UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

INSTRUMENTALIZAÇÃO DA ANÁLISE E PROJETO DE *SOFTWARE SEGURO* BASEADA EM AMEAÇAS E PADRÕES

FABRÍCIO ATAIDES BRAZ

ORIENTADOR:

LUIS FERNANDO RAMOS MOLINARO

TESE DE DOUTORADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.TD - 038/2009

BRASÍLIA, DF: ABRIL/2009

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

INSTRUMENTALIZAÇÃO DA ANÁLISE E PROJETO DE *SOFTWARE* SEGURO BASEADA EM AMEAÇAS E PADRÕES

FABRÍCIO ATAIDES BRAZ

TESE DE DOUTORADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR.

APROVADA POR:

LUIS FERNANDO RAMOS MOLINARO, Dr., ENE/UNB

(ORIENTADOR)

FLÁVIO ELIAS GOMES DE DEUS, Dr., ENE/UnB

(EXAMINADOR INTERNO)

EDUARDO FERNANDEZ-BUGLIONI, Dr., CSE/FAU/USA

(EXAMINADOR EXTERNO)

MARIA MERCEDES LARRONDO PETRIE, Dra., CSE/FAU/USA

(EXAMINADORA EXTERNA)

MAMEDE LIMA-MARQUES, Dr., UNB

(EXAMINADOR INTERNO)

BRASILIA, DF, 16 DE ABRIL DE 2.009.

Braz, Fabricio Ataides

Instrumentalização da Análise e Projeto de *Software* Seguro Baseada em Ameaças e Padrões [Distrito Federal], 2.009, XI, 137, 297 mm (ENE/FT/UnB, Doutor, Engenharia, 2.009) Tese de Doutorado – Universidade de Brasília, Faculdade de Tecnologia, Departamento de Engenharia Elétrica.

1	Engenharia de *Software*	2	Segurança da Informação
3	*Software* Seguro	4	Padrões de Arquitetura de *Software*
I.	ENE/FT/UnB	II.	Título

REFERÊNCIA BIBLIOGRÁFICA

BRAZ, F.A. (2009). Instrumentalização da Análise e Projeto de *Software* Seguro Baseada em Ameaças e Padrões. Publicação PPGENE.TD - 038/2009, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília DF, 137p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Fabrício Ataides Braz

TÍTULO DA TESE: Instrumentalização da Análise e Projeto de *Software* Seguro Baseada em Ameaças e Padrões.

GRAU/ANO: Doutor/2009

Fabrício Ataides Braz

SQN 308 Bloco D Apto. 504, Asa Norte.

70.747-020 Brasília - DF - Brasil.

Às minhas jóias preciosas Camila e Beatriz!

Agradecimentos

Agradeço:

– ao mestre dos mestres, Deus Pai, pelas oportunidades colocadas em meu caminho;

– ao Professor Mamede Lima-Marques pela orientação, determinante para o desenvolvimento desta tese. Sua habilidade, sensibilidade e fino trato ajudaram a administrar a ansiedade constante nessa jornada.

– ao Professor Eduardo B. Fernandez pela orientação e contribuição que permitiram a conclusão desta tese. Além disso, sua hospitalidade e gentileza ajudaram a amenizar as dificuldades provocadas pela distância da família e amigos do Brasil durante o período na Flórida.

– ao Professor Luiz Fernando Ramos Molinaro pela orientação e por ter aberto as portas da UnB em 2001 no mestrado, relacionamento que se consolidou em 2004, momento no qual me ingressei no programa de doutorado;

– aos professores do *Secure Systems Research Group* da *Florida Atlantic University* (FAU) Michael van Hilst e Maria Petrie .

– aos colegas do *Secure Systems Research Group* da FAU, em especial a Ingrid Buckley e Keiko Hashizume.

– à Microsoft Brasil, na pessoa de Marinês Gomes, pela parceria, cujo investimento possibilitou em parte a dedicação à essa pesquisa.

– às pessoas que se envolveram no projeto de segurança de software do SERPRO, em especial: Professor Jorge Fernandes, Marcos Allemand, Fernando Cima e Paulo Hidaka.

– ao amigo Edson Pagotto e sua família, pelo suporte dispensado no período em que estive na FAU.

– à Sra. Ruanita Keyes, locatária do quarto em Boca Raton, cujos diálogos contribuíram para acelerar o período de adaptação.

– aos meus pais, irmãos pelo incentivo constante ao longo dessa pesquisa.

– aos amigos que sempre motivaram e torceram para a conclusão desse ciclo, em especial: Adriano Santana, Daniela Garrossini, Fábio Buiati, Fabricio Olivieri, Flávio Elias de Deus, Gilberto Borges, Gloria Borges, Guilherme Borges, Honório Crispim, Roberto Zucca.

– aos colegas do Centro de Pesquisas em Arquitetura da Informação (CPAI).

– aos colegas do Núcleo de Multimídia e Internet (NMI).

– aos colegas e professores da pós-graduação em Engenharia Elétrica da Universidade de Brasília.

– à Denise Goulart, pela revisão do texto.

Por fim, agradeço a todo o incentivo e suporte dado pela minha esposa Camila, que cobriu a minha ausência e se desdobrou nos cuidados com o nosso lar e com a nossa filha Beatriz nos momentos em que estive ausente ao longo do doutorado. Em especial, agradeço pelo suporte no período em que fiquei nos EUA. Sou muito privilegiado por ter uma companheira como ela.

"A verdadeira viagem do descobrimento não consiste na procura de novas paisagens, mas em ter novos olhos"

Marcel Proust

Resumo

INSTRUMENTALIZAÇÃO DA ANÁLISE E PROJETO DE *SOFTWARE SEGURO* BASEADA EM AMEAÇAS E PADRÕES

Autor: Fabrício Ataides Braz

Orientador: Luis Fernando Ramos Molinaro

Programa de Pós-graduação em Engenharia Elétrica

Brasília, abril de 2.009

As perdas resultantes de ataques habilitados por falhas de segurança encontradas no *software* estão em escala crescente, razão pela qual se demanda novas soluções que facilitem na obtenção de *softwares* menos vulneráveis. Nesta tese, foram desenvolvidos instrumentos que possam auxiliar no desenvolvimento de softwares mais seguros que se integrem à metodologia de desenvolvimento de software seguro baseada em padrões. O primeiro instrumento é representado pela técnica de elicitação de requisitos *atividades de abuso*, cuja proposta original passou por uma melhoria de modo a incorporar a análise sobre o tipo de ameça (*spoofing, tampering, repudiation*...) e o papel do atacante. O segundo instrumento compreende um método para classificar padrões de segurança que contempla a necessidade de seus usuários. O método usa uma matriz definida pela divisão do espaço do problema por múltiplas dimensões, permitindo que os padrões ocupem células múltiplas na matriz. Por fim, encontra-se um protótipo desenvolvido para dar suporte à análise da atividade de abuso, que faz uso da matriz como método para recuperação do padrão ideal para controlar as ameaças levantadas.

Abstract

INSTRUMENTATION OF SECURE SOFTWARE ANALYSIS AND DESIGN BASED ON THREATS AND PATTERNS

Author: Fabrício Ataides Braz

Supervisor: Luis Fernando Ramos Molinaro

Programa de Pós-graduação em Engenharia Elétrica

Brasília, april of 2.009

The damage from attacks whose root cause is a software security breach has been increasing substantially. As a consequence, an innovation that may turn the software less vulnerable is required. In this dissertation we present some instruments to aid the secure software development, which integrate a methodology to build secure systems based on patterns. The first instrument is a security requirements elicitation approach called misuse activities. This approach has been extended to consider in the analysis the type of misuse (spoofing, tampering, repudiation ...) that can happen in each activity, the role of the attacker, and the context for the threat. The second instrument a classification for security patterns that addresses the needs of users. The approach uses a matrix defined by dividing the problem space along multiple dimensions, and allows patterns to occupy regions, defined by multiple cells in the matrix. It's also presented a prototype tool to aid when applying the misuse activities approach, which adopts the matrix of concerns as the method to recommend the best set of patterns to control the threats.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ABAS *Attribute-Based Architecture Style*

ACL *Access Control List*

ADD *Atribute-Driven Design*

ALE *Annual Loss Expectancy*

ANSI *American National Standards Institute*

AP Área de Processo

API *Application Programming Interface*

APTIA *Analytic Principles and Tools for Improvement of Architectures*

ATAM *Architecture Trade-off Analysis Method*

AS *Architectural Strategy*

BSI *Building Security In*

CBAM *Cost-Based Analysis Method*

CC *Common Criteria*

CERT/CC *Computer Emergency Response Team/Coordination Center*

CLASP *Comprehensive, Lightweight Application Security Process*

CMM *Capability Maturity Model*

CORBA *Common Object Request Broker Architecture*

COTS *Commercial Off-the-shelf*

CPAI Centro de Pesquisas em Arquitetura da Informação

CRUD *Create Read Updade Delete*

CSI *Computer Security Institute*

DDoS *Distributed Denial of Service*

DHS *Department of Homeland Security*

DLL *Dynamic-Link Library*

DoD *Department of Defense*

FAU *Florida Atlantic University*

FBI *Federal Bureau of Investigation*

FISMA *Federal Information Security Management Act*

FMEA *Failure Modes and Effects Analysis*

GQ Gerência de Qualidade

HIPAA *Health Insurance Portability and Accountability Act*

HTTP *Hypertext Transfer Protocol*

IBM *International Business Machines*

IEC *International Electrotechnical Commission*

INFOSEC *National Training Standard for Information Systems Security*

ISO *International Organization for Standardization*

IEEE *Institute of Electrical and Electronics Engineers*

IIS *Internet Information Services*

J2EE *Java 2 Enterprise Edition*

LPO Lógica de Primeira Ordem

MAA *Misuse Activity Analysis*

MOQARE *Misuse-Oriented Quality Requirements Engineering*

MP3 *MPEG 1 Layer-3*

NCSP *National Cyber Security Partnership Strategy*

NFR *Non-functional Requirement*

NIST *National Institute of Standards and Technology*

NMI Núcleo de Multimídia e Internet

OLAP *on-line Analytical Processing*

OMG *Object Management Group*

OOPSLA *Object-Oriented Programming, Systems, Languages & Applications*

OWASP *Open Web Application Security Project*

PCI *Payment Card Industry*

PB Prática Base

PG Prática Genérica

PLoP *Pattern Languages of Programs Conference*

Prolog Programação em Lógica

PSM *Platform Specific Model*

PIM *Platform Independent Model*

RUP *Rational Unified Process*

S4V *Siemens 4 Views*

SAAM *Software Architecture Analysis Method*

SAML *Security Assertion Markup Language*

SDL *Security Development Lifecycle*

SEI *Software Engineering Institute*

SERPRO Serviço Federal de Processamento de Dados

SCADA *Supervisory Control And Data Acquisition*

SOA *Service Oriented Architecture*

SOX *Sarbanes-Oxley Act*

SQL *Structured Query Language*

SQUARE *Security Quality Requirements Engineering*

SSE-CMM *Security System Engineering Capability Maturity Model*

SSH *Secure Shell*

SSL *Secure Sockets Layer*

STRIDE *Spoofing, Tampering, Repudiation, Denial of Service, and Elevation of Privilege*

TAM *Threat Analysis and Modeling Tool*

TI Tecnologia da Informação

TLS *Transport Layer Security*

ToE *Target of Evaluation*

UML *Unified Modeling Language*

XACML *eXtensible Access Control Markup Language*

XMI *XML Metadata Interchange*

XML *eXtended Markup Language*

XSS *Cross Site Script*

WebDAV *Web-based Distributed Authoring and Versioning*

1 *Introdução*

As organizações que desenvolvem e mantêm *software* estão cada vez mais obrigadas a garantir a segurança oferecida pelos seus produtos. Essa necessidade é motivada pelas expressivas perdas, tangíveis ou não, acarretadas por ataques habilitados por falhas de segurança no *software*.

Entretanto, observa-se que as organizações que se lançam no desenvolvimento de *software* seguro encontram bastante dificuldade causada por várias razões, incluindo-se: o despreparo técnico dos desenvolvedores para lidar com esse problema e a falta de suporte metodológico e ferramental adequados para aplicar a segurança ao longo do ciclo de desenvolvimento de *software*. Soma-se, ainda, o incremento constante na sofisticação das técnicas e do aparato tecnológico dos atacantes.

Em resposta a essa realidade, esta tese de doutorado busca facilitar o desenvolvimento de *software* seguro, a partir do uso de instrumentos especialmente concebidos para dar apoio às fases de requisitos e projeto do ciclo de desenvolvimento de *software*.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo principal desta tese é desenvolver instrumentos de apoio à análise e projeto de *software* seguro na forma de técnica, procedimento e método que explorem a perspectiva de

ameaça e o conhecimento retido nos padrões de *software*.

1.1.2 Objetivos Específicos

- Propor uma técnica de elicitação de requisitos de segurança de *software* que auxilie o analista no levantamento de todas as possíveis ameaças ao *software* em questão.

- Desenvolver um instrumento que facilite a recuperação de padrões de segurança de *software*, para facilitar sua adoção durante o desenvolvimento e manutenção do *software*.

- Propor uma especificação de um *software* para facilitar a análise e o projeto de *software* que se baseie em ameaças e padrões.

1.2 Justificativa

Desenvolver *software* seguro é um problema importante e difícil (REDWINE, 2007). Sua importância está relacionada basicamente ao impacto e tendência. Uma pesquisa realizada em parceria entre o *Computer Security Institute* (CSI) e o *Federal Bureau of Investigation* (FBI) revelou que, anualmente, cerca da metade de todas as bases de dados sofreram ataques, gerando uma perda média de quase US$ 4 milhões (Computer Security Institute, 2002). Mostrou também o uso estabelecido da *Internet* para crimes, que variam de vandalismo a roubo de informação sensível e fraude financeira. As estatísticas divulgadas são bem negativas e revelam-se pessimistas quanto à mudança desse quadro. O CERT-CC (2008), por exemplo, registrou um crescimento de 3867% no índice de vulnerabilidades de 1995 a 2005, consolidado pela Figura 1. A dificuldade vem da complexidade inerente ao desenvolvimento de *software* e da amplitude das iniciativas necessárias para se chegar a um resultado adequado em termos de segurança, que não raro devem contemplar áreas como autenticação, auditoria, autorização, confidencialidade,

integridade e não repúdio (COMMON CRITERIA SPONSORING ORGANIZATIONS, 2006b).

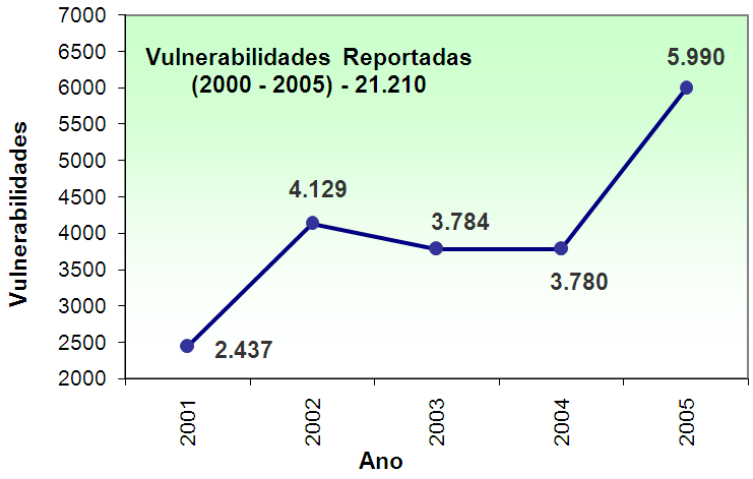

Figura 1: Vulnerabilidades Reportadas ao CERT/CC, modificado de (CERT-CC, 2008)

Não surpreende o fato da maioria das vulnerabilidades divulgadas hoje ter como origem *software* de baixa qualidade, conforme mostrado por Viega e McGraw (2001), na medida em que o estado da arte em engenharia de segurança encontra-se desbalanceado. No topo encontra-se a *camada criptográfica* com suas limitações e problemas em aberto identificados (KONHEIM, 2007). Depois vem a *camada de protocolo de segurança*, com uma variedade de procedimentos e padrões para serviços, como comunicação segura e autenticação. O estado da arte nessa camada fornece lógicas específicas e técnicas formais de verificação de protocolos que evidenciam erros e pressupostos equivocados (DIMACS, 1997). A *camada de sistema* oferece serviços padrão, como SSH, SSL e tecnologias como Anti-XSS e HTTP*Only*. Ela fornece certo nível de segurança, mas ainda é suscetível a ataques, como, por exemplo, o DDoS. Já o estado da arte no que tange à *camada de aplicação* é bem mais limitado, com poucas evidências quantitativas de sucesso (DAVIS et al., 2004; GOERTZEL et al., 2006; REDWINE, 2007).

Até recentemente, a resposta padrão dos fornecedores de *software* frente aos seus problemas de segurança consistia na liberação de atualização contendo a correção da última vulnerabilidade encontrada ou culpar os usuários pela imperícia, porém, apoiar-se em correções

e na boa intenção do usuário não é uma solução razoável. Muitos sistemas não são atualizados no prazo necessário (RESCORLA, 2003). Quando o são, a própria correção pode causar novas vulnerabilidades (National Cyber Security Partnership Strategy, 2004; KREBS, 2008). Basicamente, são empregadas duas abordagens para tratar dessa problemática: 1) examinar o resultado final do desenvolvimento na tentativa de localizar possíveis problemas, como, por exemplo, estouro de *buffer*, ou 2) planejar para que a segurança seja empreendida desde o princípio do ciclo de desenvolvimento[1], em todas as atividades, incluindo: requisitos, projeto[2], codificação, testes, implantação e manutenção. Esta última abordagem tem demonstrado resultados mais consistentes (MCGRAW, 2006), sendo a opção que embasa esta tese.

Os desenvolvedores se deparam mais e mais com desafios e alternativas. Devido aos padrões usados no passado, as aplicações de hoje são grandes e complexas. Elas envolvem múltiplos domínios, com componentes de várias fontes, usando diferentes tipos de rede e ainda operando sobre várias plataformas. Considere apenas a questão dos componentes-fonte. Eles podem vir de código novo, aplicação legada, produto de prateleira, *software* livre, *web service*, biblioteca reutilizável, *scripts* de tempo de execução, código advindo de geradores automáticos, transformação de modelos, desenvolvimento terceirizado etc. É raro aquele projeto que não faça uso de mais de um par dessas fontes. Maneiras apropriadas de se contemplar a segurança diferem dependendo da situação. Os métodos atuais de ensino e aplicação da segurança não são suficientes. Uma metodologia início-fim, por si, não contempla de forma realista as implicações de segurança de muitas situações e variações em situações facilmente experimentadas em uma única aplicação.

Cumpre observar que, mesmo particularizando para o quesito da segurança, esta problemática tem como origem os mesmos problemas genéricos de qualidade de *software*. Tal

[1] O emprego da segunda abordagem não implica ignorar a primeira. Esta separação reflete as principais abordagens aplicadas pela indústria de *software*.

[2] Como a tradução da palavra *design* do inglês, no contexto de uma das fases do ciclo de desenvolvimento de *software*, não está estabelecida para o português, opta-se pelo uso de *projeto*, assim como adotado pelos tradutores de Engenharia de Software, de Roger Pressman.

fato sinaliza que ações isoladas, como, por exemplo, o uso de criptografia, tem resultados inexpressivos para a segurança como um todo. Ao contrário, deve-se embasar as iniciativas em segurança de *software* nos pilares já estabelecidos pela engenharia de *software*, muitos deles já considerados por Brooks (1987).

Por várias razões, os métodos formais puros ou abordagens práticas *ad hoc* usando programação procedimental não são suficientes para se construir sistemas complexos seguros. Os métodos formais não são convenientes para descrever as propriedades estruturais de um sistema (FERNANDEZ; FRANCE; WEI, 1996; WING, 1998), além de serem de difícil uso para a maioria dos desenvolvedores. Provas em modelos exemplo por si não garantem sua correta codificação pelo desenvolvedor. A programação procedimental não tem o poder de manipular sistemas complexos pela sua deficiência acerca da abordagem conceitual e abstração. As características de segurança devem ser contempladas de maneira ampla em todas as atividades do ciclo de vida do *software*. Métodos de componentes e objetos podem ajudar, mas a segurança deve ser considerada em todos os componentes e todas as interfaces entre eles.

Qualquer vulnerabilidade pode causar ou contribuir para brechas de segurança. Estudos mostram que os ataques podem advir de múltiplas origens, não somente de indivíduos externos à organização, mas principalmente de pessoal do quadro (RANDAZZO et al., 2004). Os sistemas podem ser comprometidos e mantidos indisponíveis a partir de um ataque a qualquer ponto de controle, mesmo aqueles não relacionados com a manipulação de ativos. As vulnerabilidades e a necessidade de se reduzir as perdas devem ser consideradas na especificação dos requisitos, projeto, implementação, teste, implantação, operação, manutenção e qualquer outra atividade no ciclo de vida.

Em se tratando de vulnerabilidades originadas no projeto (detalhes na Seção 2.1.1 do capítulo 2), observa-se que as pesquisas da área de arquitetura de *software* oferecem uma orientação apropriada para tratá-las (CHUNG; NIXON; YU, 1995). Ainda assim, ressalta-se a difi-

culdade na elaboração de um projeto adequado às necessidades de segurança do *software*. Sua realização pressupõe a correta definição dos requisitos de segurança de *software*, bem como a seleção dos mecanismos de segurança apropriados, que permitirão o atendimento dos referidos requisitos. Isso se traduz no envolvimento de desenvolvedores altamente especializados e experientes em segurança de *software*, recurso bastante escasso no mercado de TI (BISHOP; FRINCKE, 2005a; SHUMBA et al., 2006).

Os padrões de arquitetura voltados para segurança representam uma fonte rica para a orientação dos desenvolvedores para o desenvolvimento do projeto de *software*, considerando todos os aspectos inerentes ao *software* (BUSCHMANN; HENNEY; SCHMIDT, 2007), e, em particular, contribuem em alto grau para a sua segurança. A partir deles, torna-se possível que uma *experiência* de projeto bem sucedida envolvendo um determinado *problema*, num *contexto* específico, se imortalize, permitindo que inúmeros outros desenvolvedores possam reutilizar e até aprimorar tal conhecimento (SCHUMACHER et al., 2006). Porém, observa-se uma curva de aprendizado extensa para adquirir o vocabulário de padrões, sendo ela agravada quando se considera a habilidade de uso do padrão correto para a realidade de determinado sistema (HOFSTADER, 2008).

Evidencia-se, então, um alinhamento com o consenso existente na comunidade de que a segurança deve ser considerada desde a concepção do sistema para que ela seja efetiva (DEVANBU; STUBBLEBINE, 2000) e destaca-se a relevância da pesquisa e desenvolvimento de alternativas para facilitar e apoiar o desenvolvimento de *software* seguro.

1.3 Metodologia

A pesquisa realizada pode ser classificada quanto a sua abordagem, como pesquisa qualitativa, uma vez que considera a existência de uma relação dinâmica entre mundo real e sujeito. É

descritiva e utiliza o método indutivo. O processo é o foco principal (FLICK, 2009).

De acordo com sua natureza, é caracterizada como pesquisa aplicada, pois se trata da geração conhecimento para aplicação prática dirigido à solução de problemas específicos. Envolve verdades e interesses locais ao contrário da pesquisa básica, que gera conhecimento para o avanço da ciência sem aplicação prática prevista. O desenvolvimento do protótipo é um forte indício desta característica, pois faz uso de teorias desenvolvidas.

Quanto aos objetivos da pesquisa, evidencia-se tanto a pesquisa exploratória, momento em que uma densa pesquisa bibliográfica foi realizada para delimitação e aprofundamento no tema, bem como a pesquisa explicativa, com a preocupação central de identificar os fatores que determinam ou contribuem para a ocorrência dos fenômenos.

Com relação a sua modalidade, a pesquisa é classificada como uma pesquisa teórica, por ser dedicada a reconstruir uma teoria e propor técnica, processo ou método. Ainda quanto à modalidade, pode ser considerada como pesquisa metodológica em sua proposta de generalização da solução (DEMO, 2000).

1.4 Estrutura

Esta tese está estruturada em capítulos que se iniciam com a Introdução, na qual está registrada uma visão geral da pesquisa, apresentando seus objetivos geral e específicos, a justificativa acerca de sua relevância, bem como os aspectos metodológicos que orientaram o seu desenvolvimento. A realidade da segurança de *software* é apresentada no capítulo 2. O capítulo 3 traz uma revisão sobre a arquitetura de *software*. O capítulo 4 aborda um dos elementos fundamentais para o desenvolvimento desta tese, os padrões de *software*. As contribuições efetivas da pesquisa começam a ser apresentadas no capítulo 5, que aborda a técnica para elicitação de requisitos de segurança baseada em ameaças. Já o capítulo 6

apresenta um método para classificação de padrões de segurança centrado nos seus usuários, e o capítulo 7, registra uma especificação do *software* para dar suporte à aplicação da técnica de requisitos, bem como o uso da classificação de padrões. Já o capítulo 8 traz as conclusões, juntamente com sugestões de trabalhos futuros. Por fim, encontram-se as referências bibliográficas.

2 *Software Seguro*

O esforço desprendido no combate a problemas de segurança computacional ao longo dos últimos anos tem sido enorme. Entre as categorias-alvo, uma das que mais receberam atenção foi a de filtro de pacotes, que, por consequência, encontra-se em estágio avançado de desenvolvimento. O mesmo não se pode dizer de aplicação, que passou a receber maior ênfase da última década em diante (DEVANBU; STUBBLEBINE, 2000). Como resultante, observa-se índices elevados de ataques, cuja origem está na aplicação. Como exemplo, tem-se o relatório "Ameaças de Segurança na *Internet* da Symantec" (SYMANTEC..., 2006), no qual se deflagrou que falhas em aplicações *Web* foram responsáveis por 69% dos ataques no segundo semestre de 2005.

Como a área da segurança da informação mostra-se muito ampla e, por vezes, permite interpretações diversas, faz-se necessário a localização do contexto da segurança considerado nesta tese. No atendimento a esta demanda, este capítulo discorre sobre os conceitos e a realidade da segurança de *software*. Sua contribuição está no nivelamento conceitual sobre *software* seguro e vai ao encontro de se proporcionar uma referência em língua portuguesa sobre esse tema.

2.1 Conceitos Básicos

Antes de adentrar ao conteúdo de segurança de *software* propriamente dito, se faz necessário considerar alguns conceitos-chave definidos pela segurança da informação, área

cuja própria definição carece de esclarecimento. Essa realidade tem seduzido pesquisadores na tentativa de estabelecer uma definição que consiga incorporar suas dimensões. Segundo Peltier, Peltier e Blackley (2000), esta dificuldade seria comparada à definição de infinito, devido à abrangência dessa área.

O mesmo Peltier (2001, p. 25) definiu:

> "A segurança da informação engloba o uso de controles de acesso lógico e físico para garantir o uso apropriado da informação e para proibir modificação não autorizada ou acidental, destruição, revelação, perda ou acesso a registros manuais ou automatizados e arquivos, bem como a perda, o dano ou o uso indevido dos ativos de informação".

A citação anterior evidencia uma atitude comum observada pela comunidade quando demandada a definir segurança da informação. O resultado tende a explicar não necessariamente o que ela *é*, mas sim o que ela *faz*, motivação para a pesquisa de uma definição genuína por Anderson (2003, p. 310), cuja proposta está documentada em seu artigo: *Senso bem fundamentado da garantia que os riscos à informação e respectivos controles estão em equilíbrio*.

