UNIVERSIDADE SAGRADO CORAÇÃO

WILLIAM DIAS SILVA DE CASTRO SOUZA

ANÁLISE DAS PRINCIPAIS VULNERABILIDADES PRESENTES EM APLICAÇÕES WEB E IMPLEMENTAÇÕES DE SEGURANÇA UTILIZANDO A LINGUAGEM INTERPRETADA LIVRE PHP

BAURU
2013

WILLIAM DIAS SILVA DE CASTRO SOUZA

ANÁLISE DAS PRINCIPAIS VULNERABILIDADES PRESENTES EM APLICAÇÕES WEB E IMPLEMENTAÇÕES DE SEGURANÇA UTILIZANDO A LINGUAGEM INTERPRETADA LIVRE PHP

Trabalho de conclusão de curso apresentado ao Centro de Ciências Exatas e Sociais Aplicadas como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, sob orientação do Prof. Espec. Henrique Pachioni Martins.

BAURU
2013

WILLIAM DIAS SILVA DE CASTRO SOUZA

ANÁLISE DAS PRINCIPAIS VULNERABILIDADES PRESENTES EM APLICAÇÕES WEB E IMPLEMENTAÇÕES DE SEGURANÇA UTILIZANDO A LINGUAGEM INTERPRETADA LIVRE PHP

Trabalho de Conclusão de Curso apresentado ao Centro de Ciências Exatas e Sociais Aplicadas da Universidade Sagrado Coração como parte dos requisitos para obtenção do título de Bacharel em Ciências da Computação sob orientação do Prof. Espec. Henrique Pachioni Martins.

Banca examinadora:

_____

Espec. Henrique Pachioni Martins.
Universidade Sagrado Coração

_____

Prof. Ms. Patricia Bellin Ribeiro
Universidade Sagrado Coração

_____

Prof. Patrick Pedreira Silva
Universidade Sagrado Coração

Bauru, 27 de novembro de 2013.

Dedico este trabalho à minha família, à minha companheira e ajudante Bruna e à todos os envolvidos.

RESUMO

A Internet é o principal meio de comunicação da atualidade, e vem facilitando inúmeras tarefas cotidianas e profissionais. Dentre essas a gestão de negócios, comercialização de produtos e publicidades em geral. Porém, esta facilidade pode gerar grandes complicações ,tanto para a empresa que faz uso, quanto para os desenvolvedores que constroem estas aplicações *web*. Atualmente o sistema de gestão empresarial *online* tem se popularizado muito entre as empresas, por sua mobilidade e facilidade de acesso à clientes e estatísticas. Com informações tão valiosas em jogo, a questão de segurança nunca deve ser deixada de lado. A partir do crescente número de ataques à *sites* e sistemas web, e do grande número de sites que possuam algum tipo de brecha para estes ataques, foi realizado um estudo das principais vulnerabilidades web, explicando o que são, como agem e quais os riscos de se tornar vítima de um eventual ataque, em seguida foram levantadas orientações de como se prevenir destas brechas, através de programação, configuração de servidor, entre outras. Evidenciando que tratar entradas de usuário é essencial para o desenvolvimento de uma aplicação segura, além da conscientização e treinamento do desenvolvedor Web.

Palavras-chave: Vulnerabilidades web, Segurança, Ataque Hacker.

ABSTRACT

The Internet is the main medium of communication today, and has facilitated numerous professional and everyday tasks. Among these business management, product marketing and advertising in general. However, this facility can generate major complications, both for the company that makes use, as for developers who build these web applications. Nowadays the online business management has become very popular among companies, by their mobility and ease of access to customers and statistics. With such valuable information at stake, the issue of security should never be overlooked. From the growing number of attacks on web sites and systems , and the large number of sites that have some sort of loophole for these attacks , a study of the major web vulnerabilities was conducted , explaining what they are, how they act and what risks becoming a victim of any attack, then guidance on how to prevent these gaps, through programming, server configuration, among others, were raised. Showing that treat user input is essential for the development of a secure application, beyond awareness and training Web Developer.

Keywords: Web Vulnerabilities, Security, Hacker Attack.

SUMÁRIO

1 INTRODUÇÃO

Sabe-se atualmente sobre as enormes transformações que a Internet vem causando na comunicação, no trabalho, no comércio e no entretenimento ao redor do mundo. Ela vem crescendo exponencialmente tanto em quantidade de sites, contabilizando 582.716.657 em janeiro de 2012 e 629.939.191 em janeiro de 2013 (NETCRAFT, 2013), como em quantidade de usuários, passando de 2,1 bilhões em 2011 para 2,4 bilhões em 2012 (PINGDOM, 2012), e a tendência é aumentar cada vez mais (PINGDOM, 2012) devido à sua popularização, facilidade e comodidade.

Com o crescimento da Internet e suas facilidades, muitas aplicações que antes eram apenas desktop passaram a ser desenvolvidas e utilizadas online, crescendo muito sua popularização (COVA, 2007). Uma série de fatores como comodidade, rapidez, facilidade e portabilidade também contribuíram para um crescente número de organizações e indivíduos confiarem em aplicações baseadas na Web, oferecendo assim, uma grande variedade de serviços. Atualmente, aplicações Web são rotineiramente utilizadas em ambientes vitais, como sistemas médicos, financeiro e militar (COVA,2007).

Ataques contra aplicações disponíveis na Internet representam uma grande parte dos incidentes de segurança ocorridos nos últimos anos. O avanço das tecnologias voltadas para a web e a falta da devida preocupação com requisitos de segurança tornam a Internet um ambiente repleto de vulnerabilidades e alvo de freqüentes ataques (CERON, 2008).

Sistemas e aplicações Web são compostos por itens de infraestrutura, tais como: servidor Web, bancos de dados e código fonte específico – tanto *client side,* como o HTML e JavaScript, como *server side*, englobando PHP, ASP, JSP, e diversas outras linguagens. Enquanto os componentes de infraestrutura são normalmente desenvolvidos por técnicos experientes, com habilidades sólidas de segurança, o código fonte das aplicações, comumente é desenvolvido por programadores com pouco treinamento em seguraça e sob estritas restrições de tempo (COVA, 2007). Como resultado, muitas aplicações vulneráveis são implantadas e disponibilizadas por toda a Internet, criando pontos de entrada facilmente exploráveis para o comprometimento de redes inteiras.

Segundo Netcraft (2013), dos 629.939.191 sites contabilizados em Janeiro de 2013, cerca de 244 milhões utilizam a linguagem interpretada livre PHP, notou-se também, que o principal servidor Web Apache ainda mantém uma maioria, contando com 55,26% do mercado, portanto, segundo SIDDHARTH, grande parte destas aplicações não possui proteção contra várias vulnerabilidades existentes. Estas aplicações, por trafegarem informações e dados valiosos para a empresa ou o usuário, não podem continuar vulneráveis aos possíveis ataques.

Outro ponto importante é o crescimento do número de desenvolvedores de softwares no mundo (BLS, 2012). Isto significa que atualmente, muitas pessoas sabem ou têm noções de programação e codificação de aplicações. Com o aumento do número de pessoas que possuem conhecimento sobre o desenvolvimento de um site e quais são suas possíveis defesas, conseqüentemente aumentou-se o número de ataques a sites vulneráveis.

Através destas informações, foram levantadas as principais vulnerabilidades presentes em *sites* institucionais e aplicações Web, analisando-as através de referencial teórico, identificando assim, do que se tratam e como funciona um ataque de forma maliciosa, para então ser demonstradas possíveis formas de proteção, correção e implementação de segurança contra as mesmas, podendo ser através de simples configurações do próprio servidor, ou de códigos de programação mais complexos utilizando-se da linguagem interpretada livre PHP. Foram desenvolvidas aplicações práticas para demonstrar como funciona um ataque através de uma vulnerabilidade web, e quais as implementações de segurança necessárias para proteger-se, através de simulações de uma aplicação vulnerável sofrendo o ataque, e outra aplicação protegida, repelindo o ataque.

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

Apontar as principais vulnerabilidades em aplicações Web, e mostrar implementações de segurança para as mesmas, sejam por meio de código de programação, utilizando-se da linguagem interpretada livre PHP, ou através de configurações de servidor.

1.1.2 OBJETIVOS ESPECÍFICOS

- Analisar através de referencial teórico as principais vulnerabilidades presentes em aplicações Web atuais.
- Desenvolver aplicação sem proteção para determinada vulnerabilidade e demonstrar funcionalidade de ataque contra a mesma.
- Desenvolver e demonstrar implementações de segurança para cada vulnerabilidade em questão.
- Apresentar resultados e discutir sobre as implementações de segurança validadas, através de aplicações práticas, para cada vulnerabilidade analisada.

1.2 JUSTIFICATIVA

Um ataque pode trazer conseqüências catastróficas para uma empresa com gestão *online*, ou mesmo *sites* institucionais, pois além de comprometerem a segurança e confidencialidade dos dados, podem também denegrir a imagem da instituição, pois através de um acesso ao sistema administrativo de forma maliciosa, pode-se alterar todo o conteúdo do *site*, além de ter livre acesso às suas informações confidenciais. A partir disto, foram analisadas as principais vulnerabilidades presentes nas aplicações Web, e em seguida desenvolvidos aplicações práticas - utilizando a linguagem interpretada livre PHP - demonstrando o risco de cada vulnerabilidade e as implementações de segurança necessárias para

proteger-se, e assim, incrementar segurança e qualidade dos *sites* e sistemas atuais.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 SQL INJECTION

SQL Injection é um dos muitos mecanismos de ataque da Web utilizados por *hackers* para roubar dados de organizações. É talvez uma das técnicas de ataque de camada de aplicação mais comuns usados hoje. É o tipo de ataque que tira proveito de codificação indevida de suas aplicações Web que permite que hackers possam injetar comandos SQL em um formulário de *login*, que lhes permitam ter acesso aos dados armazenados no seu banco de dados. Em essência, SQL Injection surgiu porque os campos disponíveis para a entrada do usuário permitiam que instruções SQL possam consultar o banco de dados diretamente (ACUTENIX, 2013).

SQL Injection é uma abordagem muito antiga, mas ainda é popular entre os atacantes. Esta técnica permite a um atacante obter informações cruciais de banco de dados de um servidor Web. Dependendo das medidas de segurança da aplicação, o impacto deste ataque pode variar de divulgação de informações básicas, para a execução remota de código e comprometimento total do sistema (SIDDHARTH, 2010).

2.1.1 COMO FUNCIONA

Em uma página de *login* simples, onde um usuário legítimo entraria com seu nome de usuário e senha, para entrar em uma área segura, e ver seus detalhes pessoais ou enviar seus comentários em um fórum. Quando esse usuário envia seus dados, uma consulta SQL é gerada a partir desses detalhes e submetida ao banco de dados para verificação. Se válida, o usuário tem permissão de acesso. Em outras palavras, a aplicação Web que controla a página de *login* irá se comunicar com o banco de dados por meio de uma série de comandos planejados, de forma a verificar a combinação do nome de usuário e senha. Na verificação, ao usuário legítimo é concedido o acesso apropriado (ACUTENIX, 2013).

Através de SQL Injection, o *hacker* pode dar entrada a específicos comandos SQL, com a intenção de contornar a barreira do formulário de login, e ver o que está

por trás dela. Isso só é possível se as entradas não são devidamente higienizadas, e são enviadas diretamente com a consulta SQL para o banco de dados. Vulnerabilidades de SQL Injection fornecem os meios para um *hacker* se comunicar diretamente com o banco de dados (ACUTENIX, 2013).

Segundo Cova (2007), as aplicações Web têm uma vulnerabilidade de SQL Injection, quando se usa dados não tratados para compor consultas que serão passadas para um banco de dados relacional. “Isso pode levar a consultas arbitrárias sendo executados no banco de dados com os privilégios da aplicação vulnerável” (COVA, 2007).

```
$activate = $_GET (" activate ");
$result = dbquery("SELECT * FROM new_users WHERE user_code= '$activate'");
if ( $result ) {...}
```

Figura 1 – Consulta à banco sujeita a SQL Injection.
Fonte: COVA, 2007

A função DBQuery é usada para executar uma consulta a um banco de dados relacional e retornar os resultados para a aplicação. Neste caso, a variável *$activate* está definida para o conteúdo do parâmetro de solicitação do mesmo nome. O uso pretendido da variável é conter código pessoal do usuário para compor dinamicamente o conteúdo da página. No entanto, se um atacante envia uma query, onde o parâmetro ‘*activate’* é a string **'OR 1 = 1**, a consulta irá retornar o conteúdo de toda a nova tabela de usuários, pois na query será verificada se uma das expressões são verdadeiras, o código do usuário, ou 1=1, essa por sua vez sempre será verdadeira independente do código do usuário. Se o resultado da consulta é usado mais tarde como o conteúdo da página, esta irá expor informações pessoais. Outros ataques, tais como a supressão de tabelas de dados, ou a adição de um novo utilizador, também são possíveis (COVA, 2007).

2.2 CROSS SITE SCRIPTING

Cross Site Scripting, mais conhecido como XSS, é um dos ataques mais comuns em aplicações Web. XSS normalmente utiliza *scripts* embutidos em uma página que são executados no *client-side* (navegador do usuário), em vez de serem executados no servidor. XSS em si é uma ameaça provocada pelas falhas de segurança nos scripts executados do lado do cliente, como HTML e JavaScript, que são os principais culpados. O conceito de XSS é manipular os *scripts* do lado do cliente de uma aplicação Web para executar da maneira desejada pelo usuário mal-intencionado. Tal manipulação pode inserir um *script* em uma página, que pode ser executado toda vez que a página é carregada, ou sempre que um evento associado é executado (ACUTENIX, 2013).

Segundo Spett (2005), Cross Site Scripting ocorre quando páginas da Web que são geradas dinamicamente não são devidamente validadas. Isto permite que um invasor inclua códigos JavaScript maliciosos na página gerada, e execute o *script* na máquina de qualquer usuário que vê esse *site*. Cross Site Scripting pode afetar qualquer *site* que permite aos usuários inserir dados.

> *"Um invasor que utiliza um script cross-site com sucesso, pode comprometer informações confidenciais, manipular ou roubar cookies, criar solicitações que podem ser confundidas com as de um usuário válido, ou executar um código malicioso na máquina dos usuários finais (SPETT,2005)."*

Porém, segundo Siddharth (2010), o sucesso deste ataque exige que a vítima execute uma URL maliciosa, que pode ser trabalhada de forma a parecer legítima no primeiro olhar. Ao visitar tal URL trabalhada, um atacante pode efetivamente executar algo malicioso no navegador da vítima. Alguns JavaScript maliciosos, por exemplo, serão executados no contexto do *site*, que possui a vulnerabilidade XSS.

2.2.1 COMO FUNCIONA

Teoricamente é impossível obter-se informações de uma página através de um *script* contido em outra página de um *host* diferente. A "arte" do XSS é justamente encontrar uma brecha que permita contornar os mecanismos de segurança implementados pelos navegadores. Como, por exemplo, encontrar formas inteligentes de injetar um código mal-intencionado na página visitada pela vítima. Segundo Bodmer (2007), isto é possível das seguintes formas:

-Injetando através de entradas de formulário HTML.

-Injeção de URL (URL enviados por *e-mail* ou mensagem instantânea para a vítima, ou postados em *sites* públicos).

-Injeção através de outras entradas (*e-mail*, sms, etc).

A Figura 2 evidencia um exemplo do funcionamento de um ataque XSS através de injeção de URL (BODMER, 2007). Neste caso, o *site* exibe uma saudação personalizada, "Olá Blake." (este mesmo problema também pode ocorrer, por exemplo, quando um usuário digita uma *string* de pesquisa para encontrar um produto, ou outro conteúdo no *site*.) Depois de ver o conteúdo personalizado, o atacante decide testar a segurança do *site*, adicionando um *script* no *URL*, para determinar se o site irá executá-lo.

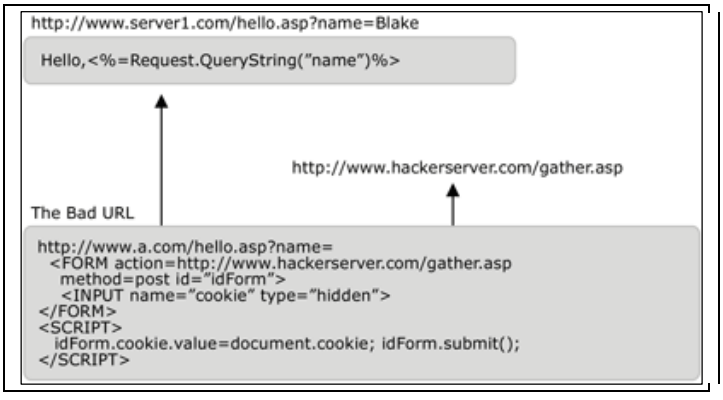

Figura 2 – EXEMPLO DE FUNCIONAMENTO XSS
Fonte: Microsoftw Developer Network, *How Cross Site Scripting Attacks Work,* 2005

No exemplo relatado na Figura 2, o *script* é executado, e, agora, todos os usuários serão redirecionados para o *site* do atacante, “www.hackerserver.com/gather.asp”. Quando os usuários acessam o *site* do atacante, o mesmo pode reunir seus dados de perfil (MICROSOFT, 2005).

