Remoção Automática de Vulnerabilidades usando Análise Estática de Código Direcionada

Paulo Antunes, Ibéria Medeiros, and Nuno Neves

LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
pdfantunes@gmail.com, imedeiros@di.fc.ul.pt, nuno@di.fc.ul.pt

Resumo As aplicações web são uma presença assídua no nosso quotidiano, sendo usadas na concretização de uma variedade de serviços e empregues nos mais diversos contextos. No entanto, a correção destas aplicações pode ser comprometida pela existência de vulnerabilidades no seu código, incorrendo em consequências nefastas, nomeadamente o roubo de dados privados e a adulteração de informação. Este artigo apresenta uma solução para a deteção e remoção automática de vulnerabilidades nas aplicações web. Através da monitorização das interações com a aplicação, com ferramentas tradicionais para a descoberta de ataques, são identificados os inputs maliciosos fornecidos pelo atacante. Estes inputs são explorados para direcionar a análise estática, permitindo comprovar eventuais problemas e corrigir os erros no programa. A solução foi concretizada na ferramenta WAr̈LoCk e validada com um conjunto de aplicações vulneráveis. Os resultados experimentais demonstraram a deteção e correção de vulnerabilidades de injeção de SQL e XSS.

Keywords: vulnerabilidades · aplicações web · análise estática de código direcionada · correção de código · segurança de software

1 Introdução

A rápida evolução das tecnologias web juntamente com o acesso fácil a poder computacional levou a que vários serviços passassem a dispor de uma vertente online. Através de serviços web pode-se realizar comodamente compras de diversos produtos, contactar amigos ou aceder a contas bancárias. Todas estas funcionalidades são geralmente concretizadas através de aplicações web e, embora sejam utilizadas por milhares de milhões de utilizadores diariamente, muitas delas incorporam vulnerabilidades latentes. Se exploradas, estas falhas poderão levar a diversas consequências que incluem o roubo de dados privados, a adulteração de informação, e a inoperabilidade dos serviços, resultando em grandes prejuízos para utilizadores e para empresas. A existência destas vulnerabilidades prende-se com práticas de programação insegura aliadas à utilização de linguagens e de sistemas gestores de conteúdos (ex., WordPress, Drupal) com poucas validações [3] [10]. Por exemplo, o PHP tem limitações na validação de tipos, mas acaba por ser a linguagem mais utilizada no desenvolvimento de aplicações web.

De entre as classes de vulnerabilidade presentes em aplicações web, as de injeção de código continuam a ser as mais prevalecentes, sendo colocadas pela

OWASP no topo da lista de 2017 [11]. As vulnerabilidades de injeção de SQL (SQLI) pertencem a esta categoria, e se exploradas levam a consequências nefastas quer aos serviços como às bases de dados. Por exemplo, recentemente, um ataque de SQLI possibilitou o acesso criminoso a dados de cartões de crédito, em que os seus danos já atingiram os 300 milhões de dólares [6]. Apesar da classe de vulnerabilidade Cross-Site Scripting (XSS) aparecer em sétimo lugar pela OWASP, estima-se que estas falhas estejam presentes em cerca de dois-terços das aplicações web. Tanto que 2017 foi considerado como sendo o ano da "onda XSS", tendo havido a expetativa de um crescimento de 166% face ao ano de 2016 [8]. Este crescimento acabou por se efetivar de acordo com a Imperva [3].

As vulnerabilidades podem ser encontradas recorrendo-se a ferramentas de análise estática [1], [7], [4], [5]. A técnica que tem sido mais usada é a análise de comprometimento, que rastreia os inputs (maliciosos) ao longo do código da aplicação e verifica se estes atingem alguma função da linguagem que pode ser explorada (ex., *echo* em PHP). A precisão destas ferramentas depende do conhecimento que elas possuem sobre as classes de vulnerabilidades a procurar, bem como na implementação dos mecanismos que elas empregam na análise do código [2], que se incompleta ou incorreta pode originar falsos positivos (indicações de vulnerabilidade que são inválidas) ou falsos negativos (a não deteção de vulnerabilidades). Por outro lado, a deteção por si só é insuficiente, uma vez que o objetivo final é a remoção das vulnerabilidades, algo que poderá ser complicado de realizar por programadores pouco familiarizados com aspetos de segurança.