Apesar de Anderson (2003) empregar uma definição de fato à expressão-alvo, ela é vista ainda como incompleta por Marciano (2006), uma vez que considera, no que tange ao risco, questões determinantes, como: percepção dos usuários, objetivos da organização, estratégias adequadas para a formulação, aplicação e verificação das políticas e para a sua atualização. O último finaliza com sua própria definição:

> "Segurança da informação é um fenômeno social no qual os usuários (aí incluídos os gestores) dos sistemas de informação têm razoável conhecimento acerca do uso destes sistemas, incluindo os ônus decorrentes expressos por meio de regras, bem como sobre os papéis que devem desempenhar no exercício deste uso" (MARCIANO, 2006, p.110).

Seguem da definição da segurança da informação as propriedades a serem resguardadas

pelos sistemas. Tais propriedades, de uma maneira geral, são classificadas comumente dentre os três primeiros itens a seguir (PFLEEGER, 1997; BISHOP, 2003), adicionadas das duas últimas, de acordo com outras abordagens (HOO, 2000; MARTIMIANO, 2006):

- *confidencialidade* - refere-se à ocultação de uma informação ou recursos. Em alguns casos, sua aplicação impõe a necessidade de ocultar inclusive a existência da informação ou recurso;

- *integridade* - refere-se à confiança sobre o dado e recursos e é expressa em termos de prevenir mudanças impróprias ou não autorizadas;

- *disponibilidade* - refere-se à habilidade de usar a informação ou o recurso autorizado desejado;

- *autenticidade* - refere-se à validade, conformidade e legitimidade da informação;

- *irretratabilidade* - refere-se à capacidade de negar a execução de determinada ação no sistema, também conhecida como não-repúdio.

2.1.1 Vulnerabilidades

Segundo Gollmann (2005), as vulnerabilidades de segurança são "pontos falhos do sistema que podem ser acidentalmente ou intencionalmente explorados de forma a provocar danos em seus ativos". As vulnerabilidades podem existir na rede, no sistema operacional ou no nível de aplicação e nas práticas operacionais.

As vulnerabilidades de *software* podem ser categorizadas segundo diversas dimensões (WEBER; KARGER; PARADKAR, 2005). A que mais interessa no contexto desse trabalho refere-se ao momento de sua origem no ciclo de desenvolvimento. Segundo esta abordagem, pode-se dividi-las em duas principais categorias: vulnerabilidades de implementação e de projeto[1]. A primeira acontece em tempo de codificação, ao se inserir uma falha no código por imperícia (ex.: uso de *Application Programming Interface* (API)s banidas) ou arbitrariamente (ex.: *backdoors*). Sua influência é local, o que facilita a correção. A segunda se refere a problemas originados na etapa de projeto do *software*, resultantes da interpretação, concepção e/ou modelagem inadequadas dos mecanismos para atender aos requisitos de segurança. Sua influência no perfil de segurança do *software* é global, portanto, pode causar seu comprometimento por completo, inviabilizando sua correção (WYSOPAL et al., 2007).

2.1.2 Ameaças, Ataques, Atacantes e Risco

A acepção mais aceita considera ameaça como sendo as possíveis ações que um atacante possa realizar no sistema para alcançar seu objetivo (SWIDERSKI; SNYDER, 2004). Cumpre observar que alguns autores, como, por exemplo Hope, Lavenhar e Peterson (2005) e Ashbaugh (2006) na Figura 2, usam o termo ameaça para a definição do atacante.

Bishop (2003) relaciona ameaças, ataques e atacantes da seguinte maneira:

> "Ameaça é uma violação potencial à segurança. O fato de que a violação possa acontecer significa que tais ações que a habilitam devem ser bloqueadas. Tais ações são denominadas ataques. Aqueles que executam tais ações ou provocam a sua execução são conhecidos como atacante".

Cumpre ressaltar que a existência da ameaça está atrelada à possibilidade da violação. Já o ataque refere-se à execução da violação.

[1] É comum encontrar na literatura a referência para vulnerabilidades de projeto e implementação respectivamente como furo de segurança (*flaw*) e erro (*bug*) (MCGRAW, 2006; WYSOPAL et al., 2007)

Ataques são ações maliciosas ou inadvertidas que têm o potencial de degradar as propriedades de segurança dos ativos. As classes mais representativas de ataques, segundo Shirey (2000), são: revelação, fraude, interrupção e usurpação, cujos detalhes podem ser observados na Tabela 1.

Tabela 1: Ameaças e Suas Realizações, modificado de (SHIREY, 2000)

Ameaça	Realização
Revelação (Não autorizada) circunstância ou evento pelo qual uma entidade obtém acesso a dados que ela não está autorizada.	**Exposição:** ação pela qual dados sensíveis são liberados diretamente a uma entidade não autorizada.
	Interceptação: ação pela qual uma entidade não autorizada acessa diretamente dados sensíveis trafegando entre origens e destinos autorizados.
	Inferência: ação pela qual uma entidade não autorizada acessa indiretamente dados sensíveis pelo raciocínio a partir das características ou subprodutos da comunicação.
	Intrusão: ação pela qual uma entidade não autorizada obtém acesso a dados sensíveis burlando as proteções de segurança do sistema.
Fraude evento ou circunstância que possa resultar na recepção de dados falsos que a entidade autorizada confia como legítimo	**Simulação:** ação pela qual uma entidade não autorizada consegue acesso ao sistema ou realiza ato malicioso se passando por uma entidade autorizada.
	Falsificação: ação pela qual dados falsos iludem uma entidade autorizada.
	Repúdio: ação pela qual uma entidade ilude outra negando falsamente sua responsabilidade por um ato.
Interrupção evento ou circunstância que interrompa ou previna a operação correta dos serviços ou funções do sistema.	**Incapacitação:** ação que previna ou interrompa a operação do sistema pela desativação de um componente do sistema.
	Corrupção: ação que altera de maneira não desejada a operação do sistema pela modificação maliciosa de suas funções ou dados.
	Obstrução: ação que interrompe a entrega de serviços do sistema pela ocultação de suas operações.
Usurpação evento ou circunstância que resulte no controle dos serviços ou funções do sistema por uma entidade não autorizada.	**Apropriação indébita:** ação pela qual uma entidade assume o controle lógico ou físico de um recurso do sistema.
	Abuso: ação que leve um componente a realizar uma função ou serviço do sistema em detrimento de sua segurança.

A segurança de *software* está submetida a duas questões determinantes, uma delas se refere à impossibilidade de se obter um *software* completamente seguro. Por maior que seja o esforço, sempre haverá maneiras de se contornar seus mecanismos de segurança. Outra questão está associada à degradação causada em outros requisitos, como, por exemplo, desempenho e usabilidade, quando se privilegia a segurança (CRANOR; GARFINKEL, 2005). Esta realidade, adicionada ao fato de que nenhum sistema goza de prazo e orçamento ilimitado para seu

desenvolvimento, exige uma justificativa da real necessidade da segurança.

O risco tem sido amplamente usado como meio de argumentação para justificar e dosar as iniciativas de segurança. O risco, segundo Gollmann (2005), é uma função que relaciona os ativos, ameaças e vulnerabilidades. Conforme descrito em (STONEBURNER; GOGUEN; FERINGA, 2002), o risco corresponde ao retorno negativo estimado causado pelo exercício de alguma vulnerabilidade, considerando tanto a probabilidade quanto o impacto da ocorrência. A análise de risco, no que tange à segurança de *software*, parte das informações associadas a ameaças, com a definição da probabilidade e impacto para determinar o seu fator de risco.

A avaliação de risco pode seguir uma orientação quantitativa ou qualitativa. Como exemplo da primeira, foi apresentado por Hoo (2000) o *framework Annual Loss Expectancy* (ALE) de avaliação de risco que considera a incerteza, permitindo com flexibilidade a variação de níveis de detalhes de modelagem, concentrando-se o foco diretamente nas decisões gerenciais e reconhecendo a importância das estatísticas de segurança. Já Ashbaugh (2006) apresenta um modelo de adoção mais facilitada pelo ciclo de desenvolvimento de *software*, dada a sua simplicidade, pois está baseado na identificação do fator de risco relativo dentre os vários riscos encontrados.

Como apresentado pela Figura 2, o risco é determinado pela probabilidade de dano ao ativo, categoria que inclui a informação. O ativo é protegido por controles, também conhecidos como mecanismos, que reduzem ou eliminam o risco, além de neutralizarem o elemento habilitador da ameaça, conhecido como vulnerabilidade.

Uma vez identificado o risco, as seguintes estratégias podem ser usadas para sua mitigação, conforme descrito em (STONEBURNER; GOGUEN; FERINGA, 2002):

- aceitação, consiste na aceitação do risco e continuação da operação ou na implementação

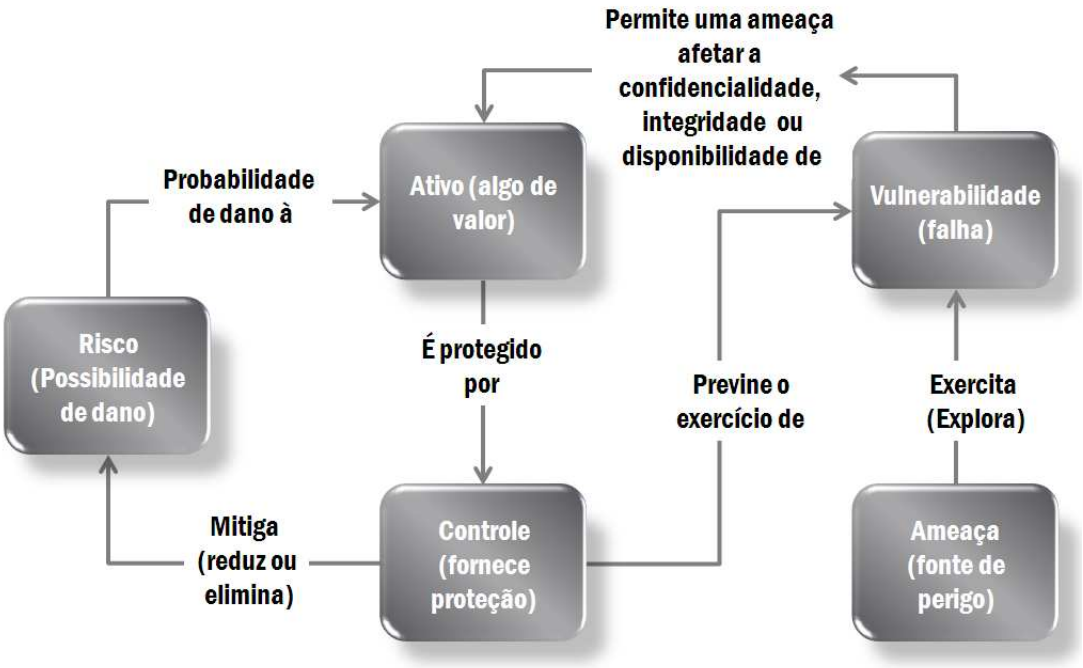

Figura 2: Modelo de Análise de Risco de Segurança, modificado de (ASHBAUGH, 2006)

de controles para a redução do risco a um nível aceitável;

- eliminação, evita o risco pelo controle de sua causa e/ou consequência;

- limitação, consiste na diminuição do campo de ação envolvido no risco pela redução do impacto adverso resultante do exercício de uma vulnerabilidade;

- transferência, repassar os efeitos dos riscos de maneira a não haver prejuízos, como, por exemplo, a aquisição de seguro.

2.1.3 Mecanismos

Os mecanismos de segurança, também conhecidos como contramedidas, são os recursos para impor as necessidades de segurança estabelecidas para um determinado sistema. Tais recursos podem ser técnicos, como o mecanismo de controle de acesso de uma aplicação

Web baseada em papéis, ou não técnicos, como a solicitação de documento com assinatura reconhecida em cartório, para que o *e-mail* de contato registrado na conta do usuário seja modificado[2].

Conforme (BISHOP, 2003, p. 12), um mecanismo de segurança pode se caracterizar como preciso, seguro ou vago, sendo que o mecanismo *preciso* limita o sistema a operar somente nos estados seguros, o *seguro* permite que o sistema opere em apenas alguns estados seguros e o *vago* permite que o sistema opere em estados seguros e inseguros. Embora a maioria dos mecanismos de segurança sejam considerados como vagos, as falhas estão geralmente associadas com o seu uso inadequado, e não à sua eficácia (JüRJENS, 2002).

2.2 Segurança de *Software*

A segurança de *software* corresponde à habilidade do *software* resistir, tolerar e se recuperar de eventos que intencionalmente ameacem sua dependabilidade (ALLEN et al., 2008). Para McGraw (2004), a segurança de software refere-se à "construção de *software* seguro: projetá-lo para que seja seguro, se certificar da sua segurança e educar desenvolvedores, arquitetos e usuários sobre como construir produtos seguros.
"

Observa-se ainda uma interpretação errônea sobre o conceito de segurança de *software*, que em muitos casos é vista simplesmente como mecanismos computacionais a serem usados, como, por exemplo, criptografia (KONHEIM, 2007) e bibliotecas de funções seguras (HOWARD; LEBLANC, 2002). Entretanto, isso corresponde a *software* de segurança. A segurança de *software* busca chegar a um *software* capaz de manter suas características de operação frente à malícia, erro ou evento inesperado (CHESS; WEST, 2007, pg. 4).

[2]Situações como esta são comuns em casos em que o usuário deixa de usar uma conta por período tão longo que o *e-mail* fornecido para contato, no momento do cadastro, não existe mais, impedindo que informações sobre as credenciais possam ser enviadas para este endereço.

Para cumprir este objetivo, a estratégia mais defendida é considerar a segurança desde a concepção do *software*, com ações integradas ao longo de todo o seu ciclo de desenvolvimento (DEVANBU; STUBBLEBINE, 2000; WILLIAMS; MACDONALD, 2006).

2.2.1 Fases do Desenvolvimento e a Segurança de *Software*

Esta seção apresenta uma visão geral das ações correntes para tratar da segurança de *software* ao longo de seu ciclo de desenvolvimento.

2.2.1.1 Requisitos de Segurança de *Software*

As políticas de segurança de alto nível ou institucionais são diretivas de segurança genéricas ou orientações com as quais quaisquer serviços, práticas ou produtos da organização devem estar em conformidade (BARMAN, 2001). Elas são definidas de acordo com os direcionadores de negócio, incluindo a visão institucional, objetivos de mercado e regulação e padrões relacionados, como, por exemplo, o *Payment Card Industry* (PCI) (PAYMENT..., 2006). Em um mundo ideal, toda organização deveria ter tais políticas bem definidas e comunicadas com clareza dentre os seus colaboradores. De fato, sem elas, torna-se impossível identificar o quão seguro os sistemas de uma organização devem ser para atender às suas necessidades e qual o esforço e recurso a ser investido no provimento de tal segurança.

Os requisitos de segurança do *software* necessitam absorver as necessidades de segurança da organização estabelecidas em suas políticas de alto nível, sob pena de comprometer a sustentabilidade do negócio associado. Observa-se como ponto-chave na especificação dos requisitos de segurança o reconhecimento das ameaças a que o software está submetido, considerando, consequentemente, as políticas de alto nível para a determinação da pertinência

de cada uma das ameaças levantadas (TØNDEL; JAATUN; MELAND, 2008).

Os usuários podem não estar totalmente conscientes dos riscos de segurança, riscos para a missão e vulnerabilidades associadas ao seu sistema. Para se definir requisitos, analistas podem, em conjunto com os usuários, realizar análises ascendentes (*bottom-up*) e descendentes (*top-down*) acerca das possíveis falhas de segurança que possam causar risco para a organização, bem como definir requisitos que controlem vulnerabilidades.

A análise para segurança, conhecida como árvore de faltas (ZHANG et al., 2005) (também conhecida como análise de árvore de ameaças ou de ataques), é uma abordagem descendente para identificar vulnerabilidades. Em uma árvore de faltas, o objetivo do atacante é colocado no topo da árvore. Então, o analista documenta as possíveis alternativas para alcançar tal objetivo. Para cada alternativa, o analista pode adicionar recursivamente alternativas precursoras para alcançar subobjetivos que compõem o objetivo principal. Este processo é repetido para cada objetivo do atacante. Pela análise dos nós de mais baixo nível da árvore de ataque resultante, o analista pode identificar todas as técnicas possíveis para violar a segurança do sistema; as prevenções dessas técnicas podem ser especificadas como requisitos de segurança do sistema.

A análise *Failure Modes and Effects Analysis* (FMEA) é uma abordagem para analisar possíveis falhas de segurança. As consequências de todas as falhas simultâneas de todos os mecanismos de segurança existentes ou planejados são documentadas e o impacto de cada falha na missão do sistema e nos seus interessados é identificado. Outras técnicas para desenvolvimento de requisitos de segurança de sistema incluem: modelagem de ameaças (SWIDERSKI; SNYDER, 2004), casos de mal uso e abuso (HOPE; MCGRAW; ANTON, 2004) e ações de abuso (FERNANDEZ et al., 2006b). Os requisitos podem ainda ser derivados dos modelos de política de segurança de sistema e alvos de segurança que descrevem os mecanismos de segurança requeridos pelo sistema, como, por exemplo, as descrições de alvos de avaliação *Target of Evaluation* (ToE) produzidos pelas avaliações do *Common Criteria* (CC).

A análise de árvore de ataque e FMEAs aumentam a complexidade dos requisitos de segurança derivados de modelos de ameaça, modelos de política de segurança e/ou alvos de segurança. O resultado da análise dos requisitos de segurança pode ser usado como base para cenários de caso de testes a serem usados durante a integração ou teste de aceitação.

2.2.1.2 Projeto

O projeto de *software* é a atividade de engenharia de *software* que usa os produtos gerados a partir da análise de requisitos, particularmente a especificação do comportamento externo do sistema, e decompõe o sistema nos seus níveis múltiplos em componentes e suas interações. O resultado do projeto compreende:

- A descrição da estrutura interna do sistema que subsidiará a garantia de adequação do projeto com sua especificação e a codificação do sistema.

- As restrições quanto ao projeto e perspectivas de sua evolução.

- Razão pelas decisões de projeto.

- Parte da declaração de garantia referente ao projeto, incluindo a argumentação e evidência sobre a conformidade do projeto com as especificações do comportamento externo, com a política de segurança do sistema e outras restrições relevantes.

O projeto de *software* seguro é similar a qualquer projeto de *software* de confiabilidade extrema, com algumas preocupações adicionais:

- Garantir consistência com a política de segurança.

- Incorporar funcionalidades necessárias de segurança.

- Garantir que o projeto atende às determinações da especificação e nada mais.

- Aplicar a separação de mecanismos, papéis e privilégios.

- Impedir a subversão ou malícia dos projetistas ou outro pessoal - infiltrados.

- Defender contra-ataques externos à infraestrutura e artefatos - externos.

- Criação de uma declaração de garantia com argumentos e evidências que a comprove e contemple o perigo de ações perspicazes e maliciosas, bem como os infortúnios aleatórios relevantes de segurança.

Alguns princípios de projeto de *software* seguro, bem como sua aplicação, devem ser difundidos entre os desenvolvedores. Faz-se necessário também que os desenvolvedores tenham *expertise* no reconhecimento se os princípios devem ou não ser usados no projeto e como avaliar o projeto em questão quanto a mudanças propostas, incluindo-se melhorias. Apresenta-se a seguir um resumo dos princípios gerais de projeto de *software* elaborado por Redwine (2007):

- Simplificar e minimizar o núcleo confiável.

- Excluir funcionalidades de segurança não relevantes.

- Separar política de mecanismo de imposição.

- Restringir dependências.

- Manter a menor quantidade de estados do *software* armazenada.

- Virtualizar papéis de *hardware* e *drivers* de dispositivos.

- Reduzir o suporte a funcionalidades que estejam além da fronteira dos componentes confiáveis.

- Reduzir possibilidade de violações de segurança.

- Negar acesso, a não ser que explicitamente autorizado.

- Implantar com segurança inicial por padrão.

- Verificar cada acesso.

- Implementar o menor privilégio.

- Evitar o uso de mecanismos em comum.

- Segregar funções.

- Segregar privilégios.

- Segregar papéis.

- Segregar domínios de confiança.

- Isolar os sistemas de acesso público dos recursos de missão crítica (ex.: dados, processos).

- Isolar a conexão física entre os recursos de acesso público e a informação crítica da organização.

- Estabelecer camadas de serviços e mecanismos entre os sistemas públicos e os sistemas seguros responsáveis pela proteção dos recursos de missão crítica.

- Usar mecanismos de fronteira ou guardiões para separar os sistema computacionais e infraestrutura de redes para controlar o fluxo de informação e acesso pelas fronteiras da rede e impor a separação apropriada de grupos de usuários.

- Reduzir o número de pontos de entrada e saída (superfície de ataque).

- Não implementar funções desnecessárias.

A consolidação das alternativas de projeto para um sistema resulta na sua arquitetura, cuja definição deve ser amparada no conjunto de recomendações listadas anteriormente. O resultado final esperado a partir do uso das contribuições apresentadas por esta tese está na definição de uma arquitetura de *software* mais adequada às suas necessidades de segurança. Por esta razão, o Capítulo 3 detalha esse tema.

Mostra-se muito útil para o alcance dos objetivos da fase de projeto o uso de padrões, assunto que, devido à sua abrangência e relevância para esta tese de doutorado, encontra-se detalhado no Capítulo 4.

2.2.1.3 Codificação

As fases do ciclo de desenvolvimento de *software* mencionadas anteriormente objetivam reduzir as chances da concepção do *software* carregar falhas de segurança, porém, é na fase de codificação que ocorre a materialização da especificação e projeto, gerando o resultado primordial para qualquer *software*: seu código.

Apesar de haver instrumentos para avaliação dos artefatos gerados pelas fases anteriores, só se torna possível uma verificação quanto às vulnerabilidades do *software* a partir da disponibilização de código.

As intervenções viáveis para garantia da segurança na fase de codificação podem ser resumidas como se segue:

- Desenvolvimento da consciência dos desenvolvedores sobre as alternativas para codificação segura, incluindo-se os recursos de segurança mais adequados de acordo com a plataforma usada para desenvolvimento, bem como suas limitações em termos de segurança, ilustrado pela coleção de ataques que existe disponível (HOWARD; LEBLANC,

2002; GRAFF; WYK, 2003; HOGLUND; MCGRAW, 2004).

- Definição de normas de codificação por plataforma, de forma a obrigar ao desenvolvedor a explorar corretamente, em termos de segurança, seus recursos. Cumpre ressaltar que o valor de qualquer norma é proporcional à medida que ela é usada na prática. Uma norma de codificação só será efetiva se houver mecanismos de imposição automáticos que impeçam ao desenvolvedor de seguir outro caminho diferente do especificado pela norma. A alternativa mais usada hoje para este fim é a análise estática de código fonte, pela qual as normas de codificação são verificadas e os desvios, bloqueados, impedindo a contaminação do código fonte. Mais detalhes na Seção 2.2.1.4.

Qualquer que seja o cenário de desenvolvimento de *software* seguro, treinamento e educação profunda dos desenvolvedores se mostram determinantes, de forma a assimilarem as técnicas, ferramentas e a razão para sua aplicação (BISHOP; FRINCKE, 2005b; SHUMBA et al., 2006).

2.2.1.4 Testes

Existem basicamente três abordagens de teste para revelar vulnerabilidades: testes caixa preta, caixa branca e caixa cinza (WYSOPAL et al., 2007). A diferença entre os três pode ser analisada sob o aspecto do recurso disponível para o testador. Em um extremo, encontra-se o teste caixa branca, cuja abordagem requer acesso completo a todos os recursos, incluindo código fonte, modelos e até entrevista com o programador. No outro extremo está o teste caixa preta, no qual os recursos internos do sistema não são disponibilizados. Já os testes caixa cinza não possuem uma definição bem estabelecida. Presume-se que, para esse tipo, o testador tenha acesso às bibliotecas compiladas e, em alguns casos, até à documentação do *software*.

Teste caixa branca

Destaca-se na abordagem de teste caixa branca a análise de código fonte na tentativa de localizar falhas de segurança. Tal análise pode ser realizada manualmente (HOWARD, 2003; HOWARD, 2006), impraticável em larga escala, e automatizada a partir do uso de ferramentas de análise de código fonte (CHESS; WEST, 2007). Um dos pontos fortes desta abordagem refere-se à cobertura, uma vez que todos os fluxos do código fonte podem ser auditados, o que pode acarretar inúmeros falso positivos[3]. A análise dos resultados gerados pelas ferramentas torna-se complexa, pois se exige uma revisão criteriosa para verificar se o registro realmente corresponde a uma ameaça, de acordo com o contexto da organização. Cumpre ressaltar que nem sempre o código fonte do *software* está disponível para gozar dessa abordagem.

Teste caixa preta

O teste caixa preta se caracteriza pelo desconhecimento por parte do testador dos elementos internos do *software*, como, por exemplo, sua arquitetura e documentação associada. O encarregado de realizar tais testes possui acesso aos mesmos recursos disponíveis para um usuário qualquer do sistema. Esta abordagem também é conhecida como teste de penetração (ARKIN; STENDER; MCGRAW, 2005). Assim como para o teste caixa branca, este pode ser realizado manualmente e com o apoio de ferramentas. Destaca-se para esta última o uso bem sucedido da técnica *fuzzing* (GALLAGHER; JEFFRIES; LANDAUER, 2006; SUTTON; GREENE; AMINI, 2007). Se, por um lado, a aplicação do teste caixa preta exige apenas o sistema em operação, de outro, aspectos como cobertura e perfil técnico do testador representam fatores que dificultam sua adoção.

Teste caixa cinza

O teste caixa cinza corresponde a uma variação das duas abordagens de teste apresentadas

[3] A expressão falso positivo é usada para qualificar uma vulnerabilidade encontrada que não representa de fato uma ameaça para o sistema em análise. Fazendo uso de um vocabulário mais coloquial, esta expressão teria o mesmo significado de "alarme falso".

anteriormente. Em muitos casos, o ativo elementar usado na análise é o binário[4], assim como para as outras abordagens, esta é realizada manualmente ou com o uso de ferramenta. Entretanto, aspectos relativos à desconstrução[5] do *software* dependem de ferramentas, já a análise, não. Sua aplicação se torna facilitada, pois os binários estão sempre presentes, com exceção de aplicações remotas. Sua aplicação pode incrementar o índice de cobertura da abordagem caixa preta. Destaca-se a complexidade relativa à engenharia reversa para extrair do binário informações que revelem detalhes de sua construção.

2.2.2 Modelos, Normas, Padrões e Recomendações para *Software* Seguro

2.2.2.1 Microsoft *Security Development Lifecycle*

A partir de 2002, a Microsoft deu início ao programa *Trustworth Computing*, no qual a segurança de *software* passou a estar dentre os seus principais objetivos de negócio. Tal iniciativa se baseia no pressuposto de que a melhoria da segurança de *software* depende da implementação de processo repetível que proporcione melhoria de segurança mensurável e confiável (HOWARD; LIPNER, 2003). Em termos práticos, esta iniciativa resultou no processo *Security Development Lifecycle* (SDL), que se fundamenta em educação dos desenvolvedores, análise, revisão e construção de modelos de ameaças e realização de mudanças no projeto do *software*, bem como ferramental de apoio com vistas a obter um resultado mais seguro (HOWARD, 2004; SWIDERSKI; SNYDER, 2004).

