Universidade do Minho
Escola de Engenharia

Mestrado Engenharia Informática

PROJETO 2

Selenium, KritiKa e Deepscan

GRUPO 13

Bruno Rodrigues pg41066

Carlos Alves pg41840

CONTEÚDO

INTRODUÇÃO

Este trabalho prático sugerido na Unidade Curricular – Engenharia de segurança tem como objetivo realizar investigação sobre três ferramentas:

- **Selenium** – É utilizado para testar aplicações baseadas na web.
- **Kritika** – Ferramenta usada para analisar código verificando erros, semânticas, duplicatas entre outros aspetos de qualidade de código.
- **DeepScan** – Ferramenta usada para analisar código estático em JavaScript, utilizando por base uma versão semelhante ao **ESLinc**, mas melhorada.

A estrutura deste documento escrito, seguirá o seguinte fluxo:

1. Descrição detalhada da ferramenta
2. *Features* das ferramentas e o seu funcionamento
3. Exemplos de analises, verificando os erros encontrados.

A utilização deste tipo de ferramentas, tanto na análise estática durante o desenvolvimento do software ou da análise dinâmica durante os testes, vai permitir que as equipas de desenvolvimento possam melhor de forma significativamente os seus códigos, quer seja em qualidade ou vantagens que delas proveem.

Consequentemente haverá um maior controlo relativamente á qualidade do código desenvolvido em *outsouring*, custos de desenvolvimento serão reduzidos, a produtividade será maior e ainda através dos relatórios emitidos pelas ferramentas criar um apoio nos esforços de conformidade, isto é, as equipas de desenvolvimento poderão criar as suas próprias regras consoante os relatórios ou ainda dar a conhecer aos gestores de projeto o "nível" em que se encontra a sua equipa no quesito de qualidade de código.

1 SELENIUM

O **Selenium**, de acordo com os próprios desenvolvedores, automatiza browsers. Esta é a pequena descrição apresentada na primeira página do website. Mas para quê que é usado?

É uma ferramenta utilizada primeiramente para testes automatizados, que correm em browsers. É utilizado para testar aplicações baseadas na web.

A vantagem do **Selenium** passa pelo facto que a sua automação e rapidez bate a alternativa, que é o teste manual das aplicações. Esta é capaz de emular o comportamento normal de um utilizador de um browser, desde abrir uma página, carregar em botões e links, preencher formulários, entre outros.

O **Selenium** em si não é uma ferramenta apenas, mas sim um conjunto, composto pelo Selenium WebDriver, Selenium IDE e Selenium Grid., sendo que vamos agora falar um pouco do que cada ferramenta faz, ou consegue fazer. Apenas o Selenium Grid não será mencionado, visto que a sua funcionalidade é semelhante ao Selenium Webdriver, sendo que o que diferencia o Grid e que este permite a execução remota de testes em múltiplas máquinas, algo que não iremos conseguir demonstrar neste trabalho.

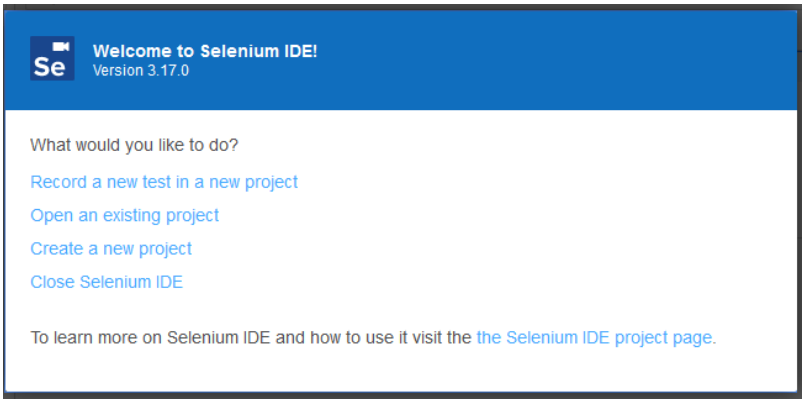
1.1 Selenium IDE

Começamos pelo Selenium IDE, que como o nome diz, é um IDE simples que permite a automação de testes através da gravação de input do utilizador, sem ser necessário conhecimento em programação.

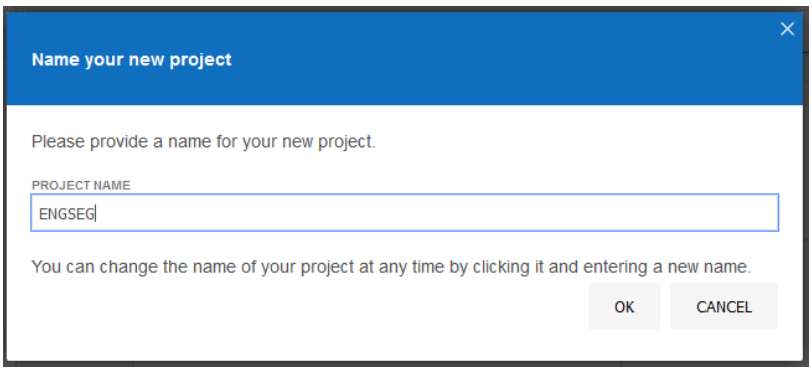
O IDE é um plugin que pode ser adicionado ao Firefox ou ao Chrome, sem ser necessário a instalação de extras para funcionar.

O fluxo de trabalho da aplicação é bastante simples de entender e os passos para realizá-los são os seguintes:

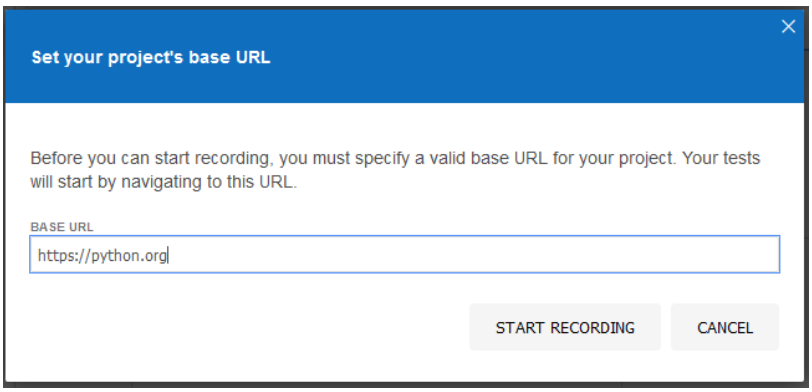
- Criar um projeto



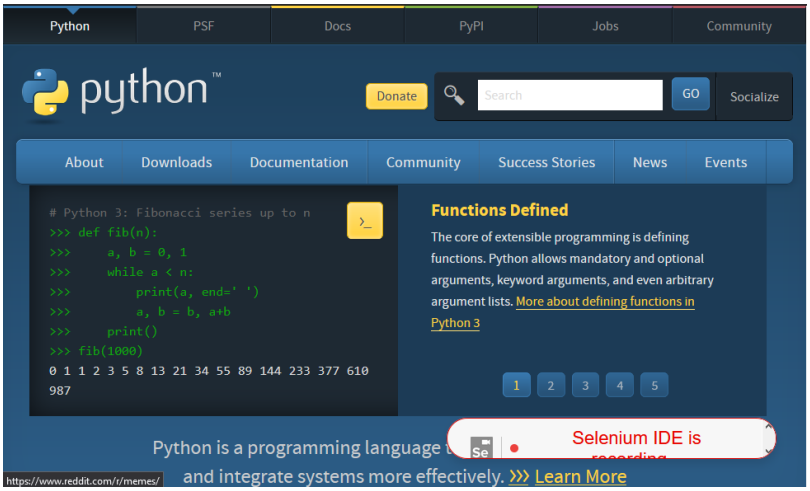
- Adicionar um novo teste ao projeto



- Uma página do browser será aberta no URL base fornecido.

