SÉRIE WEBAPP PARA PENTESTER E APPSEC

COMO CRIAR SCAN

SERVER-SIDE REQUEST FORGERY

O MANUAL PASSO A PASSO

de como criar seus próprios scripts para identificar e tratar vulnerabilidades

FERNANDO MENGALI

SUMÁRIO

INTRODUÇÃO

Nesse artigo, desenvolveremos uma ferramenta com a linguagem de programação Perl que identificará páginas de internet que possuem vulnerabilidades de SSRF a nível de código.

Primeiro, iremos apresentar o processo de **identificação manual da vulnerabilidade de SSRF a nível de código**, 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 SSRF, seja para um alvo específico ou vários alvos.

O conteúdo sobre como identificar vulnerabilidades de SSRF 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://souSSRFforge.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**.

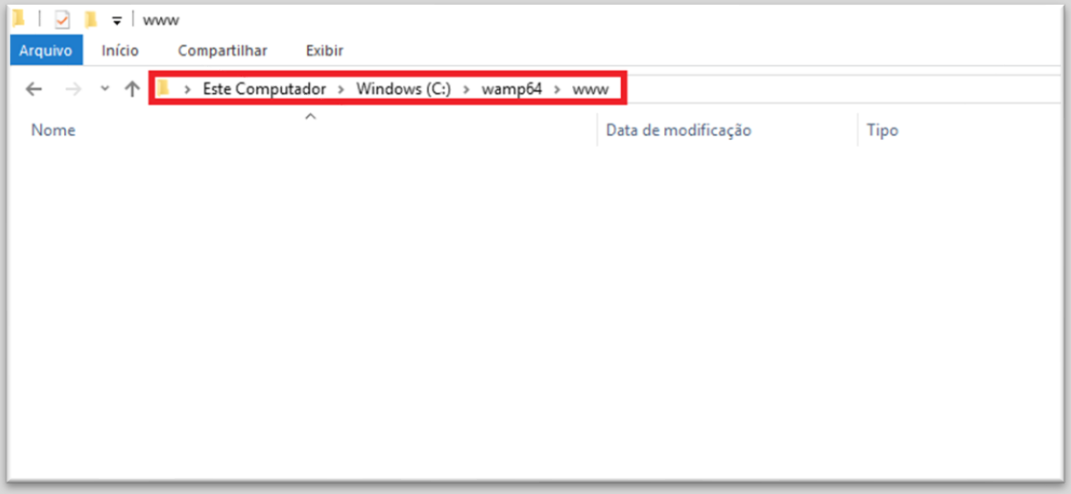
O processo de instalação é muito simples, portanto, não abordaremos.

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

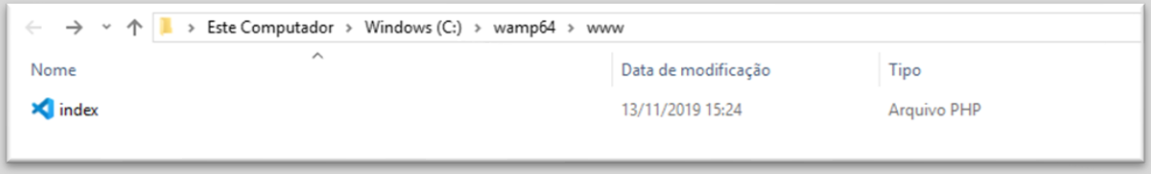
Se desejar acessar somente a seção sobre o desenvolvimento do scanning em Perl, acesse a **seção 8**.

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

Vamos criar uma página chamada index.php e adicionar ao nosso diretório wamp64/www do WAMP. Essa página será responsável por verificar se um endpoint está funcionando normalmente ou se possui algum erro.

```
<?php

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

    $id = $_GET['id'];

    if ($id == 80) {

        $content =
file_get_contents("http://192.168.0.10/index.php?id=".$id);

        if ($content === "OK") {

            print "Status: Online";

        }
        else {

            print "Status: Offline";

        }
    }
    else {

        $content =
file_get_contents("http://192.168.0.10/index.php?id=".$id);
        print $content;

    }

}

?>
```

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**”, do segundo ambiente, no meu caso, localhost.

5.1 A PÁGINA PHP COM O CÓDIGO VULNERÁVEL – SEGUNDO AMBIENTE

Agora, crie um segundo ambiente, mas em outra máquina, conforme ensinamos na seção **4.0 CRIANDO A PÁGINA PHP VULNERÁVEL** e certifique-se que a máquina do primeiro ambiente, no meu caso localhost esteja acessível a máquina do segundo ambiente, ou seja, 192.168.0.10.

O usuário precisará acessar a página index.php do enpoint e terá uma resposta sobre o status do endpoint, isto é, online ou um erro de conexão.

No mundo real, o verificador de endpoint ajuda no processamento de transações, evitando vulnerabilidade do tipo race condition.
É um sistema simples, limitado, inseguro e que possui falhas em filas de conexões, mas que muitas empresas podem adotar como forma de processar grandes quantidade de dados.

Abaixo apresentamos o código PHP vulnerável da página index.php que está armazenada no segundo ambiente, isto é, 192.168.0.10.
Abaixo está o código completo da página index.php do segundo ambiente:

```php
<?php

      if (isset($_GET["id"])) {

        $id = $_GET["id"];

        $v = file_get_contents("http://192.168.0.10:".$id);

          if ($v === "") {

            print "OK";

          } else {

            print $v;

          }
     }
?>
```

5.1.1 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"**, do segundo ambiente, no meu caso, IP 192.168.0.10

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.

