Comparativos de ferramentas para SQL Injection em Aplicações Web

Gislaine M. Coelho[1], Daves M. Silva Martins[1]

Instituto Federal do Sudeste de Minas Gerais –
Juiz de Fora – MG – Brasil

gislainemartins1990@gmail.com, daves.martins@ifsudestemg.edu.br

Abstract. *Over the years as web applications have become increasingly accessible and sought after not only as a form of disclosure but also targeting new ways to secure profits for companies with new types of services like: e-commerce, e-government . However, these services that use web applications are vulnerable to attack. This article presents one of the easiest techniques for attacks on web systems, known as SQL Injection [9]. A comparison was made between two SQL Injection tools: SQLMap and TheMole. For a comparative analysis of tools, it is necessary: attack via GET and POST method, Blind SQL injection, usability and installation.*

Resumo. *Ao longo dos anos, as aplicações web vêm se tornando cada vez mais acessíveis e mais procuradas não só como uma forma de divulgação, mas também visando novas maneiras de garantir lucros para empresas com novos tipos de serviços como: e-commerce, e-government. No entanto, esses serviços que utilizam aplicações web estão vulneráveis a sofrer ataques. Esses ataques podem causar perda de informações e roubo de dados. Esse artigo apresenta uma das técnicas mais utilizadas para ataques em sistemas web, conhecida como SQL Injection [9]. Foi realizado um comparativo entre duas ferramentas de SQL Injection: SQLMap e TheMole. Para a análise comparativa das ferramentas foram utilizadas critérios tais como: ataque via método GET e POST, Blind SQL injection, usabilidade e instalações.*

1. Introdução

Atualmente, há uma crescente dependência de aplicativos da web. Quase tudo é armazenado, disponível ou comercializado na web. As aplicações web estão muito presentes no nosso modo de vida e, em nossa economia, é tão importante que as tornam um alvo natural para mentes mal-intencionadas que desejam explorar vulnerabilidades. [23]

Diante desse cenário, é preciso se prevenir e verificar as vulnerabilidades presentes em uma aplicação web, para isso faz-se necessário o uso de algumas ferramentas. Mas qual é a melhor ferramenta para avaliação das vulnerabilidades? [32].

O *SQL Injection*, segundo Bandhakavi [6], é uma das principais vulnerabilidades presentes nas aplicações. Isto se deve principalmente, pela falta de cuidado e de conhecimento durante o desenvolvimento. O ataque pode ser facilmente aplicado em qualquer tipo de aplicação, sem necessidade de nenhum conhecimento avançado.

Este artigo tem por objetivo avaliar duas ferramentas que exploram vulnerabilidades de *SQL injection*, o *sqlMap* e o *TheMole*, duas das ferramentas gratuitas mais utilizadas para esse tipo de ataque[28].

O presente artigo está organizado da seguinte forma: na Seção 2, é abordado o referencial teórico, no qual são explicitados maiores detalhes sobre as vulnerabilidades, os conceitos de *SQL injection*, tipos de ataques existentes e apresentação de alguns métodos de prevenção. Na Seção 3, é apresentada a metodologia utilizada na

comparação das ferramentas de *SQL injetcion*, seguida pela demonstração dos resultados obtidos na comparação, e por fim as considerações finais a respeito do artigo.

2. Referencial Teórico

2.1 Vulnerabilidades

As vulnerabilidades em sistemas web são muito comuns, sejam esses sistemas feitos em qualquer linguagem e utilizando qualquer tecnologia. O problema dessas vulnerabilidades é que elas podem ser exploradas por qualquer um que tenha por objetivo, tirar algum proveito. Essas falhas de segurança podem resultar em prejuízos financeiros, quando vazam dados críticos ou quando serviços ficam inoperantes. [10]

O aumento do número de vulnerabilidades nos sistemas é um dos fatores que contribuem para impulsionar os incidentes referentes à segurança das informações. Isso ocorre, pois, muitas vezes, uma vulnerabilidade de segurança é causada por uma sequência de erros que perduram desde a análise ao desenvolvimento do projeto. Este cenário se agrava quando não é adotado um ciclo de desenvolvimento de software seguro. [18]

Segundo pesquisas realizadas pelo Instituto Ponemon [13], aproximadamente 80% das empresas disseram que seus dados ficaram mais vulneráveis a ataques. Observando a figura 1, é possível perceber que a pesquisa relata que 54% dos participantes da entrevista, afirmaram que a forma de ataque mais comum nos últimos 12 meses tem sido o *SQL Injection*, seguido pelo *Cross Site Scripting* (23% dos casos) e o *Cross Site Request Forgery* (XSS).

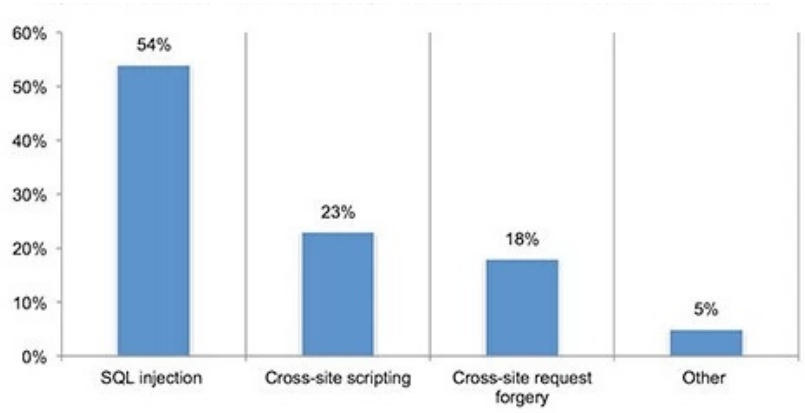

Figura 1. Aumento da vulnerabilidade SQL injection em 2015. [13]

As pesquisas realizadas por: Cheng Huang, JiaYong Liu, Yong Fang e Zheng Zuo [30] mostram que a vulnerabilidade mais comum entre os sites dinâmicos (aplicações web) da China é o *SQL injection* (SQLi). A Figura 2 apresenta uma tabela que mostra que o percentual de ataques de *SQL injection* vem aumentando ao longo dos anos, enquanto outros tipos de ataques estão diminuindo.

