MELHORIAS NO PROCESSO DE IDENTIFICAÇÃO DE COMPONENTES VULNERÁVEIS: Decidindo sobre Atualizações

Bruna Vuicik Mocelin*

Kleinner Silva Farias de Oliveira**

Resumo: Aplicações podem conter vulnerabilidades por vários motivos, um deles é a utilização de componentes vulneráveis. Uma das soluções adotadas para eliminar as vulnerabilidades inseridas por tais componentes é a atualização do componente para uma versão mais recente que corrige a vulnerabilidade, porém, a atualização de um componente pode demandar refatoração de código, atualização de outros componentes e inserir novas vulnerabilidades na aplicação. Há várias ferramentas que realizam a análise e o gerenciamento de dependências dos projetos, porém, poucas apresentam informações sobre vulnerabilidades das novas versões, incompatibilidades e atualizações das dependências dos componentes. Este artigo apresenta a *dep*|*ct* (lê-se *depict*), uma ferramenta que tem como objetivo identificar os componentes vulneráveis conhecidos utilizados pelas aplicações e auxiliar na decisão sobre a atualização de tais componentes, de modo a mitigar as vulnerabilidades adicionadas aos projetos através das dependências vulneráveis. Resultados da avaliação empírica realizada sobre dois projetos mostram que a ferramenta pode ser utilizada para auxiliar na decisão sobre a atualização de componentes vulneráveis conhecidos e que ainda há aprimoramentos a serem realizados.

Palavras-chave: Componentes vulneráveis conhecidos. Atualização de componentes. Segurança de aplicações. Vulnerabilidades.

1 INTRODUÇÃO

Uma vulnerabilidade é um defeito no projeto, implementação, controles internos ou procedimentos de segurança de um sistema que, quando explorado, pode resultar em uma violação de segurança. (MCGRAW, 2006). Aplicações podem conter vulnerabilidades por vários motivos, incluindo alteração inapropriada do código da aplicação, falhas na especificação, defeitos de implementação e pressuposições incorretas sobre as entradas recebidas. (ALLEN, 2007).

Componentes vulneráveis conhecidos são aqueles que possuem alguma vulnerabilidade divulgada em meios públicos, como ferramentas de gerenciamento

* Estudante de Ciência da Computação na Universidade do Vale do Rio dos Sinos - UNISINOS, São Leopoldo, Brasil. E-mail: bvuicik@gmail.com.

** Professor assistente do Programa de Pós-Graduação em Computação Aplicada (PIPCA) da UNISINOS. Doutor em Informática pela Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio, Rio de Janeiro, Brasil. E-mail: kleinnerfarias@unisinos.br.

de defeitos e bancos de dados de vulnerabilidades. A utilização de tais componentes aumenta o risco de ocorrência de violações de segurança, pois uma vez que a vulnerabilidade é conhecida explorá-la se torna mais fácil, podendo envolver o uso de sequências de instruções específicas ou ferramentas automatizadas. (CADARIU, 2015).

Em uma tentativa de dar suporte à comunidade sobre segurança em aplicações, a *Open Web Application Security Project* (OWASP) publica periodicamente o relatório OWASP Top 10 com os 10 riscos de segurança mais críticos encontrados em aplicações *web*. A utilização de componentes vulneráveis conhecidos foi introduzida na nona posição do último relatório, publicado em 2013. (WILLIAMS; WICHERS, 2013). Ocorrências de grande repercussão também aumentam a atenção para a utilização de tais componentes, como a divulgação em 2014 da vulnerabilidade *Heartbleed*[1] na biblioteca de criptografia OpenSSL. (DURUMERIC et al., 2014). Informações, atividades e diretrizes relacionadas ao controle da utilização de componentes de terceiros vêm sendo incorporadas a padrões e organizações internacionais, incluindo o Padrão de Segurança de Dados da Indústria de Cartões de Pagamento (PCI SECURITY STANDARDS COUNCIL, 2015) e a *Financial Services Information Sharing and Analysis Center* (FINANCIAL SERVICES INFORMATION SHARING AND ANALYSIS CENTER, 2015).

Já existem aplicações que auxiliam na identificação da utilização de componentes vulneráveis conhecidos. Uma das ações para solucionar o problema é atualizar o componente vulnerável para uma versão mais recente, que corrige a vulnerabilidade, porém, a atualização do componente pode envolver outras questões não previstas e explicitadas pelas ferramentas atuais, incluindo refatoração de código, propagação de atualização de outros componentes e verificação de novas vulnerabilidades inseridas na aplicação através da atualização.

Este artigo tem como objetivo auxiliar na decisão sobre a atualização de componentes vulneráveis conhecidos, sendo tal atualização planejada com o propósito de mitigar as vulnerabilidades adicionadas aos projetos através das dependências vulneráveis. Para alcançar tal objetivo foi realizada uma comparação entre ferramentas de detecção de componentes vulneráveis conhecidos em aplicações e propõem-se a *dep*|*ct* (lê-se *depict*), uma ferramenta para identificar tais componentes e auxiliar na decisão sobre sua atualização.

O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta conceitos necessários para o entendimento do restante do artigo, a Seção 3 expõe alguns trabalhos relacionados, a Seção 4 apresenta a ferramenta proposta, a Seção 5 descreve os resultados da avaliação empírica e a Seção 6 apresenta as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta Seção são apresentados conceitos e informações necessárias para o entendimento do artigo. A Seção 2.1 introduz o ciclo de vida das vulnerabilidades e os identificadores *Common Vulnerabilities and Exposures* (CVE). A Seção 2.2 apresenta brevemente o banco de dados de vulnerabilidades *National Vulnerability Database* (NVD). A Seção 2.3 fornece uma introdução sobre incompatibilidade entre binários. A Seção 2.4 apresenta uma introdução sobre a ferramenta Maven.

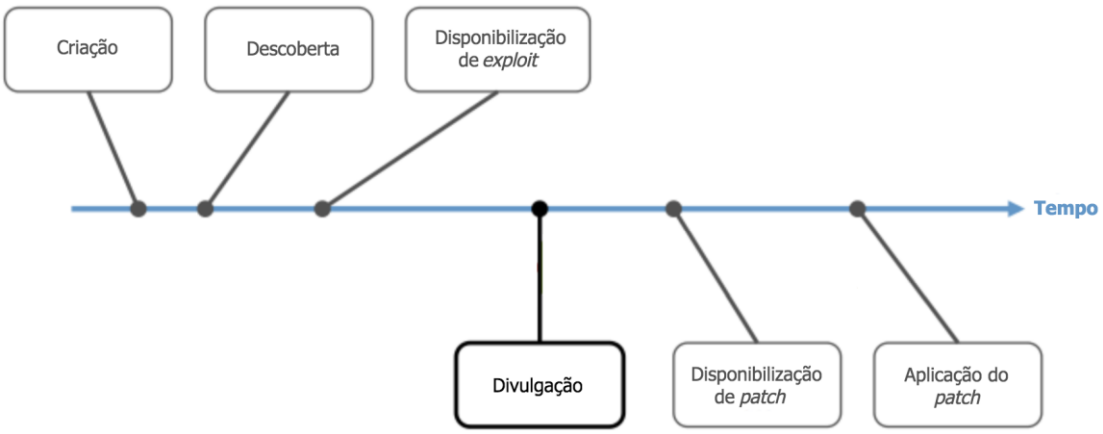
2.1 Componentes Vulneráveis Conhecidos e CVE

Segundo Frei (2013), o ciclo de vida das vulnerabilidades é composto por 6 eventos:

a) criação: momento em que a vulnerabilidade é inserida no *software*;
b) descoberta: momento em que se reconhece que a vulnerabilidade representa risco de segurança;
c) disponibilização de *exploit*: momento em que parte de um *software*, sequência de comandos ou conjunto de dados utilizados para explorar uma vulnerabilidade é disponibilizado;
d) divulgação: momento em que informações sobre uma vulnerabilidade são divulgadas em meios públicos;
e) disponibilização de *patch*: momento em que o fabricante disponibiliza uma correção que fornece proteção contra a exploração da vulnerabilidade;
f) aplicação do *patch*: momento em que os usuários do *software* aplicam a correção fornecida pelo fabricante.

Estes eventos são apresentados na Figura 1. A ordem de ocorrência pode ser alterada de acordo com o responsável pela descoberta: o fabricante pode decidir divulgar a vulnerabilidade apenas após a disponibilização da correção. (FREI, 2013).

Figura 1 - Ciclo de vida das vulnerabilidades



Fonte: Adaptado de FREI (2013, p. 6).

Componentes vulneráveis conhecidos são aqueles que possuem alguma vulnerabilidade divulgada em meios públicos, como bancos de dados de vulnerabilidades, relatórios de vulnerabilidades e ferramentas de gerenciamento de defeitos de *software*. Para facilitar a identificação das vulnerabilidades, integração entre sistemas e comunicação entre profissionais utilizam-se os identificadores CVE.

Os identificadores CVE são compostos por um número de identificação, descrição (incluindo produtos afetados e versões, impacto, pontuação calculada de acordo com o mecanismo padronizado de pontuação de vulnerabilidades *Common Vulnerability Scoring System* ou CVSS) e referências externas. (MURTAZA et al., 2016). Para identificar as versões de componentes afetadas por uma vulnerabilidade algumas fontes de dados utilizam a descrição em texto plano, outras, como o NVD, utilizam identificadores *Common Platform Enumeration* (CPE).