2.3 FILE UPLOAD ATTACK

Segundo Calin (2009) permitir que um usuário faça *upload* de arquivos em seu *site*, é como abrir mais uma porta para um usuário mal intencionado comprometer o servidor. Mesmo assim, em aplicações Web modernas, é um requisito comum, pois ajuda a aumentar a eficiência do seu negócio. *Upload* de arquivos são permitidos em aplicações Web de rede sociais, como Facebook e Twitter. Eles também são permitidos em *blogs*, fóruns, *sites* de *e-banking*, YouTube, e também em portais de suporte corporativo. Usuários têm permissão para fazer *upload* de imagens, vídeos, avatares e muitos outros tipos de arquivos.

Dalili (2011) explica que estes arquivos enviados representam um risco significativo para aplicações. O primeiro passo em muitos ataques é enviar um código para o sistema a ser atacado. Em seguida, o atacante só precisa encontrar uma maneira de executar o código no servidor.

As consequências de se permitir o envio de arquivos irrestritamente podem variar, incluindo aquisição completa do sistema, sistema de arquivos sobrecarregado, encaminhamento dos ataques aos sistemas de *back-end* ou apenas simples alterações nas configurações. Depende muito do que a aplicação faz com o arquivo enviado, e principalmente onde ele é armazenado.

2.3.1 COMO FUNCIONA

Segundo Calin (2009), um simples *upload* de arquivo geralmente consiste de um formulário HTML e um *script* PHP, ou outra linguagem Web. O formulário HTML, é a forma apresentada ao usuário, enquanto o script PHP contém o código que cuida do *upload* do arquivo.

Tendo como exemplo o formulário HTML, Figura 3.

```
<form enctype="multipart/form-data" action="uploader.php" method="POST">
	<input type="hidden" name="MAX_FILE_SIZE" value="100000"/>
	Arquivo: <input type = "file" name = "UploadedFile"/> <br />
	<input type="submit" value="Upload File" />
</form>
```

Figura 3 – Formulário HTML.
Fonte: CALIN, 2009.

Na Figura 4, o seguinte *back-end* em PHP.

```
<?php
$target_path  =  "uploads/";
$target_path  =  $target_path  .  basename($_FILES('uploadedfile')('name'));
if(move_uploaded_file($_FILES('uploadedfile')('tmp_name'),$target_path)) {
	echo "O arquivo " . basename($_FILES('uploadedfile')('name')) . "
foi enviado";
} else {
	echo "Houve  um erro! Tente novamente";
}
?>
```

Figura 4 – Back-end PHP.
Fonte: CALIN, 2009.

Quando o PHP recebe uma solicitação POST com a codificação tipo “multipart/form-data”, ele irá criar um arquivo temporário de nome aleatório em um diretório temporário (Ex. “/var/tmp/php6yXOVs”). O PHP também irá preencher o *array* global $_FILES com as demais informações sobre o arquivo enviado, como nome, tipo *mime,* tamanho em *bytes* e seu nome temporário.

A função PHP “move_uploaded_file” irá mover o arquivo temporário para um local fornecido pelo desenvolvedor. Nesse caso, o destino é abaixo da raiz do servidor. Portanto, os arquivos podem ser acessados utilizando uma *URL* como: http://www.domain.tld/uploads/uploadedfile.ext. Neste exemplo simples, não há restrições sobre o tipo de arquivos permitidos para *upload,* e, portanto, um atacante pode enviar um arquivo de extensão PHP ou .NET, com código malicioso que poderá comprometer o servidor.

2.4 SESSION HIJACKING

Segundo Owsap (2011), Session Hijacking (Sequestro de Sessão) consiste na exploração do mecanismo de controle da sessão web, que normalmente é gerada por um *token*. Como a comunicação HTTP usa muitas conexões TCP diferentes, o servidor Web precisa de um método para reconhecer as conexões de cada usuário. O método mais útil é através de um sinal que o servidor Web envia ao navegador do cliente após uma autenticação bem-sucedida, um *token*. Um *token* de sessão é normalmente composto por uma *string* de largura variável, e pode ser utilizado de diferentes maneiras, como no *URL*, no cabeçalho da requisição HTTP na forma de um *cookie*, em outras partes do cabeçalho do pedido HTTP, ou ainda no corpo da requisição HTTP. O ataque de Session Hijacking compromete o *token* da sessão roubando-o ou gerando um *token* válido através de adivinhação para ganhar acesso não autorizado ao servidor web.

Segundo ISS (1999), o sequestro de uma sessão TCP (Session Hijacking) é quando um *hacker* assume uma sessão TCP entre duas máquinas. Como a maioria das autenticações apenas ocorre no início de uma sessão de TCP, isso permite que o hacker garanta acesso à uma máquina.

2.4.1 COMO FUNCIONA

Segundo Owsap (2011), como pode ser visualizado na Figura 5, na primeira vez o atacante utiliza uma ferramenta chamada Sniffer para capturar uma sessão de *token* válida chamada "Session ID", ele então usa o *token* válido para ganhar acesso não autorizado ao servidor Web através de modificadores de cookie, que podem ser facilmente obtidos na forma de extensões para os atuais navegadores Web.

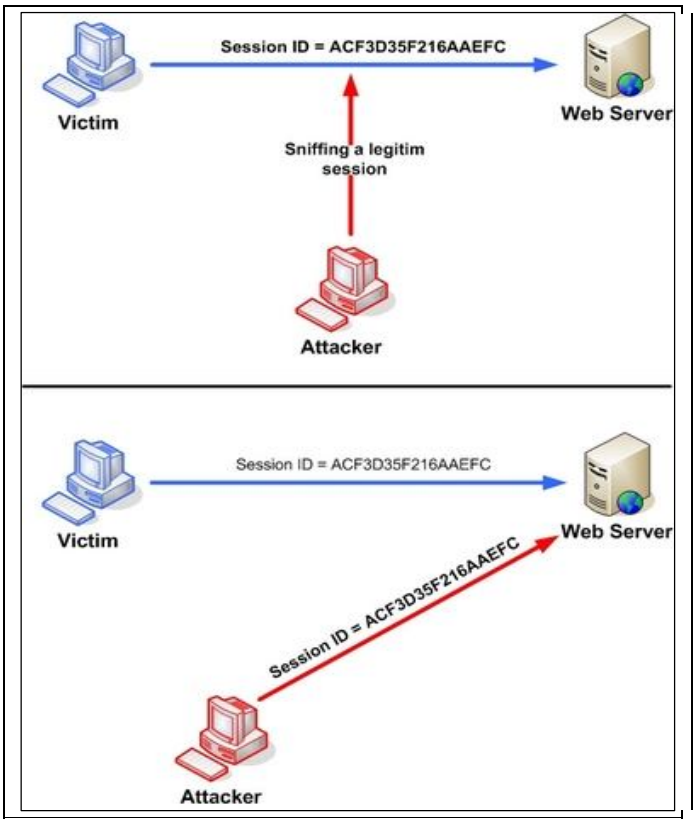

Figura 5 - Manipulando a sessão simbólica para executar o ataque de Session Hijacking
Fonte: OWSAP, 2011.

2.5 DIRECTORY TRAVERSAL

Segundo Acutenix (2013), Directory Traversal é um exploit HTTP que permite aos *crackers* acessar diretórios restritos, e executar comandos fora do diretório raiz do servidor Web. Os servidores Web fornecem dois níveis principais de mecanismos de segurança:

- Access Control Lists (ACLs)
- Diretório root (raiz)

O ACL é utilizado no processo de autorização. É uma lista que o administrador do servidor utiliza para indicar quais usuários ou grupos são capazes de acessar, modificar ou executar determinados arquivos no servidor, bem como outros direitos de acesso. Já o *root* (raiz) é um diretório específico no sistema de arquivos do servidor, no qual os usuários estão confinados. Os usuários não são capazes de acessar qualquer coisa acima dessa raiz.

2.5.1 COMO FUNCIONA

Segundo Owsap (2009), ao navegar no aplicativo, o atacante procura por *links* absolutos de arquivos armazenados no servidor. Manipulando variáveis que referenciam arquivos com "ponto-ponto-barra (../)", sequências e suas variações, é possível acessar arquivos arbitrários, como o código fonte da aplicação, configurações do servidor, e arquivos críticos do sistema operacional. O atacante usa sequências de "../" para mover-se para o diretório raiz, permitindo assim a navegação pelo sistema de arquivos.

Utilizando-se do exemplo dado por Acutenix (2013), que diz que em aplicações Web com páginas dinâmicas, a entrada é geralmente recebida pelos navegadores através de solicitações GET ou POST, como por exemplo na *URL* http://test.webarticles.com/show.php?view=oldarchive.html. Com esta *URL*, o navegador solicita a página dinâmica “show.php” no servidor, e com ela também envia o parâmetro "view" com o valor de "oldarchive.html". Quando este pedido é executado no servidor Web, “show.php” recupera o arquivo “oldarchive.html” do sistema de arquivos, renderiza-o, e em seguida envia-o de volta para o navegador que o exibe ao usuário. O atacante irá supor que “show.php” pode recuperar arquivos do sistema, e envia então uma *URL* http://test.webarticles.com/show.asp?view=../../../../../Windows/system.ini.

Isso fará com que a página dinâmica recupere o arquivo “system.ini” do sistema, e exiba-o ao usuário. A expressão “../” instrui o sistema a subir um diretório acima. O atacante portanto terá que adivinhar quantos diretórios ele deve subir até encontrar a pasta do Windows, mas isso é facilmente feito por tentativa e erro.

2.6 CROSS SITE REQUEST FORGERY

Segundo Käfer (2008), Cross Site Request Forgery (CSRF) é uma técnica que permite que o atacante engane o usuário para realizar uma ação, usando sua autoridade e credencial. Ou seja, segundo Owsap (2013), CSRF é um ataque que engana a vítima para o carregamento de uma página que contém um pedido malicioso. É mal-intencionado no sentido de que ele herda a identidade, e os privilégios da vítima para executar uma função indesejada em nome da vítima, como a mudança de endereço de *e-mail*, endereço residencial, senha, ou até mesmo comprar algo. CSRF ataca geralmente funções que provocam uma alteração de estado no servidor, mas também pode ser usado para obter dados confidenciais.

Para a maioria dos *sites*, os navegadores irão incluir automaticamente com essas solicitações quaisquer credenciais associadas ao *site*, como o *cookie* da sessão, as credenciais de autenticação básica, o endereço IP, as credenciais de domínio do Windows, etc. Portanto, se o usuário já está autenticado no *site*, o *site* não terá nenhuma maneira de distinguir isso de uma solicitação legítima.

Zeller (2008) relata que os ataques ocorrem quando um *site* malicioso faz com que o navegador Web do usuário execute uma ação indesejada em um *site* confiável. Estes ataques têm sido chamado de "gigante adormecido" das vulnerabilidades Web, porque muitos sites na Internet não protegem contra eles, e porque eles têm sido largamente ignorados pelos desenvolvedores Web e comunidades de segurança.

Os ataques CSRF não aparecem na Web Security Threat Classification (Classificação das ameaças à segurança Web), e raramente são discutidos em literaturas acadêmicas ou técnicas.

2.6.1 COMO FUNCIONA

Segundo Owsap (2013), existem inúmeras maneiras de um usuário final ser enganado para carregar, ou enviar informações em uma aplicação web. Para executar um ataque, devemos primeiro entender como gerar um pedido malicioso para a vítima executar.

Portanto vamos considerar um exemplo hipotético de um site vulnerável à um ataque de CSRF. Zeller (2008) nos mostra o exemplo de um *site* de serviços de *e-mail* que permite aos usuários enviar e receber emails. O *site* usa autenticação simples para autenticar seus usuários. Uma página, *http://example.com/compose.html*, contém um formulário HTML, que permite que o usuário insira o endereço de *e-mail* do destinatário, o assunto, e a mensagem, bem como um botão que diz "Enviar email".

```
<form action = "http://example.com/send_email.html" method = "GET">
    Endereço de email do destinatário: <input type="text" name="para">
    Assunto: <input type="text" name="assunto">
    Mensagem: <textarea name="msg"> </ textarea>
    <input type="submit" value="Enviar email">
</form>
```

Figura 6 – Formulário HTML para envio de e-mails.
Fonte: ZELLER, 2008.

Quando um usuário de “*example.com”* clica em "Enviar email", os dados inseridos serão enviados para *http://example.com/send_email.html* como um pedido GET. Uma vez que uma requisição GET simplesmente acrescenta os dados do formulário para a *URL*, o usuário será redirecionado para a seguinte *URL* (supondo que ele entrou "bob@example.com" como destinatário, "Olá" como assunto, e "Qual é a proposta?", como a mensagem):

http://example.com/send_email.html?para=bob%40example.com&assunto=Ol%C3%A1&msg=Qual%20%C3%A9%20a%20sita%C3%A7%C3%A3o%20da%20proposta%3F.

A página “*send_email.html* “levaria os dados recebidos e enviaria um *e-mail* para o destinatário do usuário. Note que “*send_email.html”* simplesmente pega os dados, e realiza uma ação com esses dados. Não importa onde originou-se a solicitação, só que o pedido foi feito. Isto significa que se o utilizador introduzir manualmente a URL em seu navegador, *example.com* ainda mandaria o e-mail.

Segundo Zeller (2008), um ataque CSRF é possível aqui porque “send_email.html” leva todos os dados que recebe e envia um *e-mail.* Ele não verifica se os dados foram originados a partir do formulário “compose.html”, portanto, se um atacante pode fazer com que o usuário envie uma solicitação para “send_email.html”, está página fará com que “*example.com”* envie um *e-mail* em

nome do utilizador, contendo os dados de escolha do invasor, assim o atacante irá ter realizado com sucesso um ataque CSRF.

3 METODOLOGIA

Após pesquisa bibliográfica sobre diversas vulnerabilidades que estão presentes em aplicações web atuais, notou-se que este é de fato um problema sério e necessita da preparação e conscientização dos desenvolvedores sobre o assunto, afinal segurança é fundamental. Com base nesta pesquisa e com o fim de validar tais vulnerabilidades e então demonstrar formas de eliminá-las, e proteger assim sua aplicação, foram desenvolvidas aplicações práticas para testes e validações. Para isto, foi configurado um servidor local, utilizando-se da distribuição gratuíta do software livre **XAMPP** (XAMPP, 2013), no sistema operacional Windows 7 *Ultimate*.

Após instalação e configuração do servidor Web, foram desenvolvidas duas aplicações na forma de sistema simples, como formulários de login e acessos à bases de dados em específico para cada vulnerabilidade citada. A primeira aplicação chamaremos de **Aplicação A,** uma aplicação totalmente vulnerável à determinada vulnerabilidade, a segunda, **Aplicação B**, uma aplicação de mesmo estrutura da Aplicação A, porém com as implementação de segurança testadas para a vulnerabilidade em questão.

Por fim, foi executado uma sequência de ataques nas duas aplicações (Aplicação A e Aplicação B) com o objetivo de demonstrar a funcionalidade e danos de cada vulnerabilidade quando não tratadas (Aplicação A), e a ineficiência dos ataques quando tratadas (Aplicação B). A seguir descrição de cada aplicação em determinada vulnerabilidade.

3.1 SQL INJECTION

Foram testadas duas aplicações contendo o mesmo formulário de *login* básico em HTML (login.php), com os campos ‘usuario’ e ‘senha’, inseridos em um Form com Method ‘*post*’, onde será feita uma consulta MySQL para autenticação.

3.1.1 APLICAÇÃO A

Aplicação A utiliza a biblioteca de conexão MySQL para PHP, e não realiza nenhum tratamento para os dados inseridos pelo usuário, os mesmo são atribuidos diretamente à *query* executada no banco, conforme Figura 7.

```
//conexão ao banco de dados mysql
$con = mysql_connect('localhost','root','');

//selecionar base
$db = mysql_select_db('tcc');

//montar query mysql
$query = "SELECT * FROM usuario WHERE usuario
='".$_POST['usuario']."' AND senha = md5('".$_POST['senha']."')
LIMIT 1";

//comitar, executar query
$result = mysql_query($query) or die(mysql_error());

//seleciona usuario
$usuario = mysql_fetch_array($result);
```

Figura 7 – Código utilizado para Aplicação A.
Fonte: Autor.