	2012	2013	2014	2015
SQL injection	853 (18.82%)	2,099 (25.05%)↑	15,792 (50.28%)↑	5,742 (44.87%)
XSS	770 (16.99%)	1,032 (12.32%)	1,492 (4.75%)	283 (2.21%)
Logic error	561 (12.38%)	1,049 (12.52%)↑	1,260 (4.01%)	202 (1.58%)
Sensitive data exposure	559 (12.33%)	607 (7.25%)	1,909 (6.08%)	1,403 (10.96%)↑
Broken access control	403 (8.89%)	869 (10.37%)↑	1,750 (5.57%)	515 (4.02%)
Command injection	340 (7.5%)	782 (9.33%)↑	1,624 (5.17%)	564 (4.41%)
Misconfiguration	277 (6.11%)	365 (4.36%)	1,295 (4.12%)	354 (2.77%)
Hack event	180 (3.97%)	293 (3.5%)	2,201 (7.01%)↑	1,734 (13.55%)↑
Weak password	169 (3.73%)	479 (5.72%)↑	2,046 (6.51%)↑	733 (5.73%)
File upload	168 (3.71%)	314 (3.75%)↑	562 (1.79%)	117 (0.91%)
Path traversal	111 (2.45%)	163 (1.95%)	343 (1.09%)	64 (0.5%)
Unvalidated redirects	78 (1.72%)	56 (0.67%)	136 (0.43%)	34 (0.27%)
CSRF	55 (1.21%)	150 (1.79%)↑	50 (0.16%)	16 (0.13%)
File include	8 (0.18%)	42 (0.5%)↑	371 (1.18%)↑	111 (0.87%)
Other	0 (0.0%)	78 (0.93%)↑	575 (1.83%)↑	924 (7.22%)↑

Figura 2. Aumento da vulnerabilidade *SQL injection* de 2012 até 2015 em organizações da China. [30]

O OWASP Top 10 [9] é uma lista com as principais vulnerabilidades em aplicações web. Tem como principal objetivo educar quem desenha, arquiteta e programa aplicações web (embora também existam para outros tipos de aplicações ou sistemas).

OWASP Top 10 – 2013 (Previous)	OWASP Top 10 – 2017 (New)
A1 – Injection	A1 – Injection
A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References - Merged with A7	A4 – Broken Access Control (Original category in 2003/2004)
A5 – Security Misconfiguration	A5 – Security Misconfiguration
A6 – Sensitive Data Exposure	A6 – Sensitive Data Exposure
A7 – Missing Function Level Access Control - Merged with A4	A7 – Insufficient Attack Protection (NEW)
A8 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards - Dropped	A10 – Underprotected APIs (NEW)

Figura 3. Atualização das principais vulnerabilidades apresentadas OWASP top 10. [9]

A figura 3 mostra as vulnerabilidades com atualizações de 2013 em relação a 2017. O Top 10 é atualizado utilizando dados de pesquisas e estatísticas sobre os principais ataques ocorridos pelo mundo.

2.2 *SQL injection*

O ataque de *SQL injection (Structured Query Language)* é uma das ameaças mais importantes para aplicações web. Esses ataques são lançados por usuários especialmente criados em aplicativos da Web que usam operações de *string* de baixo nível para construir consultas SQL [6]. Esses ataques são baseados na injeção de *strings* em consultas de banco de dados que alteram seu uso pretendido. Isso pode ocorrer se uma plataforma da Web não filtra adequadamente a entrada do usuário. [33]. Uma exploração de injeção SQL bem-sucedida pode ler dados confidenciais do banco de dados, modificar dados do banco de dados (Inserir / Atualizar / Excluir), executar operações de administração no banco de dados (como encerrar o SGBD). [23]

Os ataques de *SQL injection* podem ser realizados através de várias maneiras: via métodos *GET*, *POST* e *BLIND SQL Injecion*. [29]

2.3 Ataques via método GET

Este método é utilizado quando queremos passar uma informação de uma página para outra, através da URL (barra de endereços). A figura a seguir mostra uma requisição utilizando o método GET. [9]

Figura 4. Exemplo de método GET na barra de endereços.

No exemplo acima, a informação passada é cat valendo 3. Um ataque via método get se aproveita da alteração das informações antes do envio.

Figura 5. Exemplo de alteração de variável utilizando método GET.

Na figura 5, é possível alterar uma variável e seu valor antes de uma requisição, ou seja, antes que essas informações sejam enviadas para outra página. Em ambos os casos, o atacante modifica o valor do parâmetro 'id' no seu navegador para enviar: 'ou' 1 '=' 1. Isso altera o significado de ambas as consultas para retornar todos os registros da tabela do banco de dados.

2.4 Ataques via método POST

A grande diferença entre os métodos GET e POST provavelmente é a visibilidade. Uma requisição GET é enviada como *string*, anexada a URL, enquanto que a requisição POST é encapsulada junto ao corpo da requisição HTTP e não pode ser vista. [15]

Exemplos de método POST:

```
<form name="formContato" method="POST" action="enviar_email.php">
    <p>
        Nome: <input type="text" name="nome" /><br />
        E-mail: <input type="text" name="email" /><br />
        Mensagem:<br />
        <textarea name="mensagem" id="mensagem" cols="45"
        rows="5"></textarea>
    </p>
    <p>
        <input type="submit" name="button" id="button" value="Enviar
        Mensagem" />
    </p>
</form>
```

Figura 6. Exemplo de método *POST* no formulário HTML.O autor.

A imagem acima mostra mensagens sendo enviadas pelo método *POST*, através de um formulário HTML.

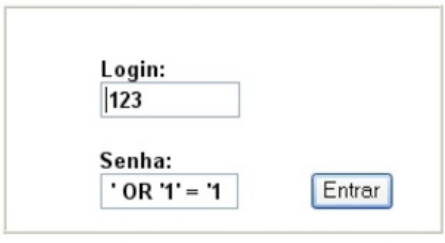

Figura 7. Exemplo *SQL injection* método POST no formulário em uma página web.

Tanto a figura 6 quanto a figura 7 representam exemplos de método POST. Entretanto, é possível perceber que a utilização do *SQL injection* acontece na figura 7, uma vez que o código injetado no campo senha significa que o comando passado nele faz com que, independente do *login* e senha digitados, a condição seja sempre verdadeira, permitindo assim o acesso do usuário à aplicação sem o mesmo possuir a devida permissão.

