

SÉRIE WEBAPP PARA PENTESTER E APPSEC

COMO CRIAR SCANNERS

PATH TRAVERSAL

O MANUAL PASSO A PASSO
de como criar seus próprios scripts para
identificar e tratar vulnerabilidades

FERNANDO MENGALI

SUMÁRIO

Introdução.....	01
2.0 PRÉ-REQUISITOS.....	04
3.0 CRIANDO O LABORATÓRIO/AMBIENTE.....	04
3.1 CRIANDO O BANCO DE DADOS.....	05
4.0 CRIANDO A PÁGINA PHP VULNERÁVEL.....	10
5.0 A PÁGINA PHP COM O CÓDIGO VULNERÁVEL.....	11
6.0 SITE VULNERÁVEL.....	13
7.0 TEORIA: COMO DETECTAR SQL INJECTION.....	14
8.0 PRÁTICA: DETECTANDO INJEÇÃO DE SQL.....	15
9.0 CONSTRUÇÃO DO SCANNING.....	15
10.0 PERL NO LINUX.....	18
11.0 CODIFICANDO A FERRAMENTA DE AUTOMAÇÃO.....	18
12.0 IMPLEMENTAÇÕES.....	24
13.0 CÓDIGO COMPLETO.....	27
14.0 CORRIGINDO VULNERABILIDADE.....	31
15.0 PROTEÇÃO COM WAF MODSECURITY OU NAXSI.....	33
16.0 SOBRE O AUTOR.....	34

INTRODUÇÃO

Nesse artigo, desenvolveremos uma ferramenta com a linguagem de programação Perl que identificará páginas de internet que possuem vulnerabilidades de Path Traversal.

Primeiro, iremos apresentar o processo de **identificação manual da vulnerabilidade de Path Traversal**, posteriormente você aprenderá como desenvolver um **script em Perl para detectar automaticamente** esse tipo de vulnerabilidade.

Esse artigo não apresenta técnicas avançadas para o desenvolvimento do nosso script em Perl para a identificação de vulnerabilidades. Para a elaboração desse artigo, utilizamos conceitos básicos, mas eficiente para identificar vulnerabilidades de Path Traversal, seja para um alvo específico ou vários alvos.

O conteúdo sobre como identificar vulnerabilidades de Path Traversal nesse artigo não são equivalentes as grandes ferramentas de mercado que atendem a metodologia DAST (Dynamic application security testing).

Não ensinamos a desenvolver algoritmos sofisticados que são utilizadas pelas ferramentas de análise dinâmica disponíveis comercialmente, mas compartilhamos informações suficientes para começar a criar suas primeiras ferramentas para identificar vulnerabilidades e continuar aperfeiçoando suas técnicas de desenvolvimento de scripts de identificação de vulnerabilidades.

2.0 PRÉ-REQUISITOS

Será necessário instalar os softwares abaixo para o desenvolvimento do laboratório:

- Sistema operacional **Microsoft Windows** (no artigo utilizei o Windows 10)
- Download do **WAMP 3.1.9**:
<https://sourceforge.net/projects/wampserver/>
- Download **Perl**:
<https://www.activestate.com/products/activeperl/downloads/>

3.0 CRIANDO O LABORATÓRIO/AMBIENTE

Nessa seção instalaremos o WAMP (Apache, MySQL e PHP) no Windows. Até o desenvolvimento desse artigo, foi utilizado o **WAMP 3.1.9**.

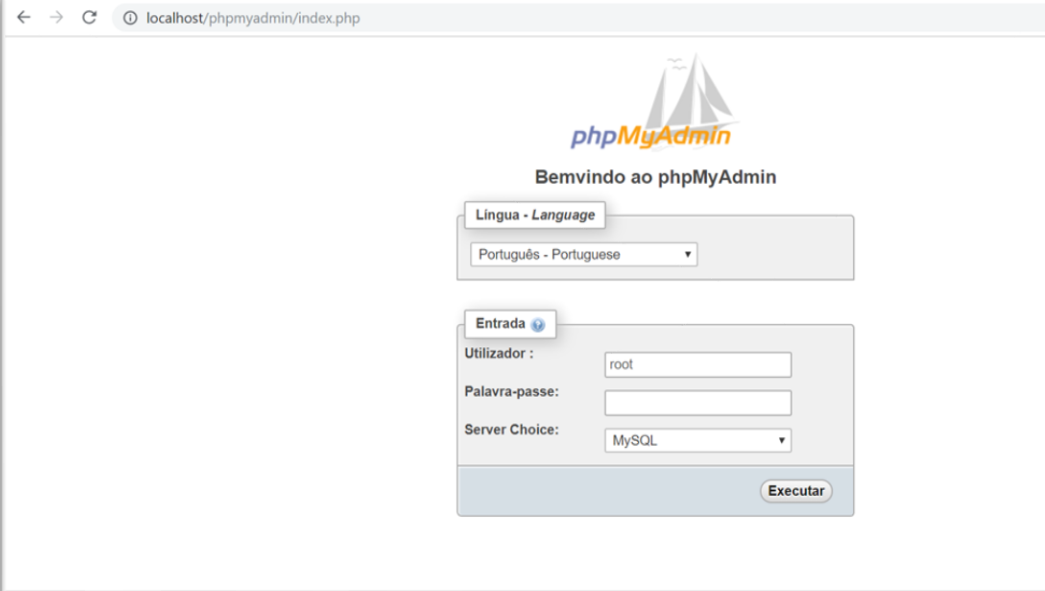
O processo de instalação é muito simples, portanto, não abordaremos.

Vamos considerar que você concluiu a instalação do WAMP e depois de instalado, vamos prosseguir com as configurações.

Se desejar acessar somente a seção sobre o desenvolvimento do scanning em Perl, acesse a **seção 8**.