3.1.2 APLICAÇÃO B

Aplicação B utiliza a biblioteca de manipulação de dados PDO (PHP Data Object) orientada à objeto, com tratamento e escape dos dados inseridos pelo usuário, enviando para o banco de dados uma *query* MySQL válida e tratada, como demonstrado na Figura 8.

```
//cria objedo de conexão PDO
$pdo= new PDO('mysql:dbname=tcc;host=localhost,'root','');

//query SQL com parâmetros preparados :usuario :senha
$stmt = $pdo->prepare("SELECT * FROM usuario
                  WHERE usuario =:usuario
                  AND senha = md5(:senha) LIMIT 1");

//prepared statments, escape e validação de strings
$params = array(
      'usuario' => $_POST['usuario'],
      'senha' => $_POST['senha'] );

//execução da query
$stmt->execute($params);

//seleciona usuario
$usuario = $stmt->fetch(PDO::FETCH_ASSOC);
```

Figura 8 – Código utilizado na Aplicação B.
Fonte: Autor.

3.2 CROSS SITE-SCRIPTING (XSS)

Teste em duas aplicações contendo uma página PHP que recebia via protocolo *HTTP-GET* uma variável denominada ‘pesquisa’, Figura 10, em seguida exibe-se uma mensagem com o termo pesquisado e o resultado da busca. Está página contem também a simulação de uma sessão de autenticação, apenas uma sessão com dados fictícios para simular uma área restrita que necessita autenticação através de *login*, no início da página encontra-se o código PHP responsável por criar uma sessão, Figura 9.

```
session_start();
$_SESSION['usuario_logado'] = array();
```

Figura 9 – Simulação de sessão logada.
Fonte: Autor.

Este código, Figura 9, cria automaticamente um ID de sessão para aquela máquina, naquele servidor, autenticando este determinado usuário fictício.

Figura 10 – Formulário de busca HTML com Form utilizando *HTTP-GET.*
Fonte: Autor

Por ser um formulário com Method *GET* o parâmetro enviado é exibido na URL, resultando no endereço: *http://localhost/tcc/xss.php?termo=William+Castro.*

3.2.1 APLICAÇÃO A

A Aplicação A recebe a variável 'termo' e a exibe sem qualquer tratamento ou validação, como evidencia a Figura 11.

```
<?php if (isset($_GET['termo'])){
      $termo = $_GET['termo'];

      /**
      /* Pesquisa em banco com o termo
      /* ....
      /**/

      echo "Resultado de pesquisa para '".$termo."': <br/>";

      /** Imprime os resultados de pesquisa **/
      /*  .... */
      ?>
 <?php }?>
```

Figura 11 – Utilização de parâmetro *Get* sem tratamento.
Fonte: Autor.

3.2.2 APLICAÇÃO B

A Aplicação B recebe a variável ‘termo', Figura 12, e então utiliza as funções PHP: *strip_tags,* responsável por remover toda e qualquer *tag* HTML, principalmente a *tag* <SCRIPT>, responsável pela injeção de XSS; e *htmlentities*, função que converte e retorna todos os caracteres especiais de uma string em uma sentença HTML válida, visando bloquear injeções XSS (PHP, 1999).

```
<?php if (isset($_GET[‘nome’])){
      $termo = $_GET['termo'];

      //remove tags HTML como <script>
      $termo = strip_tags($termo);
      //converte caracteres especiais
      $termo = htmlentities($termo);

      /**
      /* Pesquisa em banco com o termo
      /*
      /* ....
      /**/

      echo "Resultado de pesquisa para '".$termo."': <br/>";

      /** Imprime os resultados de pesquisa **/
      /*  .... */
      ?>
 <?php }?>
```

Figura 12 – Código utilizado na Aplicação B para XSS.
Fonte: Autor.

3.3 FILE UPLOAD ATTACK

Para testar esta vulnerabilidade, foram desenvolvidas duas aplicações de cadastro de currículo online, contendo um formulário HTML, com Method POST e atributo *enctype="multipart/form-data"*. Este formulário submete o currículo à uma página PHP responsável por salvar o mesmo no diretório *I:\xampp\htdocs\tcc\curriculos* .

3.3.1 APLICAÇÃO A

A aplicação A, Figura 13, recebe o currículo enviado, e transfere-o para a pasta de currículos sem nenhum tratamento ou verificação de tipo. Foi utilizada a função PHP *move_uploaded_file* para salvar o arquivo no servidor.

```
<?php
      $diretorio = "curriculos/";

      //recebe curriculo via POST
      $curriculo = $_FILES['curriculo'];

      //move curriculo para diretorio
      if  (move_uploaded_file($curriculo['tmp_name'],  $diretorio
      . $curriculo['name'])) {
          echo "Currículo enviado com sucesso.";
      } else {
          echo "Ocorreu um erro, tente novamente.";
      }
?>
```

Figura 13 – Código utilizado na Aplicação A para *file upload attack.*
Fonte: Autor.

3.3.2 APLICAÇÃO B

Aplicação B, possui diversas formas de proteção e validação dos arquivos enviados, com o objetivo de aceitar, salvar e listar apenas arquivos do tipo documento de texto do MS Word (.doc ou .docx). Em primeiro lugar, o diretório onde

são salvos os currículos, contém um arquivo **.htaccess** (Hypertext acccess), que consiste em um arquivo de configuração à nivel de diretório do servidor apache. Este arquivo contém configurações referentes à listagem de arquivos, permitindo apenas a listagem via protocolo HTTP de arquivos com extensão *.doc* ou *.docx.* A Figura 14 evidencia o código *.htaccess* utilizado.

```
deny from all
<Files ~ "^\w+\.(doc|docx)$">
      order deny,allow
      allow from all
</Files>
```

Figura 14 – Código .htaccess utilizado na Aplicação B para *file upload attack.*
Fonte: Autor.

Complementando a listagem de arquivo, a Aplicação B contém a validação do tipo de arquivo que o usuário envia ao servidor, através da verificação do tipo MIME (Multipurpose Internet Mail Extensions), que é uma norma da internet para o formato das mensagens de correio eletrônico, contendo a categoria e o tipo do arquivo (MIME, 1996). Foi autorizado então apenas o tipo MIME *application/msword*, ou seja, arquivos aceitos pelo Microsoft Word, Figura 15.

```
<?php
            $diretorio = "curriculos/";

            //recebe curriculo via POST
            $curriculo = $_FILES['curriculo'];

            //verifica MIME Type
            if($curriculo['type'] != "application/msword") {
                  echo "Desculpe, apenas documentos do tipo .doc e .docx
                        são aceitos.<br />";
                  exit;
            }

            //move curriculo para diretorio
if       (move_uploaded_file($curriculo['tmp_name'],       $diretorio       .
$curriculo['name'])) {
                  echo "Currículo enviado com sucesso.";
            } else {
                  echo "Ocorreu um erro, tente novamente.";
            }
?>
```

Figura 15 – Código utilizado na Aplicação B para *file upload attack*.
Fonte: Autor.

3.4 SESSION HIJACKING

O exemplo de *session hijacking* complementa o já citado exemplo de XSS, onde foi utilizado as mesmas aplicações. Ou seja, um formulário de pesquisa em uma área restrita da aplicação, em que a vítima de alguma forma é atraída à uma URL contendo injeção de JavaScript, este JavaScript, envia automáticamente no carregamento da página, a variável JavaScript *document.cookie*, variável que armazena dados de cookies e sessões, no caso os dados da sessão ativa criada no momento em que a vítima efetua seu login na aplicação e recebe acesso à área restrita.

A aplicação de *session hijacking* foi incrementada com um formulário de *login* em HTML, que trata e verifica a autenticidade do mesmo, e então salva um *array* na sessão com nome ‘logado’, este *array* contém informações do usuário como nome, nome de usuário, último login. Assim que o usuário válido é confirmado, a página é redirecionada à uma área restrita, com mensagem de boas-vinda.

Para ambas aplicações (A e B) foram utilizados o mesmo código para verificação da existência de uma sessão válida, Figura 16.

```
<?php
      if (!isset($_SESSION['logado'])){
            header('Location: login.php');
            exit;
      }
?>
```

Figura 16 – Código utilizado nas Aplicações A e B para *session hijacking*.
Fonte: Autor.

Caso exista a sessão 'logado', o usuário visualiza normalmente o conteúdo da página restrita, caso contrário ele é direcionado à página de *login* para iniciar uma sessão, Figura 17.

Figura 17 – Página de login para teste de Session Hijacking.
Fonte: Autor.

3.4.2 APLICAÇÃO A

A Aplicação A utiliza as configurações padrões do servidor APACHE, e é também vulnerável à injeções XSS pelo formulário de pesquisa, como evidenciado na demonstração de XSS.

3.4.3 APLICAÇÃO B

A Aplicação B contém configurações adicionais de segurança do PHP, adicionadas no arquivo *php.ini* localizado em *I:\xampp\php\php.ini,* Figura 18.

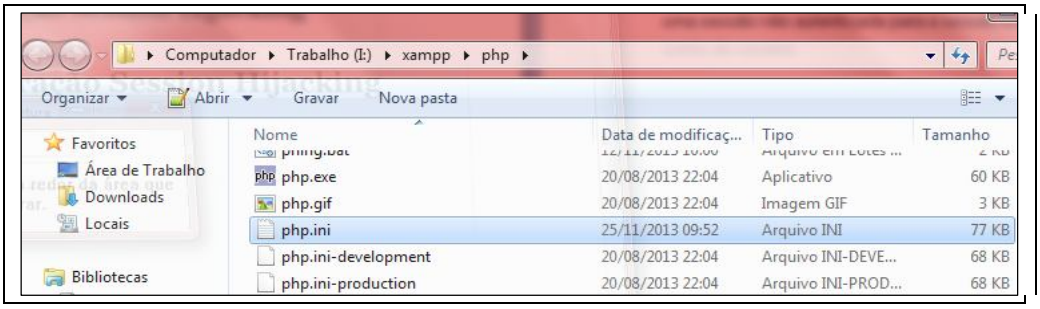

Figura 18 – Localização do arquivo de configuração do PHP.
Fonte: Autor.

Neste arquivo foram adicionadas as seguintes configurações, dentro da categoria [Session], Figura 19.

```
session.session.use_only_cookies = 1
session.cookie_secure = 1
```

Figura 19 – Configuração php.ini para *session hijacking.*
Fonte: Autor.

Segundo PHP, *session.use_only_cookies* especifica se o módulo usará apenas cookies para guardar a ID da sessão no lado do cliente. Habilitando esta configuração previne ataques envolvendo passagem de IDS de sessão em URLs. E a função *session.cookie_secure* especifíca que os cookies sejam apenas enviados sob conexões seguras.

3.5 DIRECTORY TRAVERSAL

A fim de demonstrar o funcionamento de um ataque Directory Traversal, foram desenvolvidas duas aplicações (A e B), na forma de *template* simples (layout), ou seja, um cabeçalho e rodapé fixos, na qual as páginas são carregadas dinamicamente e identificadas via *URL* por seu nome do arquivo HTML correspondente, Figura 20. Esta variável é recebida via HTTP-GET, denominada arquivo, e inclusa na aplicação através da função PHP *include,* Figura 21.

Figura 20 – URL carregando página HTML por nome.Fonte: Autor.

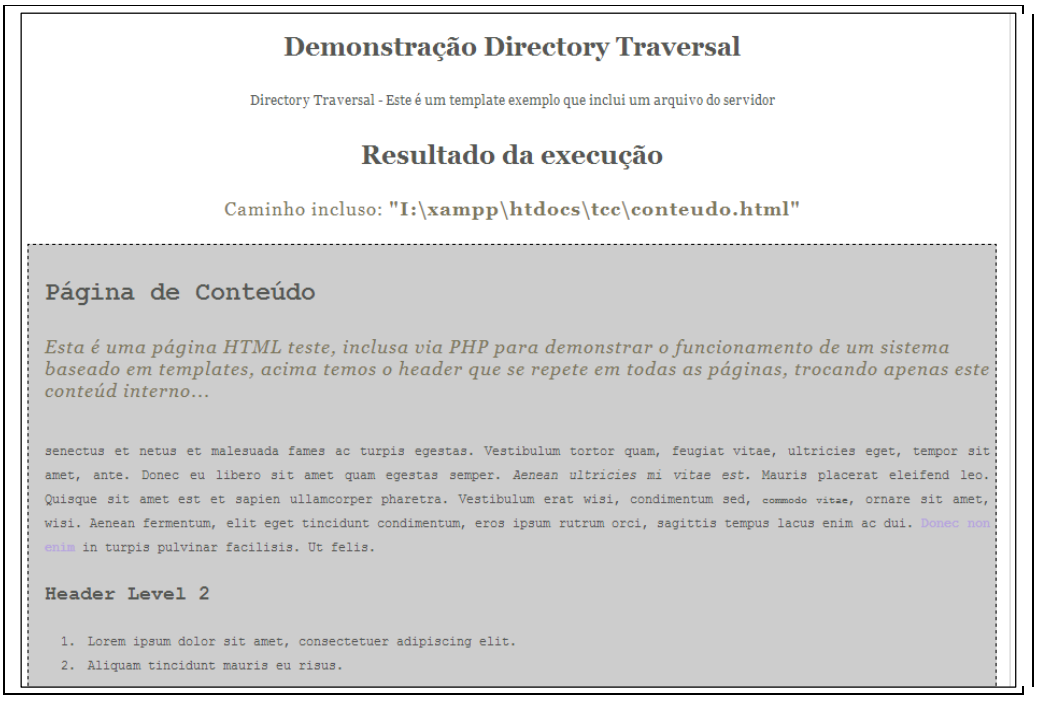

Figura 21 – Resultado de inclusão da página ‘conteudo.html’ no *template.*
Fonte: Autor.

3.5.1 APLICAÇÃO A

A Aplicação A, referencia diretamente o parâmetro recebido via URL para a variável arquivo, incluindo a própria, para ser exibida na tela, Figura 22.

```php
<?php
	$arquivo = $_GET['arquivo'];

	include($arquivo);
?>
```

Figura 22 – Inclusão de arquivo sem tratamento.
Fonte: Autor.

3.5.2 APLICAÇÃO B

A Aplicação B, antes de incluir a suposta página contida na variável arquivo, utiliza a função PHP *basename*, função responsável por retornar o nome base de um arquivo através de seu caminho absoluto (PHP, 1999).

Ao incluir, verifica-se também a existência do arquivo, caso não seja encontrado é exibida uma mensagem de erro personalizada, Figura 23.

```
<?php
	$arquivo = $_GET['arquivo'];

	if(!@include(basename($arquivo)){
		echo "404 Página não encontrada.";
	}
?>
```

Figura 23 – Inclusão de arquivo com tratamento de existência e caminho do arquivo.
Fonte: Autor.

3.6 CROSS SITE REQUEST FORGERY

Cross Site Request Forgery é a vulnerabilidade mais presente nas aplicações e menos conhecida pelos desenvolvedores. Porém, um ataque através desta vulnerabilidade é bem trabalhoso, e necessita que o atacante atraia sua vítima para uma página sua, e ainda que a vítima esteja devidamente autenticada na aplicação que será alvo do ataque.

Para isto, foram desenvolvidas duas aplicações, contendo um painel de usuário simples que chamaremos de "*Aplicação Legítima*", onde neste painel há uma página "Minha conta", Figura 24, responsável por fazer alterações nas informações do usuário, como nome, email, endereço, senha, etc. Estas alterações são feitas via formulário POST e necessitam autenticação via formulário de *login*, Figura 25.

Ambas aplicações possuem implementações de segurança para as demais vulnerabilidades citadas.

Figura 24 – Minha conta, página para gerenciar informações do usuário.
Fonte: Autor.

Figura 25 – Formulário de *login* em HTML para vulnerabilidade Cross Site Request Forgery
Fonte: Autor.

3.6.1 APLICAÇÃO A

Aplicacao A foi escrita como a grande maioria dos sistemas atuais, através do envio de dados via POST, é feita as validações das credenciais do usuário, para então atualizar os registros em banco.

Na página ‘Minha Conta’ há a verificação da existência de sessão de usuário, caso contrário, redireciona o usuário à página de *login*, conforme Figura 26.

```php
<?php
      session_start();

      if (!isset($_SESSION['logado'])){
            header('Location: login.php');
            exit;
      }
?>
```

Figura 26 – Verificação de sessão do usuário.
Fonte: Autor.

Para a alteração dos dados, fora utilizado código com confirmações e tratamento de sentenças, contra SQL Injection e outras injeções, Figura 27.

```php
<?php
if (count($_POST)){
$Label = $pdo->prepare('INSERT INTO usuario (nome,email) VALUES (?,?,?)');
$Label->bindValue(1,$_POST['nome']);
$Label->bindValue(2,$_POST['email']);
$Label->execute();
echo "Dados atualizados com sucesso.";
die();
?>
```

Fi

Figura 27 - Código da alteração dos dados de usuário, com confirmação e tratamento de sentenças para bloquear SQL Injection e outras injeções, vulnerável à CSRF.
Fonte: Autor.

3.6.2 APLICAÇÃO B

Para a Aplicação B, foi integrado um sistema CAPTCHA, Figura 28, que é uma implementação de segurança desenvolvida em PHP, que força o usuário à identificar letras de uma imagem distorcida, provando assim ser uma pessoa real.