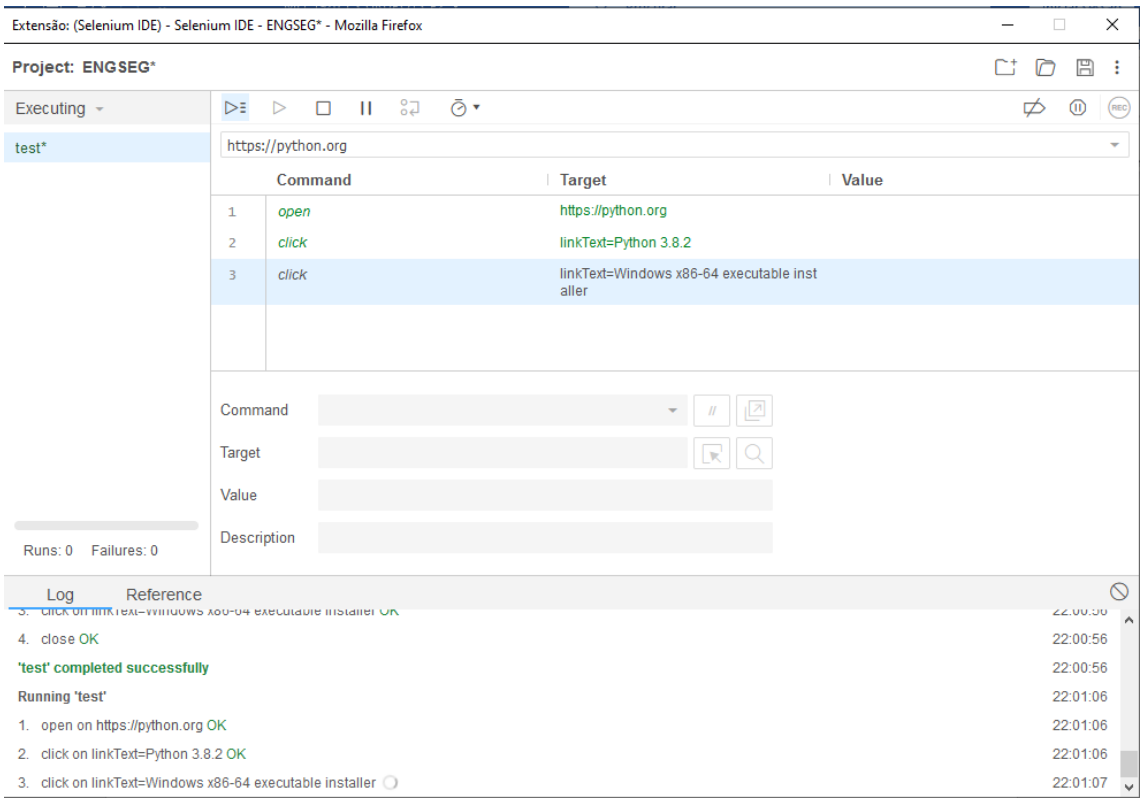


- De seguida o plugin começa a gravar o input introduzido pelo utilizador



- Quando o teste estiver completo, basta terminar a gravação.

Para testar, clicamos em alguns botões no website do python.org, algo que o Selenium IDE gravou, como podemos ver em baixo

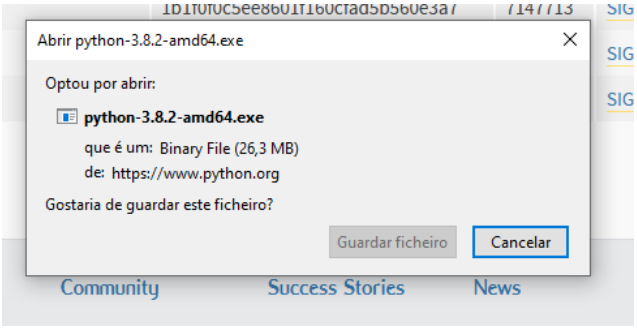


O teste que foi criado faz o seguinte:

1	*open*	https://python.org
2	*click*	linkText=Python 3.8.2
3	*click*	linkText=Windows x86-64 executable inst aller

1. Abre a página web alvo, que será o python.org
2. Carrega no elemento, um botão identificado como “Python 3.8.2”
3. Carrega no botão “Windows x86-64 executable installer”

Essencialmente, este é um pequeno teste que nos faz prompt para fazer o download do instalador mais recente do python para Windows.



O plugin também nos oferece um log onde nos relata os vários processos que realizou e onde passou/falhou. Neste caso todos foram realizados com sucesso.

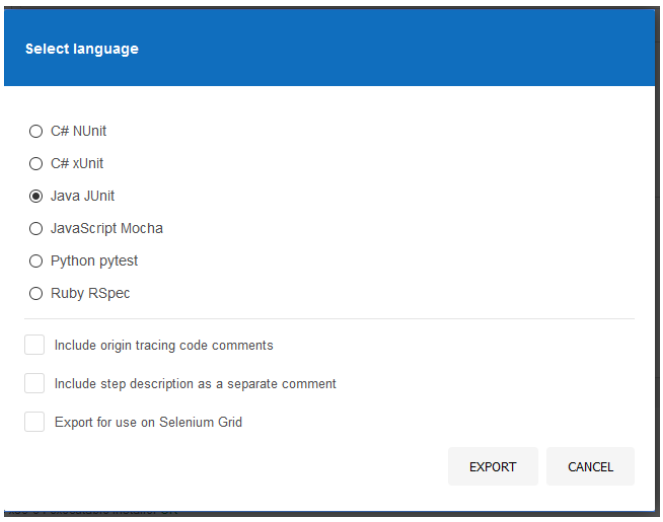
Também é possível adicionar mais comandos manualmente;

Por exemplo, adicionando o comando “close” faz com que a página seja fechada no fim

1	*open*	https://python.org
2	*click*	linkText=Python 3.8.2
3	*click*	linkText=Windows x86-64 executable in staller
4	*close*	

Command close

E podemos também exportar estes testes para uma linguagem de programação à escolha, permitindo depois ser utilizado pelo Selenium Webdriver.



Por exemplo, exportando em python podemos depois alterar o código e executá-lo, sendo que é necessário ter o Selenium Webdriver para o fazer.

```
class TestTest():
  def setup_method(self, method):
    self.driver = webdriver.Firefox()
    self.vars = {}

  def teardown_method(self, method):
    self.driver.quit()

  def test_test(self):
    self.driver.get("https://python.org")
    self.driver.find_element(By.LINK_TEXT, "Python 3.8.2").click()
    self.driver.find_element(By.LINK_TEXT, "Windows x86-64 executable installer").click()
    self.driver.close()
```

Ao executá-lo, temos exatamente o mesmo comportamento que acontece no IDE, a página web do python.org abre, e aparece-nos o prompt para fazer o download da versão mais recente do python para Windows.

Resumindo, uma das principais atrações do Selenium IDE passa por:

- Ser um plugin de browser de fácil instalação, sem ser necessário mais nada
- Permite gravar o input do utilizador e fazer playback
- Alterar e acrescentar comandos

- Não necessitar de conhecimento em programação
- Poder exportar o teste produzido para qualquer língua de programação suportada

1.2 Selenium Webdriver

O Selenium Webdriver é a ferramenta pela qual a Selenium é conhecida.

O Webdriver permite criar scripts através de programação. Estes scripts podem ser escritos em múltiplas linguagens de programação, sendo estas o Ruby, o Java, o Python, o C# e JavaScript, sendo que existem ainda outras, mas estas são as suportadas diretamente pelo projeto.

Para funcionar requer, para além da instalação da ferramenta, um web driver, que varia conforme o navegador usado (neste caso de uso foi usado o Firefox, e como tal foi necessário o geckodriver). Esta ferramenta funciona como ligação entre o Selenium e o browser, que permite ao Selenium enviar os comandos e fazer com que o browser os entenda.

Ao contrário do que vimos para o Selenium IDE, em que o input do utilizador é gravado e depois é possível reproduzi-lo, aqui somos capazes de produzir scripts mais elaborados, para além de ser capaz de realizar mais funções, sendo uma dessas razoes pela qual o Webdriver é mais poderoso do que o IDE.

Um exemplo disto pode ser visto em baixo:

Realizamos um pequeno script que cria uma conta google, introduzindo os dados pedidos e carregando continuar.

No Selenium IDE é assim que se parece:

Command	Target	Value
open	https://accounts.google.com/signup/v2/webcreateaccount?flowName=GlifWebSignIn&flowEntry=SignUp	
set window size	1003x697	
type	id=firstName	ENG
type	id=lastName	SEG
type	id=username	ENGSEGG13
type	name=Passwd	@ENGSEGg13.
type	name=ConfirmPasswd	@ENGSEGg13.
click	css=.RveJvd	
assert element present	id=phoneNumberId	
type	id=phoneNumberId	935423080

E escrito em python, utilizando o Webdriver:

```
def test_createMail(self):
    self.driver.get("https://accounts.google.com/signup/v2/webcreateaccount?flow
    self.driver.set_window_size(1003, 697)
    self.driver.find_element(By.ID, "firstName").send_keys("ENG")
    self.driver.find_element(By.ID, "lastName").send_keys("SEG")
    self.driver.find_element(By.ID, "username").send_keys("ENGSEGG13")
    self.driver.find_element(By.NAME, "Passwd").send_keys("@ENGSEGg13.")
    self.driver.find_element(By.NAME, "ConfirmPasswd").send_keys("@ENGSEGg13.")
    self.driver.find_element(By.CSS_SELECTOR, ".RveJvd").click()
    elements = self.driver.find_elements(By.ID, "phoneNumberId")
    assert len(elements) > 0
    self.driver.find_element(By.ID, "phoneNumberId").send_keys("999999999")
```

Ambos supostamente fazem exatamente o mesmo processo, ou deveriam fazer o mesmo processo, que seria:

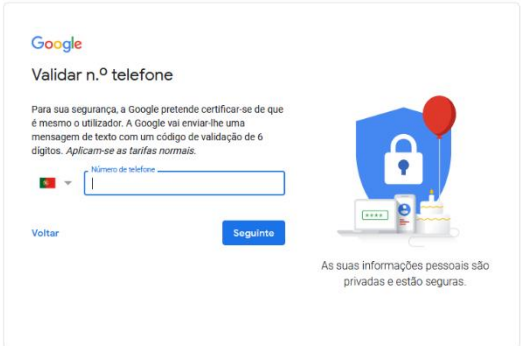
- Abrir a página de Registar.
- Introduzir o nome, a pass, carregar em continuar.
- Introduzir o número de telemóvel.

Ambas as plataformas (o IDE e o Webdriver) conseguem produzir o seguinte resultado:

Ambos preenchem os dados requeridos e carregam em “seguinte”.

Mas quando chega a parte em que é necessário introduzir o número de telemóvel, o IDE fica preso na parte em que é necessário introduzir o número, enquanto que o Webdriver realiza essa tarefa.