5.2 ENTENDO O CÓDIGOS VULNERÁVEIS

Inicialmente, as duas páginas receberão um parâmetro GET:

```php
if (isset($_GET['id'])){

    $id = $_GET['id']; // not santization - vulnerable
```

5.1.2 Observe que em ambas as páginas temos if que não valida a existência de dados ou pârametros enviados para o método GET.
Se houver algum dado trafegando via GET ele entrará no bloco IF e será armazenado na variável **id**, mas observe que não existe nenhuma função para sanitização, abrindo um leque de opções estratégicas para testar novos ataques.

É importante observer que a ausência da sanitização dos dados inputados via GET é totalmente proposital, pois o intuito é entendermos como funciona a exploração de SSRF. Na penúltima seção vamos explicar o processo de sanitização, correção e proteção.

Vamos explicar a página index.php do primeiro ambiente, veja o exemplo de acesso a nossa página PHP via a url:

http://localhost/index.php?**id**=80

Após recebermos o valor via método GET, verificamos a conectividade do sistema, assim concluímos se está disponível para realizar transações e processar dados:

```
if ($id == 80) {

        $content =
file_get_contents("http://192.168.0.10/index.php?id=".$id);

        if ($content === "OK") {

            print "Status: Online";

        }
        else {

            print "Status: Offline";

        }
    }
    else {

        $content =
file_get_contents("http://192.168.0.10/index.php?id=".$id);
        print $content;

    }
```

5.0.3 Primeiro validamos se a porta que faremos conexão será a 80, depois fazemos uma conexão para o endpoint através da função file_get_contents que será responsável por criar a conexão com o endpoint, depois validamos se o endpoint está online ou offiline.
Se um usuário alterar a porta 80 para 22,21 ou qualquer outra, forçamos a conexão e se houver erro apresentamos o resultado para o usuário.

Se a conexão for concluída com sucesso, receberemos a resposta “OK” do endpoint, indicando que o endpoint está online e funcionando normalmente.

Vamos simplificar o código para melhor entendimento.
Se recebermos a resposta offline, indicará que o endpoint estará indisponível para processar transações, mas se tivermos um problema de conexão, receberemos a mensagem do erro. Abaixo, temos um simples código condicional em PHP que será responsável pela validação do status do endpoint.

```
        if ($content === "OK") {
```

```
            print "Status: Online";

        }
        elseif ($content === "offline") {

            print "Status: Offline";

        } else {

            print $content;
        }
    }
    else {

        $content =
file_get_contents("http://192.168.0.10/index.php?id=".$id);
        print $content;

    }
```

5.0.4 Bloco resposável por validar a conectividade.

O nosso endpoint, pode ser um simples arquivo index.php com a mensagem “OK”. O intuito é apenas termos um sistema vulnerável a SSRF, entendermos como identificar a vulnerabilidade, como criar uma ferramenta de automatização de identificação de SSRF e como corrigir.

5.3 ENTENDO O CÓDIGOS VULNERÁVEIS

Aqui vamos entender a página que criamos no segundo ambiente, de IP 192.168.0.10.

Esse IP será responsável por validar conectividade do endpoint e responder ao primeiro ambiente o status de conectividade.

Esse será o nosso código:

Primeiro recebemos a porta de conexão, no caso 80, se a reposta for vazia o endpoint está funcionando ou senão receberemos uma resposta de erro do endpoint:

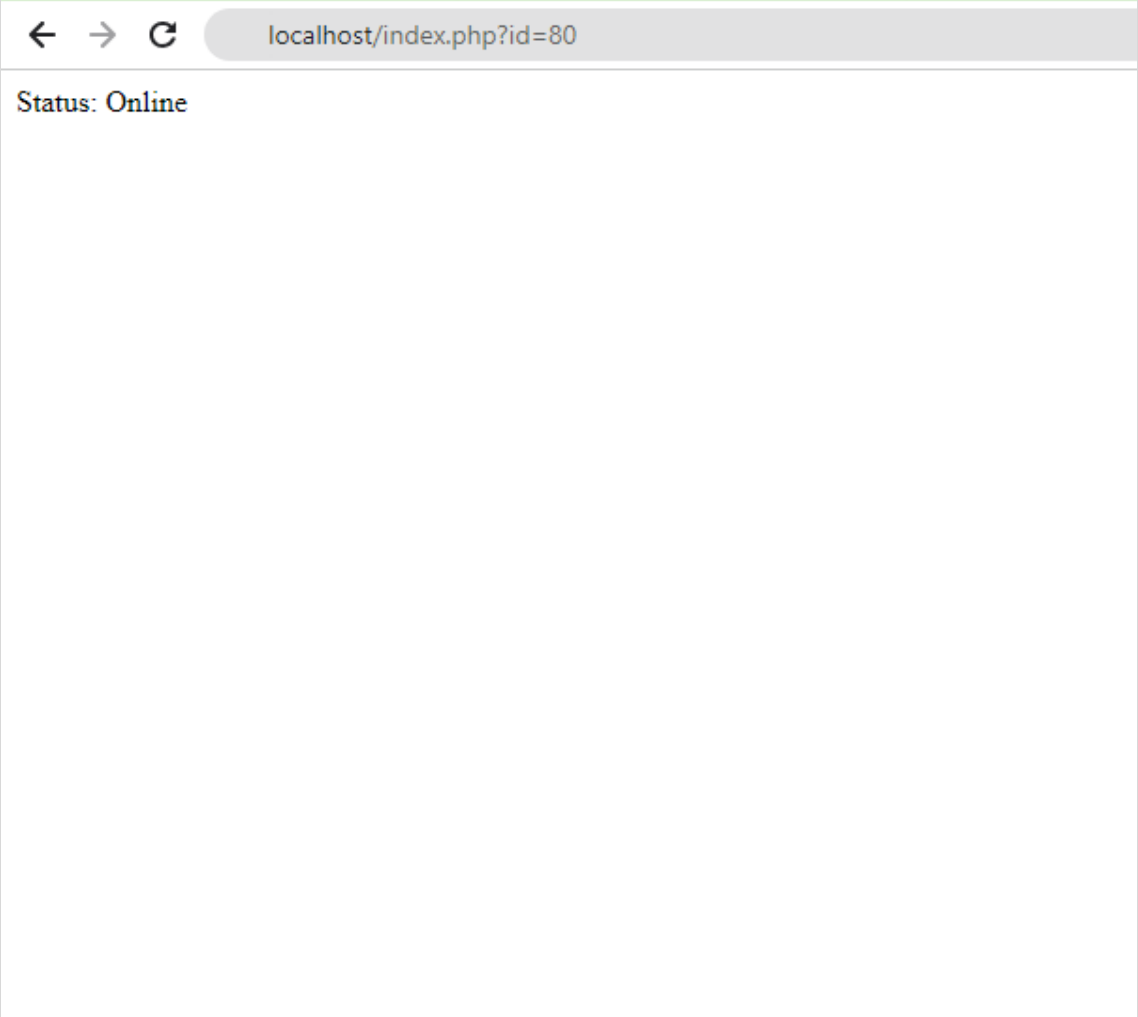
```php
<?php

if (isset($_GET["id"])) {


$id = $_GET["id"];

$v = file_get_contents("http://192.168.0.10:".$id);

if ($v === "") {

    print "OK";

} else {

    print $v;

}
}

?>
```