2.5 Ataques Blind SQL injection

Neste tipo de ataque o teste é realizado ao manipular as variáveis ou valores presentes no código SQL. Isso pode ser feito ao inserir um código SQL que representa uma tautologia, trocar nomes de variáveis ou valores.

O *Blind SQL injection* pode ser usado para testar vulnerabilidades existentes e encontrar o nome de tabelas no banco de dados. Todas as técnicas são baseadas no "ataque de adivinhação", porque existem apenas duas saídas diferentes: Verdadeiro ou falso. [33]

A figura 8 mostra um exemplo de técnica de injeção *blind sql injection*:

Figura 8. Exemplo de *Blind SQL injection*.

O exemplo da consulta na figura 8 é o mesmo em relação à figura 9, porém a consulta agora é SQL.

```
SELECT * FROM nomeDaTabela WHERE cat = 3 and 1=0
```

Figura 9. Exemplo de *Blind SQL injection* SQL.

No caso da figura 9, a consulta retornará falsa (*False*), porque "1" é diferente de zero. Se no lugar do número "um" fosse ZERO, logo, a consulta retornaria verdadeiro (*True*).

2.6 Métodos de Prevenção

Segundo a norma NBR ISO/IEC 27002 Código de Prática para a Gestão de Segurança da Informação [26], estabelecer diretrizes e princípios gerais para melhorar a gestão de segurança da informação em uma organização é essencial para proteger as informações consideradas importantes para a continuidade e manutenção dos objetivos de negócio da organização.

O perfil do desenvolvedor nos últimos anos tem sido modificado. A preocupação com a segurança deixa de estar em segundo plano e passa a se tornar requisito de projeto. Portanto, podemos dizer que os grandes trunfos de prevenção a *SQL injection* incidem em boas práticas de prevenção das aplicações. [12].

Algumas práticas podem ser adotadas, como: evitar a exibição das mensagens de erro contendo informações do servidor; estabelecer uma política de segurança rígida e criteriosa, limitando o acesso dos usuários; fazer uma validação da entrada de dados nos formulários, impossibilitando entrada de caracteres inválidos como: ('), (--) e (;); limitar a entrada de texto para o usuário no formulário de entrada de dados; uso de bibliotecas (APIs) para detecção de vulnerabilidades entre outras.

3. Metodologia

A metodologia adotada para analise comparativa das ferramentas tem como finalidade fazer uma comparação entre duas ferramentas gratuitas automatizadas de

detecção de vulnerabilidade *SQL injection*, visando tempo de execução dos ataques, ataque via métodos *GET*, *POST*, *Blind SQL injection*, instalação e usabilidade.

Para a inicialização do processo, é importante selecionar as diferentes ferramentas mais adequadas para estabelecer o ambiente de testes usado neste artigo. Assim, é importante selecionar o software de virtualização para a criação dos ambientes virtuais, seguindo-se do sistema operativo. Como software de virtualização (máquina virtual), escolhemos o *Oracle VM VirtualBox* [14], em relação ao sistema operacional separamos o *Kali Linux* [31].

As ferramentas de apoio utilizadas foram a *Uniscan* [17] e *WebScarab* [32], cujas principais funções foram essenciais para aumentar a cobertura de testes de vulnerabilidades para aplicação web. A ferramenta *Uniscan* pode ser considerada uma ferramenta que contribui para a detecção de vulnerabilidades, já que tenta localizar todos os arquivos e links dentro do site alvo. Sendo assim, utilizamos a ferramenta *Uniscan* para trazer respostas sobre possíveis palavras-chave, que serão utilizadas para o teste da ferramenta *themole* (tópico 4.1.3), uma das ferramentas avaliadas. Também utilizamos a ferramenta *Uniscan* para detectar possíveis vulnerabilidades de *Blind sql injection* (tópico 4.3). A ferramenta *webscarab* foi importante para detectar informações sobre o tipo de método utilizado em relação ao envio dos dados. Sendo assim, utilizamos o *Webscarab* para fazer testes via método *POST* conforme o tópico 4.1.

Outro fator importante a ser destacado é em relação à Seleção da Aplicação Web a ser testada. Os testes foram efetuados em uma aplicação Web exclusiva para testes de segurança disponibilizados pela empresa de segurança *Acutinex* disponível em http://testphp.vulnWeb.com/. Para a identificação de um possível caminho de ataque utilizamos a ferramenta *uniscan,* que retornou o seguinte link vulnerável: *"http://testphp.vulnweb.com/artists=2"*. Esse link será utilizado para fazer os testes de método *GET*.

Em relação à escolha das ferramentas, seguimos as seguintes regras: ferramentas automatizadas, com o objetivo de acelerar o processo de teste como forma de diminuir o trabalho manual; ferramentas sem interface gráfica com a utilização de comandos simples e compreensíveis; ferramentas de livre acesso e facilidade de instalação; ferramentas mais conhecidas em relação a ataques. As ferramentas escolhidas foram: *sqlmap* e *themole*.

4. Comparativo das Ferramentas

Essa seção tem por objetivo realizar a comparação das duas ferramentas selecionadas: *SQLmap* e *Themole*, abrangendo todos os teste apresentados na metodologia descrita no tópico 3.

4.1 Ataques via método GET

4.1.2 SQLMAP

Os testes a partir de agora serão feitos através da ferramenta automática *SQLmap* com a finalidade de acessar registros do banco de dados. Para isso, injetamos o comando mostrado na Figura 10.

Figura 10. Comando para fazer busca da base de dados. O autor.

O argumento -u indica a URL do endereço do alvo juntamente com o parâmetro a ser explorado na aplicação Web. O argumento --dbs indica para a ferramenta buscar todas as bases de dados existentes no domínio.

A ferramenta SQL nos retornou os nomes dos bancos de dados: *Acuart* e *Information_schema*, conforme podemos perceber na figura 11.

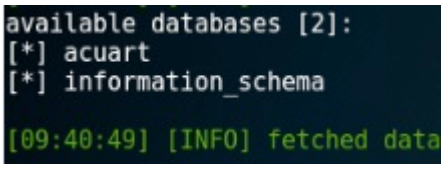

Figura 11. Nome dos bancos de dados. O autor.

Como o banco de dados *information_schemas* é padrão do MySQL(sistema de gerenciamento de banco de dados), prosseguimos os testes com o banco de dados *acuart*. Para o próximo passo, foi necessário saber o nome das tabelas existentes no banco de dados *acuart*. Sendo assim, utilizamos o comando de acordo com a Fig 11.