Com o Selenium IDE, o texto nunca chega a ser preenchido:

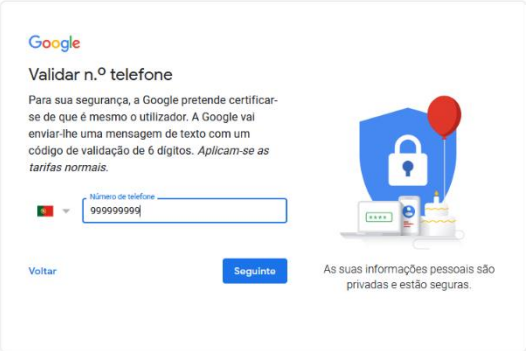


Enquanto que com o Webdriver já não acontece:

É de mencionar que estas contas não foram realmente criadas, visto que não existe a intenção de criar contas fantasmas.

O Selenium consegue fazer qualquer coisa que um usuário normal faz num browser, desde que seja composto um script funcional, para além de o fazer mais rápido.

Uma das dificuldades é apresentar exemplos concretos da utilização da ferramenta, visto que



isto varia de acordo com a necessidade de cada um, para além de a realização de testes automatizados em websites que não nos pertencem não nos é permitido, como também não é recomendado, pelo facto de poder provocar constrangimento aos donos do website. Como tal em baixo demonstramos um script que pega num ficheiro de texto pré-existente e envia cada linha numa mensagem diferente para um usuário específico do Facebook (neste caso será o Carlos Alves).

Os scripts são algo relativamente simples de compor e de entender:

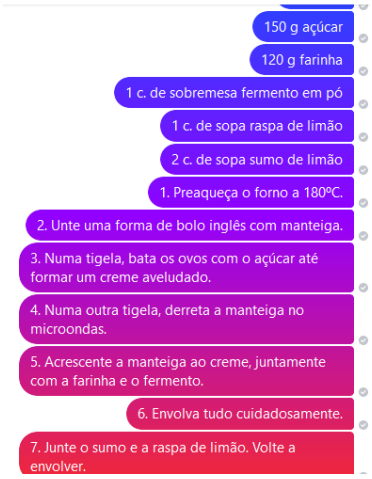
```
def setUp(self):
    self.driver = webdriver.Firefox()

def test_searchinFace(self):
    driver = self.driver
    driver.get("http://www.facebook.com")
    driver.find_element_by_name("email").send_keys("brunossj6@hotmail.com")
    driver.find_element_by_name("pass").send_keys("*******")-
    driver.find_element_by_id("loginbutton").click()
    driver.get("https://www.facebook.com/messages/t/cmdsa.pessoa")
    #textbox = driver.find_element_by_id("js_15")
    #textbox.send_keys("Messangem Automatizada")
    sendmsg = driver.find_element_by_class_name("_5rpu")
    with open("receita.txt","r",encoding="utf-8") as file:
        lines = file.readlines()
        for x in lines:
            sendmsg.send_keys(x + Keys.ENTER)


def tearDown(self):
    self.driver.close()
```

Decompondo o script, será algo assim:

- Primeiro especificamos o webdriver, que será o Firefox.

- De seguida fazemos um get do website do Facebook. Isto irá abrir a página de login do Facebook.
- Encontramos os elementos da página “email” e “pass” usando o find_element e usamos a função send_keys, que irá enviar os dados especificados. Isto irá permitir o login.
- De seguida abrimos a página de mensagens do usuário, sendo que depois temos de encontrar a caixa de texto. De seguida é só introduzir o texto, que neste caso será uma receita de bolo de limão.

O resultado:



2 KRITIKA

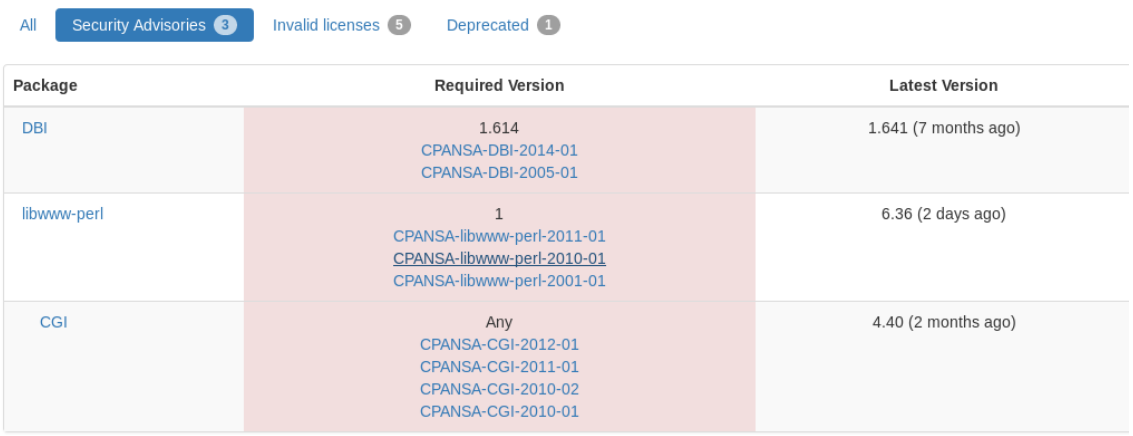
O **Kritika** permite analisar código e ainda fornecer certas informações sobre a forma como o código é escrito, a complexidade, se existem duplicações entre outros aspetos relacionados com o código redigido. Permite ainda, examinar licenças de dependências de código aberto. Se possível, o **Kritika** consegue detetar e apontar falhas de segurança nos códigos.

O **kritika** permite que o utilizador o integre ao GitHub, GitLab ou ao BitBucket. Em relação aos preços e planos que o **Kritika** fornece dependem diretamente do código analisado, ou melhor da quantidade de código analisado. A grande vantagem é que a análises de projetos open-source são totalmente gratuitas e ainda permitem o uso de todos os seus recursos, não existindo qualquer limitação. Os custos são cobrados apenas para repositórios privados e mesmo estes são limitados na quantidade de repositórios privados (5) e quantidade de linhas de código fonte para 200 mil SLOC.

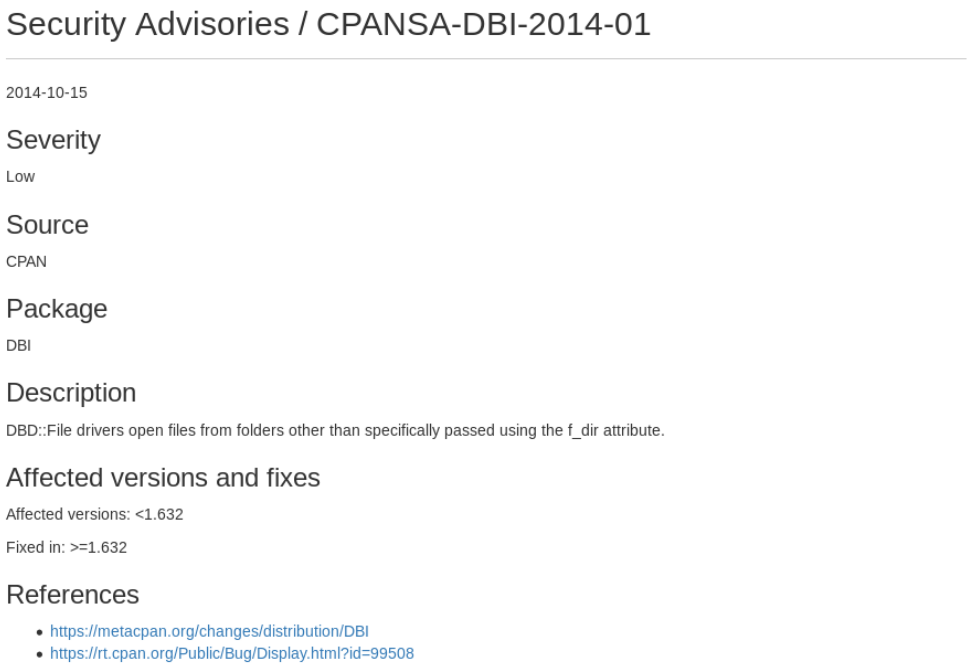
Ao analisar o próprio website da **Kritika**, é possível observar que a segurança é uma das grandes apostas desta ferramenta. Qualquer projeto analisado, não será guardado na base de dados da **Kritika**, apenas os respetivos *metadados*. Ainda preconizam para casos mais sensíveis, o uso de *git-crypt* – ferramenta que permite partilhar repositórios que contém uma mistura de conteúdo público e privado.

Em relação ás linguagens que este consegue analisar, encontram-se: Perl, PHP, Python, Java, JavaScript JSX, TypeScript, C, C++, Bash, Markdown e SQL.

Essencialmente, sempre que um repositório é analisado como também as suas dependências identificadas, estas últimas são por sua vez comparadas e verificadas a nível de vulnerabilidades existentes no banco de dados interno da própria **Kritika**. Os resultados dessas análises, de um modo geral, resultam num relatório semelhante ao da figura abaixo.