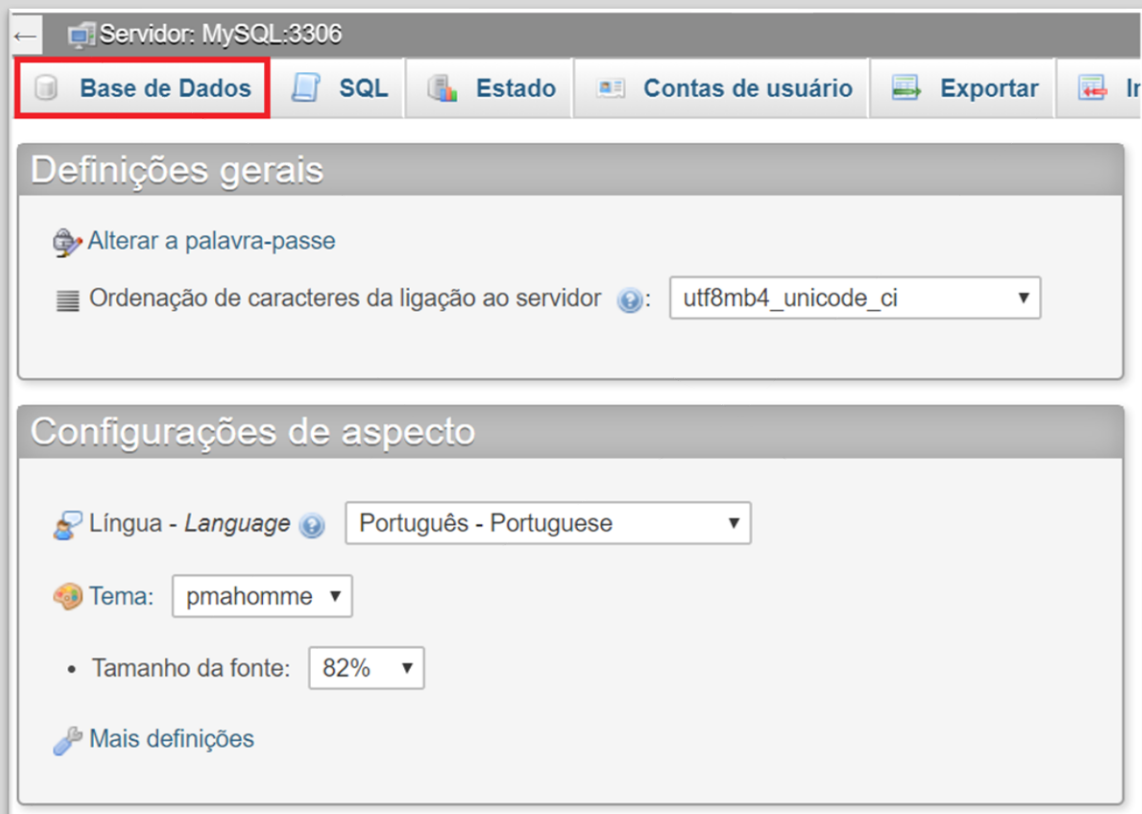
3.1 CRIANDO O BANCO DE DADOS

Acesse o phpMyAdmin: <http://localhost/phpmyadmin/index.php>

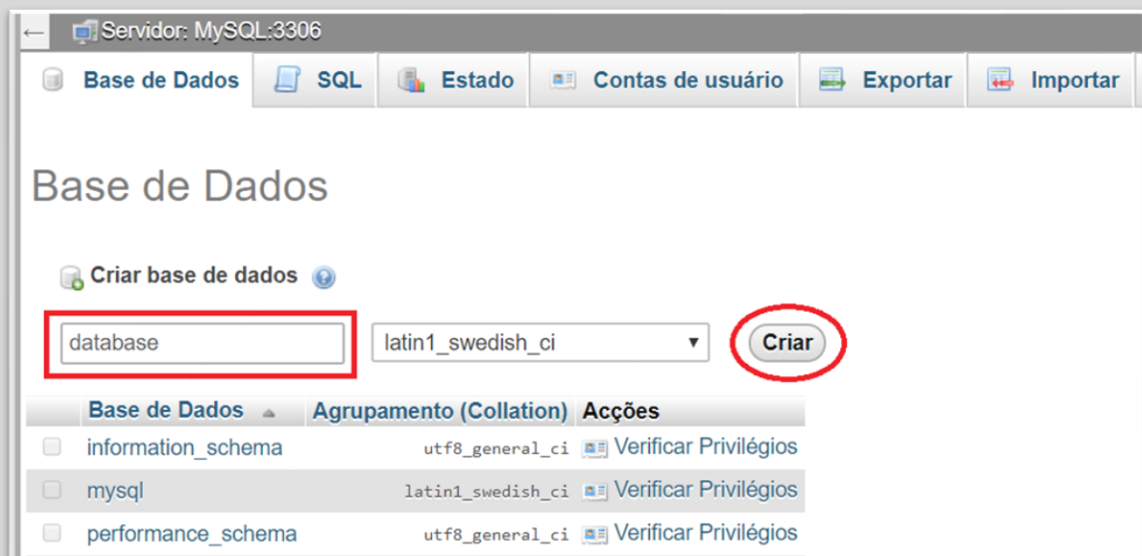


The screenshot shows the phpMyAdmin login interface in a web browser. The browser's address bar displays 'localhost/phpmyadmin/index.php'. The page features the phpMyAdmin logo at the top, followed by the text 'Bemvindo ao phpMyAdmin'. Below this, there is a language selection dropdown menu labeled 'Lingua - Language' with 'Português - Portuguese' selected. Underneath is an 'Entrada' (Login) section with a blue arrow icon. This section contains three input fields: 'Utilizador :' (Username) with 'root' entered, 'Palavra-passe:' (Password) which is empty, and 'Server Choice:' with 'MySQL' selected from a dropdown. An 'Executar' (Execute) button is located at the bottom right of the login form.

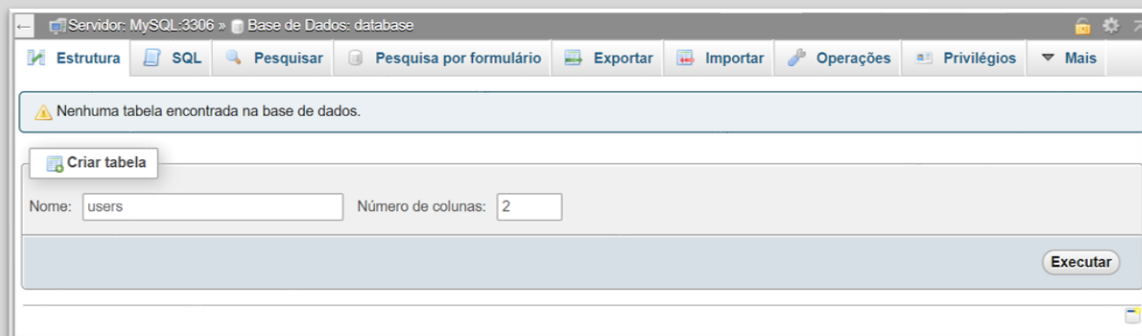
3.1.1 Por padrão, o phpMyAdmin tem o nome de usuário **root** e sem senha.



3.1.2 A próxima etapa, será criar o banco de dados.
Clique na aba **Base de Dados** para criar o banco de dados.



3.1.3 Digite o nome do banco de dados e depois clique em **Criar**.
No nosso exemplo, o nome do banco de dados é **"database"**.



3.1.4 Depois de criado o banco de dados **database**, vamos criar a **tabela**.

Acesse sua base de dados e depois a aba “**Estrutura**”.

