuleImportNode imported, CallNode func, Node arg |
3        (
4          func = imported.getAMemberCall("execFile") or
5          func = imported.getAMemberCall("execFileSync") or
6          func = imported.getAMemberCall("spawn") or
7          func = imported.getAMemberCall("spawnSync")
8        ) and (
9          func.getArgument(0).toString().matches("curl") and
10         nd = func.getArgument(1)
11       )
12     )
13 }
```

Código 7.9 – Predicado isSink para a segunda análise.

Por fim, as cláusulas *from*, *where* e *select* necessitam das modificações descritas anteriormente, como a definição de duas análises de contaminação e definir que o nó *sink* da primeira análise representa o nó *source* da segunda análise, como pode ser visto no código 7.10. O código completo desta consulta pode ser visto no apêndice B.1.

```
1  override predicate isSink(Node nd) {
2  from MyConfig cfg, PathNode source, PathNode sink, MyConfig2 cfg2, Node sink2
3  where cfg.hasFlowPath(source, sink) and cfg2.hasFlow(sink.getNode(), sink2)
4  select source, sink2
```

Código 7.10 – Consulta injeção de comando por meio da função split

7.4 CONSULTA 3: INJEÇÃO DE CÓDIGO POR MEIO DA FUNÇÃO *EVAL*

Como descrito na seção 1 do capítulo 3, a função *eval* é utilizada para executar códigos JavaScript a partir de *strings*. Portanto, a abordagem desta consulta consiste em verificar se há algum fluxo de dados a partir dos parâmetros das funções exportadas até o parâmetro de uma chamada da função *eval*. A implementação desta consulta é semelhante a consulta da seção 1, com a diferença no conteúdo do predicado *isSink*, como pode ser visto no código 7.11. A

consulta completa pode ser vista no apêndice C.1.

```
1 override predicate isSink(Node nd) {
2     exists(CallNode func |
3       func.getCalleeName() = "eval" and
4       nd = func.getAnArgument()
5     )
6 }
```

Código 7.11 – Predicado isSink para para consulta de injeção de código por meio da função eval

Descrição do código 7.11:

- Linha 2: Define uma variável do tipo *CallNode*, que representa um nó de expressão de chamada de função.
- Linha 3: Define que o nome da função que está sendo chamada é igual a *eval*.
- Linha 4: Define que o nó *sink* é representado pelo argumento da função *eval*.

7.5 CONSULTA 4: INJEÇÃO DE CÓDIGO POR MEIO DE CRIAÇÃO DE FUNÇÃO

Além da função *eval*, em JavaScript é possível também executar código a partir da definição de uma nova função e a invocação da mesma, como ilustrado no código 7.12.

```
1 function exportedFunction(userData) {
2   const newFunction = new Function(userData);
3   newFunction();
4 }
```

Código 7.12 – Exemplo de execução de código a partir da criação e invocação de nova função

Esta consulta segue o mesmo modelo da seção anterior, com a modificação do predicado *isSink*, como pode ser visto no código 7.13. A consulta completa pode ser vista no apêndice D.1.

```
1 override predicate isSink(Node nd) {
2     exists(NewExpr v, AnalyzedVarDef a, CallExpr e, Variable va |
3       v.getCalleeName() = "Function" and
4       a.getRhs() = v.flow() and
5       va = a.getAVariable() and
6       va.getName() = e.getCalleeName() and
```

```
7       e.getLocation().getStartLine() =
            va.getAnAccess().getLocation().getStartLine() and
8       nd = v.getAnArgument().flow()
9     )
10  }
```

Código 7.13 – Predicado isSink

Descrição do código 7.13:

- Linha 2: Definição das variáveis:
  - *NewExpr*: Representa uma expressão que possui a palavra-chave *new*. Por exemplo: *new Function*.
  - *AnalyzedVarDef*: Representa uma definição de variável em qualquer formato possível em JavaScript.
  - *CallExpr*: Representa uma expressão de chamada de função.
  - *Variable*: Representa uma variável qualquer.
- Linha 3: Define que o nome para a expressão *new* deve ser igual a *Function*. Por exemplo: *new Funtion()*.
- Linha 4: Define que a definição de uma variável deve conter a expressão *new* da linha 3 no lado direito da atribuição. Por exemplo: *let a = new Function()*.
- Linha 5: Obtém uma variável qualquer para o tipo *Variable*.
- Linha 6: Define que o nome da variável da linha 5 deve ser o mesmo nome da função chamadora do tipo *CallExpr*.
- Linha 6: Define que o local da chamada da função é o mesmo local de um acesso da variável da linha 5. Isto é necessário para verificar se um acesso da variável representa uma chamada de função ao invés de outro uso.
- Linha 7: Define que o nó *sink* é um argumento da expressão *new Function*.

7.6 CONSULTA 5: POLUIÇÃO DE PROTÓTIPO

7.6.1 Sobre a vulnerabilidade

A linguagem JavaScript é baseada em protótipos, sendo assim, quando variáveis primitivas, objetos e funções são criados, esta criação ocorre por meio da instanciação de um protótipo que representa o valor a ser instanciado. Por exemplo, ao criar uma variável do tipo *string*, esta criação baseia-se em um protótipo que representa o tipo *string*. Este protótipo pode

ser acessado por meio da palavra-chave *__proto__* de qualquer objeto em JavaScript, como pode ser visto no código 7.14.

```
1 let a = 'this is a string';
2 console.log(a.__proto__);
```

Código 7.14 – Exemplo de acesso __proto__

Em JavaScript é possível criar novos protótipos e alterar os protótipos existentes. O código 7.15 ilustra como é possível alterar a função *toString* do protótipo do tipo *integer*, que é uma função que converte qualquer valor numérico em um valor do tipo *string*. Contudo, nota-se neste código que ao modificar a função *toString* do protótipo e instanciar outra variável do mesmo tipo, a função *toString* continua modificada, pois a criação dela é baseada no protótipo que foi alterado.

```
1 let number = 10;
2 console.log(number.toString()); // '10'
3
4 number.__proto__.toString = () => { return 'hello'};
5 console.log(number.toString()); // 'hello'
6
7 let test = 100;
8 console.log(test.toString()); // 'hello'
```

Código 7.15 – Exemplo de alteração do protótipo

Descrição do código 7.15:

- Linha 1: Instanciação da variável *number*, representando um valor numérico.
- Linha 2: Imprime o valor da variável *number* através da chamada da função *toString*, retornando o valor *10*.
- Linha 4: Altera a função *toString* do protótipo para retornar o valor *hello*.
- Linha 5: Realiza o mesmo procedimento da linha 2, porém desta vez imprimindo o valor *hello*.
- Linha 7: Instanciação de outra variável nomeada como *test* para representar outro valor numérico.
- Linha 8: Imprime o valor da variável *test* através da chamada da função *toString*, retornando o valor *hello* também.

As vulnerabilidades neste tipo de abordagem ocorrem quando o dado de um usuário pode modificar o protótipo de alguma forma. Na prática, estas vulnerabilidades ocorrem através dos objetos que são enviados geralmente através de APIs, em que são enviados os dados no formato JSON e é necessário realizar a conversão de uma *string* no formato JSON da requisição HTTP para um objeto em JavaScript. Assim, neste processo de conversão é necessário copiar as propriedades e seus valores correspondentes em um objeto em JavaScript.

```
1 {'__proto__': {'admin': true}}
```

Código 7.16 – Exemplo de alteração do protótipo no formato JSON

Se um usuário envia o conteúdo de um arquivo JSON ilustrado no código 7.16 e o processo de conversão não possua nenhum filtro para bloquear a *string* *__proto__*, esta conversão realizará algum processo similar ao código 7.17 e, consequentemente, a aplicação se torna vulnerável.

```
1  const userString = "{'__proto__': {'admin': true}}";
2  const userJson = JSON.parse(userString);
3  let obj = {}
4  console.log(merge(obj, userJson));
5
6  function merge(obj, userJson) {
7    let value;
8
9    for (key in userJson) {
10     value = userJson[key];
11     if (typeof value === 'object') {
12       obj[key] = merge(obj[key], value);
13     } else {
14       obj[key] = value;
15     }
16
17   }
18   return obj;
19 }
```

Código 7.17 – Exemplo de conversão de um objeto JSON em objeto JavaScript

Descrição do código:

- Linha 1: *userString* representa um conteúdo JSON no formato de *string*, para ilustrar o caso do conteúdo recebido através das requisições HTTP.

- Linha 2: Realiza a conversão de *string* para um objeto JSON e atribui o resultado para a constante *userJson*.
- Linha 3: Cria um objeto JavaScript vazio que irá receber o resultado da conversão.
- Linha 4: Realiza a chamada da função *merge* para realizar a conversão do objeto JSON para o formato JavaScript e imprime o resultado na tela.
- Linha 9: Percorre através do laço de repetição as propriedades existentes no objeto JSON. Neste exemplo, só há uma propriedade nomeada como *__proto__*.
- Linha 10: Obtém o conteúdo correspondente à propriedade *__proto__* e armazena na variável *value*.
- Linha 11: Verifica se o tipo do valor da linha 10 é outro objeto JSON.
- Linha 12: Caso o resultado da verificação da linha 11 for verdadeiro, é necessário chamar a função *merge* novamente para realizar a conversão.
- Linha 14: Caso o resultado da linha 10 for falso, então ocorre atribuição no objeto JavaScript.

O código 7.17 ilustra que quando um usuário informa um conteúdo malicioso é possível poluir o protótipo dos objetos JavaScript. Esta falha é crítica, pois com a modificação do protótipo é possível fazer com que a aplicação vulnerável se comporte de maneira diferente, podendo levar à execução remota de código quando é possível modificar as funções de forma similar a abordagem da vulnerabilidade de injeção de código.

7.6.2 Consulta

Esta consulta foi modelada através de três análises de contaminação. A primeira análise consiste no fluxo de dados a partir dos argumentos de qualquer função até a leitura do conteúdo associado a uma propriedade de um objeto, por exemplo: *value = userObj[key]*. A segunda análise baseia-se no fluxo de dados a partir da leitura do conteúdo associado a uma propriedade da primeira análise até a atribuição do conteúdo de uma propriedade de outro objeto, da seguinte forma: *obj[key] = value*. A terceira análise é baseada no fluxo de dados dos argumentos das funções exportadas até os argumentos da função da primeira análise, definidos através da cláusula *where*.

```
1 override predicate isSink(Node nd) {
2     exists(PropRead prop |
3         nd = prop.asExpr().flow()
4     )
5 }
```

Código 7.18 – Predicado isSink da primeira análise

O código 7.18 ilustra o predicado *isSink* para a primeira análise. Descrição:

- Linha 2: Definição de uma variável do tipo *PropRead*, que representa a leitura do conteúdo de uma propriedade de um objeto.
- Linha 3: Define que o nó *sink* é representado por qualquer expressão do tipo *PropRead*.

```
1 override predicate isSink(Node nd) {
2     exists( PropWrite prop |
3         nd = prop.getRhs().asExpr().flow()
4     )
5 }
```

Código 7.19 – Predicado isSink da segunda análise

O código 7.19 ilustra o predicado *isSink* para a segunda análise. Descrição:

- Linha 2: Definição de uma variável do tipo *PropWrite*, que representa a atribuição de conteúdo de uma propriedade de um objeto.
- Linha 3: Define que o nó *sink* é representado por qualquer expressão que esteja no lado direito da atribuição de *PropWrite*.

