



PHP & Segurança: Uma União Possível

v. 2.1 - Abril/2007



Objetivo:

Esta apresentação tem por objetivo apresentar técnicas para o desenvolvimento de aplicações seguras utilizando a linguagem PHP, eliminando os equívocos geralmente relacionados à esta.



Conceitos:

PHP – Uma linguagem vulnerável?

Um dos equívocos mais comuns relacionados à linguagem PHP é o de que é difícil de se produzir sistemas e ferramentas seguras quando a utilizamos.

Este equívoco é fundamentado em 3 conceitos:

- A simplicidade da linguagem, com uma baixa curva de aprendizado
- A própria fragilidade da web como ambiente de aplicações
- Algumas *features* presentes na linguagem



Conceitos: Simplicidade != Vulnerabilidade

O Problema:

Por ser uma linguagem de fácil aprendizado, é grande o número de pessoas que começam a carreira de desenvolvimento de aplicações utilizando PHP.

Esta falta de experiência freqüentemente resulta em aplicações de baixa qualidade e com problemas relacionados, especialmente, com segurança.

A Solução:

A comunidade, de forma geral, precisa amadurecer e possuir acesso a recursos que a tornem melhor. Eventos, artigos e a disseminação de boas práticas de desenvolvimento são apenas alguns dos caminhos que estão sendo trilhados para chegarmos a este objetivo.



Conceitos: Web: "A Terra de Ninguém":

O Problema:

A web é um ambiente precário em termos de segurança.

Dificuldade para identificar os usuários, ameaças que proliferam às dezenas e usuários descuidados são lugares-comuns quando tratamos de uma aplicação web.

A Solução:

Criar o hábito de pensar qualquer aplicação web, por mais simples que seja, levando-se em consideração a questão de segurança.

Termos a noção de que é nossa responsabilidade como desenvolvedores criar aplicações que sejam minimamente seguras.



Conceitos: Aplicações mais maduras:

O Problema:







Algumas features que podem ser utilizadas na linguagem, como a ativação da configuração *register_globals*, facilitam o trabalho, mas inevitavelmente aumentam riscos de segurança e acostumam mal o desenvolvedor.

A Solução:

Nos acostumarmos a desempenhar nosso trabalho da melhor forma possível, tendo sempre em mente que tudo o que é fácil tem seu preço e o que dá mais trabalho hoje causará menos incômodos amanhã.



Boas Práticas: Onde tudo começa

-  *register_globals* desligada, **sempre!**
-  *require* ao invés de *include*!
-  Filtrando a entrada
-  Erros são para a fase de desenvolvimento!
-  Coloque tudo em seu devido lugar!
-  Criptografia: **USE!**



Conceitos: *register_globals* desligada, sempre!

A configuração *register_globals* é, sem dúvida, uma das mais populares disponíveis na configuração do interpretador PHP. Sua utilização porém é uma péssima idéia, por dois motivos:

! Tira do programador a noção de origem dos dados.

! Tira do interpretador da linguagem a noção de origem dos dados.

i A configuração *register_globals* é tão polêmica que passou à ser desabilitada por *default* à partir da versão 4.2.0 e será eliminada na versão 6 da linguagem.



Prova de Caso #1: Ignorando a origem dos dados

Considere o seguinte código:

```
<?php
// Arquivo: sem_origem.php
if ($autenticado) {
    echo "Usuário autenticado, prosseguir...";
} else {
    echo "O usuário não está autenticado!";
}
?>
```

Ao desprezar a origem da informação a variável **\$autenticado** que tipicamente viria de uma sessão ou cookie, por exemplo, poderá ser completamente sobrescrita de forma absurdamente simples:

http://www.galvao.eti.br/sem_origem.php?autenticado=true



Solução: Reconhecendo a origem dos dados

Observe a mudança no código:

```
<?php
// Arquivo: sem_origem.php
if ($_SESSION['autenticado']) {
    echo "Usuário autenticado, prosseguir...";
} else {
    echo "O usuário não está autenticado!";
}
?>
```

Ao explicitar a origem da informação, torna-se muito mais difícil forjar a variável **\$autenticado**, pois para isso um cracker teria que forjar toda a origem desta primeiro. Ao receber a URL:

http://www.galvao.eti.br/sem_origem.php?autenticado=true

o interpretador da linguagem jamais confundiria esta informação, pois ela não faz parte do array `$_SESSION`, mas do array `$_GET`.



