AUDITORIA DE SEGURANÇA DA INFORMAÇÃO EM APLICAÇÕES WEB: ESTUDO DE CASO SOBRE O ATAQUE *CROSS-SITE SCRIPTING* (XSS) [1]

João Marcos Barbosa Oliveira

Resumo: O presente artigo versa sobre o tema de Auditoria de Segurança da Informação em Aplicações Web, com ênfase em ataques de *Cross-Site* Scripting, também conhecido como XSS. Seu objetivo foi descrever como os ataques ocorrem, quais as suas consequências para empresa e como podem ser evitados. Os testes efetuados à aplicação Web *WebGoat* foram realizados em um ambiente simulado destinado unicamente a este fim, de modo a verificar como ocorre esse tipo de ataque. A metodologia utilizada para a realização dos testes constou em submeter os vetores de entrada de dados às técnicas descritas no OWASP Testing Guide-OTG-INPVAL 001 e 002 que trata especificamente sobre XSS. Os resultados encontrados possibilitaram uma melhor compreensão de como ocorre este ataque, suas consequências e técnicas de prevenção.

Palavras-chave: *Cross-Site Scripting*. Auditoria. Aplicações Web.

1 INTRODUÇÃO

A auditoria de sistemas de informação visa verificar a conformidade do ambiente informatizado, garantindo a integridade dos dados manipulados pelo computador. Assim, ela estabelece e mantém procedimentos documentados para planejamento e utilização dos recursos computacionais da empresa, verificando aspectos de segurança e qualidade (FONSECA, 2012). Por isso torna-se imprescindível que a equipe responsável pela auditoria tenha conhecimento das técnicas e procedimentos utilizados por atacantes para que possa auxiliar na implementação de Sistema de Gestão de Segurança da Informação seguro e confiável. Segundo o relatório anual de vulnerabilidades de aplicações Web da Acutinex de 2016, o ataque via *Cross-Site Scripting* (XSS) correspondia a mais de 30% das vulnerabilidades de alto risco, a frente

1Artigo apresentado como Trabalho de Conclusão do Curso de Especialização em Gestão de Segurança da Informação, da Universidade do Sul de Santa Catarina, como requisito parcial para a obtenção do título de Especialista em Gestão de Segurança da Informação.

de SQL Injection, senhas fracas, etc.

Enquanto muitas falhas na segurança de aplicações Web advém de vulnerabilidades no software utilizado pelo servidor, as vulnerabilidades de *Cross-Site Scripting* são um descuido do programador no desenvolvimento da aplicação, causando uma falha nas validações dos parâmetros de entrada do usuário e na resposta do servidor na aplicação Web. O objetivo geral do estudo consiste em verificar como os ataques de *Cross-Site Scripting* ocorre, quais as suas consequências para uma empresa e como podem ser evitados.

O desenvolvedor da aplicação tem fundamental importância na segurança do sistema, o que implica em um conhecimento sobre métodos de defesa e ataque aplicáveis às soluções que deseja implementar. A análise proposta permitirá obter conhecimentos relativos a ataque e defesa. O aprofundamento desses conhecimentos dá ao auditor possíveis soluções para vulnerabilidades verificadas em auditorias em aplicações Web.

Com a intenção de servir para auxiliar o aprendizado da segurança de aplicações Web e como laboratório de testes foi criado o aplicativo *WebGoat* pelo Projeto Aberto de Segurança de Aplicações Web, em inglês *Open Web Application Security Project* (OWASP). Em cada módulo, ou lição, dela o aluno tem de demonstrar seu conhecimento através da exploração de uma vulnerabilidade específica (OWASP, 2014). O aplicativo OWASP *WebGoat* será utilizado para fins de demonstração dos ataques *Cross-Site Scripting*.

Concordamos com Assunção (2008) na afirmação de que, uma das maneiras pela qual o *Cross-Site Scripting* ocorre, é quando um atacante é capaz de inserir *scripts* no banco de dados como entrada de dados e esses *scripts*, quando não são filtrados corretamente, expõem os usuários legítimos do sistema aos códigos do atacante. A pesquisa que desenvolveremos está vinculada à premissa de que muitos programadores não validam as entradas de dados do usuário, possibilitando esse tipo de ataque.

Por uma questão de fidelidade a nossa proposta, manteremos o foco da análise na do ataque *Cross-Site Scripting* utilizando-se, para fins de demonstração, da aplicação *WebGoat*, em um ambiente acadêmico controlado e fora de qualquer rede em produção.

2 APLICAÇÕE WEB

Aplicações Web podem ser definidas como aplicações que utilizam Internet ou Intranet como camada de aplicação, dessa forma a interação com o usuário se torna mais amigável e as informações são apresentadas de forma mais simples e compreensível, sendo que o usuário não necessita de um aplicativo instalado no computador, somente um navegador Web (GUMERATO, 2009). Deste modo as atualizações ou modificações feitas na aplicação são automaticamente percebidas pelos usuários sem que nenhuma alteração seja feita nos computadores clientes.

Quando utilizamos um aplicativo Web precisamos de um repositório (banco de dados) para guardar informações com que vamos trabalhar. As linguagens de programação utilizadas nos aplicativos Web (Javascript, PHP, ASP, Java, ASP.NET) não podem, sozinhas, lidar com o armazenamento e controle dos dados. Qualquer sistema desenvolvido através delas deve possuir algum meio de acessar e controlar um sistema gerenciador de banco de dados, como Oracle, MySQL ou PostgreSQL (ULBRICH, 2009).

Em um aplicativo bem implementado somente os dados validados serão enviados e processados no banco de dados, porém, em muitos sistemas, isso não ocorre e qualquer dado, válido ou não, enviado pelo usuário será armazenado no servidor.

Através da falta de validação dos dados passados pelo usuário ele é apresentado integralmente pelo navegador, como no caso de um código *javascript* que passa a ser interpretado como parte da aplicação legítima (ECCOUNCIL, 2012). Um exemplo é uma aplicação do tipo fórum, em que o usuário tenha permissão para incluir mensagens para que os outros usuários possam ler. Se este aplicativo não filtrar corretamente os códigos HTML e e/ou Javascript, um usuário mal intencionado pode inserir instruções no navegador do usuário legítimo e até mesmo executar tarefas específicas como enviar mensagens de maneira arbitrária para o fórum.