```
1 from MyConfig1 cfg1, MyConfig2 cfg2, MyConfig3 cfg3, Node source1, Node
      source2, Node sink1, PropWrite w, PropRead r, Node sink2, string propName
2 where cfg1.hasFlow(source2, sink1) and cfg2.hasFlow(sink1, sink2) and
3   r = sink1 and w.getRhs() = sink2 and
4   propName = r.getPropertyNameExpr().toString() and
5   propName.matches(w.getPropertyNameExpr().toString()) and
6   cfg3.hasFlow(source1, source2)
7 select source1, source2, sink1, sink2
```

Código 7.20 – Consulta para poluição de prototipo

O código 7.20 apresenta o conteúdo necessário para as cláusulas *from*, *where* e *select*. A consulta completa pode ser vista no apêndice E.1. Descrição:

- Linha 1: Definição de variáveis:
  - *MyConfig*: Classe que representa a configuração da análise de contaminação, em que *cfg1* representa a primeira análise, *cfg2* a segunda análise e *cfg3* a terceira análise.

  - *Node*: Nós para representar os dados providos dos predicados *isSource* e *isSink* de cada análise.
  - *PropRead*: Representa a leitura do conteúdo de uma propriedade de um objeto.
  - *PropWrite*: Representa a atribuição de conteúdo de uma propriedade de um objeto.
  - *string*: Utilizado para representar o nome das propriedades dos objetos.

- Linha 2: Cláusula *where* para obter os fluxos dos dados para primeira e segunda análise.
- Linha 3: Continuação da cláusula *where* para recuperar a expressão representada pelos nós *sinks* da primeira e segunda análise em formato de expressão. Isto é realizado para obter o nome da propriedade dos objetos encontrados.
- Linha 4: Define que o conteúdo de *propName* é o nome da propriedade do objeto encontrado na primeira análise.
- Linha 5: Define que o nome da propriedade do objeto da linha 4 deve ser igual ao objeto *sink* da segunda análise.
- Linha 6: Define que o nó *sink* da terceira análise é representado pelo argumento da função da primeira análise.
- Linha 7: Cláusula *select* para obter os resultados encontrados.

7.7 RESULTADOS

As consultas foram primeiramente testadas em códigos exemplos vulneráveis, apresentados nas seções de cada consulta. Posteriormente, as consultas foram executadas com o objetivo de encontrar as vulnerabilidades já relatadas e catalogadas através do banco de dados de consultoria do GitHub [9].

O GitHub oferece uma API para obter todos os dados necessários para a execução, em que é possível obter todas as vulnerabilidades relacionadas as bibliotecas do NPM e seus respectivos códigos de CWE (*Common Weakness Enumeration*) associados. Portanto, através dos códigos de CWE é possível obter somente as bibliotecas que possuíram alguma vulnerabilidade específica. Para a execução das consultas foi desenvolvido um algoritmo para obter as bibliotecas do NPM vulneráveis baseadas em suas versões e seus códigos de CWE associados, e um algoritmo para realizar as execuções de forma automatizada. O código completo de ambos os algoritmos pode ser visto nos apêndices F.1 e G.1.

```
1 async function main() {
2   // Command injection
3   let CWES = ['CWE-1321', 'CWE-915'];
```

```
4   let vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null,
        2);
5   fs.writeFileSync('command-injection.json', vulnerabilities);
6
7   // Code injection
8   CWES = ['CWE-1321', 'CWE-915'];
9   vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null, 2);
10  fs.writeFileSync('code-ijection.json', vulnerabilities);
11
12
13  // Prototype pollution
14  CWES = ['CWE-1321', 'CWE-915'];
15  vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null, 2);
16  fs.writeFileSync('prototype-pollution.json', vulnerabilities);
17 }
```

Código 7.21 – Algoritmo para obter as bibliotecas do NPM vulneráveis

O código 7.21 representa, de forma resumida, o primeiro algoritmo em JavaScript para obter as bibliotecas vulneráveis, realizado em três passos. Na variável *CWES* são definidos os códigos CWE associados a cada tipo de vulnerabilidade. É obtido as bibliotecas vulneráveis através da chamada da função *getVulnerabilities*, retornando assim um objeto JSON contendo o nome das bibliotecas e suas respectivas versões vulneráveis. Por fim, é realizado a escrita a deste objeto JSON em um arquivo em disco, através da chamada da função *writeFileSync*.

```
1  async function main() {
2    const vulnerabilities =
         JSON.parse(fs.readFileSync('./command-injection.json', 'utf8'));
3    let packagesQtt = 0;
4    let vulnerable = 0;
5    let version;
6
7    for (vuln of vulnerabilities) {
8      version = vuln.vulnerableVersionRange
9      NPMPackageInstall(vuln.package, getVulnerableVersion(version,
           version.match(/\./g).length));
10     createDB('javascript', vuln.package, `${vuln.package}-db`);
11
12     if (runQuery(`${vuln.package}-db`, './queries/command-injection.ql')) {
13       vulnerable++;
14     }
15     packagesQtt++;
```

```
16    }
17 }
```

Código 7.22 – Algoritmo para análise das consultadas desenvolvidas

O código 7.22 apresenta, de forma resumida, o algoritmo para analisar todas as vulnerabilidades das consultas desenvolvidas nas bibliotecas encontradas pelo código 7.21, guardadas no arquivo JSON em disco. Portanto, o código 7.22 representa apenas a execução para as análises da vulnerabilidade de injeção de comando, sendo assim executados os mesmos procedimentos para as vulnerabilidades de injeção de código e poluição de protótipo com os seus respectivos arquivos em disco que contém as bibliotecas vulneráveis. Primeiro é feito a leitura do arquivo JSON que está em disco, que contém o nome das bibliotecas vulneráveis e suas respectivas versões vulneráveis. Assim, é feito o *download* da biblioteca do NPM através da chamada da função *NPMPackageInstall*, informando nos argumentos o nome da biblioteca e a versão vulnerável. Nota-se que existe uma função *getVulnerableVersion*, que representa o tratamento para encontrar uma versão vulnerável baseado na faixa de versões vulneráveis fornecidas pela API do GitHub. Assim, é feito a geração do bando de dados CodeQL a partir do código-fonte armazenado pelo *download*, através da chamada da função *createDB*. Por fim, é feito a análise das consultas desenvolvidas através da chamada da função *runQuery*, informando nos argumentos o banco de dados CodeQL criado anteriormente e o caminho que contém o arquivo da consulta desenvolvida. Contudo, a variável *vulnerable* armazena a quantidade de bibliotecas vulneráveis e *packagesQtt* armazena a quantidade de bibliotecas analisadas. Nota-se que no código 7.22 apresenta somente a análise para a vulnerabilidade de injeção de comando por uma única consulta, entretanto na prática são executadas todas as consultas desenvolvidas relacionadas a cada vulnerabilidade.

O gráfico da figura 2 apresenta os resultados expressos no formato de quantidades de bibliotecas analisadas e porcentagem de bibliotecas sinalizadas como vulneráveis pelas consultas desenvolvidas, de acordo com cada vulnerabilidade apresentada na legenda e com a sua respectiva cor. Contudo, dentre 84 bibliotecas analisadas relacionadas à vulnerabilidade de injeção de comando, 55% foram sinalizadas como vulneráveis pelas consultas desenvolvidas. Da mesma forma, foram analisadas 47 de bibliotecas relacionadas à vulnerabilidade de injeção de código e 19% foram sinalizadas como vulneráveis. Para as bibliotecas relacionadas à vulnerabilidade de poluição de protótipo, foram analisadas 63 bibliotecas e 81% foram sinalizadas como vulneráveis.

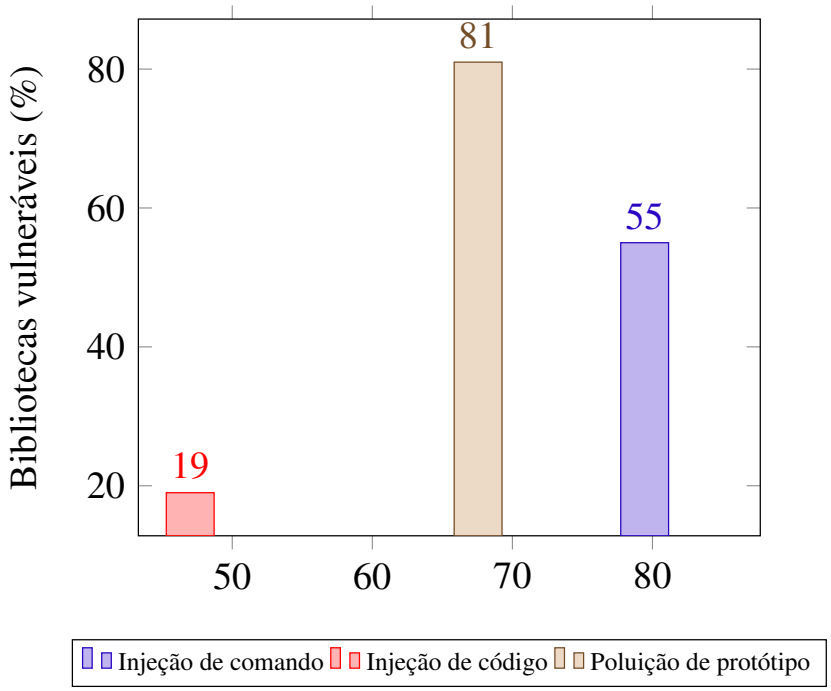

Figura 2 – Resultado das análises

8 CONCLUSÃO

Neste trabalho foi realizado o estudo teórico e prático sobre o mecanismo CodeQL, apresentando os conceitos sobre a linguagem QL, juntamente com as bibliotecas associadas aos tipos de análises de código possíveis. Foi realizado um breve estudo sobre os dez conjuntos de vulnerabilidades em aplicações web mais recorrentes, baseando-se na publicação da organização OWASP. A partir disto, foi identificado que as vulnerabilidades de injeção são as mais recorrentes e mais críticas, assim contribuindo para a decisão sobre o desenvolvimento das consultas para o CodeQL. Contudo, foram desenvolvidas duas consultas para a vulnerabilidade de injeção de comando, duas consultas para a vulnerabilidade de injeção de código e uma para a vulnerabilidade de poluição de protótipo.

Os resultados apresentados mostram que as vulnerabilidades de injeção de comando, poluição de protótipo e injeção de código obtiveram eficácia de aproximadamente 80%, 55% e 20%, respectivamente. Com este resultado pode-se concluir que o conjunto de consultas que abrangeram a maior capacidade de variantes das vulnerabilidades baseadas em termos de *sources* e *sinks* apresentaram melhores resultados. Consequentemente, pelo fato das vulnerabilidades de injeção de código poderem ter muitas variantes já que a linguagem JavaScript proporciona uma grande flexibilidade em termos de codificação, a existência de funções sensíveis que levam à vulnerabilidade de injeção de código pode aumentar consideravelmente; o que pode ser o motivo da menor eficácia comparado às outras análises.

O desenvolvimento das consultas e o objetivo para este trabalho não abordou o desempenho de cada consulta, o que pode ser feito nos trabalhos futuros relacionados a este trabalho, medindo o tempo de execução de cada consulta e propor melhorias para os algoritmos desenvolvidos. Da mesma forma, não foi abordado a questão dos falsos positivos que as consultas podem sinalizar, pois é possível ter funções que atuam como filtros sendo executadas antes de qualquer função sensível que pode levar à alguma vulnerabilidade, como apresentado na seção 5 do capítulo 5 sobre este tipo de modelagem através do predicado *isBarrier*.

Por meio deste trabalho pôde-se aprender o funcionamento do mecanismo CodeQL e sua flexibilidade em encontrar vulnerabilidades em aplicações de código abertos através das modelagens de consultas com a linguagem QL e as bibliotecas fornecidas pelo CodeQL. Apesar deste trabalho abordar as vulnerabilidades no contexto de aplicações web, é possível desenvolver consultas para outras linguagens suportadas pelo CodeQL, como apresentado no quadro 1. Da mesma forma, é possível desenvolver consultas para outros escopos de vulnerabilidades, como as vulnerabilidades relacionadas à exploração de binários.

REFERÊNCIAS

[1] *About CodeQL*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/codeql-overview/about-codeql/ (acesso em 04/03/2021).

[2] *About the QL language*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/about-the-ql-language/ (acesso em 04/03/2021).

[3] *Annotations*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/annotations/ (acesso em 04/03/2021).

[4] *Code Review Guide*. OWASP. 2021. Disponível em: https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf (acesso em 04/03/2021).

[5] *CodeQL library for JavaScript*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-javascript/ (acesso em 04/03/2021).

[6] *CodeQL repository*. GitHub. 2021. Disponível em: https://github.com/github/codeql (acesso em 04/03/2021).

[7] *Expressions*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/expressions/ (acesso em 04/03/2021).

[8] *Formulas*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/formulas/ (acesso em 04/03/2021).

[9] *GitHub Advisory Database*. GitHub. 2021. Disponível em: https://github.com/advisories (acesso em 04/03/2021).

[10] *Magic Methods*. PHP. 2021. Disponível em: https://www.php.net/manual/en/language.oop5.magic.php (acesso em 04/03/2021).

[11] *Modules*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/modules/ (acesso em 04/03/2021).

[12] *OWASP Top 10 - 2017*. OWASP. 2021. Disponível em: https://owasp.org/www-pdf-archive/OWASP_Top_10-2017-pt_pt.pdf (acesso em 04/03/2021).

[13] A. Pavel *et al.* ".QL: Object-Oriented Queries Made Easy". *In: GTTSE* (2007). Disponível em: https://www.semanticscholar.org/paper/QL%3A-Object-Oriented-Queries-Made-Easy-Moor-Sereni/d7be52e78154dbde1884f07510fa7901b9915277?p2df.

[14] A. Pavel *et al.* “QL: Object-oriented Queries on Relational Data”. *In: ECOOP* (2016). Disponível em: https://drops.dagstuhl.de/opus/volltexte/2016/6096/pdf/LIPIcs-ECOOP-2016-2.pdf.

[15] *Queries*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/queries/ (acesso em 04/03/2021).

[16] *Recursion*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/recursion/ (acesso em 04/03/2021).

[17] Sankarapandian Shivasurya. “Detecting Exploitable Vulnerabilities in Android Applications”. *In:* (2021). Disponível em: http://hdl.handle.net/10012/17034.

[18] *Source Code Analysis Tools*. OWASP. 2021. Disponível em: https://owasp.org/www-community/Source_Code_Analysis_Tools (acesso em 04/03/2021).

[19] *Static Code Analysis*. OWASP. 2021. Disponível em: https://owasp.org/www-community/controls/Static_Code_Analysis (acesso em 04/03/2021).

[20] Lv Tao *et al.* “RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection”. *In:* (2020). Disponível em: https://doi.org/10.1145/3372297.3423360.

[21] *Types*. GitHub. 2021. Disponível em: https://codeql.github.com/docs/ql-language-reference/ (acesso em 04/03/2021).

[22] Bandara Vinuri *et al.* “Fix that Fix Commit: A real-world remediation analysis of JavaScript projects”. *In: IEEE* (2020). Disponível em: https://doi.org/10.1109/SCAM51674.2020.00027.

APÊNDICE A – CÓDIGO COMPLETO DA CONSULTA 1