5.0.5 Bloco do endpoint do IP 192.168.0.10 resposável por validar se o endpoint está funcionando.

6.0 SITE VULNERÁVEL

Nessa seção acessaremos a página **index.php** vulnerável:
http://localhost/index.php?**id**=80

Veja o resultado no browser:



6.0.1 Recebemos o status de online, porque o sistema está funcionando, está no ar!

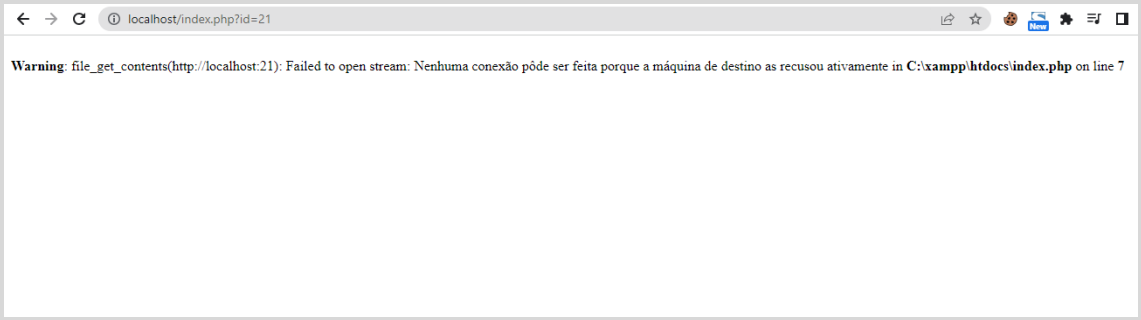
7.0 TEORIA: COMO DETECTAR SSRF

Para identificar o nosso alvo vulnerável, utilizaremos uma técnica simples, de injetar o endereço da placa de rede local do servidor: localhsot, seguido da porta que tentaremos conectar.
No cenário real, depois de descobrir a vulnerabilidade, podemos adicionar outros endereços de IPs internos com o intuito de descobrirmos hosts vulneráveis.
Na nossa URL de exemplo, a página **index.php** possui uma chave ou índice denominado **id** e um respectivo valor, 80.
O endereço localhost está associado ao endereço que faremos a conexão local.
Entendido o funcionamento para a exibição do status do endpoint, isto é, se ele está online ou offline, vamos entender na teoria como funciona a detecção de SSRF.

A estrutura da nossa URL será algo parecido como:

http://localhost/index.php?id =80

Se a aplicação possuir algum tipo de vulnerabilidade de SSRF, erros serão exibidos no browser, sendo o resultado similar ao apresentado abaixo:



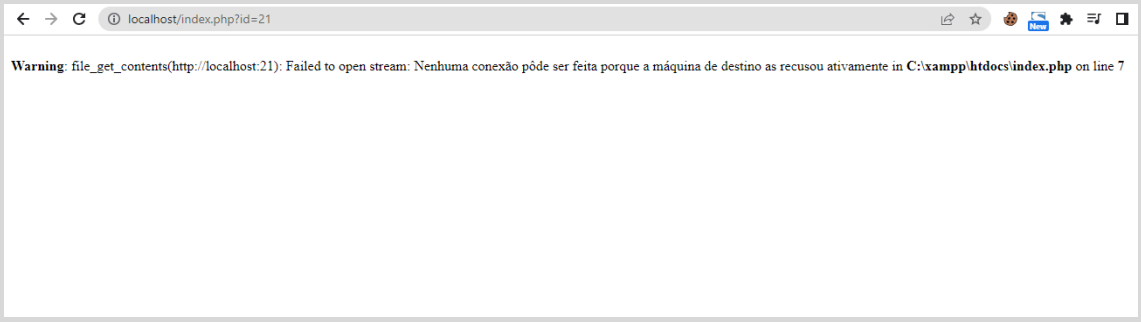
7.0.1 Resultado ao tentarmos acessar uma porta diferente do padrão.