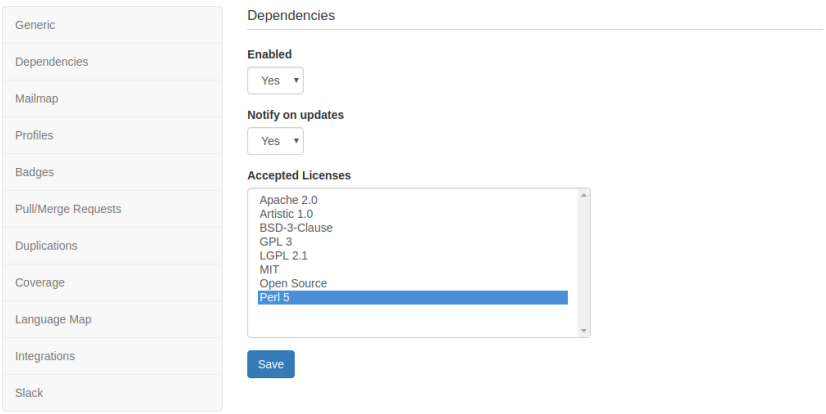


No relatório resultado da análise, é possível observar links de aviso de segurança, estes ao serem selecionados redirecionam o utilizador ou até mesmo o profissional de segurança a observar de forma mais detalhada sobre a vulnerabilidade ou erro encontrado. Nestes detalhes, permitem auxiliar o utilizador a tomar a melhor ação possível.



A base de dados referida anterior que serve como base de comparação é própria da **Kritika** e é pública. As fontes referidas na construção dessa base de dados são: base de dados CVE, distribuidores de pacotes específicos de determinadas linguagens e ainda *Feeds* consultivos de segurança de distribuição BSD e Linux.

Uma outra funcionalidade que o **Kritika** possui é a de controlar licenças e atualizações de dependências, isto pode ser ativado nas configurações do repositório e ao abrir a aba Dependências, ativando, conseguimos restringir as licenças que queremos e ainda ativar a deteção de dependências, recursivamente. Abaixo pode ser observado a tal funcionalidade.



Module	Distribution	Required Version	Latest Version	License	Deprecated
Apache::LogFormat::Compiler	Apache-LogFormat-Compiler	0.33	0.35 (2 years ago)	perl_5	No
Class::Inspector	Class-Inspector	1.12	1.32 (a year ago)	perl_5	No
Compress::Raw::Bzip2	Compress-Raw-Bzip2	2.081	2.081 (5 months ago)	perl_5	No
Compress::Raw::Zlib	Compress-Raw-Zlib	Any	2.081 (5 months ago)	perl_5	No
Cookie::Baker	Cookie-Baker	0.07	0.09 (7 months ago)	perl_5	No
Cwd	PathTools	Any	3.75 (22 days ago)	perl_5	No
Data::Dumper	Data-Dumper	Any	2.161 (2 years ago)		No
Devel::StackTrace	Devel-StackTrace	1.23	2.03 (10 months ago)	artistic_2_0	No
Devel::StackTrace::AsHTML	Devel-StackTrace-AsHTML	0.11	0.15 (2 years ago)	perl_5	No
Encode::Locale	Encode-Locale	1	1.05 (3 years ago)	perl_5	No
ExtUtils::MakeMaker	ExtUtils-MakeMaker	Any	7.34 (6 months ago)	perl_5	No

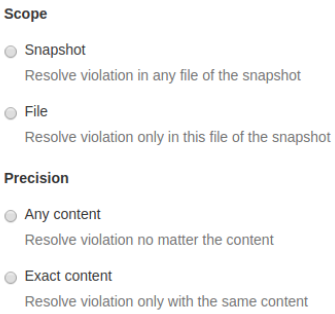
Dependências do projeto que vão ser analisadas

E para completar esta funcionalidade, é nos fornecido um painel informativo com a quantidade de dependências utilizadas, licenças inválidas e o número de quantas necessitam de ser atualizadas.

Todas estas informações podem ser reportadas diretamente para o nosso email, deste modo conseguimos ter acesso a estes dados rapidamente.

Na resolução de violações de código menos importantes, é possível “resolver” a violação sem modificar o código. É nos indicado ao lado da violação reportada, um link especial que nos reencaminha para um formulário, onde é necessário especificar certos parâmetros:

- Scope, Precision, Comentário entre outros aspetos.



1. Na resolução do Scope ou escopo, pode ser feita a partir de dentro do projeto ou do arquivo que nos encontramos. Aqui é aconselhado ser usado apenas para resoluções

temporárias ou mesmo violações que achamos necessárias a ser ignoradas, claro se for o último caso, é preferível desativar a violação do perfil.

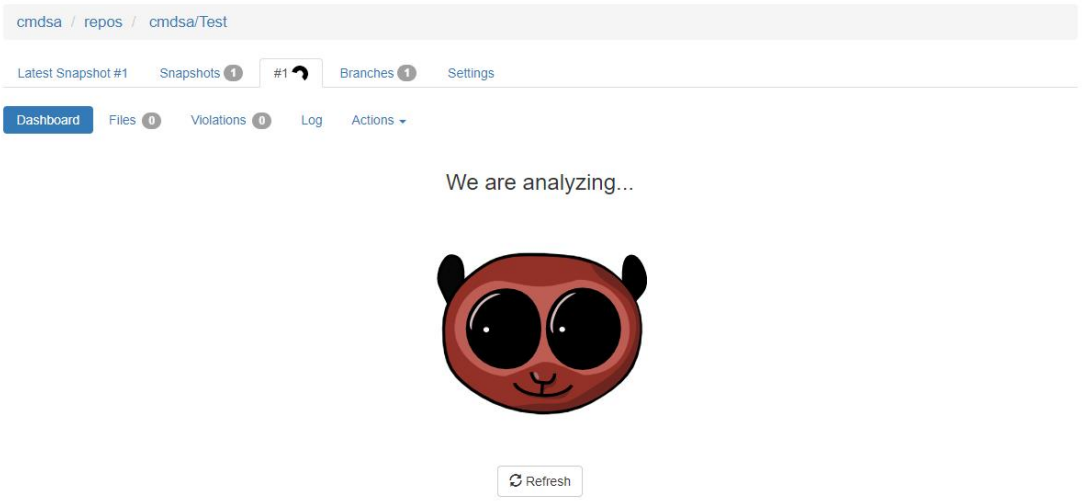
2. Na resolução de Precisão, pode ser orientada ao conteúdo, isto é, resolverá as violações causadas pelo mesmo código.
3. Temos ainda a opção de tempo de expiração da resolução, usada naquelas violações que desejamos ignorar apenas temporariamente.

Outra funcionalidade do **Kritika** é a deteção de código duplicado, isto apenas está disponível para poucas linguagens de programação, observado apenas com Perl. Esta funcionalidade vem "combater" a má prática (considerada por alguns) do *Copy e Paste*. O exemplo que melhor explica o objetivo de tal funcionalidade, é o seguinte: Nós temos um projeto onde é necessário implementar um botão, mas como somos preguiçosos ou temos pouco tempo restante, acabamos por copiar o código de um outro botão feito á uns tempos, mas tal código possui um bug. Imagine-se que nós somos demitidos e outro nosso colega (recém-chegado) pega no código e consegue resolver, mas como ele não sabe que existem outros botões escritos da mesma forma, o bug não desaparecerá até que sejam corrigidas em todos os botões que possuem o mesmo código. Isto é apenas um exemplo, mas este tipo de problema pode causar ainda problemas em arquiteturas e abstrações.

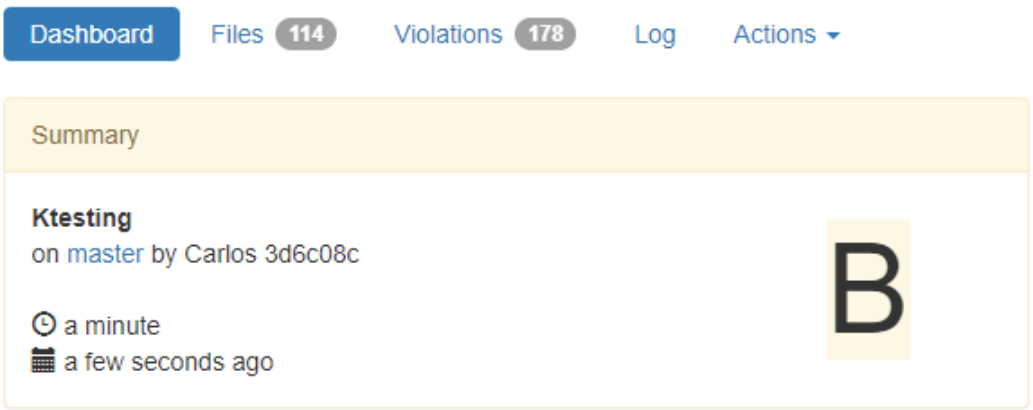
A **Kritika** como outras ferramentas do género, ao implementarem esta funcionalidade decidiram combinar todos os métodos (baseado em *String*, *Semântica*, *Árvore* e *token*). Essencialmente começa por gerar uma árvore do código, onde é ignorado os espaços em branco, os comentários e a documentação. Depois extrai as subárvores, neste estágio varia muito, pois depende muito do que pretendemos a analisar, no fim deste estágio obtemos as subárvores normalizadas e guardadas para serem analisadas posteriormente. O próximo estágio baseia-se na substituição *tokens* literais, chamadas de função ou métodos e ainda classes pelo seu respetivo *hash*, por exemplo temos *identifier*, após este processo passa a *id*. Com isto feito, são então comparadas as subárvores uma a uma e com uma percentagem de desvio é possível evitar sequências irrelevantes. Esta funcionalidade termina ao fazer a comparação LCSS das duplicações finais, onde é executado de forma recursiva LCSS, calculando assim uma percentagem dos dados mais comuns. Ao testar esta funcionalidade, observamos que esta ainda está numa fase inicial, deste modo é necessário ativa-lo nas configurações do repositório.