```
/*
  Command Injection - Default
*/

import DataFlow
import DataFlow::PathGraph
import semmle.javascript.NPM
import javascript

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "Command Injection - Default" }

   Node moduleExport(Module m) {
       exists(PropWrite pw |
          pw.getFile() = m.getFile() and
          pw.toString() = "module.exports" and
          result = pw.getRhs().getAFunctionValue()
       )
    }

    Node export(Module m) {
        result = m.getAnExportedValue(_).getAFunctionValue()
    }

  override predicate isSource(Node nd) {
    exists( NPMPackage pkg , Module m, FunctionNode node |
          m = pkg.getMainModule() and (node = export(m) or node =
              moduleExport(m)) and
            nd = node.getAFunctionValue().getAParameter()
      )
  }
  override predicate isSink(Node nd) {
    exists(ModuleImportNode imported, CallNode func |
      (
        func = imported.getAMemberCall("exec") or
        func = imported.getAMemberCall("execSync")
      ) and nd = func.getArgument(0)
    )
  }
```

```
39 }
40
41 from MyConfig cfg, PathNode source, PathNode sink
42 where cfg.hasFlowPath(source, sink)
43 select "vulnerable", source, sink
```

Código A.1 – Consulta 1

APÊNDICE B – CÓDIGO COMPLETO DA CONSULTA 2

```
/*
  Command Injection - split
*/

import DataFlow
import DataFlow::PathGraph
import semmle.javascript.NPM
import javascript

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "From a parameter to split function" }

   Node moduleExport(Module m) {
       exists(PropWrite pw |
          pw.getFile() = m.getFile() and
          pw.toString() = "module.exports" and
          result = pw.getRhs().getAFunctionValue()
       )
    }

    Node export(Module m) {
        result = m.getAnExportedValue(_).getAFunctionValue()
    }

  override predicate isSource(Node nd) {
    exists( NPMPackage pkg , Module m, FunctionNode node |
          m = pkg.getMainModule() and (node = export(m) or node =
              moduleExport(m)) and
          nd = node.getAFunctionValue().getAParameter()
      )
  }
  override predicate isSink(Node nd) {
    exists(CallExpr callSplit |
      callSplit.getCalleeName() = "split" and
      callSplit.getAnArgument().getStringValue().matches(" ") and
      nd = callSplit.flow()
    )
  }
}
```

```
39
40 class MyConfig2 extends TaintTracking::Configuration {
41   MyConfig2() { this = "From split function to sinks" }
42
43   override predicate isSource(Node nd) {
44     nd = nd
45   }
46
47   override predicate isSink(Node nd) {
48     exists(ModuleImportNode imported, CallNode func, Node arg |
49       (
50         func = imported.getAMemberCall("execFile") or
51         func = imported.getAMemberCall("execFileSync") or
52         func = imported.getAMemberCall("spawn") or
53         func = imported.getAMemberCall("spawnSync")
54       ) and (
55         //func.getArgument(0).toString().matches("curl") and
56         nd = func.getArgument(1)
57       )
58     )
59   }
60 }
61
62 from MyConfig cfg, PathNode source, PathNode sink, MyConfig2 cfg2, Node sink2
63 where cfg.hasFlowPath(source, sink) and cfg2.hasFlow(sink.getNode(), sink2)
64 select "vulnerable", source, sink2
```

Código B.1 – Consulta 2

APÊNDICE C – CÓDIGO COMPLETO DA CONSULTA 3

```
/*
  Code Injection - Eval
*/

import DataFlow
import DataFlow::PathGraph
import semmle.javascript.NPM
import javascript

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "Command Injection - eval" }

   Node moduleExport(Module m) {
       exists(PropWrite pw |
          pw.getFile() = m.getFile() and
          pw.toString() = "module.exports" and
          result = pw.getRhs().getAFunctionValue()
       )
    }

    Node export(Module m) {
        result = m.getAnExportedValue(_).getAFunctionValue()
    }

  override predicate isSource(Node nd) {
    exists( NPMPackage pkg , Module m, FunctionNode node |
          m = pkg.getMainModule() and (node = export(m) or node =
              moduleExport(m)) and
            nd = node.getAFunctionValue().getAParameter()
      )
  }
  override predicate isSink(Node nd) {
    exists(CallNode func |
      func.getCalleeName() = "eval" and
      nd = func.getAnArgument()
    )
  }
}

```

```
39 from MyConfig cfg, PathNode source, PathNode sink
40 where cfg.hasFlowPath(source, sink)
41 select "vulnerable", source, sink
```

Código C.1 – Consulta 3

APÊNDICE D – CÓDIGO COMPLETO DA CONSULTA 4

```
/*
  Code injection - new Function
*/

import DataFlow
import DataFlow::PathGraph import semmle.javascript.NPM
import javascript

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "Code injection - new Function" }

   Node moduleExport(Module m) {
       exists(PropWrite pw |
          pw.getFile() = m.getFile() and
          pw.toString() = "module.exports" and
          result = pw.getRhs().getAFunctionValue()
       )
    }

    Node export(Module m) {
        result = m.getAnExportedValue(_).getAFunctionValue()
    }

  override predicate isSource(Node nd) {
    exists( NPMPackage pkg , Module m, FunctionNode node |
          m = pkg.getMainModule() and (node = export(m) or node =
              moduleExport(m)) and
            nd = node.getAFunctionValue().getAParameter()
      )
  }

  override predicate isSink(Node nd) {
    exists(NewExpr v, AnalyzedVarDef a, CallExpr e, Variable va |
      v.getCalleeName() = "Function" and
      a.getRhs() = v.flow() and
      va = a.getAVariable() and
      va.getName() = e.getCalleeName() and
      e.getLocation().getStartLine() =
          va.getAnAccess().getLocation().getStartLine() and
```

```
38        nd = v.getAnArgument().flow()
39      )
40    }
41  }
42
43
44  from MyConfig cfg, PathNode source, PathNode sink
45  where cfg.hasFlowPath(source, sink)
46  select "vulnerable", source, sink
```

Código D.1 – Consulta 4

APÊNDICE E – CÓDIGO COMPLETO DA CONSULTA 5

```
/*
  Prototype pollution
*/

import javascript
import DataFlow
import DataFlow::PathGraph
import semmle.javascript.NPM

class MyConfig1 extends TaintTracking::Configuration {
  MyConfig1() { this = "From a parameter to an object reads" }

  override predicate isSource(Node nd) {
    exists(FunctionNode node |
          nd = node.getAFunctionValue().getAParameter()
      )
  }

  override predicate isSink(Node nd) {
    exists(PropRead prop |
      nd = prop.asExpr().flow()
    )
  }

}

class MyConfig2 extends TaintTracking::Configuration {
  MyConfig2() { this = "Object assignment" }

  override predicate isSource(Node nd) {
    nd = nd
  }

  override predicate isSink(Node nd) {
    exists( PropWrite prop |
        nd = prop.getRhs()
      )
  }

```