3 REQUISIÇÕES

Aplicações Web são um exemplo de arquitetura cliente e servidor. O navegador (agindo como cliente) envia uma requisição ao servidor e espera por uma resposta. O servidor recebe a requisição, gera uma resposta, e a envia de volta para o cliente. Obviamente deve existir uma espécie de entendimento entre as duas partes, caso contrário o cliente enviaria uma requisição qualquer e o servidor não saberia como respondê-la. A maneira pela qual o cliente e o servidor se comunicam é através de um protocolo, neste caso, HTTP. Para TANENBAUM (2004, p. 651):

> Embora o HTTP tenha sido desenvolvido nos primórdios da Internet ele já possuía mecanismos que possibilitariam o seu uso em aplicações futuras. Por essas razões, as operações, chamados de métodos, são mais do que apenas requisitar uma página Web. Cada requisição consiste em uma ou mais linhas de caracteres cuja primeira palavra dita o nome do método.

Os métodos de requisição HTTP mais utilizados em ataques de injeção SQL são o GET e o POST que passaremos a descrever para melhor esclarecimento.

O método GET requisita ao servidor Web para que mande uma página ou um arquivo (TANENBAUM, 2004). Ele é aquele em que a requisição se encontra no próprio URL. URL significa, em português, Localizador Uniforme de Recurso e se refere ao endereço de rede, neste caso, do site. Esse é o método normalmente utilizado quando se acessa um link na internet. O navegador envia uma requisição utilizando o método GET ao servidor e, após receber a resposta, monta a página na tela.

Ele é utilizado quando a requisição não tem por objetivo alterar dados do servidor sendo utilizado, por exemplo, na realização de uma consulta. A requisição 1 mostra um exemplo completo de requisição através do método GET ao servidor do *WebGoat*.

Requisição 1 – Requisição enviada através do método GET

```
GET http://localhost:8080/WebGoat/login.mvc
Host[localhost:8080]
User-Agent[Mozilla/5.0 (Windows NT 10.0; WOW64; rv:54.0)
Accept[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8]
Accept-Language[pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3]
Cookie[JSESSIONID=AEED9E201A56D6D3D1123C710401B6CF]
Connection[keep-alive]
Upgrade-Insecure-Requests[1]
Cache-Control[max-age=0]
```

Fonte: o autor, 2017.

Esse tipo de requisição envia variáveis inseridas na URL no seguinte formato:

http://www.exemplo.com/Index.php?variavel1=valor1&variavel2=valor2

O servidor receberá essas variáveis e as processará. No método GET, pode-se manipular as requisições apenas mudando os parâmetros diretamente na barra de endereços do navegador.

O método POST é o normalmente utilizado quando se preenche um formulário e tem de se clicar em um botão para enviá-lo ao servidor. Esse método deixa o URL separada dos dados que serão enviados, ao contrário do GET, o que permite que sejam transmitidos qualquer tipo de dados por este método. A Requisição 2 mostra um exemplo fictício de requisição completa através do método POST à página de login da aplicação *WebGoat* .

Requisição 2 – Requisição enviada através do método POST

POST http://localhost:8080/*WebGoat*/j_spring_security_check
Host[localhost:8080]
User-Agent[Mozilla/5.0 (Windows NT 10.0; WOW64; rv:54.0) Firefox/54.0
Accept[text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8]
Accept-Language[pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3]
Accept-Encoding[gzip, deflate]
Content-Type[application/x-www-form-urlencoded]
Content-Length[29]
Referer[http://localhost:8080/*WebGoat*/login.mvc]
Cookie[JSESSIONID=AEED9E201A56D6D3D1123C710401B6CF]
Connection[keep-alive]
Upgrade-Insecure-Requests[1]
Post Data:
username[guest]
password[guest]

Fonte: o autor, 2017.

O formato das variáveis e dos valores enviados é bem parecido com o do método GET, porém estão localizados no final da requisição e não são visíveis através do URL. Enquanto que, no método GET é possível ao usuário modificar a requisição alterando a URL, no método POST, essa alteração é mais complexa de ser realizada. Existem basicamente duas maneiras de alterar uma requisição através do método POST: através da extensão do navegador e do servidor Proxy;

Extensões do navegador são recursos adicionais desenvolvidos normalmente por terceiros que funcionam no navegador e permitem que ele realize funções extras, para as quais não estava originalmente apto a realizar. Como os *plugins*, também chamados de extensões,

funcionam sobre o navegador, eles tem acesso à página requisitada e podem modificar seu conteúdo e aparência (TANENBAUM, 2014).

Como exemplo de extensões podemos citar o *Adobe Flash Player*, que permite a visualização de vídeos, animações, entre outras. No navegador *Mozilla Firefox*, existe a extensão *Tamper Data*, que concede ao usuário o poder de ver e modificar as requisições de seu navegador. A extensão age, dessa maneira, entre a aplicação (ou página) Web no navegador do usuário e o servidor com o qual o ele deseja se comunicar. Para a verificação das requisições nesse Artigo usaremos a extensão *Tamper Data* do *Mozilla Firefox*.

4 CROSS-SITE SCRIPTING

Cross-Site Scripting é a vulnerabilidade de alto risco mais comum na Internet (ACUTINEX, 2016), e reside em descuidos na programação da aplicação Web e não em uma vulnerabilidade do servidor Web ou do banco de dados. Devido à pressão dos gestores e de tempo, a maioria dos programadores acaba não atentando à verificação dos dados passados pelo usuário, possibilitando que invasores explorem essa vulnerabilidade e injetem códigos Javascript e/ou HTML na aplicação Web do servidor (ASSUNÇÃO, 2008).

Ataques do tipo *Cross-Site Scripting* permitem que um atacante insere códigos em Websites vulneráveis, fazendo com que um usuário da aplicação Web execute *scripts* maliciosos em seu navegador. As falhas que permitem que esses ataques tenham êxito são bastante difundidas e ocorrem em qualquer lugar em que uma aplicação Web usa a entrada de um usuário na saída gerada sem validá-la ou codificá-la.

O navegador do usuário final não tem como saber que o *script* não deve ser confiável e executará o script. Como pensa que o *script* veio de uma fonte confiável, o *script* malicioso pode acessar todos os cookies, tokens de sessão ou outras informações confidenciais mantidas pelo navegador e usadas com esse site. Esses *scripts* podem até reescrever o conteúdo da página HTML.