CPE é um método padronizado de identificação de aplicações, sistemas operacionais e dispositivos de *hardware* mantido pelo *National Institute of Standards and Technology* (NIST). Sua representação como *Uniform Resource Identifier* (URI) possui a seguinte formação: cpe:/[tipo]:[fornecedor]:[produto]:[versão]:[atualização]:[edição]:[idioma]. (CHEIKES; WALTERMIRE; SCARFONE, 2011).

2.2 Bancos de Dados de Vulnerabilidades e NVD

Dados sobre vulnerabilidades podem ser obtidos a partir de várias fontes, incluindo listas de e-mail, boletins de segurança e bancos de dados de vulnerabilidades. (JOSHI et al., 2013). Bancos de dados de vulnerabilidades são

repositórios que mantêm e disponibilizam informações sobre vulnerabilidades, a exemplo do NVD e HPI-VDB.

NVD trata-se de um banco de dados de vulnerabilidades mantido pelo governo dos Estados Unidos que inclui informações sobre vulnerabilidades para as quais já foi atribuído um identificador CVE e cuja situação seja diferente de “reservada”; tais informações também são disponibilizadas em arquivos XML. (NVD, [2016?]). Este banco de dados vem sendo utilizado em várias pesquisas – incluindo as apresentadas por Murtaza et al. (2016), Joshi et al. (2013), Cadariu et al. (2015) e Plate, Ponta e Sabetta (2015) – e ferramentas, como a OWASP Dependency Check.

2.3 Incompatibilidades entre Binários

De acordo com a especificação da linguagem Java, uma alteração em um tipo (uma classe, por exemplo) é compatível na forma binária se os binários ligados antes da alteração continuam a ser ligados sem erros após a alteração. (GOSLING et al., 2015). Deste modo, a incompatibilidade entre binários é gerada a partir de alterações – como a remoção de métodos públicos ou alteração do tipo de retorno – que requerem que o código que os utilizam seja modificado para fazer uso da nova versão.

Para identificar se uma versão de um componente adiciona mudanças incompatíveis com a versão anterior – indicando que possivelmente o código-fonte deverá ser alterado para utilizar a versão mais recente – pode-se utilizar o versionamento semântico, que sugere a utilização de 3 dígitos, cada um incrementado de acordo com as alterações realizadas na API pública, porém, muitos projetos ainda não aderiram a este esquema de versionamento. (RAEMAEKERS; VAN DEURSEN; VISSER, 2014).

2.4 Maven

Maven é uma ferramenta utilizada para o gerenciamento de projetos de *software*, incluindo o gerenciamento de dependências e o processo de construção. Os detalhes de configuração e dependências do projeto são incluídos em um arquivo XML chamado *Project Object Model* (POM). (THE APACHE SOFTWARE FOUNDATION (ASF), 2016a).

A identificação única dos projetos e dependências é realizada através de informações especificadas pelas *tags groupId* e *artifactId*. A *tag groupId* identifica a organização que criou o projeto; a *tag artifactId* indica o nome base do artefato gerado pelo projeto (geralmente o nome do arquivo sem a versão). A versão de determinado projeto ou dependência é especificada pela *tag version*. (ASF, 2016a).

A Figura 2 apresenta um exemplo de declaração de dependência: o projeto *Projeto de exemplo* utiliza a versão 1.10 da biblioteca *Commons Compress*. Quando uma dependência é incluída no projeto, todas as dependências desta dependência (chamadas de dependências transitivas) também são incluídas. (ASF, 2016b).

Figura 2 - Exemplo de declaração de dependência no arquivo *pom.xml*

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>br.unisinos.exemplo</groupId>
    <artifactId>projeto-de-exemplo</artifactId>
    <version>1.0.0</version>
    <name>Projeto de exemplo</name>

    [...]

    <dependencies>

        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-compress</artifactId>
            <version>1.10</version>
        </dependency>

        [...]