Prova de Caso #2: Ignorando a origem de um script

Considere o seguinte código:

```
<?php
// Arquivo: sem_trava.php
include_once("$DOCUMENT_ROOT/trava.php");

if ($trava) {
    die("Problemas de autenticação!");
} else {
    echo "Tudo OK!";
}
?>
```

Se estivermos com a diretiva *register_globals* ativada, nosso script pode ser facilmente desviado para outro presente em, por exemplo, /home/cracker/public_html:

http://www.galvao.eti.br/sem_trava.php?DOCUMENT_ROOT=%2Fhome%2Fcracker%2Fpublic_html



Solução: Comece fechando o caminho

A solução passa por várias ações, mas começa certamente pela não utilização da *register_globals*:

```
<?php
// Arquivo: sem_trava.php
include_once($_SERVER['DOCUMENT_ROOT'] . "/trava.php");

if ($trava) {
    die("Problemas de autenticação!");
} else {
    echo "Tudo OK!";
}
?>
```

Novamente explicitamos a origem do dado, impedindo que este dado seja forjado através de um caminho alternativo.



Boas práticas: *require* ao invés de *include*!

A utilização do comando `include` implica em uma séria questão de segurança: se o arquivo que está sendo solicitado ao servidor não existir ou se qualquer outro imprevisto cause a falha da inclusão do referido arquivo o script continuará normalmente sua execução, como se nenhum problema tivesse ocorrido.



A falha de um comando `include` não interrompe a execução do script!

Colocando em termos mais simples possíveis, você estará simplesmente jogando o seu script fora. Variáveis, funções e tudo o mais será completamente ignorado e o script que faria uso deste código continuará executando como se nada tivesse acontecido.



Prova de caso #3: o *include* falhou, o script não!

- 1) Considere que o arquivo `autenticar.php` barra o usuário em caso de problemas de autenticação e/ou autorização.
- 2) Considere o seguinte código, que confia nas rotinas presentes em **`autenticar.php`** para mostrar ou não informações sigilosas:

```
<?php
include_once("auteticar.php");

if (!isLoggedIn()) {
    die("Problemas de autenticação!");
} else {
    echo "Informações sigilosas do usuário";
}
?>
```

Se por qualquer eventualidade – como um erro de digitação por exemplo – o comando *include* falhar ainda assim o script continuará sendo executado normalmente, como se não houvesse problemas.



Solução: Se é importante não corra riscos!

A simples troca do comando *include* pelo comando *require* informa ao interpretador PHP – note o sentido da palavra – que o arquivo não está sendo simplesmente incluído, mas é **requerido** para que a aplicação funcione corretamente.

```
<?php
require_once("auteticar.php");

if (!isLoggedIn()) {
    die("Problemas de autenticação!");
} else {
    echo "Tudo OK!";
}
?>
```


A diferença – crucial, nesse caso – é que nosso script nem mesmo fará o teste condicional, ele será imediatamente interrompido na primeira linha, quando o arquivo não for encontrado.



Boas práticas: Não confie em dados de estranhos

Uma das principais técnicas de segurança envolve a filtragem de dados. A falta de filtragem é a principal causas de muitas vulnerabilidades, entre elas a famosa *SQL Injection*, ou injeção de SQL.

Quando trabalhamos levando em conta a filtragem de dados todo o dado é suspeito, ou contaminado, até que seja validado.

 **Jamais, em hipótese alguma, confie no usuário!**

Um dos equívocos mais comuns nesse sentido é quando tratamos esta “falta de confiança” como uma premissa de que o usuário é mal intencionado. Na realidade problemas de segurança podem também ser causados por erros e descuidos.



Prova de caso #4: Dados contaminados == SQL Injection!

1) Considere este código que chamaremos de mostraUsuario.php:

```
<?php
$conn = mysql_connect('servidor', 'usuario', 'senha');
mysql_select_db('meu_banco');

$sql = 'SELECT email FROM usuarios
      WHERE idusuario = ' .
      $_GET['userid'];

$recordSet = mysql_query($sql);

...
?>
```

2) Observe que não há filtragem do dado de entrada: `$_GET['userid']`

! Seu código aceitará qualquer informação vinda da query string!



Prova de caso #4: Dados contaminados == SQL Injection!

Desta forma é simples injetar código SQL na aplicação, pois qualquer dado passado para o script será aceito sem hesitação:

`http://www.galvao.eti.br/mostraUsuario.php?userid=198%20OR%20'x'='x'`

Após a conversão dos caracteres codificados (%20, ou seja, espaço em branco), esta é a query que será executada em nosso ingênuo script:

```
SELECT email FROM usuarios  
WHERE idusuario = 198 OR 'x' = 'x'
```

Como 'x' é sempre igual à 'x' a condição anterior terá sido invalidada pelo OR e o cracker terá em mãos todos os e-mails armazenados na base de dados.



Solução: Se a query aguarda um número, receba um número!