Para esta aplicação foi implementado o sistema CAPTCHA Securimage (PHPCAPTCHA, 2012).

Figura 28 – Sistema CAPTCHA em formulário de alteração de dados.

Fonte: Autor.

Fora utilizado o mesmo código de verificação das credenciais de *login* que a aplicação A. E o código, Figura 29, para realizar a alteração dos dados de usuário.

```php
<?php
if (count($_POST)){
 $Label = $pdo->prepare('INSERT INTO usuario (nome,email) VALUES (?,?,?)');
 $Label->bindValue(1,$_POST['nome']);
 $Label->bindValue(2,$_POST['email']);
 // Bloqueio CSRF
 require_once('securimage/securimage.php');
 $image = new Securimage();

 //verifica se captcha é valido
 if ($image->check($_POST['captcha']) == true) {
      $Label->execute();
      echo "Dados atualizados com sucesso.";
      die();
 } else {
      echo "Código incorreto, favor redigite o código.";
      die();
 }
?>
```

Figura 29 - Alterações de dados com implementação de sistema CAPTCHA, bloqueando CSRF.
Fonte: Autor.

4 RESULTADOS

Para cada vulnerabilidade citada, foram desenvolvidas aplicações práticas e funcionais a fim de demonstrar a ação de um proveniente ataque, e suas comprovadas formas de defesa. A seguir os resultados para cada ataque proveniente de uma vulnerabilidade *web.*

4.1 SQL INJECTION

As aplicações A e B possuem como dados válidos para autenticação o nome de usuário **admin** e a senha **adminsenha**, que garantem o acesso à aplicação como mostra a Figura 30 o formulário de *login* construído em HTML.

A Figura 31, demonstra o resultado de execução ao utilizar os dados válidos citados anteriormente, este resultado contém o usuário digitado, a senha criptografada em *MD5*, a *query* SQL executada no banco e a resposta da autenticação, neste caso o resultado é garantido com sucesso. Conforme Figura 32, foram inseridos dados inválidos para usuário e senha, **admin** e **senhaerrada** respectivamente, e o resultado demonstra a autenticação não sucedida.

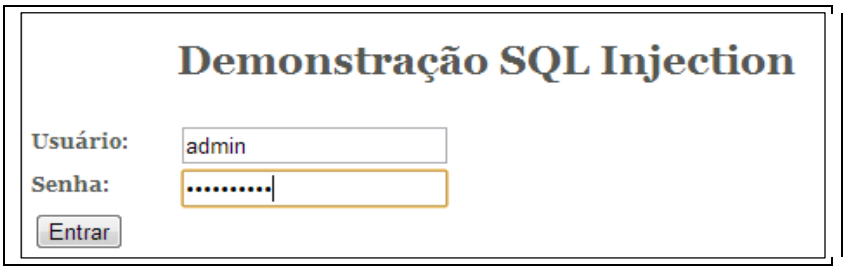

Figura 30 – Formulário de *login* HTML.

Fonte: Autor.

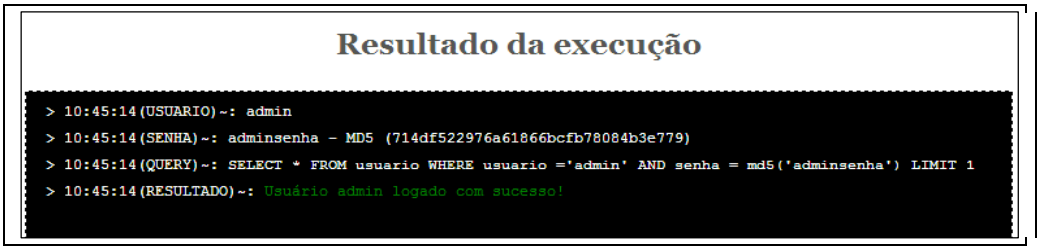

Figura 31 – Resultado de execução utilizando dados válidos.

Fonte: Autor.

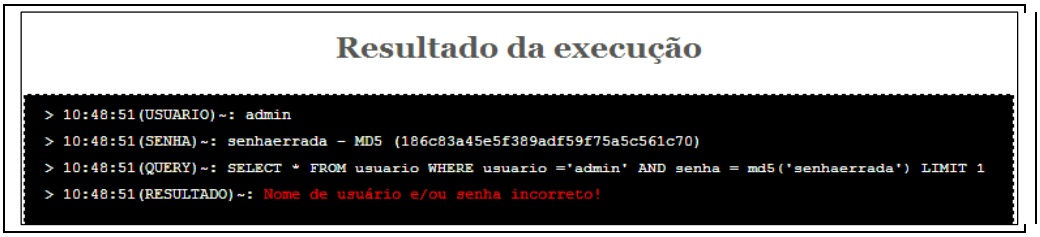

Figura 32 – Resultado de execução utilizando dados inválidos.
Fonte: Autor.

Para *SQL Injection*, foram simulados dois diferentes ataques nas aplicações A e B.

4.1.1 ATAQUE 1

Assumindo que o atacante desconheça qualquer dado da tabela ou dos usuários, como nome de campos, ou um nome de usuário válido por exemplo. Para isto foi inserido no campo usuário a injeção **‘ or 1=1;--** , Figura 33.

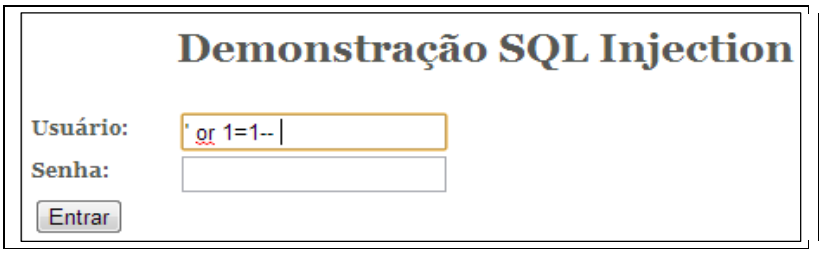

Figura 33 – Simulação de injeção SQL 1.
Fonte: Autor.

4.1.1.1 APLICAÇÃO A

Ao receber o parâmetro **‘ or 1=1--** para o campo usuário**,** a aspa no inicio da sentença fechou a *string* relativa ao nome de usuário e inseriu-se uma condição (ou 1=1) que sempre será verdadeira, porém só está condição retornaria falsa, pois a validação de senha ainda estava ativa, então ao adicionar os dois traços no final da sentença todo o resto da *query* e suas condições foram comentadas e ignoradas pelo banco (Figura 35), resultando na seguinte sentença que sempre retornará verdadeira, Figura 34.

```
SELECT * FROM usuario WHERE usuario = '' OR 1=1
```

Figura 34- Query SQL com injeção de condição sempre verdadeira.
Fonte: Autor.

No caso da seleção de usuário, Figura 34, a consulta resulta no primeiro registro encontrado na tabela *usuario,* um registro que poderia não possuir privilégios administrativos, podendo ser um usuário comum do sistema, porém certamente garante um acesso não autorizado à aplicação.

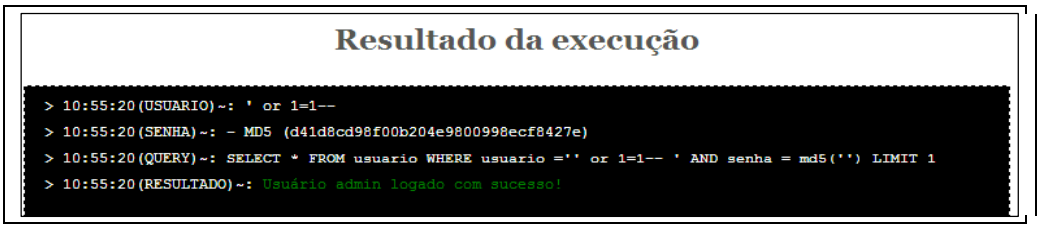

Figura 35 – Injeção SQL bem sucedida, acesso garantido com o primeiro registro da tabela.
Fonte: Autor.

4.1.1.2 APLICAÇÃO B

Já na aplicação B, ao receber como parâmetro de usuário a injeção **‘ or 1=1--** a função do PDO *prepare*, prepara esta *string* para uma sentença SQL válida, através de diversos testes e validações confirmadas que evitam e bloqueiam qualquer tipo de injeção SQL, segundo PHP “Se um aplicativo usa exclusivamente instruções preparadas, o desenvolvedor pode ter certeza de que não irá ocorrer a injeção de SQL.”.

A Figura 36 confirma que a utilização de instruções preparadas evitam com sucesso injeções de SQL. Nota-se que na *query* originada do método preparado, não há o fechamento da comparação no campo *usuario*, pois o PDO escapa a aspa digitada, adicionando barra invertida (\) na frente, fazendo com que a sentença seja reconhecida pelo SQL como uma *string* e não como um comando ou instrução SQL. Não criando então a comparação 1=1, nem comentando o restante da sentença, esta então é verificada como deveria e resulta em uma autenticação não sucedida, bloqueando com sucesso o ataque.

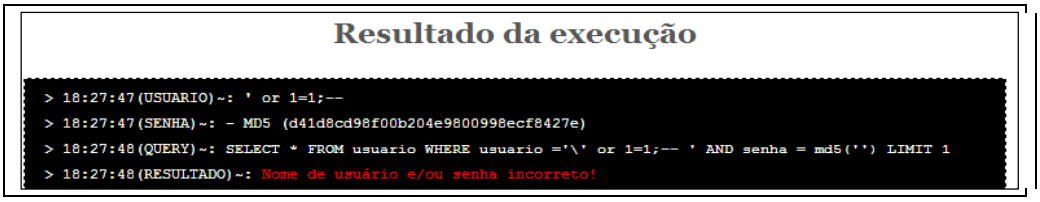

Figura 36 – Bloqueio do Ataque 1 com escape e tratamento das entradas.
Fonte: Autor.

4.1.2 – ATAQUE 2

No Ataque 1 assumiu-se que o atacante não tenha conhecimento de nenhum nome de usuário da aplicação, resultando assim na autenticação do primeiro registro da tabela, um usuário que pode não conter privilégios administrativos, o que resultaria em menores danos. Já o Ataque 2, assumiu-se que o atacante tenha conhecimento através de tentativa e erro ou outros métodos não digitais da existência do nome de usuário **admin**, este contendo total controle da aplicação e privilégios ilimitados.

Foi inserido no campo usuário a injeção SQL **admin'--,** Figura 37.

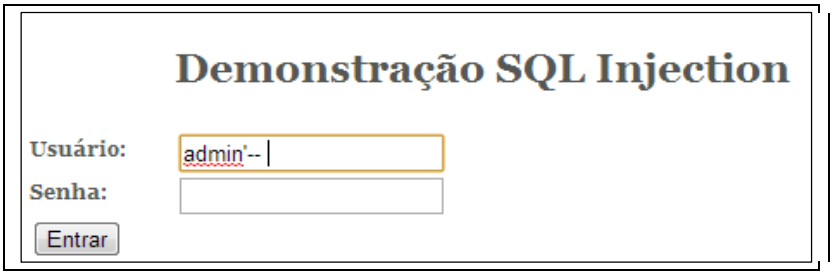

Figura 37 – Simulação de injeção SQL 2.
Fonte: Autor.

4.1.2.1 – APLICAÇÃO A

Ao receber o parâmetro **admin'--** para o campo usuário**,** a aspa após o termo *admin* fecha a *string* relativa ao nome de usuário, e então ao adicionar os dois traços no final da sentença todo o resto da *query* e suas condições foram comentadas e ignoradas pelo banco, Figura 39, resultando na seguinte sentença que trás o registro referente ao usuário admin, Figura 38.

```
SELECT * FROM usuario WHERE usuario = 'admin'
```

Figura 38 – Query SQL com injeção, resultando no usuário admin.
Fonte: Autor.

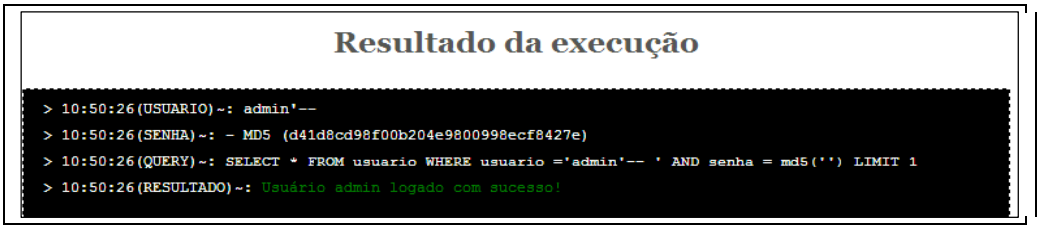

Figura 39 – Injeção SQL sucedida, acesso administrativo garantido.
Fonte: Autor.

4.1.2.2 – APLICAÇÃO B

Na aplicação B, como no Ataque 1 o método de instruções preparadas o PDO escapou todos os caracteres inválidos para uma sentença SQL e fez o tratamento da entrada para ser reconhecida sempre como uma *string* não uma instrução SQL, adicionando então a barra invertida antes da aspa e removendo a *tag* comentário *(--)*, tornando assim ineficaz a injeção SQL, Figura 40.

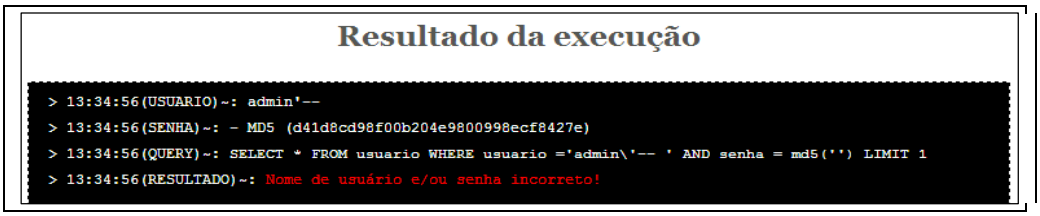

Figura 40 – Bloqueio do Ataque 2 com escape e tratamentos.
Fonte: Autor.

4.2 CROSS SITE-SCRIPTING (XSS)

As aplicações A e B têm como objetivo receber um termo de pesquisa no formato *string*, executar a consulta (não é o foco da vulnerabilidade), e exibir o resultado dessa consulta junto com o termo digitado, Figura 41.

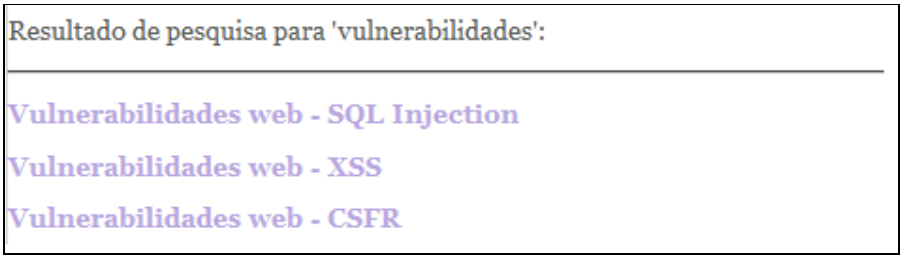

Figura 41 - Resultado de consulta com termo "vulnerabilidade".
Fonte: Autor.

4.2.1 ATAQUE

Foi incluso após o termo de pesquisa um arquivo JavaScript externo. Este arquivo tem o objetivo de enviar para uma página PHP externa pertencente ao atacante, informações como o *PHPSESSID* (ID de sessão) automaticamente sem o conhecimento da vítima, este ID de sessão poderá ser utilizado para efetuar *session hijacking* que será descrito no próximo item. O arquivo incluso é denominado xss.js e possui o seguinte código, Figura 42.