2.1 EXEMPLOS DA SUA UTILIZAÇÃO

Inicialmente temos de escolher o repositório que desejamos que seja analisado, depois de selecionado.



Para exemplo de experimentação foi utilizado o repositório (https://github.com/fportantier/vulpy), neste repositório encontra-se uma aplicação *python* vulnerável com o objetivo de demonstrar e ensinar segurança desenvolvimento. Nela temos duas pastas, uma '**GOOD'** e outra '**BAD'**: Na 'BAD' temos uma serie de código com inúmeros erros e vulnerabilidades, e na '**GOOD'** temos a sua resolução. Por isso o que nos interessa analisar é o que está dentro da pasta '**BAD'**, ao analisar esta obtivemos o seguinte relatório:



Na imagem acima podemos verificar que foram verificados 114 ficheiros, e nesses foram encontradas 178 violações. Ainda é possível observar a forma como o **Kritika** abordou a análise, na aba Log informando a construção das tais subárvores, como foi explicado anteriormente. E por fim, o **Kritika** ainda nos permite voltar a analisar o repositório ou apagar o relatório. Esta

análise teve como score B, este score está compreendido entre a pontuação máxima A e a pontuação mínima F.

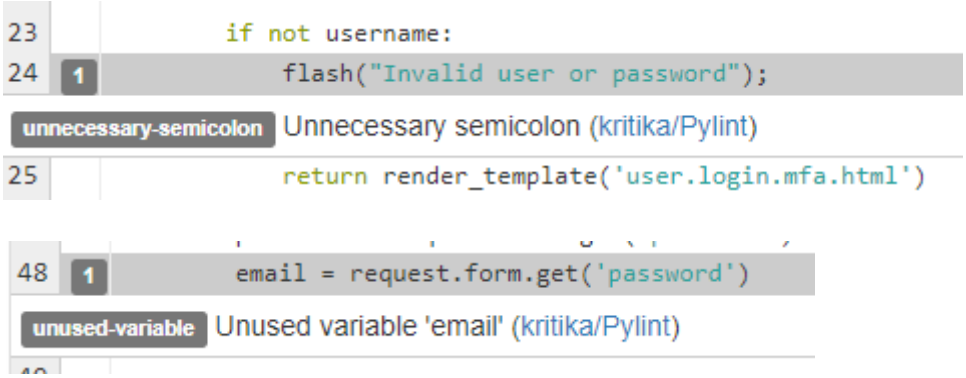
Path	Language	Churn	Violations	Complexity	SLOC	Score
leaked_passwords.txt		1	0	—	44.1K	A
libapi.py	Python	1	10	—	37	F
libmfa.py	Python	1	9	—	64	F
libposts.py	Python	1	3	—	20	B
libsession.py	Python	1	4	—	39	C

No *ScreenShot* acima, observamos de forma mais precisa o ficheiro analisado, em que linguagem se encontra redigido, número de violações, número de linhas de código fonte analisadas e por fim um *Score*. Desta forma, conseguimos concluir onde o nosso software se encontra mais "sujo ou com mais erros" podendo assim corrigi-lo mais rapidamente, pois também é nos informado o suposto "erro".

Vejamos o script mod_user.py, que foi avaliado com a pior pontuação F, contendo 5 violações. Ao abrir o script podemos ver algumas marcações no lado esquerdo das linhas de código, essas marcações indicam o suposto erro/vulnerabilidade no nosso código, com isto podemos corrigi-lo no mesmo instante ao carregar no botão resolver.

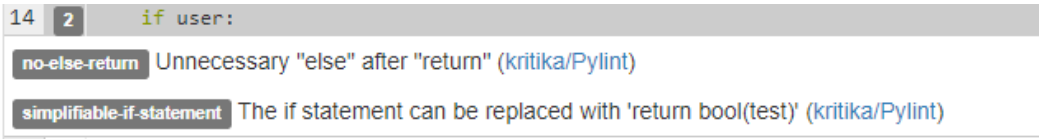
Neste caso o erro aparece, pois é feito o *import* de um modulo, o **sqlite3**, que nunca é usado. O erro em si não é muito grave, mas cria uma dependência que não precisa existir e pode dificultar a leitura do código. A solução aqui seria apagar esta dependência, como já referido o **Kritika** pode fazer isso por nós se assim desejar-mos.

O próximo erro encontra-se no método responsável pelo login, numa das estruturas *if*. O erro consiste essencialmente no uso de um ponto e virgula no fim da linha, visto que em Python não é de todo necessário o uso deste.



Acima podemos verificar que o outro erro detetado é o de uma variável não utilizada, deste modo também é de fácil resolução, basta apagar.

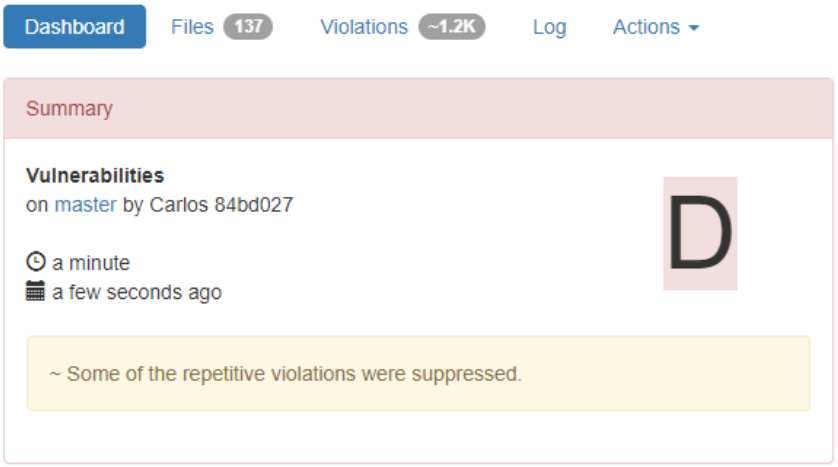
Para terminar este exemplo mais simples, concluímos que o **Kritika** através da sua análise consegue detetar erros de sintaxe, duplicações, dependências desnecessárias entre outros aspetos que nos ajudaram a ter um código mais 'belo' e limpo possível, um outro exemplo é o que se segue abaixo.



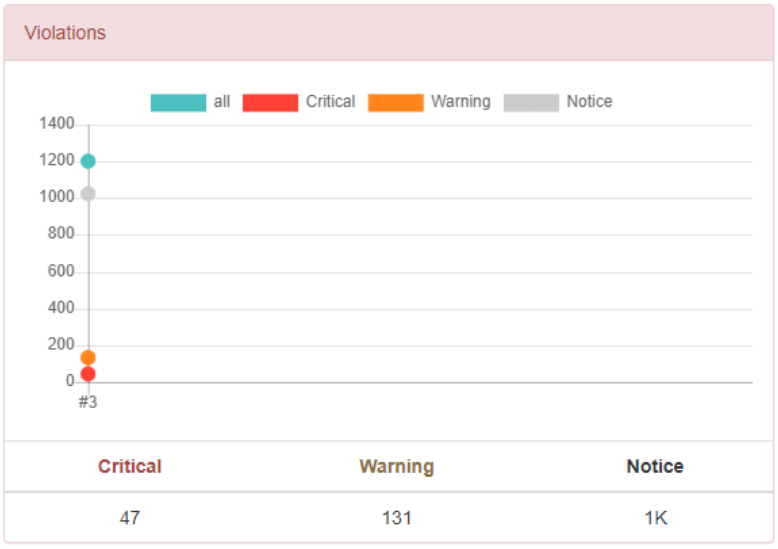
No relatório é possível ver uma lista das vulnerabilidades encontradas e dos responsáveis (o script), isto através de um top 10.

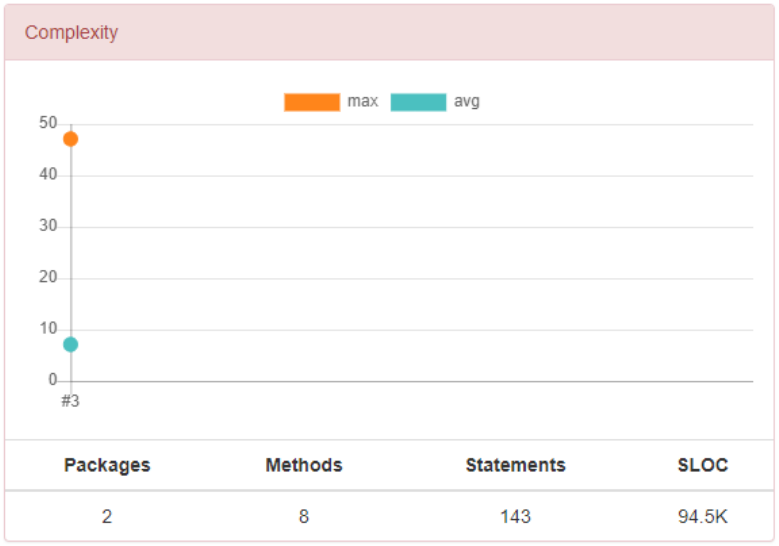
Ao analisar a imagem a cima, concluímos que neste nosso exemplo grande parte das violações são derivadas a *imports* de módulos que não são utilizados, parâmetros sem valores, imports feitos de forma errada, e outros relativos a simplificar o código.