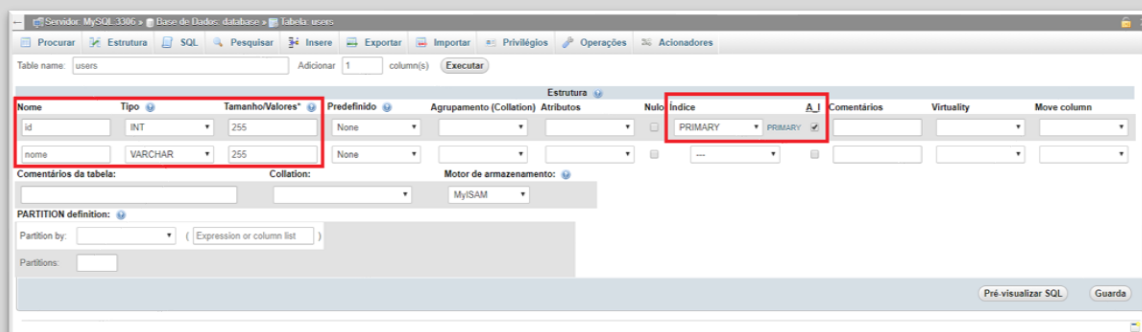
No campo **Nome**, digite o nome da tabela.

No nosso exemplo a tabela terá o nome de “**users**”.

No campo **Número** de colunas, digite a quantidade de colunas.

No nosso exemplo será 2 colunas.

Depois clique em **Executar**.



3.1.5 Na coluna **Nome**, informe o nome das colunas.

No exemplo será **id** e **nome**.

Na coluna **Tipo** defina o campo **id** como **INT** e o tamanho de **255**.

No campo **nome**, defina o **Tipo** como **VARCHAR** e o **Tamanho** **255**.

Não esqueça da coluna **índice**, será necessário definir como **PRIMARY**.

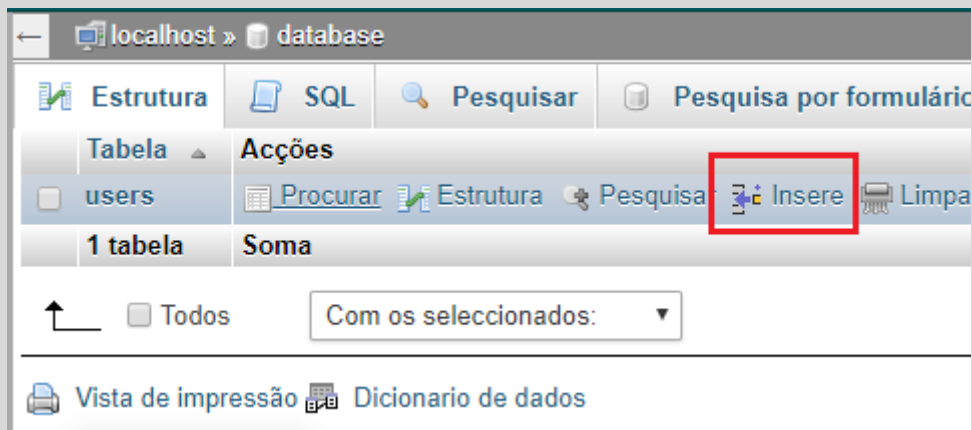
Na coluna **A_I**, defina como **PRIMARY** para colunas do **id**.

Caso deseje ser mais rápido, acesse a segunda aba chamada de **SQL** e informe o código abaixo para criar sua tabela:

```
CREATE TABLE IF NOT EXISTS `users` (
  `id` int(255) NOT NULL AUTO_INCREMENT,
  `nome` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
COMMIT;
```



3.1.6 Depois clique em “Executar”.



3.1.7 Agora, vamos criar um usuário.

Clique na aba “Estrutura” e depois clique no link “Inserir”.

Coluna	Tipo	Funções	Nulo	Valor
id	int(255)			
nome	varchar(255)			John Doe

Executar

3.1.8 Adicione o usuário **John Doe** e depois clique na opção “**Executar**”.

Tabela	Acções	Registos	Tipo	Agrupamento (Collation)	Tamanho	Suspensão
users	Procurar Estrutura Pesquisar Inserir Limpar Eliminar	1	MyISAM	latin1_swedish_ci	2 KB	-
1 tabela	Soma	1	InnoDB	latin1_swedish_ci	2 KB	0 Bytes

1 linha inserida.
Inserted row id: 1

```

INSERT INTO `users` (
  `id`,
  `nome`
) VALUES (
  NULL, 'John Doe'
);

```

[Editar] [Criar código PHP]

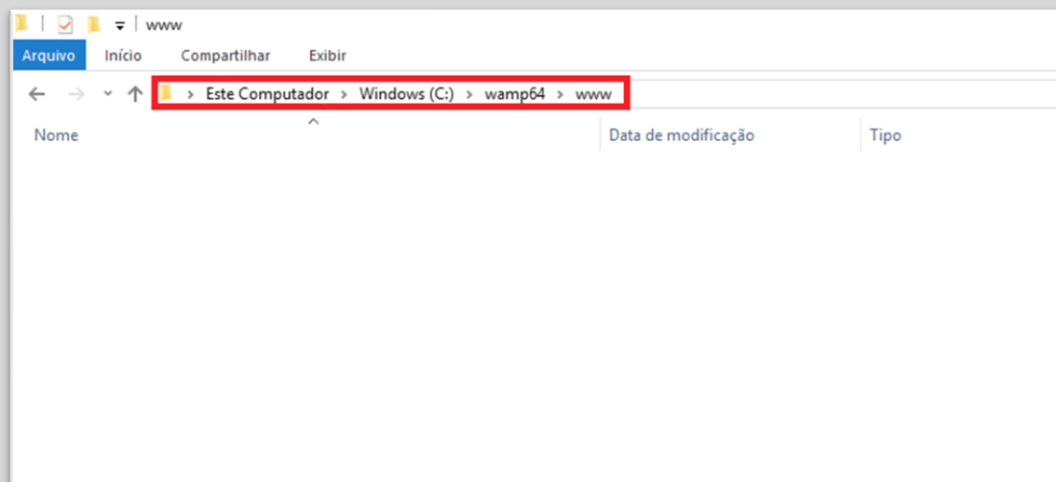
3.1.9 O usuário foi criado com sucesso.

Pronto, concluímos todas as etapas necessárias do banco de dados!
Agora, vamos para a última etapa, o desenvolvimento da página vulnerável.

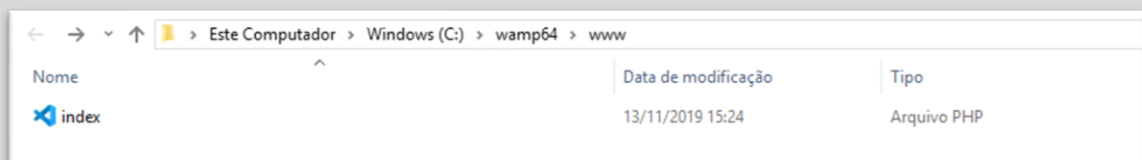
4.0 CRIANDO A PÁGINA PHP VULNERÁVEL

Acesse o diretório **www** para criarmos a página em PHP. Se você utilizou a sugestão do Windows para a instalação, o caminho será

“C:\Windows\wamp64\www”. Veja abaixo:



4.1.1 Quando você acessar o conteúdo do diretório **“www”**, visualizará alguns arquivos. Particularmente, eu removi todos os arquivos, deixando o diretório **“www”** vazio. A remoção dos arquivos do diretório **“www”** é sua escolha, eu acho melhor para trabalhar.