A Figura 3 mostra as fases do SDL em alto nível, bem como suas atividades de segurança mapeadas ao ciclo de vida de desenvolvimento de *software*, como descrito a seguir:

- Definição das características de segurança durante a fase de requisitos.

[4]Binário refere-se ao resultado da compilação do *software* e sua avaliação é conhecida, também, como auditoria de binário.

[5]Desconstrução corresponde à tradução do termo *disassembly* em inglês (Babylon Translator, 2007)

- Projeto da arquitetura segura, melhores práticas e modelagem de ameaças durante do projeto.

- Análise estática de código fonte a partir de ferramentas durante a implementação.

- Aplicação de padrões de codificação e testes durante a implementação.

- Revisão de código fonte durante a implementação.

- Teste de segurança envolvendo *fuzzing* e penetração na fase de verificação.

- Serviços e resposta de segurança durante a operação (após lançamento) (HOWARD; LIPNER, 2006).

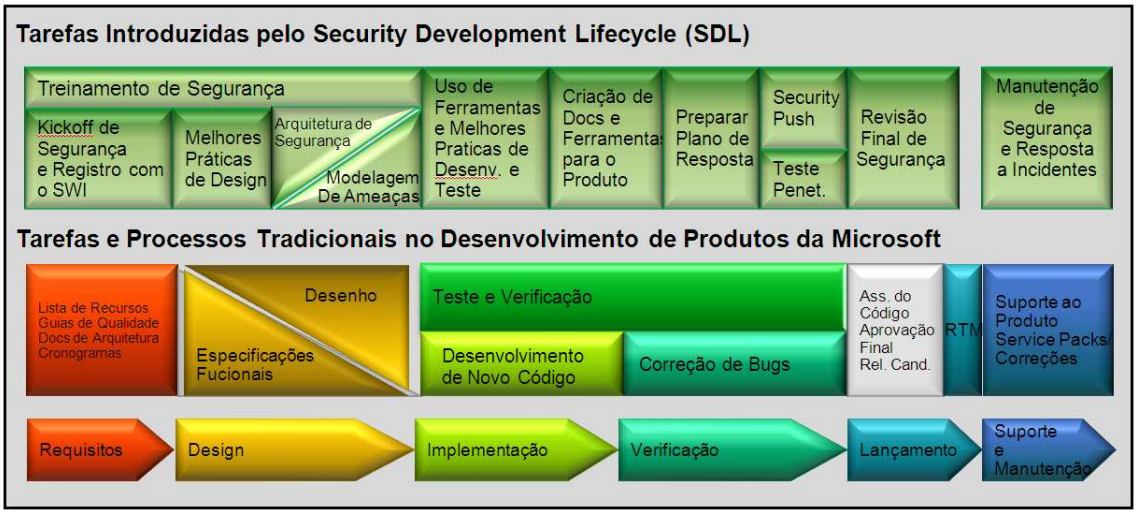

Figura 3: Microsoft *Security Development Lifecycle*, modificado de (HOWARD; LIPNER, 2006)

2.2.2.2 *Comprehensive, Lightweight Application Security Process*

Na mesma linha, o *Comprehensive, Lightweight Application Security Process* (CLASP), processo originalmente desenvolvido pela Secure *Software*[6], hoje sob os cuidados da OWASP[7], busca a melhoria da segurança pela ênfase em papéis e responsabilidades a partir de um processo leve, habilitando organizações e projetos de qualquer porte a usá-lo (GRAHAM, 2006).

O processo CLASP é composto por um conjunto de atividades de segurança organizadas de acordo com as seguintes melhores práticas:

1. Instituir um programa de conscientização.

2. Realizar avaliação da aplicação.

3. Capturar requisitos de segurança.

4. Implementar práticas de desenvolvimento seguro.

5. Estabelecer procedimentos de remediação de vulnerabilidades.

6. Definir e monitorar métricas.

7. Publicar orientações de segurança operacionais.

[6]Empresa adquirida pela Fortify *Software* Inc. www.fortifysoftware.com
[7]www.owasp.org

Dois pontos do CLASP merecem destaque: a orientação para o desenvolvimento de *software* seguro desde sua concepção, bem como para o aspecto da manutenção segura de *software*. Outro ponto está na ênfase em melhoria e maturidade de processo de *software*, conforme evidenciado pela Figura 4. Nela, a área da curva corresponde aos custos para obtenção de tal maturidade relativos a treinamento, ferramentas e implementação de atividades.

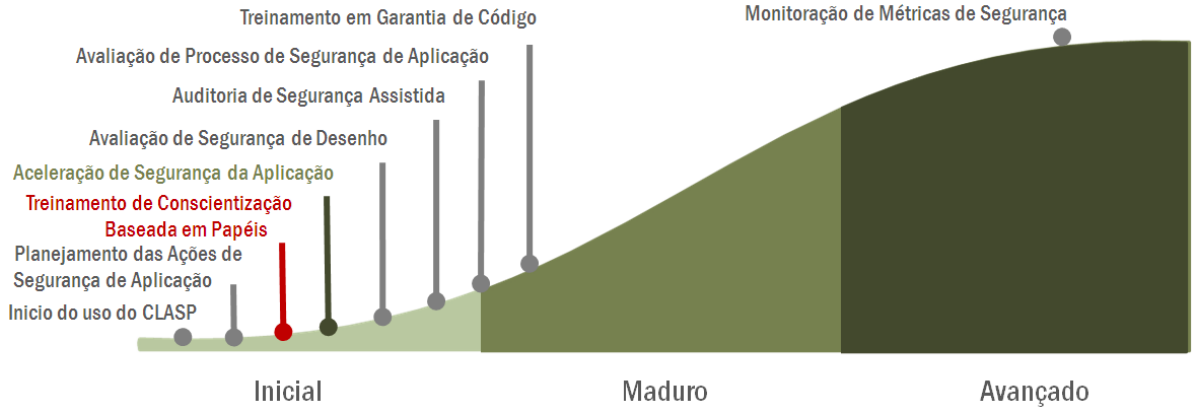

Figura 4: Curva de Maturidade do *Comprehensive, Lightweight Application Security Process*, modificado de (MORANA, 2008)

Convém registrar que a pesquisa desenvolvida por Gregoire et al. (2007) propõe um processo resultante da consolidação e complementação das deficiências encontradas na análise crítica dos processos SDL e CLASP.

2.2.2.3 *Building Security In*

McGraw (2006) sugere uma abordagem para desenvolvimento de *software* seguro conhecida como *Building Security In* (BSI)[8], fundamentada em três pilares, gestão de riscos, sete pontos de controle ao longo do ciclo de desenvolvimento (requisitos de segurança de software; casos de abuso; revisão de código fonte com ferramenta; análise de risco de arquitetura; teste de penetração; testes de segurança baseados em risco e operação com segurança), além da especificação de uma base do conhecimento para suportar as decisões e formação continuada

[8] A expressão *Building Security In*, precedida originalmente por *Software Security*, pode ser compreendida como a aplicação de princípios de desenvolvimento de *software* desde o princípio de seu ciclo.

em segurança.

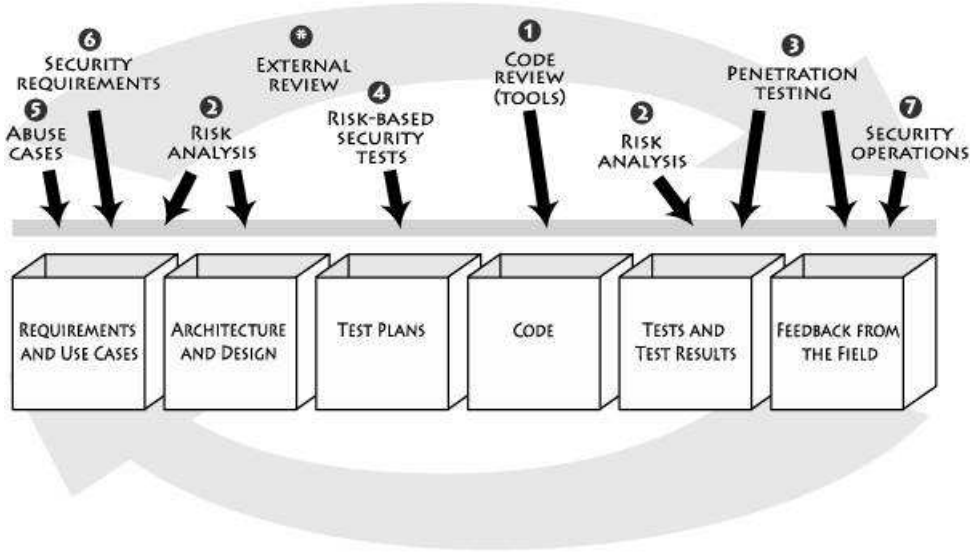

Figura 5: Distribuição dos Pontos de Controle do *Building Security In* pelo Ciclo de Desenvolvimento de *Software*, (MCGRAW, 2006)

Os pontos de controle podem ser aplicados em diferentes fases do ciclo de desenvolvimento de *software* e são associados aos artefatos, ao invés das fases do ciclo propriamente ditas. O que facilita sua adoção tanto por abordagens lineares, como, por exemplo, o ciclo de desenvolvimento em cascata, bem como em evolucionárias, como por exemplo o RUP[9]. Ainda assim, a proposta sugere uma ordem de implementação dos pontos de controle, classificada pelo esforço e impacto associado à mudança, conforme apresentado pela Figura 5. Observa-se a existência de um ponto de controle adicional marcado com asterisco. Ele corresponde a um ponto bônus referente à avaliação externa, que eventualmente pode ser usado de acordo com as possibilidades da organização, bem como as necessidades do projeto de desenvolvimento de *software* em questão.

[9]O RUP é um processo de desenvolvimento de *software* desenvolvido pela Rational, hoje *International Business Machines* (IBM).

2.2.2.4 Extensões ao *Rational Unified Process*

Alguns grupos aproveitaram-se da solidez e difusão do RUP e propuseram recursos adicionais para desenvolvimento de *software* seguro. Esses recursos foram consolidados em forma de uma extensão ao RUP. Jaferian et al. (2005) propôs uma adaptação das disciplinas "modelagem de negócio" e "requisitos e análise" (KRUCHTEN, 2003), com a modificação e adição de tarefas, artefatos e papéis, como uma alternativa para considerar a segurança de *software* desde o princípio. Já Paes e Hirata (2007) criou uma disciplina específica para segurança que incorpora atividades relacionadas a requisitos de segurança, bem como análise de ameaças, refinamento da arquitetura e elaboração de testes de segurança. Ambas incorporam em sua proposta a abordagem de requisitos de segurança "Casos de Abuso" (SINDRE; OPDAHL, 2005).

2.2.2.5 Recomendações do *Gartner Group*

Baseado no pressuposto de que até 2008 a segurança da aplicação se tornaria um critério de avaliação com peso semelhante à funcionalidade do sistema, o Gartner Group, representado por Williams e MacDonald (2006), apresentou um relatório no formato de recomendação de melhores práticas a serem consideradas ao longo do ciclo de desenvolvimento de *software*. Uma síntese dessas recomendações pode ser observada na Figura 6.

2.2.2.6 *Security System Engineering Capability Maturity Model*

Para dar uma orientação normativa, o Systems... (2003) desenvolveu o *Security System Engineering Capability Maturity Model* (SSE-CMM), que veio a se transformar na norma ISO/IEC 21827 (INFORMATION..., 2002), cujo objetivo é orientar, em termos de processo, as necessidades do ciclo de desenvolvimento de sistemas seguros. Seu fundamento está no conceito de maturidade explorado pelos modelos *Capability Maturity Model* (CMM) do *Software Engineering Institute* (SEI).

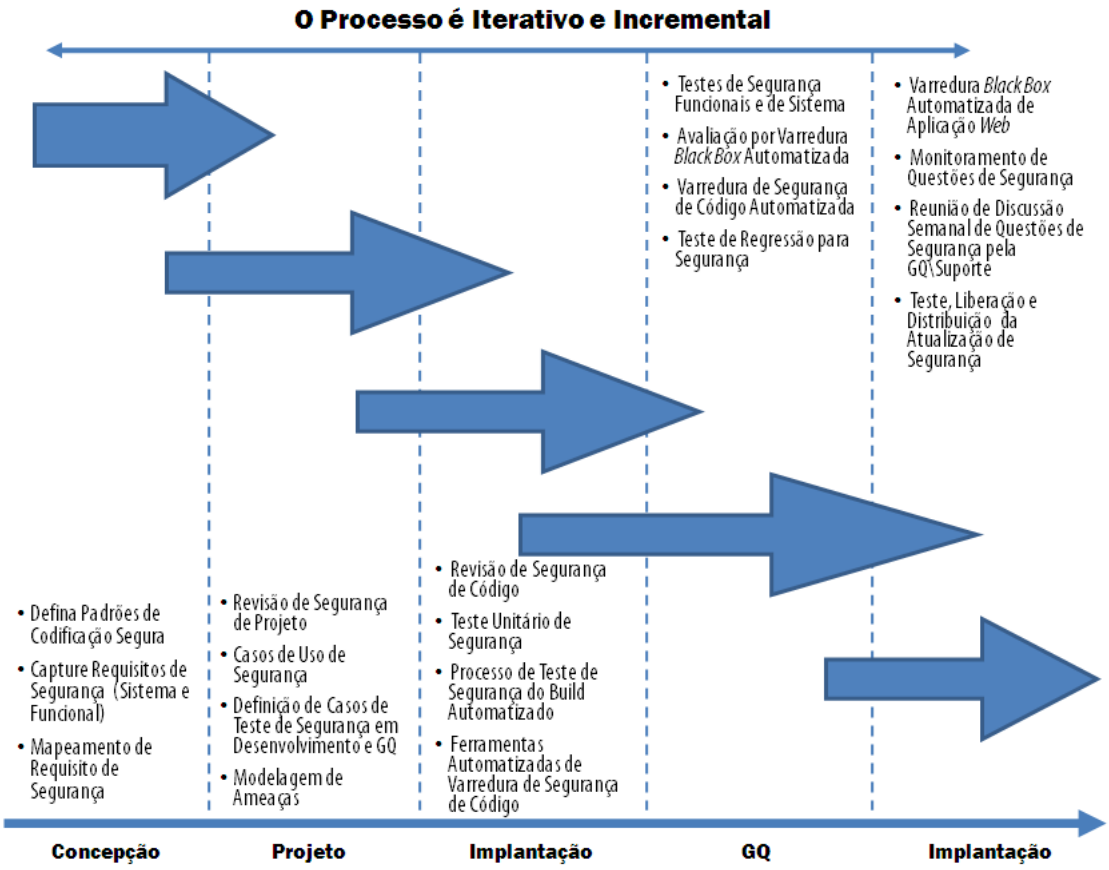

Figura 6: Melhores Práticas a Serem Incorporadas ao Ciclo de Desenvolvimento de *Software*, modificado de (WILLIAMS; MACDONALD, 2006)

O modelo SSE-CMM visa ao desenvolvimento de sistemas mais seguros a partir da articulação entre os processos de engenharia, de garantia e de riscos. Para isso, definiu-se um conjunto de Áreas de Processo (APs) específicas para tratar da segurança, que foram adicionadas às outras já contempladas pelo CMM. Estas últimas se concentram em questões gerenciais e organizacionais. Cumpre ressaltar que, assim como no CMM, uma AP é suportada por uma coleção de Práticas Base (PBs) e Práticas Genéricas (PGs) que orientam sua implantação. As PBs estão diretamente relacionadas ao objetivo da AP, enquanto as PGs dão o suporte no que tange à institucionalização das PBs. A Tabela 2 mostra um extrato do modelo SSE-CMM correspondente às APs de segurança, bem como suas respectivas PBs.

2.2.2.7 ISO/IEC 17799:2005

A ISO/IEC 17799:2005 é uma norma com o objetivo de orientar a implantação de um programa de gestão da segurança da informação (ISO/IEC..., 2005). Mesmo que ela não tenha sido elaborada com o enfoque específico no desenvolvimento de *software* seguro, encontra-se em sua Seção 12ª "Aquisição, desenvolvimento e manutenção de sistemas de informação" um conjunto de controles para este fim, distribuídos por seis categorias. Os elementos citados da norma podem ser observados na Tabela 3.

Vale ressaltar a pertinência dos controles estabelecidos que, apesar de carecerem de mais detalhes quanto à sua implantação, se mostram adequados à necessidade corrente em termos de segurança no desenvolvimento de *software*. Para exemplificar, considere os controles 12.02.1 e 12.02.4. Sua implementação efetiva pode eliminar por completo a possibilidade de diversos ataques, como, por exemplo, o estouro de *buffer*, injeção *Structured Query Language* (SQL) e o *Cross Site Script* (XSS) (HOWARD; LEBLANC, 2002; GROSSMAN et al., 2007).

Tabela 2: Áreas de Processo Diretamente Relacionadas à Segurança no *Security System Engineering Capability Maturity Model*, modificado de (SYSTEMS..., 2003)

Área de Processo	Prática Base
AP01 - Administrar Controles de Segurança	PB01.01 - Estabelecer Responsabilidades de Segurança
	PB01.02 - Gerenciar Configuração de Segurança
	PB01.03 - Gerenciar Programas de Conscientização, Treinamento e Educação em Segurança para Todos os Usuários e Administradores
	PB01.04 - Gerenciar Serviços de Segurança e Mecanismos de Controle
AP02 - Avaliar Impacto	PB02.01 - Priorizar Capacidades
	PB02.02 - Identificar Ativos de Sistema
	PB02.03 - Selecionar Métricas de Impacto
	PB02.04 - Identificar Relacionamento entre Métricas
	PB02.05 - Identificar e Caracterizar os Impactos
	PB02.06 - Monitorar os Impactos
AP03 - Avaliar Risco de Segurança	PB03.01 - Selecionar o Método da Análise de Risco
	PB03.02 - Levantar Exposição
	PB03.03 - Avaliar Risco de Exposição
	PB03.04 - Avaliar Incerteza Total
	PB03.05 - Priorizar os Riscos
	PB03.06 - Monitorar os Riscos e suas Caracteristicas
AP04 - Avaliar as Ameaças	PB04.01 - Identificar Ameaças Naturais
	PB04.02 - Identificar Ameaças Provocadas Pelo Homem
	PB04.03 - Identificar Unidades de Medida de Ameaças
	PB04.04 - Avaliar a Capacidade do Agente da Ameaça
	PB04.05 - Avaliar Probabilidade de Ocorrência da Ameaça
	PB04.06 - Monitorar suas Ameaças e Caracteristicas
AP05 - Avaliar Vulnerabilidades	PB05.01 - Selecionar o Método da Análise de Vulnerabilidades
	PB05.02 - Idetificar as Vulnerabilidades
	PB05.03 - Levantar Informações das Vulnerabilidades
	PB05.04 - Sintetizar as Vulnerabilidades do Sistema
	PB05.05 - Monintorar as Vulnerabilidades e Suas Caracteristicas
AP06 - Elaborar Argumento de Segurança	PB06.01 - Identificar Objetivos de Garantia
	PB06.02 - Definir estratégia de Garantia
	PB06.03 - Controlar Evidência de Garantia
	PB06.04 - Analisar Evidências
	PB06.05 - Divulgar Argumento de Segurança
AP07 - Coordenação de Segurança	PB07.01 - Definir Objetivos da Coordenação
	PB07.02 - Identificar Mecanismos de Coordenação
	PB07.03 - Estabelecer Coordenação Assistida
	PB07.04 - Coordenar Decisões de Segurança e Recomendações
AP08 - Monitorar Postura de Segurança	PB08.01 - Analisar os Registros de Eventos
	PB08.02 - Monitorar Mudanças
	PB08.03 - Identificar Incidentes de Segurança
	PB08.04 - Monitore as Contramedidas de Segurança
	PB08.05 - Revisar Postura de Segurança
	PB08.06 - Gerenciar resposta aos incidentes de segurança
	PB08.07 - Proteger Artefatos de Monitoramento de Segurança
AP09 - Fornecer Suprimentos de Segurança	PB09.01 - Enternder Suprimento de Segurança
	PB09.02 - Determinar Restrições e Considerações de Segurança
	PB09.03 - Identificar Alternativas de Segurança
	PB09.04 - Analisar Segurança das Alternativas de Engenharia
	PB09.05 - Fornecer Orientações de Engenharia de Segurança
	PB09.06 - Fornecer Orientações de Segurança Operacional
AP10 - Especificar necessidades de segurança	PB10.01 - Obter entendimento das necessidades dos usuários
	PB10.02 - Identificar Leis, Politicas e Restrições Aplicáveis
	PB10.03 - Identificar Contexto de Segurança do Sistema
	PB10.04 - Capturar Visão de Segurança da Operação do Sistema
	PB10.05 - Capturar as Metas de Alto Nível de Segurança
	PB10.06 - Definir Requisitos Relacionados a Segurança
	PB10.07Obter Compromisso em Segurança
AP11 - Verificar e Validar a Segurança	PB11.01 - Identificar Alvos da Verificação e Validação
	PB11.02 - Definir abordagem para verificação e validação
	PB11.03 - Realizar Verificação
	PB11.04 - Realizar Validação
	PB11.05 - Prover Resultados da Verificação e Validação

Tabela 3: Seção 12ªda ISO/IEC 17799:2005, modificado de (ISO/IEC..., 2005)

Categoria	Controle
12.1 - Requisitos de Segurança de Sistemas	12.01.1 - Análise a Especificação dos Requisitos de Segurança
12.2 - Processamento Correto das Aplicações	12.02.1 - Validação dos Dados de Entrada
	12.02.2 - Controle do Processamento Interno
	12.02.3 - Integridade das Mensagens
	12.02.4 - Validação dos Dados de Saída
12.3 - Controles Criptográficos	12.03.1 - Politicas para Uso de Controles Criptográficos
	12.03.2 - Gerenciamento de Chaves
12.4 - Segurança dos Arquivos do Sistema	12.04.1 - Controle do Sistema Operacional
	12.04.2 - Proteção dos Dados para Teste de Sistema
	12.04.3 - Controle de Acesso ao Código-fonte de Programa
12.5 - Seguraça em Processo de Desenvolvimento e de Suporte	12.05.1 - Procedimentos para controle de mudanças
	12.05.2 - Análise Técnica Crítica das Aplicações após Mudanças no Sistema Operacional
	12.05.3 - Restrições sobre Mudanças em Pacotes de *Software*
	12.05.4 - Vazamento de Informações
	12.05.5 - Desenvolvimento Terceirizado de *Software*
12.6 - Gestão de vulnerabilidades Técnicas	12.06.1 - Controle de Vulnerabilidades Técnicas

3 *Arquitetura de Software*

A arquitetura de *software* tem recebido grande ênfase pela engenharia de *software* desde a década passada, que resulta no seu amplo desenvolvimento, com diversas publicações (SHAW; CLEMENTS, 2006). Este desenvolvimento veio ao encontro da demanda da comunidade, pois o desenvolvimento de *software* guiado por arquitetura tem auxiliado na obtenção de ótimos resultados quanto ao atendimento de requisitos-chave, como confiabilidade, desempenho, portabilidade, escalabilidade, interoperabilidade etc. Mesmo com esta evolução, ainda existem inúmeros desafios e questões a serem resolvidas pela arquitetura de *software*, fazendo desta disciplina um campo fértil de pesquisas.

Como a melhoria da segurança do *software* defendida por este trabalho tem como alvo a sua arquitetura, torna-se relevante o nivelamento sobre a arquitetura de *software*, bem como sua influência no aspecto da segurança. Este Capítulo busca fazer este esclarecimento.

3.1 Definição

Embora a maioria das definições de arquitetura de *software* se fundamentem nos termos *estruturas*, *elementos* e *conexões*; cada uma apresenta um enfoque diferenciado. Isso exige um esclarecimento sobre a preocupação evidenciada por elas. Esta Seção traz algumas das definições mais difundidas, com o comentário acerca do respectivo enfoque observado.

O padrão IEEE (2000, p.3) define arquitetura de *software* como "a organização fundamental do sistema representada por seus componentes, seus relacionamentos com os outros e com o ambiente, e os princípios e diretrizes que governam seu projeto e evolução".

Esta definição destaca o processo, relacionando informações secundárias, como os princípios e as diretrizes associadas à elaboração e evolução da arquitetura de *software*. De fato, estas informações são determinantes e devem ser consideradas em tempo de definição da arquitetura de *software*, mas são secundárias, uma vez que a existência da arquitetura, bem como sua análise, independe do processo que a gerou e/ou designado para sua evolução.

Já Garlan (2000, p. 94) estabelece a seguinte definição:

> "A arquitetura de *software* descreve sua estrutura como um todo. Esta estrutura ilumina as decisões de alto nível de projeto, incluindo, por exemplo, como o sistema é composto de partes que se interagem, onde estão as principais vias de interação e quais são as propriedades-chave das partes."

A acepção Garlan (2000) remete a dois pontos comumente confundidos. O primeiro refere-se à interpretação errônea de que a arquitetura de *software* é representada por uma única estrutura, pois a multiplicidade estrutural está intimamente associada à arquitetura de *software*. Outro ponto passível de confusão ocorre no uso dos termos arquitetura de *software* e projeto para o mesmo fim. O objetivo do primeiro se encerra na medida em que os elementos e suas relações estejam definidos, não importando os detalhes em profundidade inerentes aos elementos ou aos requisitos funcionais, sendo estes objetivos do segundo.

A confusão mencionada é em parte justificada pela vacuidade das definições relacionadas à arquitetura de *software*, como, por exemplo, o primeiro princípio definido por Kazman, Bass e Klein (2006, pg. 1). Este princípio estabelece que "uma arquitetura de *software* deve ser definida em termos de elementos de maneira suficientemente rudimentar para o controle

intelectual humano e suficientemente específica para um raciocínio significativo".

Ainda, convém destacar a explicação de Booch (2006) para os conceitos de arquitetura e projeto de *software*:

> "Como nome, projeto corresponde à estrutura ou comportamento de um sistema, cuja presença resolve ou contribui para a resolução de uma força ou forças do sistema. O projeto representa então, um ponto no espaço potencial de decisão. Um projeto pode ser singular (representando uma decisão única) ou uma coleção (representando um conjunto de outras decisões).
>
> Como verbo, projetar é a atividade de tomar tais decisões. Dado um conjunto expressivo de forças, um conjunto relativo de materiais maleáveis, adicionado de um vasto campo de ação, resulta num espaço de decisão grande e complexo. Por isso se associa ao projeto uma ciência (análise empírica pode sugerir regiões adequadas ou pontos exatos no espaço de projeto), bem como uma arte (relacionado ao grau de liberdade que vai além de decisões empíricas; existem oportunidades para elegância, beleza, simplicidade, inovação e astúcia).
>
> ...
> A arquitetura refere-se às decisões representativas de projeto que molda o sistema, onde representatividade é medida pelo custo da mudança".

Por fim, expõe-se a definição de Bass, Clements e Kazman (2003, pg. 3), a seguir:

> "A arquitetura de *software* de um programa ou sistema computacional é uma estrutura ou estruturas do sistema, que compreende elementos de *software*, as propriedades externamente visíveis de tais elementos e o relacionamento entre eles".