O resultado acima com a mensagem de erro é o suficiente para entendermos que temos uma vulnerabilidade de SSRF.

8.0 PRÁTICA: DETECTANDO A VULNERABILIDADE DE SSRF

Agora vamos descobrir se a página está vulnerável a SSRF.
Para essa etapa, tentaremos acessar a porta 21 do servidor local, ou seja, o endereço localhost:21.

Vejamos:

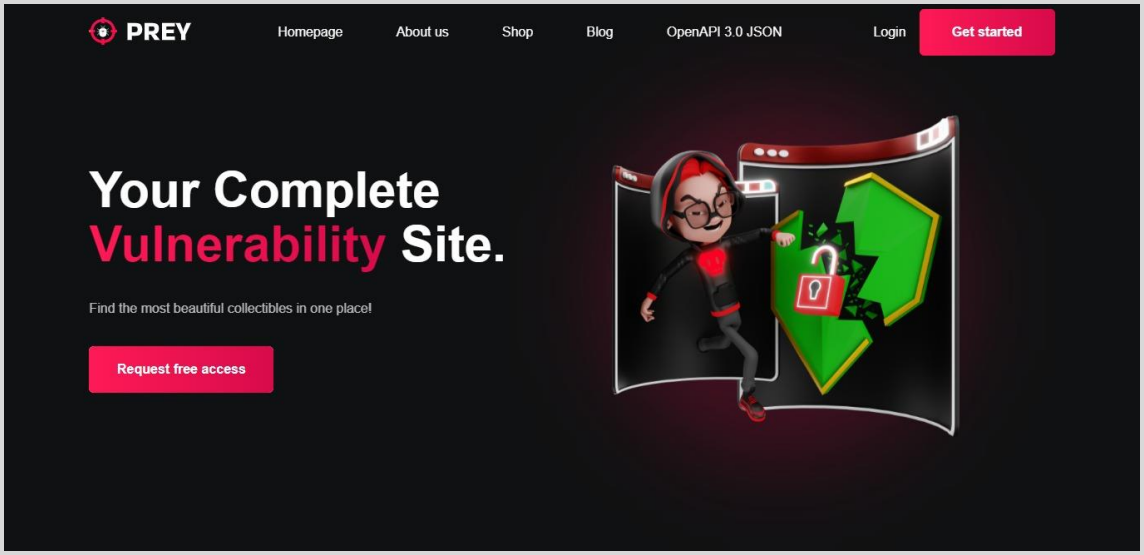


8.0.1 Observe a resposta de recusou ou refused em inglês, observe que a página está vulnerável a SSRF.

8.1 FRAMEWORK PARA TESTAR SSRF

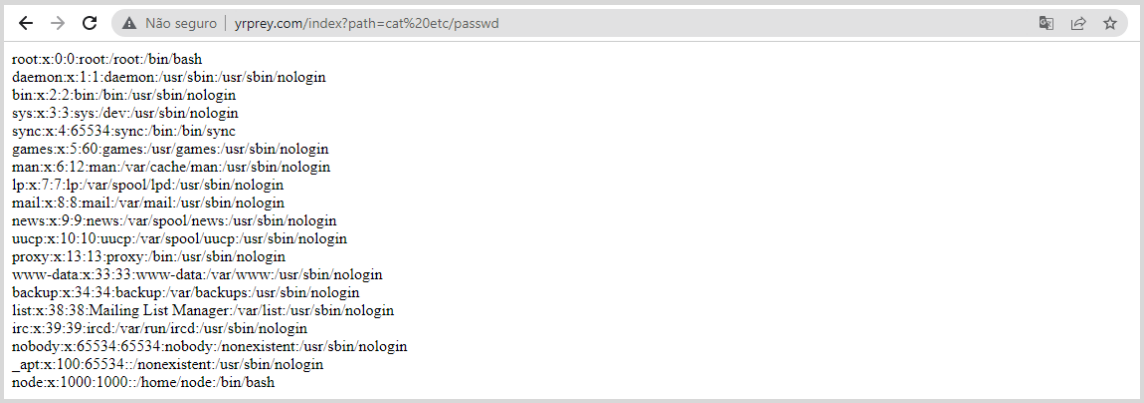
Caso você não queira criar um ambiente pronto, existem frameworks vulneráveis para interessados em aprender e aprimorar técnicas de invasão e mitigação de vulnerabilidades.

Um exemplo é o framework yrprey, totalmente gratuito e com x vulnerabilidades para ser explorado. Você pode testá-lo online pelo endereço: http://yrprey.com.



8.1.1 interface do yrprey.com até a elaboração desse artigo.

Abaixo, temos um exemplo de exploração da vulnerabilidade de SSRF no yrprey.com, para explorarmos a vulnerabilidade, utilizamos a mesma estratégia do nosso exemplo, mas existem estratégias diferentes e mais sofisticadas para identificar vulnerabilidades de SSRF. Veja o exemplo:



8.1.2 explorando a vulnerabilidade de SSRF na aplicação online.