    </dependencies>
</project>
```

Fonte: Elaborado pela autora.

O Maven permite especificar o escopo de utilização das dependências, incluindo escopos como *test* e *compile*, que indicam, respectivamente, que a dependência é utilizada para os testes da aplicação e que está disponível em todos os *classpath* do projeto, sendo *compile* o escopo padrão utilizado. (ASF, 2016b).

3 TRABALHOS RELACIONADOS

Pesquisas relacionadas a componentes vulneráveis conhecidos buscam verificar o estado da utilização destes componentes nas aplicações, bem como automatizar os processos de identificação e análise. Desse modo, procura-se tornar a resposta às vulnerabilidades mais rápida. Além disso, há várias ferramentas que realizam a análise das dependências de projetos. Esta Seção apresenta as pesquisas relacionadas, ferramentas disponíveis e uma análise comparativa.

3.1 Utilização de Componentes Vulneráveis

Williams e Dabirsiaghi (2014) apresentaram os resultados da pesquisa realizada com mais de 3.300 participantes, na qual verificou-se que 75% das empresas não controlam efetivamente os componentes utilizados nas aplicações e apenas 22% das empresas baniram a utilização de componentes de código aberto que apresentam vulnerabilidades. Este relatório contempla também os resultados da análise de 113 milhões de *downloads* do repositório central do Maven realizados no período de um ano por mais de 60 mil organizações relativos a 31 *frameworks* e bibliotecas em Java. Através dessa análise verificou-se que 26% dos *downloads* são relativos a bibliotecas que apresentam vulnerabilidades conhecidas.

Hejderup (2015) realizou um estudo baseado no repositório *Node Package Manager* (npm) com o objetivo analisar a utilização de componentes vulneráveis conhecidos nos módulos JavaScript, contemplando 99.631 módulos e 22 avisos de segurança publicados entre julho de 2013 e outubro de 2014 no site *Node Security Project.* Através deste estudo verificou-se que 1.029 módulos possuem dependência direta com um componente vulnerável conhecido e outros 405 módulos possuem dependência indireta (transitiva). Este estudo indica que o contexto de utilização do módulo ou incompatibilidades entre as versões podem configurar razões para a não atualização do componente para uma versão que corrige a vulnerabilidade. Além disso, o estudo qualitativo posteriormente realizado sugere que na maioria dos módulos não há conhecimento sobre tais vulnerabilidades.

3.2 Automatização da Análise de Componentes

Cadariu et al. (2015) apresentaram um estudo de caso que buscou analisar se ferramentas de verificação de vulnerabilidades em dependências podem ser utilizadas no processo de garantia de qualidade de *software* realizado por empresas externas. O artigo apresenta um serviço de alerta de vulnerabilidades que utiliza a ferramenta *OWASP Dependency Check* para localizar vulnerabilidades em sistemas ao longo do ciclo de vida e gerar alertas para os operadores técnicos e partes interessadas. O serviço foi implantado no processo de monitoramento de qualidade de *software* oferecido pelo *Software Improvement Group* (SIG) e utilizado em 75 sistemas proprietários. O estudo revelou que 54 dos 75 projetos analisados utilizam

pelo menos uma biblioteca vulnerável e a avaliação concluiu que a maioria dos alertas gerados foram úteis; sem o serviço, impactos de segurança das bibliotecas desatualizadas geralmente não eram considerados, afirmando a possibilidade de integração de ferramentas similares no processo. (CADARIU et al., 2015).

O serviço proposto por Cadariu et al. (2015) não apresenta informações sobre novas versões dos componentes e consequências da atualização. Estes requisitos foram elencados, mas não foram implementados no serviço (CADARIU, 2014) e são disponibilizados pela ferramenta proposta neste trabalho. A emissão de alertas configura melhoria a ser incluída na ferramenta proposta.

Plate, Ponta e Sabetta (2015) apresentaram uma abordagem que visa facilitar a avaliação do impacto das vulnerabilidades dos componentes no seu contexto de utilização na aplicação. A abordagem tem como base a hipótese de que existe um risco de explorar a vulnerabilidade se a aplicação executa algum fragmento de código do componente vulnerável que foi alterado na versão com a correção. Foi implementado um protótipo que analisa as vulnerabilidades de componentes Java e interage com os sistemas de controle de versão Git e Subversion (SVN) para obter o código-fonte da versão vulnerável e não vulnerável do componente. Para verificar as mudanças entre as versões do componente foi utilizada a ferramenta ANTLR e para estabelecer as versões do componente que possuem a vulnerabilidade utilizam-se as *tags* do sistema de controle de versão ou as informações fornecidas pelo banco de dados NVD. Uma aplicação Java de teste foi submetida para análise e o protótipo identificou corretamente o componente vulnerável e a versão do componente com a correção de segurança. (PLATE; PONTA; SABETTA, 2015).

Enquanto Plate, Ponta e Sabetta (2015) verificam se a parte considerada vulnerável do componente é executada no contexto do projeto sendo analisado, a ferramenta proposta neste trabalho apenas sinaliza que o componente possui vulnerabilidades, sem verificar se o componente ou o trecho vulnerável é realmente utilizado pela aplicação. Este é considerado um trabalho futuro para a ferramenta.

3.3 Ferramentas

Há várias ferramentas disponíveis que visam a análise e o gerenciamento das dependências de projetos com relação a componentes vulneráveis conhecidos. Suporte a diferentes tipos de projetos e integração no processo de desenvolvimento

são características comuns a várias ferramentas, enquanto informações sobre vulnerabilidades das novas versões do componente, incompatibilidades entre versões e propagação de atualizações caracterizam oportunidades de melhoria.

Com o objetivo de proporcionar uma visão comparativa entre a ferramenta proposta e ferramentas semelhantes, o Quadro 1 apresenta a lista de ferramentas analisadas neste estudo e seu atendimento a alguns requisitos de acordo com informações e documentações obtidas através do site de cada ferramenta. O critério de avaliação “Facilidade de uso” não foi considerado na análise, pois as ferramentas possuem modos de utilização distintos (interface *web*, linha de comando, integração no processo de construção, dentre outros). Uma breve descrição sobre cada ferramenta analisada com informações auxiliares à comparação está disponível no Anexo A. As ferramentas *Contrast* e OWASP *Wordpress Vulnerability Scanner* foram classificadas como não aplicáveis para o critério “Integração no processo de desenvolvimento” porque realizam a análise do projeto através de sua execução.

De acordo com o comparativo, pode-se observar que metade das ferramentas analisadas obtém informações de vulnerabilidades de mais de uma fonte de dados, porém, as fontes são pré-definidas e novas fontes não podem ser adicionadas. Isto impede que o usuário selecione as fontes que deseja utilizar ou que utilize fontes próprias, com informações específicas e alinhadas às necessidades da equipe, sobre bibliotecas e *frameworks* implementados e utilizados pela organização. A ferramenta proposta possui maior flexibilidade, permitindo que novas integrações com fontes de dados sejam adicionadas através da implementação da interface *IVulnerabilityDatabase*, expandindo a diversidade de informações utilizadas.

Nota-se também que informações sobre as vulnerabilidades das outras versões do componente, propagação de atualizações e incompatibilidades entre versões são disponibilizadas por poucas ferramentas e apresentam áreas de melhoria em relação às ferramentas atuais. Estas informações auxiliam na medição do esforço a ser despendido com a atualização e ajudam a avaliar os riscos da atualização para as demais versões do componente. Estas três funcionalidades são contempladas na ferramenta *dep|ct* e auxiliam na visualização de informações relevantes a serem levadas em consideração na atualização dos componentes.

Quadro 1 - Comparativo entre ferramentas de análise de vulnerabilidades em dependências

Ferramenta	Critérios de Avaliação							
	Utilizar e manter fontes de dados atualizadas	Integração de fontes de dados distintas	Suporte a diferentes tipos de projetos	Integração no processo de desenvolvimento	Novas versões do componente	Incompatibilidades entre versões do componente	Identificar propagação de atualizações	Vulnerabilidades das outras versões do componente
dep|ct	+	+	+	+	+	+	+	+
Application Health Check	+	+	+	-	+	-	-	+
bitHound	+	-	+	+	+	+	-	-
Black Duck Hub	+	+	+	+	+	-	-	+
bundler-audit	+	+	-	+	+	-	-	-
Codenomicon AppCheck	+	+	+	+	-	-	-	-
Contrast	+	?	+	NA	+	-	-	-
Gemnasium	+	+	+	+	+	-	-	-
Nexus Lifecycle	+	+	+	+	+	-	-	+
Node Security Project	+	+	-	+	-	-	-	-
OWASP Dependency Check	+	-	+	+	-	-	-	-
OWASP Wordpress Vulnerability Scanner	+	-	-	NA	-	-	-	-
Palamida	+	-	+	?	-	-	-	-
Retire.js	+	+	-	+	-	-	-	-
SRC:CLR	+	+	+	+	+	-	-	+
The Victims Project	+	-	+	+	-	-	-	-