Seguindo a base das definições anteriores, Bass, Clements e Kazman (2003) enfatiza a estrutura do *software*, suas relações, com a adição do ambiente externo, cuja influência na definição da arquitetura é determinante, uma vez que ele caracteriza os atributos de qualidade dos requisitos (KAZMAN; BASS; KLEIN, 2006).

Mesmo que a definição de arquitetura de *software* se revele como um tema de extrema relevância, não é pretensão desta tese exauri-lo. De qualquer maneira, aponta-se a definição estabelecida por Bass, Clements e Kazman (2003) como a mais alinhada a esta proposta de tese.

3.2 Os Papéis da Arquitetura de *Software*

A arquitetura de *software* é determinante para o alcance dos objetivos da engenharia de *software*, que se resumem em produto com a qualidade, duração e custo de desenvolvimento esperados. Nesse contexto, a arquitetura influencia diretamente alguns aspectos inerentes ao ciclo de vida do *software*, dentre os quais destacam-se:

- compreensão: simplifica a habilidade de compreensão de sistemas grandes, apresentando-os no nível de abstração apropriado para facilitar o entendimento do projeto em alto nível do sistema;

- comunicação: uma vez que facilita o entendimento do *software* em termos de suas estruturas principais e relações, se coloca como uma alternativa rica para difusão desse conhecimento pela equipe, no intuito de obter uma consciência coletiva sobre o sistema;

- reuso: as descrições de arquiteturas suportam o reuso em múltiplos níveis, partindo de componentes simples, componentes complexos, até *frameworks*;

- base de conhecimento: uma vez que estão representadas as decisões mais precoces de projeto do *software*, ela atua como fonte de conhecimento para apoio ao raciocínio sobre projetos futuros, bem como à formação continuada de arquitetos;

- construção: fornece uma "planta" para o desenvolvimento pela indicação dos componentes mais importantes e suas dependências;

- evolução: expõe as dimensões nas quais se espera a evolução do *software*. Adicionalmente, fornece elementos para análise sobre os desdobramentos de mudanças no

software, habilitando estimativas mais precisas de custo e esforço;

- análise: fornece novas alternativas para análise, incluindo a verificação de consistência, conformidade a restrições impostas pelo padrão de arquitetura, conformidade aos atributos de qualidade, análise de dependências etc.;

- gerenciamento: é apontada pela indústria de *software* como um fator determinante em projetos que alcançam o sucesso, sendo a análise de risco um de seus desdobramentos mais significativos (GARLAN, 2000; CLEMENTS et al., 2002a; BASS; CLEMENTS; KAZMAN, 2003; GORTON, 2006).

3.3 Projeto de Arquitetura de *Software*

A arquitetura de *software* foi privilegiada nos últimos 15 anos com diversas pesquisas, cujos resultados se traduzem em técnicas, processos, diretrizes e melhores práticas para seu projeto (PERRY; WOLF, 1992; HOFMEISTER; NORD; SONI, 1999; HOFMEISTER; NORD; SONI, 2000; CLEMENTS et al., 2002a; CLEMENTS et al., 2002b; BASS; CLEMENTS; KAZMAN, 2003).

A importância de uma arquitetura tem a duração muito próxima do ciclo de vida completo do *software*. Consequentemente, sua comunicação aos seus interessados[1] é tão importante quanto a sua própria existência. Segundo Clements et al. (2002b), "se a arquitetura não puder ser entendida, de forma que se possa construir, analisar e manter sistemas e aprender a partir dela, todo esforço despendido na sua construção terá sido em vão".

Este cenário motivou a elaboração do padrão *American National Standards Institute* (ANSI)/*Institute of Electrical and Electronics Engineers* (IEEE)-1471-2000 com o intuito

[1]Termo resultante da tradução de *stakeholders* em inglês.

de disciplinar a engenharia no que tange à documentação de arquiteturas de *software*. Seu conteúdo considera as melhores práticas para documentação de arquitetura de *software*.

A Figura 7 ilustra a relação entre os conceitos explorados pela ANSI/IEEE-1471-2000, na qual um *sistema* possui uma *arquitetura*, que é descrita por uma *descrição da arquitetura*. Uma *descrição da arquitetura* seleciona um ou mais *pontos de vista*, cada um cobrindo uma ou mais *preocupações* dos *interessados*. Um *ponto de vista* estabelece métodos para um ou mais *modelos*.

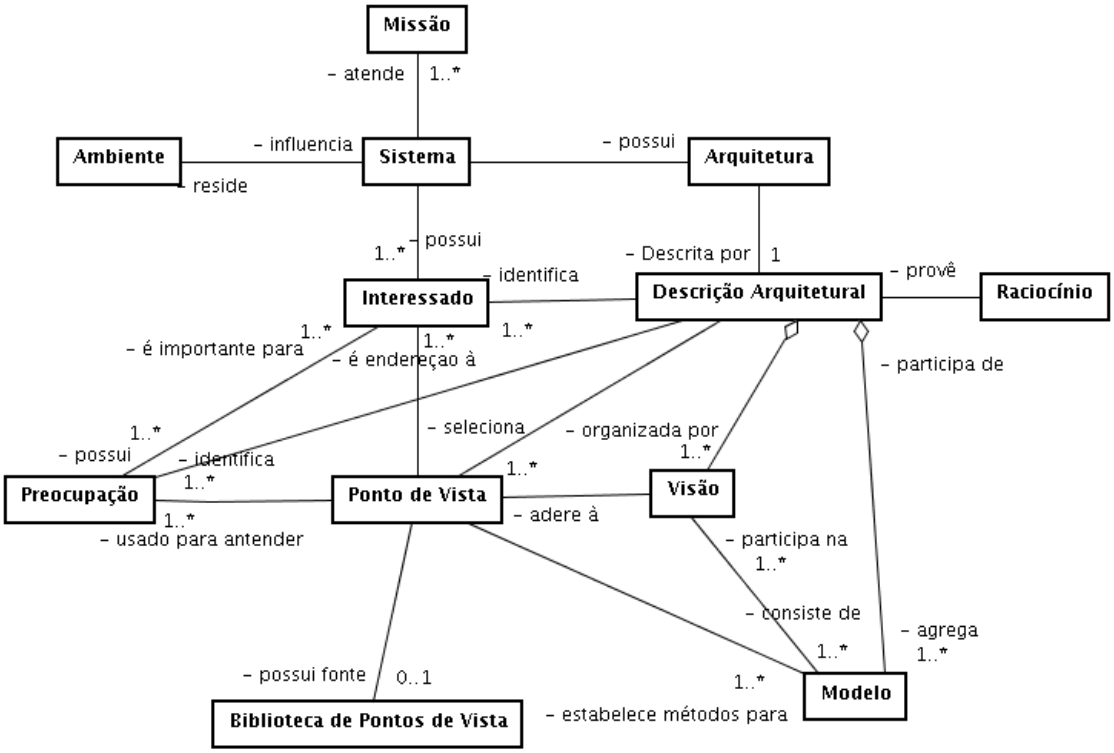

Figura 7: Modelo Conceitual de Descrição de Arquitetura, modificado de (IEEE, 2000).

Dentre as definições contidas no padrão, seguem detalhes das principais:

- descrição de arquitetura - um conjunto de visões e informações adicionais sobre a arquitetura;

- ambiente de sistema ou contexto - determina a configuração e circunstâncias do desenvolvimento, operacionais, políticas e outras influências sobre o sistema;

- interessado - um indivíduo, grupo ou organização que possui ao menos um interesse relacionado ao sistema;

- preocupação - um requisito funcional ou não funcional;

- visão - um conjunto de modelos representando um sistema da perspectiva de um conjunto de interesses relacionados;

- ponto de vista - as convenções para criação, descrição e análise de uma visão;

- modelo - um diagrama em particular ou descrição construída segundo o método definido no ponto de vista. Estes fornecem uma descrição específica do sistema, que pode incluir subsistemas e elementos identificáveis (IEEE, 2000).

Destaca-se como contribuição da ANSI/IEEE-1471-2000 os fundamentos conceituais relativos à documentação de arquitetura de *software*, mas a aplicação prática de tais conceitos soa um tanto quanto vaga, uma vez que não existem orientações práticas sobre a elaboração de pacotes de documentação (CLEMENTS et al., 2002b). O que faz desse padrão um referencial terminológico, mas não substituiu os diversos métodos de projeto de arquitetura existentes, como, por exemplo:

- *Atribute-Driven Design* (ADD) (BASS; CLEMENTS; KAZMAN, 2003), desenvolvido pelo SEI.

- *Siemens 4 Views* (S4V) (HOFMEISTER; NORD; SONI, 1999), desenvolvido pela *Siemens Corporate Research*.

- Visão 4 + 1 do RUP (KRUCHTEN, 1995), desenvolvido pela *Rational Software*.

- *Strategic Scenario-Base Architecting* (IONITA; AMERICA; HAMMER, 2005), desenvolvido em parceria entre a *Eindhoven University of Technology* e a *Philips Research*.

Hofmeister et al. (2007) fez uso de suas experiências no desenvolvimento de métodos de projeto de arquitetura para desenvolver um modelo geral para projeto de arquitetura de *software* derivado de abordagens de mercado. Este modelo, além de consolidar os pontos em comum entre os métodos analisados, fornece uma base sólida para comparação com outros métodos não considerados. E, devido à sua amplitude, será utilizado com o intuito de esclarecer de forma ampla a atividade de projeto de arquitetura.

O modelo geral de projeto de arquitetura de *software* organiza-se em duas dimensões, sendo que a primeira parte de uma perspectiva estrutural, concentrando-se nas atividades, sumarizada pela Figura 8. A segunda dimensão remete ao tempo de execução das atividades, partido do pressuposto de que elas não são executadas sequencialmente, compondo um *backlog*[2] de atividades a serem recuperadas repetidamente ao longo do ciclo de desenvolvimento de *software*.

3.3.1 Atividades e Artefatos do Projeto

No que tange à sua estrutura, o modelo genérico de Hofmeister et al. (2007) é composto pelas seguintes atividades e artefatos:

[2]Backlog: acúmulo, pedidos pendentes, demanda não atendida/reprimida, atraso no atendimento, lista de espera. (Babylon Translator, 2007)

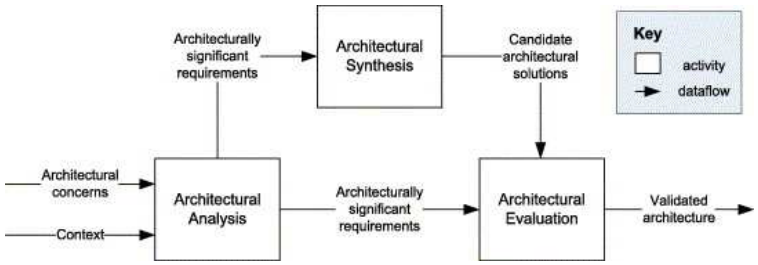

Figura 8: Atividades de Projeto de Arquitetura (HOFMEISTER et al., 2007).

- *preocupações de arquitetura* - este artefato congrega informações relacionadas às preocupações a serem atendidas pelo *software*. A definição de preocupação segue a mesma apresentada pela ANSI/IEEE-1471-2000;

- *contexto* - assim como o artefato anterior, este segue a definição do termo contexto ou ambiente da ANSI/IEEE-1471-2000;

- *análise de arquitetura* - esta atividade tem como atribuição identificar e caracterizar os problemas a serem contemplados pela arquitetura. Seu resultado corresponde aos requisitos significativos para a arquitetura;

- *requisitos significativos para a arquitetura* - corresponde àqueles requisitos realmente relevantes para a arquitetura. Vale observar que nem todos os requisitos se encontram nesse estado. Eles podem se originar a partir da análise de requisitos e surgir, adicionalmente, de outras preocupações de arquiteturas e contextos do sistema;

- *síntese de arquitetura* - esta atividade corresponde à análise dos problemas a serem resolvidos, identificação das alternativas mais pertinentes, bem como às justificativas para elas e consecutiva documentação que incorpore toda esta atividade intelectual. Seu resultado corresponde às soluções de arquiteturas candidatas;

- *soluções de arquiteturas candidatas* - representam as possíveis soluções de arquiteturas, em conjunto com as respectivas justificativas pela escolha ou recusa. Ou seja, registra-se nela as bases do raciocínio realizado pelo arquiteto nessa definição;

- *avaliação de arquitetura* - garante que as decisões de projeto de arquitetura são pertinentes, em comparação com os requisitos significativos para a arquitetura. Como resultado dessa atividade, tem-se uma solução de arquitetura validada ou invalidada;

- *arquitetura validada* - consiste naquela solução de arquitetura candidata consistente com os requisitos significativos para a arquitetura.

3.3.2 *Backlog*

Devido à sua alta complexidade, a atividade de projeto de arquitetura não se adapta a um fluxo de trabalho em série, como mostrado pela Figura 8. Na prática, o que acontece é um processo irregular, no qual o arquiteto mantém implícita ou explicitamente um *backlog* de necessidades, questões e problemas que necessitam ser contemplados, bem como idéias possíveis de serem consideradas. Este *backlog*, que de fato orienta o fluxo das atividades, é constantemente priorizado segundo, principalmente, fatores externos, como, por exemplo: riscos, marcos de entrega, pressão sobre a equipe para começar a trabalhar em uma parte específica do sistema, percepção de dificuldade de algum item etc. A Figura 9 ilustra a dinâmica relacionada ao *backlog*.

O modelo geral de projeto de arquitetura de *software* apresentado nessa Seção facilita a compreensão em alto nível da atividade de projeto de arquitetura, mas deixa a desejar no que tange à sua implementação, por não apresentar uma alternativa de uso aos próprios modelos usados como base para sua concepção. Por esta razão, serão considerados nas Seções seguintes

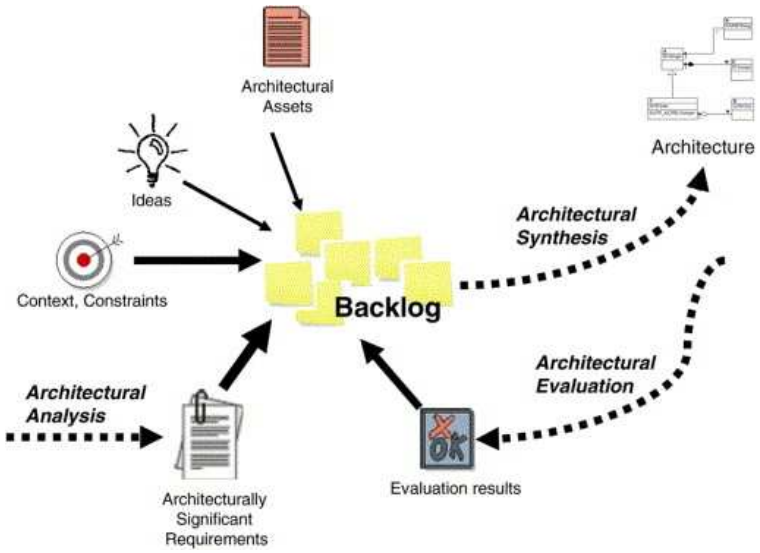

Figura 9: Dinâmica do *Backlog* (HOFMEISTER et al., 2007).

os dois principais modelos existentes. Antes, porém, será abordado o conceito de estilos ou padrões, considerados por ambos.

3.4 Padrões ou Estilos

Os aspectos em comum reconhecidos em uma coleção de sistemas podem ser sintetizados em estilos ou padrões de arquiteturas, que os descrevem em termos de componentes e conectores, adicionados das restrições impostas sobre estes. Esta descrição está em consonância com a definição posta por Garlan e Shaw (1993):

> "O estilo de arquitetura determina o vocabulário de componentes e conectores que podem ser usados em instâncias do estilo, juntamente com um conjunto de restrições para suas combinações".

A principal vantagem dos estilos está na documentação de padrões de organização de sistemas, na qual seus atributos de qualidade, bem como sua aplicabilidade[3] estão registrados. Isso permite que o arquiteto selecione um estilo baseado nos requisitos dos sistemas sem a necessidade de partir do princípio (BASS; CLEMENTS; KAZMAN, 2003, pg. 25).

[3]Garlan e Shaw (1993) detalha vários estilos de arquiteturas, valendo-se de comparação entre eles para fins de aplicabilidade.

Alguns dos padrões mais comuns já identificados e documentados são cliente servidor, *pipe and filter* (BARBOSA, 2005), modelo-visão-controle (KRASNER; POPE, 1988), Chiron 2 (MEDVIDOVIC, 1995), orientado a objetos e o estilo em camadas (GARLAN; SHAW, 1993). É comum perceber em um mesmo sistema o uso de vários padrões em sua arquitetura.

3.5 *Atribute-Driven Design*

O ADD é um método de definição de arquitetura de *software* cujo processo de decomposição se baseia nos atributos de qualidade[4] a serem atendidos pelo *software*. Este é um processo de decomposição recursiva pelo qual, em cada fase, táticas e padrões de arquitetura são escolhidos para satisfazer o conjunto de cenários de qualidade e, então, a funcionalidade é alocada para instanciar os tipos de módulos fornecidos pelo padrão (BASS; CLEMENTS; KAZMAN, 2003).

3.5.1 Método de Projeto

Os cenários de qualidade mais importantes para o sistema direcionam o projeto da arquitetura no ADD. Como conseqüência, tem-se que os requisitos do sistema, representados no formato de cenários de qualidade, são pré-requisitos para se começar o projeto de arquitetura. Assim, deve haver uma confiança mínima sobre estes cenários para dar início às atividades que, segundo Bass, Clements e Kazman (2003), são organizadas de acordo com as atividades:

1. escolha o módulo a se decompor. O módulo por onde começar é geralmente o sistema como um todo. Todas as entradas exigidas para este módulo devem estar disponíveis (restrições, requisitos funcionais, requisitos de qualidade).

[4]Os atributos de qualidade estão fortemente associados aos requisitos não funcionais, bem como às necessidades do negócio.

2. Refine os módulos de acordo com os passos:

   (a) "Escolha os direcionadores de arquiteturas[5] do conjunto de cenários concretos de qualidade e requisitos funcionais. Este passo determina o que é importante para esta decomposição.

   (b) Escolha um padrão de arquitetura que satisfaça às orientações. Crie (ou selecione) o padrão baseado nas táticas que podem ser usadas para alcançar as orientações. Identifique módulos filhos exigidos para implementar as táticas.

   (c) Instancie módulos e funcionalidades alocadas a partir dos casos de uso e represente os resultados usando múltiplas visões.

   (d) Defina as interfaces dos módulos filhos. A decomposição fornece módulos e restrições quanto ao tipo de interações entre os módulos. Documentar esta informação no artefato de interface para cada módulo.

   (e) Verifique e refina os casos de uso e cenários de qualidade e faça deles restrições para os módulos filhos. Este passo verifica que nada importante foi esquecido e prepara os módulos filhos para decomposição detalhada ou implementação."

3. Repita estes passos anteriores para todos os módulos que necessitam de decomposição detalhada.

Uma orientação de cunho mais prático acerca do método ADD pode ser encontrada na revisão do modelo realizada por Wojcik et al. (2006, pg. 33).

3.5.2 Visões

As visões recomendadas para o ADD são resultantes do extenso trabalho desenvolvido pelo SEI, conhecido como abordagem *"Views and Beyond"* (CLEMENTS et al., 2002a), cuja síntese é apresentada a seguir:

- Módulo: este corresponde à visão estrutural da arquitetura, compreendendo os módulos de código como classes, pacotes e subsistemas do projeto. Ele captura também

[5] Os direcionadores de arquiteturas são consequência da premissa de que em qualquer sistema alguns atributos de qualidade são prioritários a outros. Sendo assim, a arquitetura de *software* deve buscar refletir aqueles atributos mais importantes

decomposição, herança, associações e agregações de módulos.

- Componente e conector: esta visão descreve os aspectos comportamentais da arquitetura. Componentes são tipicamente objetos, *threads* ou processos e os conectores descrevem como os componentes interagem. Conectores comuns são soquetes, camada intermediária como *Common Object Request Broker Architecture* (CORBA) ou memória compartilhada.

- Alocação: esta visão mostra como os processos na arquitetura são mapeados ao *hardware* e como eles se comunicam usando rede e/ou banco de dados. Ele captura também uma visão do código fonte no sistema de gerência de configuração e quem na equipe de desenvolvimento possui responsabilidade por cada módulo.

3.5.3 Avaliação

Em termos de avaliação, o ADD pode contar com pelo menos dois métodos desenvolvidos pelo próprio SEI e fundamentado nas mesmas bases dele: o *Architecture Trade-off Analysis Method* (ATAM) (KAZMAN; KLEIN; CLEMENTS, 2000) e *Cost-Based Analysis Method* (CBAM) (KAZMAN; ASUNDI; KLEIN, 2002).

3.5.3.1 *Architecture Trade-off Analysis Method*

O método de avaliação de arquitetura ATAM surgiu com o intuito de preencher uma lacuna deixada por outros métodos semelhantes, cujo alvo gravitava geralmente em torno de algum dos atributos de qualidade. Tal abordagem os distanciava da realidade, uma vez que, na prática, o que acontece é um balanceamento quanto ao atendimento desses atributos de qualidade pela arquitetura, de acordo com a relevância de cada um (KAZMAN et al., 1999).

A análise no ATAM se fundamenta na análise de risco, de maneira a identificar áreas de potenciais riscos na arquitetura de sistemas de *softwares* complexos. Como implicações, tem-se que: 1) o ATAM pode ser aplicado precocemente no ciclo de vida de desenvolvimento de *software*; 2) ele pode ser realizado rapidamente e a custos baixos, pois se concentra em artefatos de projeto; 3) seu sucesso independe de atributos de qualidade mensuráveis, mas sim da percepção da influência de decisões de arquitetura em qualquer dos atributos de qualidade de interesse.

De acordo com Kazman, Klein e Clements (2000), o método ATAM é realizado segundo as seguintes etapas:

1. Apresentar o método - momento em que o método é apresentado para os interessados.

2. Apresentar as orientações de negócio - apresentação pelo gerente do projeto dos objetivos de negócio que levaram ao esforço de desenvolvimento, que serão, consequentemente, os direcionadores primários da arquitetura.

3. Apresentar a arquitetura - descrição da arquitetura proposta pelo arquiteto, concentrando-se em como contemplar os direcionadores de negócio.

4. Identificar as abordagens de arquiteturas - o arquiteto identifica as abordagens usadas na definição da arquitetura, sem avaliá-las.

5. Gerar a árvore de utilidade de atributos de qualidade - os fatores de qualidade (desempenho, disponibilidade, segurança, modificabilidade etc.) relacionados ao sistema são

elicitados, especificados a partir de cenários, anotados quanto a estímulos e respondidos e priorizados.

6. Analisar as abordagens de arquitetura - com base nos fatores identificados no passo 5, as abordagens de arquitetura que os contemplam são elicitadas e analisadas. Neste passo, os riscos, pontos sensíveis e de equilíbrio são identificados.

7. *Brainstorm* e priorização de cenários - os cenários gerados no passo 5 são priorizados segundo votação pelos interessados.

8. Analisar as abordagens de arquitetura - este passo repete o passo 6, com o diferencial de ter em mãos cenários classificados do passo 7 para esta análise.

9. Apresentar os resultados - baseado na informação coletada pelo ATAM, a equipe apresenta a consolidação dos resultados encontrados, bem como as estratégias de mitigação propostas.

Por fim, cumpre destacar que o ATAM deriva de técnicas de três fontes: o *Software Architecture Analysis Method* (SAAM) (KAZMAN et al., 1996), cujo foco está no uso de cenários para análise dos atributos de qualidade; comunidades de atributos de qualidade e na noção de estilos de arquitetura, principalmente sob o enfoque do *Attribute-Based Architecture Style* (ABAS) (KLEIN; KAZMAN, 1999).

3.5.3.2 *Cost-Based Analysis Method*

O CBAM vem complementar o ATAM no que se refere ao risco financeiro, pois se baseia fortemente na relação de custo/benefício das decisões de arquitetura. Esta metodologia defende

que as estratégias de arquiteturas (ASs) afetam os atributos de qualidade do sistema, que, por sua vez, fornecem algum *benefício*, também referido como *utilidade*, aos interessados no sistema. Entretanto, cada AS possui também seu custo associado e demanda tempo de implementação. Dadas estas informações, o CBAM pode ajudar os interessados na escolha das ASs com base em seu retorno sob o investimento fornecido (KAZMAN; ASUNDI; KLEIN, 2002).

Como complemento a todo o *framework* do SEI que trata de arquitetura, foi desenvolvido recentemente um novo método, conhecido como *Analytic Principles and Tools for Improvement of Architectures* (APTIA), cujo objetivo é estender os métodos de análise existentes tanto em profundidade quanto em extensão. A profundidade está relacionada ao nível de detalhe do APTIA ser maior que nos outros e o extensão se refere ao fato de que seu resultado corresponde a alternativas de projeto que melhorarão a arquitetura do *software* por completo (KAZMAN; BASS; KLEIN, 2006).

3.6 Visão 4 + 1 do *Rational Unified Process*

Um dos principais fundamentos do RUP é o desenvolvimento orientado à arquitetura (KRUCHTEN, 2003). Esta abordagem é fruto do trabalho desenvolvido por Kruchten (1995) na Rational, denominado modelo de visão de arquitetura de *software* 4 + 1.

Este modelo consiste de múltiplas visões concorrentes que habilitam contemplar diversas preocupações dos interessados de maneira segmentada. As quatro visões preconizadas pelo método para o projeto são: visão lógica, de processo, de desenvolvimento, de implantação, todas elas amarradas pela visão de casos de uso (+1), que as relacionam com o contexto e as metas.

A Figura 10 ilustra a relação entre estas visões, bem como seu público-alvo. O projeto de

arquitetura preconizado por Kruchten (1995) é orientado por um processo iterativo, no qual o ponto de partida está na descrição dos cenários ou casos de uso mais relevantes para a arquitetura do *software*, bem como a especificação suplementar (requisitos não funcionais), a partir dos quais o arquiteto abstrai os principais atributos do domínio do problema, representando-os na *visão lógica*. As classes lógicas, por sua vez, podem ser mapeadas a módulos e pacotes na *visão de desenvolvimento* e os processos e atividades na *visão de processo*. Finalmente, os processos e os módulos são mapeados ao *hardware* a partir da *visão de implantação*. Nas iterações subsequentes, cenários adicionais são modelados de acordo com esta dinâmica, até que a arquitetura esteja estável. O ciclo que se encerra com a estabilização da arquitetura tem maior demanda nas fases de concepção e elaboração do RUP.

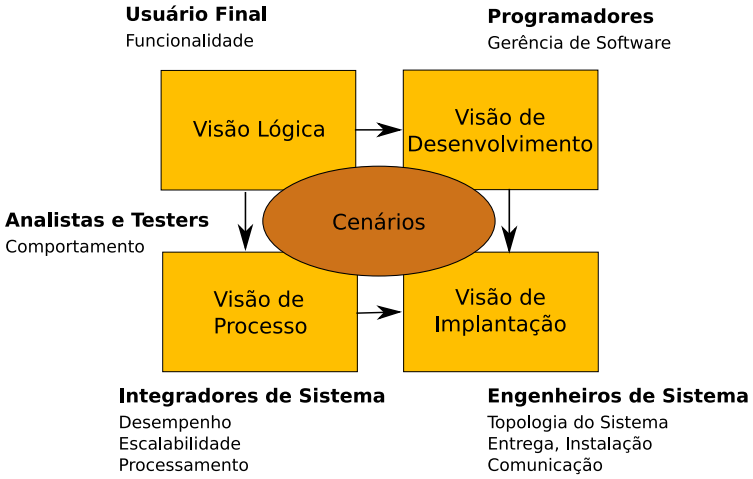

Figura 10: Modelo Visão 4 + 1 do RUP, adaptado de (KRUCHTEN, 2003).