```
}

class MyConfig3 extends TaintTracking::Configuration {
  MyConfig3() { this = "From exported functions to functions vulnerable" }

   Node moduleExport(Module m) {
       exists(PropWrite pw |
          pw.getFile() = m.getFile() and
          pw.toString() = "module.exports" and
          result = pw.getRhs().getAFunctionValue()
       )
    }

    Node export(Module m) {
        result = m.getAnExportedValue(_).getAFunctionValue()
    }

 override predicate isSource(Node nd) {
    exists( NPMPackage pkg , Module m, FunctionNode node |
      m = pkg.getMainModule() and (node = export(m) or node = moduleExport(m))
          and
          nd = node.getAFunctionValue().getAParameter()
      )
  }

  override predicate isSink(Node nd) {
    nd = nd
  }

}

from MyConfig1 cfg1, MyConfig2 cfg2, MyConfig3 cfg3, Node source2, Node sink1,
    Node source1, PropWrite w, PropRead r, Node sink2, string propName
where
  cfg1.hasFlow(source2, sink1) and cfg2.hasFlow(sink1, sink2) and
  r = sink1 and w.getRhs() = sink2 and
  propName = r.getPropertyNameExpr().toString() and
  propName.matches(w.getPropertyNameExpr().toString()) and
  cfg3.hasFlow(source1, source2)
select "vulnerable", source, sink1, sink2, propName
```

Código E.1 – Consulta 5

APÊNDICE F – ALGORITMO PARA OBTER AS BIBLIOTECAS VULNERÁVEIS

```
1  const axios = require('axios');
2  const fs = require('fs');
3
4  const GIT_TOKEN = "<Your GitHub token>";
5
6  async function get(url) {
7    res = await axios.get(url, {
8      headers: {
9        'Authorization': `Bearer ${GIT_TOKEN}`
10     }
11   })
12
13   return res.data
14 }
15
16 async function post(url, query) {
17   res = await axios.post(url, query, {
18     headers: {
19       'Authorization': `Bearer ${GIT_TOKEN}`
20     }
21   })
22
23   return res.data
24 }
25 async function getInit(ecosystem) {
26   const data = {
27     query : `query { securityVulnerabilities(ecosystem: ${ecosystem},
         first:100) { totalCount, pageInfo{endCursor, hasNextPage}}}`
28   };
29
30   return (await post('https://api.github.com/graphql', data));
31 }
32
33 function parseCWE(CWES, nodes) {
34   let found = false;
35   let node = 0;
36   let CWE;
37
```

```
38    do {
39      CWE = 0;
40      do {
41        if (nodes[node].cweId === CWES[CWE++])
42          found = true;
43      } while (!found && CWE < CWES.length)
44      node++;
45    } while (!found && node < nodes.length)
46
47    return found;
48  }
49
50  async function getVulnerabilities(CWES) {
51    const vulnerabilities = [];
52
53    const init = await getInit('NPM');
54    let hasNextPage = init.data.securityVulnerabilities.pageInfo.hasNextPage;
55    let cursor = init.data.securityVulnerabilities.pageInfo.endCursor;
56    const limit = 100;
57    let count = 1;
58
59    while(hasNextPage) {
60      let data = {
61        query : `query {
62          securityVulnerabilities(ecosystem: NPM, first: ${limit}, after:
              "${cursor}") {
63            pageInfo {
64              endCursor,
65              hasNextPage
66            },
67            nodes {
68              advisory{
69                cvss{ score },
70                cwes(first: 10) {
71                  totalCount,
72                  nodes{ cweId }
73                }
74              },
75              package{name},
76              vulnerableVersionRange
77            }
78          }
```

```
79        }‘
80      };
81
82      res = await post('https://api.github.com/graphql', data);
83      array = res.data.securityVulnerabilities.nodes;
84      hasNextPage = res.data.securityVulnerabilities.pageInfo.hasNextPage;
85      cursor = res.data.securityVulnerabilities.pageInfo.endCursor;
86
87      console.log(cursor, hasNextPage, count++);
88
89      for (i of array) {
90        let obj = {};
91        obj.cwes = i.advisory.cwes.nodes;
92        if (obj.cwes.length > 0 && parseCWE(CWES, obj.cwes)) {
93          obj.package = i.package.name;
94          obj.vulnerableVersionRange = i.vulnerableVersionRange;
95          vulnerabilities.push(obj);
96        }
97      }
98    }
99    return vulnerabilities;
100 }
101
102 async function main() {
103   // Command injection
104   let CWES = ['CWE-1321', 'CWE-915'];
105   let vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null,
          2);
106   fs.writeFileSync('command-injection.json', vulnerabilities);
107
108   // Code injection
109   CWES = ['CWE-1321', 'CWE-915'];
110   vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null, 2);
111   fs.writeFileSync('code-iection.json', vulnerabilities);
112
113
114   // Prototype pollution
115   CWES = ['CWE-1321', 'CWE-915'];
116   vulnerabilities = JSON.stringify(await getVulnerabilities(CWES), null, 2);
117   fs.writeFileSync('prototype-pollution.json', vulnerabilities);
118 }
119
```

```
120 (async () => {
121   main();
122 })();
```

Código F.1 – Algoritmo para obter as bibliotecas vulneráveis

APÊNDICE G – ALGORITMO PARA REALIZAR AS ANÁLISES

```
const { execSync } = require('child_process');
const fs = require('fs');

const numThreads = 8;

// CodeQL path
const path = '~/codeql-source/ql/ql';

const NPMPath = "packages";

function createDB(language, sourceCode, db) {
  const cmd = `codeql database create --quiet --overwrite --threads
      ${numThreads} --language=${language} --source-root ${sourceCode} ${db}`;
  execSync(cmd);
}

function runQuery(db, query) {
  const cmd = `codeql query run --warnings=error --quiet --threads
      ${numThreads} --search-path ${path} --database ${db} ${query}`;
  const result = execSync(cmd, { maxBuffer: 1024*1024*1024
      }).toString().split('vulnerable');
  if (result.length > 1) {
    console.log('It\'s vulnerable!');
    return true;
  } else {
    console.log('It\'s not vulnerable!');
    return false;
  }
}

function NPMPackageInstall(package, version) {
  let cmd = `npm install --loglevel=error --prefix ./${NPMPath}
      "${package}@${version}"`;
  execSync(cmd);
  cmd = `mv ./${NPMPath}/node_modules/${vuln.package} ${vuln.package}`;
  execSync(cmd);
}

function clear(package) {
```

```
36    const cmd = `rm -rf ${package} ${package}-db`;
37    try {
38      execSync(cmd);
39    } catch {
40      console.log('Error delete');
41    }
42  }
43
44  function fixGreaterVersion(version, pos) {
45    if (pos === 0)
46      return version;
47
48    let res = '';
49    const array = version.split('.');
50
51    for (i in array) {
52      if (i == pos) {
53        if (array[i] !== '9')
54          res += (parseInt(array[i]) + 1).toString() + '.';
55        else {
56          if (pos - 1 === 0) {
57            res = (parseInt(array[0]) + 1).toString() + '.'
58          } else {
59            const _new = res.slice(0, res.length - 1);
60            return fixGreaterVersion(_new, _new.match(/\./g).length);
61          }
62        }
63      } else {
64        res += array[i] + '.';
65      }
66    }
67
68    return res.slice(0, res.length - 1);
69  }
70
71  function fixLessVersion(version, pos) {
72    if (pos === 0)
73      return version;
74
75    let res = '';
76    const array = version.split('.');
77
```

```
78    for (i in array) {
79      if (i == pos) {
80        if (array[i] !== '0')
81          res += (parseInt(array[i]) - 1).toString() + '.';
82        else {
83          return fixGreaterVersion(version, pos - 1);
84        }
85      } else {
86        res += array[i] + '.';
87      }
88    }
89
90    return res.slice(0, res.length - 1);
91  }
92
93  function getVulnerableVersion(version) {
94    let res;
95
96    if (version.includes('=')) {
97      res = version.split('=')[1].split(' ')[1];
98    } else if (version.includes('>=')) {
99      res = version.split('>=')[1].split(' ')[1];
100   } else if (version.includes('>')) {
101     const v = version.split('>')[1].split(' ')[1];
102     res = fixGreaterVersion(v,v.match(/\./g).length);
103   } else if (version.includes('<=')) {
104     res = version.split('<=')[1].split(' ')[1];
105   } else if (version.includes('<')) {
106     const v = version.split('<')[1].split(' ')[1];
107     res = fixLessVersion(v,v.match(/\./g).length);
108   }
109
110   return res.replaceAll(',', '');
111 }
112
113 function sleep(sec) {
114   return new Promise(resolve => setTimeout(resolve, (sec * 1000)));
115 }
116
117 function scan(files) {
118   const vulnerabilities = JSON.parse(fs.readFileSync(`./${files[0]}.json`,
        'utf8'));
```

```
119   let packagesQtt = 0;
120   let version;
121   let vulnerable = 0
122
123   for (vuln of vulnerabilities) {
124     console.log(`Qtt: ${packagesQtt} | vulnerable: ${vulnerable}`);
125     console.log(vuln.package,
          getVulnerableVersion(vuln.vulnerableVersionRange));
126
127     try {
128       version = vuln.vulnerableVersionRange
129       NPMPackageInstall(vuln.package, getVulnerableVersion(version,
            version.match(/\./g).length));
130       createDB('javascript', vuln.package, `${vuln.package}-db`);
131
132       for (query of files) {
133         if (runQuery(`${vuln.package}-db`, `./queries/${query}.ql`)) {
134           vulnerable++;
135           break
136         }
137       }
138
139       packagesQtt++;
140     } catch (error) {
141       console.log('Error!');
142     }
143     clear(vuln.package);
144     console.log('-'.repeat(100));
145   }
146
147 }
148
149 async function main() {
150
151   const config = {
152     'command-injection': ["command-injection", "command-injection-split"],
153     'code-injection': ["code-injection", "code-injection-function"],
154     'prototype-pollution': ["prototype-pollution"]
155   }
156
157   for (i in config) {
158     console.log('-'.repeat(100));
```