O artigo apresenta uma solução para a deteção e remoção automática de vulnerabilidades em aplicações web, baseada numa *análise estática de código direcionada*. Através da monitorização da aplicação em execução e das interações com ela, são identificados os inputs potencialmente maliciosos. Estes inputs são explorados para direcionar a análise estática, permitindo comprovar eventuais vulnerabilidades e removê-las automaticamente através da inserção de *fragmentos de código* na aplicação, gerando uma versão nova e mais segura da aplicação. A solução foi concretizada numa plataforma que inclui a ferramenta *Web Application Live Correction* (WArLoCk) que analisa aplicações PHP, tendo sido validada com um conjunto de aplicações que contêm vulnerabilidades SQLI e XSS. Foram detetadas e corrigidas 174 vulnerabilidades, onde 2 delas ainda não tinham sido descobertas (i.e., *zero-days*) e não foram observados falsos positivos.

As contribuições deste artigo são: (1) uma solução que permite a deteção e correção de vulnerabilidades em tempo real de aplicações em execução, usando análise estática de código direcionada; (2) a ferramenta WArLoCk que implementa solução proposta; (3) uma avaliação experimental com 5 aplicações web.

2 Vulnerabilidades de SQLI e de XSS

Esta secção apresenta resumidamente as vulnerabilidades de SQLI e de XSS. Uma vulnerabilidade de injeção permite a um adversário fornecer código malicioso a uma aplicação, por exemplo através de formulários, na esperança que este seja usado de forma desprotegida, em alguma função da linguagem ou numa

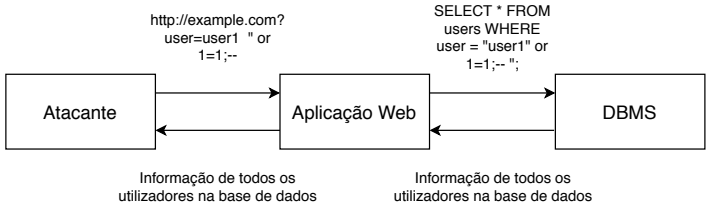

Figura 1. Exemplo de injeção de SQL

parte do programa que seja sensível. As vulnerabilidades de SQLI são um caso particular desta classe de vulnerabilidades, uma vez que permitem que o código malicioso seja inserido em comandos SQL para serem usados numa base de dados contactada pela aplicação. Assim sendo, um ataque é bem sucedido quando os dados maliciosos são empregues sem qualquer sanitização em funções específicas do programa (ex., *mysqli_query*) que realizam pedidos a uma base de dados. A Figura 1 exemplifica um ataque deste tipo, onde o atacante consegue inserir um código (`user1" or 1=1;--`) num pedido formulado pela aplicação à base de dados (`SELECT * FROM users WHERE user = "user1" or 1=1;--`), para extrair a informação sobre todos os utilizadores.

Uma vulnerabilidade de XSS ocorre quando um atacante injeta código malicioso (um script) para mais tarde ser executado no browser de uma vítima. Apesar de existirem três variantes desta falha: refletida, persistente e DOM (Document Object Model); apresentamos somente a primeira. Nesta variante, a aplicação web recebe um script (ex., código JavaScript) que depois inclui no código HTML de uma página web. Esta página ao ser recebida e processada pelo browser da vítima, causa a execução do script levando a cabo alguma ação maliciosa.

3 Arquitetura da Solução Proposta

Esta secção apresenta a visão geral da solução proposta, bem como as componentes que a revestem.

3.1 Visão Geral

A solução tem por objetivo detetar e remover vulnerabilidades em aplicações web em execução (i.e., em tempo real), usando uma *análise estática de código direcionada*. Para o efeito, a solução propõe a monitorização das interações com a aplicação em execução, por forma a identificar os inputs maliciosos que lhe são dirigidos. Em seguida, utiliza a informação recolhida para focar a análise de comprometimento, identificando no código da aplicação as variáveis que recebem os inputs observados e realizando a análise estática de código a partir daí. A remoção das vulnerabilidades é conseguida através da inserção de *fragmentos de código* no programa da aplicação, que passarão a ser responsáveis pela validação dos inputs, impedindo os seus efeitos perniciosos.

A análise estática proposta, ao invés de analisar todo o código da aplicação, é direcionada à parte do código que envolve as variáveis que recebem os inputs

inseridos, i.e., os pontos de entrada e variáveis dependentes destas. Neste sentido, consegue-se potencialmente um processamento mais célere e preciso, uma vez que se restringe a análise aos pontos de entrada assinalados. Por acréscimo, consegue-se limitar os caminhos de execução a serem estudados, focando-se apenas naqueles que se iniciam nos pontos de entrada selecionados. Por fim, a remoção de vulnerabilidades apenas tem de ocorrer nos caminhos onde a vulnerabilidade se manifesta, evitando alterações desnecessárias noutras zonas do programa.