Figura 12. Comando para buscar nomes das tabelas dentro do banco de dados *acuart*. O autor.

O argumento -D indica o alvo que foi explorado, ou seja, o banco de dados *acuart* e o argumento *--tables* indica para a ferramenta listar todas as tabelas existentes no banco de dados.

Figura 13. Nomes das tabelas existentes no banco acuart. O autor.

Analisando a Figura 13, é possível perceber que a ferramenta trouxe como resultado todos os nomes das tabelas existentes dentro do banco de dados *acuart*.

O passo sucessor é identificar as colunas existentes dentro das tabelas. Escolhemos a tabelas *user* (usuário) para que possamos continuar com o processo e obter informações mais aprofundadas em relação à base de dados *acuart*, conforme o comando exibido na figura 14.

Figura 14. Comando para buscar nomes das colunas dentro da tabela *users*. O autor.

O argumento -T indica a tabela do banco de dados a ser listada juntamente com o argumento *--columns* que indica a listagem das colunas da tabela.

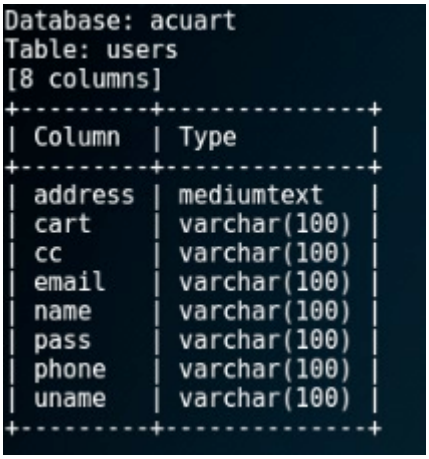

Figura 15. Resultado dos nomes das colunas existentes no dentro da tabela users. O autor.

Para finalizar o processo, foi necessário obtermos informações sobre qual dado do usuário queríamos ter acesso. Optamos por saber o nome, a senha e login. Para isso, o comando da imagem abaixo foi adequado para obtermos essas informações.

Figura 16. Comando para buscar nomes das colunas dentro da tabela *users*. O autor.

De posse de todas as informações da estrutura do banco de dados por meio das injeções anteriores, podemos agora fazer um despejo (*dump*) dos dados desejados. Para a demonstração, foi utilizada a tabela *users* onde estão os dados dos usuários do sistema como login, senha e nome.

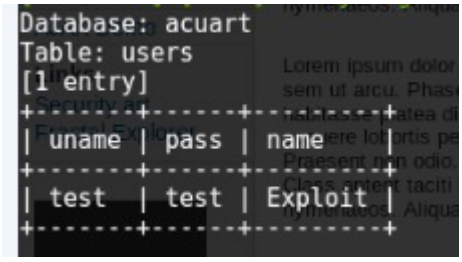

Figura 17. Resultado das informações pessoais do usuário. O autor.

Como podemos observar na Figura 17, a ferramenta *sqlmap* foi eficiente ao conseguir alcançar todas as informações possíveis de usuários registrados no banco de dados a partir do método *GET*.

4.1.3 THEMOLE

Essa ferramenta apresenta um uso um pouco diferente da *sqlmap*. No terminal do *Kali Linux*, digitamos o comando *themole* para que a ferramenta seja inicializada, conforme a Figura 18. Para iniciar o processo de teste, precisamos fornecer para a ferramenta duas informações iniciais, que são: *url* e *needle*. A *url* é o endereço do *site* que queremos fazer o ataque e o *needle* é uma palavra-chave existente no site (Figura 20). Como não temos conhecimento sobre qual é a palavra chave, utilizamos a ferramenta chamada *uniscan* conforme explicado na metodologia. As palavras-chave encontradas pela ferramenta foram: *consectetuer* e *aliquam* (figura 19).

Figura 18. Início da ferramenta *themole*. O autor.

A Figura 18 tem como objetivos mostrar como fazer a inicialização da ferramenta *themole*, para isso basta digitar o comando *themole* no terminal do *kali Linux*.

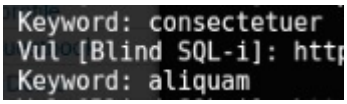

Figura 19. Palavras chaves encontradas pela ferramenta *Uniscan*. O autor.

Selecionamos a palavra *aliquam* para fazer o teste. Em primeiro lugar, preenchemos as informações *url*, *needle* e depois digitamos *schemas* para encontrar a base de dados, conforme a Figura 20.

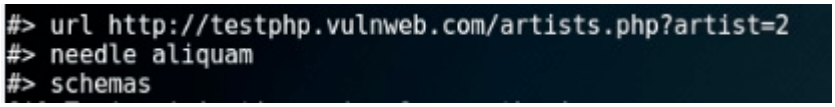

Figura 20. Comando para encontrar banco de dados. O autor.

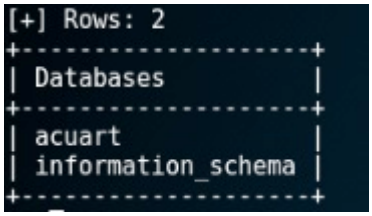

Figura 21. Imagem das informações do banco de dados. O autor.

De acordo com a imagem acima é possível perceber que a ferramenta *themole* também encontrou os resultados referentes ao banco de dados *acuart*, com a utilização de comandos simples. Logo, para descobrimos as tabelas contidas no banco de dados *acuart*, escrevemos o comando abaixo:

Figura 22. Comando utilizado para buscar tabelas dentro da base de dados *acuart*. O autor.

A palavra *tables* refere-se ao argumento utilizado para buscar todas as tabelas do banco de dados.

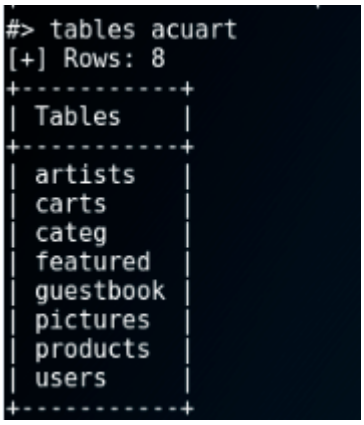

Figura 23. Nomes das tabelas existentes no banco de dados *acuart*. O autor.