4.1.2 Nessa etapa iremos criar um arquivo com a extensão **“PHP”** com o nome de **“index”** no diretório **“www”**.

Depois de criar a página index, iremos adicionar o conteúdo ou código PHP vulnerável na página **index.php**.

Se você não codifica em PHP, não se preocupe, abaixo apresentamos o código e depois descrevemos o funcionamento de cada linha.

5.0 A PÁGINA PHP COM O CÓDIGO VULNERÁVEL

Abaixo apresentamos o código PHP vulnerável completo:

```
<?php

if (isset($_GET['id'])){
    $id = $_GET['id'];

    /* Setup the connection to the database */
    $mysqli = new mysqli('localhost', 'root', '', 'database');

    /* Check connection before executing the SQL query */
    if ($mysqli->connect_errno) {
        printf("Connect failed: %s\n", $mysqli->connect_error);
        exit();
    }

    /* SQL query vulnerable to Path Traversal */
    $sql = "SELECT username FROM users WHERE id = '$id'";

    /* Select queries return a result */
    if ($result = $mysqli->query($sql)) {

        $exist = $result->num_rows; /* Check User id exist */

        if ($exist > 0) {          /* if true show username*/

            while($obj = $result->fetch_object()){
                print($obj->username);
            }
        }
        else {

            /* or include value id*/

            include($id);

        }
    }

    /* If the database returns an error, print it to screen */
    elseif($mysqli->error){
        print($mysqli->error);
    }
}
```

```

else {

    /* Bad Code Quality */

}

}
?>

```

5.0.1 Você poderá copiar esse código e adicionar para a sua página **index.php**.

Não esqueça, sua página **index.php** deverá estar em “C:\Windows\wamp64\www”.

Essa etapa é bem simples, você não precisa ter conhecimentos de PHP para entender o código.

Se você quiser entender o código, continue lendo essa seção, pois descreverei cada linha na próxima página.

Inicialmente, nosso código receberá um parâmetro GET:

```

if (isset($_GET['id'])){
    $id = $_GET['id'];
}

```

5.0.2 Temos o if que valida a existência de dados ou parâmetros enviados para o método GET. Se houver algum dado trafegando via GET ele entrará no commando bloco IF e será armazenado na variável **id**.

Vejo o exemplo de acesso a nossa página PHP via a url: <http://localhost/index.php?id=1>

Após recebermos o valor, iniciamos a conexão com o banco de dados “**database**”.

Nas próximas linhas, **validamos** se a conexão com o banco de dados está funcionando.

```

$mysqli = new mysqli('localhost', 'root', '', 'database');

```

5.0.3 Nessa linha ocorre a conexão com o banco de dados.

A próxima etapa é validar a conexão com o banco de dados.

```

if ($mysqli->connect_errno) {

```

```

        printf("Connect failed: %s\n", $mysqli->connect_error);
        exit();
    }
}

```

5.0.4 No segundo if, validamos se há problemas na conexão com o banco de dados.

Se a conexão funcionar, seguimos para a seção **5.0.5** que apresenta a consulta com o banco de dados.

Senão funcionar, tivemos um problema na conexão com o banco de dados e receberemos uma mensagem de erro.

A mensagem de erro sempre ajudará na identificação do problema na conexão com o banco de dados.

```

/* SQL query vulnerable to Path Traversal */
$sql = "SELECT username FROM users WHERE id = '$id'";

```

5.0.5 Nessa linha, iniciamos a consulta com o nome de usuário na tabela **users**.

A consulta na tabela **username** utilizará o campo **id** e o valor armazenado na variável **\$id** para criar uma condição na busca.

```

if ($result = $mysqli->query($sql)) {

    $exist = $result->num_rows; /* Check User id exist */

    if ($exist > 0) {          /* if true show username*/

        while($obj = $result->fetch_object()){
            print($obj->username);
        }
    }
    else {

        /* or include value id*/

        include($id);

    }

}

```

5.0.6 Agora, verificamos a existência do usuário na tabela do banco de dados.

utilizamos a query armazenada na variável **\$sql** no **while**. Depois, utilizamos o **fetch_object** que retorna os dados como um objeto.

O comando **print**, será responsável por imprimir ou apresentar os dados armazenados no banco de dados.

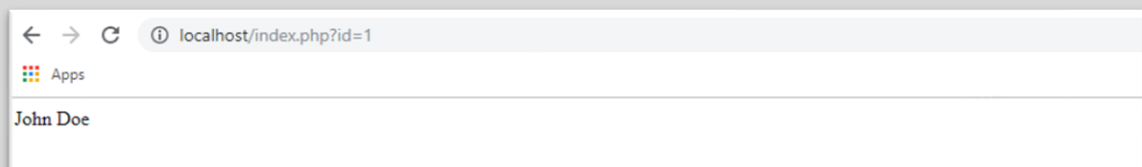
Se a nossa query gerar algum erro, será exibido na tela as informações do erro.

6.0 SITE VULNERÁVEL

Nessa seção acessaremos a página **index.php** vulnerável:

<http://localhost/index.php?id=1>.

Veja o resultado no browser:



6.0.1 O nome **John Doe** é exibido no browser, pois nome está associado ao número 1 da tabela **users** do banco de dados.

7.0 TEORIA: COMO DETECTAR PATH TRAVERSAL

Para identificar o nosso alvo vulnerável, utilizaremos uma técnica simples: **a inclusão de um arquivo externo**.

Na nossa URL de exemplo, a página **index.php** possui uma chave ou índice denominado **id** e um respectivo valor, o número **1**.

O número **1** está associado ao nome do usuário **John Doe** registrado na tabela **users** do banco de dados.

Entendido o funcionamento para a exibição de conteúdo por meio de parâmetros com o número de **id**, vamos entender como funciona a detecção de Path Traversal.