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'ataque_xss.php', true);
xhr.setRequestHeader('Content-type','application/x-www-form-urlencoded');
xhr.onload = function () {
    // do something to response
    console.log(this.responseText);
};
xhr.send('PHPSESSID='+document.cookie);
```

Figura 42 - Arquivo xss.js, responsável por enviar ID de sessão para página maliciosa.
Fonte: Autor.

Pelo fato da página possuir um formulário do Method *GET,* os parâmetros enviados são exibidos na URL da mesma.

Para um ataque XSS funcionar, é necessário que a vítima de alguma forma seja direcionada à esta *URL* com a injeção do código JavaScript, e esteja logada na aplicação, pois ataques XSS são executados *client-side* ou seja, no próprio navegador da vítima.

Para atrair a vítima, pode ser utilizado *banners* de propaganda com o *link* alterado, *e-mails* falsos, direcionamento através de outros *sites*, entre outras formas que não são o foco da vulnerabilidade em si.

Portanto, o ataque consiste em atrair a vítima para a URL que contém como parâmetro ‘termo’ a injeção XSS, resultando na URL evidenciada na Figura 43.

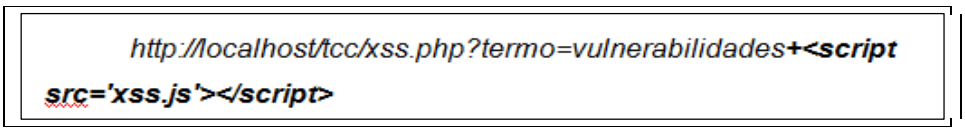

Figura 43 - Url com injeção XSS.
Fonte: Autor.

4.2.1.1 APLICAÇÃO A

O resultado do ataque na Aplicação A foi como o esperado, a vítima julgou ser uma URL segura por ter o domínio e credenciais válidos da aplicação em que costuma logar-se, mas ao entrar nesta URL desconhece o fato de que seu ID de sessão foi enviado para um site malicioso, no qual será armazenado e poderá resultar em futuros ataques de *session hijacking*. Para a vítima, a página de consulta está funcionando normalmente, exibindo os resultados de busca para o termo ‘vulnerabilidades’, Figura 44.

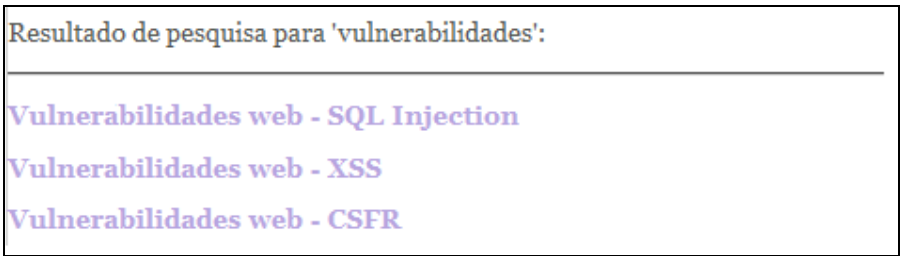

Figura 44 - Resultado da URL maliciosa na visão da vítima.
Fonte: Autor.

Porém, ao analisar o código fonte e elementos HTML da página, encontra-se o arquivo JavaScript *xss.js* incluso após o termo ‘vulnerabilidades’ como pode ser visto na Figura 45.

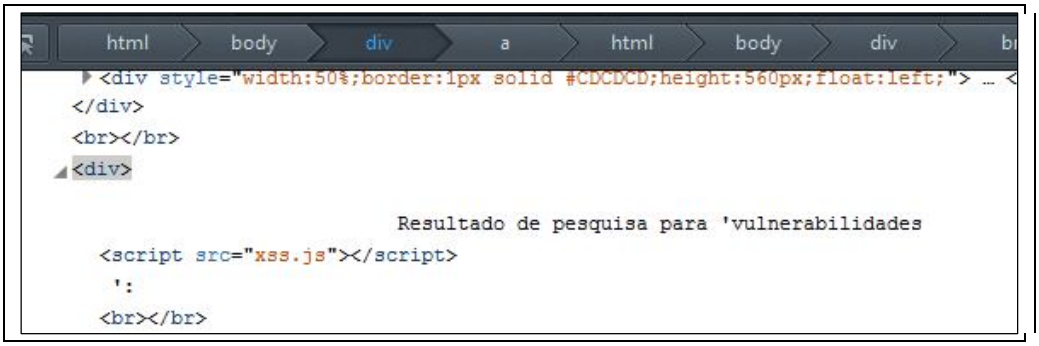

Figura 45 – Injeção de código JavaScript, ataque XSS.

Fonte: Autor.

Ao incluir este arquivo (xss.js), automaticamente ao carregar a página, é realizada uma requisição POST via AJAX, que envia os dados da sessão atual, Figura 46.

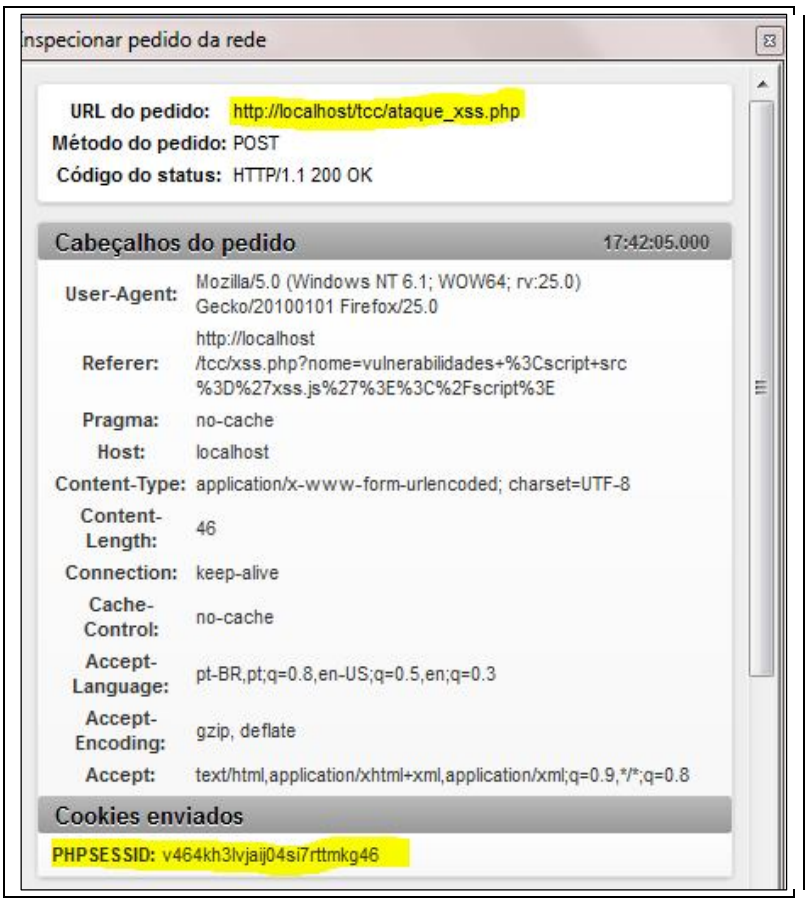

Figura 46 – Envio do PHPSESSID via AJAX para atacante.

Fonte: Autor.

Portanto, a aplicação A sofreu com sucesso um ataque XSS, e teve suas informações sigilosas enviadas para um atacante que poderá, por meio de outras técnicas, garantir um acesso não autorizado e causar prejuízos à aplicação.

4.2.1.2 APLICAÇÃO B

O ataque não foi bem sucedido na aplicação B, mesmo com a vítima clicando na URL maliciosa contendo a inclusão de um arquivo JavaScript, pois ao receber o termo de busca via formulário GET ou pela URL, o mesmo é tratado e escapado, tanto os caracteres especiais (ex. aspas e ponto e vírgula), como as *tags* HTML (ex:*tag* <SCRIPT> que inclui um novo *script* JS).

A Figura 47 mostra que a mesma URL maliciosa, ao ser carregada, não exibe o código JavaScript responsável pelo envio de dados à um servidor externo.

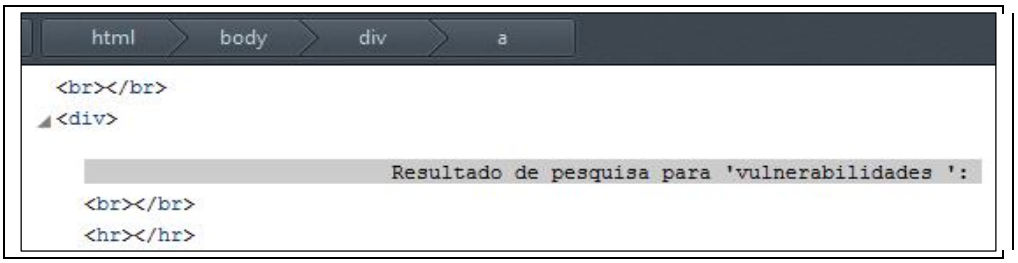

Figura 47 – Escape das *tags* HTML e caracteres especiais. XSS não sucedido.
Fonte: Autor.

Comprova-se, desta forma, que as funções PHP *strip_tags* e *htmlentities* bloqueiam com sucesso injeções XSS, e tornam a aplicação segura para receber entradas do usuário que serão exibidas na tela.

Notou-se ao longo dos testes das aplicações, notou-se que o único navegador em que o ataque de XSS não funcionou foi o Google Chorme®. Após pesquisa bibliográfica (GOOGLE, 2010) descobriu-se que por padrão o navegador da Google já possui verificação e bloqueio para injeções XSS, através do tratamento das URLs. Sendo assim, é um navegador seguro e recomendado para navegação em sites que possam ser vulneráveis à ataques XSS.

4.3 FILE UPLOAD ATTACK

Existem inúmeros arquivos, que enviados à um servidor web através de uma brecha de segurança, podem causar danos, vazamento de informações e comprometimento da aplicação. Entanto, poucos podem ser tão prejudiciais quanto o envio do arquivo *hacker* conhecido como *shellc99.php*, este arquivo é um sistema desenvolvido em PHP que simula um FTP, exibindo arquivos e diretórios do servidor, e possibilitanto a visualização, alteração e exclusão dos mesmos, além de novos *uploads* e até execuções de comandos (Linux).
As aplicações A e B têm como objetivo o cadastro e *upload* de currículos no formato documento de texto do Microsoft Word (.doc ou .docx), para o dono da aplicação poder baixá-los via URL, Figura 48.

Figura 48 – Download de currículo via URL.
Fonte: Autor.

4.3.1 ATAQUE

A simulação de ataque para esta vulnerabilidade, foi, o envio do já citado arquivo *shellc99.php* para ambas aplicações, Figura 49. Caso o envio seja bem sucedido, o atacante precisaria através de tentativa e erro, descobrir o diretório em que o arquivo foi salvo, no caso das aplicações testadas, foram salvos no diretório *curriculos*.

Figura 49 – Upload de arquivo *shellc99.php* através de formulário HTML.
Fonte: Autor.

4.3.1.1 APLICAÇÃO A

Por não conter nenhuma verificação de tipo do arquivo, o arquivo *shell99.php* foi enviado com sucesso na aplicação A, Figura 50.

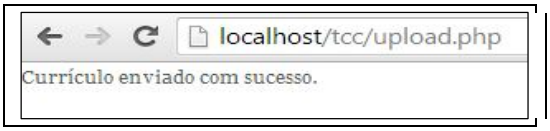

Figura 50 – Envio de arquivo *shell99.php* bem sucedido.
Fonte: Autor.

Deduzindo que o atacante descubra através de tentativa e erro o diretório em que o arquivo foi salvo, o resultado foi controle **total** do código fonte da aplicação. Ao acessar a URL *http://localhost/tcc/curriculos/shell99.php* o atacante acessa o 'FTP' da aplicação , Figura 51 e Figura 52.

Figura 51 - Shellc99 em execução, listagem de arquivos e diretórios.

Fonte: Autor.

Figura 52 - Diretório de aplicação teste listado pelo *script* Shellc99.

Fonte: Autor.

Através do sistema Shell foi possível visualizar o arquivo de configuração que contém todos os dados de conexão da aplicação, Figura 53.

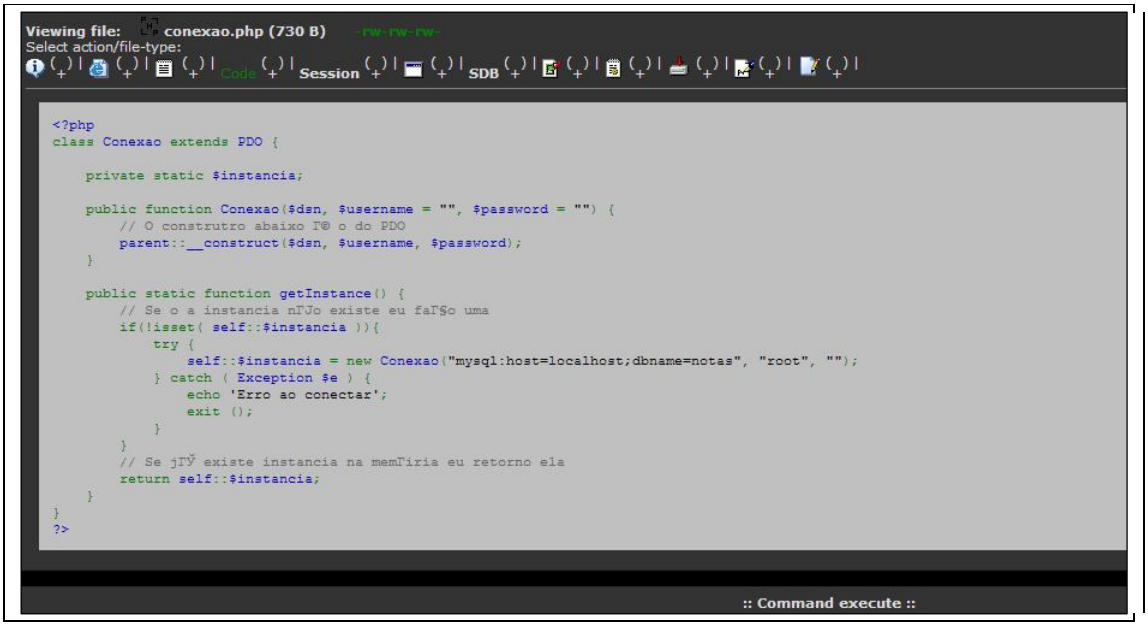

Figura 53 - Arquivo *conexao.php*, contendo dados de conexão com a base de dados, listado pelo *script* Shellc99.
Fonte: Autor.

A aplicação A ao aceitar o envio do arquivo de extensão PHP, tornou-se totalmente vulnerável, com potencial de prejuízo total, pois o atacante tem acesso à todo o código fonte da aplicação, e a possibilidade de alterá-los sem restrições, podendo assim furtar,divulgar e colher novas informações sigilosas indiscriminadamente.

4.3.1.2 APLICAÇÃO B

Utilizando-se do resultado da Aplicação A, assumindo que o arquivo shellc99.php já tenha sido, ou já exista no diretório *curriculos*, as configurações contidas no arquivo **.**htaccess criado no diretório currículo da Aplicação B, bloqueiam a listagem e o acesso desse arquivo via navegador, ou seja, mesmo sendo possível o envio do arquivo malicioso, o atacante não pode acessar e executar o arquivo no servidor, Figura 54.

Figura 54 – Bloqueio .htaccess, acesso permitido apenas para arquivos de extensão *.doc* ou *.docx.*
Fonte: Autor.

Apenas esta implementação de segurança ainda não seria o suficiente, pois arquivos maliciosos ainda poderiam ser enviados ao servidor. A verificação do tipo MIME do arquivo enviado na Aplicação B, garantiu um *upload* seguro para arquivos do tipo *application/msword* (.doc, .docx), bloqueando as demais extensões de arquivo (como .php do ataque em questão) com sucesso, Figura 55.

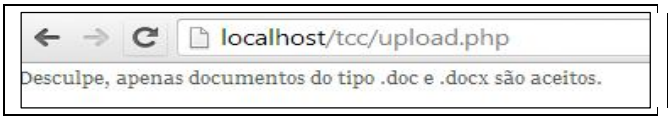

Figura 55 – Bloqueio de extensões não permitidas no momento do upload.
Fonte: Autor.

4.4 SESSION HIJACKING

Existe um complemento para navegadores como *Mozilla Firefox* e *Google Chrome* denominado Cookie Editor, se trata de um gerenciador de cookies. Onde pode-se adicionar, excluir, editar, procurar, proteger e bloquear cookies.

4.4.1 ATAQUE

Foi simulado o ataque, levando-se em conta que o atacante já possua o valor da sessão (*PHPSESSID*), que pôde ser obtido de diversas formas, como *links*

maliciosos, códigos XSS, entre outros. Este exemplo complementa o ataque XSS, em que o atacante recebe através de uma requisição AJAX, o *PHPSESSID* (ID de sessão) ativo na máquina da vítima. Com este PHPSESSID em mãos, utilizou-se a extensão Cookie Editor para modificar o valor de sessão que é automaticamente criada no carregamento da página. Para isto foi utilizada uma janela normal do navegador, demonstrando uma autenticação bem sucedida com dados reais de *login* , Figura 56 e Figura 57, está sessão válida resultou em um valor do *PHPSESSID* igual a “mgp7scrvj2h9d3hos2pnn54vp1”.

Figura 56 – Autenticação com dados válidos **admin** e **adminsenha.**

Fonte: Autor.

Figura 57 – Mensagem de boas vindas para usuário **admin.**

Fonte: Autor.

Para o ataque em si, foi utilizada uma nova janela anônima do navegador, janela que não contém dados de sessão ou cookies. Nesta janela foi ativada a extensão Cookie Editor (Figura 58), carregou-se a página de login da aplicação e então foi modificado o valor do *PHPSESSID* para o valor da sessão válida da vítima, obtido via XSS, no caso “mgp7scrvj2h9d3hos2pnn54vp1”, Figura 59.

Figura 58 – Extensão Cookie Editor em funcionamento no navegador *Google Chrome*.
Fonte: Autor.

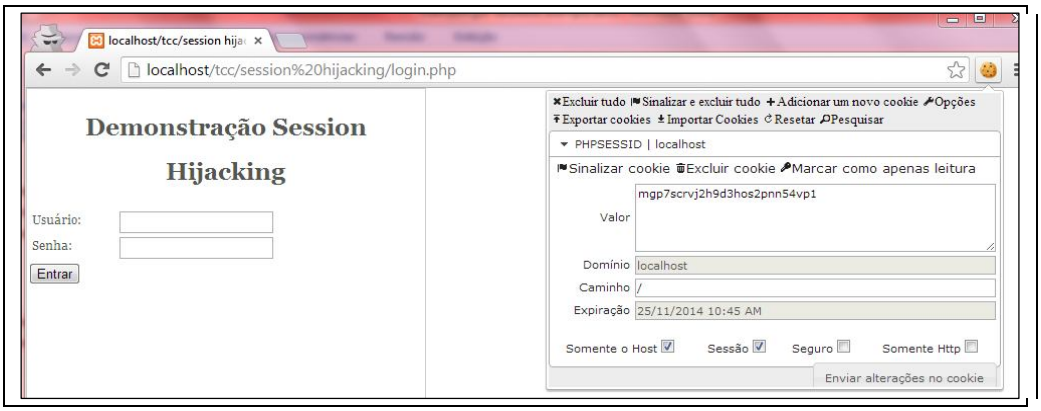

Figura 59 – Alteração do valor do *cookie PHPSESSID* para sessão válida da vítima.
Fonte: Autor.

4.4.1.1 APLICAÇÃO A

A aplicação A por utilizar as configurações padrões em seu servidor APACHE, sofreu um seqüestro de sessão bem sucedido. Ao alterar os valores de sessão com o Cookie Editor para a sessão válida da vítima, e em seguida atualizar a página, o atacante garantiu acesso à aplicação em nome da vítima com suas credenciais válidas, Figura 60.

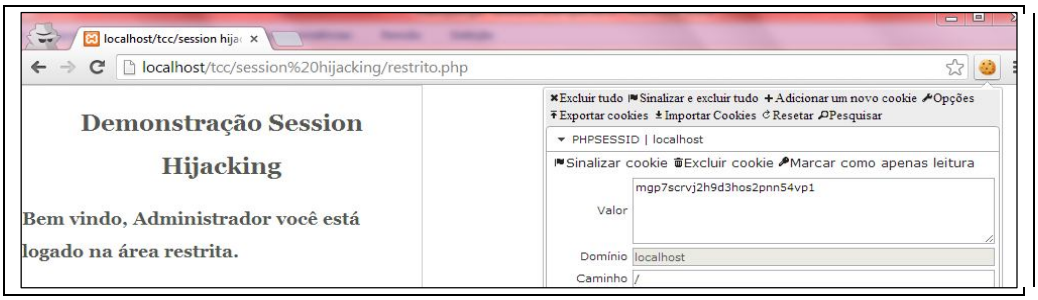

Figura 60 – Acesso em nome da vítima através de Session Hijacking**.**
Fonte: Autor.

A Aplicação A em conjunto com os testes de XSS provaram que o Session Hijacking é uma vulnerabilidade existente em inúmeras aplicações *web*, e de alto risco para as mesmas.

4.4.1.2 APLICAÇÃO B

A Aplicação B, por utilizar configurações personalizadas do servidor APACHE, como a habilitação do uso de *cookies* apenas (*session.use_only_cookies),* e implementações de segurança (*session.cookie_secure)*, impediram com sucesso um ataque de Session Hijacking, mesmo a vítima de alguma forma vazando suas credenciais de sessão (*PHPSESSID*) para o atacante.

Primeiramente, ao abrir a extensão Cookie Editor, já nota-se que não há mais o registro de sessão *PHPSESSID,* Figura 61, isso por que a configuração APACHE *session.use_only_cookies* especifica que a aplicação usou ***somente*** cookies para guardar a ID. Segundo PHP, “habilitando esta configuração previne ataques envolvendo passagem de ID de sessão nas URLs”. Já a configuração *session.cookie_secure* garante que todos os cookies originem de uma conexão segura, impossibilitando assim a alteração do valor do mesmo. A combinação das duas configurações resultam em uma aplicação segura contra tentativas de Session Hijacking, a Figura 62, demonstra que mesmo criando um novo valor de sessão com nome *PHPSESSID*, o atacante não garantiu acesso à aplicação.

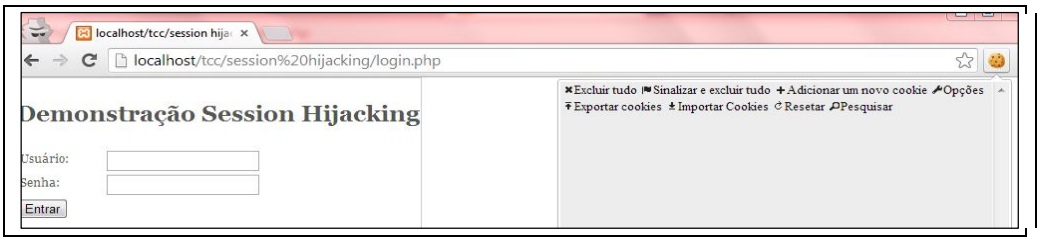

Figura 61 – Cookies protegidos contra Session Hijacking**.**
Fonte: Autor.

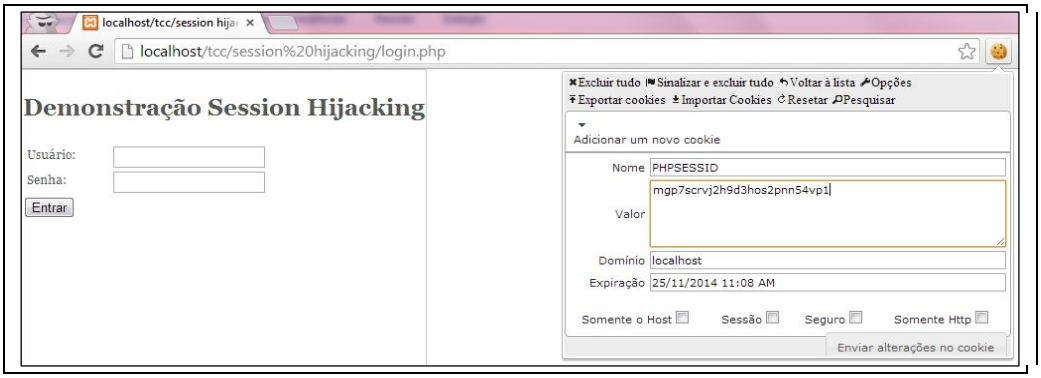

Figura 62 – Tentativa de criação de cookie *PHPSESSID* com valor de sessão válida**.**
Fonte: Autor.

4.5 DIRECTORY TRAVERSAL

Ataques de *directory traversal* são muito comuns e aplicações que incluem arquivos via URL ou outros métodos. A função de inclusão *include* do PHP, inclui um arquivo a partir de um caminho absoluto, não apenas por seu nome, desta forma, pode-se subir diretórios e subdiretórios, incluindo arquivos que não se encontram na raiz, ao serem inclusos, a função irá executar o arquivo em questão.

4.5.1 ATAQUES

A simulação de ataque para direcotry traversal faz justamente isto, a inclusão de um arquivo que não se encontra na pasta de páginas HTML da aplicação. A aplicação pretende incluir apenas os arquivos HTML existentes na sua pasta raiz.

O servidor de teste configurado possui o seguinte caminho absoluto: *I:\xampp\htdocs,* a pasta *htdocs* é a pasta raiz da aplicação, nela encontra-se o template PHP (*traversal.php*) e as páginas HTML que devem ser inclusas.

O ataque consiste em sair desta pasta e incluir os arquivos de configurações do APACHE ou até mesmo arquivos de configuração do Sistema Operacional. Como foi utilizado um servidor Windows, a quebra de diretórios pode ser entendida como barra normal (/) ou barra invertida (\), diferente de outros sistemas operacionais em que aceita-se somente barras normais (/).

Através de tentativa e erro um atacante pode descobrir o diretório das configurações no sistema, e visualizá-los no navegador.

4.5.1.1 ATAQUE 1

A fim de visualizar as configurações do servidor APACHE, foi inserido uma URL como evidencia a Figura 63.

Figura 63 – Directory traversal incluindo arquivo de configuração APACHE**.**
Fonte: Autor.

4.5.1.2 ATAQUE 2

Com intuito de listar dados do sistema operacional, foi carregada então a URL como mostra a Figura 64.

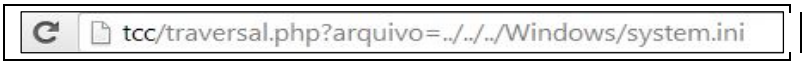

Figura 64 – Directory Traversal incluido dados do sistema operacional**.**
Fonte: Autor.

4.5.2 APLICAÇÃO A

A Aplicação A por tentar incluir diretamente o caminho recebido, sem tratar ou validar a existencia do mesmo, mostrou-se vulnerável à ataques de Directory

Traversal. Primeiramente como mostra a Figura 65, ao tentar incluir um arquivo inexistente, ou um diretório inexistente, no caso (../) , o PHP exibe um erro na tela, dizendo que o arquivo em questão não foi encontrado, e exibe o diretório absoluto em questão (Figura 66), entregando assim ao atacante o caminho da aplicação no servidor *(I:\xampp\htdocs).*

Figura 65 – Provocação de erro para exibição do caminho absoluto**.**
Fonte: Autor.

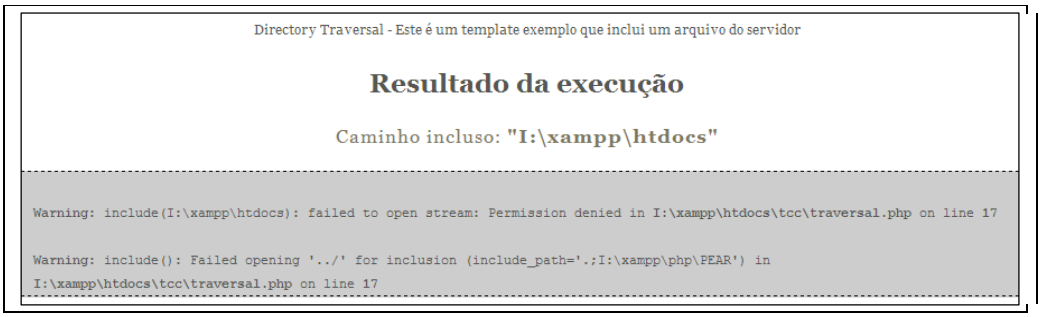

Figura 66 – Erro de inclusão, arquivo inexistente.
Fonte: Autor.

4.5.2.1 ATAQUE 1

Para a Aplicação A, o resultado do primeiro ataque foi a exibição de toda a configuração do servidor APACHE, Figura 67.

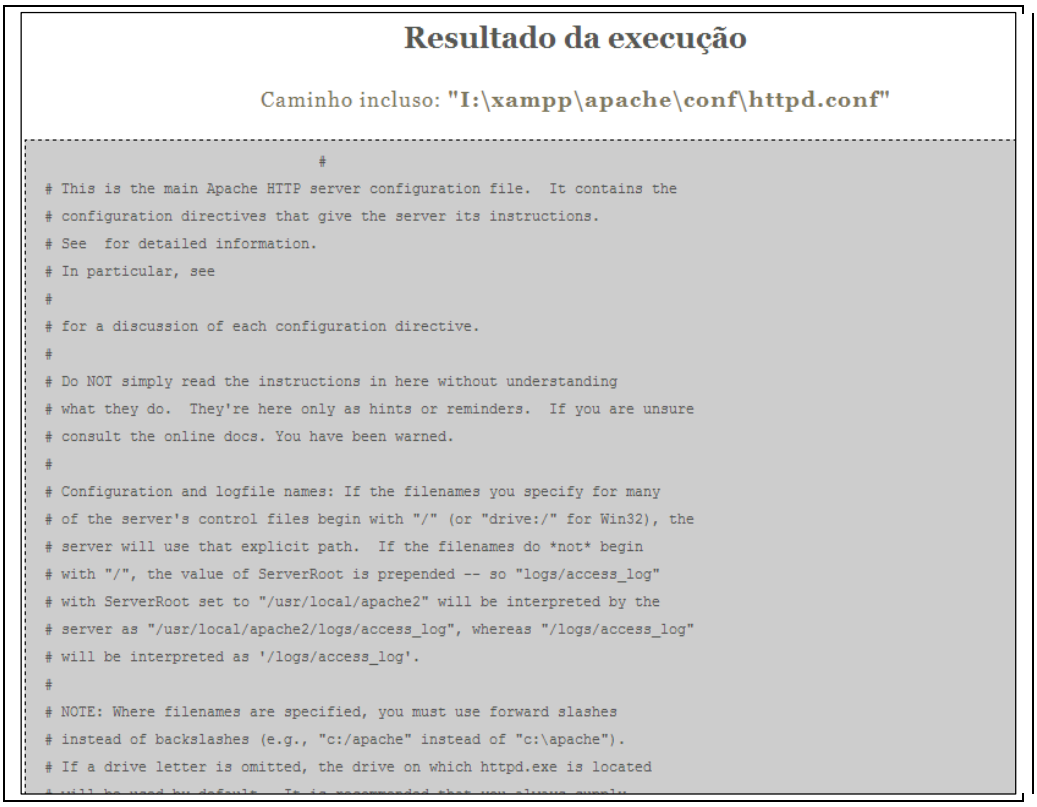

Figura 67 – Inclusão de configuração do APACHE via Directory Traversal**.**
Fonte: Autor.

4.5.2.2 ATAQUE 2

Indo mais afundo, o Ataque 2, na Aplicação A foi capaz de listar as configurações do próprio sistema operacional que *hosteia* a aplicação, Figura 68.

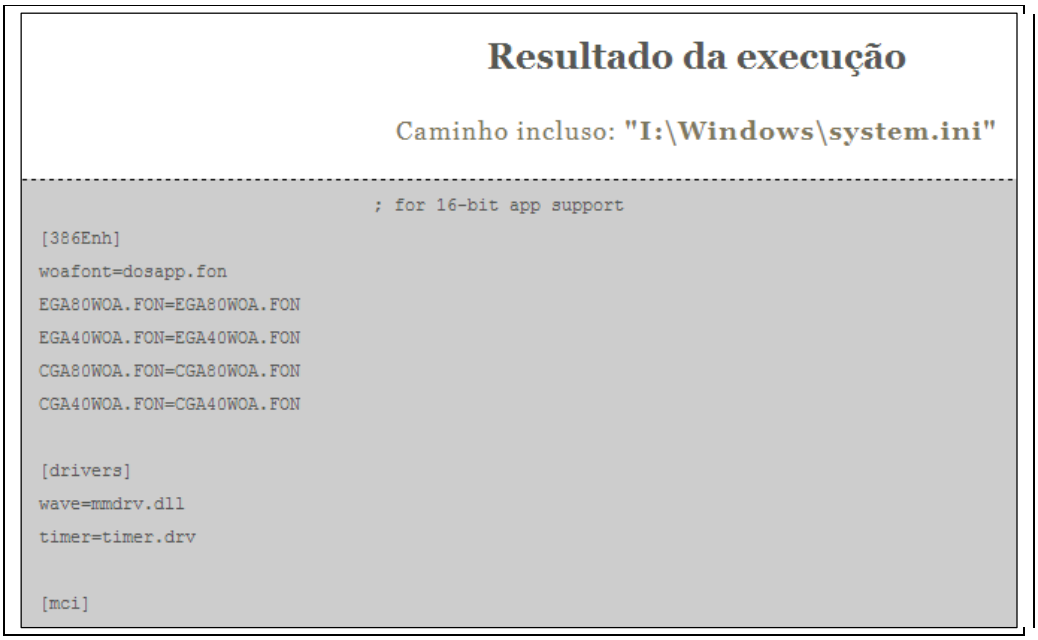

Figura 68 - Inclusão de configuração do Windows via Directory Traversal**.**

Fonte: Autor.

4.5.3 APLICAÇÃO B

As validações e implementações da Aplicação B, impediram com sucesso um ataque de Directory Traversal, pois primeiramente, ao verificar a existência do arquivo, caso o mesmo não seja encontrado ou não exista como fora evidenciado na Figura 41, o PHP não exibirá o erro padrão que mostra o diretório da aplicação, como ocorreu na Aplicação A, mas sim apenas uma mensagem de erro personalizada (Figura 69).

Figura 69 – Arquivo não encontrado**.**

Fonte: Autor.

4.5.3.1 ATAQUES

Na Aplicação B, ambos os ataques tiveram o mesmo resultado “404 Página não encontrada.”. Isto aconteceu por que, a função *basename* retorna apenas o nome do arquivo, e não seu diretório, eliminando assim todas as tentativas de alteração de diretório (../../).

A Aplicação B provou que tratar a existência e o caminho do arquivo à ser incluso protegem com sucesso uma aplicação contra ataques de Directory Traversal.

4.6 CROSS SITE REQUEST FORGERY

O Cross Site Request Forgery só é possível caso a vítima, usuária da ‘Aplicação Legítima’ (aplicação alvo do ataque), seja atráida por meio de técnicas variadas, que não são o foco da vulnerabilidade (*fishing*, propagandas, emails falsos, etc), à uma página maliciosa, Figura 70, que contenha um formulário HTML, com campos **escondidos,** Figura 71, que submetem para a página real da aplicação, esta página requer devida autenticação. Esta página maliciosa pode conter conteúdo diversos, ou mesmo ser uma cópia da própria aplicação.

Figura 70 – Site do atacante, página com formulário escondido**.**

Fonte: Autor.

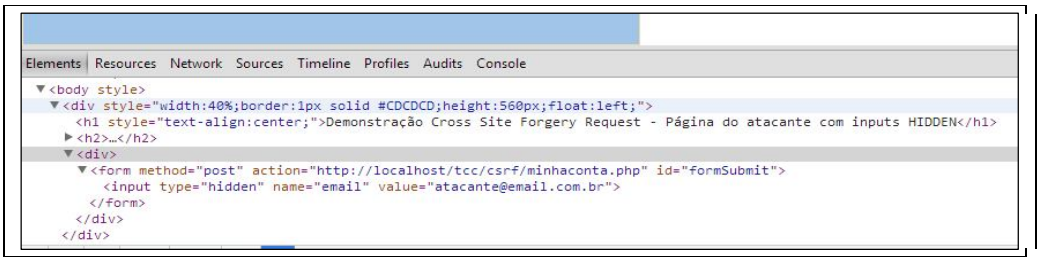

Figura 71 – Formulário escondido com *action* direcionada à 'Aplicação Legítima'**.**
Fonte: Autor.

4.6.1 ATAQUE

Para realizar o teste, assumiu-se que uma vítima logada (com acesso garantido) na 'Aplicação Legítima', fora atraída para uma página maliciosa.
Assim que ele carrega a página, sem seu conhecimento, o formulário escondido é submetido via JavaScript para a 'Aplicação Legítima', através do seguinte código jQuery, Figura 72.

