por Pablo Dall'Oglio

Protegendo códigos em PHP-GTK

Este artigo apresenta algumas formas simples de proteger o acesso ao código-fonte de aplicações desenvolvidas em PHP-GTK.

O PHP é uma linguagem que nasceu na Web e é neste ambiente que é mais adotado. O ambiente Web é regido pela arquitetura cliente-servidor, em que há o código do programa interpretado pelo servidor e o resultado de sua execução. Normalmente nesse modelo, o resultando é código HTML para ser visualizado no navegador do cliente. Neste caso, a própria arquitetura cliente-servidor impede o usuário de ter acesso ao código-fonte do programa, a menos que a pessoa tenha acesso ao servidor da aplicação.

Quando desenvolvemos uma aplicação gráfica em PHP-GTK, rodamos a aplicação de forma local, utilizando recursos da máquina cliente, da mesma forma que em aplicações em Delphi ou Visual Basic. Mas, no caso do PHP-GTK, o código da aplicação é interpretado diretamente pelo executável do PHP. Desta forma, o código-fonte fica exposto ao cliente, caso ele queira investigá-lo.

O objetivo de se proteger um código em PHP-GTK pode ser comercial, impedindo que o cliente tenha acesso indevido ao código-fonte e realize cópias piratas, mas também pode evitar que um usuário "curioso" investigue o código do mesmo e interfira no seu funcionamento, tomando conhecimento de senhas de acesso ao banco de dados, por exemplo. As técnicas que iremos estudar se aplicam tanto para a utilização com PHP-GTK, quanto para utilização no desenvolvimento de uma aplicação PHP Web.

1. O programa

Para demonstrar as técnicas de proteção de código, em primeiro lugar vamos construir uma pequena aplicação a ser protegida, chamada "app.php".

A nossa aplicação de exemplo é extremamente simples. Ela está toda contida em uma classe chamada *Application*. A classe *Application* na verdade é filha da classe *GtkWindow*, logo, pelo mecanismo de herança, ela é uma janela.

Dentro da janela *Application*, criaremos uma caixa vertical (*GtkVBox*) e dentro desta caixa vertical colocaremos dois *widgets*. O primeiro ($this->nome) é um objeto *GtkEntry* para digitação de um nome qualquer. Já o segundo, ($botao) é um botão que ao ser clicado executará o método *onClick*().

A tarefa do método *onClick* é exibir uma janela de mensagem com o nome digitado pelo usuário.

```
<?php
class Application extends GtkWindow
{
private $nome;
      /*
      * método construtor
      * cria a janela e os campos
      */
      function __construct()
      {
            parent::__construct();
            $this->set_title('Aplicação');
            // cria uma caixa vertical
            $vbox = new GtkVBox;
            // campo para digitação do nome
            $this->nome = new GtkEntry;
            // cria o botão
            $botao = new GtkButton('Exibe');
            // conecta o botão à ação
            $botao->connect('clicked',
            array($this, 'onClick'));
            // adiciona os campos na vbox
            $vbox->add($this->nome);
            $vbox->add($botao);
            // adiciona a vbox na janela
            $this->add($vbox);
      }
      /*
      * método onClick
      * Exibe o nome digitado pelo usuário
      */
      function onClick()
      {
      $dlg =new GtkMessageDialog(null,
      Gtk::DIALOG_MODAL,
      Gtk::MESSAGE_INFO, Gtk::BUTTONS_OK,
      $this->nome->get_text());
      $dlg->run();
      $dlg->destroy();
      }
}
?>
```

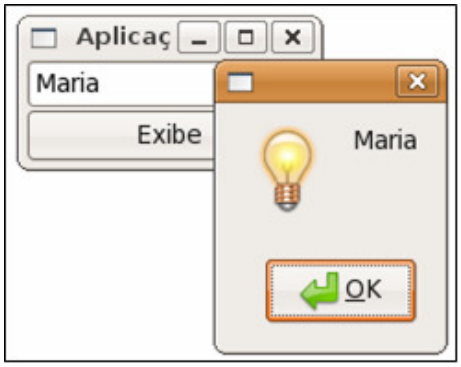

Figura 1 - Aplicação de exemplo

Agora, para executar tal aplicação, teremos de escrever um segundo programa para incluir a classe "app.php" e, então, instanciar um objeto desta classe e executar seu método *show_all*(), para exibir a janela.

```
<?php
 // inclui a classe
 include_once 'app.php';
 // instancia um objeto
 $app=new Application;
 // exibe a janela
 $app->show_all();
 Gtk::Main();
?>
```

2. Compactação

A forma mais simples de se proteger o código de um programa é compactando-o. Este mecanismo protege o acesso ao programa de grande parte dos usuários, mas é facilmente reversível, tendo em vista que basta o usuário final da aplicação descobrir que ele deve descompactar um arquivo para ter acesso ao seu código-fonte.