Veracode Software Composition Analysis	+	-	+	+	+	-	-	-
WhiteSource	+	?	+	+	-	-	+	-

Notas: + Suportado - Não suportado ? Não identificado NA Não aplicável

Fonte: Elaborado pela autora.

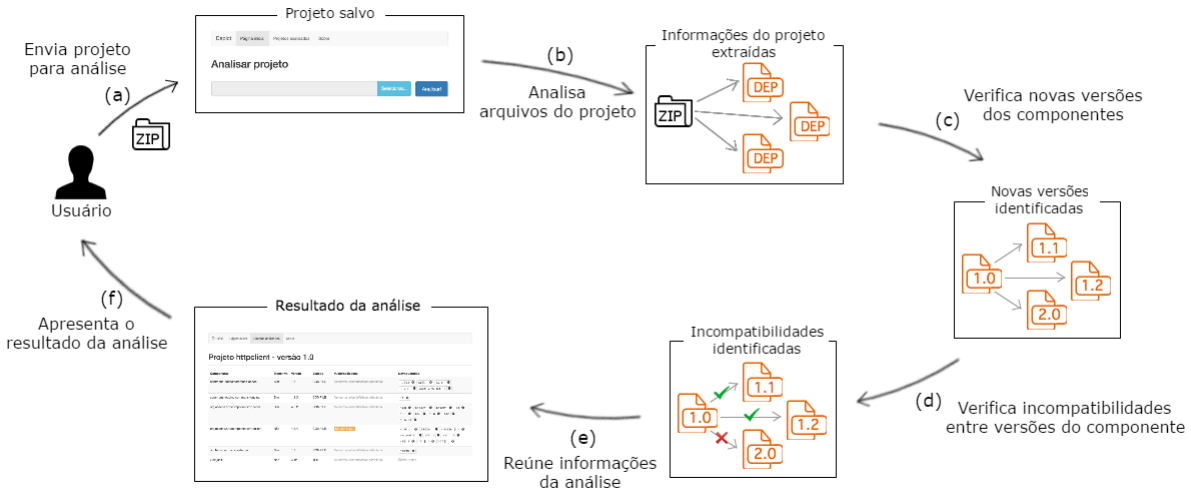
4 FERRAMENTA PROPOSTA

Esta Seção tem como objetivo apresentar a *dep|ct*, uma ferramenta para identificar componentes vulneráveis conhecidos utilizados pelas aplicações e auxiliar na decisão sobre a atualização de tais componentes. Para isto, a Seção 4.1 apresenta uma visão geral da ferramenta, a Seção 4.2 apresenta as etapas realizadas para o projeto da ferramenta e a Seção 4.3 apresenta os aspectos de implementação.

4.1 Visão Geral

A ferramenta proposta *lê* o código-fonte de projetos de *software* e *identifica* os componentes configurados como dependências, suas vulnerabilidades e as incompatibilidades entre as versões atual e mais recentes. Após, a ferramenta *recomenda* versões de componentes que sejam mais recentes, não apresentam – ou apresentam menos – vulnerabilidades conhecidas e causem menor impacto de atualização considerando as incompatibilidades entre versões.

Figura 3 - Fluxo geral da análise de um projeto



Fonte: Elaborado pela autora.

A Figura 3 apresenta uma visão geral da ferramenta com os principais passos executados para realizar a análise de um projeto, descritos a seguir:

a) o usuário submete um projeto para análise;

b) o sistema extrai informações sobre o projeto, como nome, versão e componentes configurados como dependências;
c) o sistema verifica se os componentes possuem versões mais recentes;
d) o sistema verifica se há incompatibilidades entre a versão do componente utilizada pelo projeto e versões mais recentes;
e) o sistema reúne as informações geradas pela análise, verifica se os componentes possuem vulnerabilidades conhecidas e recomenda uma configuração de versões dos componentes; e
f) o resultado da análise é apresentado ao usuário.

Os passos *b*, *c* e *d* podem ser customizados através da implementação de interfaces, que caracterizam pontos de extensão do comportamento da ferramenta.

4.2 Projeto da Ferramenta

O projeto da ferramenta foi realizado em duas etapas. Na primeira, foram identificados os requisitos de ferramentas de detecção de componentes vulneráveis de acordo com a literatura. Deu-se prioridade àqueles que auxiliam na decisão sobre a atualização de tais componentes. Na segunda etapa, um projeto arquitetural foi elaborado visando dar suporte aos requisitos previamente elicitados. Neste sentido, foi elaborada uma arquitetura em camadas, bem como um diagrama de componentes da UML, com objetivo de identificar os principais componentes da ferramenta. Ambas as etapas são descritas a seguir.

4.2.1 Requisitos da Ferramenta

Os requisitos foram obtidos através da leitura de relatórios, observações realizadas sobre outras pesquisas e através da comparação entre ferramentas similares à proposta. Uma breve descrição de cada requisito funcional (RF) e não funcional (RNF) é apresentada a seguir.

RF01: utilizar e manter fontes de dados atualizadas. A lista de vulnerabilidades utilizada pela aplicação deve ser atualizada periodicamente, pois uma vez que a vulnerabilidade é publicada, explorá-la se tornará mais fácil. O Relatório de Investigações de Violações de Dados publicado em 2015 pela Verizon mostra que aproximadamente 50% das vulnerabilidades que possuem CVE

exploradas em 2014 o foram no período de 1 mês após a publicação do CVE (VERIZON ENTERPRISE SOLUTIONS, 2015).

RF02: integração de fontes de dados distintas de vulnerabilidades. Através da utilização de diferentes fontes de dados é possível obter informações complementares sobre determinada vulnerabilidade ou novas vulnerabilidades. Atualmente, as fontes de dados de vulnerabilidades são heterogêneas e descentralizadas. O banco de dados NVD, por exemplo, apresenta apenas vulnerabilidades com situação diferente de “reservada” e que possuem identificador CVE (NVD, [2016?]). Dessa forma, vulnerabilidades como a CVE-2015-3192[2], que em maio de 2016 ainda não estava disponível no NVD mesmo que descrita em relatórios da empresa mantenedora desde maio de 2015, demoram a ser disponibilizadas.

RF03: permitir extração de dependências de diferentes tipos de projetos. A ferramenta deve permitir a extração e análise de dependências de diferentes tipos de projetos, implementados em diferentes linguagens de programação. Esta funcionalidade é implementada pela maioria das ferramentas similares analisadas nos trabalhos relacionados.

RF04: listar novas versões do componente. A ferramenta deve indicar se novas versões do componente que corrigem a vulnerabilidade estão disponíveis, pois neste caso atualizar o componente é uma das soluções para remover a vulnerabilidade do projeto. Cadariu (2014) destaca a importância desta funcionalidade em sua pesquisa.

RF05: listar incompatibilidades entre a versão atual e a nova versão do componente. A ferramenta deve sinalizar incompatibilidades entre versões do componente de modo a auxiliar na identificação de possíveis alterações a serem realizadas no código-fonte do projeto ao alterar a versão do componente. Cadariu (2014) apresentou este requisito em seu estudo de caso e, apesar de não ter sido implementado no estudo, foi sugerido como um requisito que desperta interesse e que poderia ser considerado pelas ferramentas.

RF06: identificar propagação de atualizações. A ferramenta deve indicar a propagação de atualizações, pois a atualização de um componente pode implicar na atualização de suas dependências. Esta identificação potencializa um melhor planejamento da atualização e foi elencada como trabalho futuro no estudo de caso realizado por Cadariu (2014).

RF07: listar vulnerabilidades das outras versões do componente. A ferramenta deve identificar as vulnerabilidades das novas versões do componente. Versões mais recentes de componentes podem conter as mesmas vulnerabilidades das versões anteriores ou novas vulnerabilidades, talvez mais críticas.

RNF01: facilidade de uso. Deve ser simples e rápido submeter um projeto e a visualização da análise deve ser direta, não exigindo navegação extensiva na ferramenta ou conhecimentos avançados sobre segurança. Allen (2007) aponta a falta de treinamento e conhecimentos sobre segurança como um dos motivos da inclusão de vulnerabilidades em aplicações. Além disso, Wurster e Van Oorschot (2008) apontam a importância de não exigir conhecimentos avançados sobre segurança em ferramentas de análise de segurança, de modo que o desenvolvedor consiga compreender os erros reportados pela ferramenta.

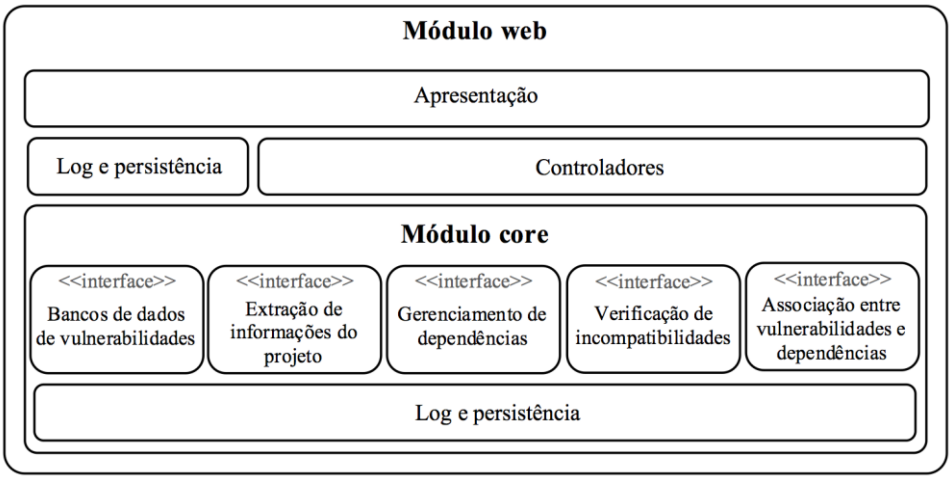
RNF02: possibilidade de estender a utilização da ferramenta. A ferramenta deve ser desenvolvida de modo a possibilitar sua reutilização, permitindo integrá-la no processo de desenvolvimento de *software* – através de *plugins* para ambientes integrados de desenvolvimento (IDEs) ou aplicações de integração contínua, por exemplo. Deste modo, é possível integrar a aplicação com as ferramentas já utilizadas pelos desenvolvedores, pois como ressaltam Wurster e Van Oorschot (2008), não há como garantir que todos os desenvolvedores utilizem uma ferramenta de suporte que deve ser executada de forma independente.

4.2.2 Arquitetura e Componentes

Visando um projeto flexível, extensível e com alto grau de reuso, propõe-se uma arquitetura baseada em componentes organizada em camadas. A arquitetura possui dois módulos principais – *core* e *web* – cuja disposição em camadas e componentes em alto nível são apresentados na Figura 4. A Figura 5, por sua vez, apresenta o componente principal da ferramenta e suas interfaces requeridas e providas pela implementação padrão.

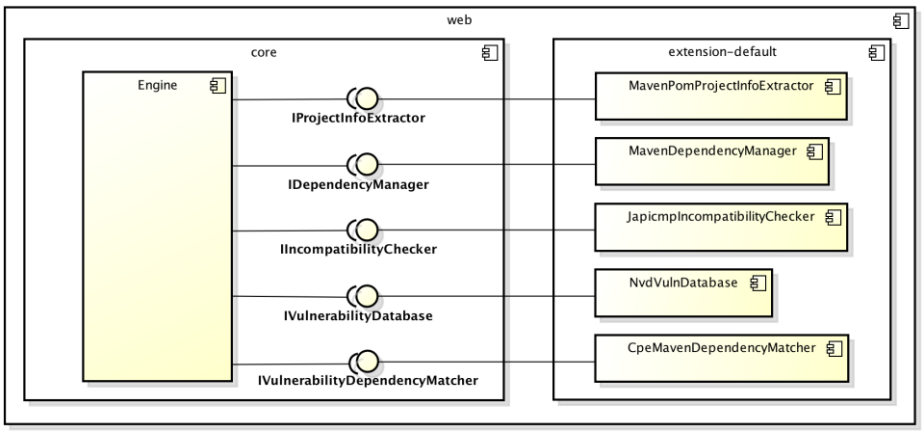
O módulo *core* é responsável por coordenar o fluxo de execução da aplicação. Este módulo não possui interface gráfica e tem como finalidade a utilização por outros módulos ou aplicações. O principal componente do módulo *core* é a *Engine,* responsável pela análise dos projetos e sincronização das fontes de dados de vulnerabilidades.

Figura 4 - Módulo *web* e seus componentes em alto nível



Fonte: Elaborado pela autora.

Figura 5 - Componente principal da ferramenta e suas interfaces requeridas e providas pela implementação padrão



Fonte: Elaborado pela autora.

A *Engine* exige a implementação de determinadas interfaces, que se caracterizam como pontos de extensão onde o comportamento da ferramenta poderá ser alterado, são elas:

a) *IProjectInfoExtractor*: define as operações que devem ser implementadas para extração de informações (como nome e versão) e dependências do projeto a ser analisado;

b) *IDependencyManager*: define as operações que devem ser implementadas para verificar as versões disponíveis para determinado componente;

c) *IIncompatibilityChecker*: define as operações a serem implementadas por componentes que verificam incompatibilidades entre duas versões de um

mesmo componente;