```
<script type="text/javascript">
      $(function(){
            $('#formSubmit').submit();
      });
</script>
```

Figura 72 - Código jQuery para envio automático de formulário HTML.
Fonte: Autor.

4.6.1.1 APLICAÇÃO A

Na Aplicação A, no momento em que a vítima entra na página maliciosa e o formulário é submetido via JavaScript, seus dados são alterados sem seu consentimento, Figura 73, pois a vítima estava *logada* na aplicação, e possuía então credenciais válidas para através de uma requisição POST fazer alteração em seus dados. O atacante então alterou o email da vítima, para um email de seu acesso, podendo assim usar funções como "Recuperar Senha" para enviar uma

nova senha à ele, e receber assim acesso à aplicação. A Aplicação A sofreu com sucesso um ataque de Cross Site Request Forgery.

Figura 73 – Cross Site Request Forgery bem sucedido, dados da vítima alterados vi página maliciosa**.**
Fonte: Autor.

4.6.1.2 APLICAÇÃO B

A Aplicação B por necessitar o preenchimento e validação de um CAPTCHA, bloqueou com sucesso um ataque de Cross Site Request Forgery. Pois o código CAPTCHA é gerado apenas para a página 'Minha Conta', sendo assim, o atacante não há como prever o código que será gerado em uma solicitação válida, ou seja, dentro da aplicação. Desta forma, ao tentar submeter um formulário situado em uma página externa, recebe a mensagem de falha na digitação do código CAPTCHA, Figura 74.

Sendo assim, utilizar CAPTCHA para formulários que alteram informações sigilosas em aplicações fechadas, é um modo eficaz de proteção contra ataques provenientes da vulnerabilidade Cross Site Request Forgery.

Figura 74 – CAPTCHA impedindo alteração de conteúdo via páginas de terceiros**.**
Fonte: Autor.

5 CONSIDERAÇÕES

Este trabalho demonstrou através de referencial teórico e experimentos em aplicações práticas a existência e funcionalidades das vulnerabilidades mais comuns em aplicações *web* atuais.

Concluiu-se que as linguagens de programação *web* atualmente possuem funções nativas para combater estas vulnerabilidades, como por exemplo, a linguagem PHP que contém a nova biblioteca *PDO* (PHP Data Object), que ao ser utilizada corretamente, exclui qualquer possibilidade de uma injeção SQL; a função *strip_tags* e *htmlentities*, funcionam efetivamente na eliminação de qualquer *caracter* ou *tag* não permitido nas entradas do usuário, prevenindo assim ataques como XSS. Os desenvolvedores devem ser conscientizados sobre a existência e riscos destas e outras vulnerabilidades, a segurança deve ser priorizada, e a linguagem optada para desenvolvimento deve ser estudada à fundo, pois a mesma já oferece implementações de segurança cabíveis e confiáveis.

Em geral, tratar a entrada do usuário é essencial para o desenvolvimento de uma aplicação segura, foi provado que nunca deve-se utilizar os parâmetros recebidos, diretamente na aplicação, pois há diversas vulnerabilidades existentes que utilizam deste método para formalizar um ataque.

Porém, não só o desenvolvedor deve ser conscientizado, mas o usuário ao navegar na internet também deve ficar atento, não clicando em *links* suspeitos ou informando dados pessoais em sites não confiáveis. Fora detectado também, que alguns navegadores como o *Google Chrome* já possuem por padrão uma implementação de segurança contra XSS, mesmo em sites e aplicações que possuam esta vulnerabilidade, tornando-se assim, uma boa ferramenta para o usuário previnir-se.

Para aqueles que visam dar continuidade à esta pesquisa, aconselha-se pesquisar mais a fundo as citadas vulnerabilidades, e demonstrar diferentes implementações de segurança, como permissões de acessos para vulnerabilidades como *directory traversal* e *file upload attack,* outras formas de validação das entradas do usuário, utilização de outras bibliotecas de conexão que também bloqueiam efetivamente injeções de SQL, como o MYSQLi, pois fora evidenciado

apenas o básico e a essência de cada, servindo como porta de entrada para futuros projetos.

6 REFERÊNCIAS

ACUTENIX, *Website Security,* 2013. Disponível em: <http://www.acutenix.com/websitesecurity>
Acessado em 15 de Abril de 2013.

APACHE, T*he Apache Software Foundation,* 1999. Disponível em: <http://www.apache.org>
Acessado em 10 de Maio de 2013.

BLS, *Software Developers,* 2012. Disponível em: <http://www.bls.gov/ooh/Computer-and-Information-Technology/Software-developers.htm#tab-1>
Acessado em 4 de Maio de 2013.

BODMER, Fabrice. *Cross-Site Scripting (XSS) – Computer and Network Security Seminar*, UNIFR, 2007. Disponível em:
<http://diuf.unifr.ch/drupal/tns/sites/diuf.unifr.ch.drupal.tns/files/Teaching/2006_2007/Computer_Security_Threats_and_Counter_Measures/Bodmer_CrossSiteScripting.pdf>
Acessado em 21 de Maio de 2013.

CALIN, Bogdan. *Why File Upload Forms are a Major Security Threat*, 2009. Disponível em:
<http://www.acunetix.com/websitesecurity/upload-forms-threat/>
Acessado em 22 de Maio de 2013.

CERON, João M., FAGUNDES, Leonardo L, LUDWIG, Glauco A., TAROUCO, Liane e BERTHOLDO, Leandro. - Vulnerabilidades em Aplicações Web: uma Análise Baseada nos Dados Coletados em Honeypots, 2008. Disponível em:
<http://www.pop-rs.rnp.br/arquivos/2008/honeypot_web_sbseg.pdf >
Acessado em 10 de Dezembrode 2013.

CGISECURITY. *The Cross-Site Scripting (XSS) FAQ.* Disponível em: <http://www.cgisecurity.com/xss-faq.html>
Acessado em 21 de Maio de 2013.

COVA, Marco; FELMETSGER, Viktoria; VIGNA, Giovanni. *Vulnerability Analysis of Web-Based Applications,* p 363 – 394, 2007.

DALILI, Soroush. *Unrestricted File Upload*, 2011. Disponível em: <https://www.owasp.org/index.php/Unrestricted_File_Upload>
Acessado em 23 de Maio de 2013.

GOOGLE, *Web Application Exploits and Defenses, 2010.* Disponível em: <http://google-gruyere.appspot.com/part2>
Acessado em 20 de Novembro de 2013.

HALFOND, William G. J.; VIEGAS, Jeremy; ORSO, Alessandro. *A Classification of SQL Injection Attacks and Countermeasures*, Georgia Institute of Technology, 2006. Disponível em:
<http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf>
Acessado em 15 de Maio de 2013.

ISS, *Internet Security Systems – Session Hijacking, 1999.* Disponível em: <http://www.iss.net/security_center/advice/Exploits/TCP/session_hijacking/default.htm>
Acessado em 28 de Maio de 2013.

KÄFER, Konstantin. *Cross Site Request Forgery,* 2008. Disponível em: <http://dump.kkaefer.com/csrf-paper.pdf>
Acessado em 29 de Maio de 2013.

MANAGEMENTSTUDYGUIDE, *Growth and Historical Evolution of ERP*, 2013. Disponível em: < http://www.managementstudyguide.com/growth-and-historical-evaluation-of-erp.htm>.
Acessado em 18 de Abril de 2013.

MEIER, J. D.; MACKMAN, Alex; VASIREDDY, Srinath; DUNNER, Michael; ESCAMILLA, Ray; MURUKAN, Anandha. *Microsoft: Improving Web Application Security: Threats and Countermeasures.* Disponível em: <http://msdn.microsoft.com/en-us/library/ms994921.aspx>
Acessado em 13 de Maio de 2013.

MICROSOFT, Developer Network. *How Cross-site Scripting Attacks Work.* Disponível em: < http://msdn.microsoft.com/en-us/library/ee810614(v=cs.20).aspx>
Acessado em 21 de Maio de 2013.

MIME, Multipurpose Internet Mail Extensions, 1996. Disponível em: <http://tools.ietf.org/html/rfc2045>
Acessado em 20 de Novembro de 2013.

NETCRAFT, *January 2013 Web server survey,* 2013. Disponível em: <http://news.netcraft.com/archives/2013/01/07/january-2013-web-server-survey-2.html>
Acessado em 28 de Abril de 2013.

OWSAP. *The Open Web Application Security Project.*
Cross-Site Request Forgery (CSRF), 2013. Disponível em:
<https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>
Acessado em 29 de Maio de 2013.
Session Hijacking Attack, 2011. Disponível em:
<https://www.owasp.org/index.php/Session_hijacking_attack>
Acessado em 29 de Maio de 2013.
Path Traversal, 2009. Disponível em:
<https://www.owasp.org/index.php/Path_Traversal>
Acessado em 29 de Maio de 2013.

PHP, *popular general-purpose scripting language that is especially suited to web development*, 1999. Disponível em:
<http://www.php.net/>
Acessado em 20 de Novembro de 2013.

PHPCAPTCHA, *A free open-source PHP CAPTCHA script for generating complex CAPTCHA images and audio files,* 2012. Disponível em:
<http://www.phpcaptcha.org/>
Acessado em 20 de Novembro de 2013.

PINGDOM, *Internet 2012 in numbers*, 2013. Disponível em:<http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers>.
Acessado em 17 de Abril de 2013.

SARWATE, Amol. *Hot or not: Web application firewalls for security and regulatory compliance,* 2008. Disponível em:
<http://www.scmagazine.com/hot-or-not-web-application-firewalls-for-security-and-regulatory-compliance/article/113146/>
Acessado em 10 de Maio de 2013.

SIDDHARTH, Sumit; DOSHI, Pratiksha. *Five common Web application vulnerabilities*, 2010. Disponível em: <http://www.symantec.com/connect/articles/five-common-web-application-vulnerabilities>.
Acessado em 5 de Maio de 2013.

SPETT, Kevin. *Cross-site Scripting – Are your web application vulnerable?*. SPI DYNAMICS , 2005. Disponível em :
<http://www.rmccurdy.com/scripts/docs/spidynamics/SPIcross-sitescripting.pdf>
Acessado em 21 de Maio de 2013.

ZELLER, William; FELTEN W. Edward. *Cross-Site Request Forgeries: Exploitation and Prevention.* 2008. Disponível em:
<https://www.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf>
Acessado em 25 de Maio de 2013.

Análise das principais vulnerabilidades presentes em aplicações web e implementações de segurança utilizando a linguagem interpretada livre PHP

William D. S. Castro Souza[1]
[1]Universidade do Sagrado Coração (USC)
Bauru – SP – Brazil

william.castro@grpm.com.br

Abstract. *The Internet is the main medium of communication today, and has facilitated numerous professional and everyday tasks. Among these, business management, product marketing, and advertising in general. The security should never be overlooked. From the growing number of attacks on web sites and systems, and the large number of sites that have some kind of loophole, an analysis of the main web vulnerabilities was performed, explaining what they are, how they work and what are the risks of an eventual attack, then been demonstrated guidelines on how to protect yourself,, programmatically, using the free scripting language PHP, or just server settings.*

Resumo. *A Internet é o principal meio de comunicação da atualidade, e vem facilitando inúmeras tarefas cotidianas e profissionais. Dentre essas, a gestão de negócios, comercialização de produtos, e publicidades em geral. A questão de segurança nunca deve ser deixada de lado. A partir do crescente número de ataques à sites e sistemas web, e da grande quantidade de sites que possuam algum tipo de brecha, foi realizada uma análise das principais vulnerabilidades web, explicando o que são, como agem e quais os riscos de um eventual ataque, em seguida foram levantadas orientações de como proteger-se, através de programação, utilizando-se da linguagem interpretada livre PHP, ou apenas configurações de servidor.*

1. INTRODUÇÃO

Sabe-se atualmente sobre as enormes transformações que a Internet vem causando na comunicação, no trabalho, no comércio e no entretenimento ao redor do mundo. Ela vem crescendo exponencialmente tanto em quantidade de sites, contabilizando 582.716.657 em janeiro de 2012 e 629.939.191 em janeiro de 2013 (NETCRAFT, 2013), como em quantidade de usuários, passando de 2,1 bilhões em 2011 para 2,4 bilhões em 2012 (PINGDOM, 2012), e a tendência é aumentar cada vez mais (PINGDOM, 2012) devido à sua popularização, facilidade e comodidade.

Com o crescimento da Internet e suas facilidades, muitas aplicações que antes eram apenas desktop passaram a ser desenvolvidas e utilizadas online, crescendo muito sua popularização (COVA, 2007). Uma série de fatores como comodidade, rapidez, facilidade e portabilidade também contribuíram para um crescente número de organizações e indivíduos confiarem em aplicações baseadas na Web, oferecendo assim, uma grande variedade de serviços. Atualmente, aplicações Web são rotineiramente utilizadas em ambientes vitais, como sistemas médicos, financeiro e militar (COVA,2007).

Sistemas e aplicações Web são compostos por itens de infraestrutura, tais como: servidor Web, bancos de dados e código fonte específico – tanto *client side,* como o HTML e JavaScript, como *server side*, englobando PHP, ASP, JSP, e diversas outras linguagens. Enquanto os componentes de infraestrutura são normalmente desenvolvidos por técnicos experientes, com habilidades sólidas de segurança, o código fonte das aplicações, comumente é desenvolvido por programadores com pouco treinamento em segurança e sob estritas restrições de tempo (COVA, 2007). Como resultado, muitas aplicações vulneráveis são implantadas e disponibilizadas por toda a Internet, criando pontos de entrada facilmente exploráveis para o comprometimento de redes inteiras.

Segundo Netcraft (2013), dos 629.939.191 sites contabilizados em Janeiro de 2013, cerca de 244 milhões utilizam a linguagem interpretada livre PHP, notou-se também, que o principal servidor Web Apache ainda mantém uma maioria, contando com 55,26% do mercado, portanto, segundo SIDDHARTH, grande parte destas aplicações não possui proteção contra várias vulnerabilidades existentes. Estas aplicações, por trafegarem informações e dados valiosos para a empresa ou o usuário, não podem continuar vulneráveis aos possíveis ataques.

Outro ponto importante é o crescimento do número de desenvolvedores de softwares no mundo (BLS, 2012). Isto significa que atualmente, muitas pessoas sabem ou têm noções de programação e codificação de aplicações. Com o aumento do número de pessoas que possuem conhecimento sobre o desenvolvimento de um site e quais são suas possíveis defesas, conseqüentemente aumentou-se o número de ataques a sites vulneráveis.

Através destas informações, foram levantadas as principais vulnerabilidades presentes em *sites* institucionais e aplicações Web, analisando-as através de referencial teórico, identificando assim, do que se tratam e como funciona um ataque de forma maliciosa, para então ser demonstradas possíveis formas de proteção, correção e implementação de segurança contra as mesmas, podendo ser através de simples configurações do próprio servidor, ou de códigos de programação mais complexos utilizando-se da linguagem interpretada livre PHP. Foram desenvolvidas aplicações práticas para demonstrar como funciona um ataque através de uma vulnerabilidade web, e quais as implementações de segurança necessárias para proteger-se, através de simulações de uma aplicação vulnerável sofrendo o ataque, e outra aplicação protegida, repelindo o ataque.

2. VULNERABILIDADES

Vulnerabilidades são brechas, *bugs*, pelo qual uma aplicação computacional pode tornar-se vítima de um ataque ou invasão. A seguira análise das 5 principais vulnerabilidades presentes em aplicações Web (*Sql Injection, Cross Site-Scripting, File Upload Attack, Session Hijacking* e *Cross Sifte Request Forgery*), junto com implementações de seguranças validadas através de aplicações práticas, e seus resultados.

2.1 SQL INJECTION

SQL Injection é a técnica de ataque à aplicação mais comum atualmente, trata-se de tirar proveito de codificação indevida de aplicações, permitindo que atacantes possam injetar comandos SQL diretamente no banco de dados através de entradas de usuário, como formulários de *login* (ACUTENIX, 2013).

Um ataque de SQL Injection, pode ocorrer através de um formulário de *login*, em que a aplicação comunica-se com o banco de dados por meio de uma série de comandos planejados, de forma à verificar a combinação do nome de usuário e senha, porém esta verificação não utiliza as entradas do formulário de forma higienizada, comparando e enviando diretamente a consulta SQL para o banco de dados.

Nota-se que a variável *$query* possui a sentença SQL formada recebendo diretamente a entrada do usuário, ou seja, os parâmetros 'usuario' e 'senha'. Uma injeção de SQL é possível nesta forma de codificação. Simulando uma injeção, ao enviar como parâmetro 'usuário' a sentença **' or 1=1;-** , consegue-se garantir acesso à aplicação, pois ao receber o parâmetro **' or 1=1--** para o campo usuário**,** a aspa no inicio da sentença fechou a *string* relativa ao nome de usuário e inseriu-se uma condição (ou 1=1) que sempre será verdadeira, porém só está condição retornaria falsa pois a validação de senha ainda estava ativa, então ao adicionar os dois traços no final da sentença todo o resto da *query* e suas condições foram comentadas e ignoradas pelo banco, resultando na sentença reproduzida pela Figura 1 que sempre retornará verdadeira.