De seguida, será documentado a análise feita pelo **Kritika.io** aos scripts fornecidos pelo docente para realização das fichas de trabalho semanais.

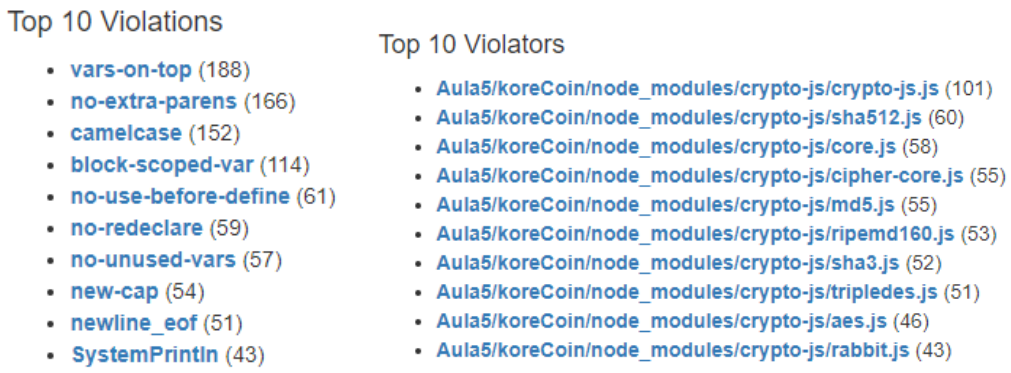


Sucintamente, esta análise a 137 ficheiros classificou o repositório com um Score D, o que é péssimo. Ainda nesta análise foram encontradas mil e duzentas violações ou mais (algumas foram suprimidas), desta vez foram colocadas num gráfico de violações e outro gráfico de complexidade. Como é possível verificar abaixo, existem 47 violações classificadas como **Critical**, 131 **Warning** e 1000 **Notice**.





Neste exemplo só iremos explicar e demonstrar as violaçoes classificadas como **CRITICAL**. A primeira violação e que aparece enumeras vezes é o uso de ***System.out.println***, isto pois é uma péssima prática para logging, visto que não possui opçao de ligar ou desligar, não tem a capacidade de diferentes niveis de saída (**TRACE, DEBUG, INFO, WARN, ERROR**), sem ter que recompilar o código e o caso mais grave é que a saída padrão de um programa pode ser redirecionada. A outra violação classificada como **CRITICAL** é ***AvoidUsingShortType***.



Vemos também que o Kritika detetou violações à **lei de Demeter** – principio do menor conhecimento, **vars-on-top** – gerado quando as declarações de variáveis não são usadas serialmente na parte superior da função ou do próprio programa (javascript), Restrição no uso de parenteses apenas onde são realmente necessários; Entre outras violações.

Concluído a nossa análise a esta ferramente, verificamos ao longo de várias análises aos diferentes repositorios, o **Kritika.io** não foi capaz de encontrar os enumeros erros e vulnerabilidades que nós conseguimos explorar na realização dos trabalhos práticos semanais, mas ainda assim é louvavel a quantidade de avisos e de violaçoes que este nos indicou. De um modo geral é uma excelente ferramenta para manter o nosso código mais limpo.

3 DEEPSCAN

Por último o **DeepScan**, este possui algumas particularidades semelhantes com a ferramenta **Kritika.io** entre elas o fato de conseguir estar sempre atualizado, pois também é sincronizado com o repositório no GitHub, à medida que são feitas alterações no repositório o **DeepScan** vai inspecionando e isto manterá o status atualizado. Também funciona bem no gerenciamento do status de qualidade de uma equipa de desenvolvedores e como já referido sempre sincronizado.

DeepScan é orientado a programas JavaScripts, conseguindo realizar análise semânticas, isto é seguindo a execução e o fluxo de dados do programa, o **DeepScan** consegue detetar mais problemas do que as ferramentas linter. Como por exemplo:

- Uso de verificações nulas inconsistentes;
- Uso de conversão implícita de tipo;
- Atribuição com os mesmo valores;
- E ainda Código inacessível.

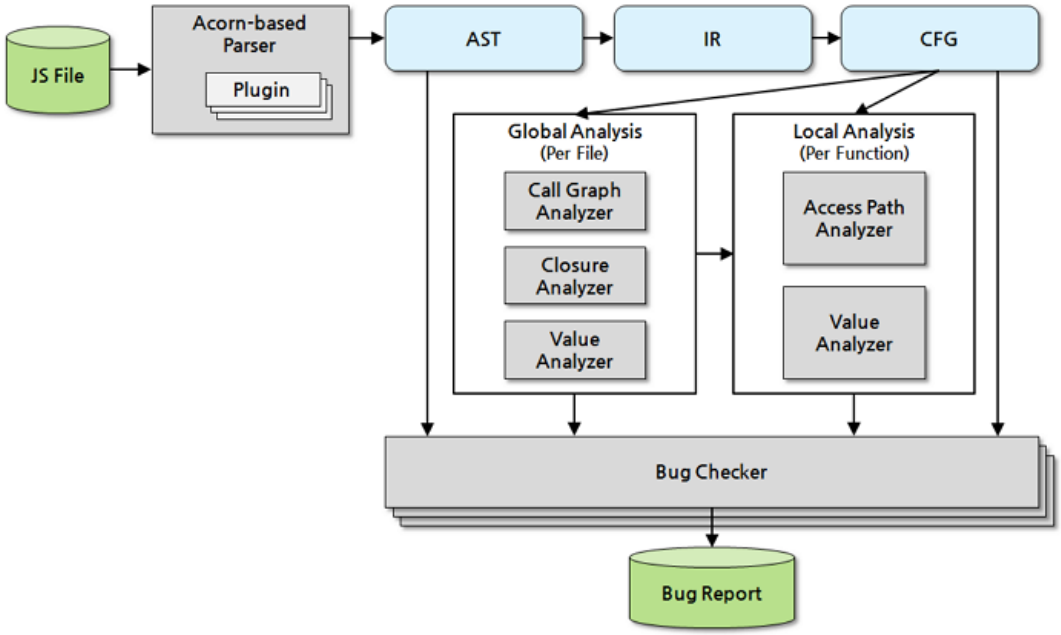
Após algum tempo de utilização, verificamos que o **DeepScan** é deveras rápido nas suas análises, chegando a analisar 7000 linhas por segundo.

Como no **Kritika.io**, o **DeepScan** também possui um conjunto de regras de ponta para a qualidade do código, mas apenas para JavaScript. Possui por exemplo, regras para verificar erros de tempo de execução e problemas de qualidade de código. Utilizaram diversas fontes credíveis no desenvolvimento de tais regras, tais como: CWE, FindBugs, PMD e outros. Sempre que o **DeepScan** deteta um erro este atribui um Score/Impacto (“Alto, “Medio” e “Baixo”).

Além das regras já mencionadas estes também se preocupam em oferecer suporte a padrões e frameworks Web. E claro as regras especializadas do React e do Vue.js, permitindo assim uma ajuda aos desenvolvedores que utilizem tais frameworks.

Segundo os próprios, a deteção é bastante precisa, tendo uma taxa de falsos positivos abaixo de 5%. Esta taxa reduzida deve-se essencialmente à análise semântica e à filtragem efetuada dos problemas detetados.

Talvez o fator que o diferencia de outras ferramentas parecidas seja na análise Semântica, visto que a análise estática não existe por causa dos recursos dinâmicos do JavaScript. Então a **DeepScan** criou uma versão melhorada do **ESLinc**, acrescentando o que lhe fazia falta: seguir o fluxo do programa, deste modo consegue detetar ainda mais problemas. Para isto o **DeepScan** utiliza Grafos de fluxo de controlo(Control Flow Graph).



3.1 EXEMPLOS

No caso de saber qual o valor de uma determinada variável

```
var destroyObject = function (object) {
    var type = Object.prototype.toString.call(object[i]); // Use of uninitialized variable
    for (var i in object) {
        if (type === '[object Object]' || type === '[object Array]') {
            destroyObject(object[i]);
        }
        object[i] = null;
        delete object[i];
    }
}
```

Vejamos na imagem acima um código responsável pela destruição de um objeto. Mas

analisando o ciclo for, determinamos que não funciona recursivamente pois a variável *type* é "indefinido" pela variável não inicializada i. Ao realizar uma análise a este código verificamos que o **DeepScan** consegue rastrear possíveis valores de várias variáveis, isto em determinados pontos do programa, tal feito não é visto no **ESLint**. O **DeepScan** consegue detetar *object[i]* como um erro, pois ele verificou o estado e o valor da variável i no scopo.

Outro exemplo seria no Controlo de Fluxo, se for colocado um *return* de algo dentro do ciclo for, o ciclo será imediatamente interrompido após a primeira iteração e o **DeepScan** emite um erro, pois é capaz de "conhecer" o fluxo de controlo, neste caso o for do programa.