A arquitetura da solução é apresentada na Figura 2. Esta proporciona a possibilidade de recolher informação sobre inputs que sejam fornecidos, e de determinar se estes estão na base ou não de um ataque. Se existir uma vulnerabilidade no código que possa ser explorada por esse ataque, então esta falta começa por ser localizada e depois será corrigida em tempo de execução. Os principais componentes que compõem a arquitetura são:

- *Injetor de Inputs:* permite enviar dados (inputs) para a aplicação, os quais podem ser injetados por um cliente através de um browser, por exemplo, ou por recurso a um *fuzzer*;
- *Intercetor de Inputs:* interceta os inputs injetados e determina se estes são ou não maliciosos, analisando os seus conteúdos com base num conjunto de regras. Dependendo da sua concretização, este pode deixar passar sempre os inputs para a aplicação ou poderá apenas fazê-lo caso estes sejam considerados benignos;
- *Aplicação Web:* a aplicação web em produção (execução) que receberá os inputs fornecidos pelo utilizador e executará algum serviço;
- *Armazenamento de Dados:* sistemas de armazenamento de dados que a aplicação pode interagir, tais como uma base de dados e/ou um sistema de ficheiros;
- *Monitor:* coordena a interação entre a descoberta de vulnerabilidades e os inputs injetados e intercetados, e regista toda a atividade realizada;
- *Detetor de Vulnerabilidades:* executa a análise estática de código direcionada, com base nos inputs maliciosos capturados pelo intercetor de inputs. Identifica os pontos de entrada da aplicação requeridos pela análise, a informação necessária para a remoção de vulnerabilidades, e confirma se o alerta emitido pelo intercetor de inputs é um verdadeiro ou falso positivo;
- *Corretor de Vulnerabilidades:* corrige automaticamente as vulnerabilidades presentes no código, com base nos resultados da análise realizada pelo detetor de vulnerabilidades. Produz uma versão nova e mais segura da aplicação web, que estará pronta para substituir a que atualmente se encontra em execução.

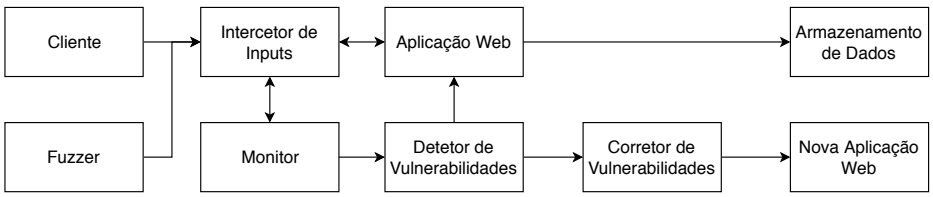

Figura 2. Visão geral da arquitetura da solução proposta.

Nas secções seguintes são apresentados em maior detalhe os componentes principais da solução, nomeadamente o *intercetor de inputs*, o *detetor de vulnerabilidades*, o *corretor de vulnerabilidades* e o *monitor*.

3.2 Intercetor de Inputs

O intercetor de inputs captura os pedidos com os dados injetados pelo cliente da aplicação web. Um pedido inclui também o URL e o nome dos parâmetros que recebem estes dados. A estrutura de um pedido é `<url>?<param>=<value>&...&<param>=<value>`, onde `<url>` representa o URL base da aplicação web e pode conter o nome do ficheiro a quem se dirige os dados injetados; `<param>` os parâmetros; e `<value>` os dados atribuídos aos parâmetros. Os parâmetros são os pontos de entrada da aplicação e estão definidos no código desta, por exemplo recebendo valores que foram preenchidos num formulário HTML.

Este componente pode ser visto como um proxy que recebe todos os pedidos que são enviados para a aplicação, com a capacidade de: (1) determinar se os dados são maliciosos ou benignos, usando para tal um conjunto de regras que identificam padrões de atividade maliciosa de ataques, como por exemplo a presença do caráter $'$ para SQLI e do caráter $<$ para XSS; e (2) gerar alertas caso seja identificada atividade maliciosa, sempre que os dados contidos num pedido coincidirem com uma ou mais regras.

O alerta contem uma mensagem indicativa de atividade maliciosa e o pedido em análise. O alerta é então transmitido para o *monitor* para que este desencadeie a análise estática pelo *detetor de vulnerabilidades* e registe a atividade (ver secção seguinte). Caso contrário, o pedido é considerado benigno e é reencaminhado para a aplicação web, prosseguindo com a execução normal.