O conteúdo malicioso enviado ao navegador geralmente assume a forma de um segmento de JavaScript, mas também pode incluir HTML, Flash ou qualquer outro tipo de código que o navegador possa executar. A variedade de ataques com base em XSS é quase ilimitada, mas geralmente incluem a transmissão de dados privados, como *cookies* ou outras informações de sessão, ao atacante, redirecionando a vítima para conteúdo da Web controlado

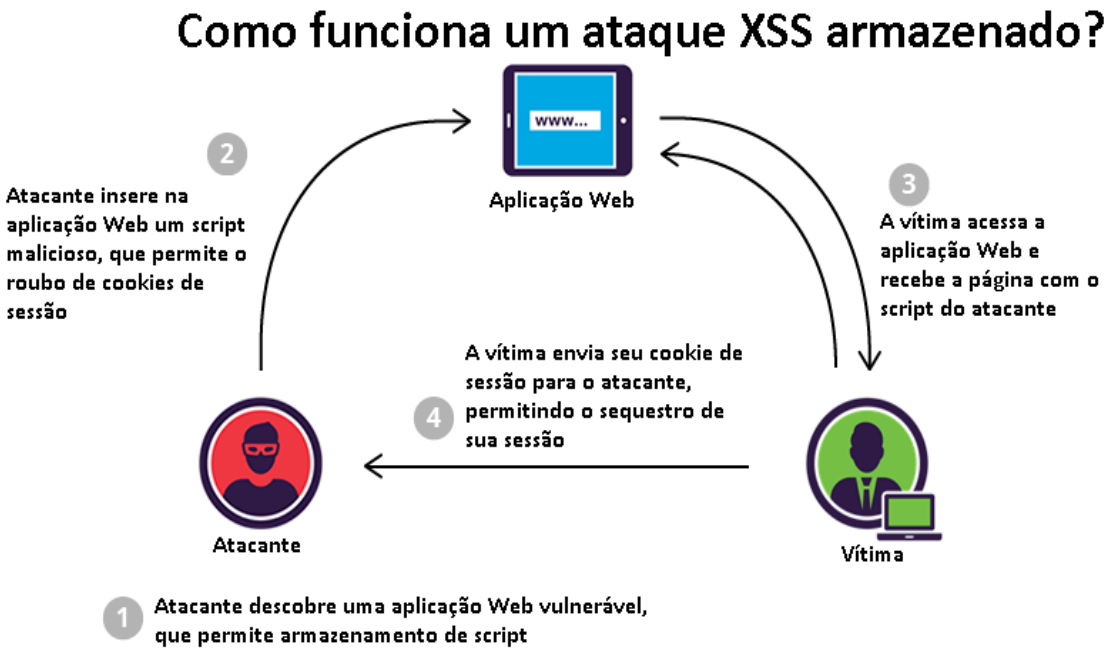
pelo atacante ou realizando outras operações maliciosas na máquina do usuário Sob o disfarce do site vulnerável. O motivo mais comum para um ataque XSS é para *session hijacking* (em português sequestro de sessão, algumas vezes também conhecido como sequestro de cookie) que é a exploração de uma sessão de computador válida, às vezes também chamada de uma chave de sessão - para obter acesso não autorizado a informações ou serviços em um sistema de computador (GOODRICH, 2013).

Os ataques XSS geralmente podem ser categorizados em duas categorias: armazenados e refletidos:

4.1 – *Cross-Site Scripting* do tipo "armazenado"

Os ataques armazenados são aqueles em que o *script* injetado é armazenado permanentemente nos servidores de destino, como em um banco de dados, em um fórum de mensagens, *log* de visitantes, campo de comentários, etc. A vítima recupera o *script* malicioso do servidor quando solicita o armazenamento em formação.

Figura 1 – Ataque XSS do tipo "armazenado"

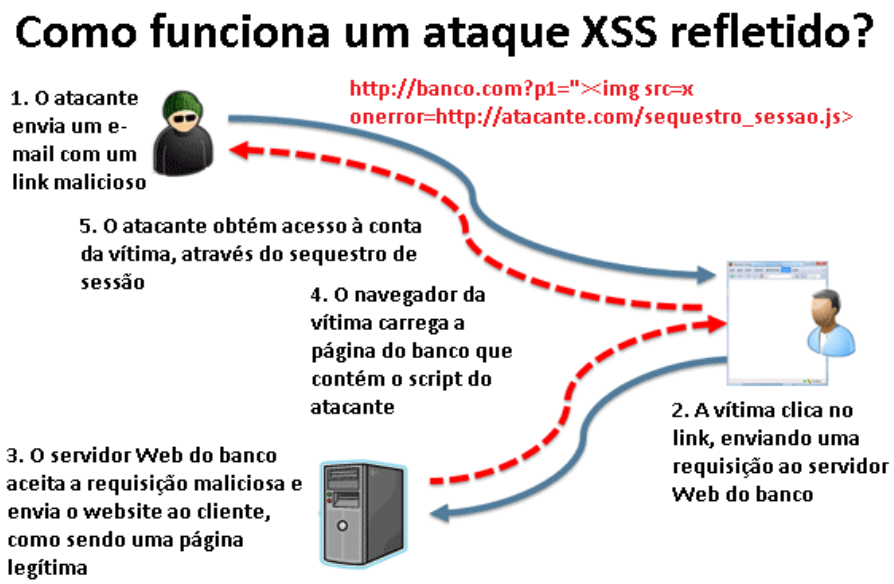


Fonte: o autor, 2017.

4.2 – *Cross-Site Scripting* do tipo “refletido”

Os ataques refletidos são aqueles em que o *script* injetado é refletido no servidor da Web, como em uma mensagem de erro, resultado de pesquisa ou qualquer outra resposta que inclua parte ou a totalidade da entrada enviada ao servidor como parte da solicitação. Os ataques refletidos são entregues às vítimas através de outra rota, como em uma mensagem de e-mail ou em algum outro site. Quando um usuário é enganado ao clicar em um link malicioso, enviando um formulário especialmente criado, ou mesmo apenas navegando para um site malicioso, o código injetado viaja para o site vulnerável, o que reflete o ataque de volta ao navegador do usuário. O navegador então executa o código porque ele veio de um servidor "confiável".