Para compactar a aplicação, iremos utilizar o aplicativo GZ do Linux, que compacta o programa "app.php", gerando um outro, chamado "app.php.gz".

```
# gzip app.php
```

Agora, para utilizar o programa ao invés de simplesmente executarmos o comando *include_once* sobre o arquivo compactado, precisamos indicar qual o *wrapper* utilizado para sua interpretação. Neste caso, o *wrapper* é "compress.zlib". O arquivo é descompactado instantaneamente e a classe fica disponível.

```
<?php
// inclui a classe compactada
include_once 'compress.zlib://app.php.gz';
// instancia um objeto
$app=new Application;
// exibe a janela
$app->show_all();
Gtk::Main();
?>
```

3. Criptografia Simples

Podemos melhorar um pouco mais o algoritmo anterior, criptografando o código da aplicação além de compactá-la. Para tal, iremos escrever um pequeno programa "empacotador" da aplicação. Este programa irá ler o código da aplicação por meio da função *file_get_contents*(), para então remover as tags de início e fim do <?php?> com a função *substr*(). Após, iremos codificar o fonte do programa com a função *base64_encode*(). Note que aqui poderíamos estar utilizando métodos mais poderosos de criptografia. Ao final, gravamos o código compactado e criptografado no arquivo "app.php.cri".

```
<?php
$code = file_get_contents('app.php');
$code = substr(trim($code),6, -2);
$code = base64_encode($code);
file_put_contents('compress.zlib://
app.php.cri', $code);
?>
```

Agora, para podermos executar tal programa criptografado, temos de escrever outro programa. Este programa irá ler o conteúdo do arquivo criptografado, chamado "app.php.cri", descriptografá-lo e executar este código por meio da função *eval*(). Após, o funcionamento do programa continua o mesmo.

```
<?php
// inclui a classe criptografada
$code =
file_get_contents('compress.zlib://
app.php.cri');
eval(base64_decode($code));
// instancia um objeto
$app=new Application;
// exibe a janela
$app->show_all();
Gtk::Main();
?>
```

Você deve ter percebido que as duas técnicas demonstradas até aqui são bastante simples. Da mesma forma que são simples de utilizar são simples de serem desfeitas, uma vez que o usuário terá acesso ao código-fonte que irá descriptografar e executar o código fonte. Desta forma, facilmente utilizamos um comando *echo*() no lugar do *eval*() para exibirmos o código-fonte em tela. Claro que nenhum usuário final faz isto, mas qualquer desenvolvedor com conhecimentos básicos em PHP pode fazer isto.

4. bcompiler

O *bcompiler* é uma extensão do PHP que faz parte do PECL (*The PHP Extension Community Library*), que é um repositório de extensões comunitárias para PHP, escritas em linguagem C. Funciona assim como o PEAR (*PHP Extension and Application Repository*), que é um repositório de extensões escritas em PHP.

O *bcompiler* foi escrito com o objetivo de codificar arquivos, classes ou funções por completo em uma aplicação proprietária.

De acordo com o autor da *bcompiler*, Alan Knowles, a utilização da extensão pode melhorar o desempenho em até 30%, embora o código codificado seja bem maior em termos de tamanho físico ocupado pelo arquivo em

relação à aplicação normal. Em termos de proteção de código, é seguro dizer que é impossível recriar o exato código-fonte original.

Para utilizar a *bcompiler*, primeiro devemos instalá-la. Alguns pré-requisitos devem ser satisfeitos em alguns ambientes. No caso do sistema operacional Ubuntu, será necessário instalar algumas bibliotecas relacionadas à compactação de arquivos (bzip2).

```
# apt-get install libbz2-dev bzip2
```

Após isto, podemos proceder para a instalação da *bcompiler* em si. Para tal, utilizamos o binário "pecl", que acompanha a instalação do PHP. Para tal, digitamos o seguinte comando a partir do diretório onde os binários do PHP estão instalados.

```
# <php>/bin/pecl install bcompiler-0.8
```

Agora que temos a *bcompiler* instalada, teremos de escrever dois programas. O primeiro programa irá gerar os *byte-codes* e o segundo irá interpretá-los. Para distribuir o programa ao usuário final, precisaremos distribuir apenas os *byte-codes* e o programa interpretador.