Lembrando que até o desenvolvimento desse documento a vulnerabilidade de SSRF estava disponível no framework yrprey, mas pode ter tido atualizações, portanto, a vulnerabilidade de SSRF pode estar em outra path da aplicação vulnerável, por exemplo:

http://yrprey.com/verify/id=21

Caso queira replicar o ambiente online na sua máquina, você pode baixar a imagem completa e executá-la. A imagem do ambiente com a aplicação vulnerável está disponível para download em:

Docker/container => http://docker.com

VirtualBox/ snaphost => http://gihthu

Vmware/ snaphost => http://github

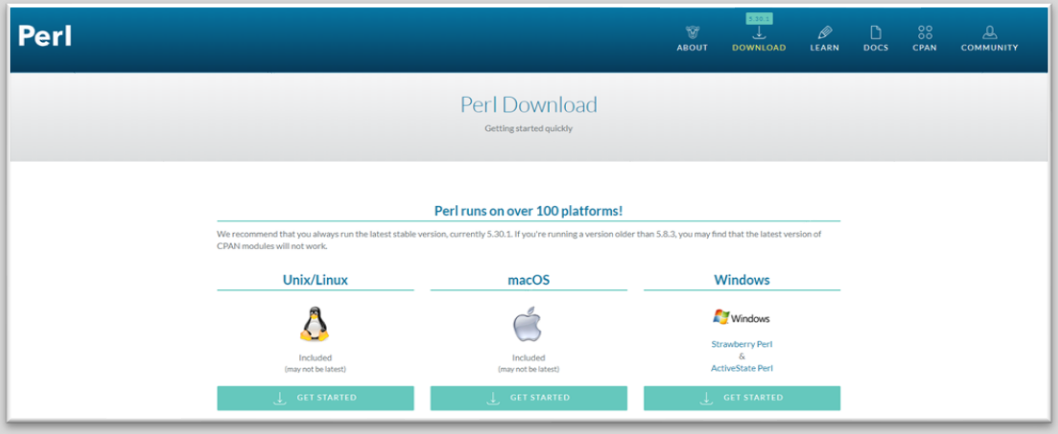
Você pode subir as imagens localmente na sua máquina e testar as vulnerabilidades de SSRF existentes.

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

9.0 CONSTRUÇÃO DO SCANNNING

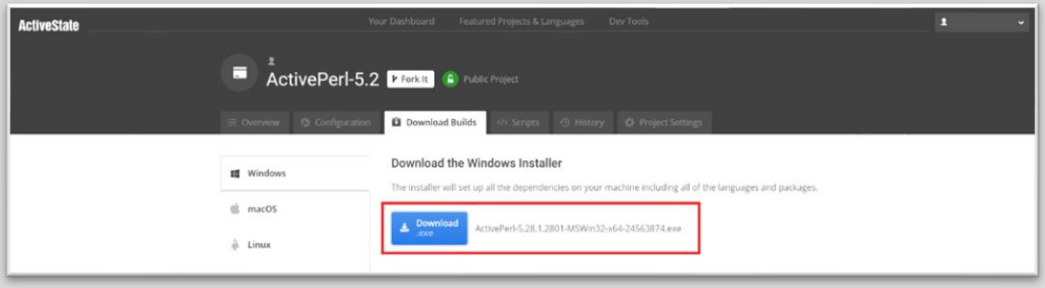
A linguagem de desenvolvimento escolhida para o desenvolvimento do script será o Perl. Você precisará de conhecimentos de programação em Perl, pois a ferramenta terá erros propositais, ou seja, apenas desenvolvedores, analistas de seguranças e interessados com aptidões de desenvolvimento entenderão melhor 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.

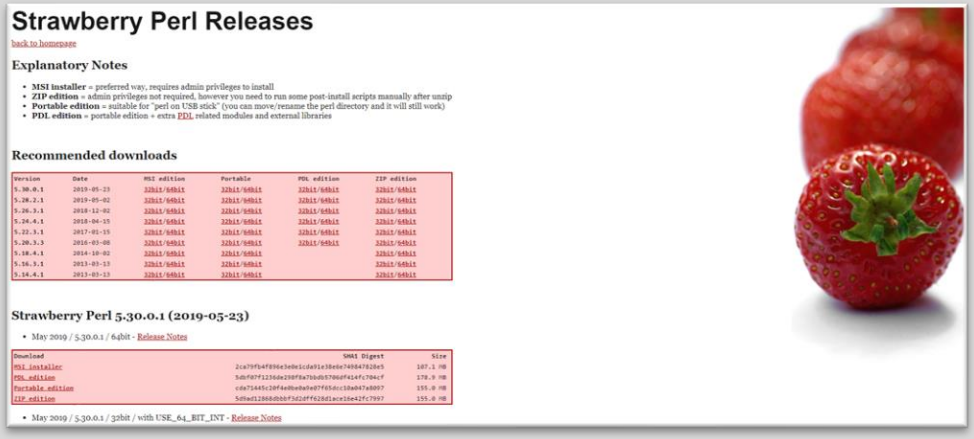
Depois de autenticado, você poderá baixar o Perl:



9.1.2 Clique no botão "**Download**".

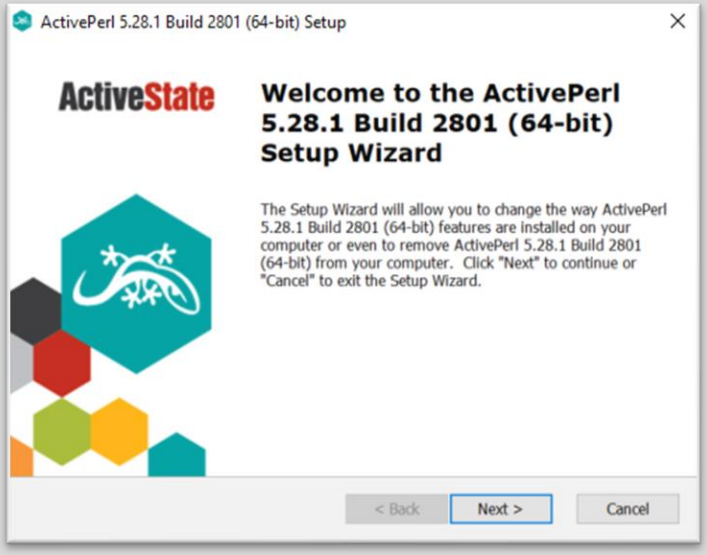
9.2 OPÇÃO 2: STRAWBERRY PERL PARA WINDOWS

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



9.2.1 Veja a página acima.

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

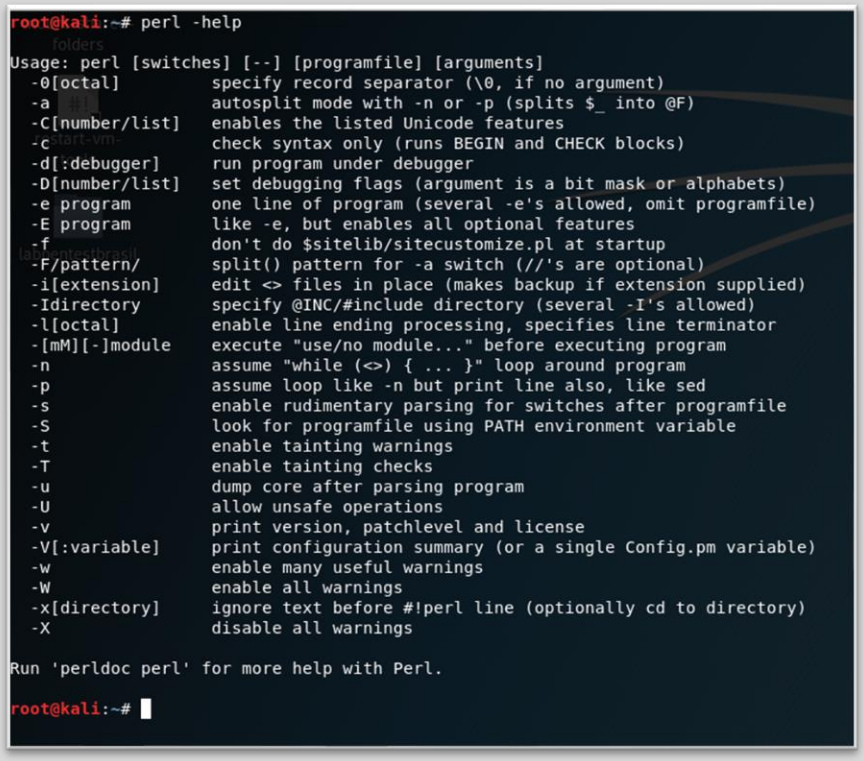


9.2.2 O restante você já sabe.

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:



11.0 CODIFICANDO A FERRAMENTA DE AUTOMAÇÃO

Nessa etapa iremos utilizar uma requisição para nossa página http://localhost/index.php?id=21.

Utilizaremos o endereço local e tentamos acessar a porta 21 do servidor.

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::Simple

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;
```

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** criarmos 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 Armazenando na variável $url temos o endereço que iremos acessar e validar a vulnerabilidade.

```
11
12  $url = $url."21";
13
```

11.3.3 Aproveitando a variável $url, vamos adicionar o payload com o comando type para ler o arquivo hosts ao final da 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 com um comando type e tentando 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    if ($resultado->content =~ /"refused"/) {
24
25      print "\n\n Vulnerable! \n\n";
26
27    }
28    else {
29      print "\n\n Not vulnerable! \n\n";
30    }
31
```

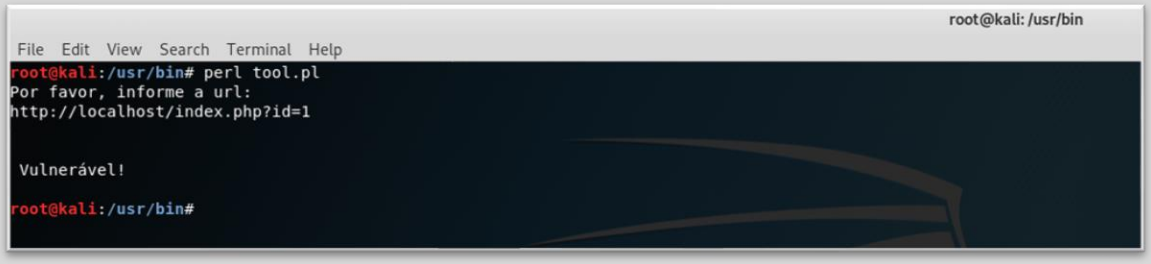
11.3.5 Podemos adicionar outros erros para serem comparados com o conteúdo de uma página.

Embora a ferramenta tenha a feature de identificação de vulnerabilidades de SSRF, pode não funcionar em determinados alvos, por causa de validação de headers como UserAgent, Cookies Custom ou outros tipos de Headers Customs, mas o conteúdo apresentado nesse documento é extremamente útil e importante para você aprender como criar seus primeiros scripts e aperfeiçoá-los.

Apenas precisa ser manipulado a tratamento do response.