d) *IVulnerabilityDatabase*: define as operações que devem ser implementadas para obtenção das vulnerabilidades a partir de uma fonte externa. O componente que implementa esta interface também poderá sinalizar os componentes e versões afetados pela vulnerabilidade.
e) *IVulnerabilityDependencyMatcher*: define as operações que devem ser implementadas para realizar a associação entre vulnerabilidades e dependências. Esta interface deve ser implementada quando a integração com as fontes de dados de vulnerabilidades não realizar esta associação.

O módulo *web* utiliza o módulo *core* e é responsável por disponibilizar a interface *web* (camada de apresentação) para o usuário final. A comunicação entre a camada de apresentação e a aplicação é realizada através de serviços REST, disponibilizados pela camada de controladores.

O módulo *web* utiliza também o módulo secundário *extension-default*, que fornece componentes com implementações padrão.

4.3 Aspectos de Implementação

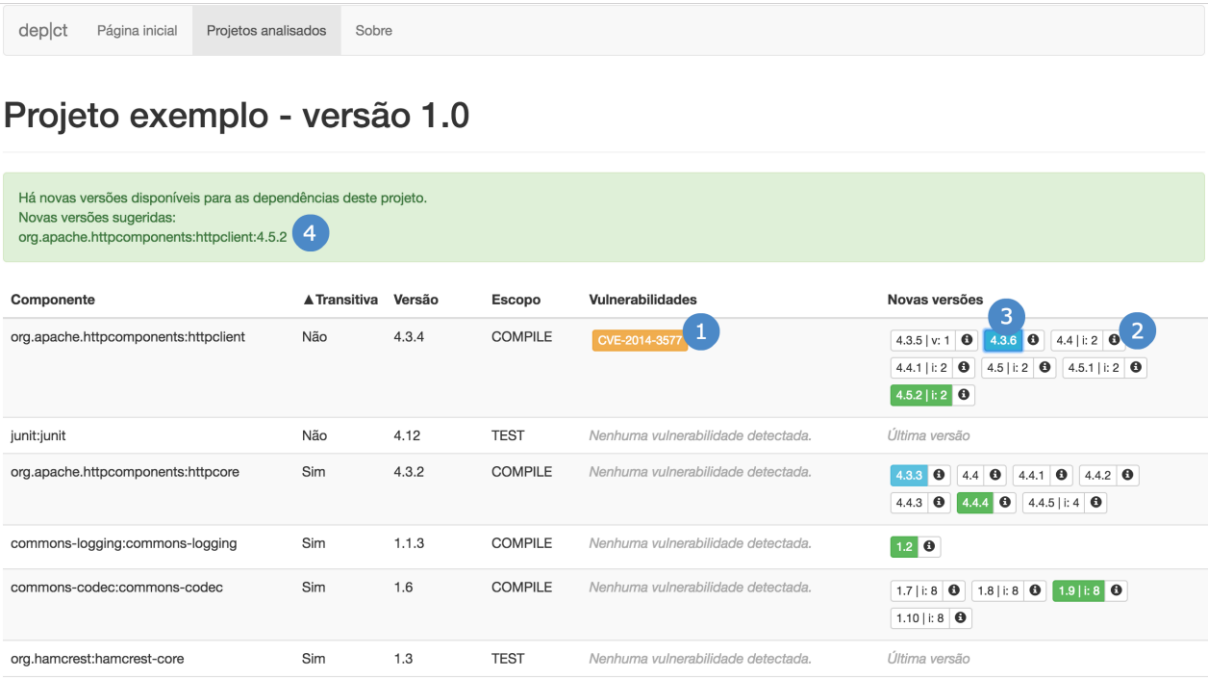
A ferramenta foi implementada na linguagem Java e utiliza Maven para o gerenciamento da construção e dependências do projeto. Foi utilizado o *framework* Spring, incluindo Spring Boot para simplificar a configuração e tornar a aplicação *stand-alone*, Spring Web para implementação da API REST e Spring Data JPA em conjunto com o *framework* Hibernate para a camada de persistência. A interface *web* foi desenvolvida utilizando os *frameworks* AngularJS e Bootstrap. Para o armazenamento dos dados foi utilizado o banco de dados relacional PostgreSQL.

A Figura 6 apresenta o resultado da análise de um projeto de exemplo que contém apenas duas dependências: *httpclient* e *junit*. A visualização do resultado da análise apresenta os componentes identificados, versão, escopo de utilização na aplicação, vulnerabilidades e novas versões identificadas. As informações do projeto foram extraídas pelo componente *MavenPomProjectInfoExtractor* a partir do arquivo pom.xml utilizado pelo Maven.

Ao clicar sobre o item 1 o sistema apresenta mais informações sobre a vulnerabilidade. O componente *NVDVulnDatabase* busca as vulnerabilidades no NVD e o componente *CpeMavenDependencyMatcher* associa as vulnerabilidades às

dependências do Maven pesquisando pelas informações “fornecedor”, “produto”, “versão” e “atualização” contidas no CPE em índices construídos com base no repositório remoto do Maven. É importante ressaltar que esta abordagem pode acarretar em falsos positivos e falsos negativos. A diferença de nomenclatura entre CPEs e dependências do Maven já foi sinalizada por outros autores, como Cadariu et al. (2015) e Plate, Ponta e Sabetta (2015) e não será elaborada neste artigo. O Apêndice A apresenta o quadro de critérios utilizados na pesquisa.

Figura 6 - Resultado da análise de um projeto



Fonte: Elaborado pela autora.

Ao clicar sobre o botão de informações sinalizado pelo número 2 o sistema apresenta mais informações sobre a nova versão do componente, incluindo lista de vulnerabilidades encontradas para a versão, lista de incompatibilidades entre a versão utilizada pelo projeto e a versão selecionada e lista de outros componentes que deverão ser atualizados ao utilizar a versão selecionada. O componente *MavenDependencyManager* identifica as novas versões através de pesquisa nos índices do Maven utilizando os atributos *groupId* e *artifactId*.

O botão relativo à nova versão (sinalizado pelo número 3) apresenta a contagem de incompatibilidades encontradas entre a versão utilizada pelo projeto e a nova versão (indicada pela letra “i”) e a contagem de vulnerabilidades (indicada

pela letra “v”). Ao clicar sobre este botão o sistema sinaliza em azul todos os componentes que deverão ser alterados caso o componente seja atualizado para a versão selecionada. As incompatibilidades são identificadas através do componente *JapicmpIncompatibilityChecker*, que utiliza a ferramenta japicmp[3].

Figura 7 - Fórmula utilizada para calcular o custo de uma configuração

$$CustoVulnerabilidades = \sum_{i=0}^{numVulns} (pontuacaoVuln_i) + numVulns * 5$$

$$CustoIncompatibilidades = numIncompatibilidades$$

$$CustoConfiguracao = CustoVulnerabilidades + CustoIncompatibilidades$$

Fonte: Elaborado pela autora.

Figura 8 - Algoritmo utilizado para seleção da configuração recomendada de versões

Algoritmo: Seleção da configuração recomendada de versões

Entrada: Configuração de versões do projeto

Saída: Configuração recomendada de versões para o projeto

1: Configura *custo padrão* igual a 5
2: Configura *penalidade por vulnerabilidade* igual a 5
3: Configura *penalidade por incompatibilidade* igual a 1
4: Calcula o *custo da configuração atual*
5: **Se** o custo da configuração atual é 0 **Então** utiliza o *custo padrão*
6: **Para cada** dependência do projeto armazena as novas versões disponíveis
7: **Faça**
8: Seleciona última versão armazenada de cada componente[1]
9: Monta uma nova configuração recomendada com todas as versões selecionadas
10: Calcula custo da configuração recomendada
11: **Se** custo da configuração recomendada é maior do que o custo da configuração atual
Então remove versão armazenada de um dos componentes[2]
12: **Enquanto** foi possível remover versão armazenada de um dos componentes **E**
o custo da configuração recomendada é maior do que o custo da configuração atual

[1] Apenas versões com custo de vulnerabilidades menor do que a versão utilizada pelo projeto são selecionadas.
[2] A versão com maior custo é selecionada para remoção, exceto quando possui menos vulnerabilidades do que a versão atual utilizada pelo projeto.

Fonte: Elaborado pela autora.

O item 4 sinaliza a recomendação de configuração apresentada pela ferramenta. Uma configuração é um conjunto de versões das dependências não transitivas de determinado projeto. Esta recomendação prioriza as versões mais recentes e com menos vulnerabilidades dos componentes utilizados no projeto e é realizada com base no custo da configuração atual e da configuração recomendada

para o projeto. A Figura 7 apresenta a fórmula utilizada para calcular o custo de uma configuração. Na fórmula são consideradas as vulnerabilidades de todas as dependências (transitivas e não transitivas) com escopo diferente de “teste” e as incompatibilidades apenas das dependências não transitivas.

A Figura 8 apresenta em alto nível o algoritmo utilizado para seleção da configuração recomendada para um projeto.

5 AVALIAÇÃO DA FERRAMENTA

Esta Seção apresenta uma avaliação empírica da ferramenta. A Seção 5.1 apresenta o método utilizado para a avaliação. A Seção 5.2 apresenta os resultados, incluindo melhorias identificadas para a ferramenta.

5.1 Método de Avaliação

O propósito da avaliação da ferramenta é apresentar alguns resultados obtidos a partir da análise dos projetos utilizando a implementação padrão dos pontos de extensão da ferramenta, demonstrando que a ferramenta auxilia na decisão sobre a atualização de componentes vulneráveis conhecidos.

Para avaliar a ferramenta foram selecionados 2 projetos: Apache Storm e Activiti. Estes projetos foram selecionados pois tratam-se de aplicações robustas, desenvolvidas em Java, que utilizam Maven para o gerenciamento de dependências, estão publicadas no GitHub e possuem propósitos distintos. Além disso, o fornecedor da primeira ferramenta é mundialmente conhecido e possui diversas ferramentas. Como estes projetos são compostos por vários módulos (vários arquivos POM) e a implementação padrão permite analisar apenas um arquivo POM de cada vez, foi selecionado um módulo de cada projeto para ser analisado.

Foram selecionadas 3 versões de cada projeto da seguinte forma: primeira versão na qual o módulo foi introduzido, versão intermediária – considerando a primeira e a última versão selecionadas – e última versão. O Quadro 2 apresenta uma breve descrição sobre cada projeto, além dos módulos e versões selecionadas, incluindo a data da última alteração da versão.

Para submeter os projetos foi obtido o arquivo compactado contendo todos os arquivos do projeto na versão selecionada através da interface do GitHub. Após isso,

o arquivo foi descompactado e foram removidas todas as pastas não relativas ao módulo selecionado, criando-se um novo arquivo compactado apenas com o módulo selecionado para análise e demais arquivos da raiz do projeto.

A descrição das métricas extraídas são apresentadas no Quadro 3. Nenhuma das métricas considera dependências com escopo de testes.

Quadro 2 - Descrição dos projetos selecionados

Projeto	**Apache Storm**	**Activiti**
Descrição	Sistema de computação distribuído que permite processar dados em tempo real.	Plataforma de gerenciamento de processos de negócio (BPM) e fluxo de trabalho.
Módulo	storm-core	activiti-engine
Endereço	http://storm.apache.org/	http://activiti.org/
Repositório	https://github.com/apache/storm	https://github.com/Activiti/Activiti
Versões selecionadas	0.9.3 (19/11/2014) 0.10.0 (23/10/2015) 1.1.0 (15/07/2016)	5.0 (31/01/2011) 5.14 (21/10/2013) 5.21.0 (13/06/2016)

Fonte: Elaborado pela autora.

Quadro 3 - Descrição das métricas

