



Segurança da Informação:

Tratando dados em PHP



Objetivo:

O objetivo desta palestra é demonstrar os riscos inerentes de se trabalhar com informações externas à aplicações desenvolvidas em PHP, como o descuido pode tornar uma aplicação vulnerável e apresentar conceitos de filtragem, tratamento e limpeza de dados.

Tópicos

- Conceituando dados;
- Aplicando conceitos no desenvolvimento;
- Riscos presentes em dados não tratados;
- Filtragem e tratamento;
- Referências.



Importância:

Filtragem e tratamento de dados é uma questão importantíssima e obrigatória porque a falta de rotinas que implementem estas questões é a causa das principais vulnerabilidades de aplicações:

- *SQL Injection*
- *Cross Site Scripting (XSS)*
- *Cross Site Request Forgeries (CSRF)*



Conceituando dados:

Um dado ou informação é tudo o que é externo à aplicação, seja por ser produzido por um meio externo para que esta processe ou produzido por ela mesma para uso posterior. A aplicação utiliza dados para processamento, geração de resultado e/ou comunicação com outros sistemas.

Dados geralmente possuem duas características:

- Tipo: O tipo de dado esperado pela aplicação (presumido) e o tipo do dado de fato (real). Exemplos: *strings*, números, etc...;
- Direção: se este dado está sendo informado para a aplicação (*input*) ou gerado por ela mesma (*output*).



Ao presumir que um dado será informado para a aplicação com um tipo definido corre-se o risco de, ao receber um dado de tipo diferente, a aplicação seja comprometida. Observe o seguinte exemplo:

```
$sql = "SELECT email FROM usuarios WHERE id_usuario=$dado_externo";
```

Onde, em situações normais, ao receber o dado externo a aplicação monta *queries* como estas:

```
$sql = "SELECT email FROM usuarios WHERE id_usuario=106";
```

A falha presente nesta aplicação é uma das mais comuns encontradas no mercado: excesso de confiança e ingenuidade, ou seja, **o programador confia em um dado que não é controlado pela aplicação.**



Como nossa aplicação não faz nenhuma espécie de tratamento da informação antes de processá-la, torna-se brincadeira de criança para um *cracker* enganar a aplicação para que ela assuma um comportamento diferente do esperado e traga o resultado que **ele** quer:

```
$sql = "SELECT email FROM usuarios WHERE id_usuario=0 OR 'x'='x';"
```

Observe como a perversão do tipo de dado (agora a aplicação está recebendo uma *string* ao invés de um número inteiro), perverte todo o conceito da aplicação, tornando-a uma ferramenta de coleta de endereços de e-mail de toda a base cadastrada.



A solução neste caso é forçarmos o tipo presumido do dado que será recebido por nossa aplicação, de forma que ele se torne, de fato, o tipo correto e dessa forma torne a aplicação mais coerente e segura:

```
settype($dado_externo, 'integer');  
$sql = "SELECT email FROM usuarios WHERE id_usuario=$dado_externo";
```

Ao forçar a conversão do tipo de dado, o resultado será diferente para casos diferentes:

- Caso o dado possua o tipo correto ele não será alterado:

106 => 106

- Caso o dado possua um tipo diferente, mas o tipo correto no início, apenas a porção que contém o tipo correto será considerada:

106 or 'x'='x' => 106

- Em todos os demais casos, o dado será convertido para o inteiro 0 (zero):

'galvão' or 1=1 => 0



A direção de um dado e conseqüentemente sua origem e/ou destino, é uma característica importantíssima à se levar em conta. Esta importância se deve aos seguintes fatores:

- Forma de entrada de dados: É muito mais simples para um *cracker* forjar um dado passado por *query string* do que um dado passado por uma sessão ou formulário que envia informações pelo método POST, por exemplo.
- Dependendo do destino de um dado após o seu processamento ele deverá receber um tratamento específico antes de ser gravado. Exemplos comuns:
 - Banco de Dados: Deve-se fazer o escape de caracteres específicos de forma à evitar ataques de SQL *Injection*.
 - Gravação de dados para posterior exibição: Deve-se fazer o escape de caracteres diferentes, de forma à se evitar ataques de XSS e CSRF.



Direção e, especialmente origem da informação tem sido uma questão polêmica quando falamos especificamente da linguagem PHP. Isto ocorre pela famosa configuração *register_globals*.



A configuração *register_globals* faz com que o interpretador da linguagem converta, automaticamente, índices de informações que um *script* está recebendo em variáveis, independentemente da origem da informação.



Considere um formulário HTML típico, onde há um campo oculto chamado 'codigo':

```
<input type="hidden" name="codigo" value="a983kjas208u762od60pcn1256jsg0il" />
```

Quando consideramos trabalhar com a configuração *register_globals*, optamos pelo **fácil**, pelo que é **menos trabalhoso** e, equivocadamente, mais produtivo:

```
if (in_array($codigo, $codigos_validos)) {  
    //Seu código aqui  
} else {  
    die('Código inválido!');  
}
```

A falha, uma vez mais, é o excesso de confiança e ingenuidade: esperamos que a origem da informação seja sempre e inequivocadamente correta.