11.5 EXECUTANDO O SCRIPT

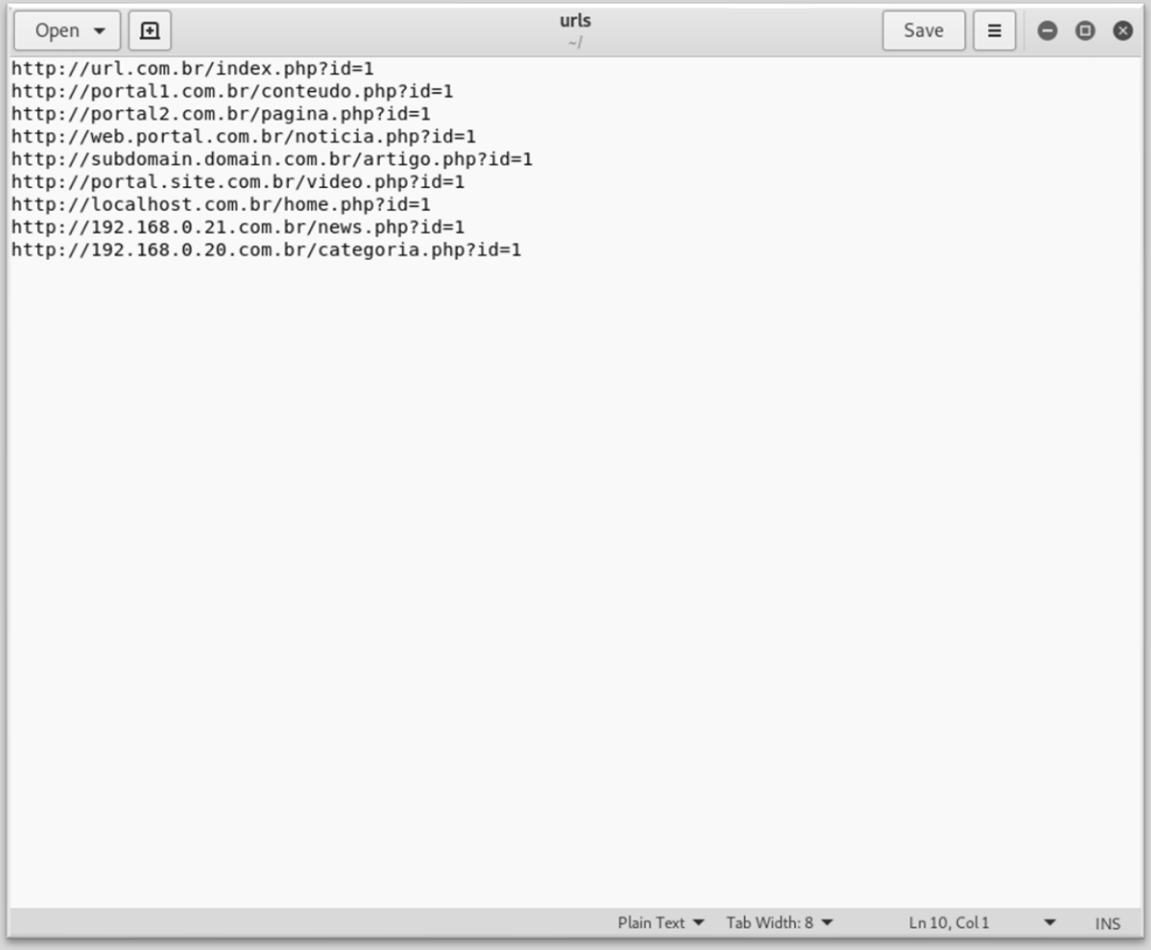
Esse é o resultado do script.



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:



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

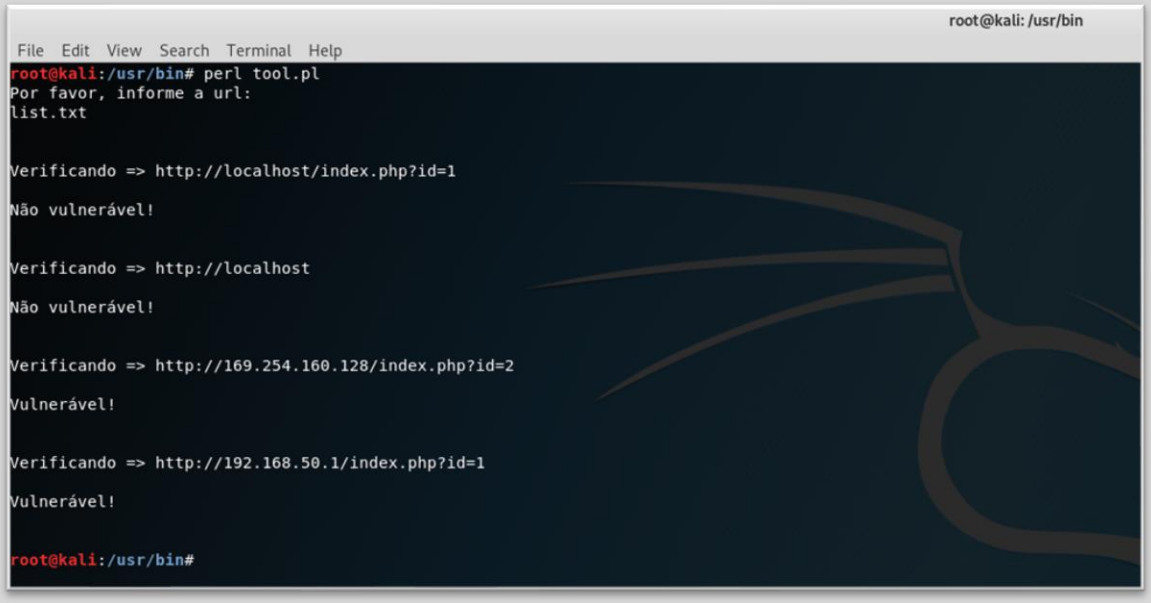
```
6
7     print "Por favor, informe a url: \n";
8     $url = <stdin>;
9
10  open ( URL, " $url" ) or die ( "Can't open file: $!");
11
12  @vector = <URL>;
13
14  $last = $#vector;
15
16  for ($i = 0; $i <= $last; $i++) {
17
18    print "verificando => " .vector[$i]."\n";
19
20    $url = $vector[$i].":21";
```