Métrica	**Descrição**
#Deps	Número total de dependências do projeto
#DepsTransitivas	Número de dependências transitivas do projeto
#DepsComVulnerabilidades	Número total de dependências com vulnerabilidades
#DepsTransitivasComVulnerabilidades	Número de dependências transitivas com vulnerabilidades
#Vulns	Número total de vulnerabilidades detectadas
#VulnsTransitivas	Número de vulnerabilidades detectadas introduzidas através de dependências transitivas
#VulnsOWASPDependencyCheck	Número total de vulnerabilidades detectadas pela ferramenta OWASP Dependency Check

Fonte: Elaborado pela autora.

5.2 Resultados

O Quadro 4 apresenta os valores obtidos de cada métrica para cada versão dos projetos analisados. Para referência, o número de vulnerabilidades encontradas pela ferramenta *OWASP Dependency Check* também foi adicionado. De acordo com os dados, pode-se notar que há crescimento no número de dependências para os

dois módulos e que há grande divergência entre o número de vulnerabilidades encontradas pela ferramenta proposta e pela *OWASP Dependency Check*, ocasionada pelas duas limitações descritas a seguir.

Quadro 4 – Métricas obtidas através da análise dos projetos

Módulo	**Apache Storm storm-core**			**Activiti activiti-engine**		
Versão	**0.9.3**	**0.10.0**	**1.1.0**	**5.0**	**5.14**	**5.21.0**
#Deps	63	66	75	21	34	38
#DepsTransitivas	38	26	28	12	19	20
#DepsComVulnerabilidades	5	4	4	4	3	3
#DepsTransitivasComVulnerabilidades	3	1	1	2	2	2
#VulnsOWASPDependencyCheck	8	6	6	9	9	3
#VulnsTransitivas	5	1	1	7	8	2
#VulnsOWASPDependencyCheck	15	16	15	8	14	5

Fonte: Elaborado pela autora.

A partir da execução das análises verificou-se que as implementações padrão não identificaram vulnerabilidades que não possuem CPEs listados para todas as versões afetadas, como é o caso do CVE-2015-5262, que afeta todas as versões do componente *Apache HttpComponents HttpClient* anteriores à versão 4.3.6. Isto ocorre devido ao formato de representação dos dados nos arquivos do NVD, pois a ferramenta utiliza os *feeds* disponibilizados na versão 2.0 e este formato não possui nenhuma indicação quando a vulnerabilidade afeta versões anteriores a do CPE indicado. Como melhoria, o componente responsável pela integração com o NVD poderia extrair esta informação do *feed* na versão 1.2.1 ou da descrição da vulnerabilidade.

Outra limitação identificada diz respeito à sinalização de versões incompletas no CPE da vulnerabilidade, como é o caso do CVE-2007-5613, que indica as versões compostas por 1 e 2 dígitos *cpe:/a:mortbay_jetty:jetty:6* e *cpe:/a:mortbay_jetty:jetty:6.1*. Como a ferramenta adiciona caracteres curinga no final da versão indicada no CPE para obter componentes com complementos de versão como “release” e “final”, outras versões deste componente também são incluídas na lista de componentes vulneráveis (por exemplo, 6.1.26).

Estes dois aprimoramentos da implementação padrão são importantes para

diminuir o número de falsos positivos e falsos negativos gerados pela ferramenta.

Para verificar se os dados apresentados pela ferramenta – e utilizados na recomendação de configuração – auxiliam a decidir sobre a atualização dos componentes com o objetivo de melhorar o quadro de vulnerabilidades das aplicações, o Quadro 5 apresenta as métricas de vulnerabilidades das configurações recomendadas considerando a última versão de cada módulo analisado.

Quadro 5 - Métricas de vulnerabilidades sobre a configuração recomendada

Módulo	**Apache Storm storm-core**		**Activiti activiti-engine**	
Versão	**1.1.0**	**Recomendação**	**5.21.0**	**Recomendação**
#Vulns	6	3	3	2
#VulnsTransitivas	1	1	2	1
#DepsComVulnerabilidades	4	2	3	2
#DepsTransitivasComVulnerabilidades	1	1	2	1

Fonte: Elaborado pela autora.

As vulnerabilidades restantes do módulo *storm-core* são relativas ao componente *Apache ZooKeeper* (que não possui novas versões) e ao componente *Apache HttpComponents HttpClient*, que possui versão mais recente, porém, como é uma dependência transitiva do componente *Apache Thrift* e este não possui novas versões, não foi considerada na recomendação.

A vulnerabilidade removida do módulo *activiti-engine* foi consequência da atualização do componente *spring-beans* da versão 4.1.5.RELEASE para a versão 4.3.3.RELEASE. As duas vulnerabilidades restantes são falsos positivos introduzidos pela forma de associação dos componentes afetados pela vulnerabilidade.

De modo geral, considera-se que a ferramenta apresentada pode ser utilizada para auxiliar na decisão sobre a atualização de componentes vulneráveis conhecidos, porém, ainda há melhorias a serem implementadas e devem ser realizados mais estudos sobre a efetividade da ferramenta.

Além das melhorias citadas anteriormente, outras melhorias foram identificadas através da submissão dos projetos e análise dos resultados: (a) considerar o impacto da vulnerabilidade no contexto de uso do componente na aplicação, tanto para sinalizar a vulnerabilidade, quanto para recomendar uma nova

configuração; (b) integração com sistemas de controle de versão e notificação por e-mail dos resultados da análise e inclusão de novas vulnerabilidades, evitando assim a intervenção manual para execução da ferramenta; (c) apresentar resumo com métricas do projeto analisado, incluindo número de dependências e vulnerabilidades, tanto da configuração atual quanto da configuração recomendada; (d) exibir a árvore de dependências, pois atualmente não é possível identificar a origem da inclusão de uma dependência no projeto; e (e) aprimorar o processamento dos projetos, pois quanto maior o número de dependências e versões destas dependências, maior o tempo de processamento necessário para analisar o projeto.

6 CONCLUSÃO

A utilização de componentes vulneráveis conhecidos é um problema que ganhou maior destaque após ser incluído no relatório da OWASP em 2013. Várias ferramentas auxiliam a identificar a utilização de tais componentes, porém, poucas apresentam informações que auxiliam na mitigação das vulnerabilidades através da atualização do componente.

Este artigo apresentou a *dep*|*ct*, uma ferramenta flexível que identifica a utilização de componentes vulneráveis conhecidos, apresenta informações que auxiliam na decisão sobre a atualização de tais componentes e recomenda uma configuração de versões com o objetivo de mitigar as vulnerabilidades adicionadas aos projetos através das dependências vulneráveis. Foram descritas as implementações padrão das interfaces requeridas pelo módulo *core*, que utilizam o NVD como fonte de dados de vulnerabilidades e permitem a análise de projetos Java que utilizam Maven para o gerenciamento de dependências.

A avaliação empírica realizada sobre a implementação padrão demonstra que a ferramenta pode ser utilizada para auxiliar na decisão sobre a atualização dos componentes vulneráveis conhecidos identificados. As limitações encontradas na avaliação são relativas à implementação padrão e não à arquitetura da ferramenta e são consideradas melhorias a serem implementadas. Tanto a implementação padrão quanto a ferramenta ainda necessitam de melhorias, sendo as principais: aprimorar a associação entre vulnerabilidades e dependências, considerar o impacto da vulnerabilidade no contexto de uso do componente na aplicação, permitir integração com sistemas de controle de versão e enviar notificações por e-mail.

IMPROVEMENTS IN THE PROCESS OF IDENTIFYING VULNERABLE COMPONENTS: DECIDING ON UPGRADES

Abstract: Applications can contain vulnerabilities for many reasons, one of them being the usage of vulnerable components. One solution to remove the vulnerabilities introduced by these components is to upgrade the component to a more recent, non-vulnerable version, but upgrading a component may require code refactoring, the upgrade of other components and can add new vulnerabilities to the application. There are many tools which can be used to analyze and manage the dependencies of a project, but few of them show information about the vulnerabilities of new versions, incompatibilities and upgrades of the dependencies of the components. This paper introduces dep|ct (pronounced as depict), a tool that aims to identify the known vulnerable components used by applications and to help decide on upgrading these components as a way to mitigate the vulnerabilities added to projects by the usage of such vulnerable dependencies. Results of the empirical evaluation conducted with two projects show that the tool can be used to help decide on the known vulnerable components upgrade and that there are still enhancements to be implemented.

Keywords*:* Known vulnerable components. Components update. Application security. Vulnerabilities.

NOTAS(S) EXPLICATIVAS (S)