O próximo comando utilizado foi responsável por trazer todas as colunas da tabela usuário *users*. Esse comando pode ser mais bem visualizado analisando a Figura 24.

Figura 24. Comando utilizado para obter as colunas existentes na tabela de usuários. O autor.

O argumento *columns* refere-se às colunas existentes do banco de dados. Informamos o comando da Figura 24 para a ferramenta e ela traz como resultado todas as colunas do banco de dados *acuart* da tabela *users*.

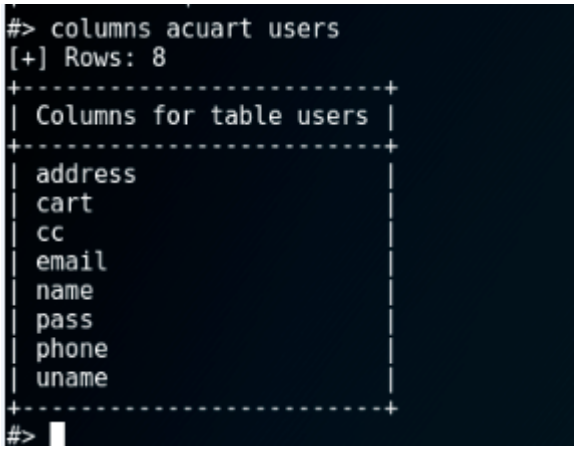

Figura 25. Imagem das colunas da tabela users. O autor.

Para finalizar, é necessário fazer uso do comando *query* seguido dos campos encontrados na tabela *users* (Figura 25). Sendo assim, escolhemos um dos campos mais importantes utilizados pelos *hackers*. Esses são: *uname* (usuário)**,** *name* (nome) e *pass* (senha), conforme a Figura 26.

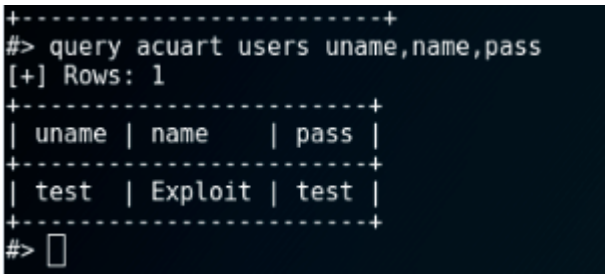

Figura 26. Resultado das informações pessoais do usuário. O autor.

Observando as imagens acima podemos perceber que tanto a ferramenta *SQLmap* quanto a ferramenta *Themole* foram capazes de chegar ao mesmo resultado utilizando o método GET.

4.1 Ataques via método POST

Para a realização dos testes utilizando o método *POST*, foi necessário uma ferramenta de *scanner* chamada *webScarab*. Com esse software é possível identificar como os formulários são enviados: método *GET* ou método *POST*. Dessa maneira, o WebScarab trouxe dados (ou resultados) que confirmam que as informações estão sendo enviadas pelo método POST. Além de trazer outras informações, como nome do domínio e número da porta usada, como mostra a figura 21. Sendo assim, foi preciso criar um arquivo com extensão .txt, com todas as informações descritas pelo *webscarab*, conforme informações abaixo.

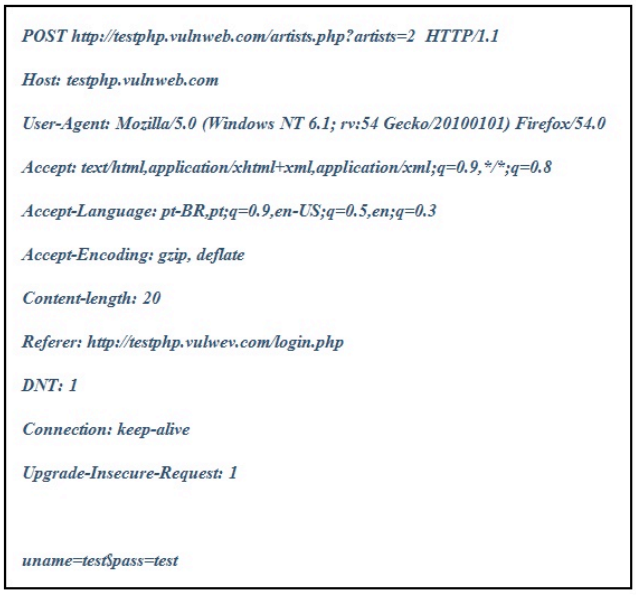

Figura 27. Requisição método POST encontradas pela ferramenta *WebScarab*.O autor.

4.2.1 SQLMAP

Para a realização do próximo passo, utilizamos o arquivo criado com nome POST.txt. Então, aplicamos um comando apropriado para encontrarmos a base de dados no site *vulnweb* utilizando o método POST. Esse comando pode ser visto na Figura 28.

Figura 28. Comando para encontrar a base de dados. O autor.

O comando –r é utilizado para identificação do nome do arquivo com extensão .txt, nesse caso POST, e o argumento –p serve para identificação do nome da pasta, nesse caso *uname*. O *sqlmap* retornou como resultado o nome de duas bases de dados *acuart* e *information_schema*, conforme os testes anteriores. É possível verificar os nomes das tabelas através da figura 24.

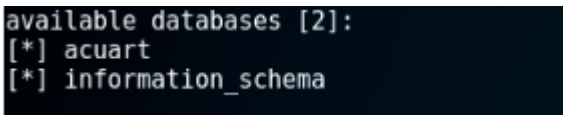

Figura 29. Nomes das bases de dados método POST. O autor.

Logo, o passo seguinte foi encontrar as tabelas existentes dentro da base de dados *acuart*. Portanto, utilizamos o comando mostrado na figura abaixo.

Figura 30. Comando para encontrar as tabelas do banco *acuart*. O autor.

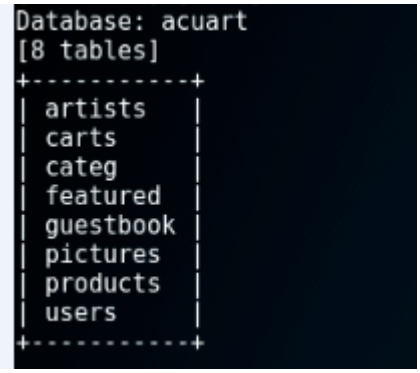

Figura 31. Nomes das tabelas do banco acuart. O autor.