Figura 2 – Ataque XSS do tipo “refletido”



Fonte: o autor, 2017

5 METODOLOGIA

Nesta seção do texto, faremos a definição dos parâmetros do que será um estudo documental e experimental sobre o tema. Os procedimentos metodológicos foram os seguintes: leituras preliminares para aprofundamento do tema e análise da aplicação *WebGoat* utilizada para simulação de um ataque *Cross-Site Scripting*. Ao estabelecermos as bases práticas para a pesquisa, asseguraramos a sua execução respeitando o cronograma proposto, além de permitir a verificação das etapas do estudo.

Por se tratar de um campo de investigação com relativa produção de conhecimento científico, mas não a necessária divulgação, realizamos uma pesquisa do tipo aplicada e empírica.

No decorrer de pesquisa realizamos os seguintes procedimentos:

- Apresentação da pesquisa bibliográfica relacionada à temática, procedemos à definição de ataque *Cross-Site Scripting* e suas variações.

- O procedimento seguinte foi o ataque, em ambiente simulado, ao OWASP *WebGoat* de modo a verificar a eficiência das soluções de segurança implementadas nesse sistema.

- Em seguida foi analisado o impacto desse tipo de ataque e realizada a descrição das técnicas de prevenção, com o objetivo de sanar as vulnerabilidades das aplicações.

A observação sistemática foi composta de um registro de dados que acontecerá no momento em que os testes na aplicação *WebGoat* ocorrerem.

A operação de sistemas não compreende somente a execução isolada de um sistema, mas a integração com os demais sistemas da empresa. Muitas vezes esse controle acaba não sendo de fácil verificação, devido ao volume de transações ou pela possibilidade de os sistemas terem sidos desenvolvidos por fornecedores diferentes. Auditar a operação, e não o código-fonte, pode ser uma forma mais fácil e eficaz, porque tem a possibilidade de verificar diretamente os resultados iniciais, o processamento e as saídas (BOTELHO, 2012).

A entrada de dados tem uma importância fundamental para uma aplicação Web, pois caso eles sejam inseridos de forma inconsistente, comprometem a relação do processamento e da saída. Normalmente pode ocorrer de duas formas, ou sendo inseridas pela usuário, através de algum vetor de entrada de dados, ou pela saída de algum sistema. O processamento de dados está entre a entrada e a saída de dados e pode ser uma porta de manipulação indevida de dados de fácil implantação, pois dependendo da maneira como o desenvolvedor implementar o sistema uma rotina pode simplesmente ser iniciada ou escondida por um simples comando de operação de um *hacker*. A saída de dados é o resumo da entrada e o processamento. Muitos erros que ocorrem na entrada ou no processamento somente são descobertos na saída de dados.

5.1 – Ambiente de Testes

Os testes para simulação de um ataque *Cross-Site Scripting* foram executados em um ambiente acadêmico controlado, em uma rede isolada destinada unicamente a este fim e apartada de qualquer rede.

O servidor da aplicação *WebGoat* funcionou em um computador que possui a seguinte configuração: processador Intel i3, 4 GB de RAM, 500 GB de disco rígido e Sistema Operacional Windows 10. O *WebGoat* 7.1 foi desenvolvido na linguagem de programação Java, utilizando o servidor *Web* Apache.

5.2 – Requisito Legais

O *WebGoat* é adota um modelo de licença livre para o código-fonte, isto significa que ele está sujeito à Licença Pública Geral (*General Public License v2*) quanto às normas de seu uso e distribuição. A Fundação do Software Livre (*Free Software Foundation*, em inglês) é a organização sem fins lucrativos que regulamenta a Licença Pública Geral. Segundo a *Free Software Foundation* (2015):

> Por "software livre" devemos entender aquele software que respeita a liberdade e senso de comunidade dos usuários. Grosso modo, os usuários possuem a liberdade de executar, copiar, distribuir, estudar, mudar e melhorar o software.

Sendo assim é lícito a qualquer pessoa, obter uma cópia do *WebGoat* e testá-la em uma rede particular, desvinculada de qualquer rede pública, com o objetivo de estudar a aplicação. Ao testá-la em um ambiente acadêmico controlado, em uma rede isolada e apartada de qualquer rede, visamos não incorrer no crime de invasão de dispositivo informático, previsto na Lei Nr 12.747 de 30 de Novembro de 2012:

> Invadir dispositivo informático alheio, conectado ou não à rede de computadores, mediante violação indevida de mecanismo de segurança e com o fim de obter, adulterar ou destruir dados ou informações sem autorização expressa ou tácita do titular do dispositivo ou instalar vulnerabilidades para obter vantagem ilícita.

5.3 – Metodologia de Testes

A auditoria pode ser definida como uma atividade que engloba o exame das operações, processos, sistemas e responsabilidades gerenciais de uma determinada entidade, com o intuito de verificar a sua conformidade com certos objetivos e políticas institucionais, orçamentos, regras, normas ou padrões (MELLO, 2015).

A metodologia utilizada para a realização dos testes foi a OWASP Testing Guide v4 OTG-INPVAL-001 que descreve como deve ser conduzido um ataque de *Cross-Site Scripting* do tipo "refletido" e a OWASP Testing Guide v4 OTG-INPVAL-002 que descreve como deve ser conduzido um ataque de *Cross-Site Scripting* do tipo "armazenado".

Cada vetor de entrada de dados das páginas do *WebGoat* relacionadas à XSS será submetida às técnicas de detecção de vulnerabilidade (*Detection Techniques*) e às técnicas de exploração de vulnerabilidade (*Exploitation Techniques*).

5.3.1 – Realização dos Testes

Seguindo a metodologia da OWASP, um teste para verificação de vulnerabilidade à ataques XSS ocorre em três etapas:

a) Detecção vetores de entrada: Para cada página da Web, o testador deve determinar todas as variáveis definidas pelo usuário da aplicação Web e como inseri-las. Isso inclui entradas ocultas ou não evidentes, como parâmetros HTTP, dados POST, valores de campo de formulário oculto e valores de rádio ou seleção predefinidos. Normalmente, os editores HTML no navegador ou *proxies* Web são usados para visualizar essas variáveis ocultas.

b) Análise de cada vetor de entrada para detectar potenciais vulnerabilidades: Para detectar uma vulnerabilidade XSS, o testador geralmente usará dados de entrada especialmente criados com cada vetor de entrada. Esses dados de entrada normalmente são inofensivos, mas desencadeiam respostas do navegador da Web que manifestam a vulnerabilidade. Os dados de teste podem ser gerados usando uma lista predefinida automatizada de ataque conhecidas ou manualmente. Um exemplo desse dado de entrada é o seguinte:

<script>alert("Teste de XSS!")</script>

c) Para cada entrada de teste tentada na fase anterior, o testador analisará o resultado e determinará se representa uma vulnerabilidade que tenha um impacto realista na segurança da aplicação Web.