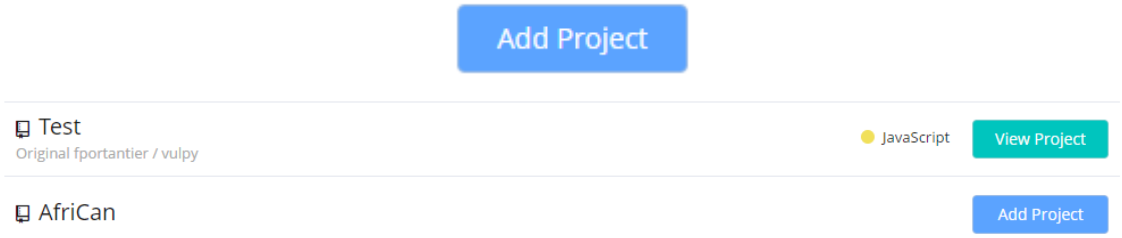
```
_checkCacheExpire : function(sKey) {
    if (sKey) {
        for (var i = 0, n = this._waKeyList.length(); i < n; i++) {
            return this._checkCacheExpire(this._waKeyList.get(i));
        }
    }
}
```

Para realizar um teste mais aprofundado ao DeepScan.io, utilizamos um repositório de um projeto desenvolvido em NodeJS[1], é um projeto com vulnerabilidades identificadas e foi desenvolvido com o propósito de testar a qualidade de ferramentas como a DeepScan, realizar testes de penetração e ainda ensinar de alguma maneira o que não fazer em JavaScript.

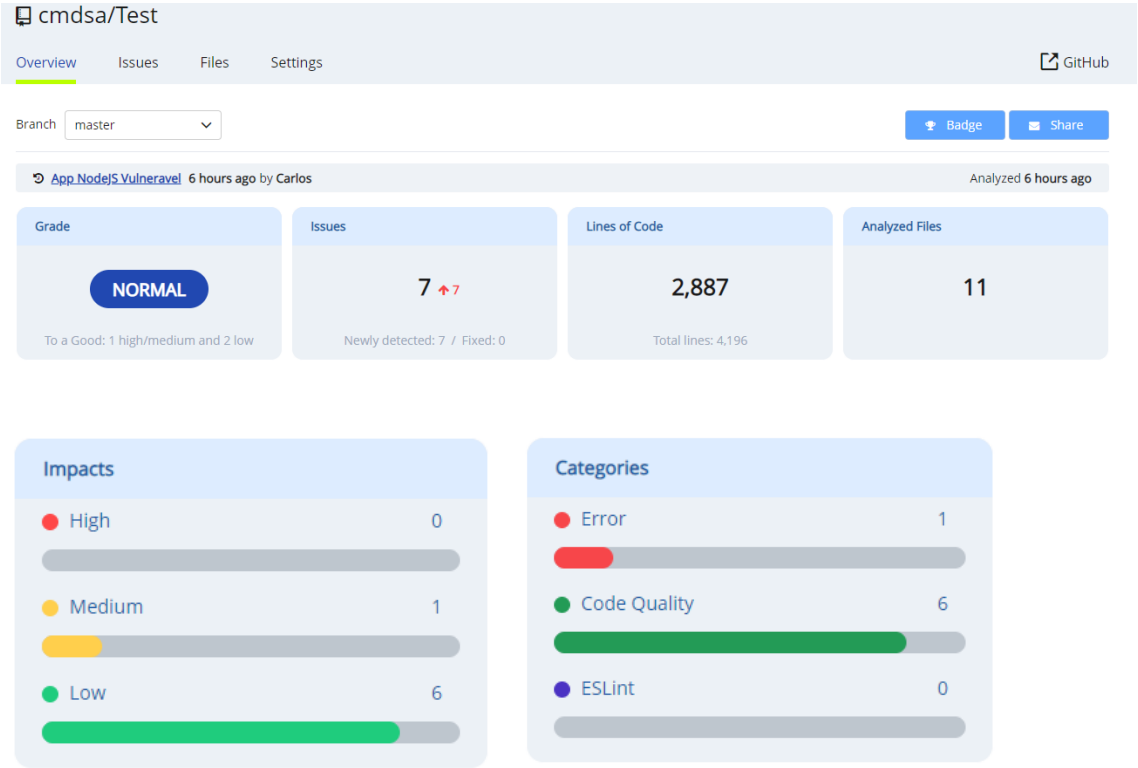
O projeto em questão, possui as vulnerabilidades mais comuns do top 10 OWASP:

- Injection;
- Broken Authentication and Session Management;
- Cross-Site Scripting (XSS);
- Insecure Direct Object References;
- Security Misconfiguration;
- Sensitive Data Exposure;
- Cross-Site Request Forgery (CSRF);
- Unvalidated Redirects and Forwards.

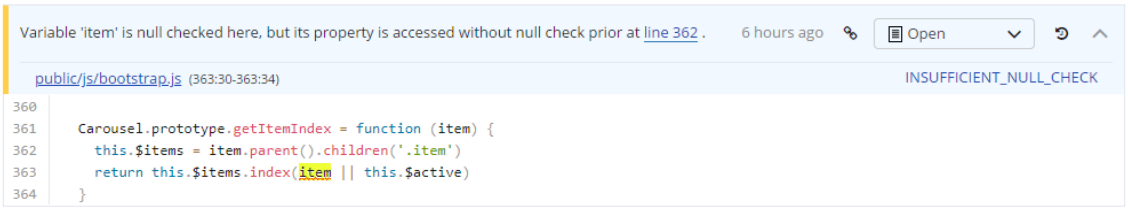
Primeiramente para realizar uma análise no **DeepScan** devemos colocar o projeto no github, de seguida no Dashboard da nossa conta, adicionamos o projeto para análise.



A análise em si é relativamente rápida, dependendo diretamente da quantidade de linhas de código a serem analisadas e das funcionalidades utilizadas. Após concluída análise, obtemos um relatório semelhante ao do **Kritika.io**. Infelizmente não foi possível realizar a análise com auxílio do Eslint devido a diversos erros de configuração, mas o resultado seria praticamente o mesmo que relatório do **Kritika.io**.



Como é visível no primeiro *ScreenShot*, o **DeepScan** avaliou o nosso projeto como NORMAL, contendo 7 problemas, 2887 linhas de código e 11 ficheiros. Dos sete problemas encontrados, apenas 1 foi classificado como Error e os restantes foram vistos como forma de melhorar a qualidade do código. O fato do projeto analisado conter inúmeros erros e vulnerabilidades o DeepScan apenas encontrou erros relativos a módulos instanciados, mas não usados, a variáveis que também não são utilizadas e duplicação de código. O único problema de concreto encontrado foi relativo a variáveis que não são verificadas se são NULL ou não, como mostra o *ScreenShot* abaixo.



Para mais testes e exemplos, incluídos exemplos CWE, React e Vue.js – Anexo A.

Com os testes realizados com esta ferramenta facilmente concluímos que esta encontra erros de todo o género e ainda aconselha o utilizador a tomar medidas no princípio da qualidade de código, facilitando assim o trabalho do desenvolvedor.

CONCLUSÃO

Neste trabalho tínhamos como objetivo analisar e explorar as ferramentas **Selenium**, **Kritika** e **Deepscan**, sendo que com estas deveriam também ser apresentados alguns exemplos da sua utilização.

Como tal, ao longo do relatório fomos apresentando o que cada ferramenta consegue fazer, tendo sempre o cuidado de acompanhar essa análise com exemplos demostrativos, para que seja mais fácil compreender o poder da ferramenta de que se está a falar.

Falamos do **Selenium**, bastante conhecida, uma ferramenta *open-source* capaz de automatizar browsers de forma a melhorar o processo de testes de *websites*, como também reduzir o trabalho manual. O **Kritika**, uma ferramenta útil para descobrir *bad smells*, violações de boas práticas e erros, e o **Deepscan**, uma ferramenta de análise ao JavaScript, que se identifica como uma alternativa superior aos outros **Linters**.

Acreditamos que a análise realizada foi satisfatória, e resultou numa aprendizagem para os membros do grupo. Com a realização do trabalho acabamos por descobrir novas ferramentas úteis que reforçam boas práticas em programação como também ajudam a mitigar riscos e possíveis falhas de segurança.

4 BIBLIOGRAFIA

(s.d.). Obtido de DeepScan: https://deepscan.io/

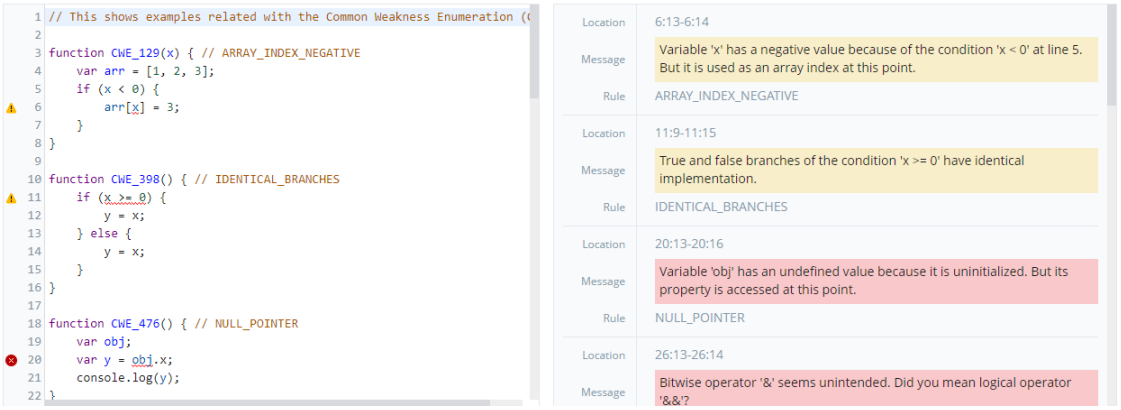
(s.d.). Obtido de Kritika: kritika.io

(s.d.). Obtido de Selenium: https://www.selenium.dev/

[1]cr0hn. (27 de Junho de 2019). Obtido de https://github.com/cr0hn/vulnerable-node