A figura 31 mostra todos os nomes referentes às tabelas do banco de dados *acuart*. Dando prosseguimento aos testes, agora o objetivo é encontrar as colunas dentro da tabela *users* (tabela escolhida nos testes anteriores). Para isso, utilizamos o comando conforme a Figura 32.

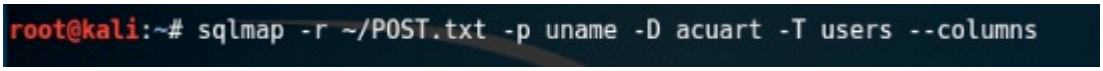

Figura 32. Comando para localizar as colunas da tabela *users*. O autor.

Os resultados dos nomes das colunas da tabela *users* (usuários) foram: *address*, *cart*, *cc*, *email*, *name*, *pass*, *phone* e *uname*, como demonstra a Figura 33.

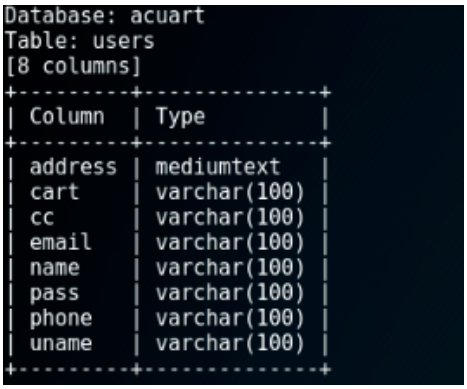

Figura 33. Colunas da tabela users. O autor.

Para finalizar os testes com a ferramenta *sqlmap* utilizando o método *POST,* foi empregado o comando mostrado na figura 34, para acessar todas as informações pessoais dos usuários.

```
root@kali:~# sqlmap -r ~/POST.txt -p uname -D acuart -T users -C name,pass,uname --dump
```

Figura 34. Comando para acessar informações pessoais do usuário. O autor.

```
Database: acuart
Table: users
[1 entry]
+-------+------+---------+
| uname | pass | name    |
+-------+------+---------+
| test  | test | Exploit |
+-------+------+---------+
```

Figura 35. Informações pessoais de usuário. O autor.

Como resultado, a ferramenta *sqlmap* trouxe exatamente as informações pessoais do usuário selecionadas como: *uname* (usuário), *pass* (senha) e *name* (nome). Com essas informações, é possível ter acesso à área restrita de usuários dentro do *site* e fazer quaisquer alterações, como: excluir nome, alterar nome, alterar endereço entre outros campos.

4.2.2 THEMOLE

O *themole* não foi capaz de detectar vulnerabilidades utilizando o método POST, pois com essa ferramenta os tipos de testes suportados para *SQL injection* são: testes de união, testes blind SQL injection e método GET. [27]

4.3 Ataques *Blind SQL injection*

Para a realização dos testes *BLIND SQL injection*, utilizamos a ferramenta *Uniscan* para detectar possíveis falhas *BLIND*. O resultado da ferramenta pode ser observado na figura 36, que traz todos os links como possibilidades de vulnerabilidades *BLIND*.

```
Blind SQL Injection:
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/listproducts.php?cat=2+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/artists.php?artist=1+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/artists.php?artist=2+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/artists.php?artist=3+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/listproducts.php?cat=1+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=6+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=1+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=2+AND+1=1
[+] Keyword: aliquam
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=4+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=3+AND+1=1
[+] Keyword: aliquam
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/product.php?pic=5+AND+1=1
[+] Keyword: consectetuer
[+] Vul [Blind SQL-i]: http://testphp.vulnweb.com/listproducts.php?artist=1+AND+1=1
[+] Keyword: consectetuer
```

Figura 36. Resultados de vulnerabilidades *Blind SQL injection* ferramenta *Uniscan*. O autor.

4.3.1 SQLMAP

Com todas essas possibilidades de vulnerabilidades que a ferramenta *Uniscan* detectou no site *vulnweb*, selecionou-se o primeiro link da figura 37. A partir disso, usamos o comando da imagem abaixo para acessar o banco de dados.

```
root@kali:~# sqlmap -u http://testphp.vulnweb.com/listproducts.php?cat=2+AND+1=1 --dbs
```

Figura 37. Comando para acessar banco de dados utilizando *BLIND SQL injecion*. O autor.

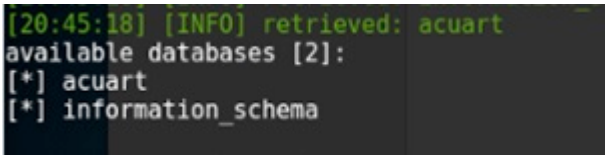

Figura 38. Imagem de informações de banco de dados utilizando *Blind SQL*. O autor.

O passo seguinte foi encontrar as tabelas do banco *acuart* pela técnica *BLIND*. Para isso, usamos o comando da figura 39.

Figura 39. Comando para encontrar as tabelas do banco de dados *acuart*. O autor.

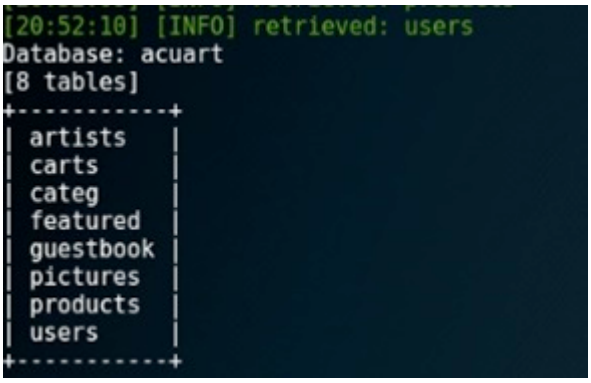

Figura 40. Nomes das tabelas do banco de dados *acuart* teste *BLIND SQL injection*. O autor.

Para encontrar as colunas de tabela *users* (usuários) pela técnica *Blind*, utilizamos o comando da figura 41. O resultado obtido por esse comando pode ser observado na figura 42.

Figura 41. Comando para encontrar as colunas da tabela *users*. O autor.