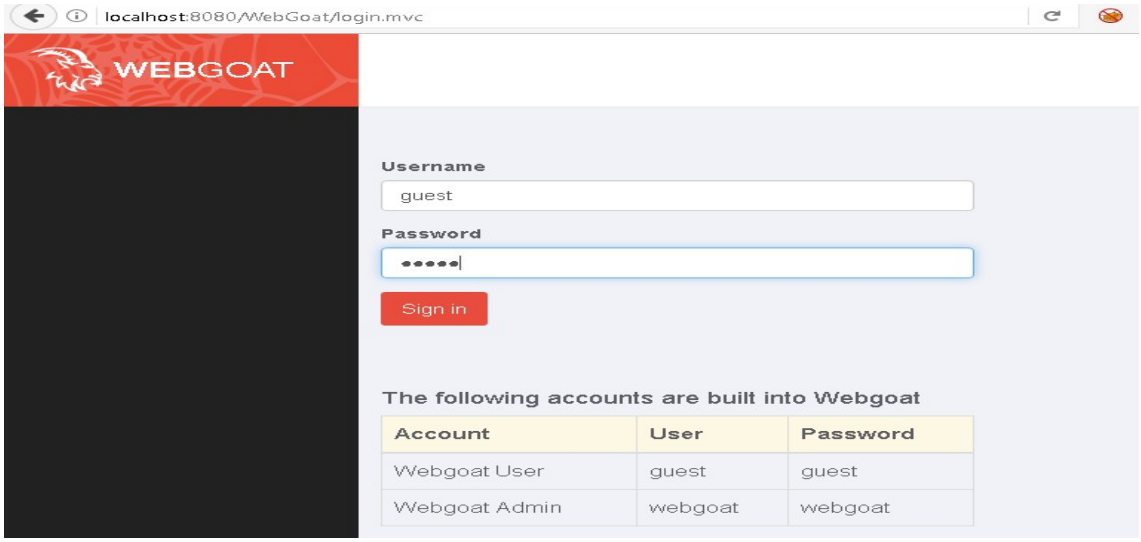
6 APRESENTAÇÃO E ANÁLISE DOS DADOS COLETADOS

6.1 – Apresentação dos Dados Coletados

Cross-Site Scripting é uma das vulnerabilidades mais comuns que existem na atualidade e, consequentemente, uma das mais exploradas (ACUTINEX, 2016). O objetivo de atacar uma aplicação Web através dessa técnica é entender o conceito por trás desse ataque e verificar suas consequências para uma empresa.

O *WebGoat* é uma aplicação Web programada, intencionalmente, de forma que seja insegura pela OWASP. Ela é voltada para treinamentos de invasão e defesa e demonstração de vulnerabilidades (OWASP, 2014). Ele pode ser instalado através do *Website* http://owasp.org na seção *Downloads*. Após a instalação somos avisados, antes de executar o *WebGoat*, que o nosso Sistema Operacional ficará vulnerável à invasões e que devemos tomar precauções para não atacarmos máquinas sem autorização.

Figura 3 – Página inicial do *WebGoat*



Fonte: o autor, 2017

Nos primeiros dias da Internet, as páginas eram estáticas, isto é, um usuário poderia apenas visualizá-la, mas não modificar e nem adicionar nenhum tipo de conteúdo, somente o administrador do servidor Web teria esse privilégio. Com o tempo foram surgindo diversas maneiras de torná-las dinâmicas, iniciando-se o período de aplicações Web (ECCOUNCIL, 2012). São justamente as aplicações Web que permitem a modificação, de alguma maneira, do

Website, as mais vulneráveis à ataques de *Cross-Site Scripting* ou XSS, como também é conhecido.

6.1.1. *Cross-Site Scripting* do tipo "refletido" (Reflected XSS Attacks)

Essa lição do *WebGoat* se parece uma típica página de compra de produtos. Os campos de entrada de dados incluem as quantidades dos itens selecionais, o cartão de crédito e o código de três dígitos do cartão. É possível mudar as quantidades e clicar em "*update*" para atualizar a página. Seguindo a nossa metodologia, vamos prosseguir com a detecção dos vetores de entrada.

Requisição 3 – Requisição enviada através do método POST

POST http://127.0.0.1:8080/*WebGoat*/attack?Screen=1406352188&menu=900
[...]
Post Data:
QTY1[1]
QTY2[1]
QTY3[1]
QTY4[1]
field2[4128+3214+0002+1999]
field1[111]
SUBMIT[Purchase]

Fonte: o autor, 2017

Podemos observar que existem 6 (seis) vetores de entrada de dados na aplicação, a saber: QTY1, QTY2, QTY3, QTY4, field2, field1 e SUBMIT. A seguir será enviada um requisição alterada a fim de testar a aplicação com o script:

<SCRIPT>alert('Teste de XSS + Campo');</SCRIPT>

Requisição 4 – Requisição enviada através do método POST

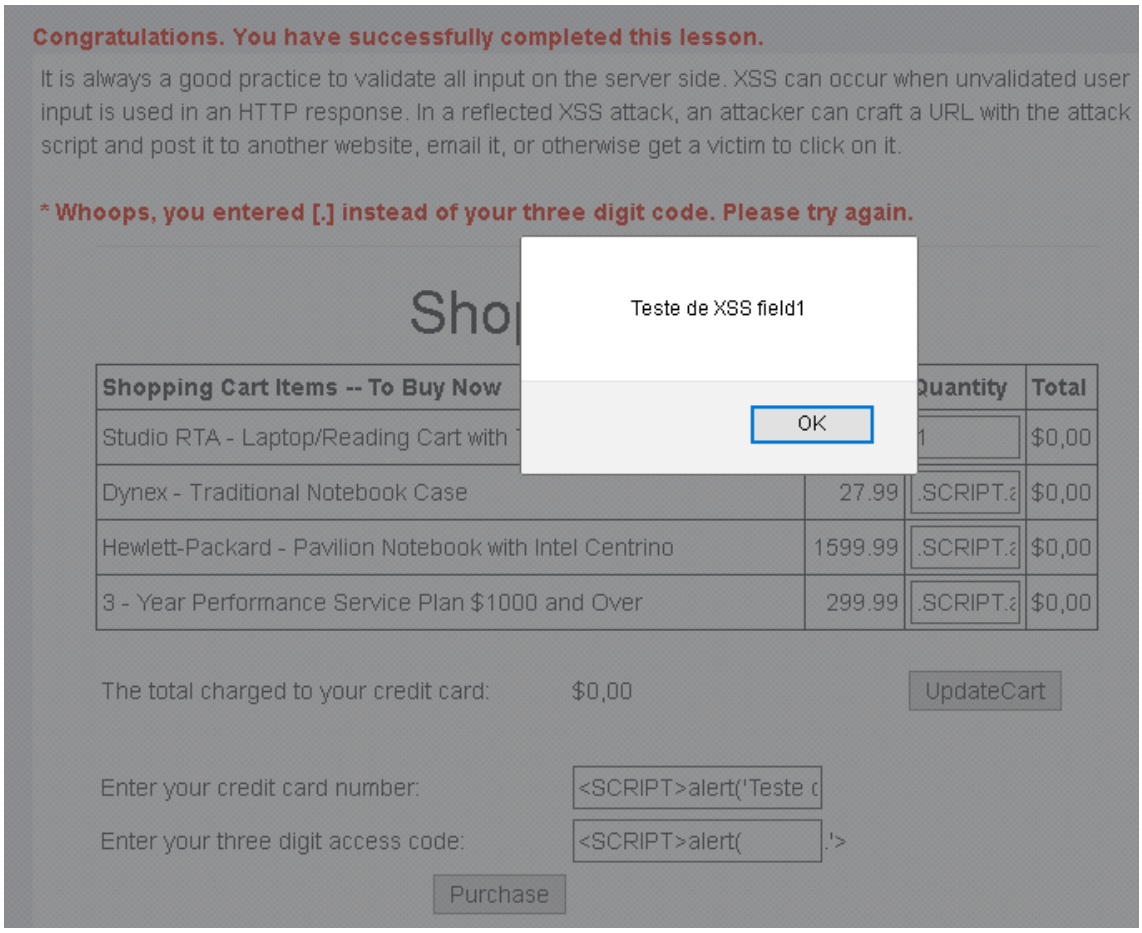
POST http://127.0.0.1:8080/*WebGoat*/attack?Screen=1406352188&menu=900
[…]
Post Data:
QTY1[<SCRIPT>alert('Teste de XSS QTY1');</SCRIPT>]
QTY2[<SCRIPT>alert('Teste de XSS QTY2');</SCRIPT>]
QTY3[<SCRIPT>alert('Teste de XSS QTY3');</SCRIPT>]
QTY4[<SCRIPT>alert('Teste de XSS QTY4');</SCRIPT>]
field2[<SCRIPT>alert('Teste de XSS field2');</SCRIPT>]
field1[<SCRIPT>alert('Teste de XSS field1');</SCRIPT>]
SUBMIT[<SCRIPT>alert('Teste de XSS SUBMIT');</SCRIPT>]

Fonte: o autor, 2017.

Essa página tem uma vulnerabilidade: o campo para os três dígitos do código do cartão (field1) não tem nenhum tipo de validação. Para esta lição realizamos um ataque do tipo "refletido", enviando dados para a aplicação através de diversos campos. O único campo vulnerável foi o "field1", como demonstra a figura abaixo.

A análise do ataque e suas consequências será exposto no tópico a seguinte.

Figura 4 – Demonstração do ataque XSS refletido



Fonte: o autor, 2017.

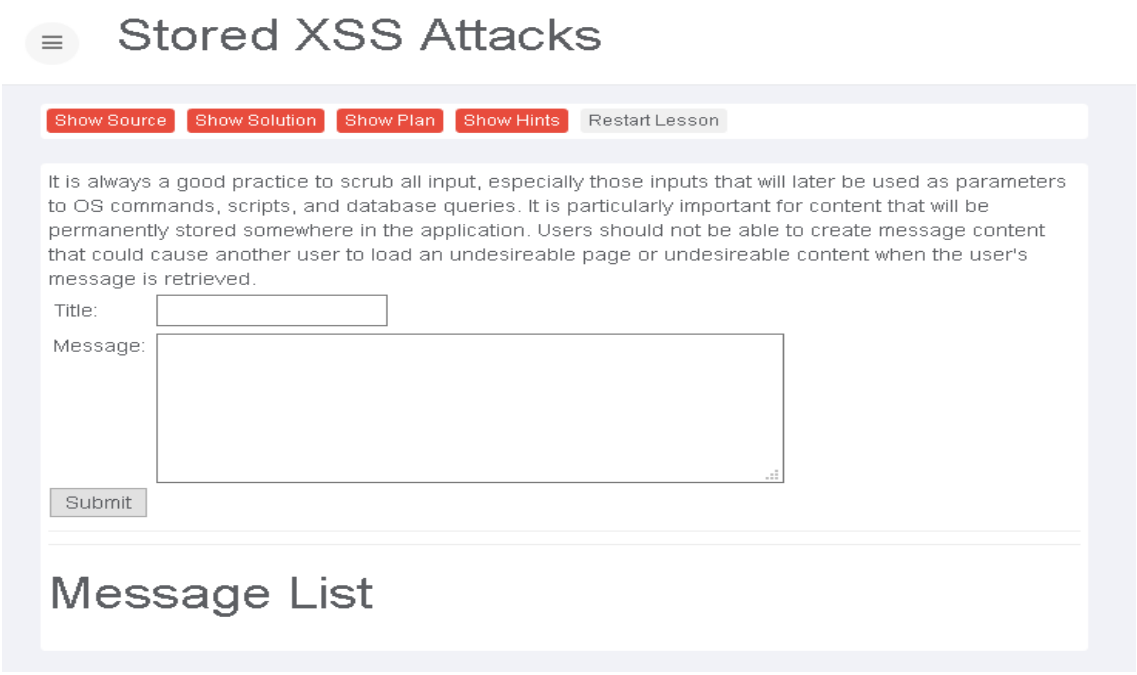
6.1.2. *Cross-Site Scripting* do tipo "armazenado"

Ataques de *Cross-Site Scripting* do tipo "armazenado" podem ser mais perigosos por várias razões. Primeiramente, é fácil de fazer usuários executarem os *scripts* armazenados, pois o navegador os inicia automaticamente ao acessar a aplicação Web. Quando se recebe e-mails

não solicitados é simples não clicar nos links, porém esse tipo de ataque pode comprometer *Website*s vistados regularmente por milhões de pessoas.