Este primeiro programa listado a seguir é o codificador, sua função é ler o conteúdo de nosso programa "app.php", codificá-lo e gravar o conteúdo codificado dentro de um arquivo chamado "compiled.phpb", que além de codificado será também compactado. Para codificar o arquivo, utilizamos a função *bcompiler_write_file* (), além de outras com o objetivo de criar o cabeçalho e rodapé do arquivo.

```
<?php
// carrega a extensão bcompiler
dl('bcompiler.so');
// define o arquivo a ser gravado os byte-
codes
$fh = fopen("compress.zlib://
compiled.phpb","w");
// escreve o cabeçalho do arquivo
bcompiler_write_header($fh);
// codifica o arquivo 'app.php'
bcompiler_write_file($fh, 'app.php');
// escreve o rodapé do arquivo
bcompiler_write_footer($fh);
fclose($fh);
?>
```

Após rodarmos o programa codificador, o arquivo "compiled.phpb" será gerado. O conteúdo do arquivo codificado se parecerá em muito com o trecho listado a seguir, ou seja, totalmente ilegível.

```
80><8a>^C2.Ï^U^P^PqAA^P^Q^EÄ^E<9
Ñ5òß<94>^Lø^Z`M^[§h^A½8E`<9f>^T!
PÎ¨ôZl<88>LÊM<83>¯,6¡ª^]<90>9B<9
<88><82>ç²<99>TÀ»¯!`
c¶í6^B<9e>4ö3 c<8d><9a>^Bõà½yz
ê¤Ûêôó,£4ÝÕ'^SõÙÃëÃ/Eô©Í-<98><89
\+4@+¤¥pVàWcVhÀ-H~O<8c>Øêðõ¯DñÉÕ
>7M^NÔL® <99>7<82>ì7<86>3^Zu¦^U
```

Após codificarmos nosso programa, podemos escrever o programa interpretador que será distribuído juntamente com os *byte-codes* (compiled.phpb). O programa interpretador irá carregar a extensão "bcompiler.so", que disponibiliza as funções da *bcompiler*, abrir o arquivo de *byte-codes* por meio da função *fopen*() e interpretar o arquivo codificado por meio da função *bcompiler_read*(). Note que esta função disponibiliza para aplicação exatamente o mesmo código que foi codificado. Desta forma, o programador terá neste momento acesso à classe "*Application*", que faz parte do arquivo original "app.php" que foi codificado. Após executarmos a *bcompiler_read*(), fechamos o arquivo e então já podemos instanciar objetos da classe *Application*. No final do programa instanciamos um objeto e exibimos ele em tela pelo método *show_all*(). Neste momento o programa será executado da mesma forma que em nosso primeiro exemplo onde interpretávamos diretamente o código-fonte.

```
<?php
// carrega a bcompiler
dl('bcompiler.so');
// abre o arquivo compactado
$fh = fopen("compress.zlib://
compiled.phpb","r");
// lê os byte-codes
bcompiler_read($fh);
// fecha o arquivo
fclose($fh);
// intancia o objeto
$app=new Application;
// exibe a janela
$app->show_all();
Gtk::Main();
?>
```

Considerações finais

Neste artigo estudamos algumas técnicas muito simples de se proteger o acesso ao código-fonte até abordarmos a *bcompiler*, uma extensão bastante complexa e poderosa do PHP. Sabemos da existência de diversas ferramentas comerciais com o objetivo de proteger ou ofuscar códigos em PHP. Mesmo assim, focamos este artigo em soluções simples, livres e que funcionassem não apenas no ambiente Web, mas também para quem desenvolve em PHP-GTK.

Referências e links sugeridos

[PHP-GTK Brasil] – http://www.php-gtk.com.br
[Livro PHP-GTK] – http://www.php-gtk.com.br/book
[Site do Autor] – http://www.pablo.blog.br
[Site do PECL] – http://pecl.php.net
[Site bcompiler] – http://www.php.net/bcompiler

Pablo Dall'Oglio - pablo@dalloglio.net
Pablo Dall'Oglio é bacharel em Informática pela UNISINOS. Autor dos livros "PHP Programando com Orientação a Objetos" e "PHP-GTK Criando Aplicações Gráficas com PHP", pela editora Novatec. Também foi o criador do PHP-GTK Brasil. Atualmente, é diretor de tecnologia e proprietário da Adianti Solutions (www.adianti.com.br).