Quando um atacante deseja encontrar uma vulnerabilidade de Path Traversal será preciso adicionar o caminho de algum arquivo que existe no servidor na final da URL, exemplo:

<http://localhost/index.php?id=C:\Windows\System32\drivers\etc\hosts>

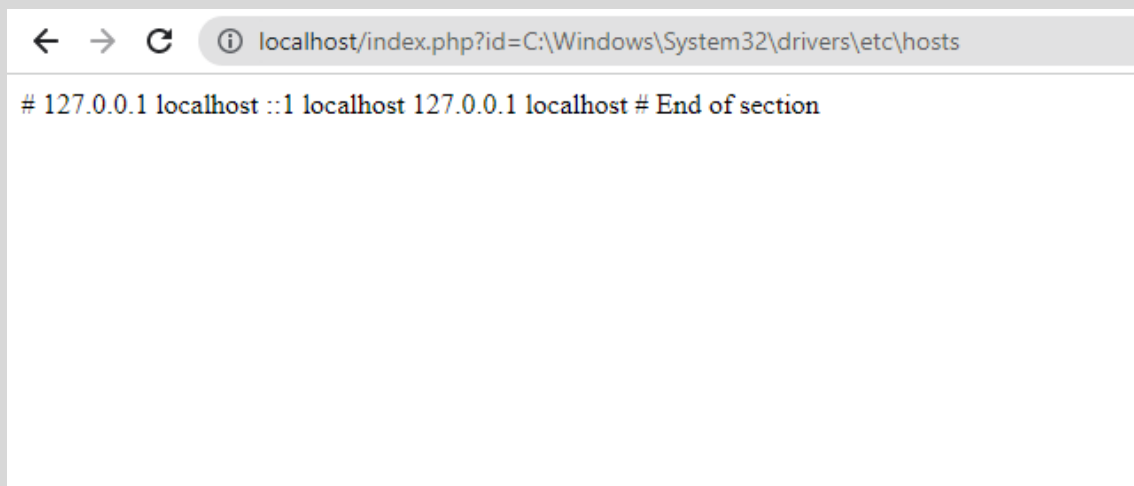
Se a aplicação possuir algum tipo de vulnerabilidade de Path Traversal o conteúdo do arquivo que estamos tentando ler será exibido.

8.0 PRÁTICA: DETECTANDO A VULNERABILIDADE DE PATH TRAVERSAL

Agora vamos descobrir se a página está vulnerável a Path Traversal. Para essa etapa, apontaremos para um arquivo externo e validaremos se a vulnerabilidade existe.

Agora vamos injetar a path do arquivo hosts do Windows na aplicação, conforme o exemplo da url:

<http://localhost/index.php?id=C:\Windows\System32\drivers\etc\hosts>



8.0.2 A página está vulnerável a Path Traversal.

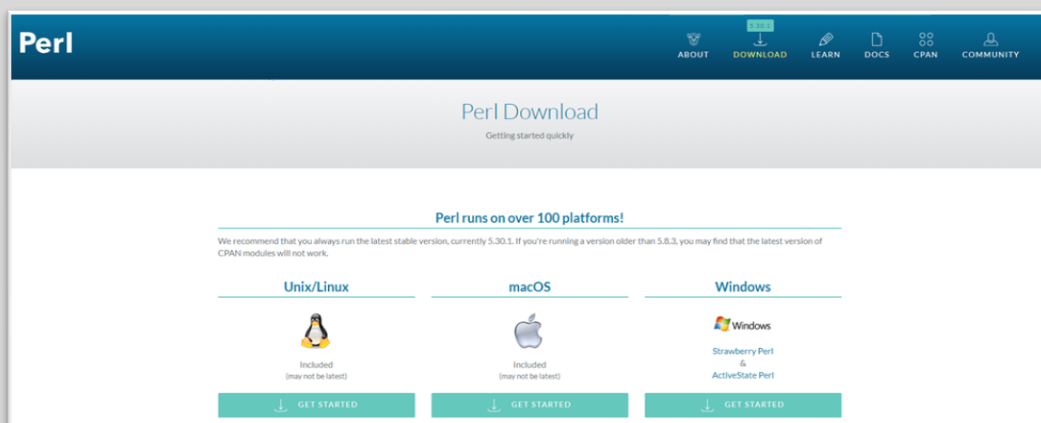
Entendemos como funciona a vulnerabilidade de Path Traversal, agora desenvolveremos a ferramenta para a automatização de identificação de vulnerabilidade Path Traversal, mas antes vamos instalar o interpretador de Perl para programar o script ou ferramenta em Perl.

9.0 CONSTRUÇÃO DO SCANNING

A linguagem de desenvolvimento escolhida para o desenvolvimento do script será o Perl. Você precisará de conhecimentos de programação em Perl, pois a ferramenta terá erros propositais, ou seja, apenas

desenvolvedores, analistas de segurança e interessados com aptidões de desenvolvimento entenderão o código.

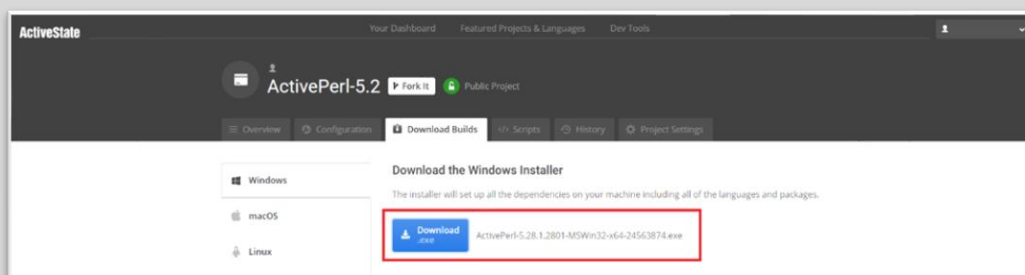
9.1 BAIXANDO O PERL PARA WINDOWS



9.1.1 Acesse a URL <https://www.perl.org/get.html> e escolha a plataforma que você está utilizando.

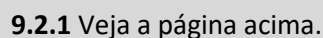
Você será redirecionado e solicitado a autenticar ou criar uma conta para baixar o Perl.

Depois de autenticado, você poderá baixar o Perl:

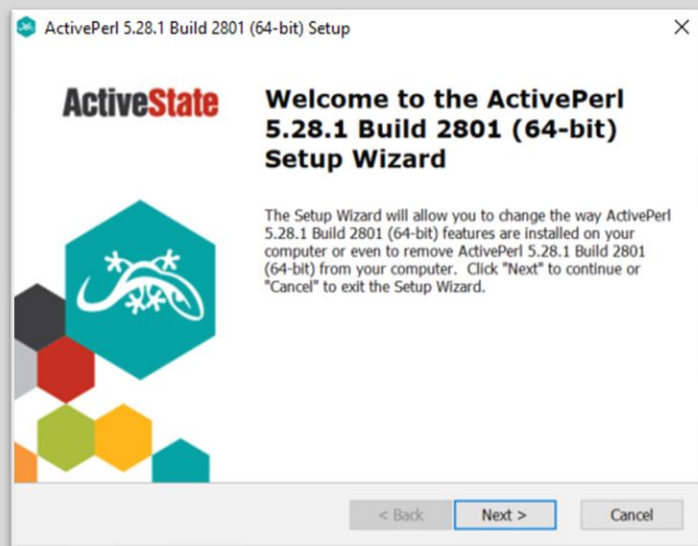


9.1.2 Clique no botão “Download”.

Outra opção é utilizar **Strawberry Perl**:
<http://strawberryperl.com/releases.html>



Depois de baixar o Perl para Windows, siga os procedimentos comum de instalação no Windows:



9.2.2 O restante você já sabe.

10.0 PERL NO LINUX

Se você está utilizando uma distribuição Linux, de preferência o Kali Linux, por padrão o Perl já está instalado.