```
159     console.log('Scanning:', i)
160     console.log('-'.repeat(100), '\n\n');
161     scan(config[i])
162   }
163
164 }
165
166 (async () => {
167   await main();
168 })();
```

Código G.1 – Algoritmo para realizar as análises

APÊNDICE H – ARTIGO

Análise de código com CodeQL para descoberta de vulnerabilidades em aplicações web

Renan R. S. dos Santos[1], **Carla M. Westphall**[2]

[1]Departamento de Informática e Estatística - INE
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brasil

renan.rocha@grad.ufsc.br, carla.merkle.westphall@ufsc.br

Abstract. *Due to the fact that open-source projects are often maintained in Git repositories, Git providers may add an extra layer to the code update process to help projects automatically fix security vulnerabilities before the codes are in production. GitHub is one of the most popular Git providers used by large organizations and provides a solution through the CodeQL engine, which has a query language similar to SQL syntax and performs code analysis from queries. Thus, with CodeQL is possible to write a query to find security vulnerabilities in source code based on data provided by the user and sensitive functions which lead to vulnerability, known as taint analysis. Therefore, this work aims to prepare queries for the CodeQL engine to find vulnerabilities in web applications developed in JavaScript.*

Resumo. *Devido ao fato que os projetos de código aberto serem frequentemente mantidos em repositórios Git, os provedores Git podem adicionar uma camada a mais no processo de atualização de código para ajudar os projetos a corrigirem vulnerabilidades de segurança de maneira automatizada antes dos códigos estarem em produção. O GitHub é um dos provedores Git mais populares utilizado por grandes organizações e fornece uma solução por meio do mecanismo CodeQL, que possui uma linguagem de consulta semelhante à sintaxe SQL e realiza análise de código a partir de consultas. Assim, com o CodeQL é possível escrever uma consulta para encontrar vulnerabilidades de segurança em códigos-fonte com base nos dados providos pelo usuário e funções sensíveis que levam a alguma vulnerabilidade, conhecido como análise de contaminação. Portanto, este trabalho tem como objetivo elaborar consultas para o mecanismo CodeQL para encontrar vulnerabilidades em aplicações web desenvolvidas em JavaScript.*

1. Introdução

As análises de códigos são comumente utilizadas no processo de desenvolvimento e atualização de *softwares* para apresentar possíveis erros que podem levar a um problema de execução, ou até mesmo alguma vulnerabilidade de segurança que possa ser explorada por um usuário malicioso. Estas análises podem ser realizadas através de especialistas em análise de código ou por *softwares* que realizam este processo de forma automatizada. Nos softwares, as análises podem ser realizadas de forma estática, baseando-se somente no código-fonte ou de forma dinâmica, baseando-se na aplicação em execução [OWASP 2021c]. O resultado das análises pode apresentar especificamente em qual área

do código ocorre um erro ou apresentar partes sensíveis do código para que futuramente especialistas possam analisar e validar o problema. Isto pode ocorrer em casos complexos em que *softwares* não são capazes de distinguir se um erro realmente está ocorrendo; como as análises de segurança, que através do fluxo de execução podem ter mitigadores sendo executados antes de uma área de código vulnerável, o que resulta em um código não vulnerável.

Os *softwares* de análises de código para encontrar problemas de segurança cada vez mais estão sendo necessários devido à imensa quantidade de códigos sendo criados e atualizados. Assim, um processo automatizado ajuda a solucionar as vulnerabilidades já conhecidas e otimizam o tempo em que um especialista poderia demorar para encontrar as falhas, ou possivelmente as falhas poderiam não ser encontradas. Várias soluções são lançadas todos os anos apresentando novas funcionalidades e diferentes abordagens para melhorar a eficácia e eficiência das análises e, dentre as diversas soluções públicas e privadas existentes [OWASP 2021b], o CodeQL é um dos mecanismos possíveis para realizar análise de código estática em códigos abertos, que está atualmente integrado nos repositórios do GitHub. Esta solução propõe a possibilidade de realizar análise de código baseado em consultas, similarmente às consultas feitas na linguagem SQL em banco de dados relacionais. Assim, é possível modelar um tipo de vulnerabilidade já conhecida e suas variantes através da escrita de uma consulta, e portanto utilizá-la para encontrar vulnerabilidades em qualquer projeto de código aberto.

2. Vulnerabilidades em aplicações web

Os *softwares* inseguros podem causar problemas de forma direta ou indireta a infraestrutura financeira, de saúde, defesa, energia e outras infraestruturas críticas. À medida que os *softwares* estão se tornando cada vez mais críticos, complexos e interligados, a dificuldade em garantir a segurança cresce exponencialmente [OWASP 2021a]. O ritmo acelerado dos processos modernos de desenvolvimento de *softwares* faz com que seja ainda mais crítico identificar estes riscos de forma rápida e precisa [OWASP 2021a]. Assim, a organização OWASP é responsável por publicar as dez categorias de vulnerabilidades mais recorrentes e críticas dos últimos anos através da OWASP Top 10. Portanto, o objetivo desta seção é descrever sucintamente as vulnerabilidades do conjunto de vulnerabilidades de injeção, pois representa o conjunto mais crítico baseado na publicação de 2017.

As falhas de injeção ocorrem quando dados não confiáveis são enviados para um interpretador como parte de um comando ou consulta. A consequência da injeção dos dados providos pelo atacante é fazer o interpretador executar comandos não pretendidos.

2.1. Injeção de comando

O código 1 ilustra um exemplo de vulnerabilidade de injeção de comando através da biblioteca *child_process* do *runtime* Node.js. Para este exemplo, o usuário malicioso cria um arquivo nomeado por *file* no sistema operacional que está executando a aplicação vulnerável. Descrição do código 1:

- Linha 1: Importação da função *execSync* da biblioteca *child_process* para possibilitar a execução de comandos no sistema operacional de forma síncrona.
- Linha 2: Constante *user* representando um usuário fornecido por um usuário qualquer.

```
1 const { execSync } = require('child_process');
2 const user = "test;touch file";
3 console.log(execSync('id ' + user));
```

Código 1. Exemplo de injeção de comandos no sistema operacional

- Linha 3: Realiza a execução do comando *id* para verificar se o usuário existe no sistema operacional. Posteriormente, utiliza o comando *console.log* para imprimir o resultado na tela.

O ponto principal na injeção deste tipo de exploração acontece devido ao fato que o atacante pode realizar a concatenação de comandos. Assim, o conteúdo de *id* do código 1 mostra a concatenação realizada através do símbolo *;* e posteriormente a execução do comando *touch*, que realiza a criação de um arquivo. Contudo, neste cenário o atacante poderia inserir qualquer tipo de comando do sistema operacional, tornando o sistema completamente vulnerável.

2.2. Injeção de código

A vulnerabilidade de injeção de código acontece similarmente à vulnerabilidade de injeção de comando, entretanto ao invés de executar um comando diretamente no sistema operacional, um código é interpretado. O exemplo de código 2 ilustra a constante *data* sendo executada como argumento da chamada da função *console.log*, implicando na impressão do conteúdo da constante *data* na tela. Nota-se que este comportamento está sendo executado pela função *eval*, que possibilita executar código JavaScript através de uma *string*.

```
1 const data = "test";
2 eval('console.log("' + data + '")');
```

Código 2. Exemplo de injeção de código

Neste exemplo, supondo que o atacante tem acesso a esta constante *data*, é possível realizar também uma concatenação de comando, mas neste caso o atacante teria somente a possibilidade de executar códigos JavaScript a princípio. Entretanto, em alguns casos é possível importar a biblioteca *child_process* diretamente e executar comandos no sistema operacional, conforme ilustrado no código 3.

```
1 const data = 'test"); require("child_process").execSync("touch
     file") //';
2 eval('console.log("' + data + '")');
```

Código 3. Exemplo de injeção de código 2

2.3. Poluição de protótipo

A linguagem JavaScript é baseada em protótipos, sendo assim, quando variáveis primitivas, objetos e funções são criados, esta criação ocorre por meio da instanciação de um

protótipo que representa o valor a ser instanciado. Por exemplo, ao criar uma variável do tipo *string*, esta criação baseia-se em um protótipo que representa o tipo *string*. Este protótipo pode ser acessado por meio da palavra-chave *__proto__* de qualquer objeto em JavaScript, como pode ser visto no código 4.

```
1 let a = 'this is a string';
2 console.log(a.__proto__);
```

Código 4. Exemplo de acesso __proto__

Em JavaScript é possível criar novos protótipos e alterar os protótipos existentes. O código 5 ilustra como é possível alterar a função *toString* do protótipo do tipo *integer*, que é uma função que converte qualquer valor numérico em um valor do tipo *string*. Contudo, nota-se neste código que ao modificar a função *toString* do protótipo e instanciar outra variável do mesmo tipo, a função *toString* continua modificada, pois a criação dela é baseada no protótipo que foi alterado.

```
1 let number = 10;
2 console.log(number.toString()); // '10'
3
4 number.__proto__.toString = () => { return 'hello'};
5 console.log(number.toString()); // 'hello'
6
7 let test = 100;
8 console.log(test.toString()); // 'hello'
```

Código 5. Exemplo de alteração do protótipo

Descrição do código 5:

- Linha 1: Instanciação da variável *number*, representando um valor numérico.
- Linha 2: Imprime o valor da variável *number* através da chamada da função *toString*, retornando o valor *10*.
- Linha 4: Altera a função *toString* do protótipo para retornar o valor *hello*.
- Linha 5: Realiza o mesmo procedimento da linha 2, porém desta vez imprimindo o valor *hello*.
- Linha 7: Instanciação de outra variável nomeada como *test* para representar outro valor numérico.
- Linha 8: Imprime o valor da variável *test* através da chamada da função *toString*, retornando o valor *hello* também.

As vulnerabilidades neste tipo de abordagem ocorrem quando o dado de um usuário pode modificar o protótipo de alguma forma. Na prática, estas vulnerabilidades ocorrem através dos objetos que são enviados geralmente através de APIs, em que são enviados os dados no formato JSON e é necessário realizar a conversão de uma *string* no formato JSON da requisição HTTP para um objeto em JavaScript. Assim, neste processo de conversão é necessário copiar as propriedades e seus valores correspondentes em um objeto em JavaScript.

```
{'__proto__': {'admin': true}}
```

Código 6. Exemplo de alteração do protótipo no formato JSON

Se um usuário envia o conteúdo de um arquivo JSON ilustrado no código 6 e o processo de conversão não possua nenhum filtro para bloquear a *string* *__proto__*, esta conversão realizará algum processo similar ao código 7 e, consequentemente, a aplicação se torna vulnerável.