Outra variação perigosa desse ataque é quando a vítima está logada na aplicação Web quando executa o *script* malicioso, permitindo ao atacante roubar seus cookies e realizar o sequestro de sua conta. Neste caso as possibilidades do atacante são ilimitadas. Essa lição do *WebGoat* tem por objetivo ensinar sobre as técnicas de *Cross-Site Scripting* do tipo "armazenado".

Figura 5 – Página de XSS do tipo "armazenado"



Fonte: o autor, 2017.

Nessa lição o *WebGoat* simula um pequeno fórum de mensagens. É possível inserir o título e o texto e salvá-lo, para que seja mostrado na lista de mensagens. Seguindo a nossa metodologia, vamos prosseguir com a detecção dos vetores de entrada.

Requisição 5 – Requisição enviada através do método POST

```
POST http://127.0.0.1:8080/WebGoat/attack?Screen=598569451&menu=900
[...]
Post Data:
title[Teste]
```

message[Teste]
SUBMIT[Submit]

Fonte: o autor, 2017.

Podemos observar que existem 3 (três) vetores de entrada de dados na aplicação, a saber: title, message e SUBMIT. A seguir será enviada um requisição alterada a fim de testar a aplicação com o script:

<script language="javascript" type="text/javascript">alert("Teste de XSS + Campo");</script>

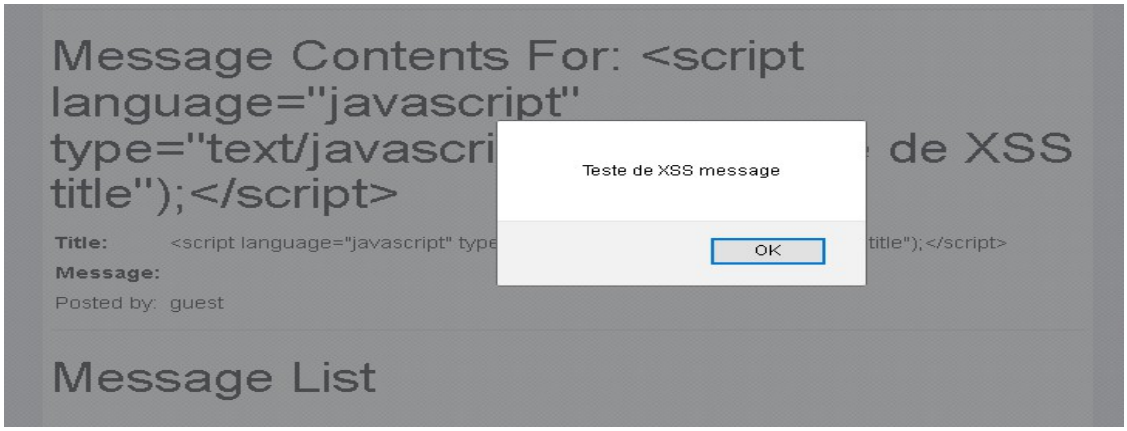
Requisição 6 – Requisição enviada através do método POST

POST http://127.0.0.1:8080/*WebGoat*/attack?Screen=598569451&menu=900
[...]
Post Data:
title[<script language="javascript" type="text/javascript">alert("Teste de XSS title");</script>]
message[<script language="javascript" type="text/javascript">alert("Teste de XSS message"); </script>]
SUBMIT[<script language="javascript" type="text/javascript">alert("Teste de XSS SUBMIT");</script>]

Fonte: o autor, 2017.

O título e o conteúdo da mensagem não são importantes, somente o *script* inserido. Após a mensagem ser enviado será possível visualizá-la na lista de mensagens.

Figura 6 – Demonstração do ataque XSS armazenado



Fonte: o autor, 2017.

6.2 – Análise dos Dados Coletados

Na primeira lição, a de XSS do tipo “refletido”, quando o botão “Comprar” (*Purchase*) foi selecionado, o *script* dentro do formulário foi enviado ao servidor Web. O servidor *Web* não realizou nenhum forma de validação no campo código do cartão, permitindo que o *script* fosse inserido juntamente com o código do cartão “111”. Quando o servidor enviou a página atualizada de volta ao navegador do usuário, o *script* foi incluído e executado.

Embora se trate de uma demonstração inofensiva do ataque, deve-se atentar para as suas consequências mais severas. O que aconteceria se alguém lhe enviasse um *URL* com um *script* de uma página que você considere segura, como o sistema de gerenciamento eletrônico de documentos da empresa? Muitos funcionários já tem o discernimento de não clicar em páginas suspeitas, mas seguramente poucos usuários tem o conhecimento de que mesmo páginas pertencentes a instituições verídicas podem ser alvos desse tipo de ataque.

Na segunda lição, a de XSS do tipo “armazenado”, o ataque foi direcionado a inserir um *script*, de forma persistente, na aplicação Web, de forma a que os usuários da aplicação sejam afetados. Diferentemente da primeira lição, o atacante não precisa enviar *URL*s para as vítimas nem fazer divulgação do *Website*, basta que os usuários o acessem que já estarão comprometidos, tendo em vista que o *script* foi armazenado no servidor.

Os principais métodos para se evitar esse tipo de vulnerabilidade é se assegurar de que todos os vetores de entrada de dados dos usuários não são confiáveis, isto é, eles devem ser verificados para que se tenha certeza que não contenham nenhum conteúdo dinâmico (Javascript, ActiveX, Flash, etc) e que sejam codificados de maneira apropriada. Uma solução muito conhecida para a segurança das aplicações Web é o *Firewall* de Aplicações Web (*Web Application Firewall*), que pode ser tanto um dispositivo de rede como um *software*. Ele serve para adicionar recursos de segurança na aplicação, tendo a capacidade de prevenir ataques, realizar monitoramento e detecção de intrusão (CLARKE, 2012).