[1] A biblioteca OpenSSL é utilizada em servidores como Apache e Nginx para prover segurança às conexões e através da vulnerabilidade *Hearthbleed* é possível roubar informações (como chaves criptográficas, usuários e senhas) de sistemas protegidos pelas versões vulneráveis. Informações sobre a vulnerabilidade estão disponíveis no NVD em https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[2] Vulnerabilidade CVE-2015-3192 relativa à utilização de declarações DTD (*Document Type Definition*) para realizar ataques de negação de serviço. Página do identificador CVE-2015-3192 disponível em https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3192. Acesso em: 9 maio 2016.

[3] Página da ferramenta japicmp disponível em https://siom79.github.io/japicmp/.

REFERÊNCIAS

ALLEN, Julia. Why is Security a Software Issue?. **EDPACS: The EDP Audit, Control, and Security Newsletter**, Pittsburgh, v. 36, n. 1, p. 1-13, Aug. 2007. Disponível em: <http://www.tandfonline.com/doi/ref/10.1080/07366980701500734>. Acesso em: 13 mar. 2016.

BITHOUND INC. **FEATURES**. Kitchener, 2016. Disponível em: <https://www.bithound.io/features>. Acesso em: 19 jun. 2016.

BLACK DUCK SOFTWARE, INC. **Black Duck Hub Provides Most Comprehensive and Earliest Alerts on New Open Source**. Burlington, 2015. Disponível em: <https://www.blackducksoftware.com/about/news-events/releases/black-duck-hub-provides-most-comprehensive-and-earliest-alerts-on-new-open-source>. Acesso em: 19 jun. 2016.

_____. **Black Duck | Hub**. Burlington, 2016. Disponível em: <https://info.blackducksoftware.com/rs/872-OLS-526/images/BlackDuck_HUB_UL.pdf>. Acesso em: 19 jun. 2016.

BUNDLER-AUDIT. **Patch-level verification for Bundler**. [S.l.], 2016. Disponível em: <https://github.com/rubysec/bundler-audit>. Acesso em: 19 jun. 2016.

CADARIU, Mircea. **Tracking Known Security Vulnerabilities in Third-party Components**, Netherlands. 2014. 86 f. Dissertação (Mestrado em Ciência da Computação) - Programa de Pós-Graduação em Ciência da Computação, Delft University of Technology, Holanda, 2014.

CADARIU, Mircea et al. Tracking Known Security Vulnerabilities in Proprietary Software Systems. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**, Montreal, QC, p. 516-519, Mar. 2015. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7081868&tag=1>. Acesso em: 27 nov. 2015.

CHEIKES, Brant A.; WALTERMIRE, David; SCARFONE, Karen. Common Platform Enumeration: Naming Specification Version 2.3. **NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST)**, Gaithersburg, Aug. 2011. Disponível em: <http://csrc.nist.gov/publications/nistir/ir7695/NISTIR-7695-CPE-Naming.pdf>. Acesso em: 5 jun. 2016.

CODENOMICON LTD. **AppCheck - Frequently Asked Questions**. Oulu, [2016?]. Disponível em: <http://www.codenomicon.com/products/appcheck/faq/>. Acesso em: 19 jun. 2016.

CONTRAST SECURITY. **Complete Coverage Of Today's Modern Applications**. Palo Alto, 2016. Disponível em: <https://www.contrastsecurity.com/supported-technologies>. Acesso em: 19 jun. 2016.

DURUMERIC, Zakir et al. The Matter of Heartbleed. **Proceedings of the 2014 Conference on Internet Measurement Conference**, Vancouver, p. 475-488, Nov. 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2663755>. Acesso em: 17 abr. 2016.

FINANCIAL SERVICES INFORMATION SHARING AND ANALYSIS CENTER. **Appropriate Software Security Control Types for Third Party Service and Product Providers**. [S.l.], Oct. 2015. Disponível em: <https://www.fsisac.com/sites/default/files/news/Appropriate%20Software%20Security%20Control%20Types%20for%20Third%20Party%20Service%20and%20Product%20Providers.pdf>. Acesso em: 17 abr. 2016.

FREI, Stefan. The Known Unknowns: Empirical Analysis of Publicly Unknown Security Vulnerabilities. **NSS Labs**, Austin, Dec. 5, 2013. Disponível em: <https://www.nsslabs.com/research-advisory/library/infrastructure-security/general/the-known-unknowns/>. Acesso em: 2 jun. 2016.

GEMNASIUM. **FEATURES**. Paris, 2016. Disponível em: <https://gemnasium.com/features>. Acesso em: 19 jun. 2016.

_____. **Security alerts go free**. Paris, May 5, 2015. Disponível em: <http://blog.gemnasium.com/post/118186520636/security-alerts-go-free>. Acesso em: 19 jun. 2016.

GOSLING, James et al. **The Java® Language Specification**: Java SE 8 Edition. Redwood City: Oracle America, Inc. and/or its affiliates, 2015. Disponível em: <http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>. Acesso em: 8 jun. 2016.

HEJDERUP, Joseph. **In Dependencies We Trust:** How Vulnerable are Dependencies in Software Modules?. 2015. 89 f. Dissertação (Mestrado em Ciência da Computação) -- Programa de Pós-Graduação em Ciência da Computação, Delft University of Technology, Holanda, 2015.

JOSHI, Arnav et al. Extracting Cybersecurity Related Linked Data from Text. In: **2013 IEEE Seventh International Conference on Semantic Computing (ICSC)**, Irvine, CA, p. 252-259, Sept. 2013. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6693525>. Acesso em: 23 maio 2016.

MCGRAW, Gary. **Software Security**: Building Security In. Upper Saddle River, NJ: Addison-Wesley, 2006.

MILNER, Steve; VICTIMS PROJECT TEAM. **Why Does This Exist?**. [S.l.], 2016. Disponível em: <https://victi.ms/about.html>. Acesso em: 19 jun. 2016.

MURTAZA, Syed Shariyar et al. Mining Trends and Patterns of Software Vulnerabilities. **Journal of Systems and Software**, [S.l.], v. 117, p. 218-228, Jul. 2016. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0164121216000790>. Acesso em: 29 out. 2016.

NATIONAL VULNERABILITY DATABASE (NVD). **NVD Frequently Asked Questions**. US, [2016?]. Disponível em: <https://nvd.nist.gov/faq>. Acesso em: 9 maio 2016.

NODE SECURITY. **Tools**. Richland, [2016?]. Disponível em: <https://nodesecurity.io/tools>. Acesso em: 19 jun. 2016.

OFTEDAL, Erlend. **Retire.js**: What you require you must also retire. [S.l., 2016?]. Disponível em: <http://retirejs.github.io/retire.js/>. Acesso em: 19 jun. 2016.

OPEN WEB APPLICATION SECURITY PROJECT (OWASP). **OWASP Dependency Check**. Bel Air, June 16, 2016. Disponível em: <https://www.owasp.org/index.php/OWASP_Dependency_Check>. Acesso em: 19 jun. 2016.

_____. **OWASP Wordpress Vulnerability Scanner Project**. Bel Air, Dec. 8, 2015. Disponível em: <https://www.owasp.org/index.php/OWASP_Wordpress_Vulnerability_Scanner_Project>. Acesso em: 19 jun. 2016.

PALAMIDA, INC. **STANDARD EDITION**. San Francisco, 2015. Disponível em: <http://www.palamida.com/files/Palamida-Standard-Edition-Datasheet.pdf>. Acesso em: 19 jun. 2016.

PCI SECURITY STANDARDS COUNCIL. **Payment Card Industry (PCI) Data Security Standard**. [S.l.], Apr. 2015. Disponível em: <https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-1.pdf>. Acesso em: 17 abr. 2016.

PLATE, Henrik; PONTA, Serena Elisa; SABETTA, Antonino. Impact Assessment for Vulnerabilities in Open-source Software Libraries. In: **2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**, Bremen, p. 411-420, Sept. 2015. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7332492>. Acesso em: 23 maio 2016.

RAEMAEKERS, Steven; VAN DEURSEN, Arie; VISSER, Joost. Semantic versioning versus breaking changes: A study of the maven repository. In: **Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)**, Victoria, BC, p. 215-224, Sept. 2014. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6975655&isnumber=6975619>. Acesso em: 27 nov. 2015.

SONATYPE INC. **Application Health Check**. Fulton, 2008. Disponível em: <http://www.sonatype.com/download-application-health-check>. Acesso em: 19 jun. 2016.

_____. **Nexus IQ Server Documentation**. Fulton, June 16, 2016. Disponível em: <http://books.sonatype.com/sonatype-clm-book/pdf/book-clm.pdf>. Acesso em: 19 jun. 2016.

SOURCECLEAR. **Better Science**: Better science on over 5 million libraries. San Francisco, [2016?]. Disponível em: <https://srcclr.com/features/science>. Acesso em: 19 jun. 2016.

THE APACHE SOFTWARE FOUNDATION (ASF). **Maven Getting Started Guide**. [S.l.], 2016a. Disponível em: <https://maven.apache.org/guides/getting-started/index.html>. Acesso em: 5 jun. 2016.

_____. **Introduction to the Dependency Mechanism**. [S.l.], 2016b. Disponível em: <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>. Acesso em: 23 out. 2016.

VERACODE. **Software Composition Analysis**. Burlington, [2016?]. Disponível em: <https://info.veracode.com/datasheet-software-composition-analysis.html>. Acesso em: 19 jun. 2016.

VERIZON ENTERPRISE SOLUTIONS. **2015 Data Breach Investigations Report**. [S.l.], Apr. 2015. Disponível em: <http://news.verizonenterprise.com/2015/04/2015-data-breach-report-info/>. Acesso em: 11 abr. 2016.

WHITESOURCE SOFTWARE. **Continuously Audit Open Source Components in Your Code**. New York, 2016. Disponível em: <http://www.whitesourcesoftware.com/open_source_scanning_software/>. Acesso em: 26 maio 2016.

WICHERS, Dave. Libraries and Application Security Part One: Known Vulnerabilities. **Contrast Security**, Palo Alto, Jan. 16, 2014. Disponível em: <https://www.contrastsecurity.com/blog/libraries-and-application-security-part-one-known-vulnerabilities>. Acesso em: 19 jun. 2016.

WILLIAMS, Jeff; DABIRSIAGHI, Arshan. The unfortunate reality of insecure libraries. **Contrast Security**, Columbia, 2014. Disponível em: <https://www.contrastsecurity.com/the-unfortunate-reality-of-insecure-libraries>. Acesso em: 1 dez. 2015.