Uma vez que o intercetor de inputs não tem qualquer contexto sobre a aplicação web, este poderá gerar falsos alarmes, sendo que a indicação de existência de um ataque não implica que haja necessariamente uma vulnerabilidade na aplicação. Os falsos positivos serão mitigados pela análise estática efetuada pelo detetor de vulnerabilidades e posterior gestão de inputs realizada pelo monitor.

É de notar que a forma de operar que foi descrita, permite que este componente seja baseado numa *Web Application Firewall (WAF)*, como o ModSecurity [9]. Por esta razão, utilizamos este software no nosso protótipo.

3.3 Monitor

Este componente gere os resultados do intercetor de inputs e do detetor de vulnerabilidades, interligando-os de acordo com os resultados recebidos. O monitor tem as funcionalidades de *gerir alertas* e *ativar a pesquisa de vulnerabilidades*.

Gerir alertas. Os alertas provenientes do intercetor de inputs são analisados pelo monitor para verificar se estes correspondem a ataques que tentam explorar vulnerabilidades. O alerta pode não ser relativo a uma vulnerabilidade, mas sim a tráfego anormal (ex., inserir o IP como input), sendo neste caso somente registada a ocorrência da atividade. Por outro lado, se o alerta descrever uma

tentativa de exploração de uma vulnerabilidade web, o monitor ativa a pesquisa de vulnerabilidades a ser efetuada pelo detetor de vulnerabilidades, de acordo com os parâmetros contidos no pedido (i.e., os `<param>`). Também, é registada a ocorrência como sendo uma tentativa de exploração de uma vulnerabilidade.

Como foi referido anteriormente, o intercetor de inputs é isento do contexto da aplicação, o que significa que este pode gerar falsos positivos, ou seja, emitir alertas sobre ataques que tentam explorar vulnerabilidades não existentes. Para mitigar os falsos positivos do intercetor de inputs e evitar ativar desnecessariamente o detetor de vulnerabilidades, o monitor guarda o resultado do processamento do alerta, mantendo informação se existe ou não uma vulnerabilidade na aplicação. Na análise de alertas seguintes, o monitor, antes de entregar o pedido ao detetor de vulnerabilidades, verifica se já foi processado um pedido com URL e parâmetros iguais àquele em análise. Em tal situação, o pedido é reencaminhado para a aplicação porque este já foi analisado anteriormente, implicando que: (1) a aplicação já se encontra protegida porque foram detetadas e corrigidas vulnerabilidades; ou (2) o pedido é um falso positivo do intercetor de inputs porque não foram encontradas vulnerabilidades no código da aplicação.

Ativar a pesquisa de vulnerabilidades. Para configurar o detetor de vulnerabilidades, são extraídos do pedido os parâmetros e o nome do ficheiro da aplicação que manuseia estes parâmetros. Os parâmetros servirão para identificar os pontos de entrada no ficheiro, para depois dar início à análise deste. Por exemplo, do pedido `http://exemplo.com/login.php?user=user"or 1=1;-- &pass =foo` são extraídos os parâmetros `user` e `pass` e o nome do ficheiro `login.php`.

3.4 Detetor de Vulnerabilidades

O detetor de vulnerabilidades é responsável por efetuar uma análise estática direcionada sobre a parte do código que seria o alvo do ataque que foi previamente observado. O detetor configura os vetores de pontos de entrada (ex., $_GET, $_POST) com os parâmetros recebidos, obtendo um conjunto de possíveis pontos de entrada (ex., $_GET['user'], $_POST['user'], $_GET['pass'], $_POST['pass']) e inicia a análise de comprometimento no ficheiro.

A análise assinala como comprometidos os pontos de entrada que pertencem ao referido conjunto e que efetivamente se encontram definidos no ficheiro, rastreia-os ao longo do código, bem como as variáveis que derivam deles, até que algum seja usado num parâmetro de alguma função sensível (ex., a função *echo* para XSS). No entanto, se ao longo do caminho de execução algum dos pontos de entrada e/ou os seus dependentes foram protegidos por funções de sanitização (ex., *htmlentities* para XSS), estes deixam de estar comprometidos. Terminada a análise, o detetor de vulnerabilidades retorna a lista das vulnerabilidades encontradas. Cada vulnerabilidade é caraterizada pelo caminho de execução, i.e., o fluxo de dados desde os pontos de entrada até às funções associadas às vulnerabilidades, as variáveis comprometidas pelos pontos de entrada e as linhas do código do fluxo. Esta informação é posteriormente fornecida ao corretor de forma a encontrar uma proteção que adequadamente impeça o ataque.