Quando nossa query espera receber um número a solução é simples: forçamos o dado à ser numérico mesmo que ele não seja, fazendo com que qualquer tentativa de injeção de SQL se transforme no inocente número 1:

```
<?php
$conn = mysql_connect('servidor', 'usuario', 'senha');
mysql_select_db('meu_banco');

settype($_GET['userid'], integer);

$sql = 'SELECT email FROM usuarios
      WHERE idusuario = ' .
      $_GET['userid'];

$recordSet = mysql_query($sql);


...
?>
```



Boas práticas: Longe dos olhos... dos usuários

Mensagens de erro são cruciais. É através dela que o trabalho de desenvolvimento se torna mais ágil, e são elas que nos apontam nossos pequenos deslizes.


O problema é que, quando mantemos as mensagens de erro visíveis para qualquer usuário, corremos o risco de passar informações importantes sobre o sistema e até mesmo sobre o servidor para qualquer usuário ler.

 Mensagens de erro frequentemente carregam informações comprometedoras!



Prova de caso #5: Informação demais!

Considere um erro de conexão MySQL típico:

 Warning: **mysql_connect()** [function.mysql-connect]:
Access denied for user '**foo**'@'**localhost**'
(using password: YES)
in **/usr/local/apache/htdocs/script.php** on line 2

Esta inocente mensagem revela 5 informações que só são úteis ao próprio desenvolvedor ou à um cracker:

- 1) O tipo de banco utilizado: **mysql**
- 2) O usuário de conexão: **foo**
- 3) Que este usuário está tentando se conectar de dentro do próprio servidor: **@localhost**
- 4) O path absoluto em disco do script: **/usr/local/apache/htdocs**
- 5) O nome do script: **script.php**



Solução: Se está em produção, grave um log!

Três diretivas de configuração serão responsáveis por esconder os erros e gravá-los em um arquivo de log:



- 1) **display_errors**: `off` ou `0`
- 2) **log_errors**: `on` ou `1`
- 3) **error_log**: `/caminho/para/arquivo.log`

A primeira desativa a exibição dos erros.

A segunda ativa a gravação de erros em arquivo.

A terceira diz que arquivo será esse que receberá as mensagens.



Boas práticas: Longe do browser

O servidor web, e conseqüentemente o browser, não precisa acessar todos os arquivos que você possui na sua aplicação, especialmente se este arquivo não gera saída HTML ou se serve apenas de arquivo de configuração.



Se o Browser enxerga, qualquer um enxerga!



Prova de caso #6: Entregando o ouro!

Considere o seguinte arquivo de configuração típico, que chamaremos pelo enigmático nome de config.php:

```
<?php
$dbserver = 'localhost';
$dbtype= 'mysql';
$dbuser = 'foo';
$dbpass = 'bar';
?>
```

Se este arquivo estiver gravado em um diretório acessível ao servidor web, ele estará acessível à um script PHP que lerá se ele gera saída de informações, por exemplo:

```
<?php
file_get_contents('http://www.galvao.eti.br/config.php');
?>
```



Solução: Esconda o que não precisa ser visto!

Contanto que as permissões (*chmod/chown*) sejam setadas corretamente, o arquivo PHP pode tranquilamente ser colocado em um diretório longe da raiz utilizada pelo servidor web.



O que não está na raiz web não pode ser acessado pelo servidor web e nem pelo browser.



Boas práticas: 97c6105c1d97d600ec16ab4abace6d4c

Dados sigilosos são chamados assim por um motivo. Quando tratamos especificamente de senhas é impressionante a quantidade de aplicações web que grava senhas em texto puro na base de dados.



Senhas são **personais, intransferíveis e confidenciais! Ninguém além do próprio usuário deve ter acesso à senha!**

PHP implementa criptografia das mais variadas formas, **pesquise, aprenda e use!**

- MD5
- SHA1
- mcrypt

etc...



Sobre o autor:

Er Galvão Abbott trabalha há mais de dez anos com programação de websites e sistemas corporativos com interface web.

Autodidata, teve seu primeiro contato com a linguagem HTML em 1995, quando a internet estreava no Brasil.

Além de lecionar em cursos, escrever artigos e ministrar palestras, tem se dedicado ao desenvolvimento de aplicações web, tendo nas linguagens PHP, Perl e JavaScript suas principais paixões.

É o fundador e líder do UG PHP RS, além de trabalhar como consultor e desenvolvedor para uma empresa norte-americana da área de segurança.



Contatos:

Contatos com o autor:

Web:

<http://www.galvao.eti.br>

<http://blog.galvao.eti.br/>

<http://www.phprs.com.br>



E-mail:

galvao@galvao.eti.br

IM:

ergalvao@hotmail.com (MSN)

er.galvao@gmail.com (GoogleTalk)