```
1  const userString = "{'__proto__': {'admin': true}}";
2  const userJson = JSON.parse(userString);
3  let obj = {}
4  console.log(merge(obj, userJson));
5
6  function merge(obj, userJson) {
7    let value;
8
9    for (key in userJson) {
10     value = userJson[key];
11     if (typeof value === 'object') {
12       obj[key] = merge(obj[key], value);
13     } else {
14       obj[key] = value;
15     }
16
17   }
18   return obj;
19 }
```

Código 7. Exemplo de conversão de um objeto JSON em objeto JavaScript

Descrição do código 7:

- Linha 1: *userString* representa um conteúdo JSON no formato de *string*, para ilustrar o caso do conteúdo recebido através das requisições HTTP.
- Linha 2: Realiza a conversão de *string* para um objeto JSON e atribui o resultado para a constante *userJson*.
- Linha 3: Cria um objeto JavaScript vazio que irá receber o resultado da conversão.
- Linha 4: Realiza a chamada da função *merge* para realizar a conversão do objeto JSON para o formato JavaScript e imprime o resultado na tela.
- Linha 9: Percorre através do laço de repetição as propriedades existentes no objeto JSON. Neste exemplo, só há uma propriedade nomeada como *__proto__*.
- Linha 10: Obtém o conteúdo correspondente à propriedade *__proto__* e armazena na variável *value*.
- Linha 11: Verifica se o tipo do valor da linha 10 é outro objeto JSON.
- Linha 12: Caso o resultado da verificação da linha 11 for verdadeiro, é necessário chamar a função *merge* novamente para realizar a conversão.
- Linha 14: Caso o resultado da linha 10 for falso, então ocorre atribuição no objeto JavaScript.

O código 7 ilustra que quando um usuário informa um conteúdo malicioso é possível poluir o protótipo dos objetos JavaScript. Esta falha é crítica, pois com a modificação do protótipo é possível fazer com que a aplicação vulnerável se comporte de maneira diferente, podendo levar à execução remota de código quando é possível modificar as funções de forma similar a abordagem da vulnerabilidade de injeção de código.

3. CodeQL

O CodeQL é um projeto de código aberto coordenado pelo GitHub que atua como um mecanismo para automatizar verificações de segurança por pesquisadores para realizar análises de variantes de vulnerabilidades existentes via linguagem de consulta [GitHub 2021a, Pavel et al. 2007a]. No CodeQL as vulnerabilidades de segurança são modeladas como consultas que podem ser executadas em bancos de dados extraídos do código-fonte [GitHub 2021a]. O mecanismo é composto pela funcionalidade de gerar os bancos de dados a partir de códigos e por um conjunto de bibliotecas que tornam possíveis as análises de variantes através de consultas feitas com o suporte da linguagem QL. Assim, o fluxo de uma análise acontece da seguinte forma:

1. Criação do banco de dados baseado no código.
2. Execução de consultas no banco de dados.
3. Interpretação dos resultados da consulta.

As análises de código com CodeQL através das consultas feitas em linguagem QL necessitam de bancos de dados gerados a partir dos códigos de origem. Portanto, é extraída primeiro uma única representação relacional de cada arquivo de origem na base de código [GitHub 2021a]. Para linguagens compiladas, a extração funciona monitorando o processo normal de construção. Cada vez que um compilador é chamado para processar um arquivo, é feita uma cópia desse arquivo e todas as informações relevantes sobre o código-fonte são coletadas [GitHub 2021a]. Isto inclui dados sintáticos sobre a árvore de sintaxe abstrata, e dados semânticos sobre a vinculação de nomes e informações de tipagem [GitHub 2021a]. E para linguagens interpretadas, o extrator é executado diretamente no código-fonte, resolvendo as dependências para fornecer uma representação precisa da base de código.

Em CodeQL existe uma extensa biblioteca para analisar códigos escritos na linguagem JavaScript, podendo apresentar informações sobre o código-fonte em diferentes níveis, tais como: textual, léxico, sintático, fluxo de controle e fluxo de dados. As classes nesta biblioteca apresentam os dados de um banco de dados CodeQL em uma forma orientada a objetos [Pavel et al. 2007b], fornecendo abstrações e predicados [Pavel et al. 2016]. A biblioteca é implementada como um conjunto de módulos QL, no qual a maioria podem ser importados diretamente através do módulo *javascript.qll* [GitHub 2021d, GitHub 2021b].

4. Implementação

Esta seção consiste na apresentação das consultas elaboradas para identificar as vulnerabilidades em códigos de aplicações desenvolvidas em JavaScript, especificamente de bibliotecas desenvolvidas para Node.js. A abordagem para identificar as vulnerabilidades através das consultas baseia-se na análise de contaminação, que consiste na identificação das variáveis que podem ser contaminadas com entradas controláveis pelo usuário

```
exports.function = function () {};
module.exports = function () {};
```

Código 9. Formas de exportação de funções em JavaScript

(*sources*) e as rastreia para possíveis funções vulneráveis (*sinks*). Se algum dado de entrada contaminado for encaminhado para alguma função vulnerável, é então sinalizado como uma vulnerabilidade. A modelo de consulta para a análise de contaminação pode ser visto no código 8, em que no predicado *isSource* são definidas as entradas controláveis pelo usuário e no predicado *isSink* são definidas as funções vulneráveis. Portanto, na cláusula *where* do código 8 consiste apenas na verificação se há algum fluxo de dado a partir dos *sources* até os *sinks*, através do predicado *hasFlowPath*.

```
override predicate isSource(Node nd) {
  ...
}

override predicate isSink(Node nd) {
  ...
}

from MyConfig cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select source, sink
```

Código 8. Modelo de consulta para análise de contaminação

Nas subseções sobre as consultas, são descritas primeiramente como ocorrem as vulnerabilidades nos códigos em JavaScript e posteriormente é apresentada a modelagem através do modelo de análise de contaminação apresentado no código 8, a partir da definição dos *sources* e *sinks* de cada vulnerabilidade. As implementações das consultas para encontrar as vulnerabilidades foram organizadas da seguinte forma:

- Injeção de comando: Consultas 1 e 2.
- Injeção de código: Consultas 3 e 4.
- Poluição de protótipo: Consulta 5.

4.1. Entradas de dados controláveis (sources)

Para todas as consultas foi definido como entrada de dados todos os argumentos de funções em que o usuário tem acesso, ou seja, os argumentos de funções exportadas pelo código a ser analisado. Assim, quando ocorre a importação de um código em JavaScript, é possível acessar somente as funções que forem explicitamente exportadas. As duas linhas do código 9 ilustram as duas formas de exportar uma função de um código em JavaScript.

As bibliotecas utilizadas pelas aplicações em Node.js são gerenciadas através do *software* NPM. O CodeQL proporciona uma biblioteca com alguns predicados para obter algumas informações sobre as bibliotecas desenvolvidas para o NPM, chamada de *semmle.javascript.NPM*.

```
1  import javascript
2  import DataFlow
3  import semmle.javascript.NPM
4
5  Node moduleExport(Module m) {
6    exists(PropWrite pw |
7      pw.getFile() = m.getFile() and
8      pw.toString() = "module.exports" and
9      result = pw.getRhs().getAFunctionValue()
10   )
11 }
12
13 Node export(Module m) {
14    result = m.getAnExportedValue(_).getAFunctionValue()
15 }
16
17 from NPMPackage pkg, Module m, FunctionNode node
18 where m = pkg.getMainModule() and
19   (node = export(m) or node = moduleExport(m))
20 select node.getAFunctionValue().getAParameter()
```

Código 10. Consulta para obter os argumentos das funções exportadas

O código 10 ilustra a consulta completa para obter as duas formas de exportação de função baseada em uma biblioteca do NPM. Descrição do código:

- Linha 1, 2 e 3: Importação da biblioteca JavaScript, importação da biblioteca que proporciona obter os fluxos de dados e a importação da biblioteca para obter informações sobre as bibliotecas do NPM.
- Linha 5 a 11: Obtém as funções exportadas através de *module.exports*:
  - Linha 5: Parâmetro *Module* representa um módulo de uma biblioteca NPM.
  - Linha 6: Obtém uma variável do tipo *PropWrite* para obter a escrita no formato apresentado no código 9.
  - Linha 7: Define que o arquivo a ser analisado deve ser o mesmo arquivo que contém a biblioteca.
  - Linha 8: Define que a expressão de atribuição, como uma *string*, deve ser igual a *module.exports*.
  - Linha 9: Define que o resultado será a função da parte direita da atribuição de *module.exports*.
- Linha 13 à 15: Obtém as funções exportadas através de *exports.function*.
  - Linha 13: Parâmetro *Module* representa um módulo de uma biblioteca NPM.
  - Linha 14: O resultado é definido pelo predicado *getAnExportedValue* fornecido pela biblioteca *semmle.javascript.NPM*. Assim, o predicado *getAnExportedValue* é invocado com o parâmetro *don't care*, representando qualquer valor de exportação; e posteriormente é obtido a função que este valor representa.

```
override predicate isSource(Node nd) {
  exists( NPMPackage pkg , Module m, FunctionNode node |
       m = pkg.getMainModule() and (node = export(m) or node =
          moduleExport(m)) and
        nd = node.getAFunctionValue().getAParameter()
    )
}

override predicate isSink(Node nd) {
  exists(ModuleImportNode imported, CallNode func |
    (
      func = imported.getAMemberCall("exec") or
      func = imported.getAMemberCall("execSync")
    ) and nd = func.getArgument(0)
  )
}
```

Código 12. Consulta injeção de comando direta

- Linha 17 à 20: Consulta para obter os argumentos das funções exportadas.
  - Linha 17: Cláusula *from* para definir as variáveis do tipo *NPMPackage*, representando uma biblioteca; *Module*, representando um módulo; e *FunctionNode*, representando um nó do fluxo de dados de uma função.
  - Linha 18: Cláusula *where* para obter o módulo principal exportado pela biblioteca NPM.
  - Linha 19: Continuação da cláusula *where* para definir que as funções podem ser do tipo dos predicados criados anteriormente para representar as duas formas de exportação de função.
  - Linha 20: Cláusula *select* para selecionar os argumentos das funções que satisfazem a cláusula *where*.

4.2. Consulta 1: Injeção de comando de forma direta

Esta consulta representa a vulnerabilidade de injeção de comando por meio da biblioteca *child_process*. Como esta biblioteca apresenta diferentes maneiras de executar comandos no sistema operacional, esta consulta aborda a execução das funções *exec* (função assíncrona) e *execSync* (função síncrona), em que ambas recebem o comando a ser executado pelo primeiro argumento da função, como ilustrado no código 11.

```
const { exec, execSync } = require('child_process');
exec('command 1');
execSync('command 2');
```

Código 11. Exemplo de injeção de comando

Esta consulta define no predicado *isSource* os argumentos de funções exportadas obtidos pela consulta 10, como ilustrado no código 12.

O objetivo do predicado *isSink* é obter os nós do fluxo de dados que representam o primeiro argumento das funções *exec* e *execSync* que são executadas através da biblioteca *child_process*. Descrição:

- Linha 10: Define a variável do tipo *ModuleImportNode* para obter um módulo que é importado no código em análise e a variável do tipo *CallNode*, que representa um nó de chamada de função.
- Linha 12: Através da biblioteca importada, obtém uma função que esta biblioteca acessa, cujo o nome é *exec*.
- Linha 13: Realiza o mesmo procedimento da linha 12 para o nome *execSync*.
- Linha 14: Define que o nó *sink* é representado pelo primeiro argumento das funções obtidas nas linhas 17 ou 18.

As cláusulas *from*, *where* e *select* seguem o mesmo padrão da consulta apresentada no código 8. Portanto, a análise de contaminação é validada através do predicado *hasFlowPath*, em que verifica se há algum fluxo a partir do nós *source* até os nós *sink*. A consulta completa pode ser vista na referência [dos Santos 2021].

4.3. Consulta 2: Injeção de comando por meio da função split

Esta consulta representa uma extensão da consulta anterior para obter as outras maneiras de executar comandos no sistema operacional através da biblioteca *child_process*, tais como: *execFile*, *execFileSync*, *spawn* e *spawnSync*. O comportamento destas funções ocorrem diferente das funções da consulta anterior, em que estas funções recebem no primeiro parâmetro o binário a ser executado e os argumentos para este binário no segundo argumento como *array* de *strings*, como ilustrado no código 13.