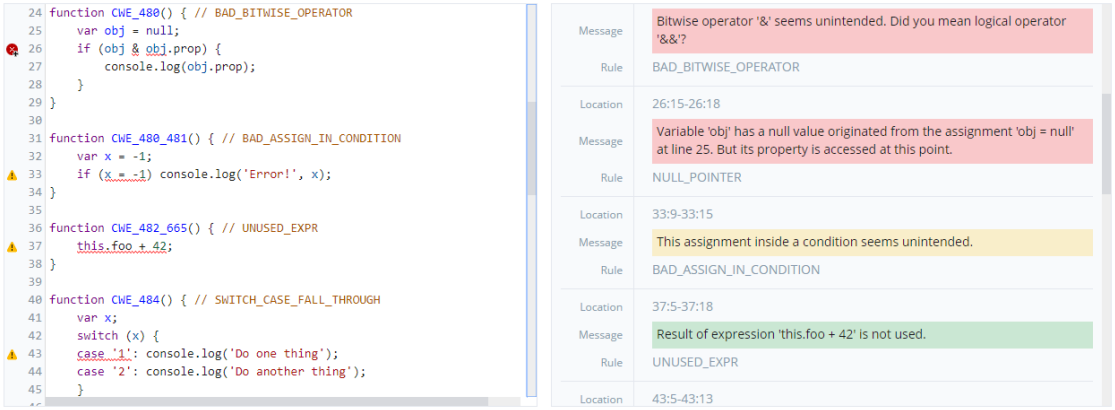
Anexo A

Vejamos alguns exemplos simples de Common Weakness Enumeration (CWE) sendo detetados pelo DeepScan.io.



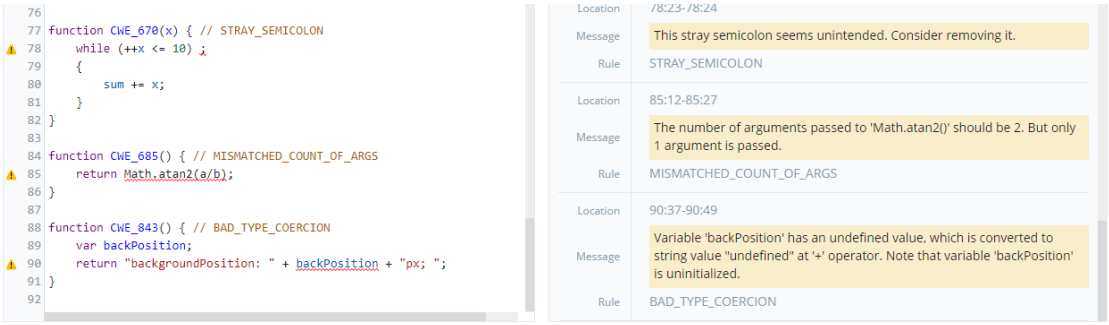
```
1  // This shows examples related with the Common Weakness Enumeration (
2
3  function CWE_129(x) { // ARRAY_INDEX_NEGATIVE
4      var arr = [1, 2, 3];
5      if (x < 0) {
6          arr[x] = 3;
7      }
8  }
9
10 function CWE_398() { // IDENTICAL_BRANCHES
11     if (x >= 0) {
12         y = x;
13     } else {
14         y = x;
15     }
16 }
17
18 function CWE_476() { // NULL_POINTER
19     var obj;
20     var y = obj.x;
21     console.log(y);
22 }
```

Location	6:13-6:14
Message	Variable 'x' has a negative value because of the condition 'x < 0' at line 5. But it is used as an array index at this point.
Rule	ARRAY_INDEX_NEGATIVE
Location	11:9-11:15
Message	True and false branches of the condition 'x >= 0' have identical implementation.
Rule	IDENTICAL_BRANCHES
Location	20:13-20:16
Message	Variable 'obj' has an undefined value because it is uninitialized. But its property is accessed at this point.
Rule	NULL_POINTER
Location	26:13-26:14
Message	Bitwise operator '&' seems unintended. Did you mean logical operator '&&'?



```
24 function CWE_480() { // BAD_BITWISE_OPERATOR
25     var obj = null;
26     if (obj & obj.prop) {
27         console.log(obj.prop);
28     }
29 }
30
31 function CWE_480_481() { // BAD_ASSIGN_IN_CONDITION
32     var x = -1;
33     if (x = -1) console.log('Error!', x);
34 }
35
36 function CWE_482_665() { // UNUSED_EXPR
37     this.foo + 42;
38 }
39
40 function CWE_484() { // SWITCH_CASE_FALL_THROUGH
41     var x;
42     switch (x) {
43     case '1': console.log('Do one thing');
44     case '2': console.log('Do another thing');
45     }
```

Message	Bitwise operator '&' seems unintended. Did you mean logical operator '&&'?
Rule	BAD_BITWISE_OPERATOR
Location	26:15-26:18
Message	Variable 'obj' has a null value originated from the assignment 'obj = null' at line 25. But its property is accessed at this point.
Rule	NULL_POINTER
Location	33:9-33:15
Message	This assignment inside a condition seems unintended.
Rule	BAD_ASSIGN_IN_CONDITION
Location	37:5-37:18
Message	Result of expression 'this.foo + 42' is not used.
Rule	UNUSED_EXPR
Location	43:5-43:13



```
76
77 function CWE_670(x) { // STRAY_SEMICOLON
78     while (++x <= 10) ;
79     {
80         sum += x;
81     }
82 }
83
84 function CWE_685() { // MISMATCHED_COUNT_OF_ARGS
85     return Math.atan2(a/b);
86 }
87
88 function CWE_843() { // BAD_TYPE_COERCION
89     var backPosition;
90     return "backgroundPosition: " + backPosition + "px; ";
91 }
92
```

Location	78:23-78:24
Message	This stray semicolon seems unintended. Consider removing it.
Rule	STRAY_SEMICOLON
Location	85:12-85:27
Message	The number of arguments passed to 'Math.atan2()' should be 2. But only 1 argument is passed.
Rule	MISMATCHED_COUNT_OF_ARGS
Location	90:37-90:49
Message	Variable 'backPosition' has an undefined value, which is converted to string value "undefined" at '+' operator. Note that variable 'backPosition' is uninitialized.
Rule	BAD_TYPE_COERCION

Anexo B

Vejamos alguns problemas React sendo detetados pelo DeepScan.io.

Location	14:5-14:22
Message	'componentDidUpate' could be a typo. Did you mean 'componentDidUpdate'?
Rule	REACT_API_TYPO
Location	26:13-26:20
Message	The 'render()' function of React component 'BadReactApiReturnValue' returns an undefined value at this point. Consider returning null instead.
Rule	REACT_BAD_API_RETURN_VALUE
Location	42:35-42:56
Message	No value is returned from the function 'recordFieldFocus' defined at line 36.
Rule	MISSING_RETURN_VALUE
Location	51:35-51:46
Message	'frameborder' is not a valid prop for React DOM element. Did you mean 'frameBorder' instead?

Location	116:26-116:42
Message	Event handler of a React DOM element cannot be specified with a string value. Consider specifying a function instead.
Rule	REACT_BAD_EVENT_HANDLER
Location	131:67-131:89
Message	Function 'this.trackUpgradeClick' is used as a React event handler without 'this' binding. But 'this' object is accessed in the function body at line 125.
Rule	REACT_EVENT_HANDLER_INVALID_THIS
Location	140:25-140:26
Message	The 'style' prop of a React DOM element cannot be specified with a numeric value. Consider using an object instead.
Rule	REACT_BAD_STYLE_PROP

Anexo C

Vejamos alguns problemas de Vue.js sendo detetados pelo DeepScan.io.

Location	9:5-9:62
Message	An event listener is added to the 'window' object at 'mounted()'. But it is never removed afterwards. Consider calling 'window.removeEventListener()' at 'beforeDestroy()'.
Rule	VUE_MISSING_CLEANUP_IN_LIFECYCLE

Location	18:27-18:31
Message	'v-if' is not recognized as a Vue directive because it is used as the argument of 'v-bind'. Consider removing the preceding ':' if a directive was intended.
Rule	VUE_V_BIND_ON_DIRECTIVE

Location	20:17-20:28
Message	Consider removing '"shouldBye"' because 'v-else' cannot have an attribute value.
Rule	VUE_BAD_DIRECTIVE_FORMAT

Location	22:11-22:15
Message	Variables defined at 'v-for' are not used in this 'v-if' condition. Consider moving 'v-if' to a wrapper element if the condition does not change across iterations.
Rule	VUE_V_IF_WITH_V_FOR

Location	24:6-24:10
Message	Vue component iterated with 'v-for' should have a 'v-bind:key' attribute.
Rule	VUE_MISSING_KEY_ATTRIBUTE

Location	27:15-27:21
Message	'v-show' does not support '<template>'. Consider using 'v-if' instead.
Rule	VUE_TEMPLATE_WITH_V_SHOW