Afirma-se ser este o modelo mais usado pela indústria de *software*, fato este justificado pela sua maturidade em termos de técnicas, modelos, ferramental de apoio integrados ao processo de desenvolvimento de *software* e para condução das atividades de desenvolvimento da arquitetura ao longo do ciclo de desenvolvimento de *software*.

3.7 Arquitetura de *Software* Seguro

O potencial do projeto de *software* seguro orientado pela arquitetura pode ser ilustrado pela abordagem de projeto aplicada ao sistema operacional Windows Server 2003, cuja segurança teve uma melhora significativa após a redefinição de sua arquitetura, sem lançar mão de recursos de segurança adicionais, como, por exemplo, criptografia.

O servidor web *Internet Information Services* (IIS) foi lançado em 1995. A partir desta data, ele passou por diversas mudanças durante os anos seguintes, alcançando a versão 5.1 em 2001. Ao longo deste curso, ele foi responsável por várias vulnerabilidades, algumas das quais causaram danos graves (MS03-007..., 2006). A maior mudança de arquitetura, conforme apresentado pela Tabela 4, veio a ser introduzida na versão 6.0 (REN, 2006). Esta versão se mostrou muito mais segura que as anteriores devido às mudanças de arquitetura apresentadas. Observa-se, ainda, que pouca novidade tecnológica fora adicionada, porém, a mudança na arquitetura foi drástica.

Tabela 4: Visão da Arquitetura do *Microsoft Windows* 2003 com Enfoque em Segurança de seu Servidor *Web*, modificado de (REN, 2006)

Problema Potencial	**Mecanismo de Proteção**	**Princípio de Projeto**
A DLL (ntdll.dll) associada não era vulnerável pois...	O código se tornava mais robusto durante o *Security Push*.	Verificar pré-condição
Mesmo que ela fosse vulnerável...	O IIS 6.0 não entra em operação por padrão no Windows 2003.	Seguro por padrão
Mesmo que ele entrasse em operação...	IIS 6.0 não possui o WebDAV habilitado por padrão.	Seguro por padrão
Mesmo que o WebDAV estivesse habilitado...	O tamanho máximo da URL no IIS 6.0 é por padrão 16KB (¿ 64KB são necessários para explorar.)	Pré-condição restritiva, seguro por padrão
Mesmo que o *buffer* fosse grande o suficiente...	O processo é interrompido devido ao código de detecção de estouro de *buffer* inserido pelo compilador.	Pré-condição restritiva, verificação de pré-condição
Mesmo que houvesse um estouro de *buffer* explorável	Isto aconteceria no w3wp.exe, que é executado como um serviço de rede (ao invés de um administrador).	Menor privilégio

Deflagra-se nesse exemplo o reprojeto da arquitetura, no qual as principais mudanças são fundamentadas em princípios de projeto seguro, como, por exemplo, defesa em profundidade, segurança por padrão e uso do menor privilégio, ao invés de soluções de alta complexidade.

Merece destaque, também, os resultados obtidos pelo *qmail*[6], servidor de *email* para *UNIX*, cujo desenvolvimento foi motivado pelas inúmeras vulnerabilidades encontradas no *sendmail*[7]. Sua concepção teve como ponto de partida a análise das causas das vulnerabilidades encontradas no *sendmail*, de forma que sua arquitetura contemplasse tais defeitos. Seus resultados em termos de segurança são fabulosos e motivaram seu desenvolvedor a oferecer um prêmio, em aberto desde 1997, para aqueles que conseguissem subverter o *software*, de modo a obter controle sobre outra conta de usuário.(BERNSTEIN, 1997).

[6]http://www.qmail.org/top.html
[7]http://www.sendmail.org/

4 *Padrões de Software*

Este Capítulo aborda os conceitos associados aos padrões de segurança, partindo do histórico dos padrões de projeto, chegando à metodologia de desenvolvimento baseada em padrões.

4.1 Histórico

O uso sistemático de padrões foi originalmente concebido pelo arquiteto Christopher Alexander, que registrou mais de 250 padrões de planejamento urbano e arquitetura de construção, partindo de padrões muito abrangentes, que contemplavam regiões e cidades, passando pelo detalhe da vizinhança, grupos de prédios, prédios, quartos, chegando a detalhes da construção. Com a experiência, logo foi elaborada a estrutura de padrões contexto-problema-solução (ALEXANDER et al., 1977).

A abordagem de padrões no desenvolvimento de *software* foi primeiramente empregada por Ward Cunningham e Kent Back, em 1987, quando decidiram usar padrões para auxiliar os analistas de seus clientes, em muitos casos novatos no SmalTalk, a aproveitarem os pontos fortes da ferramenta e evitar erros comuns. O resultado foi tão satisfatório que eles apresentaram a referida experiência no *Object-Oriented Programming, Systems, Languages & Applications* (OOPSLA) (BECK; CUNNINGHAM, 1987).

Somando-se à iniciativa anterior, Coplien (1992), mesmo sem dar ênfase aos padrões, explorou os conceitos de melhores práticas aplicadas a C++. Em paralelo, Coad (1992) destacou o paradigma dos padrões para o desenvolvimento de *software*. Já Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (a gangue dos quatro (GoF)) trabalharam numa compilação de padrões nos workshops dedicados a esta matéria no OOPSLA de 1991 e 1992.

Em 1993, o Grupo Hillside, organização que hoje é líder absoluta no assunto de padrões, foi concebida sob as bases estabelecidas pelo trabalho da GoF, e em 1995 foram publicados os primeiros anais da conferência *Pattern Languages of Programs Conference* (PLoP), evento dedicado ao tema e criado pelo grupo.

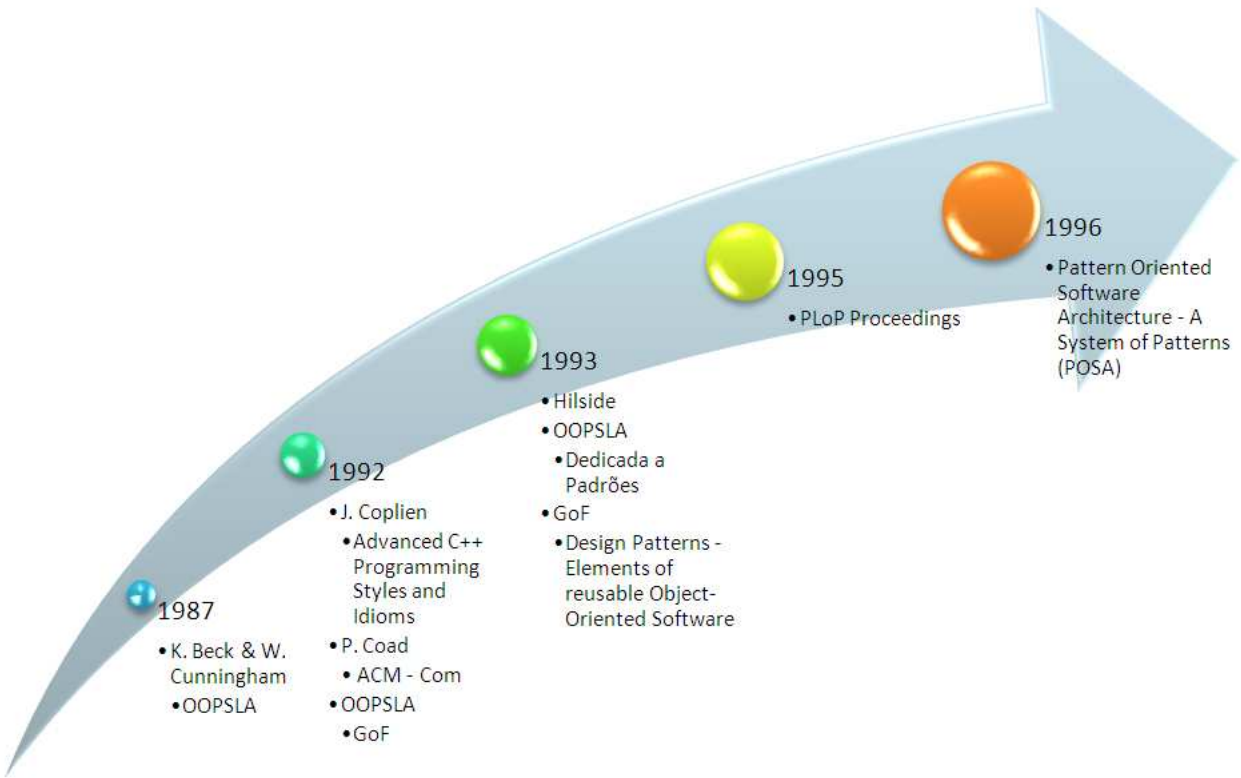

Figura 11: Marcos na História dos Padrões

A consolidação da iniciativa da GoF no livro de Gamma et al. (1994), adicionada da série de livros sobre arquitetura e padrões, que se iniciou com Buschmann et al. (1996), trouxe para a comunidade de desenvolvimento de *software* a fundamentação teórica e prática para a

difusão do uso de padrões. A Figura 11 resume os marcos históricos relacionados aos padrões de desenvolvimento de *software*.

4.2 Conceitos Básicos

Os padrões fornecem soluções funcionais, concretas e adaptáveis para problemas que surgem repetidamente em certas situações durante o desenvolvimento de *software*, variando de contextos organizacionais a de programação (BUSCHMANN; HENNEY; SCHMIDT, 2007).

Um fator determinante sobre a existência de um padrão é sua recorrência. A recorrência pressupõe a existência de padrões bons e outros, cujo uso possa até vir a ser um senso comum, mas não são recomendados. Isso exige que o analista de padrões tenha o discernimento sobre o que é simplesmente um cenário recorrente de projeto de outra situação singular. Esta última sim merece a promoção a um padrão. Existem situações em que o padrão é de boa qualidade, mas sua aplicabilidade carece de detalhes, levando o analista a usar um bom padrão em uma situação inadequada. Para se evitar esse desentendimento, a descrição do padrão deve ser clara o bastante para tornar possível a visualização de suas qualidades.

Um padrão deve informar à sua audiência como construir uma solução de maneira a ser aberta o suficiente para ser geral, mas fechada para evitar vacuidade ou ambiguidade acidental. Sua solução deve descrever tanto o "processo" quanto a "coisa". A "coisa" é "resultado" do processo. Para a maioria dos padrões de *software*, a "coisa" significa um esboço de projeto em alto nível ou a descrição de detalhes de código. O "processo", nesse caso, consiste na dinâmica que envolve a solução de um problema único e específico - objetivo de qualquer padrão individual. Apesar de sua tônica microscópica, seu uso combinado mostra-se mais adequado a representar um contexto mais amplo, realidade de qualquer projeto de *software*.

Cumpre reforçar que para um padrão ser considerado útil, ele deve não apenas propor qualquer solução para um problema em questão, mas apresentar uma solução robusta que resolva o problema de maneira ótima. A solução em um bom padrão necessita, ainda, de registros que a comprovem. Para que um padrão seja de qualidade, não basta a representação de idéias que podem vir a funcionar, mas sim evidências de experiências repetidas e bem sucedidas de sua aplicação no passado.

Para que a solução de um padrão consiga contemplar otimamente um determinado problema, é desejável que a caracterização desse também seja ótima. Ela pode envolver um conjunto de elementos incluindo requisitos, propriedades desejáveis e fatos. Esse conjunto de informações adicionais ao problema, conhecido como forças, habilitam a sintetização de uma solução adequada ao problema. O atendimento de uma força em particular pode acontecer em detrimento de outra de um mesmo problema. Por exemplo, eficiência pode se contrapor a flexibilidade e vice-versa. Nesses casos, uma solução que atenda, não completamente, mas suficientemente as forças de uma maneira equilibrada é desejável.

Em síntese, os padrões devem corresponder à seguinte especificação:

- "Cada padrão descreve um problema recorrente em um ambiente e, então, descreve o núcleo da solução para ele."

- "Cada padrão é uma regra de três partes, consistindo na relação entre um certo um contexto, um problema e uma solução".

Conforme especificado por Buschmann, Henney e Schmidt (2007), os elementos fundamentais para se descrever um padrão são:

- Identificação: nome e classificação para identificação do padrão.

- Contexto: situação na qual o problema surge.

- Problema: conjunto de forças que surgem recorrentemente no contexto.

- Solução: configuração que equilibra as forças.

- Consequências: aquilo que surge com a aplicação do padrão.

4.3 Padrões de Segurança

Yoder e Barcalow (1997) escreveu o primeiro artigo sobre padrões de segurança, apesar de já haver artigos desenvolvidos antes disso, considerando modelos orientados a objetos de mecanismos de segurança (FERNANDEZ; LARRONDO-PETRIE; GUDES, 1993; ESSMAYR; PERNUL; TJOA, 1997). Os padrões de segurança congregam o conhecimento extensivo acumulado sobre segurança com a estrutura fornecida pelos padrões de projeto, oferecendo orientações para projeto e avaliação de sistemas seguros. Os padrões de segurança descrevem um modelo genérico preciso para um mecanismo de segurança. Eles contribuem também para o entendimento de sistemas complexos e para o ensino de conceitos de segurança.

Em (SCHUMACHER, 2003), um padrão de segurança é definido como um padrão que descreve um problema particular recorrente que emerge em um determinado contexto e apresenta uma solução genérica comprovada para isso. No campo da segurança, o problema ocorre quando um ativo, como uma empresa, um sistema ou uma aplicação é protegida de maneira insuficiente contra um abuso ou uma situação que surge, permitindo violação de segurança.

Os padrões de segurança podem se referir à arquitetura do *software* ou até contemplar um nível mais alto de abstração, envolvendo a organização, processos etc. Em (SCHUMACHER et al., 2006), um catálogo de padrões de segurança pode ser apreciado. Para ilustrar, apresenta-se a seguir um exemplo de padrão presente nesse catálogo:

Monitor de Referência

Em um ambiente computacional cujos usuários ou processos requisitam dados ou recursos, este padrão impõe restrições de acesso declaradas quando uma entidade ativa requisita recursos. Ele descreve como definir um processo abstrato que intercepte todas as requisições a um recurso e as verifica quanto à conformidade com as autorizações.

Denominações Adicionais
Ponto de Imposição de Política

Exemplo
Em um hospital, declara-se os acessos permitidos à informação para médicos e outro pessoal. Entretanto, espera-se uma adequação voluntária dos afetados pelas regras, o que não vem acontecendo, pois o pessoal frequentemente subverte as regras à sua conveniência e não existe uma maneira de impô-las.

Contexto
Um ambiente computacional no qual usuários ou processos requisitem dados ou recursos.

Problema
Não se impor as autorizações definidas é o mesmo que não tê-las, usuários e processos podem realizar todo tipo de operação ilegal. Qualquer usuário pode ler qualquer arquivo, por exemplo.
A solução para esse problema deve resolver as seguintes forças:

- Definir regras de autorização não é suficiente, elas devem ser impostas sempre que um usuário ou processo requisitar um recurso.
- Existem muitas formas possíveis de imposição, dependendo da unidade ou nível de arquitetura envolvido. É necessário um modelo de imposição que se aplique a todo nível do sistema.

Solução
Definir um processo abstrato que intercepte qualquer requisição e verifique quanto à conformidade com as autorizações preestabelecidas.

Estrutura
A Figura 12 mostra o diagrama de classe que descreve a materialização do Monitor de Referência. Nessa Figura "*Authorization Rules*" denota uma

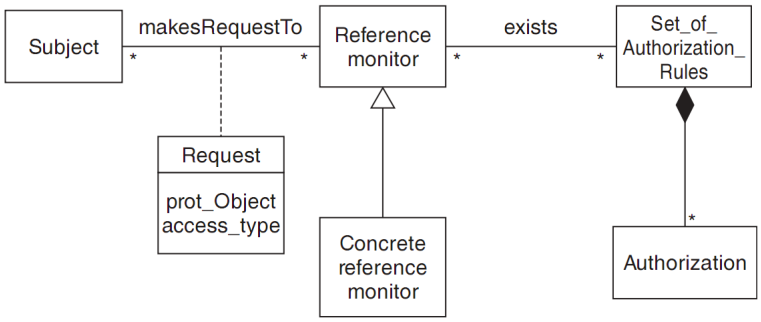

Figura 12: Diagrama de Classe do Monitor de Referência.

coleção de regras de autorização organizadas como *Access Control List* (ACL) ou em outra forma.

Dinâmica

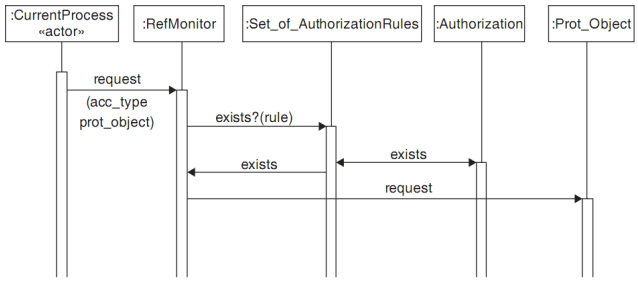

Figura 13: Diagrama de Sequência do Monitor de Referência.

A Figura 13 é um diagrama de sequência mostrando como uma requisição de um processo é verificada. O Monitor de Referência procura regras existentes que autorizem a requisição. Caso elas existam, a requisição terá sua realização autorizada.

Implementação

Um Monitor de Referência é requerido em cada seção do sistema que possua recursos que possam ser requisitados. Exemplos incluem gerenciamento de memória (para controlar acesso à memória), e gerenciamento de arquivo (para controlar o uso dos arquivos).

Exemplo Resolvido

O hospital comprou um sistema de banco de dados para registrar informação de pacientes. Agora, quando o usuário tenta acessar os dados de pacientes, sua autorização é verificada antes de se efetivar tal acesso. Ações como a leitura ou escrita são controladas. Por exemplo, somente médicos e enfermeiras são habilitados a modificar os registros dos pacientes.

Exemplos de Uso A maioria dos sistemas operacionais implementam este conceito, incluindo-se Solaris 9, Windows 2000, AIX e outros. O *Java Security Manager* é outro exemplo. Sistemas gerenciadores de banco de dados também possuem um sistema de autorização que controla acesso a dados requisitados via *queries*.

Consequências
Os seguintes benefícios podem ser esperados a partir da aplicação deste padrão:

- Caso todas as requisições sejam interceptadas, é possível certificar que elas estejam em conformidade com as regras.
- O processo abstrato descrito aqui não restringe a implementação do sistema.

As seguintes deficiências potenciais podem emergir da aplicação deste padrão:

- Implementações específicas são necessárias para cada tipo de recurso. Por exemplo, um gerenciador de arquivo é necessário para controlar as requisições a arquivos.
- A verificação de cada requisição pode resultar em uma perda de desempenho intolerável. Pode haver a necessidade de realizar algumas verificações em tempo de compilação, por exemplo, e não repeti-la em tempo de execução.

Veja Também
Este padrão é um caso especial do *Check Point*. O padrão *Interceptor* pode atuar como Monitor de Referência em alguns casos. Versões concretas do Monitor de Referência incluem sistemas de controle de arquivo e *firewalls*.

4.4 Metodologia de Desenvolvimento de *Software* Baseada em Padrões

Este trabalho é baseado no contexto de uma metodologia de construção de sistemas seguros. Obviamente, não existe obrigação de aplicação como parte dessa abordagem, mas a metodologia fornece um contexto para o desenvolvimento. A idéia-chave da metodologia está na aplicação de princípios de segurança em cada etapa do ciclo de desenvolvimento de *software* e que cada etapa possa ser verificada quanto à sua conformidade com os princípios de segurança. Outra idéia básica é o uso de padrões para orientar a segurança em cada etapa no princípio de segurança de (FERNANDEZ et al., 2006a). A Figura 14 mostra um ciclo de vida de segurança de *software*, indicando onde a segurança pode se aplicada (setas brancas) e onde se pode validar quanto à conformidade com os princípios e políticas de segurança (setas pretas).

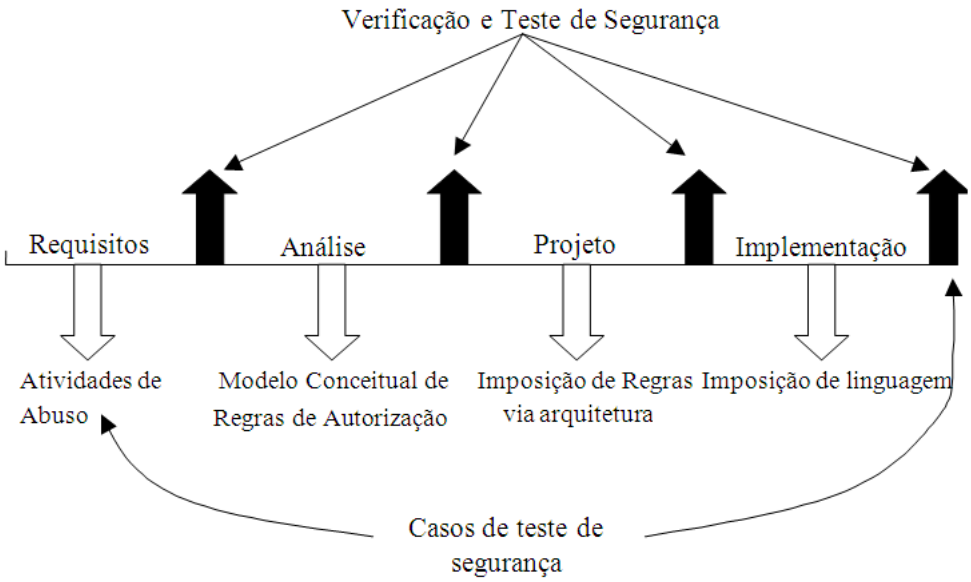

Figura 14: Ciclo de Vida de Segurança de *software*, adaptado de (FERNANDEZ et al., 2006a).

Este projeto propõe orientações para incorporar segurança a partir da fase de requisitos, passando por análise, projeto, implementação, teste e implantação. A metodologia-base considera as seguintes fases do desenvolvimento:

Análise do domínio: um modelo de negócio é definido. Os sistemas legados são identificados e suas implicações de segurança, analisadas. As restrições do domínio e de regulação são identificadas. Políticas devem ser definidas antecipadamente, a partir desta fase. A adequação da equipe de desenvolvimento é avaliada, levando a possíveis treinamentos adicionais. As questões de segurança dos desenvolvedores e de seu ambiente podem ser consideradas também em alguns casos. Esta fase pode ser realizada somente uma vez para cada domínio. As restrições de segurança podem ser aplicadas no modelo de domínio baseado em semântica.

Requisitos: os casos de uso definem as interações exigidas com o sistema. Aplicando o princípio de que a segurança deve se iniciar de uma visão de alto nível, é apropriado relacionar ataques aos casos de uso. Cada atividade de um caso de uso é avaliada com o objetivo de levantar possíveis ameaças (contribuição desta tese). A partir dos casos de uso, é possível determinar os direitos necessários para cada ator e então aplicar a política "necessidade de

informação". Cumpre observar que o conjunto de todos os casos de uso define os possíveis usos do sistema e a partir deles é possível determinar todos os direitos para cada ator. Os casos de teste de segurança para o sistema como um todo são também definidos nesta fase.

Análise: padrões de análise podem ser usados para a elaboração do modelo conceitual de forma mais confiável e eficiente. Os padrões de segurança descrevem modelos e mecanismos de segurança. Pode-se construir um modelo conceitual no qual aplicações repetidas de um padrão de modelo de segurança materializam os direitos determinados a partir dos casos de uso. De fato, os padrões de análise podem ser construídos com autorizações predeterminadas de acordo com os papéis em seus casos de uso. A partir daí, é necessário especificar adicionalmente apenas os direitos para aquelas partes não consideradas pelos padrões. Podemos partir da definição de mecanismos (contramedidas) para prevenir ataques.

Projeto: a partir do momento em que se identificaram os ataques possíveis para o sistema, os mecanismos de projeto são selecionados para controlar tais ataques. As interfaces dos usuários devem corresponder aos casos de uso e podem ser usadas para impor as autorizações definidas na fase de análise. As interfaces de segurança impõem as autorizações definidas quando o usuário interage com o sistema. Os componentes podem ser assegurados pelo uso de regras de autorização disponíveis em bibliotecas Java ou .NET. A distribuição traz outra dimensão na qual outras restrições de segurança podem ser aplicadas. Os diagramas de implantação podem definir configurações de segurança para serem usados pelos administradores do sistema. Uma arquitetura multicamada é necessária para impor restrições de segurança definidas pelo nível da aplicação. Para cada nível usam-se padrões para representar de maneira apropriada os mecanismos de segurança. As restrições de segurança devem ser mapeadas entre os níveis.

Implementação: esta fase requer a transformação das regras de segurança definidas na fase de projeto em código. Pelo fato dessas regras serem documentadas como classes, associações

e restrições, elas podem ser implementadas como classes em linguagem orientada a objetos. Nesta fase, pode-se selecionar pacotes ou *Commercial Off-the-shelf* (COTS) com mecanismos específicos de segurança, como, por exemplo: produto de filtro de pacotes, pacote de criptografia etc. Alguns dos padrões identificados nas fases anteriores podem ser substituídos por COTS.

5 *Análise de Atividades de Abuso*

Este Capítulo contempla a técnica desenvolvida para elicitação de requisitos de segurança de software, conhecida como Atividades de Abuso. Essa técnica, conhecida originalmente como "Ações de Abuso", foi delineada por Fernandez et al. (2006b) e instrumentalizada por esta tese, com o objetivo de auxiliar o analista na identificação do maior conjunto de ameaças possível a que um *software* possa estar submetido.

5.1 Requisitos de Segurança de *Software*

O que se pode entender por requisitos de segurança de *software*? Devanbu e Stubblebine (2000) declarou que "requisitos de segurança é uma manifestação da política organizacional de alto nível nos requisitos detalhados de um *software* específico". Esta definição é apropriada, políticas de alto nível definem os requisitos, mas elas não são suficientes. Para Haley et al. (2008), requisitos de segurança são "restrições nas funções do *software*, no qual tais restrições operacionalizam um ou mais objetivos de segurança". Porém, alguns requisitos de segurança, como, por exemplo, auditoria, representam mais do que restrições. Já a definição de Zuccato (2004): "uma necessidade ou restrição de um usuário, um interessado ou um ambiente relacionado com o objetivo de melhorar a segurança do sistema", complementa a definição em (HALEY et al., 2008).

No escopo desta tese, requisitos de segurança são as necessidades do *software* para que ele atenda às políticas regulatórias e institucionais do seu negócio. Portanto, o papel dos requisitos

de segurança é de fornecer informação sobre a real necessidade do sistema ou aplicação de forma a alcançar seus objetivos de negócio.

Mostra-se como uma atividade com alto grau de dificuldade reconhecer as necessidades de segurança do *software*. Problemas de segurança são, em sua maioria, consequência de atitudes maliciosas. Consequentemente, uma maneira efetiva de identificar quais são os requisitos de segurança dos sistemas é identificar os objetivos que o atacante deseja alcançar, como, por exemplo: roubo da identidade do cliente, transferência monetária para sua conta etc., para que, a partir dessa informação, se derive o controle necessário para impedir que tais objetivos sejam alcançados.