Somado a essa solução, o tráfego deve ser monitorado com um *Web Application Security* (WAP) para que mesmos esses usuários com poucos direitos tenham suas conexões verificadas, filtradas e bloqueadas de acordo com regras baseadas nos *scripts* semelhantes às apontadas nessa pesquisa.

7 CONCLUSÕES

Nossa pesquisa teve como objetivo verificar como os ataques de *Cross-Site Scripting* ocorrem, quais as suas consequências para uma empresa e como podem ser evitados. Foram observados alguns objetivos específicos como: metodologia de um ataque e uma simulação à aplicação Web *WebGoat*, de modo a verificar o nível de segurança da aplicação e as principais soluções para vulnerabilidades verificadas em auditorias em aplicações Web.

A necessidade de realização de auditoria na área da tecnologia da informação faz-se presente pela importância que os sistemas integrados de informação têm para com uma organização. Como os sistemas na atualidade são vitais para a continuidade e competitividade de uma organização, não é aconselhável e seguro deixá-los sem constantes auditorias. Em muitos casos, os erros e deficiências nos sistemas computacionais só são percebidos quando ocorrem, gerando, certamente, prejuízos que poderiam ser evitados por auditorias (BOTELHO, 2012).

Os resultados encontrados demonstraram a facilidade de se explorar vulnerabilidades de *Cross-Site Scripting* em aplicações Web que não validam as entradas de dados do usuário, o que possibilita esse tipo de ataque. As consequências para o usuário variam de acordo com a natureza da vulnerabilidade, podendo ir de um inofensivo pop-up à sequestro de sessões válidas no sistema, roubo de credenciais ou realização de atividades arbitrárias em nome do usuário afetado. Para a empresa o impacto é principalmente no tocante à sua imagem, e a possibilidade de utilização da falha para a distribuição de *phishing* e facilitação de fraudes.

Como vantagens da realização de uma auditoria em Aplicações Web, podemos citar que ela reduz os riscos relacionados à TI, pois avalia a disponibilidade, integridade e confidencialidade dos sistemas dando uma visão clara das ações necessárias para reduzir ou eliminar os riscos, melhora a segurança dos dados, pois os pontos de falha dos sistemas são reajustados e melhora a governança em TI, pois garante que a empresa cumpra normas, regulamentos, conformidades e leis relacionadas ao seu negócio.

A empresa pode enfrentar algumas dificuldades para a realização de sua auditoria, tais como: falta de profissionais qualificados, tendo em vista que muitos

sistemas podem ser implementados sem que seus administradores tenham ciência de seu funcionamento, falta de cultura da empresa, pois implementar algo novo sempre requer mudança de rotina, e tecnologia variada e abrangente, nesta pesquisa realizamos testes referente somente a Cross-Site script, mas aplicações Web complexas utilizam uma infinidade de recursos, cada qual sendo sujeito à ataques variados.

Foram observados métodos para que a empresa evite esse tipo de vulnerabilidade, sendo o principal se assegurar de que todos os vetores de entrada de dados dos usuários não são confiáveis, isto é, que eles devem ser verificados para que se tenha certeza que não contenham nenhum conteúdo dinâmico (Javascript, ActiveX, Flash, etc) e que sejam codificados de maneira apropriada. Como simples exemplo de uma função, em linguagem PHP, que impede o *Cross-Site Scripting* podemos citar o *htmlspecialchars()*:

```
$variavel = htmlspecialchars($dados_usuario, ENT_QUOTES, 'UTF-8')
```

Esse comando, entre outras funções, remove possíveis scripts da variável dados_usuário. Para outro método de defesa sugere-se a utilização de um filtro de aplicações Web, mas conhecido como *Web Application Firewall* (WAF), que é um software que trabalha entre o servidor Web e cliente, filtrando entradas do cliente e saídas e seguindo sempre regras de segurança, tornando possível registrar ataques e bloquear. Vale citar que do lado do usuário estão sendo implementados mecanismos de prevenção nos navegadores modernos.

8 REFERÊNCIAS

ACUTINEX. **Acunetix *Web* Application Vulnerability Report 2016**. Disponível em: <https://www.acunetix.com/acunetix-*Web*-application-vulnerability-report-2016/>. Acessoem:19Mar2017.

ASSUNÇÃO, Marcos Fávio Araújo. **Segredos do Hacker Ético**. 2ª Ed. Florianópolis: Visual Books, 2008.

CLARKE, Justin. **SQL Injection Attacks and Defense**. Burlington: Syngress, 2009.

ECCOUNCIL. **Cross-Site Scripting**. Certified Ethical Hacker v7. 2012. Disponível em: <http://thomaslorenzato.fr/CEH/CEHV7-Module2012-XSS.pdf>. Acesso em: 10 Set. 2014.

FREE SOFTWARE FOUNDATION. **A Definição de Software Livre**. Disponível em:

http://www.gnu.org/philosophy/free-sw.pt-br.html Acesso em 20 Maio. 2015.

GUMERATO, Ronaldo. **SQL Injection em aplicações *Web***. Uberlândia: 2009.

GOODRICH, Michael T.; Roberto Tamassia (2013). **Introdução à Segurança de Computadores**.1ª ed. Porto Alegre: Bookman.

JÚNIOR, Armando Gonçalves da Silva. **Cross-Site Scripting: Uma Análise Prática**. Recife, PE, 2009.

LENTO, Luiz Otávio Botelho; Márcio Ghishi Guimarães. Auditoria de segurança da informação : livro digital. Palhoça: UnisulVirtual, 2012.

MELLO, Agostinho de Oliveira. Instituto dos auditores internos do Brasil. **Organização básica da auditoria interna**. Biblioteca Técnica de Auditoria Interna. 2005.

OWASP. ***WebGoat* Project**. 5 Set. 2014. Disponível em: <https://www.owasp.org/index.php/Category:OWASP_*WebGoat*_Project>. Acesso em 10 Set. 2014

ULBRICH, Henrique César. **Livro de Exercícios Universidade Hacker**. 2º Ed. São Paulo: Digerati, 2009.