Uma vez que é empregue uma análise estática direcionada, a análise de comprometimento segue apenas os caminhos que seriam explorados pelos inputs do pedido. Desta maneira, evita-se uma análise que trate genericamente de toda a aplicação, que recolheria *todos* os pontos de entrada e caminhos possíveis, e que teria necessariamente de efetuar simplificações para evitar uma explosão dos testes que deveriam de ser investigados. Esta abordagem permite assim reduzir os falsos positivos e também torna o exame do código mais eficiente.

No final da análise o detetor tem de: (1) detetar qualquer problema no código e ordenar a sua eliminação por parte do corretor de vulnerabilidades; (2) confirmar os alertas emitidos pelo intercetor como sendo falsos ou verdadeiros positivos e enviar esta informação ao monitor para que possa ser associada ao pedido.

3.5 Corretor de Vulnerabilidades

Este módulo utiliza os resultados e as indicações da análise estática para corrigir o programa através da inserção de *fragmentos de código* que empregam funções de sanitização. O corretor de vulnerabilidades itera sobre as vulnerabilidades detetadas, usando os detalhes destas (classe de vulnerabilidades, linhas de código e variáveis comprometidas), de forma a identificar quais as variáveis a serem sanitizadas e as linhas do código a alterar. Este pré-processamento é importante porque as correções frequentemente não podem ser aplicadas na linha na qual é indicada a vulnerabilidade. Por exemplo, com SQLI a correção deverá ser efetuada antes da formação do comando que depois é utilizado na função que transmite o pedido à base de dados. No caso de XSS, a alteração pode ser aplicada na própria função que recebe a variável comprometida.

A execução do corretor de vulnerabilidades resulta numa versão mais segura da aplicação em questão. Uma cópia do ficheiro original é sempre mantida por uma questão de precaução e de referência, permitindo aos programadores compreender o problema.

3.6 Exemplo de Deteção e Correção de Vulnerabilidades

Para ilustrar a deteção e correção de código consideremos o seguinte excerto de código do ficheiro `test.php`, que contem uma vulnerabilidade de SQLI. O input da linha 1 é inserido na instrução SQL sem qualquer sanitização e que depois é executada (linhas 2 e 3).

```
1  $id=$_GET['id'];
2  $query="SELECT first_name, last_name FROM users WHERE user_id = '$id';";
3  $result=mysqli_query($query);
```

Consideremos também que um atacante realiza o seguinte ataque de SQLI, injetando código SQL no parâmetro `id`, formando o seguinte pedido malicioso:

```
www.exemplo.com/test.php?id=a' UNION SELECT "text1","text2";--
-&Submit=Submit
```

Este pedido é intercetado pelo intercetor de inputs e analisado com base num conjunto de regras do ModSecurity, as quais detetam o ataque gerando o evento

referente a este e de acordo com as regras que o detetaram. Seguidamente, o intercetor de inputs gera o alerta seguinte com base nos eventos e no pedido que intercetou, incluindo uma `[tag]` indicativa do incidente detetado. As *tags* são geradas a partir do conhecimento que o ModSecurity tem sobre os inputs maliciosos (e ataques). Assim sendo, quão mais completo e atualizado for esse conhecimento, melhor será a capacidade de se categorizar ataques mais sofisticados e subtis. Posteriormente o alerta é enviado para o monitor.

```
[tag "attack-sqli"] [uri "www.exemplo.com/test.php?
id=a' UNION SELECT "text1","text2";-- -&Submit=Submit]
```

O monitor, ao receber o alerta, consulta a `[tag]` de forma a interpretar o alerta e a natureza da possível vulnerabilidade, e de seguida extrai do URI enviado, o parâmetro `id`, que corresponde ao ponto de entrada na aplicação, e o nome do ficheiro `test.php`. Estes dados são usados para configurar e ordenar a execução do detetor de vulnerabilidades de forma a considerar apenas estes dados durante a análise estática. Ao analisar o ficheiro `test.php` através dos pontos de entrada especificados, a variável `$id` é considerada comprometida (i.e., *tainted*) por o vetor $_GET conter o ponto de entrada `id` (i.e., $_GET['id']). Esta por sua vez é usada para formular uma instrução SQL e ser executada pela função `mysqli_query`, sendo assim detetada a vulnerabilidade de SQLI.