5.2 Atividades de Abuso

A abordagem de análise de requisitos baseada em atividades de abuso consiste em uma maneira sistemática de identificar ameaças e determinar as políticas que bloqueiem suas causas ou mitiguem seus efeitos. Para que isso aconteça, duas atividades são realizadas. A primeira corresponde à análise do fluxo de eventos em um caso de uso ou grupo de casos de uso, no qual cada atividade é analisada sob a perspectiva do atacante, de forma a encontrar as respectivas ameaças. Esta análise deve ser realizada para todos os casos de uso do sistema. A segunda atividade compreende a seleção de políticas de segurança adequadas para bloquear e/ou mitigar as ameaças identificadas (FERNANDEZ et al., 2006b).

Os requisitos de segurança devem definir as necessidades do sistema sem se comprometer com mecanismos específicos, focando no ataque em potencial ou ameaça. Para analisar essas ameaças, o insumo básico desta abordagem é o conjunto de diagramas de atividades correspondentes aos casos de uso e/ou fluxos de trabalho mais representativos do sistema[1](FOWLER,

[1]Pressupõe-se que todos os cenários de uso possíveis do sistema estejam representados pelos seus casos de uso.

2003). As atividades de um diagrama de atividades devem ser analisadas de modo a identificar os possíveis abusos a serem realizados em cada uma delas.

A análise do fluxo de eventos em um caso de uso implica a investigação detalhada de cada atividade, identificando-se qualquer forma de subvertê-lo sob o enfoque da segurança. Todas as informações das atividades de abuso podem ser registradas no próprio diagrama de atividades, a partir da extensão de sua notação. Os retângulos de vértices arredondados e linha pontilhada correspondem a atividades de abuso ou ameaças. As linhas de conexão pontilhadas representam fluxo de controle de abuso, que associam atividades de abuso e os objetos relacionados.

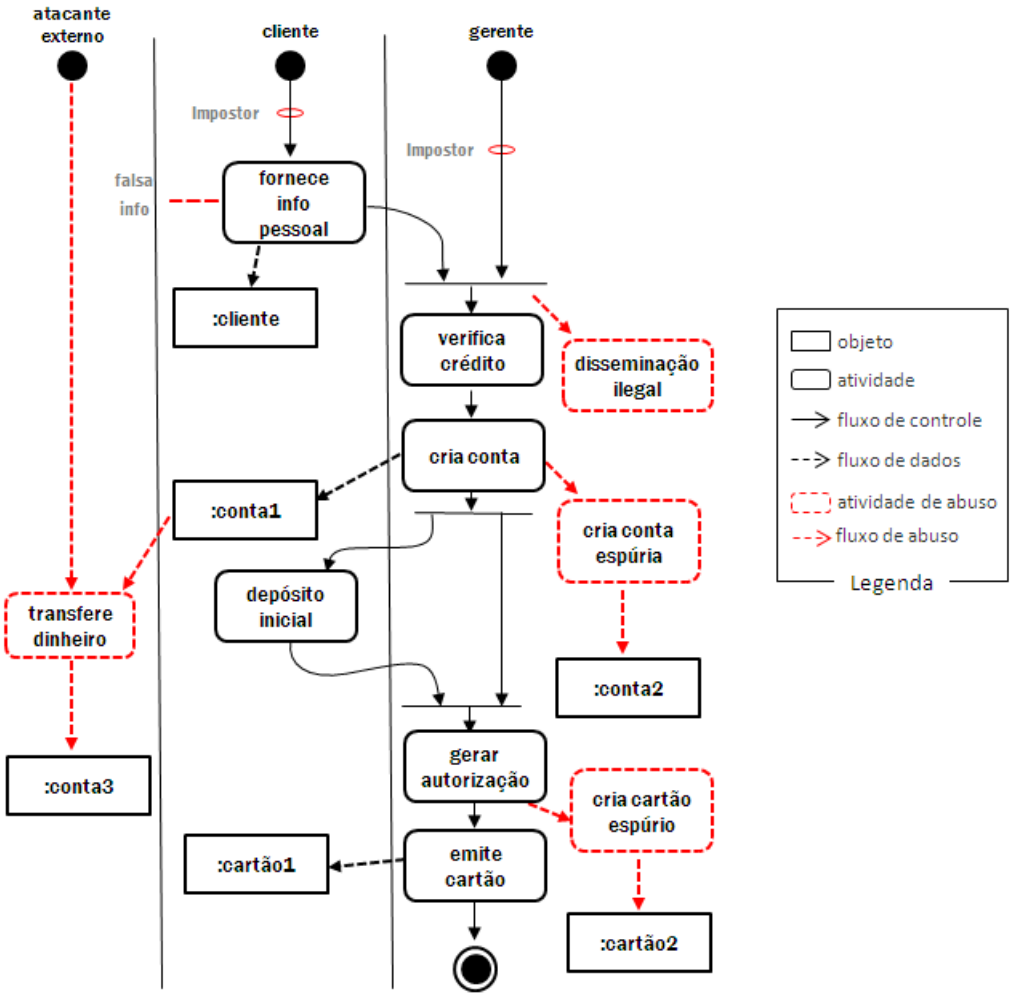

Figura 15: Diagrama de Atividades Estendido para “Atividades de Abuso”.

Um exemplo de diagrama com as extensões mencionadas é mostrado pela Figura 15, na

qual várias ameaças foram identificadas, nomeadas de A1 a A8, como mostrado na seção a) da Figura 16. Essas ameaças são levantadas pelo analista a partir do uso de seu conhecimento acerca da aplicação; tal abordagem se mostra sistêmica, pois todas as atividades de todos os casos de uso são analisadas. Por exemplo, a ameaça A4 indica que após o cliente ter fornecido informação pessoal, o gerente pode disseminar ilegalmente tal informação.

Uma vez que as ameaças foram levantadas, as políticas de segurança apropriadas devem ser selecionadas para impedir/controlá-las. Tal atividade pode ser apoiada por uma lista abrangente de políticas de segurança (GOLLMANN, 2005). Entretanto, é desejável que tal seleção resulte num conjunto mínimo de mecanismos, ao invés de mecanismos empilhados, simplesmente por parecerem úteis[2]. Por exemplo, para mitigar as ameaças A5 e A6 mostradas na Figura 16, pode-se selecionar a política "segregação de funções". Outros exemplos podem também ser encontrados na Figura 16.

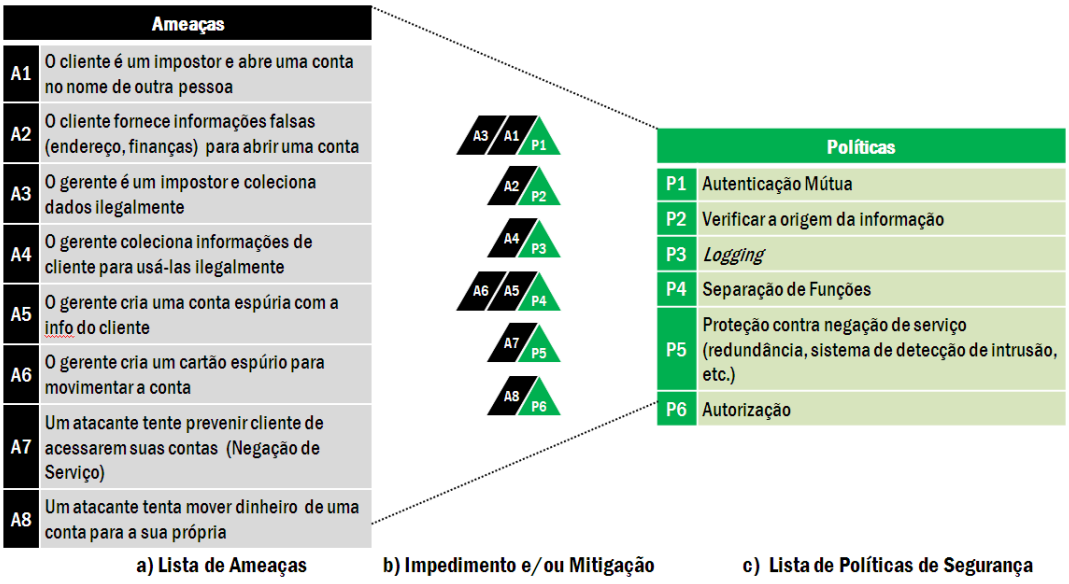

Figura 16: Identificação das Políticas para Bloquear/Mitigar Ameaças.

Apesar da abordagem "atividades de abuso" mostrar uma forma sistemática de identificar e diagramar ameaças, percebe-se uma lacuna quanto aos detalhes de sua aplicação, que fica mais

[2]Cabe ressaltar que o empilhamento de mecanismos pode ser adequado para sistemas de missão crítica, pois sua confiabilidade pode exigir controles cascateados.

evidente quando algumas questões, como as seguintes, são colocadas:

- Quais os critérios-base para se analisar uma atividade, de forma a caracterizar a ameaça?

- Como partir da ameaça e chegar à política de segurança ou ao mecanismo para controlá-la?

5.3 Extensão da Abordagem "Atividades de Abuso"

De forma a preencher a lacuna referida na seção anterior para instrumentalizar a abordagem "Atividades de Abuso", duas novas dimensões, além da dimensão *casos de uso*, apresentada em (FERNANDEZ et al., 2006b), foram adicionadas no escopo da análise (BRAZ; FERNANDEZ; VANHILST, 2008). A primeira consiste em categorias de ataques derivadas do modelo *Spoofing, Tampering, Repudiation, Denial of Service, and Elevation of Privilege* (STRIDE) (HOWARD; LEBLANC, 2002) e a segunda diz respeito à origem da ameaça (COLE; RING, 2005).

Foi realizada uma adaptação no modelo STRIDE para ele ficasse mais adequado ao momento no ciclo de vida de desenvolvimento de *software* em questão, visto que sua definição original pressupõe a existência de informação disponível apenas em estágios avançados da fase de projeto. Analisar "Elevação de Privilégio", cuja letra "E" se refere, mostra-se inviável na fase de requisitos. Cumpre observar, ainda, que essa categoria de ataque é, de fato, uma maneira de alcançar outros ataques e não um tipo de ataque em si. Em cada atividade deve-se verificar a possibilidade dos seguintes tipos de ataques:

- *Spoofing*: declarar uma identidade falsa quando executando uma atividade.

- *Tampering*: mudanças não autorizadas na informação relacionada à atividade (objeto de fluxo).

- *Repudiation*: realizar alguma atividade sem ser responsabilizado.

- *Information Disclosure*: leitura não autorizada da informação relacionada à atividade (objeto de fluxo).

- *Denial of Service*: tornar a atividade ou sua informação indisponível a um usuário autorizado.

A origem da ameaça remete aos privilégios de posse do "atacante" para executar o ataque. A relevância deste aspecto tem sido crescente, a partir do surgimento de relatórios que evidenciam a alta quantidade e impacto dos ataques empreendidos por internos à organização, como, por exemplo, (RANDAZZO et al., 2004). Os atacantes podem ser categorizados em: externo, formado por usuários sem autorização para acesso a qualquer parte do sistema; interno autorizado, no qual usuários têm acesso ao sistema e para executar a referida atividade e interno não autorizado, aquele que possui acesso ao sistema mas não à função em questão (COLE; RING, 2005).

Deve-se escrutinar as atividades tendo como base a seguinte questão: "Qual ¡*tipo de abuso*¿ pode acontecer na ¡*atividade*¿ provocada por uma ¡*origem*¿ que comprometa o ¡*ativo*¿?". Nesta questão, o *tipo de ataque* corresponde a uma das categorias do STRIDE mencionadas anteriormente, a *atividade* refere-se a cada atividade do diagrama de atividades, a *origem* diz respeito à (origem da ameaça) e o *ativo* refere-se ao objeto de fluxo, como exemplo, tem-se o objeto "Conta2" como ativo ameaçado na Figura 15.

A Figura 17 mostra os resultados da aplicação da abordagem *atividades de abuso* ao diagrama de atividades da Figura 15. De forma a evidenciar as diferenças, as ameaças identificadas pela abordagem original em (FERNANDEZ et al., 2006a) estão destacadas. Como apresentado pela Figura 17, várias outras atividades de abuso foram descobertas, comparando-se com a aplicação da abordagem original, como, por exemplo, "A11 - Muda a informação de crédito do cliente resultando para aprovação de cadastro". Neste caso, a análise de violação de integridade realizada por um *interno autorizado* leva a este cenário de ameaça. Tal resultado se adere ao principal objetivo da abordagem, atividades de abuso, que é de suportar a análise de requisitos de segurança de maneira a não se esquecer nenhuma ameaça. Adicionalmente, quando se dá o tratamento de primeira classe às atividades (abuso) realizadas por usuários internos, pode-se lidar com a fonte mais volumosa de ameaças de maneira adequada. Ainda é possível que alguma ameaça não seja encontrada por esta abordagem, mas, em contrapartida, a evolução da abordagem mostra-se mais efetiva que a original, pois sistematiza em detalhes o levantamento, resultando na identificação de um maior número de ameaças.

		Atividades de Abuso				
Ator	**Atividade**	**#**	**STRID**	**Origem InA/InN/Ex**	**Descrição**	**Ativo**
Cliente	Fornecer Info. Pessoal	A1	R	InA	Negar ter aberto a conta.	Conta
		A2	**D**	**Ex**	**Inundação de requisições.**	**N/A**
		A3	I	Ex/InN	Escuta não autorizada.	Cliente
		A4	I	Ex	Revelação não autorizada do relacionamento do cliente com a instituição na tentativa de criar uma conta com sua info.	Cliente
		A5	**T**	**InA**	**Fornecer info inválida (financeira, endereço)**	**Cliente**
		A6	**S**	**InA**	**Fornecer info de outra pessoa (nome, endereço, CPF)**	**Cliente**
Gerente	Verificar Crédito	A7	R	InA	Negar ter modificado a informação de crédito do cliente.	Cliente
		A8	**I**	**InA**	**Coletar a informação pessoal do cliente para disseminar ilegalmente.**	**Cliente**
		A9	I	Ex/InN	Escuta não autorizada.	Cliente
		A10	**I**	**Ex**	**Coletar informação como um impostor.**	**Cliente**
		A11	T	InA	Mudar as info de crédito do cliente para conseguir mais clientes.	Cliente
Gerente	Criar Conta	A12	R	InA	Negar a criação de conta espúria.	Conta
		A13	I	InA	Coletar informação da conta do cliente para disseminar ilegalmente.	Conta
		A14	I	Ex/InN	Escuta não autorizada.	Conta
		A15	**T**	**InA**	**Criar uma conta espúria.**	**Conta**
Cliente	Efetuar Depósito Inicial		-	-	-	-
Gerente	Criar Autorização	A16	D	InA	Não emitir o cartão.	Cartão
Gerente	Gerar Cartão	**A17**	**T**	**InA**	**Criar uma autorização/cartão espúrio.**	**Cartão**
Cliente	Transferir valor	A18	R	InA	Negar a autorização de transferência financeira.	Conta
		A19	**D**	**Ex**	**Inundação de requisições.**	**N/A**
		A20	I	Ex/InN	Escuta não autorizada.	Cliente
		A21	**T**	**Ex**	**Transferir ilegalmente valores entre contas.**	**Conta**

Figura 17: Consolidação das Ameaças Levantadas na Figura 15

Cabe ressaltar a importância do *contexto* no qual o *software* está inserido. Ele pode levar a diferentes conclusões em termos de relevância e adequação de algumas ameaças. No exemplo da Figura 15, pode-se considerar que não seja possível a abertura remota de conta, o que significa que todo cliente que queira abrir uma conta deva se dirigir a uma agência bancária para fazê-lo. Neste caso, algumas ameaças podem deixar de existir, por exemplo, "A3 - Realiza escuta" e "A4 - Descobre relacionamento de cliente com instituição, pela tentativa de abertura de nova conta em seu nome".

5.4 Trabalhos Relacionados

Uma abordagem relacionada é o conceito de casos de abuso (ALEXANDER, 2003; SINDRE; OPDAHL, 2005). Os casos de abuso são casos de uso independentes iniciados por atacantes externos ao sistema. Tal abordagem, em si, mostra-se incompleta, pois não esclarece quais casos de abuso devam ser considerados. Outra abordagem relacionada é a análise de risco. Na análise de risco, ameaças à conclusão do sistema e ao uso do sistema são identificadas e analisadas. A probabilidade de ameaças e consequências é considerada a partir de uma análise de custo/benefício, e planos são elaborados para tratá-los. A análise de risco, em si, padece de um método de identificação sistemática de riscos, ela se concentra no efeito das ameaças no sistema.

A abordagem *casos de abuso* lida com a análise e definição de requisitos de segurança. O caso de abuso é uma adaptação do caso de uso, no qual o papel do ator é exercido por atacantes, permitindo que se concentre em levantar seus objetivos de exploração do sistema (ALEXANDER, 2003; SINDRE; OPDAHL, 2005). Enquanto esta abordagem pode ajudar na elicitação de requisitos de segurança em alto nível, não é apresentado como associar os casos de abuso a comportamentos legítimos e ativos concretos, consequentemente, não fica claro quais casos de abuso devem ser considerados, nem em qual contexto.

Se a modelagem de ameaças da Microsoft (SWIDERSKI; SNYDER, 2004), por um lado, fornece uma idéia clara sobre como elicitar, classificar, priorizar e mitigar ameaças e ainda possui ferramenta de apoio (Microsoft Corporation, 2006), por outro, sua aplicação demanda informação disponível apenas tardiamente na etapa de desenho, o que implica o início do desenho com enfoque em segurança em um momento já avançado no ciclo de desenvolvimento de *software*. A abordagem *atividades de abuso* pode ser aplicada desde o princípio, uma vez que requer apenas os casos de uso do sistema.

Jaatun e Tøndel (2008) apresenta uma proposta para elicitação de requisitos baseada na identificação e classificação de ativos de *software*, cuja sistemática se assemelha à apresentada pela modelagem de ameaças apresentada por Myagmar, Lee e Yurcik (2005), mas com menos detalhes. Entretanto, a identificação de ativos a partir do diagrama de atividades é mais efetiva e sistemática.

A metodologia desenvolvida por Jackson (2001), conhecida como *problem frames*, tem sido bastante usada em pesquisa envolvendo requisitos de segurança de *software*. Argumenta-se ser a alternativa corrente mais adequada para definição do problema a ser resolvido pelo sistema, com a premissa de que outras abordagens induzem à definição do problema pela solução, como, por exemplo, a UML.

Hatebur, Heisel e Schmidt (2008) descreve um processo de engenharia de segurança de *software* para desenvolver sistemas baseados em *problem frames*, a partir da coleção de padrões[3] de segurança, adicionado a componentes como uma maneira de lidar com a solução, enquanto a abordagem *problem frames* parece útil em alguns casos, mas não se compara à UML para desenho. Adicionalmente, ela é de difícil absorção dado o perfil corrente do desenvolvedor.

[3]Os padrões considerados são próprios da abordagem *problem frames* e não tem relação com os padrões abordados no Capítulo 4

Das abordagens baseadas em *Problem Frames*, a desenvolvida por (HALEY et al., 2008) se apresenta como a mais completa, pois oferece um *framework* para definição dos requisitos de segurança de software, considerando os pressupostos de segurança adjacentes (HALEY et al., 2006), e validação de que tais requisitos satisfazem os objetivos de segurança do sistema. Entretanto, sua dificuldade de adoção mostra-se elevada, uma vez que demanda ainda o conhecimento de linguagem formal na fase de validação. Cumpre ressaltar também que existe uma limitação conceitual, pois todo seu trabalho se fundamenta na definição do requisito de segurança de software como sendo "restrições nas funções do sistema", entretanto, alguns requisitos de segurança, como, por exemplo, auditoria, excedem o conceito de restrição.

6 *Classificação Multidimensional para Usuários de Padrões de Segurança*

Este Capítulo apresenta uma classificação para padrões de segurança que contemple a necessidade dos usuários. O método usa uma matriz definida a partir da divisão do espaço do problema por dimensões múltiplas e permite aos padrões ocuparem regiões definidas por múltiplas células na matriz. Ele suporta o filtro pela abrangência do padrão, permite a navegação pelos eixos de interesse e identifica lacunas no espaço do problema que não possui padrão que o contemple. Os resultados destacam os ganhos comparados com as classificações existentes (VANHILST; FERNANDEZ; BRAZ, 2009).

6.1 Matriz de Interesses

Para contemplar a classificação de padrões e o problema da cobertura, propõe-se o uso de uma matriz multidimensional de interesses. Cada dimensão da matriz é uma lista distinta de interesses, juntamente com um eixo único, com um conceito único e um conjunto de distinções que definem as categorias. As categorias de um eixo ou dimensão devem ser entendidas e representar classificações amplamente usadas e aceitas que dizem respeito ao conceito em questão. Por exemplo, uma dimensão seria uma lista de atividades do ciclo de vida, contemplando análise do domínio, requisitos, análise do problema, projeto, implementação, integração, implantação (incluindo configuração), operação, manutenção e descontinuidade.

A lista de tipos de componentes-fonte forma outra dimensão. Componentes podem originar-se de código novo, código aberto, scripts de tempo de execução, transformação de modelos, geração de código automática, legado, biblioteca reutilizável, desenvolvimento terceirizado, produto de prateleira e *webservice* remoto. Tipos de resposta podem formar ainda uma terceira dimensão, contemplando: contenção, dissuasão, prevenção, detecção, mitigação, recuperação, investigação (forense).

As células nas intersecções entre duas ou mais dimensões representam uma preocupação que seja mais específica do que poderia ser expressa por uma lista de classificação em uma única dimensão. Por exemplo, com duas dimensões podemos mirar em padrões de segurança para requisitos quando usando COTS ou componentes de terceiros. Similarmente, pode-se mirar padrões de segurança para análise e projeto usando *webservices*, e para estes, mais especificamente, padrões que enderecem detecção e recuperação.

O projeto da matriz é motivado pela noção de cobertura de interesses, na qual uma cobertura abrangente denotaria a existência de exemplares para cada célula de cada intersecção de cada dimensão. Desta forma, destaca-se a preocupação, não somente com os padrões que existem, mas também na identificação das lacunas onde não existem padrões.

O método inicia-se com o espaço completo do problema, que é dividido a partir de interesses diferentes, juntamente com as diferentes dimensões. A idéia de dividir o espaço psicológico pode ser associada aos elementos Euclidianos. Seu uso neste contexto é suportado pelas idéias de Kelly (1955). Na teoria do construto pessoal de Kelly, um construto é um eixo de referência de dois pólos opostos. Riqueza, por exemplo, é um eixo de rico e pobre. O espaço entre os pólos define a faixa de conveniência que adquire maior relevância com planos adicionais de distinção. "Um construto é um eixo de referência dicotômico. Ele define uma família de planos ortogonais que dividem o espaço" (SHAW M. L. G., 1992). Para o método aqui exposto, não é relevante os planos de distinções, que cria as separações, bem como os espaços

entre dois planos, que fornece a conveniência da classificação. Kelly descreve uma matriz de conceitos que incorpora as intenções de uma pessoa e molda sua resposta - o que ele denomina "rede de repertório do papel". Um tratamento mais formal da divisão do espaço psicológico pode também ser encontrado em Brown (1971).

Registra-se aqui o esforço para definir uma matriz de conceitos que incorpore a necessidade de proteção da informação e o uso da matriz para classificar padrões. Na definição das dimensões ou eixos para classificação, cada eixo deve corresponder a um construto lógico isolado. No modelo de Kelly, um construto é definido pelos pólos dicotômicos. Este método aspira o mesmo. Cada eixo ou dimensão é dividida em regiões de interesse. As regiões podem ser definidas fracamente e serem hierárquicas, disjuntas e sobrepostas. As regiões ou classificações de interesse dentro de uma dimensão podem ser baseadas nas distinções que são facilmente ou bem entendidas pelos usuários-alvo - neste caso, cientistas da computação, analistas de segurança etc. A definição de regiões é como definir interesses a partir da decomposição de cima para baixo. Na lógica, a distinção define tanto o que está incluído como o que não está (BROWN, 1971). Em contraste, quando uma classificação começa com uma coleção conhecida de itens, mas desestruturada, e os colocam em grupos, não existe uma maneira de saber o que está faltando.

Os usuários-alvo deste trabalho cobrem uma faixa de interessados em segurança. Adicionalmente, aos desenvolvedores de *software*, considera-se as necessidades e perspectivas de gestores de segurança da informação com maior perspectiva do sistema, escritores de padrões que buscam lacunas a serem contempladas e um contexto mais abrangente com o qual visualizar seus padrões, além de alunos à procura de expandir seu conhecimento da variedade de questões e perspectivas de segurança.

Os desenvolvedores que usarem a matriz podem identificar seu foco corrente ou interesse pela escolha do elemento aplicável, ou uma faixa de elementos, juntamente com cada dimensão,

e então localizar os padrões que caiam na intersecção de todas. Nesse sentido, não é diferente que rotular padrões com chaves identificadoras e usar tais chaves para a busca. Mas existem algumas importantes diferenças. Primeiro, a partir da definição de rótulos em uma sequência em um espaço com *n* dimensões, o desenvolvedor consegue navegar para juntar regiões do espaço para melhor contextualização e compreensão. Segundo, observando o número de células que um padrão cobre, e a região que ele representa, desenvolvedores obtêm a percepção não somente sobre o grau de generalidade, mas também sobre o tipo de generalidade que o padrão envolve. Terceiro, ao se observar regiões e células nas quais os padrões estão ausentes, escritores de padrões podem identificar lacunas que ainda precisam ser contempladas.

O uso da matriz de interesses não é dependente de nenhuma metodologia. Desenvolvedores, a partir de qualquer abordagem, podem usar o conceito de matrizes para melhor identificar tarefas e interesses e localizar padrões relevantes para um interesse específico. O método escolhido orienta na sequência e agendamento de certas atividades, enquanto a matriz fornece uma orientação para um conhecimento mais específico ou idéias sobre como realizar tal atividade. Essa abordagem é consistente, por exemplo, com a noção dos pontos de controle de McGraw, apresentada na seção 2.2.2.3.

A matriz pode ser facilmente estendida para adaptar às novas tecnologias e interesses, ou focar em uma necessidade particular. Novas instâncias podem ser adicionadas junto às dimensões existentes ou dimensões inteiramente novas. A extensão da matriz não obsolesce matrizes anteriores - elas apenas não incluem tantas distinções.

6.2 Dimensões Primárias

As dimensões primárias são geralmente úteis e deveriam ser consideradas em qualquer classificação. Foram introduzidas na matriz exposta anteriormente mais três dimensões:

camada de arquitetura, nível de restrição e domínio. Essas seis dimensões são mostradas pela Figura 18. A classificação "todos" é usada quando as distinções ao longo do eixo não possuem relevância para o padrão. Consideramos também dimensões secundárias que identificam ou destacam interesses adicionais que podem ser úteis em alguns casos, mas não necessitam estar em foco para outros. Mais adiante, introduzimos ainda um conceito de dimensões auxiliares. Uma dimensão auxiliar apresenta uma lista de interesses que podem ser úteis para graduar ou posicionar padrões, como um checklist, mas não divide o espaço junto com uma sequência para preocupações de navegação ou subdivisão de tarefas.