```
SELECT * FROM usuario WHERE usuario = '' OR 1=1
```

Figura 1: Resultado de consulta em banco com injeção SQL.

2.1.1 PHP DATA OBJECT (PDO)

Uma maneira de evitar injeções Sql é utilizando a biblioteca de manipulação de dados orientada à objeto do PHP, está trata e envia para o banco de dados uma *query* MySQL válida, através de *prepared statments.*

Segundo PHP (2013), "Se um aplicativo usa exclusivamente instruções preparadas, o desenvolvedor pode ter certeza de que não irá ocorrer a injeção de SQL", declaração esta confirmada ao realizar a mesma injeção SQL na aplicação utilizando a biblioteca PDO e não obter um acesso indevido.

2.2 CROSS SITE-SCRIPTING (XSS)

Cross Site Scripting mais conhecido como XSS, é a manipulação de *scripts* do lado do cliente (navegador), de uma aplicação Web para executar ações maliciosas (ACUTENIX, 2013). O XSS ocorre quando páginas da Web que são geradas dinamicamente não são devidamente validadas. Isto permite que um invasor inclua códigos JavaScript maliciosos na página gerada, e execute o *script* na máquina do usuário.

Como por exemplo, uma página PHP que recebia via protocolo *HTTP-GET* uma variável denominada 'pesquisa' em seguida exibe-se uma mensagem "Resultado de pesquisar para 'termo':", junto com o resultado da consulta. Esta consulta gera uma URL com o parâmetro 'termo', contendo o valor inserido no campo de pesquisa, por exemplo, *http://localhost/tcc/xss.php?**termo=busca+teste.***

Caso este 'termo' não seja devidamente tratado, ou seja, exibido diretamente (Figura 2), esta aplicação está vulnerável à injeções XSS. Um atacante então poderá inserir um código JavaScript no lugar do termo de busca, código que poderá enganar usuários que acessarem está página com a URL alterada, podendo assim ter acesso à informações como *cookies* e executar outros ataques como *session hijacking*.