Caso você deseje verificar se o Perl está instalado, digite os comandos **perl**

-help no terminal do Kali Linux:

```
root@kali:~# perl -help
Usage: perl [switches] [--] [programfile] [arguments]
  -0[octal]          specify record separator (\0, if no argument)
  -a                autosplit mode with -n or -p (splits $_ into @F)
  -C[number/list]    enables the listed Unicode features
  -c                check syntax only (runs BEGIN and CHECK blocks)
  -d[:debugger]      run program under debugger
  -D[number/list]    set debugging flags (argument is a bit mask or alphabets)
  -e program         one line of program (several -e's allowed, omit programfile)
  -E program         like -e, but enables all optional features
  -f                don't do $sitelib/sitecustomize.pl at startup
  -F/pattern/        split() pattern for -a switch (//s are optional)
  -i[extension]      edit <> files in place (makes backup if extension supplied)
  -Idirectory        specify @INC/#include directory (several -I's allowed)
  -l[octal]          enable line ending processing, specifies line terminator
  -[mM][.]module     execute "use/no module..." before executing program
  -n                assume "while (<>) { ... }" loop around program
  -p                assume loop like -n but print line also, like sed
  -s                enable rudimentary parsing for switches after programfile
  -S                look for programfile using PATH environment variable
  -t                enable tainting warnings
  -T                enable tainting checks
  -u                dump core after parsing program
  -U                allow unsafe operations
  -v                print version, patchlevel and license
  -V[:variable]      print configuration summary (or a single Config.pm variable)
  -w                enable many useful warnings
  -W                enable all warnings
  -x[directory]      ignore text before #!perl line (optionally cd to directory)
  -X                disable all warnings

Run 'perldoc perl' for more help with Perl.
root@kali:~#
```

11.0 CODIFICANDO A FERRAMENTA DE AUTOMAÇÃO

Nessa etapa iremos utilizar uma requisição para nossa página

<http://localhost/index.php?id=1>.

Utilizaremos umas aspas simples e trataremos a resposta.

11.1 CLASSES DE REQUISIÇÕES

Nosso código precisará de duas classes para fazer requisições para a página com vulnerabilidade.

São elas:

- LWP::UserAgent
- HTTP::Request
- LWP::Simple0

LWP::UserAgent

É uma classe responsável por atuar como um agente, durante uma requisição ou solicitação da web. Quando uma requisição é realizada será criado um objeto LWP::UserAgent com valores padrões.

HTTP::Request

A classe HTTP::Request faz uma requisição da URL ou página web que definiremos.

Conforme apresentado acima, teremos o cabeçalho **HTTP::Request** no nosso código para automatizar requisições.

LWP::Simple

É uma versão simplificada da biblioteca libwww-perl.

Possui várias funções e possibilita maior controle nos campos de cabeçalho.

O LWP::Simple busca rapidamente uma página e devolve a resposta. As respostas poderão ser: **is_error** ou **is_success**.

11.2 COMEÇANDO COM A CODIFICAÇÃO

Utilizando os três módulos descritos acima, poderemos adicioná-los no início do script e depois criar a estrutura de requisição em Perl para a página vulnerável.

```

1  #!/usr/bin/perl
2
3  use LWP::UserAgent;
4  use HTTP::Request;
5  use LWP::Simple;
6

```

11.2.1 Não esqueça de usar `#!/usr/bin/perl`, se estiver usando linux.

11.3 ENTRADA DE DADOS

A entrada de dados será utilizada para informar qual URL será verificada. Caso não queira informar a URL utilizando uma entrada de dados, poderá deixar o endereço de forma estática na variável.

Na **linha 7** criamos uma mensagem ou um prompt para o usuário digitar a URL.

Na **linha 8** criamos uma variável `$url` e depois adicionamos a entrada padrão `<stdin>`.

STDIN, poderá ser substituída por `<>`.

```

6
7  print "Por favor, informe a url: \n":
8  $url = <stdin>;
9

```

11.3.1 Caso o desenvolvedor não opte por utilizar uma entrada de dados, poderá utilizar uma variável estática para armazenar a URL que será verificada, exemplo:

```

9
10  $url = "http://localhost/index.php?id=";
11

```

11.3.2 Aproveitando a variável `$url`, vamos adicionar uma aspa simples ao final da url.

```

11
12  $url = $url."C:\\Windows\\System32\\drivers\\etc\\hosts";
13

```

11.3.3 No exemplo acima, estamos armazenando o endereço da url mais a aspa simples na variável `$url`.

Agora, podemos desenvolver a arquitetura da requisição:

```

13
14 $requisicao = HTTP::Request->new(GET=>$url);
15
16 $my $ua=LWP::UserAgent->new();
17 exit;
18 $ua->timeout(15);
19
20 my $resultado=$ua->request($requisicao);
21

```

11.3.4 A estrutura da requisição.

“Nessa etapa, exige conhecimentos de programação em Perl ou similar”.

Na **linha 14** utilizamos o módulo HTTP::Request, o método GET e o endereço da URL que analisaremos a resposta.

Na **linha 16** utilizamos um UserAgent e na **linha 18** passamos um parâmetro de 15 segundos de timeout.

Na **linha 20** a variável \$resultado armazena a requisição que será executada.

Como nossa requisição acessa uma página e tenta ler o conteúdo do arquivo hosts do Windows teremos uma resposta do seu conteúdo e validamos.

Nessa etapa recebemos a resposta do conteúdo da página. Se o conteúdo contém **a resposta 127.0.0.1**, a página é vulnerável!

```

23
24 if ($conteudo =~ "127.0.0.1") {
25
26     print "\n\n Vulnerable! \n\n";
27
28 }
29 else {
30     print "\n\n Not vulnerable! \n\n";
31 }

```

11.4.1 Veja a estrutura condicional para identificar uma página com vulnerabilidade de Path Traversal.

11.5 EXECUTANDO O SCRIPT

Esse é o resultado do script.

A screenshot of a terminal window with a dark background and light text. The window title is 'root@kali: /usr/bin'. The menu bar shows 'File Edit View Search Terminal Help'. The command 'perl tool.pl' has been executed. The script prompts 'Por favor, informe a url:' and the user has entered 'http://localhost/index.php?id=1'. The script then outputs 'Vulnerável!' and returns to the prompt.

```
root@kali: /usr/bin
File Edit View Search Terminal Help
root@kali: /usr/bin# perl tool.pl
Por favor, informe a url:
http://localhost/index.php?id=1

Vulnerável!
root@kali: /usr/bin#
```