WILLIAMS, Jeff; WICHERS, Dave. **OWASP Top-10 2013**. [S.l.], Feb. 2013. Disponível em: <https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/owasptop10/OWASP%20Top%2010%20-%202013.pdf>. Acesso em: 1 dez. 2015.

WURSTER, Glenn; VAN OORSCHOT, Paul C. The Developer is the Enemy. In: **Proceedings of the 2008 Workshop on New Security Paradigms**. Lake Tahoe, California, USA, p. 89-97, Sept. 2008. Disponível em: <http://dl.acm.org/citation.cfm?id=1595691>. Acesso em: 11 abr. 2016.

APÊNDICE A – QUADRO DE CRITÉRIOS UTILIZADOS PARA ASSOCIAÇÃO ENTRE CPES E DEPENDÊNCIAS DO MAVEN

Este quadro apresenta os critérios utilizados para associação entre CPEs e dependências do Maven. Cada linha representa uma pesquisa, sendo que [fornecedor], [produto], [versão] e [atualização] representam os componentes do CPE sendo pesquisado e a coluna indica o atributo do Maven onde tal componente é pesquisado.

GroupId	***ArtifactId***	***Version***	**Pesquisa exata?**	**Pontuação**
[fornecedor]	[produto]	[versão]-[atualização]	Sim	78
[fornecedor]	[produto]	[versão]	Sim	74
[fornecedor]	[produto]	[versão]-[atualização]*	Sim	70
[fornecedor]	[produto]	[versão]*	Sim	66
[fornecedor]	[produto]	[versão]-[atualização]	Não	62
[fornecedor]	[produto]	[versão]	Não	58
[fornecedor]	[produto]	[versão]-[atualização]*	Não	54
[fornecedor]	[produto]	[versão]*	Não	50
	[produto]	[versão]-[atualização]	Sim	46
	[produto]	[versão]	Sim	42
	[produto]	[versão]-[atualização]*	Sim	38
	[produto]	[versão]*	Sim	34
	[produto]	[versão]-[atualização]	Não	30
	[produto]	[versão]	Não	26
	[produto]	[versão]-[atualização]*	Não	22
	[produto]	[versão]*	Não	18
[produto]	core	[versão]-[atualização]	Não	14
[produto]	core	[versão]	Não	10
[produto]	core	[versão]-[atualização].*	Não	8
[produto]	core	[versão]-[atualização]*	Não	6
[produto]	core	[versão].*	Não	4
[produto]	core	[versão]*	Não	2

Fonte: Elaborado pela autora.

A pesquisa exata indica que o texto fornecido por parâmetro deve corresponder de forma completa ao texto do atributo da dependência. O “*” é um caracter curinga que representa 0 ou mais caracteres arbitrários. A pontuação é

utilizada para filtrar os resultados retornados pela pesquisa: todos os critérios são utilizados em uma única pesquisa, porém apenas as dependências com a maior pontuação retornada nos resultados são associadas ao CPE.

ANEXO A - INFORMAÇÕES AUXILIARES SOBRE AS FERRAMENTAS DE ANÁLISE DE VULNERABILIDADES EM DEPENDÊNCIAS CONSIDERADAS NO COMPARATIVO

Application Health Check: ferramenta mantida pela Sonatype que permite a verificação de binários de aplicações Java nos formatos .war, .jar, .ear, .zip, .tar.gz, dentre outros. As vulnerabilidades são obtidas de fontes públicas e proprietárias e o resultado da análise é enviado por e-mail. (SONATYPE INC, 2008).

bitHound: analisa arquivos JavaScript, TypeScript e JSX, módulos npm e componentes do bower. As vulnerabilidades das dependências são verificadas através do *Node Security Project* e o resultado da análise inclui a sinalização de incompatibilidades entre as versões de acordo com o esquema de versionamento semântico. Possui integração com repositórios Git hospedados no GitHub e no BitBucket, com os *softwares* de integração entre equipes Slack e HipChat e plataforma de integração contínua Codeship. (BITHOUND INC, 2016).

Black Duck Hub: suporta a identificação de dependências em projetos Java, C++, C# – dentre outros – e obtém as vulnerabilidades dos bancos de dados NVD e VulnDB. Além de listar as vulnerabilidades das versões dos componentes incluídas no projeto, permite pesquisar as demais versões e suas vulnerabilidades. Permite integração com a ferramenta de integração contínua Jenkins. (BLACK DUCK SOFTWARE, INC, 2015, 2016).

bundler-audit: verifica as vulnerabilidades das dependências de aplicações Ruby que utilizam Bundler para o gerenciamento das gems. O resultado da análise lista as vulnerabilidades encontradas e possíveis soluções, como a atualização para outra versão. Utiliza o banco de dados ruby-advisory-db, que obtém dados através do banco de dados OSVDB e de contribuições ao projeto. (BUNDLER-AUDIT, 2016).

Codenomicon AppCheck: utiliza o arquivo executável (.msi, .exe, .dmg, .dpkg, .jar, dentre outros) para extrair as informações de dependências da aplicação. As vulnerabilidades são obtidas através de várias fontes de dados, incluindo o banco de dados NVD. Para permitir a análise automática de projetos a ferramenta

disponibiliza API para submissão e obtenção do resultado da análise. (CODENOMICON LTD, [2016?]).

Contrast: ferramenta integrada no servidor de aplicações que monitora a execução da aplicação, obtendo informações sobre as vulnerabilidades à medida em que o código é executado. Possui suporte às linguagens Java, .NET, Node.js, C# e Visual Basic e permite integração com o IDE Eclipse para auxiliar na identificação de vulnerabilidades como *SQL Injection* e *Cross-Site Scripting*. (CONTRAST SECURITY, 2016; WICHERS, 2014).

Gemnasium: suporta projetos que utilizam gems, npm, pypi, packagist e bower para o gerenciamento de dependências. Esta ferramenta envia alertas quando novas versões de dependências são disponibilizadas, verifica o conjunto de dependências compatíveis de modo a prover atualização automática do componente e permite receber notificações por e-mail ou através dos *softwares* de integração entre equipes Slack e Campfire. As vulnerabilidades são obtidas a partir de várias fontes, incluindo as listas de e-mail *oss-security*, *Node Security Project* e o banco de dados ruby-advisory-db. (GEMNASIUM, 2015, 2016).

Nexus Lifecycle: possui todas as funcionalidades da ferramenta *Application Health Check* e integração com ferramentas utilizadas no processo de *software*, como Eclipse, Maven, Atlassian Bamboo, Jenkins, dentre outras. (SONATYPE INC, 2016).

Node Security Project: este projeto disponibiliza informações sobre vulnerabilidades nos módulos Node.js e ferramentas que verificam as dependências do projeto de acordo este banco de dados de vulnerabilidades. O resultado da análise apresenta a versão mínima que corrige a vulnerabilidade (quando disponível). Vulnerabilidades podem ser submetidas através do site. (NODE SECURITY, [2016?]).

OWASP Dependency Check: suporta projetos em Java, .NET, Ruby, Node.js, Python e C/C++ (que utilizam CMake ou autoconf). Possui como fonte de dados o NVD e permite integração com Maven, Ant e Jenkins. (OPEN WEB APPLICATION SECURITY PROJECT (OWASP), 2016).

OWASP Wordpress Vulnerability Scanner: verifica sites em Wordpress em busca de vulnerabilidades nas configurações, versão do Wordpress, *plugins* e temas. (OWASP, 2015).

Palamida: as versões Standard e Enterprise disponibilizam a verificação de vulnerabilidades em projetos que utilizam C/C++, C#, Delphi, Go, Java, Lua, PHP, Python, Ruby, dentre outras linguagens. As vulnerabilidades são obtidas através do banco de dados NVD. (PALAMIDA, INC, 2015).

Retire.js: verifica se bibliotecas Javascript e módulos Node.js utilizados por projetos *web* possuem alguma vulnerabilidade conhecida. A lista de vulnerabilidades conhecidas é mantida manualmente pelo projeto e engloba informações publicadas no *Node Security Project*, GitHub, fóruns do Google Groups, entre outros. Possui integração com as ferramentas de automação de tarefas Grunt e Gulp e extensões para Firefox e Chrome. (OFTEDAL, [2016?]).

SRC:CLR: utiliza como fonte de dados de vulnerabilidades logs de sistemas de controle de versão, ferramentas de gerenciamento de defeitos, listas de e-mails e bancos de dados de vulnerabilidades. Suporta linguagens como Java, JavaScript, Python, Ruby, ferramentas de controle de versão como GitHub e BitBucket e possui integração com ferramentas de gerenciamento de dependências, automação e integração contínua como Maven, Gradle, npm, Jenkins, Travis CI, dentre outras. O resultado da análise, além de sinalizar os componentes vulneráveis, indica se o projeto utiliza métodos vulneráveis do componente e sugere sua atualização, incluindo alerta sobre possíveis incompatibilidades. (SOURCECLEAR, [2016?]).

The Victims Project: disponibiliza um banco de dados de vulnerabilidades de componentes Java mantido pela Red Hat e *plugins* para Maven e Ant que identificam os componentes vulneráveis. (MILNER; VICTIMS PROJECT TEAM, 2016).

Veracode Software Composition Analysis: analisa arquivos binários e suporta linguagens como Java e .NET. O resultado da análise, além de listar as vulnerabilidades dos componentes, apresenta sua versão mais recente. Obtém as vulnerabilidades do banco de dados NVD. (VERACODE, [2016?]).

WhiteSource: é executada no processo de construção da aplicação (possui suporte a ferramentas como Ant, Maven, Gradle, Grunt, Atlassian Bamboo, Jenkins, dentre outras) e envia alertas quando são adicionados novos componentes com vulnerabilidades, novas vulnerabilidades e quando são disponibilizadas novas versões dos componentes. Através da ferramenta é possível verificar as dependências de cada versão dos componentes utilizados. (WHITESOURCE SOFTWARE, 2016).