Ao desconsiderar a importância da origem da informação, mantemos no caso da configuração *register_globals* estar ativada, diversas portas desnecessariamente abertas para que o dado nos seja enviado. Torna-se, uma vez mais, fácil forjar o dado e “enganar” nossa aplicação:

<http://www.galvao.eti.br/script.php?codigo=askjd2387skdjs118292skdj>

Confiando apenas no índice da informação à ser recebida, passamos a aceitar esta informação mesmo que ela venha de uma fonte inválida.

```
if (in_array($codigo, $codigos_validos)) {  
    //Seu código aqui  
} else {  
    die('Código inválido!');  
}
```



A solução torna-se, pura e simplesmente, explicitar a origem da qual o dado deve, obrigatoriamente, proceder:

```
if (in_array($_POST['codigo'], $codigos_validos)) {  
    //Seu código aqui  
} else {  
    die('Código inválido!');  
}
```

É importante notarmos que, embora parte da “culpa” por este problema possa ser atribuída à linguagem em si, medidas já foram tomadas para resolver o problema: já faz algum tempo que a configuração *register_globals* vem desativada por *default*, e já foi oficialmente anunciado que o PHP 6, a próxima grande versão da linguagem não possuirá mais esta configuração.



Observe uma técnica simples de *phishing*. Esta técnica, como a maioria das outras se baseia na falta de filtragem de informação. Quando recebemos um dado e o tratamos, por exemplo, para evitar um ataque de SQL *Injection*, esquecemos que este dado pode, em determinado momento, ser exibido em uma tela ou página qualquer.

```
while ($registro = mysql_fetch_assoc($resultado)) {  
    echo $registro['nome'];  
}
```

Agora considere que, no momento do cadastro, o usuário entrou a seguinte informação:

```
<script language='Javascript'>self.location.href='http://www.galvao.eti.br/crack.php';</script>
```

Ou seja, devemos ainda tomar cuidado com a falsa sensação de segurança: não basta realizarmos tratamento e limpeza de informações quando estas estão entrando, mas **é igualmente importante tratá-las antes de enviá-las para a saída.**



Como já foi comentado anteriormente, o tratamento de um dado deve ser aplicado considerando-se características específicas deste, como tipo, origem e destino. Existem, porém, algumas recomendações genéricas que valem para qualquer rotina de tratamento e limpeza de informação:

- Jamais confie cegamente em um dado externo;
- Seja paranóico, mas cuide para que isso tenha o mínimo de interferência na usabilidade do sistema;
- Sempre force a tipagem do dado;
- Sempre explicita a origem do dado;
- Concentre-se no que é aceito como válido para informação ou, na pior hipótese, em caracteres que são indiscutivelmente problemáticos. Evite desenvolver rotinas que possuam foco no bloqueio de caracteres/palavras;
- Quando for possível utilize soluções prontas e consolidadas;



Lembre-se dos caracteres que possuem significado sintático na linguagem SQL:

- Aspa simples ('): Delimitador de *strings*;
- Hífen (-): Quando usado em dupla (--) é o comentário SQL;
- Pipe (|): Quando usado em dupla (||) concatena *strings*;
- etc ...

Use soluções consolidadas e confiáveis:

- `mysql_real_escape_string`: Função nativa da linguagem;
- `sql_escape_string`: Função nativa da linguagem;
- Classes de abstração de dados, como PDO, MDB2 e ADODB;
- etc ...



Lembre-se dos caracteres que possuem significado sintático na linguagem HTML:

- Maior, Menor (>, <): Delimitador de *tags*;
- *Ampersand* e ponto-e-vírgula (&, ;): Definem entidades que são traduzidas em caracteres HTML automaticamente pelo *browser*;
- etc ...

Use soluções consolidadas e confiáveis:

- `htmlspecialchars`: Função nativa da linguagem;
- `htmlentities`: Função nativa da linguagem;
- etc ...



Referências:

Existe uma variedade de sites que podem auxiliar na construção de aplicações mais coerentes, seguras e menos vulneráveis:

- Documentação oficial da linguagem:

<http://docs.php.net/>

- PHP Security Consortium:

<http://www.phpsec.org/>

Procure por profissionais dedicados à segurança de aplicações PHP:

- Er Galvão Abbott (RS)
- Márcio Pessoa (SP)
- Ricardo Striquer (PR)

Procure por eventos que apresentam palestras sobre segurança de aplicações PHP:

FISL (RS), CONISLI (SP), Latinoware (PR), PHP Conference Brasil (SP)



Diversas mudanças estão previstas para as próximas versões da linguagem, que certamente possibilitarão ao desenvolvedor incrementar a segurança de suas aplicações:

- *Taint Mode*
- O fim da *register_globals*
- Tipagem explícita de parâmetros



Muito Obrigado!

Programador, especializado em segurança de aplicações PHP,
ministra cursos e presta consultoria na área.
galvao@galvao.eti.br / www.galvao.eti.br



Galvão
Especialista em TI

Fundador e Líder do PUG PHPBR, que conta hoje com
mais de 800 associados em todo o Brasil.
galvao@phpbr.com.br / www.phpbr.com.br



Diretor de Conteúdo da PHP Conference Brasil, o
principal evento de PHP da América Latina.

PHP Conference Brasil '08: 27, 28 e 29 de Novembro
galvao@phpconf.com.br / www.phpconf.com.br