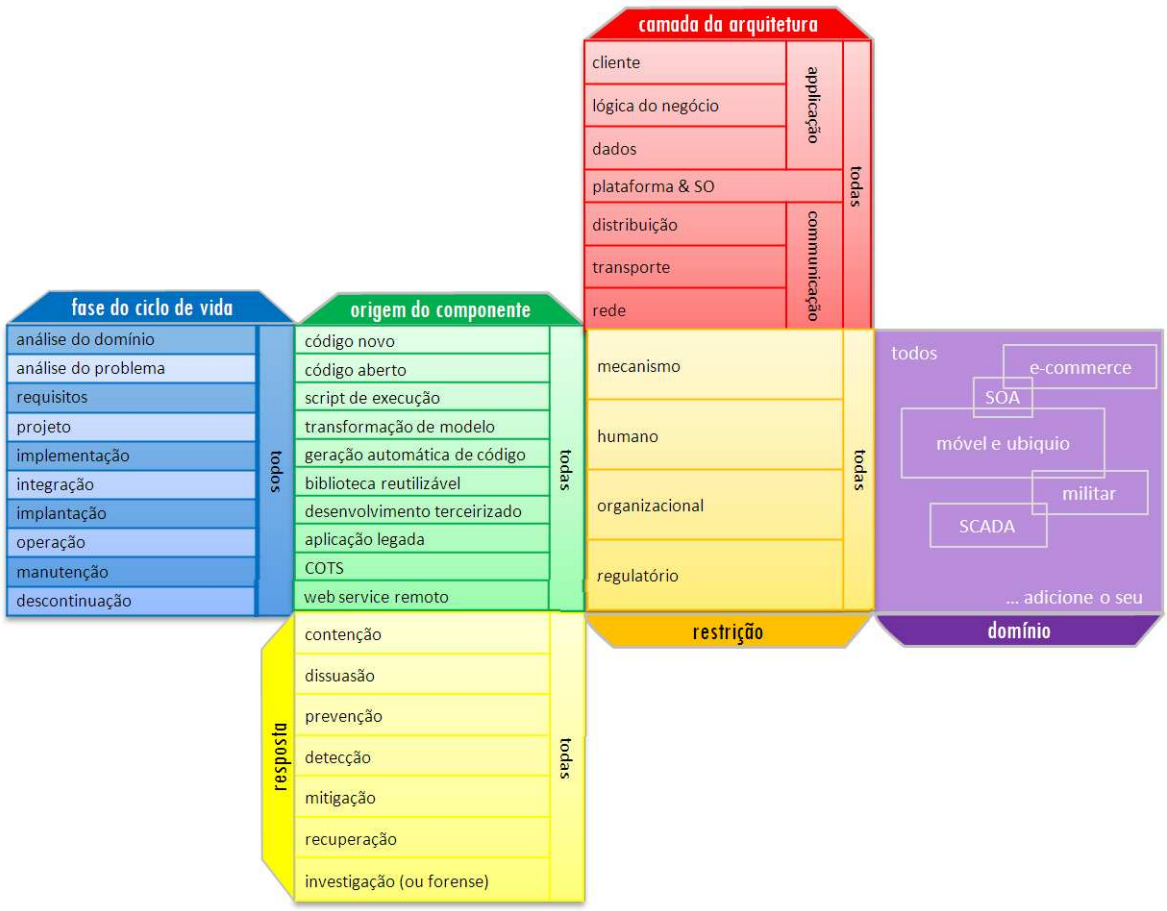

Figura 18: Seis classificações primárias, mostradas como um cubo.

A classificação no eixo para a fase do ciclo de vida é ordenada de acordo com a dicotomia da inicialização e do encerramento. Ela inicia de fato com um pré-início ou potencial, no qual a análise de domínio se encontra, e poderia terminar em pós-encerramento com descontinuidade. As categorias dos componentes-fonte são organizadas ao redor da dicotomia de controle interno

versus externo. Em "códigos novos", o desenvolvedor possui controle completo sobre todos os detalhes. Em "serviço remoto" existe pouco, ou ausência de, controle ou até conhecimento dos detalhes e habilidade limitada de verificação. O eixo resposta baseia-se na possibilidade do ataque acontecer ou não e na sua extensão, partindo-se do estágio "contenção", até aquele que trata do ataque bem sucedido "forense".

A camada de arquitetura oferece outra dimensão muito útil, uma vez que os problemas e suas soluções em diferentes camadas da arquitetura diferem. A grosso modo, a dimensão de arquitetura foi dividida em diferentes formas para protocolos de comunicação, sistemas de negócio e ambientes de execução, mas sempre com uma ordenação de baixo a alto nível de abstração, e de rede para plataforma para aplicação. O eixo que engloba este conceito parte do nível de abstração correspondente ao bit sem significado até as semânticas das tarefas fim (negócio). A combinação de elementos de múltiplos domínios, e permitindo a sobreposição entre visões, foram escolhidas as seguintes distinções: rede, transporte, distribuição (incluindo-se *gateways* e *brokers*), plataforma e sistema operacional, dados, lógica de negócio e cliente. Qualquer aplicação, por mais simples que seja, consegue englobar as últimas três. Rede, transporte e distribuição podem também ser agrupados como comunicação. Uma vez que os padrões podem se colocar em mais de uma célula, não existe uma necessidade real para uma classificação exata ou disjuntiva.

Leveson (2004) define quatro níveis de restrições: via mecanismo, humana (operador ou desenvolvedor), organizacional e regulatória. Cada nível de restrição atua em um importante papel em falhas de segurança e nas suas prevenções. Por extensão, foram usados os mesmos níveis para segurança com um eixo com níveis partindo de coisas para a sociedade. Enquanto a maioria dos padrões de segurança descreve mecanismos, o *National Training Standard for Information Systems Security* (INFOSEC) (National Security Agency, 1994) concentra-se em práticas, políticas e regulação. O *Common Criteria* possui requisitos funcionais que se aplicam no nível de mecanismos (COMMON CRITERIA SPONSORING ORGANIZATIONS, 2006a). Entre-

tanto, ele possui também requisitos de garantia de interessados nos processos organizacionais para documentar ações realizadas (COMMON CRITERIA SPONSORING ORGANIZATIONS, 2006c). O desenvolvimento de um plano de gerência de configuração é um requisito de garantia comum no *Common Criteria* que se aplica ao nível organizacional no ciclo de vida de análise de domínio. O *Common Criteria* e outros padrões, como o *Sarbanes-Oxley Act* (SOX) e SSE-CMM, em si, pertencem ao plano regulatório.

O domínio da aplicação pode fornecer um diferencial ou filtro importante para reduzir o campo do conhecimento aplicável. Algumas soluções são específicas para um domínio particular ou tipo de aplicação. Computação ubíquia, comércio eletrônico e *Supervisory Control And Data Acquisition* (SCADA) estão diante de desafios de segurança distintos. Por exemplo, a segurança para uma aplicação de negócio estruturada em três camadas pode se diferenciar de soluções de sistemas SCADA de sensores e controles. Este eixo é uma exceção, pois não apresenta dicotomia ou ordenação, o que faz com que o espaço seja definido livremente. As organizações podem criar padrões para seus próprios domínios como uma forma de capturar o conhecimento.

6.3 Dimensões Secundárias e Auxiliares

Não foi incluída a fase de testes no ciclo de vida, pois ela representa um interesse ortogonal que se aplica a todas as fases. Mas as fases do ciclo de vida, em contrapartida, podem ser subdivididas pela aplicação de outra dimensão, com elementos para preparação, execução, validação/verificação, e transição (para outra fase). A granularidade adicionada desta dimensão secundária não somente cria um local para teste e verificação formal, mas, para propósitos de cobertura de verificação, eleva todas as distinções nessa dimensão, e seus interesses, em cada uma das fases do ciclo de vida do software.

As coleções de práticas de segurança frequentemente incluem uma lista de princípios de segurança, como o princípio do *menor privilégio*. Viega e McGraw (2001) usa uma lista de 10 princípios de segurança, enquanto que em (STEEL; NAGAPPAN; LAI, 2005) existem 12. A *Open Web Application Security Project* (OWASP) lista 15 princípios, bem como 10 princípios de segurança de código, 20 pontos fracos ou vulnerabilidades e 12 contramedidas (Open Web Application Security Project, 2008). Tais listas não dividem de fato o espaço do problema. Mas elas podem prover uma dimensão auxiliar para classificar padrões, baseados na quantidade e quais princípios se aplicam ou contemplam.

Um perigo na composição de soluções pontuais ocorre na interface entre componentes da solução. Por exemplo, em um sistema heterogêneo, algumas partes podem ser .NET, enquanto outras em J2EE. Códigos novos podem se "interfacear" com sistemas legados ou *webservices* intercambiáveis. Em uma dimensão diferente, os subsistemas podem se formados pela combinação de código de terceiros com código legado. Problemas particulares de segurança ocorrem na interface na qual os dois interagem e coexistem. A matriz pode ser usada para isolar e documentar problemas de interface pela replicação de um eixo ortogonal a si mesmo. A fatia de duas dimensões resultante é análoga a um gráfico de milhagem, com listas de cidades nos dois eixos. Elementos do gráfico representam interfaces entre os componentes correspondentes: de terceiro com legado, legado com *webservice*, .NET com J2EE etc.

6.4 Uso

A matriz vem sendo aplicada para a classificação de padrões de segurança para *Service Oriented Architecture* (SOA), cujas observações são registradas nesta seção.

Os padrões são mapeados da mesma maneira na qual eles seriam rotulados. Cada dimensão é considerada separada. Os padrões são identificados no ponto de cada dimensão, e então na

matriz, na qual o seu conteúdo afete decisões que serão realizadas. Caso uma distinção de uma dimensão não seja relevante, por exemplo, caso o padrão não seja específico para nenhum domínio, mas aplicável a todos, então sua classificação naquela dimensão é "qualquer" ou "todas". Como exemplo, os antipadrões definidos por Kis (2002) se aplicam no nível do desenvolvedor de restrição na fase de requisitos. Alguns padrões se referem a componentes legados, enquanto que para outros a fonte do componente é irrelevante. Em um teste com diferentes membros da equipe do System Security Group da Engenharia da Computação da Universidade Flórida Atlântica, que classificaram independentemente seis padrões do Open Group, e (HAFIZ; ADAMCZYK; JOHNSON, 2007), os resultados mostraram pequenas diferenças em cinco dos eixos primários. O domínio foi a exceção, na qual se provou ser difícil se atribuir faixas de aplicabilidade.

A Figura 19 mostra a classificação do padrão "Avaliador de Controle de Acesso" *eXtensible Access Control Markup Language* (XACML). O padrão descreve um mecanismo que é aplicado na fase de projeto no desenvolvimento. O tipo de resposta é prevenção, uma vez que o ataque acontece e é repelido. O controle é usado no nível de distribuição da arquitetura do sistema. A origem do componente não é relevante, uma vez que ele se aplica a todas as classificações por esse eixo. O padrão é específico para o domínio de comércio eletrônico e SOA, no qual o XACML é usado. Uma vez que ele descreve um componente da solução da arquitetura, uma classificação em um eixo adicional para interesse da solução pode ser útil.

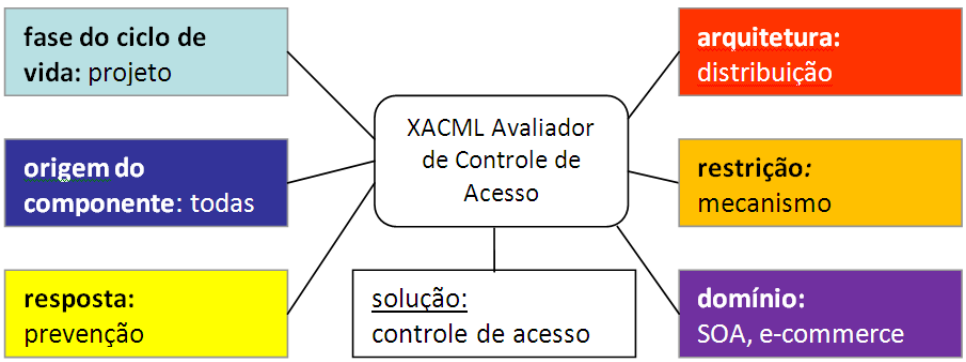

Figura 19: Classificação do Padrão Avaliador de Controle de Acesso XACML .

A Figura 20 ilustra um mapeamento dos padrões de projeto em duas fases do ciclo de vida e em níveis diferentes de arquitetura. Somente uma amostra pequena de padrões é apresentada. Enquanto todos os padrões na Figura 20 são aplicáveis à SOA, alguns se aplicam mais genericamente a outros domínios também. Os padrões estão agrupados dentro de "Projeto" juntamente com uma dimensão secundária com "Filtro", "Controle de Acesso" e "Autenticação". Na Figura 20 estão os padrões da fase de análise de domínio, na qual o desenvolvedor deve encontrar padrões que explicam os padrões do domínio e tecnologias, posteriormente usadas na fase de projeto. Um desenvolvedor pode navegar para juntar células da fase de análise (não contemplado) para encontrar padrões genéricos relacionados a filtro, controle de acesso e autenticação. Enquanto os padrões são encontrados nessas posições na matriz, o entendimento de seu papel no sistema e como eles se relacionam uns com os outros ainda exige um diagrama de linguagem de padrões e outras ferramentas e métodos para sua aplicação.

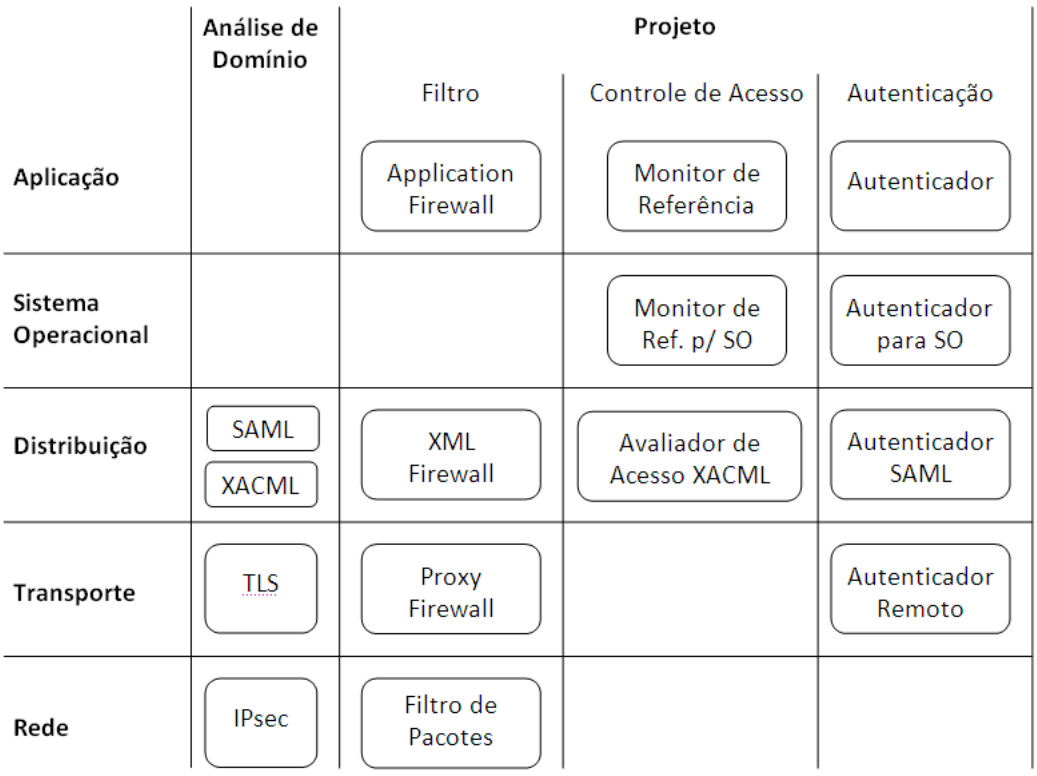

Figura 20: Amostra de padrões classificados em um extrato da matriz em duas dimensões.

Muitos dos padrões observados estão registrados também em (STEEL; NAGAPPAN; LAI, 2005). Steel, Nagappan e Lai (2005) agrupou seus padrões somente de acordo com as

camadas em uma arquitetura em quatro camadas, enquanto que, aqui, foram aplicadas outras distinções. Um número de diferenças pode ser observado. Obrigando-se cada padrão a ocupar somente uma célula, informação importante é perdida ou distorcida. Por exemplo, em (STEEL; NAGAPPAN; LAI, 2005), o padrão *CheckPoint* de Yoder e Barcalow foi identificado como um padrão cliente (Web) e mesclado com o padrão *Checkpointed System* da Open Group como um grupo de padrões de *checkpoint* na mesma célula. Na classificação apresentada neste trabalho, o *checkpoint* é um padrão da fase de análise aplicável a qualquer das três camadas de aplicação e endereça uma abordagem genérica de prevenção, enquanto o *Checkpointed System* é um padrão da fase de projeto aplicável à camada cliente e contempla um mecanismo de recuperação. Os dois padrões são muito diferentes.

Neste trabalho, não foram contempladas questões de visualização. Os métodos usuais de redução de informação multidimensional como estreitamento (*narrowing*), achatamento (*flattening*) e projeção (*projection*) - como em planilhas eletrônicas e *on-line Analytical Processing* (OLAP) - podem ser aplicados. Em seu trabalho de dividir o espaço de arquitetura corporativo Trowbridge et al. (2004), membros do grupo Microsoft's Patterns and Practices restringiram a si mesmos a duas dimensões devido ao problema de visualização. A preocupação-alvo é cobertura do espaço do problema e sobre o potencial pela perda de informação - especialmente perda de informações sobre as lacunas.

6.5 Trabalhos Relacionados

Listas são frequentemente usadas na áreas de segurança. O *Department of Defense* (DoD) e *National Institute of Standards and Technology* (NIST) mantêm conjuntos de listas de verificação para configuração segura de várias aplicações de *software*, enquanto o *Computer Emergency Response Team/Coordination Center* (CERT/CC) e NIST listam mais de 24.000 ataques em software divulgados. O *Common Criteria* (COMMON CRITERIA SPONSORING

ORGANIZATIONS, 2006a) e SSE-CMM (ISO/IEC 21827) (SYSTEMS..., 2003), ambos incluem listas de áreas de garantia que devem ser documentadas para satisfazer à certificação. Hoglund e McGraw (2004) listaram 49 tipos de ataques em *software* (Hoglund and McGraw, 2004). A Microsoft tem produzido listas de furos, árvore de ataques e processo de desenvolvimento seguro (HOWARD; LEBLANC, 2002; HOWARD; LIPNER, 2006), vide seção 2.2.2.1 do capítulo 2. Em contraste a esta vasta lista de interesses heterogêneos, cada dimensão da matriz apresentada abrange uma faixa mais coesa e concisa de conceitos com partições reconhecidas.

Schumacher definiu uma metodologia para projeto de sistemas seguros usando padrões de segurança (SCHUMACHER; ACKERMANN; ATEINMETZ, 2000; SCHUMACHER, 2003). Assim como neste trabalho, eles propuseram a aplicação da segurança em todas as fases do ciclo de vida do *software*. Propuseram o uso de uma base de dados de vulnerabilidade para manter o registro de possíveis ataques e contramedidas, mas não oferecem detalhes sobre como aplicar a segurança em todas as fases do desenvolvimento, nem como verificar a cobertura dos interesses.

Trowbridge et al. (2004) descreveu uma "tabela de organização" para organizar padrões e identificar lacunas e incluiu uma discussão sobre a identificação de relacionamentos pela exploração de células adjacentes, mas limitou-se em duas dimensões heterogêneas para pontos de vistas e interrogações. Seus 5 pontos de vista são divididos em arquiteto, projeto e desenvolvedor. Os 110 padrões classificados situam-se somente em duas categorias: integração e aplicação, e três interesses: dados, função e rede.

Hafiz, Adamczyk e Johnson (2007) identificou quatro dimensões potenciais de classificação: tipo de proteção, contexto da aplicação, ameaça e os pontos de vista de Trowbridge et al. (2004). Como conclusão, ele propôs uma hierarquia baseada na aplicação do contexto orientado pela ameaça. A perspectiva do colecionador evidenciou seu esforço para enquadrar cada padrão e sua preocupação, de forma que muitos padrões situem-se em duas poucas células. No artigo, existe a declaração: "Qualquer esforço organizacional deve se

iniciar pela coleção de itens a serem organizados". Esta noção é rejeitada aqui, e começa com o espaço a ser contemplado.

Fernandez et al. (2008) classificou padrões baseados nos níveis de arquitetura e interesses. Por exemplo, controle de acesso definido na aplicação é refletido na base de dados do sistema operacional. Os níveis de arquitetura e interesses de segurança são duas possíveis dimensões na matriz definida.

German e Cowan (2001) classificou os padrões de interface do usuário em uma hierarquia da fase de domínio, elaboração, geral para específico e valor (um tipo de graduação do Google). Suas categorias de elaboração ordenadas fazem sentido para projeto de interface de usuário, mas não são relacionadas à segurança. Evidencia-se, a partir de seu exemplo, que a matriz deveria ser estendida caso haja a intenção de classificar padrões em domínios diferentes de segurança.

Deve ser observado que um espaço multidimensional pode ser alinhado com as divisões celulares em (TROWBRIDGE et al., 2004), (HAFIZ; ADAMCZYK; JOHNSON, 2007) e mesmo em (GERMAN; COWAN, 2001), sem agrupamento hierárquico. Da perspectiva do usuário, a redução de classificações ortogonais para uma única hierarquia alcança quase nada e esconde a exploração de relacionamentos nas diferentes dimensões.

7 *Protótipo de Suporte à "Atividades de Abuso" e Definição de Padrões para seu Controle*

Este capítulo aborda detalhes do protótipo desenvolvido para dar suporte à realização da análise de *atividades de abuso*, bem como escolha do padrão de segurança mais apropriado para conter a ameaça levantada.

7.1 Especificação

A ferramenta se classifica como um sistema de informação que incorpora aspectos de um sistema de apoio à decisão baseado em regras. Seu objetivo é apoiar a execução da técnica *atividades de abuso* para especificação de requisitos de segurança de *software*, cujos fundamentos encontram-se detalhados no capítulo 5, bem como a identificação do padrão de *software* que controle as ameaças identificadas, que remete ao método apresentado no capítulo 6. As funcionalidades da ferramenta podem ser resumidas nos casos de uso apresentados pela Figura 21.

Os casos de uso da Figura 21, cujo nome se inicia com CRUD possuem funções de natureza idêntica, pois são responsáveis pela manutenção das informações relativas a cada elemento, conforme especificado a seguir:

- CRUD *Object Flow* - mantém as informações dos objetos de fluxo dos diagramas de atividades.

- CRUD *Activity* - mantém as informações das atividades dos diagramas de atividades.

- CRUD *Activity Diagram* - mantém as informações dos diagramas de atividades.

- CRUD *Actor* - mantém as informações dos atores dos diagramas de atividades.

- CRUD Misuse Activity - mantém as informações das atividades de abuso.

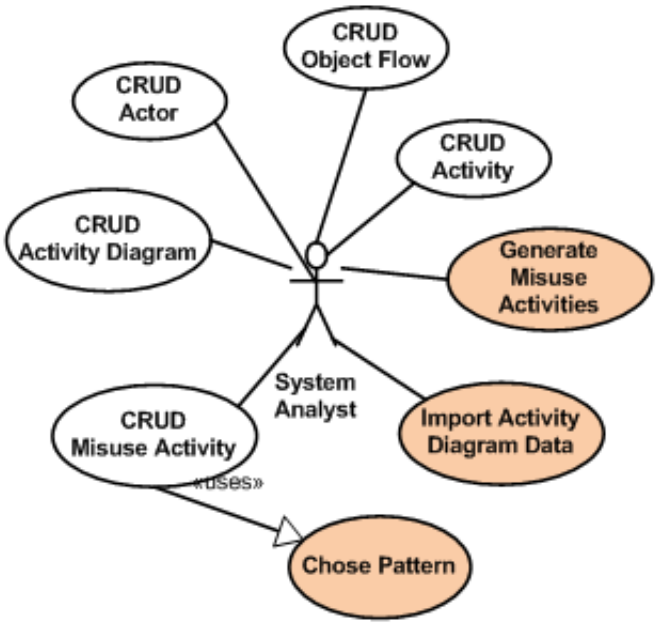

Figura 21: Diagrama de Casos de Uso para a Ferramenta de Apoio a “Atividades de Abuso”.

Já os casos de uso em destaque na Figura 21 possuem operações completamente distintas entre si e aos casos de uso mencionados anteriormente. Seguem suas descrições:

Import Activity Diagram Data Seu objetivo é importar para a ferramenta as informações para aqueles diagramas de atividades que estejam descritos no formato *XML Metadata Interchange* (XMI)[1].

Generate Misuse Activities Incorpora a orientação para levantamento de atividades de abuso descritas na seção 5.3 do capítulo 5 e gera automaticamente uma combinação de atividades de abuso para cada atividade de cada diagrama de atividades existente na ferramenta.

Choose Pattern Este caso de uso é responsável pela sugestão do padrão que controla a atividade de abuso levantada, a partir da aplicação do método de classificação de padrões detalhada no capítulo 6.

O *software* ainda contempla as funções básicas de armazenamento e recuperação das informações geradas pela análise, incluindo-se aquelas originadas no XMI, em arquivo no formato *eXtended Markup Language* (XML), cuja extensão é *Misuse Activity Analysis* (MAA).

As seções seguintes detalham os casos de uso destacados na Figura 21.

7.2 Import Activity Diagram Data

A manipulação das informações de diagrama de atividades e qualquer outro da UML exige uma estrutura de dados que implemente as especificações do seu metamodelo (Object

[1]O XMI é um formato de arquivo com sintaxe e semântica definidas pela *Object Management Group* (OMG), adotado por algumas ferramentas de análise e projeto baseados na *Unified Modeling Language* (UML), o que facilita a interoperabilidade entre elas. Exemplos:
- Poseidon - `www.gentleware.com`
- Magic Draw - `www.magicdraw.com`

Management Group, 2005), caso contrário, o potencial semântico da linguagem pode ser comprometido. Existem alguns componentes de terceiros que oferecem a referida estrutura, como, por exemplo, Novosoft UML library (NOVOSOFT, 2002) e nUml (NUML, 2007).

Pela familiaridade do autor desta tese com a plataforma Microsoft .NET, optou-se pelo uso do componente nUml. Esse componente é uma biblioteca que implementa o metamodelo da UML versão 2.0. Ele opera com Microsoft(R) .NET Framework, Mono::, e DotGNU e ainda suporta serialização para XMI versão 2.1.

Apesar da gama de possibilidades que a nUml viabiliza, como, por exemplo, a transformação de modelos (BROWN, 2004), no que tange esta tese suas funções podem ser resumidas em:

- Estrutura de dados - Sua estrutura estática, representada pelas suas classes e relacionamentos, permite manipular toda a informação relacionada ao diagrama de atividades, bem como de outros diagramas de UML. A Figura 22 apresenta a estrutura de classes fundamental para descrever o diagrama de atividade. Essa estrutura fundamental, bem como todas as outras especificadas pelo (Object Management Group, 2005), encontram-se disponíveis em (NUML, 2007).

- Desserialização - Esse método compreende a interpretação do arquivo estruturado no formato XMI, e sua consecutiva representação em objetos, respeitando a estrutura estática definida em (NUML, 2007). Uma vez desserializado o diagrama de atividades, toda sua informação pode ser processada via código.