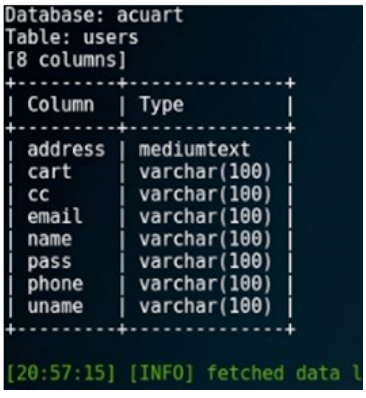

Figura 42. Informações dos nomes das colunas das tabelas *users* método *BLIND*. O autor.

Para ter acesso a informações pessoais dos usuários utilizando o *BLIND SQL injection* lançamos mão do comando mostrado na figura 43.

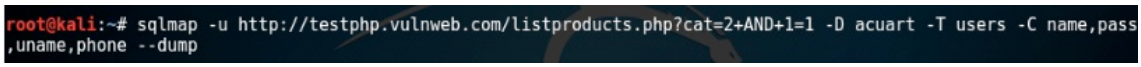

Figura 43. Comando para encontrar informações pessoais de usuários. O autor.

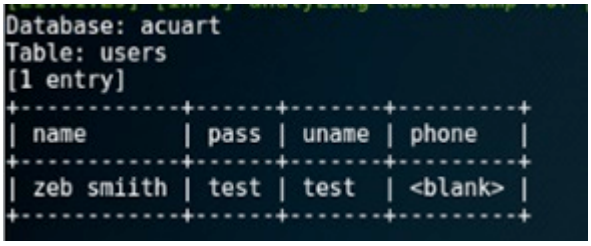

Figura 44. Resultado final de informações de usuários método *BLIND*. O autor.

Analisando a figura 44, percebese que foi possível obter informações do *name* (usuário), *pass* (senha), *uname* (usuário) e *phone* (telefone).

4.3.2 *TheMole*

Para a detecção de vulnerabilidades usando a ferramenta *THEMOLE*, basta somente escrevermos o comando *mode blind* depois de acharmos os nomes das colunas com os nomes dos campos escolhidos: *uname e pass*, conforme exemplo da Figura 45.

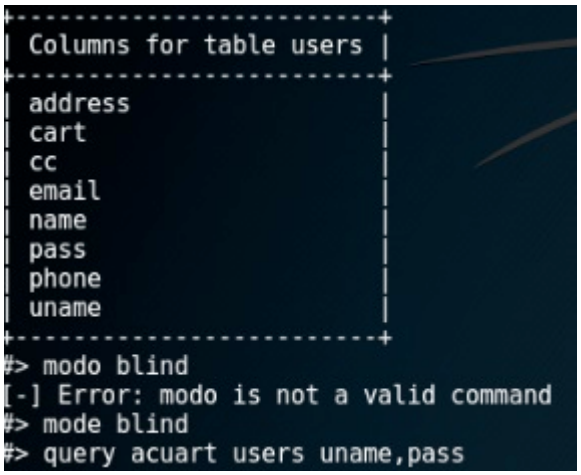

Figura 45. Utilização do comando *blind*. O autor.

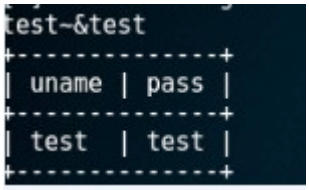

Figura 46. Resultado do comando *blind*. O autor.

Como podemos observar na figura 46, a ferramenta *TheMole* encontrou os valores referentes às variáveis *uname* e *pass* que representam informações pessoais de usuários.

5. Análise das ferramentas selecionadas

Para melhor compreensão da análise das ferramentas *sqlmap* e TheMole, classificamos os tipos de testes. A classificação das vulnerabilidades pode ser vista na Figura 40.

	SQLMAP	THEMOLE
Ataque via método GET	X	X
Ataque via método POST	X	
BLIND SQL injection	X	X
INSTALAÇÃO	X	
USABILIDADE		X

Tabela 47. Análise para comparação de ferramentas. O autor.

Pode-se perceber que tanto a ferramenta *sqlmap* quanto a ferramenta *themole* foram capazes de realizar quase todos os testes estabelecidos na metodologia. Observa-se que a ferramenta *sqlmap* foi apta para realizar todos os testes relacionados à *SQL injection*, porém não conseguiu atingir eficiência na facilidade de uso já que os comandos utilizados são considerados extensos e complexos para entendimento. Em relação ao *themole* percebe-se que ela não consegue fazer os testes de ataques via método POST, pois ela é desenvolvida para explorar as injeções baseadas em união, *BLIND SQL injection* entre outros, mas não é apropriada para detecção de vulnerabilidade de método *POST*. Outro fator importante é em relação à ferramenta *themole* que não obteve sucesso no item de instalação, pois para a utilização da mesma foi necessário uma instalação no sistema operacional Kali Linux.

5. Considerações finais

A metodologia apresentada torna-se eficiente, considerando que as duas ferramentas avaliadas foram capazes de passar em quase todos os testes estabelecidos pela metodologia. É importante destacar que a ferramenta *sqlmap* obteve maior sucesso em ralação a ferramenta *themole*. O *sqlmap* deixou de realizar apenas um teste (facilidade de uso) que tinha por objetivo a utilização e comandos simples. Já com a ferramenta *themole* não foi possível obter eficiência em relação aos testes de ataques via método POST e instalação.

Sendo assim, podemos dizer que este artigo ressaltou que é possível explorar as vulnerabilidades com certa facilidade, utilizando ferramentas livres e com pouco conhecimento técnico. Além disso, apresentou uma metodologia de testes lançando mão de critérios baseados em uma das vulnerabilidades mais conhecidas como o *SQL injection*. O recurso a ferramentas gratuitas automáticas para detecção de vulnerabilidades facilita o trabalho dos desenvolvedores e possibilita ter uma aplicação web mais segura. Outro fator importante a ser considerado é que com a metodologia estabelecida é possível obter maior facilidade para escolher uma ferramenta apropriada para testes de SQL injection, pois os resultados mostram que a ferramenta *sqlmap* apresentou maior eficiência em relação a ferramenta *themole*.

Referencias

[1] Arisholm Erik, Briand C Lionel and Johanessen B. Eivind (2009) “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models”.

[2] Fonseca Jose, Vieira Marco, Madeira Henrique (2009) “Testing and comparing web vulnerability scanning tools for SQL injection and XSS attack”.