Por fim o detetor compõe a lista de vulnerabilidades, indicando para cada vulnerabilidade encontrada a seguinte informação.

```
Vulnerabilidade: SQLI [1, 2, 3]
Variáveis iniciais: $id: 1
variáveis dependentes: $query: 2
função: mysqli_query: 3
```

O detetor de vulnerabilidades ativa o corretor de vulnerabilidades, passando-lhe a lista de vulnerabilidades. O corretor identifica o tipo de vulnerabilidade e insere o *fragmento de código* referente a ela, o mais próximo possível da função. Será então inserido o fragmento `san_sqli(1,$id)` para sanitizar a variável `$id` (linha 3). Esta versão resultante da aplicação tem a query sanitizada de forma a prevenir injeção de código, removendo assim a vulnerabilidade de SQLI.

```
1 $id=$_GET['id'];
2 $query="SELECT first_name, last_name FROM users
3         WHERE user_id = '.san_sqli(1,$id).';";
4 $result=mysqli_query($query);
```

4 Implementação do WArLoCk

A arquitetura proposta foi concretizada numa plataforma, para a qual foram desenvolvidos em Java os componentes intercetor de inputs e monitor, e em PHP os componentes detetor e corretor de vulnerabilidades. A plataforma integra o ModSecurity [9] como gerador de alertas, e interage com o intercetor de inputs. O ModSecurity é uma WAF que permite monitorização, bem como logging e controlo de acesso. A versão atual da ferramenta de análise de código WArLoCk

deteta faltas de SQLI e de XSS em aplicações desenvolvidas em PHP, e corrige automaticamente vulnerabilidades de SQLI.

O intercetor de inputs funciona como uma proxy dupla. Por um lado, ele utiliza o ModSecurity como gerador de alertas. Por outro lado, através de um módulo desenvolvido em Java, ele captura os pedidos enviados pelo cliente/fuzzer e reencaminha-os para o monitor ou para a aplicação web, dependendo se são gerados alertas, respetivamente. Em caso afirmativo, os pedidos capturados também são enviados para o monitor.

O monitor também foi desenvolvido em Java. Este examina o teor dos alertas que lhe chegam para apurar se correspondem a uma possível tentativa de exploração de uma vulnerabilidade do código, parametrizando o detetor de vulnerabilidades, caso assim se verifique.

O detetor de vulnerabilidades está implementado em PHP e utiliza a função *token_get_all* para quebrar o código em tokens, e depois gerar a *Abstract Syntax Tree* (AST) a ser utilizada na análise estática. A análise é parametrizável e permite que sejam indicados quais os parâmetros que deverão ser considerados comprometidos. Adicionalmente, a análise estende-se para outros ficheiros de código incluídos também no ficheiro alvo. No final da execução, o componente produz duas listas, uma que fornece informação sobre todas as vulnerabilidades encontradas e outra que indica o estado das variáveis que foram processadas.

O corretor de vulnerabilidades, implementado em PHP, faz uso de ambas as listas para efetuar as correções. A correção do código é aplicada o mais perto possível das funções que podem ser exploradas por inputs maliciosos. O código da aplicação, incluindo as correções, é reescrito para um novo ficheiro no qual é também incluída uma biblioteca que possui as funções de sanitização, i.e., os *fragmentos de código*, para eliminar as vulnerabilidades.

5 Avaliação

O objetivo da avaliação experimental é testar e verificar a eficácia da plataforma na deteção e correção de vulnerabilidades de SQLI e XSS em aplicações escritas em PHP. Neste sentido deu-se resposta às seguintes questões: (1) *A plataforma é capaz de detetar ambas as classes de vulnerabilidades?* (2) *A plataforma é capaz de corrigir vulnerabilidades SQLI?* (3) *O detetor de vulnerabilidades tem a capacidade de identificar falsos positivos no intercetor de inputs?*

A avaliação foi realizada com 5 serviços web num ambiente que emula uma potencial situação de produção. Usaram-se os serviços: *DVWA* é uma aplicação muito simples, que processa os valores introduzidos num conjunto de formulários, e que é propositadamente vulnerável a SQLI; *ZeroCMS* é uma plataforma para partilhar artigos entre utilizadores; *AddressBook* é uma aplicação que permite aos utilizadores manter uma lista de contactos organizada por grupos; *WebChess* é uma aplicação que implementa um jogo xadrez online para dois jogadores; *refbase* é uma aplicação que disponibiliza uma base de dados com referências bibliográficas em vários formatos.