```
<?php if (isset($_GET['termo'])){
      $termo = $_GET['termo'];

      /**
      /* Pesquisa em banco com o termo
      /* ....
      /**/

      echo "Resultado de pesquisa para '".$termo."': <br/>";

      /** Imprime os resultados de pesquisa **/
      /*  .... */
      ?>
 <?php }?>
```

Figura 2: Código PHP vulnerável à XSS, parâmetros exibidos sem tratamento.

2.2.1 HTMLENTITIES E STRIP_TAGS

A função PHP *htmlentities* é responsável por converter todos os caracteres de uma *string* em uma sentença HTML válida. Já a função *strip_tags* remove toda e qualquer *tag* HTML, principalmente a *tag* <SCRIPT>, responsável pela injeção de XSS. Segundo PHP (2013), o uso de *htmlentities* e *strip_tags* bloqueiam com sucesso injeções XSS. Bloqueio este confirmado através de aplicações práticas utilizando as funções em questão, ao ler o código fonte da página, a injeção de JavaScript não foi exibida , evitando assim o vazamento de informações como *cookies*.

2.3. FILE UPLOAD ATTACK

É o envio de arquivos para o servidor sem validação ou tratamento. Podendo ser arquivos contendo códigos que serão executados pelo servidor Web. Dalili (2011) explica que estes arquivos enviados representam um risco significativo para aplicações. O primeiro passo em muitos ataques é enviar um código para o sistema à ser atacado. Em seguida, o atacante só precisa encontrar uma maneira de executar o código no servidor.

Para demonstrar esta vulnerabilidade, fora utilizado o *script* PHP conhecido como *shellc99.php*, que funciona como um serviço de FTP *online*, construído especificamente para invasão de sites e aplicações. Ao receber o arquivo via *upload*, e armazená-lo, o atacante pode acessá-lo e executá-lo via navegador, garantindo assim controle sobre os arquivos fontes da mesma.

2.3.1. VERIFICAÇÃO MIME E .htaccess

Para evitar envio de arquivos maliciosos, pode-se utilizar a verificação do tipo MIME (Multipurpose Internet Mail Extensions), que é uma norma da internet para o formato das mensagens de correio eletrônico, contendo a categoria e o tipo do arquivo (MIME, 1996), como mostra a Figura 3.

```
//verifica MIME Type
if($curriculo['type'] != "application/msword") {
      echo "Desculpe, apenas documentos do tipo .doc e .docx
      são aceitos.<br />";
      exit;
}
```

Figura 3. Verificação do tipo MIME em PHP.

Outra implementação de segurança cabível, é o bloqueio da listagem de arquivos pelo navegador, utilizando o arquivo de configuração APACHE .htaccess, arquivo este que permite gerenciar a extensão de arquivo que o navegador exibirá, bloqueando assim o acesso à arquivos de código fonte e maliciosos.

2.4. SESSION HIJACKING

Segundo Owsap (2011), *Session Hijacking* (Sequestro de Sessão) consiste na exploração do mecanismo de controle da sessão web, que normalmente é gerada por um *token*, a captura deste *token* válido, chamada "PHPSESSID", pode ser usado para ganhar acesso não autorizado ao servidor Web através de modificadores de *cookie*.

Este *token* pode ser obtido de formas variadas, através de ataques XSS, *fishing*, entre outras que não são o foco da vulnerabilidade.

2.4.1. CONFIGURAÇÕES

A forma comprovada de evitar seqüestro de sessão, e uso indiscriminado do mesmo é alterando as configurações do servidor contidas no arquivo de configuração do PHP *php.ini*. Estas configurações são *session.use_only_cookies*, responsável por especificar se o módulo usará apenas *cookies* para guardar a ID de sessão no lado do cliente. Habilitando esta configuração previne ataques envolvendo passagem de IDS de sessão em URLs. E a configuração *session.cookie_secure*, esta responsável por especificar que os *cookies* sejam apenas enviados sob conexões seguras.

2.5 DIRECTORY TRAVERSAL

Segundo Acutenix (2013), Directory Traversal é um *exploit* HTTP que permite aos *crackers* acessar diretórios restritos, e executar comandos fora do diretório raiz da aplicação Web.

Esta vulnerabilidade é comum em aplicações que referenciam o nome do arquivo na URL e não fazem as devidas validações, permitindo assim que o atacante insira expressões como ‘../’ instruindo assim o sistema subir um diretório a partir da raiz. Desta forma é possível listar na tela informações de configuração do servidor (Figura 4), como do sistema operacional.

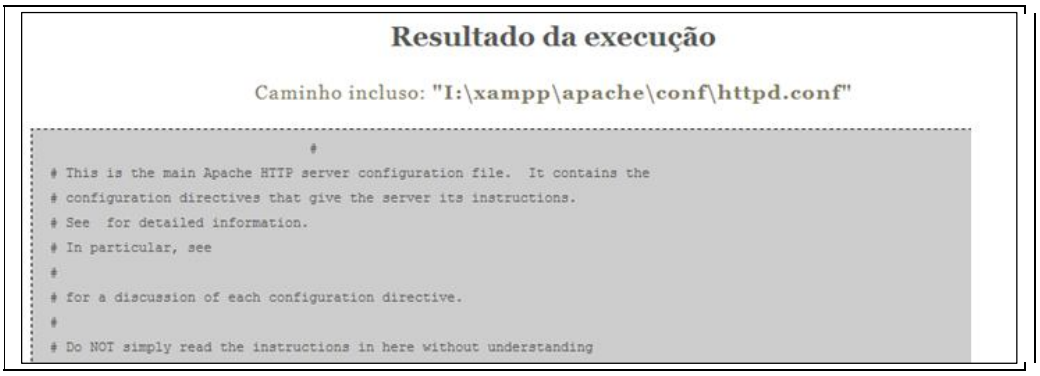

Figura 4: Listagem de configuração apache ao acessar URL http://tcc/traversal.php?arquivo=../../apache/conf/httpd.conf.

2.5.1. BASENAME

O método implementado de prevenção à este tipo de vulnerabilidade, foi a utilização da função PHP *basename*, função esta, responsável por retornar apenas o nome do arquivo, e não seu diretório absoluto, ignorando assim todas as tentativas de alteração de diretório (../../), a Figura 5 mostra um exemplo de inclusão segura de arquivo, via parâmetros na URL.

```
<?php
      $arquivo = $_GET['arquivo'];

      if(!@include(basename($arquivo)){
            echo "404 Página não encontrada.";
      }
?>
```

Figura 5: Inclusão segura de arquivos via URL utilizando função *basename.*

2.5. CROSS SITE REQUEST FORGERY

Segundo Käfer (2008), Cross Site Request Forgery (CSRF) é uma técnica que permite que o atacante engane o usuário para realizar uma ação, usando sua autoridade e credencial. Ou seja, segundo Owsap (2013), CSRF é um ataque que engana a vítima para o carregamento de uma página que contenha uma solicitação maliciosa.

Em outras palavras, um usuário *logado* em uma aplicação legítima, acessa a página do atacante, página essa que contém, um formulário escondido, com dados como endereço de *email*, referentes ao atacante e envio direcionado à aplicação legítima em que o usuário encontra-se *logado*. O usuário ao entrar nesta página é redirecionado para sua página de alteração de cadastro, salvando o *email* do atacante como seu novo *email*, sem que tenha conhecimento. Ou seja, o usuário faz uma solicitação legítima, sem seu conhecimento. Isto caracteriza CSRF, como demonstrado nas aplicações práticas, o usuário teve seus dados alterados ao entrar em uma página de terceiro (Figura 6) que continha um formulário escondido submetendo informações maliciosas em seu nome.

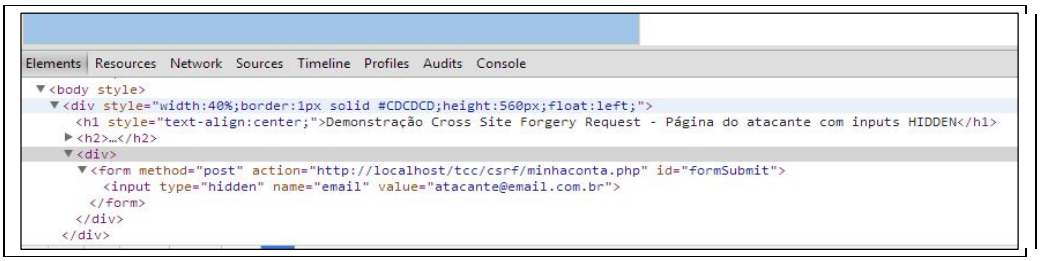

Figura 6: Formulário escondido em página do atacante, sendo submetido à aplicação legítima, alterando informações do usuário através de CSRF.

2.5.1. CAPTCHA

A forma demonstrada para evitar ataques de CSRF, foi a implementação de um sistema CAPTCHA, que trata-se uma implementação de segurança desenvolvida em PHP, com o intuito de forçar o usuário à identificar letras de uma imagem distorcida, provando assim ser uma pessoa real.

Ao utilizar CAPTCHA, ataques de CSRF não são sucedidos, pois o código CAPTCHA é gerado apenas para a página legítima, sendo assim, o atacante não tem como prever o código que será gerado em uma solicitação válida, ou seja, dentro da aplicação.

3. CONSIDERAÇÕES

Este trabalho demonstrou através de referencial teórico e experimentos em aplicações práticas a existência e funcionalidades das vulnerabilidades mais presentes em aplicações *web* atuais.

Concluiu-se que as linguagens de programação *web*, atualmente possuem funções nativas para combater estas vulnerabilidades, como a linguagem PHP contém a nova biblioteca *PDO* (PHP Data Object), que ao ser utilizada corretamente, exclui qualquer possibilidade de uma injeção SQL; a função *strip_tags* e *htmlentities*, funcionam muito bem para eliminar qualquer caracter ou *tag* não permitido nas entradas do usuário, prevenindo assim ataques como XSS. Cabe apenas a conscientização do desenvolvedor para a existência e riscos destas vulnerabilidades, pois o próprio ambiente de trabalho oferece implementações de segurança confiáveis.

Em geral, tratar a entrada do usuário é essencial para o desenvolvimento de uma aplicação segura, foi provado que nunca deve-se utilizar os parâmetros recebidos, diretamente na aplicação, pois há diversas vulnerabilidades existentes que utilizam deste método para formalizar um ataque.

Para aqueles que visam dar continuidade à esta pesquisa, aconselha-se explicar mais a fundo uma das vulnerabilidades citadas, pois aqui foi evidenciado apenas o básico e a essência de cada.

4. REFERÊNCIAS

ACUTENIX, *Website Security,* 2013. Disponível em: <http://www.acutenix.com/websitesecurity>. Acessado em 15 de Abril de 2013.

APACHE, T*he Apache Software Foundation,* 1999. Disponível em: <http://www.apache.org>. Acessado em 10 de Maio de 2013.

COVA, Marco; FELMETSGER, Viktoria; VIGNA, Giovanni.
Vulnerability Analysis of Web-Based Applications, p 363 – 394, 2007.

DALILI, Soroush. *Unrestricted File Upload*, 2011. Disponível em: <https://www.owasp.org/index.php/Unrestricted_File_Upload>. Acessado em 23 de Maio de 2013.

KÄFER, Konstantin. *Cross Site Request Forgery,* 2008. Disponível em: <http://dump.kkaefer.com/csrf-paper.pdf>. Acessado em 29 de Maio de 2013.

MIME, Multipurpose Internet Mail Extensions, 1996. Disponível em: <http://tools.ietf.org/html/rfc2045>. Acessado em 20 de Novembro de 2013.

NETCRAFT, *January 2013 Web server survey*, 2013. Disponível em: <http://news.netcraft.com/archives/2013/01/07/january-2013-web-server-survey-2.html>. Acessado em 28 de Abril de 2013.

OWSAP. *The Open Web Application Security Project.*
Cross-Site Request Forgery (CSRF), 2013. Disponível em: <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>. Acessado em 29 de Maio de 2013.
Session Hijacking Attack, 2011. Disponível em: <https://www.owasp.org/index.php/Session_hijacking_attack>. Acessado em 29 de Maio de 2013.

PHP, *popular general-purpose scripting language that is especially suited to web development*, 1999. Disponível em: <http://www.php.net/>. Acessado em 20 de Novembro de 2013.

PINGDOM, *Internet 2012 in numbers*, 2013. Disponível em: <http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers>. Acessado em 17 de Abril de 2013.

SIDDHARTH, Sumit; DOSHI, Pratiksha. *Five common Web application vulnerabilities*, 2010. Disponível em: <http://www.symantec.com/connect/articles/five-common-web-application-vulnerabilities>. Acessado em 5 de Maio de 2013.