[3] G.A Di Lucca, A. R. Fasolino, M. Mastroianni, P. Tramontana (2009) “Idenifying Cross Site Scripting Vulnerbilities in Web Applications”.

[4] Zhenyu QI, Jing XU, Dawei GONG, He TIAN (2009) “Travessing Model Desing Based on Strong-ssociation Rule for Web Application Vulnerbility Detection”.

[5] Vieira Marco, Antunes Nuno e Madeira Henrique (2009) “Using Web Security Scanners to Detect Vulnerabilities in Web Services”.

[6] Bandhakavi Sruthi, Bisht Prithvi, Madhusudan P, Venkatakrishnan V. N. (2009) “CANDID: Preveting SQL injection Attacks using Dynamic Candate Evaluations”.

[7] Antunes Nuno e Vieira Marco (2009) “Detecting SQL Injection Vulnerabilities in Web Services”, Fourth Latin-American Symposium on Dependable Computing.

[8] Antunes João, Neves Nuno, Correia Miguel, Verrisimo Paulo e Neves Ruy (2010) “Vulnerability Discovery with Attack Injection”, IEEE Transactions on Software Engineering.

[9] OWASP SQL INJECTION (2016), “Testing for SQL Injection”, Disponível em: <https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)>. Acesso em: 01 Mar 2017.

[10] Vissotto Antonio, Bosco Erick, Bruschi C. Gustavo, Silva Luis A. (2010) “Prevenção de Ataques: XSS Residente e SQL Injection em Banco de Dados PostgreSQL em ambiente WEB”.

[11] Basso Tânia, Fernandes Plínio César Simões, Jino Mario e Morais Regina. (2010) “Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool”, International Conference on Dependable Systems and Networks Workshops (DSN-W).

[12] Linhares, Heitor Magaldi; Quintão, Lima Patrícia; Bernardo, Luiz André; Almeida, Cesar Henrique Rodrigo, Lima, Santos Rogério. (2010) “SQL Injection, entenda o que é, aprenda a evitá-lo”.

[13] PONEMON INSTITUTE (2017), “Measuring Trust In Privacy And Security”, Disponível em: < https://www.ponemon.org/>. Acesso em: 16 Jun 2017.

[14] A. Dessiatnikoff, R. Akrout, E. Alata, M. Kaâniche e V. Nicomette (2011) “A clustering approach for web vulnerabilities detection”, IEEE Pacific Rim International Symposium on Dependable Computing.

[15] DEVMEDIA (2017), “SQL Injection em ambientes Web”, Disponível em: <http://www.devmedia.com.br/sql-injection-em-ambientes-web/9733>. Acesso em: 16 Jun 2017.

[16] Scholte Theodoor e Robertson William. (2012) “Preventing Input Validation Vulnerabilities inWeb Applications through Automated Type Analysis”, IEEE 36th International Conference on Computer Software and Applications.

[17] Rocha Douglas, Kreutz Diego, Rogério Turchetti. (2012) “Uma Ferramenta Livre e Extensível Para Detecção de Vulnerabilidades em Sistemas Web”.

[18] Silva, Gomes Castro Suellen. (2012) “Uma ferramenta para execução de ataques sql injection”.

[19] Holm Hannes, Ekstedt Mathias e Sommestad Teodor. (2013) “Effort estimates on web application vulnerability discovery”, Hawaii International Conference on System Sciences”.

[20] Vernotte Alexandre. (2013) “Research Questions for Model-Based Vulnerability Testing of Web Applications”, IEEE Sixth International Conference on Software Testing, Verification and Validation.

[21] Franck Lebeau, Legeard Bruno, Peureux Fabien e Vernotte Alexandre. (2013) “Model-Based Vulnerability Testing for Web Applications”, IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops.

[22] Qu Binbin, Liang Beihai, Jiang Sheng e Ye Chutian. (2013) “Design of Automatic Vulnerability Detection System for Web Application Program”.

[23] Fonseca José, Seixas Nuno, Vieira Marco e Madeira Henrique. (2014) “Analysis of Field Data on Web Security Vulnerabilities”, IEEE Transactions on dependable and secure computing.

[24] Monteverde, Aparecido Wagner. (2014) “Estudo e análise de vulnerabilidade web”.

[25] Fonseca José, Vieira Marco e Madeira Henrique. (2014) “Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection”, IEEE Transactions on dependable and secure computing.

[26] NORMA BRASILEIRA (2017), “ABNT ISSO/IEC 2702”, Disponível em: <http://www.fieb.org.br/download/SENAI/NBR_ISO_27002.pdf>. Acesso em: 12 Junho 2017.

[27] ALDEID THEMOLE (2017), “description themole”, Disponível em: <https://www.aldeid.com/wiki/TheMole>. Acesso em: 12 Abril 2017.

[28] Baden Delamore e Ko K. L. Ryan. (2014) “Escrow: A Large-Scale Web Vulnerability Assessment Tool”, International Conference on Trust, Security and Privacy in Computing and Communications.

[29] Yoo Seung-Jae e Yang Jeong-Mo (2014) “Web login Vulnerability Analysis and Countermeasures”, Computer Graphics: IEEE Xplore Digital library.

[30] Huang Cheng, Yong jia Liu, Fang Yong, Zuo Zheng (2015) “A study on Web Security incidentes in China by analyzing vulnerability disclosure platforms”.

[31] Rafique Sajjad, Humayun Mamoona, Hamid Bushra, Ansar, Abbas Muhammad Akhtar e Iqbal Kamil. (2015) “Web Application Security Vulnerabilities Detection Approaches: a Systematic Mapping Study”.

[32] Vibhandik Rohan e Bose Kumar Arijit. (2015) “Vulnerability Assessment of Web Applications A Testing Approach”.

[33] Makino Yuma e Klyuev Vitaly. (2015) “Evaluation of Web Vulnerability Scanners”, IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.

[34] Delamore Baden e Ko K. L. Ryan. (2015) “A Global, Empirical Analysis of the Shellshock Vulnerability in Web Applications”, IEEE Trustcom/BigDataSE/ISPA.

[35] Medeiros Ibéria; Neves F. Nuno; Correia Miguel. (2015) “Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives”.

[36] António, Teixeira; Rocha, José. (2016) “Identificação de vulnerabilidades em aplicações web open-source”.

[37] Simone, Quatrini; Rondini, Marco. (2016) “Blind Sql Injection with Regular Expressions Attack”.