Antes de realizar as experiências, procedeu-se a uma auditoria manual prévia do código das aplicações por forma a identificar as vulnerabilidades, determi-

nar os pontos de entrada e quais destes podem ser usados para comprometer a aplicação, e encontrar um ataque para as explorar. Da auditoria apurou-se a existência de 184 vulnerabilidades e definiu-se 145 ataques.

De realçar que todos os pedidos foram feitos de forma a que fosse gerado um alerta por parte do ModSecurity (integrado no intercetor de inputs), embora este método resulte na criação de falsos positivos pelo intercetor de input. O objetivo é que estes sejam colmatados pelo detetor de vulnerabilidades, que confirmará ou não a existência de qualquer falta no código (questão 3). Também, é de realçar que o âmbito dos testes realizados pretenderam testar a viabilidade funcional da ferramenta desenvolvida, não visando a performance da aplicação em execução. Contudo, o impacto da deteção e correção não será constante, uma vez que a análise só será despoletada nos casos em que é detetada atividade maliciosa. É também de referir que o monitor mantém um registo dos inputs que já foram sujeitos a análise estática evitando assim a repetição desnecessária da análise.

5.1 Deteção e Correção de Vulnerabilidades

Após a execução dos 145 ataques, a plataforma detetou 174 vulnerabilidades (das 184 manualmente identificadas), em que 96 delas eram do tipo SQLI e 78 de XSS. Mais tarde confirmámos que 2 duas das vulnerabilidade eram previamente desconhecidas (i.e., *zero-days*). Não foram reportados falsos positivos, mas existiram 10 falsos negativos causados por insuficiências na versão atual do analisador estático, nomeadamente no processamento de alguns tipos de cláusulas condicionais. Destas vulnerabilidades todas as de SQLI foram corrigidas.

A Tabela 1 resume os resultados da avaliação. As 5 aplicações correspondem a 800 ficheiros e a 105 mil linhas de código (colunas 2 e 3). Os 145 ataques (coluna 4) causaram a análise de 145 ficheiros (coluna 5) com cerca de 40 mil linhas de código (coluna 6). As 96 vulnerabilidades de SQLI foram encontradas em 23 ficheiros (última coluna). A correção dos erros resultou assim em 23 ficheiros novos. A aplicação WebChess foi a mais vulnerável, em ambas as classes de vulnerabilidades, enquanto que a aplicação refbase foi a menos vulnerável, com 4 vulnerabilidades. Dado os resultados, as questões 1 e 2 têm resposta positiva.

5.2 Deteção de Falsos Positivos

Considerando os 145 ataques realizados, foi avaliada a geração de alarmes do intercetor de inputs e a capacidade de descoberta de falhas no código pelo detetor de vulnerabilidades. A Tabela 2 mostra os resultados desta análise, verificando-se

Tabela 1. Tabela com informação por cada aplicação web.

Aplicação web	**Total Fich.**	**LoC**	**Ataques**	**Fich. Anal.**	**LoC Anal.**	**Total Vuln.**	**SQLI**	**XSS**	**FP**	**FN**	**Fich. Corr.**
DVWA	343	19.142	1	1	24	1	1	0	0	0	1
ZeroCMS	29	1.001	14	14	736	18	13	5	0	0	6
AddressBook	238	28.526	63	63	8280	29	7	22	0	9	4
WebChess	28	3.634	28	28	3634	122	75	47	0	1	12
refbase	156	52.200	39	39	25549	4	0	4	0	0	0
Total	**797**	**104.503**	**145**	**145**	**38.223**	**174**	**96**	**78**	**0**	**10**	**23**

Tabela 2. Tabela de ficheiros analisados.

	Intercetor inputs	Detetor de vulnerabilidades		Corretor de Vulnerabilidades
Verdadeiro Positivo	49	Deteção	39	23
		Falha	10	0
Falso Positivo	96	Deteção	0	0
		Falha	96	0

que dos 145 alertas gerados pelo ModSecurity, integrado no intercetor de inputs, efetivamente 49 deles correspondem a ataques que exploram vulnerabilidades nas aplicações. No entanto, 96 alertas são falsos positivos, por as aplicações estarem protegidas destes ataques. O detetor de vulnerabilidades também processou os 145 alertas, e corretamente identificou os 96 falsos positivos (Falha na terceira linha da tabela). De entre os 49 verdadeiros positivos, o detetor de vulnerabilidades identificou 39 como correspondendo a vulnerabilidades (Deteção na segunda linha da tabela), mas enganou-se nos restantes 10 casos (Falha na segunda linha da tabela). Assim sendo, a questão 3 tem resposta positiva.