12.0 IMPLEMENTAÇÕES

Algumas empresas possuem diversos sistemas, portais, sites internos e

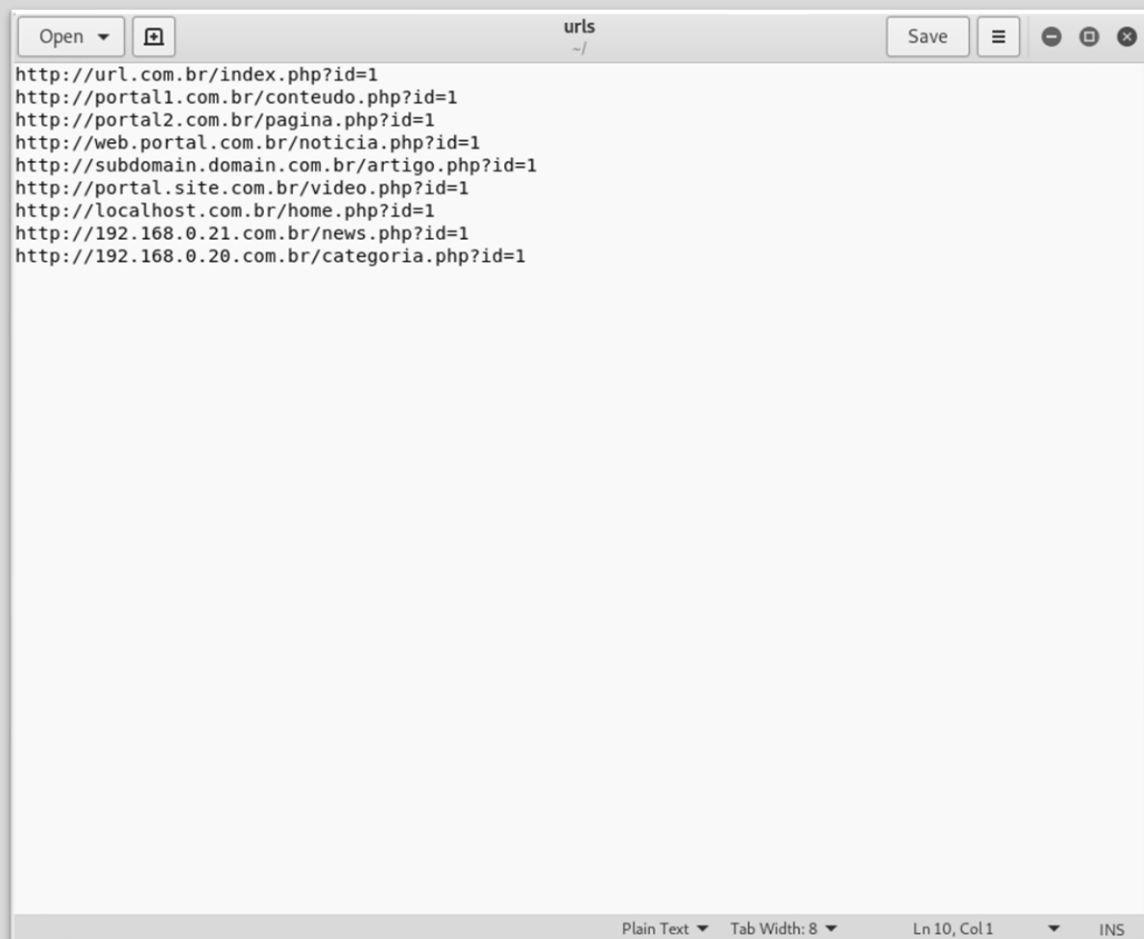
externos. Podemos adicionar todas as urls em um arquivo texto e alimentar o script **tool.pl**.

No momento que o script solicita o endereço da url, podemos substituir por uma função que solicite o nome do arquivo com as urls internas ou externas a serem testadas.

Esse processo facilita a execução de verificações e economiza tempo do analista de segurança.

Não precisa executar o script várias vezes, uma única execução é o suficiente para analisar vários endereços.

Abaixo, estou apresentando um modelo de arquivo texto com as urls a serem testadas como exemplo:



The image shows a screenshot of a text editor window. The title bar at the top reads 'urls' with a tilde icon. The window contains a list of eight URLs, each on a new line. The status bar at the bottom indicates 'Plain Text', 'Tab Width: 8', 'Ln 10, Col 1', and 'INS'.

```
http://url.com.br/index.php?id=1
http://portal1.com.br/conteudo.php?id=1
http://portal2.com.br/pagina.php?id=1
http://web.portal.com.br/noticia.php?id=1
http://subdomain.domain.com.br/artigo.php?id=1
http://portal.site.com.br/video.php?id=1
http://localhost.com.br/home.php?id=1
http://192.168.0.21.com.br/news.php?id=1
http://192.168.0.20.com.br/categoria.php?id=1
```

12.0.1 Lista de urls de sistemas, portais, sites internos e externos.

```

7  print "Por favor, informe a url: \n"; # Informe o arquivo texto com url
8  $url = <stdin>;
9
10 open( URL, "< $url" ) or die ( "Can't open file: $!");
11
12 @vector = <URL>;
13
14 $last = $#vector;
15
16 for ($i = 0; $i <= $last; $i++) {
17
18     print "verificando => ".$vector[$i]."\n";

```

12.0.2 Na linha 10 adicione o código responsável por ler o arquivo texto com urls.

A linha 12 armazena todo o conteúdo do arquivo no array **@vector**.

Na linha 14 acessamos o último elemento do array.

Na linha 16 desenvolvemos um for para acessar cada linha ou url armazenado no arquivo.

Na linha 18, temos o armazenamento de cada endereço na variável **\$url**.

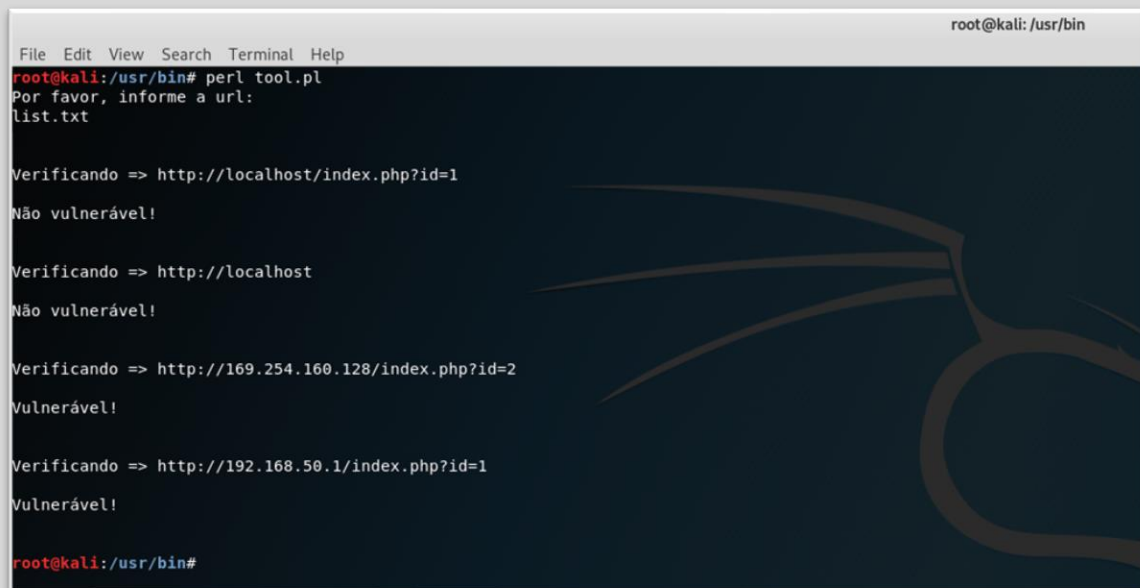
```

19
20 $url = $vector[$i]. "C:\\Windows\\System32\\drivers\\etc\\hosts";
21 my $requisicao=HTTP::Request->new(GET=>$url);
22
23 my $ua=LWP::UserAgent->new();
24 exit;
25 $ua->timeout(15);
26
27 my $resultado=$ua->request($requisicao);
28
29 if ($resultado->content =~ "127.0.0.1") {
30
31     print "\n Vulnerável! \n";
32 }
33 else {
34     print "\n Não vulnerável! \n";
35 }
36
37 }
38

```

12.0.3 Na linha 22 criamos um IF, que verifica se a resposta da página possui algum erro de banco de dados. Se houve algum erro, ele apresenta a imagem “**Vulnerável**” ou senão, “**Não vulnerável**”.