```
1 const { execFile, execFileSync, spawn, spawnSync } =
     require('child_process');
2
3 execFile('binary', ['argument 1', 'argument 2', '...']);
4 execFileSync('binary', ['argument 1', 'argument 2', '...']);
5 spawn('binary', ['argument 1', 'argument 2', '...']);
6 spawnSync('binary', ['argument 1', 'argument 2', '...']);
```

Código 13. Funções para executar comando do sistema operacional com 2 argumentos

Existem diferentes formas de como uma biblioteca pode tratar os dados providos pelo usuário até efetivamente executar estas funções e tornar o código vulnerável. Esta consulta aborda a função *split* como uma possível função a ser utilizada pelas bibliotecas com o objetivo de tornar a entrada de dados do usuário, que é uma *string*, em um *array* de *strings* para representar os argumentos de um binário, como pode ser visto no código 14. Esta abordagem ainda necessita da verificação de qual binário será executado, pois para haver de fato a execução de comandos no sistema operacional o binário precisa fornecer algum argumento que possibilite este tipo de exploração.

```
1 const { execFile } = require('child_process');
2
```

```
3 function exportedFunction(userData) {
4   const arguments = userData.split(' ');
5   execFile('bin', arguments);
6 }
```

Código 14. Exemplo de injeção de comando por meio da função split

Baseando-se nesta abordagem, foram definidas duas análises de contaminação para esta consulta. Uma análise para o fluxo de dados dos argumentos das funções exportadas até a função *split* e outra análise para o fluxo de dados da função *split* até as funções sensíveis da biblioteca *child_process*. O conteúdo do predicado *isSource* da primeira análise é representado pelo mesmo conteúdo de *isSource* do código 12 e o conteúdo de *isSink* pelo código 15.

```
1 override predicate isSink(Node nd) {
2    exists(CallExpr callSplit, AnalyzedVarDef split |
3       callSplit.getCalleeName() = "split" and
4       callSplit.getAnArgument().getStringValue().matches(" ")
           and
5       nd = callSplit.flow()
6    )
7 }
```

Código 15. Predicado isSink para a primeira análise.

Descrição do código 15:

- Linha 2: Define as variáveis do tipo *CallExpr*, que representa uma expressão de chamada de função; e *AnalyzedVarDef*, que representa uma definição de variável em qualquer forma de definição possível em JavaScript.
- Linha 3: Define que o nome da função que está sendo chamada é igual a *split*.
- Linha 4: Define que o argumento desta função deve ser igual a um espaço em branco, para representar o delimitador da criação do *array* por espaço em branco utilizado pela função *split*.
- Linha 5: Define que o nó *sink* é representado pelo nó da expressão da chamada de função.

Para a segunda análise é utilizado para o predicado *isSource* o nó que representa a função *split*, que é representado pelo nó *sink* da análise de contaminação anterior. Portanto, o nó utilizado como parâmetro do predicado *isSource* será informado através do resultado da consulta anterior na cláusula *where*. O predicado *isSink* é similar ao código 12, entretanto são utilizadas as funções da biblioteca *child_process* para este escopo, e o argumento que torna o código vulnerável é representado pelo argumento 2 (posição 1), como ilustrado no código 16. Na linha 9 é definido que o primeiro argumento da função sensível deve ser igual ao binário *curl*, porém em análises reais deve-se adicionar outros nomes de binários em que existe alguma possibilidade de execução de comandos do sistema operacional via argumentos.

Por fim, as cláusulas *from*, *where* e *select* necessitam das modificações descritas anteriormente, como a definição de duas análises de contaminação e definir que o nó

```
override predicate isSink(Node nd) {
   exists(ModuleImportNode imported, CallNode func, Node arg |
     (
       func = imported.getAMemberCall("execFile") or
       func = imported.getAMemberCall("execFileSync") or
       func = imported.getAMemberCall("spawn") or
       func = imported.getAMemberCall("spawnSync")
     ) and (
       func.getArgument(0).toString().matches("curl") and
       nd = func.getArgument(1)
     )
   )
}
```

Código 16. Predicado isSink para a segunda análise.

sink da primeira análise representa o nó *source* da segunda análise, como pode ser visto no código 17. O código completo desta consulta pode ser visto na referência [dos Santos 2021].

```
override predicate isSink(Node nd) {
from MyConfig cfg, PathNode source, PathNode sink, MyConfig2
   cfg2, Node sink2
where cfg.hasFlowPath(source, sink) and
   cfg2.hasFlow(sink.getNode(), sink2)
select source, sink2
```

Código 17. Consulta injeção de comando por meio da função split

4.4. Consulta 3: Injeção de código por meio da função *eval*

A função *eval* é utilizada para executar códigos JavaScript a partir de *strings*. Portanto, a abordagem desta consulta consiste em verificar se há algum fluxo de dados a partir dos parâmetros das funções exportadas até o parâmetro de uma chamada da função *eval*. A implementação desta consulta é semelhante a consulta da seção 1, com a diferença no conteúdo do predicado *isSink*, como pode ser visto no código 18. A consulta completa pode ser vista na referência [dos Santos 2021].

```
override predicate isSink(Node nd) {
   exists(CallNode func |
     func.getCalleeName() = "eval" and
     nd = func.getAnArgument()
   )
}
```

Código 18. Predicado isSink para para consulta de injeção de código por meio da função eval

Descrição do código 18:

- Linha 2: Define uma variável do tipo *CallNode*, que representa um nó de expressão de chamada de função.
- Linha 3: Define que o nome da função que está sendo chamada é igual a *eval*.
- Linha 4: Define que o nó *sink* é representado pelo argumento da função *eval*.

4.5. Consulta 4: Injeção de código por meio de criação de função

Além da função *eval*, em JavaScript é possível também executar código a partir da definição de uma nova função e a invocação da mesma, como ilustrado no código 19.

```
1 function exportedFunction(userData) {
2   const newFunction = new Function(userData);
3   newFunction();
4 }
```

Código 19. Exemplo de execução de código a partir da criação e invocação de nova função

Esta consulta segue o mesmo modelo da seção anterior, com a modificação do predicado *isSink*, como pode ser visto no código 20. A consulta completa pode ser vista na referência [dos Santos 2021].

```
1  override predicate isSink(Node nd) {
2     exists(NewExpr v, AnalyzedVarDef a, CallExpr e, Variable va |
3       v.getCalleeName() = "Function" and
4       a.getRhs() = v.flow() and
5       va = a.getAVariable() and
6       va.getName() = e.getCalleeName() and
7       e.getLocation().getStartLine() =
           va.getAnAccess().getLocation().getStartLine() and
8       nd = v.getAnArgument().flow()
9     )
10 }
```

Código 20. Predicado isSink

Descrição do código 20:

- Linha 2: Definição das variáveis:
  - *NewExpr*: Representa uma expressão que possui a palavra-chave *new*. Por exemplo: *new Function*.
  - *AnalyzedVarDef*: Representa uma definição de variável em qualquer formato possível em JavaScript.
  - *CallExpr*: Representa uma expressão de chamada de função.
  - *Variable*: Representa uma variável qualquer.
- Linha 3: Define que o nome para a expressão *new* deve ser igual a *Function*. Por exemplo: *new Funtion()*.
- Linha 4: Define que a definição de uma variável deve conter a expressão *new* da linha 3 no lado direito da atribuição. Por exemplo: *let a = new Function()*.

```
override predicate isSink(Node nd) {
   exists( PropWrite prop |
      nd = prop.getRhs().asExpr().flow()
   )
}
```

Código 22. Predicado isSink da segunda análise

- Linha 5: Obtém uma variável qualquer para o tipo *Variable*.
- Linha 6: Define que o nome da variável da linha 5 deve ser o mesmo nome da função chamadora do tipo *CallExpr*.
- Linha 6: Define que o local da chamada da função é o mesmo local de um acesso da variável da linha 5. Isto é necessário para verificar se um acesso da variável representa uma chamada de função ao invés de outro uso.
- Linha 7: Define que o nó *sink* é um argumento da expressão *new Function*.

4.6. Consulta 5: Poluição de protótipo

Esta consulta foi modelada através de três análises de contaminação. A primeira análise consiste no fluxo de dados a partir dos argumentos de qualquer função até a leitura do conteúdo associado a uma propriedade de um objeto, por exemplo: *value = userObj[key]*. A segunda análise baseia-se no fluxo de dados a partir da leitura do conteúdo associado a uma propriedade da primeira análise até a atribuição do conteúdo de uma propriedade de outro objeto, da seguinte forma: *obj[key] = value*. A terceira análise é baseada no fluxo de dados dos argumentos das funções exportadas até os argumentos da função da primeira análise, definidos através da cláusula *where*.

```
override predicate isSink(Node nd) {
   exists(PropRead prop |
      nd = prop.asExpr().flow()
   )
}
```

Código 21. Predicado isSink da primeira análise

O código 21 ilustra o predicado *isSink* para a primeira análise. Descrição:

- Linha 2: Definição de uma variável do tipo *PropRead*, que representa a leitura do conteúdo de uma propriedade de um objeto.
- Linha 3: Define que o nó *sink* é representado por qualquer expressão do tipo *PropRead*.

O código 22 ilustra o predicado *isSink* para a segunda análise. Descrição:

- Linha 2: Definição de uma variável do tipo *PropWrite*, que representa a atribuição de conteúdo de uma propriedade de um objeto.
- Linha 3: Define que o nó *sink* é representado por qualquer expressão que esteja no lado direito da atribuição de *PropWrite*.

O código 23 apresenta o conteúdo necessário para as cláusulas *from*, *where* e *select*. A consulta completa pode ser vista na referência [dos Santos 2021]. Descrição:

```
1 from MyConfig1 cfg1, MyConfig2 cfg2, MyConfig3 cfg3, Node
     source1, Node source2, Node sink1, PropWrite w, PropRead r,
     Node sink2, string propName
2 where cfg1.hasFlow(source2, sink1) and cfg2.hasFlow(sink1,
     sink2) and
3   r = sink1 and w.getRhs() = sink2 and
4   propName = r.getPropertyNameExpr().toString() and
5   propName.matches(w.getPropertyNameExpr().toString()) and
6   cfg3.hasFlow(source1, source2)
7 select source1, source2, sink1, sink2
```

Código 23. Consulta para poluição de prototipo

- Linha 1: Definição de variáveis:
  - *MyConfig*: Classe que representa a configuração da análise de contaminação, em que *cfg1* representa a primeira análise, *cfg2* a segunda análise e *cfg3* a terceira análise.
  - *Node*: Nós para representar os dados providos dos predicados *isSource* e *isSink* de cada análise.
  - *PropRead*: Representa a leitura do conteúdo de uma propriedade de um objeto.
  - *PropWrite*: Representa a atribuição de conteúdo de uma propriedade de um objeto.
  - *string*: Utilizado para representar o nome das propriedades dos objetos.
- Linha 2: Cláusula *where* para obter os fluxos dos dados para primeira e segunda análise.
- Linha 3: Continuação da cláusula *where* para recuperar a expressão representada pelos nós *sinks* da primeira e segunda análise em formato de expressão. Isto é realizado para obter o nome da propriedade dos objetos encontrados.
- Linha 4: Define que o conteúdo de *propName* é o nome da propriedade do objeto encontrado na primeira análise.
- Linha 5: Define que o nome da propriedade do objeto da linha 4 deve ser igual ao objeto *sink* da segunda análise.
- Linha 6: Define que o nó *sink* da terceira análise é representado pelo argumento da função da primeira análise.
- Linha 7: Cláusula *select* para obter os resultados encontrados.

5. Resultados

As consultas foram primeiramente testadas em códigos exemplos vulneráveis, apresentados nas seções de cada consulta. Posteriormente, as consultas foram executadas com o objetivo de encontrar as vulnerabilidades já relatadas e catalogadas através do banco de dados de consultoria do GitHub [GitHub 2021c].

O GitHub oferece uma API para obter todos os dados necessários para a execução, em que é possível obter todas as vulnerabilidades relacionadas as bibliotecas do NPM e seus respectivos códigos de CWE (*Common Weakness Enumeration*) associados. Portanto, através dos códigos de CWE é possível obter somente as bibliotecas que possuíram

alguma vulnerabilidade específica. Para a execução das consultas foi desenvolvido um algoritmo para obter as bibliotecas do NPM vulneráveis baseadas em suas versões e seus códigos de CWE associados, e um algoritmo para realizar as execuções de forma automatizada. O código completo de ambos os algoritmos pode ser visto na referência [dos Santos 2021].