6 Trabalho Relacionado

Esta secção não pretende exaustivamente discutir os diversos trabalhos existentes na literatura sobre a deteção de vulnerabilidades por análise estática. Portanto, apresenta uma seleção de trabalhos relacionados com a solução proposta.

Jovanic et al. apresentaram o Pixy, uma ferramenta que analisa código PHP para detetar vulnerabilidades de XSS e SQLI [4]. Na análise do código o Pixy tem em conta os *aliases* das variáveis, por forma a realizar uma análise mais precisa. No entanto, a ferramenta produz bastantes falsos positivos.

Dahse et al. propuseram o RIPS, que efetua uma análise estática dos fluxo de dados, de forma inter e intra procedimental [1]. O RIPS revela uma percentagem de 72% verdadeiros positivos e uma estimativa de 24% falsos negativos e de 28% falsos positivos, sendo que a principal causa dos seus falsos negativos e positivos é a análise de fluxo de dados insensível a campos e a caminhos.

O phpSAFE é uma ferramenta apresentada por Nunes et. al para detetar SQLI e XSS [7]. Tal como as outras ferramentas, o phpSAFE cria uma AST para ser utilizada durante a fase de análise para que sejam seguidos os caminhos que possam ser comprometidos por inputs maliciosos até que estes cheguem a alguma função suscetível de ser explorada.

O WAP é uma ferramenta proposta por Medeiros et al. que permite, para além da deteção de vulnerabilidades, a correção automática de código [5]. O WAP é constituído por três módulos, um analisador de código que analisa o código para identificar vulnerabilidades candidatas, para depois recorrendo a técnicas de mineração de dados predizer se estas são falsos positivos ou não. As vulnerabilidades efetivas são corrigidas pela inserção de *fixes* no código.

Apesar das ferramentas acima usarem análise estática, nenhuma delas a utiliza de forma direcionada, nem partindo da monitorização de aplicações em execução. Somente o WAP faz correção do código, enquanto as restantes só efetuam deteção.

7 Conclusão

Este artigo apresentou uma arquitetura para a deteção e correção automática de vulnerabilidades em aplicações web, recorrendo a uma análise estática direcionada. Monitorizando a aplicação em execução e as interações com ela, são identificados inputs maliciosos que servirão para direcionar a análise estática na pesquisa de vulnerabilidades, para posterior correção. A arquitetura foi implementada na ferramenta WArLoCk e avaliada, mostrando-se eficaz na deteção e correção de vulnerabilidades em 5 aplicações web.

Agradecimentos Este trabalho foi parcialmente suportado pelos fundos nacionais através da Fundação para a Ciência e a Tecnologia (FCT) com referência ao projeto SEAL (2/SAICT/2017 c.no 029058) e à Unidade de Investigação LASIGE (UID/-CEC/00408/2013).

Referências

1. Dahse, J., Holz, T.: Simulation of built-in PHP features for precise static code analysis. In: Proceedings of the 21st Network and Distributed System Security Symposium (Feb 2014)
2. Dahse, J., Holz, T.: Experience report: An empirical study of PHP security mechanism usage. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 60–70 (Jul 2015)
3. Imperva: The state of web application vulnerabilities in 2017 (Dec 2017), https://www.imperva.com/blog/2017/12/the-state-of-web-application-vulnerabilities-in-2017/
4. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security. pp. 27–36 (Jun 2006)
5. Medeiros, I., Neves, N.F., Correia, M.: Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Transactions on Reliability **65**(1), 54–69 (March 2016)
6. Naked Security by SOPHOS: Hackers sentenced for sql injections that cost $300 million (Jun 2018), https://nakedsecurity.sophos.com/2018/02/19/hackers-sentenced-for-sql-injections-that-cost-300-million/
7. Nunes, P., Fonseca, J., Vieira, M.: phpSAFE: A security analysis tool for OOP web application plugins. In: Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (Jun 2015)
8. Sink: XSS attacks: The next wave (Jun 2017), https://snyk.io/blog/xss-attacks-the-next-wave/
9. Trustwave SpiderLabs: ModSecurity - Open Source Web Application Firewall, http://www.modsecurity.org
10. W3Techs: World wide web technology surveys (Jun 2018), https://w3techs.com
11. Williams, J., Wichers, D.: OWASP Top 10 2017 – the ten most critical web application security risks (2017)