12.1 EXECUTANDO O SCRIPT



```
File Edit View Search Terminal Help
root@kali:/usr/bin# perl tool.pl
Por favor, informe a url:
list.txt

Verificando => http://localhost/index.php?id=1
Não vulnerável!

Verificando => http://localhost
Não vulnerável!

Verificando => http://169.254.160.128/index.php?id=2
Vulnerável!

Verificando => http://192.168.50.1/index.php?id=1
Vulnerável!

root@kali:/usr/bin#
```

12.1.2 Resultado do script verificando cada url armazenada no arquivo texto.

Você poderá adicionar novos recursos no script, como Thread, crawlers de páginas etc.

13.0 CÓDIGO COMPLETO

Abaixo é apresentado ambos os códigos em Perl para testar no seu ambiente particular.

13.1 CÓDIGO COMPLETO DAS FERRAMENTAS

Nessa seção temos a ferramenta para identificar vulnerabilidades, utilizando recursos simples de um scanning de vulnerabilidade.

```
#!/usr/bin/perl

use LWP::UserAgent;
use HTTP::Request;
```

```

use LWP::Simple;

print "Por favor, informe a url: \n";
$url = <stdin>;

$url = "http://localhost/index.php?id=";

$url = $url."C:\\Windows\\System32\\drivers\\etc\\hosts";

my $requisicao=HTTP::Request->new(GET=>$url);

my $ua=LWP::UserAgent->new();
exit;

$ua->timeout(15);

my $resultado=$ua->request($requisicao);

if ($resultado->content =~ "127.0.0.1") {

    print "\n\n Vulnerable! \n\n";

}
else {
    print "\n\n Not vulnerable! \n\n";
}

```

13.2 A FERRAMENTA COM IMPLEMENTAÇÕES

Nessa seção utilizamos um recurso no scanning para verificar vários IPs ou urls com vulnerabilidade de Path Traversal.

Para fazer essa verificação será preciso somente passar ou informar um arquivo de texto com vários ips ou urls.

```

#!/usr/bin/perl

use LWP::UserAgent;
use HTTP::Request;
use LWP::Simple;

print "Por favor, informe a url: \n"; # informe o arquivo texto com urls
a serem analisada!
$url = <stdin>;

open( URL, "< $url" ) or die ( "Can't open file: $!");

```

```
@vector = <URL>;

$last = $#vector;

for ($i = 0; $i <= $last; $i++) {

    print "verificando => ".$vector[$i]."\n";

$url = $vector[$i]." C:\\Windows\\System32\\drivers\\etc\\hosts";

my $requisicao=HTTP::Request->new(GET=>$url);

my $ua=LWP::UserAgent->new();
exit;
$ua->timeout(15);

my $resultado=$ua->request($requisicao);

if ($resultado->content =~ "127.0.0.1") {

    print "\n Vulnerável! \n";
}
else {
    print "\n Não vulnerável! \n";
}

}
```

14.0 CORRIGINDO VULNERABILIDADE

Irei demonstrar alguns tipos de proteção contra Path Traversal numa aplicação web que possui a tecnologia PHP.

1º Verificar o PHP.ini

Verificar se a função `allow_url_include` do `PHP.ini` está definida como **Off**.

2º validar o dado de entrada via GET

Se o dado que passa pelo parâmetro GET é inteiro, então sempre podemos validar se a variável é **int** ou se possui uma string.

Também podemos definir a função `htmlspecialchars` do PHP para sanitizar injeções externas de scripts e `dirname` para mitigar vulnerabilidades de **Path Traversal**.

```
if (isset($_GET['id'])){
    $id = htmlspecialchars($_GET['id']);
    $id = dirname($id);
    $id = (int)$_GET['id']; // ➔ Validando se o valor é inteiro
```

Validar se o dado é numérico

```
4
5     $id = (int)$_GET["id"];
6
7     if (is_numeric($id)) {
8
```

Recomendo criar validação de dados de entrada manualmente. Abaixo, estou compartilhando um método muito eficaz e que contempla caracteres unicodes, hexadecimais, base64 e caracteres normais.

3º Remover a função include

Não utilize a função include do php para incluir arquivo dinamicamente e use mensagens pré-definidas, que oriente o seu usuário qual o próximo passo a ser realizado e não fique perdido.

```
else {  
  
/* or include value id*/  
  
//include($id);  
print "User not found!";  
  
}
```

4º Prepare a Query

Crie sua consulta usando nomes de parâmetros precedidos por dois pontos como espaços reservados

5º Crie a declaração preparada

```
$statement = $dbh->prepare($consulta);
```

6º Vincular os parâmetros à instrução preparada

Vincule seus parâmetros à consulta.

```
$statement->bindParam(':id', $var);
```

7º Fazer as consultas

```
$statement->execute();
```

8º buscar o resultado

```
$busca = $statement->fetchColumn();
```


15.0 PROTEÇÃO COM WAF MODSECURITY OU NAXSI

Nessa etapa serei bastante rápido.

Caso você tenha um WAF como o ModSecurity e deseje fornecer uma proteção adicional, recomendo adicionar regras para proteção à Path Traversal:

<https://github.com/SEC642/modsec>

Outro Web Application Firewall conhecido é o Naxsi, sua estrutura de regra é menor e mais simples, mas também muito eficiente durante os testes.

A regra está disponível em:

https://github.com/nbs-system/naxsi/blob/master/naxsi_config/naxsi_core.rules

Ambos Web Application firewall trabalharão como um paliativo, cujo intuito é impedir que requisições associadas a Path Traversal que possam surtir efeito no sistema web atacado. A regra pode ser visualizada na seção: Directory traversal IDs:1200-1299.

Ressaltando a melhor recomendação é a correção das vulnerabilidades, mencionado na seção 14.0 desse artigo.

16.0 SOBRE O AUTOR

Paper criado por Fernando Mengali no dia 04 de março de 2025.

LinkedIn: <https://www.linkedin.com/in/fernando-mengali-273504142/>