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**.
Na **linha 20**, temos o payload JavaScript para identificar a vulnerabilidade de SSRF.

```
21
22    my $requisicao=HTTP::Request->new(GET=>$url);
23
24    my $ua=LWP::UserAgent->new();
25    exit;
26
27    $ua->timeout(15);
28
29    my $resultado=$ua->request($requisicao);
30
31     if ($resultado->content =~ /"refused"/) {
32
33       print "\n\n Vulnerável \n\n";
34
35     }
36     else {
37       print "\n\n Não vulnerável! \n\n";
38     }
39
40   }
```

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



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 = $vector[$i].":21";

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

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

 $ua->timeout(15);

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

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

    print "\n\n Vulnerável \n\n";

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

13.2 A FERRAMENTA COM IMPLEMENTAÇÕES

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

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";
 $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].":21";

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

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

 $ua->timeout(15);

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

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

    print "\n\n Vulnerável \n\n";

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

}
```

14.0 CORRIGIR A VULNERABILIDADE

Nessa seção, demonstramos como evitar uma vulnerabilidade de SSRF na sua aplicação.

Pensamos no primeiro ambiente, o nosso localhost.

O nosso localhost possui a página que é responsável por fazer uma requisição para o segundo ambiente 192.168.0.10 e validar se o endpoint está online ou offline.

Nossa página do primeiro ambiente terá o seguinte código:

```php
<?php

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

    $id = $_GET['id'];
    $id = htmlspecialchars($id);

    if ($id == 80) {

      $ch = curl_init();

curl_setopt($ch, CURLOPT_URL, 'http://192.168.0.10/index.php?id='.$id);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

$result = curl_exec($ch);

        if ($result === "OK") {

            print "Status: Online";

        }
        else {

            print "Status: Offline";

        }
    }
    else {

        print "Porta de endpoint inválida";

    }

}
```

```
?>
```

Primeiro, podemos criar uma chamada para um endpoint com o intuito de validar a disponibilidade para processamento de transações.

O nosso front-end, pode buscar um endpoint estático para validar uma porta.

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

```
      if (isset($_GET['id'])) {

          $id = $_GET['id'];
          $id = htmlspecialchars($id);
```

Se o endpoint tiver uma plataforma padrão, como no nosso exemplo, podemos criar um bloco condicional para validar a porta, conforme no exemplo abaixo:

```
    if ($id == 80) {
```

Para fazermos uma requisição segura, podemos substituir file_get_contents, pois dependo de como utilizamos a função vulnerável abrimos brechas para outros tipos de vulnerabilidades, como Remote Code Execution.

Vamos adicionar a função curl para trabalhar com requisições para o nosso endpoint:

```
$ch = curl_init();

curl_setopt($ch, CURLOPT_URL, 'http://192.168.0.10/index.php?id='.$id);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

$result = curl_exec($ch);

        if ($content === "OK") {
```

```
            print "Status: Online";

        }
        else {

            print "Status: Offline";

        }
```

Nesse exemplo, recebemos a porta 80 e fazemos a requisição para o servidor endpoint de IP 192.168.0.10, cujo intuito é validarmos se o endpoint está funcional.

14.1 CORRIGINDO A VULNERABILIDADE

Agora, vamos entender o conteúdo de exemplo do nosso endpoint, localizado no endpoint 192.168.0.10:

```
<?php

if (isset($_GET["id"])) {

    $id = $_GET["id"];
    $id = htmlspecialchars($id);

        if ($id == 80) {

            $ch = curl_init();

            curl_setopt($ch, CURLOPT_URL,
'http://localhost/index.php?id='.$id);
            curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);

            $result = curl_exec($ch);

                if ($result === "") {

                    print "OK";

                } else {

                    print "offline";
```

```
                }
        }
        else {

            print "Porta inválida!!!";
        }

?>
```

O código identifica a porta que deseja fazer a conexão, se for a porta 80 o endpoint faz uma conexão para si mesmo e teremos o resultado se o endpoint está “funcional”.

14.2 NOTAS IMPORTANTES DO LABORATÓRIO

Os códigos apresentados nesses laboratórios são educativos, com o intuito de demonstrar a exploração de vulnerabilidades de SSRF.
O código, além de estar vulnerável a SSRF é possível explorar outras vulnerabilidades na aplicação.

Outras implementações que podem ser feitas no código é utilização de tokenização ou CSRF para validar cada conexão ao endpoint.

Nesse contexto, precisamos adotar uma arquitetura mais robusta, envolver uma equipe de Blue Team para trabalhar com rules White list, ou seja, envolver mais profissionais de segmentos diferentes para criar um ambiente verdadeiramente seguro.

15.0 SOBRE O AUTOR

Paper criado por Fernando Mengali no dia 08 de março de 2025.

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