```
async function main() {
  // Command injection
  let CWES = ['CWE-1321', 'CWE-915'];
  let vulnerabilities = JSON.stringify(await
     getVulnerabilities(CWES), null, 2);
  fs.writeFileSync('command-injection.json', vulnerabilities);

  // Code injection
  CWES = ['CWE-1321', 'CWE-915'];
  vulnerabilities = JSON.stringify(await
     getVulnerabilities(CWES), null, 2);
  fs.writeFileSync('code-iection.json', vulnerabilities);


  // Prototype pollution
  CWES = ['CWE-1321', 'CWE-915'];
  vulnerabilities = JSON.stringify(await
     getVulnerabilities(CWES), null, 2);
  fs.writeFileSync('prototype-pollution.json', vulnerabilities);
}
```

Código 24. Algoritmo para obter as bibliotecas do NPM vulneráveis

O código 24 representa, de forma resumida, o primeiro algoritmo em JavaScript para obter as bibliotecas vulneráveis, realizado em três passos. Na variável *CWES* são definidos os códigos CWE associados a cada tipo de vulnerabilidade. É obtido as bibliotecas vulneráveis através da chamada da função *getVulnerabilities*, retornando assim um objeto JSON contendo o nome das bibliotecas e suas respectivas versões vulneráveis. Por fim, é realizado a escrita a deste objeto JSON em um arquivo em disco, através da chamada da função *writeFileSync*.

O código 25 apresenta, de forma resumida, o algoritmo para analisar todas as vulnerabilidades das consultas desenvolvidas nas bibliotecas encontradas pelo código 24, guardadas no arquivo JSON em disco. Portanto, o código 25 representa apenas a execução para as análises da vulnerabilidade de injeção de comando, sendo assim executados os mesmos procedimentos para as vulnerabilidades de injeção de código e poluição de protótipo com os seus respectivos arquivos em disco que contém as bibliotecas vulneráveis. Primeiro é feito a leitura do arquivo JSON que está em disco, que contém o nome das bibliotecas vulneráveis e suas respectivas versões vulneráveis. Assim, é feito o *download* da biblioteca do NPM através da chamada da função *NPMPackageInstall*, informando nos argumentos o nome da biblioteca e a versão vulnerável. Nota-se que existe

```
async function main() {
  const vulnerabilities =
     JSON.parse(fs.readFileSync('./command-injection.json',
     'utf8'));
  let packagesQtt = 0;
  let vulnerable = 0;
  let version;

  for (vuln of vulnerabilities) {
    version = vuln.vulnerableVersionRange
    NPMPackageInstall(vuln.package,
       getVulnerableVersion(version,
       version.match(/\./g).length));
    createDB('javascript', vuln.package, `${vuln.package}-db`);

    if (runQuery(`${vuln.package}-db`,
       './queries/command-injection.ql')) {
      vulnerable++;
    }
    packagesQtt++;
  }
}
```

Código 25. Algoritmo para análise das consultadas desenvolvidas

uma função *getVulnerableVersion*, que representa o tratamento para encontrar uma versão vulnerável baseado na faixa de versões vulneráveis fornecidas pela API do GitHub. Assim, é feito a geração do bando de dados CodeQL a partir do código-fonte armazenado pelo *download*, através da chamada da função *createDB*. Por fim, é feito a análise das consultas desenvolvidas através da chamada da função *runQuery*, informando nos argumentos o banco de dados CodeQL criado anteriormente e o caminho que contém o arquivo da consulta desenvolvida. Contudo, a variável *vulnerable* armazena a quantidade de bibliotecas vulneráveis e *packagesQtt* armazena a quantidade de bibliotecas analisadas. Nota-se que no código 25 apresenta somente a análise para a vulnerabilidade de injeção de comando por uma única consulta, entretanto na prática são executadas todas as consultas desenvolvidas relacionadas a cada vulnerabilidade.

O gráfico da figura 1 apresenta os resultados expressos no formato de quantidades de bibliotecas analisadas e porcentagem de bibliotecas sinalizadas como vulneráveis pelas consultas desenvolvidas, de acordo com cada vulnerabilidade apresentada na legenda e com a sua respectiva cor. Contudo, dentre 84 bibliotecas analisadas relacionadas à vulnerabilidade de injeção de comando, 55% foram sinalizadas como vulneráveis pelas consultas desenvolvidas. Da mesma forma, foram analisadas 47 de bibliotecas relacionadas à vulnerabilidade de injeção de código e 19% foram sinalizadas como vulneráveis. Para as bibliotecas relacionadas à vulnerabilidade de poluição de protótipo, foram analisadas 63 bibliotecas e 81% foram sinalizadas como vulneráveis.

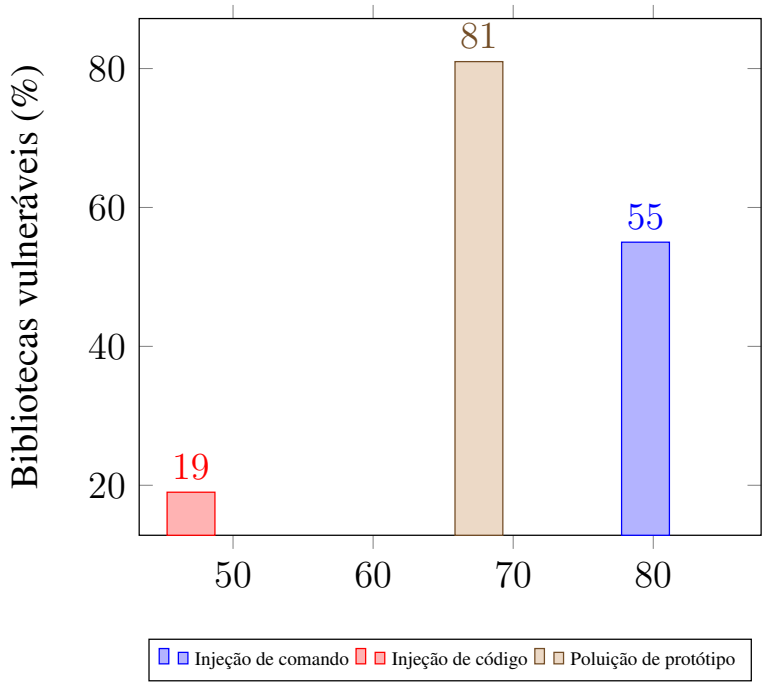

Figura 1. Resultado das análises

6. Conclusão

Neste trabalho foi realizado o estudo teórico e prático sobre o mecanismo CodeQL, apresentando os conceitos sobre a linguagem QL, juntamente com as bibliotecas associadas aos tipos de análises de código possíveis. Foi realizado um breve estudo sobre os dez conjuntos de vulnerabilidades em aplicações web mais recorrentes, baseando-se na publicação da organização OWASP. A partir disto, foi identificado que as vulnerabilidades de injeção são as mais recorrentes e mais críticas, assim contribuindo para a decisão sobre o desenvolvimento das consultas para o CodeQL. Contudo, foram desenvolvidas duas consultas para a vulnerabilidade de injeção de comando, duas consultas para a vulnerabilidade de injeção de código e uma para a vulnerabilidade de poluição de protótipo.

Os resultados apresentados mostram que as vulnerabilidades de injeção de comando, poluição de protótipo e injeção de código obtiveram eficácia de aproximadamente 80%, 55% e 20%, respectivamente. Com este resultado pode-se concluir que o conjunto de consultas que abrangeram a maior capacidade de variantes das vulnerabilidades baseadas em termos de *sources* e *sinks* apresentaram melhores resultados. Consequentemente, pelo fato das vulnerabilidades de injeção de código poderem ter muitas variantes já que a linguagem JavaScript proporciona uma grande flexibilidade em termos de codificação, a existência de funções sensíveis que levam à vulnerabilidade de injeção de código pode aumentar consideravelmente; o que pode ser o motivo da menor eficácia comparado às outras análises.

O desenvolvimento das consultas e o objetivo para este trabalho não abordou o desempenho de cada consulta, o que pode ser feito nos trabalhos futuros relacionados a este trabalho, medindo o tempo de execução de cada consulta e propor melhorias para os algoritmos desenvolvidos. Da mesma forma, não foi abordado a questão dos falsos positivos que as consultas podem sinalizar, pois é possível ter funções que atuam como

filtros sendo executadas antes de qualquer função sensível que pode levar à alguma vulnerabilidade, modelado através do predicado *isBarrier*.

Por meio deste trabalho pôde-se aprender o funcionamento do mecanismo CodeQL e sua flexibilidade em encontrar vulnerabilidades em aplicações de código abertos através das modelagens de consultas com a linguagem QL e as bibliotecas fornecidas pelo CodeQL. Apesar deste trabalho abordar as vulnerabilidades no contexto de aplicações web, é possível desenvolver consultas para outras linguagens suportadas pelo CodeQL. Da mesma forma, é possível desenvolver consultas para outros escopos de vulnerabilidades, como as vulnerabilidades relacionadas à exploração de binários.

Referências

dos Santos, R. R. S. (2021). Implementações. https://gist.github.com/EffectRenan/653d7e1001312059ad393d884dd45b43. Acesso em 2021-03-04.

GitHub (2021a). About codeql. https://codeql.github.com/docs/codeql-overview/about-codeql/. Acesso em 2021-03-04.

GitHub (2021b). Codeql library for javascript. https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-javascript/. Acesso em 2021-03-04.

GitHub (2021c). Github advisory database. https://github.com/advisories. Acesso em 2021-03-04.

GitHub (2021d). Modules. https://codeql.github.com/docs/ql-language-reference/modules/. Acesso em 2021-03-04.

OWASP (2021a). Owasp top 10 - 2017. https://owasp.org/www-pdf-archive/OWASP_Top_10-2017-pt_pt.pdf. Acesso em 2021-03-04.

OWASP (2021b). Source code analysis tools. https://owasp.org/www-community/Source_Code_Analysis_Tools. Acesso em 2021-03-04.

OWASP (2021c). Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis. Acesso em 2021-03-04.

Pavel, A., Moor, O., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J., and Verbaere, M. (2007a). Keynote address: .ql for source code analysis. SCAM.

Pavel, A., Moor, O., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J., and Verbaere, M. (2007b). .ql: Object-oriented queries made easy. GTTSE.

Pavel, A., Moor, O., Jones, M., and Schäfer, M. (2016). Ql: Object-oriented queries on relational data. ECOOP.