Cabe ressaltar que, na forma como se encontra o protótipo, uma vez desserializado o arquivo XMI, as informações contidas nos objetos são transferidas para um modelo próprio da

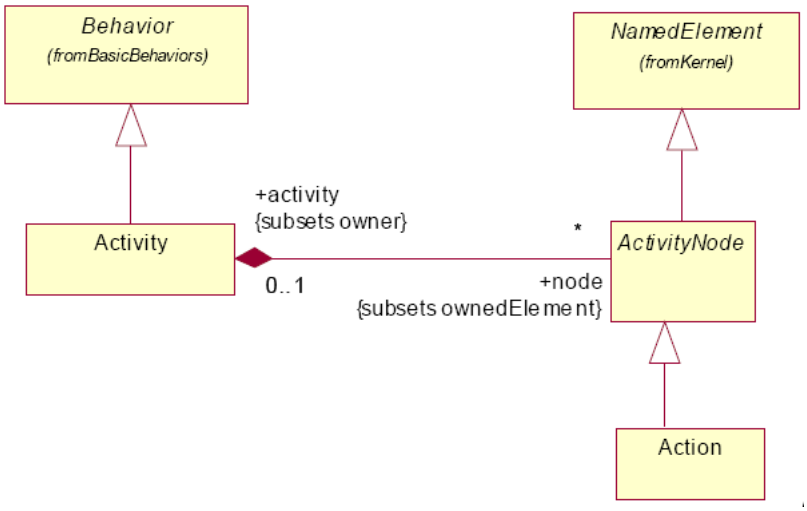

Figura 22: Conjunto fundamental de nós que viabilizam a manipulação de informação do diagrama de atividades. Fonte: (Object Management Group, 2005)

.

ferramenta, cujo fragmento de interesse encontra-se na Figura 23, decorre daí o nome desse caso de uso, que se refere explicitamente à importação.

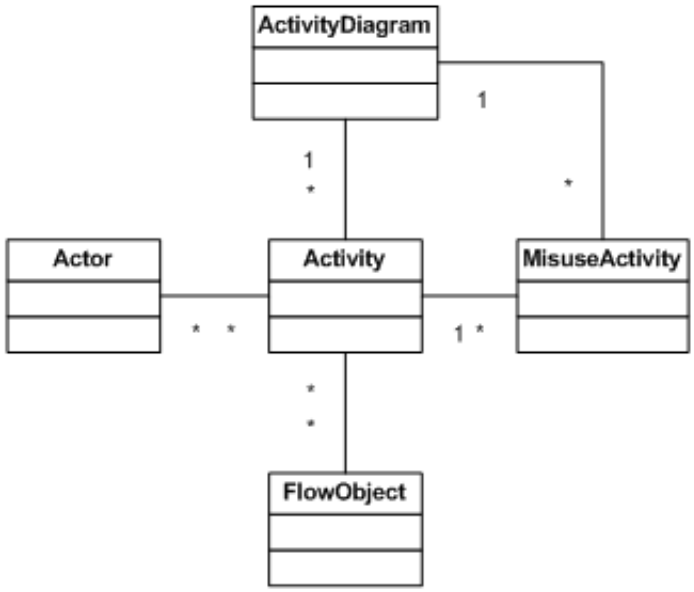

Figura 23: Modelo Usado no Protótipo para Representação da Informação do Diagrama de Atividades.

.

7.3 Generate Misuse Activities

Uma vez registrada qualquer atividade de um diagrama de atividades, o protótipo permite que sejam geradas automaticamente as ameaças à atividade.

Essa operação obedece à regra de multiplicação de matrizes, vide Figura 24, na qual *A* corresponde ao conjunto de atividades, *B* ao tipo, *C* à origem do de abuso e *D* à atividade de abuso.

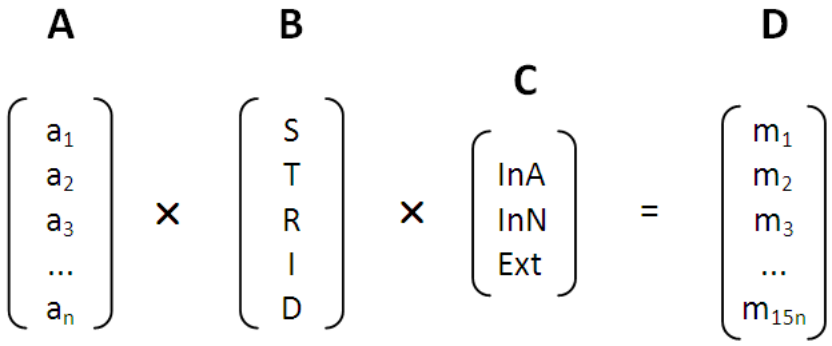

Figura 24: Produto de Matriz Usada para Geração Automática das Ameaças. Informação do Diagrama de Atividades.

.

A geração automática de ameaças provocará a reflexão do analista sobre a subversão na qual cada uma das atividades dos diagramas de atividades está submetida, ficando a seu critério eliminar os falsos-positivos, ou seja, as indicações incompatíveis com o contexto do *software* objeto de sua análise. Um exemplo do uso desse caso de uso pode ser observado na Figura 25, que possui um fragmento da tela do protótipo após a geração automática das atividades de abuso.

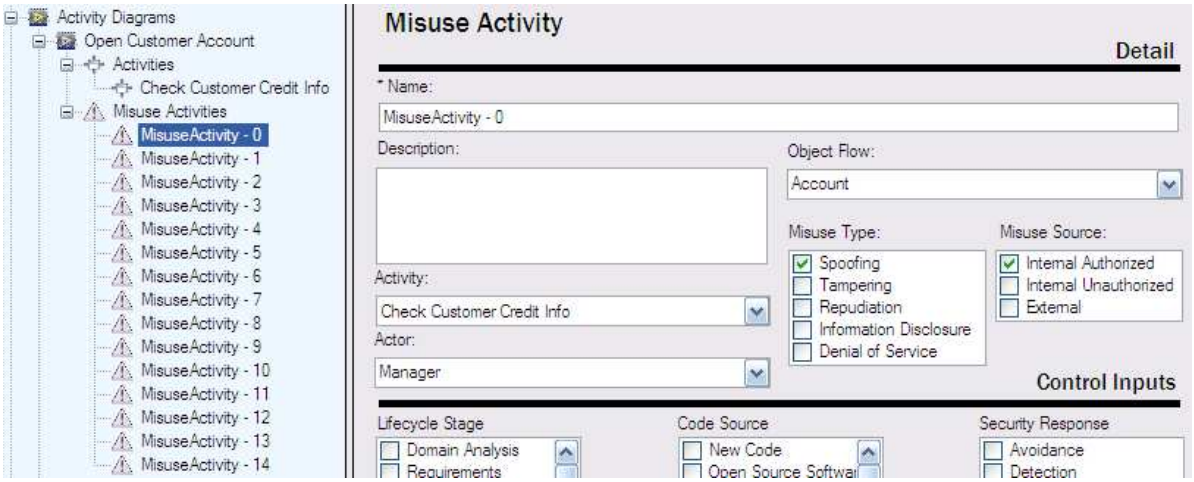

Figura 25: Interface Gráfica do Protótipo após a Geração Automática das Atividades de Abuso.

.

7.4 Choose Pattern

Para controle das ameaças levantadas, são utilizados padrões de *software* como elemento norteador das ações do analista. De fato, a expectativa para o caso de uso descrito nesta seção é que, a partir de sua realização, o analista tenha como retorno o padrão mais adequado a ser usado para controlar a ameaça em questão.

O módulo responsável pela recomendação de padrões se fundamenta no conceito de sistemas especialistas, com a incorporação do método de classificação de padrões, detalhado no capítulo 6, bem como as dimensões do escopo de análise da atividade de abuso detalhada no capítulo 5.

Um sistema especialista é um sistema baseado em conhecimento, isto é, um programa de computador inteligente que utiliza conhecimento e procedimentos de inferências com o objetivo de resolver problemas de dificuldade que requerem uma perícia humana. Os conhecimentos necessários para se chegar à solução, bem como os procedimentos de inferências utilizadas podem ser assemelhados a um conjunto de conhecimento específico que está relacionado à área de domínio de um perito[2]. Uma arquitetura típica de um sistema especialista é apresentada na

[2]No que tange esta tese, o domínio explorado é da engenharia de *software* seguro e padrões de *software*

Figura 26.

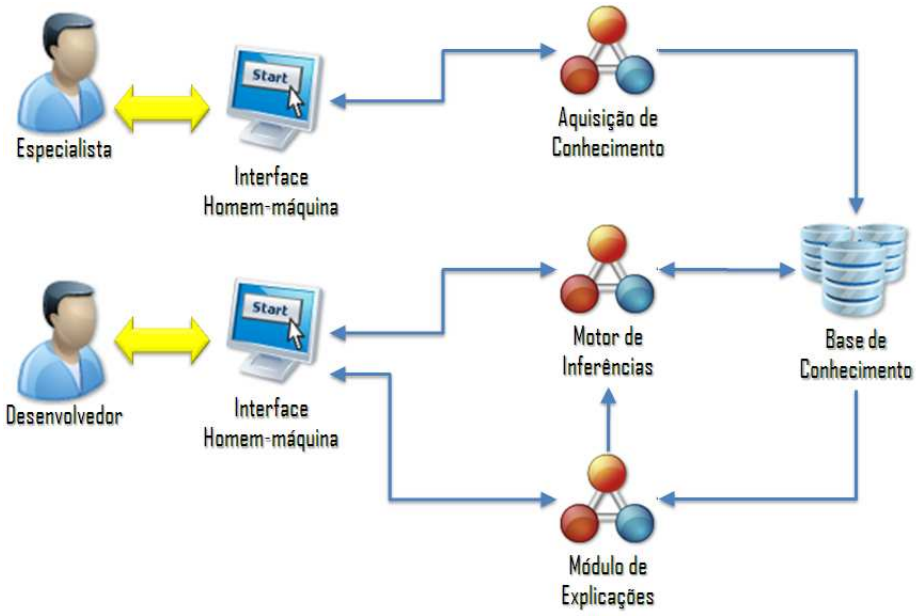

Figura 26: Especificação Típica de um Sistema Especialista, modificado de (GIARRATANO; RILEY, 2005).

7.4.1 Base de Conhecimento

A *base de conhecimento* é o repositório do conhecimento do especialista no domínio específico. A base deve ser construída de modo a permitir que o especialista, que não tem conhecimento de programação, tenha acesso ao conhecimento representado, no formato mais próximo possível de sua maneira de expressá-lo.

Neste protótipo, a base de conhecimento é baseada em fatos que especificam os padrões de *software* de acordo com as dimensões definidas pela matriz de interesses, vide capítulo 6, adicionadas das dimensões: tipo e origem da ameaça, descritas pela seção 5.3 do capítulo 5. A especificação dos fatos obedece à sintaxe apresentada pela Figura 27.

Como pode ser observado na Figura 27, a especificação da classificação do padrão segue a orientação da Programação em Lógica (Prolog). A opção pelo uso do Prolog possui as

```
pattern(id,lifecycle(),response(),architecture(),domain(),component(),constraint(),threatSource(),threatType()).
    id -> pattern identification
    lifecycle(dom,req,pro,des,imp,int,dep,ope,mai,dis)
        dom -> domain analysis
        req -> requirements
        pro -> problem analysis
        des -> design
        imp -> implementation
        int -> integration
        dep -> deployment
        ope -> operation
        mai -> maintenance
        dis -> disposal
    response(avo,det,pre,dtr,mit,rec,for)
        avo -> avoidance
        det -> detection
        dtr -> deterrence
        mit -> mitigation
        rec -> recovery
        for -> forensics
    architecture(app,pOS,com)
        app -> application
        pOS -> platform & OS
        com -> communication
        app(cli,log,dat)
            cli -> client
            log -> logic
            dat -> data
        com(dst,tra,net)
            dst -> distribution
            tra -> transport
            net -> network
    domain(eco,soa,mbl,sca,mil)
        eco -> e-commerce
        soa -> SOA
        mbl -> mobile & ubiquitous
        sca -> SCADA
        mil -> military
    component(new,oss,rts,mod,wiz,rli,oso,leg,cot,wsc)
        new -> new code
        oss -> open source software
        rts -> run time script
        mod -> model transformation
        wiz -> wizard code
        rli -> reused library
        oso -> outsourced
        cot -> off-the-shelf
        wss -> remote webservice
    constraint(reg,org,hum,mec)
        reg -> regulatory
        org -> organizational
        hum -> human
        mec -> mecanism
    threatsource(ina,inn,ext)
        ina -> internal authorized
        inn -> internal non authorized
        ext -> external
    threattype(spo,tam,rep,inf,den)
        spo -> spoofing
        tam -> tampering
        rep -> repudiation
        inf -> information disclosure
        den -> denial of service
```

Figura 27: Regra de Formação do Predicado para os Padrões de *Software*

seguintes justificativas:

- Adequação para a representação do conhecimento, uma vez que se pode traduzir praticamente todos os formalismos usando para este fim a Lógica de Primeira Ordem (LPO).

- Ao contrário de programas em Pascal, C, C++, um programa em lógica não é a descrição de um procedimento para se obter a solução de um problema. Na realidade, o sistema utilizado no processamento de programas em lógica é inteiramente responsável pelo procedimento a ser adotado na sua execução.

- Linguagem baseada no princípio declarativo pelo qual o fluxo de controle fica a cargo do interpretador, diferentemente das linguagens imperativas (Pascal, C++ etc.), no qual o

fluxo de controle deve ser especificado explicitamente para que se consiga executar certa operação (LIMA-MARQUES, 1997).

Para exemplificar como se dá a classificação do padrão usando o predicado definido pela Figura 27, segue a declaração que especifica a classificação do padrão apresentado pela Figura 19:

```
pattern(xacml_access_control_evaluator,
    lifecycle(null,null,null,des,null,null,null,null,null,null),
    response(null,null,pre,null,null,null,null),
    architecture(app(null,null,null),null,com(dst,null,null)),
    domain(eco,soa,null,null,null),
    component(new,oss,rts,mod,wiz,rli,oso,leg,cot,wsc),
    constraint(null,null,null,mec),
    threatsource(ina,inn,ext),
    threattype(null,tam,null,inf,null)).
```

As ocorrências de "*null*" denotam que o padrão não pode ser incluído no conjunto da categoria em questão.

7.4.2 Mecanismo de Inferência

O mecanismo de inferência é responsável pelo raciocínio automático usando a regra de resolução como expressão fundamental de inferência. O mecanismo busca os conhecimentos necessários na base de fatos e os aplica a partir de um encadeamento de busca em profundidade.

Duas regras foram criadas em Prolog para atender às necessidades do protótipo. A primeira se refere à regra para encontrar dentre os padrões já classificados (fatos) aqueles que atendem à necessidade de controle delimitada pelo analista. A regra apresentada a seguir coleciona uma lista de padrões *L* cujas categorias correspondam às fornecidas pelo analista:

```
findPatterns(Dom,Req,Pro,Des,Imp,Int,Dep,Ope,Mai,Dis,Avo,Det,Pre,Dtr,Mit,Rec,For,Cli,Log,Dat,POS,Dst,Tra,Net,
    Eco,Soa,Mbl,Sca,Mil,New,Oss,Rts,Mod,Wiz,Rli,Oso,Leg,Cot,Wsc,Reg,Org,Hum,Mec,Ina,Inn,Ext,Spo,Tam,Rep,Inf,
    Den,L) :-
            setof(PatternId,Dom^Req^Pro^Des^Imp^Int^Dep^Ope^Mai^Dis^Avo^Det^Pre^Dtr^Mit^Rec^For^Cli^Log^Dat^
               POS^Dst^Tra^Net^Eco^Soa^Mbl^Sca^Mil^New^Oss^Rts^Mod^Wiz^Rli^Oso^Leg^Cot^Wsc^Reg^Org^Hum^Mec^
               Ina^Inn^Ext^Spo^Tam^Rep^Inf^Den^pattern(PatternId,lifecycle(Dom,Req,Pro,Des,Imp,Int,Dep,Ope,
               Mai,Dis),response(Avo,Det,Pre,Dtr,Mit,Rec,For),architecture(app(Cli,Log,Dat),POS,com(Dst,Tra,
               Net)),domain(Eco,Soa,Mbl,Sca,Mil),component(New,Oss,Rts,Mod,Wiz,Rli,Oso,Leg,Cot,Wsc),
               constraint(Reg,Org,Hum,Mec),threatsource(Ina,Inn,Ext),threattype(Spo,Tam,Rep,Inf,Den)),L).
```

A segunda regra implementa o outro recurso habilitado pela matriz de interesses, que é a identificação de lacunas que denotam a oportunidade de redação do padrão, no caso dele ainda não ter sido publicado, ou da inclusão de determinado padrão na base de fatos, tornando a sugestão do padrão pela ferramenta mais robusta.

7.4.3 Interface Homem-Máquina

A interface homem-máquina é responsável pela interação entre usuário e sistema, quer seja para o analista ou para o especialista. A interface entre o especialista e o sistema acontece a partir da definição dos fatos relacionados aos padrões a partir de declarações de predicados que sigam as orientações explicitadas pela Figura 27. O objeto de manipulação corresponde a um arquivo texto com as referidas declarações.

Já o analista precisa fornecer as entradas necessárias para que o sistema raciocine a partir delas e dos fatos e regras estabelecidos para resolver o problema. Tais entradas correspondem às necessidades de controle da ameaça levantadas pelo analista, adicionadas das características da ameaça: origem e tipo.

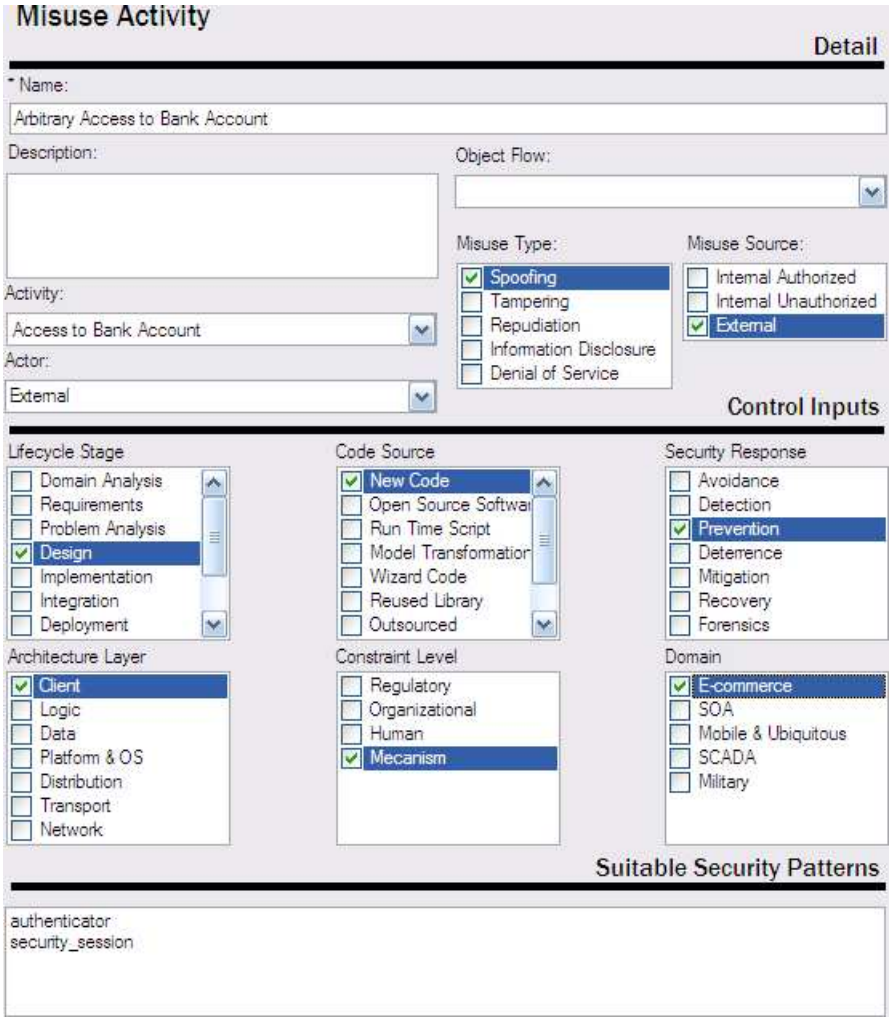

Figura 28: Interface do Analista para Manutenção das Informações da Ameaça, incluindo-se a seleção do padrão.

Para facilitar a atividade do analista, o protótipo está preparado para receber as entradas, mencionadas anteriormente, a partir da interface gráfica mostrada pela Figura 28. As entradas relacionadas diretamente à ameaça usadas na seleção do padrão estão dispostas nos itens: *threat source* e *threat type*, que se referem respectivamente à origem da ameaça (interno autorizado, interno não autorizado ou externo) e tipo da ameaça (STRIDE). Já as informações relacionadas ao controle desejado estão dispostas na seção da tela denominada "*Control Input*", cujos grupos de caixas de verificação correspondem às dimensões da matriz.

Os itens selecionados nas caixas de verificação denotam que o padrão *deve* estar categorizado no conjunto do item relacionado. Já os itens não selecionados denotam que a

categorização do padrão para o item referido é indiferente. Por exemplo, para que o *software* tenha sugerido os padrões "authenticator" e "security_session" mostrados pela Figura 28, é mandatório que eles sejam classificados pelo especialista de acordo com as necessidades em questão: ciclo de vida (projeto), código fonte (novo), resposta (prevenção), camada da arquitetura (cliente), nível de restrição (mecanismo), domínio (*e-commerce*), origem da ameaça (externo) e tipo de ameaça (*spoofing*). Suas outras eventuais classificações são ignoradas para efeito da recomendação do padrão, que é registrado na seção "*Suitable Security Patterns*".

A consulta à base de fatos e seu consecutivo retorno é habilitado pelo uso do Amzi Prolog & Logic Server, que permite a integração de ambientes com interface gráfica rica (C++, C, Delphi, C#, VB.NET, ASP.NET, Java etc.) com seu interpretador de Prolog (Amzi! Inc., 2008).

7.4.4 Módulo de Explicação

Uma das características dos sistemas especialistas é que ele deve explicar seu raciocínio em tempo de execução. Como, normalmente, o raciocínio envolvido é bem complexo, os módulos de explicação cumprem um importante papel na tarefa de esclarecer e acompanhar as etapas da argumentação envolvida. Apesar de não ter sido explorado o retorno ao usuário sobre a explicação pela sugestão do padrão, essa informação pode ser observada a partir da interpretação do código em um interpretador de Prolog.

7.5 Trabalho Relacionado

O protótipo apresentado tem como objetivo auxiliar o desenvolvedor na fase de análise e projeto de *software* para que o resultado final de seu trabalho seja um produto mais seguro. Dentro dessa mesma perspectiva, a ferramenta *Threat Analysis and Modeling Tool* (TAM) busca instrumentalizar o desenvolvedor para que a análise crítica sobre as ameaças a que o *software*

está submetido fiquem mais evidentes, oferecendo ainda um conjunto de mecanismos-padrão, específicos da plataforma *Windows* e *.NET*, que podem ser relacionados ao gosto do analista, registrando suas opções para controlar as ameaças.

A TAM está à frente do protótipo no que tange a estabilidade e recursos gráficos disponíveis, como, por exemplo, árvores de ameaça, porém, se limita a tratar de informações disponíveis somente em momento avançado na fase de projeto. A expectativa de uso do protótipo desde as primeiras etapas do ciclo de desenvolvimento de *software* e a exploração do conhecimento empacotado pelos padrões de *software*, adicionado dos recursos de recuperação da melhor opção de padrão, de acordo com os interesses do analista (dimensões), pontuam os diferenciais que colocam o protótipo em vantagem comparado ao TAM.

O código-fonte do protótipo, bem como todos os componentes na versão usada na sua construção, estão disponíveis no CD que acompanha esta tese, bem como no site `http://code.google.com/p/misuseactivities/`.

8 Conclusão

As perdas relacionadas aos ataques em sistemas habilitadas por falhas de segurança em *software* adicionaram mais uma dimensão na complexidade inerente ao desenvolvimento de *software*. Além da entrega de um *software* funcional e adequado às necessidades do seu negócio, exige-se que ele garanta um nível de segurança que o habilite a resistir às adversidades provocadas pelos inúmeros atacantes existentes. Baseada na presunção de que a segurança deve ser considerada em todas as etapas do ciclo de vida de seu desenvolvimento para se obter um *software* seguro, esta tese apresentou instrumentos que auxiliam especificamente as etapas de análise e projeto, explorando a visão do atacante, a partir da análise de ameaças e do conhecimento estruturado registrado no formato de padrões.

Primeiramente, uma readequação da técnica da análise de requisitos de segurança *atividades de abuso* foi desenvolvida, de modo a aproximar o resultado de sua execução ao seu objetivo de levantar a maior quantidade possível de ameaças ao *software* objeto de análise. Para isso, a técnica foi incrementada, a partir da revisão de seu processo e adição de novas perspectivas de análise, incluindo-se o tipo de ameaça, representada pelas categorias estabelecidas pelo STRIDE, bem como a origem da ameaça, representada pelo privilégio do atacante (interno autorizado ou não e externo à organização). Os resultados mostraram a melhoria efetiva da técnica na identificação de maior quantidade de ameaças.

Então, partiu-se para o desafio de facilitar a recuperação dos padrões de *software*, com o intuito de facilitar sua adoção no ciclo de desenvolvimento. Diferentemente das alternativas

similares, elegeu-se como foco por esta tese o usuário que demanda esse tipo de conhecimento. O método apresentado tem como objetivo, então, classificar os padrões de *software* para seu usuário final, a partir da definição de matriz de interesses, cujas dimensões variam de acordo com o seu interesse e ainda podem ser especializadas em dimensões secundárias. Um dos resultados observados foi a homogeneidade nas classificações realizadas por diversos especialistas usando as dimensões apresentadas, justificando a consistência do método desenvolvido.

Por fim, destaca-se o protótipo, cuja pretensão original era desenvolver apenas a especificação de um *software* de apoio à atividade de abuso e à recuperação de padrões, mas que se chegou à implementação propriamente dita das seguintes funcionalidades: importação das informações do diagrama de atividades em XMI, geração automática de atividades de abuso e recuperação de padrões a partir das informações dos controles, representados pelas dimensões originais da matriz de interesse, adicionadas da dimensão de tipo e origem da ameaça.

Os resultados alcançados vão ao encontro das expectativas estabelecidas pelos objetivos da pesquisa e ampliam as oportunidades de investigação científica a serem empreendidas futuramente, dentre as quais cabe destacar:

- Realização de pesquisa experimental para validar a efetividade da metodologia de desenvolvimento de *software* baseada em padrões como um todo, a partir das contribuições apresentadas (TRAVASSOS; GUROV; AMARAL, 2002).

- Refinamento da técnica de elicitação de requisitos de segurança, incluindo-se de forma objetiva a influência do contexto para que diminua a incidência de falsos-positivos.

- Evoluir a interface gráfica do protótipo para permitir a adição de novos padrões na base

de conhecimento, acesso a relatórios analíticos com o resultado da análise e padrões sugeridos para controle, bem como para indicar lacunas de padrões.

- Como o protótipo lida com a codificação UML especificada pelo XMI, adicionar funcionalidade para transformar o modelo a partir dos padrões sugeridos. Isso exigiria o desenvolvimento dos *Platform Independent Model* (PIM) para cada padrão cadastrado, bem como o seu *Platform Specific Model* (PSM), que poderia explorar o conceito de cartuchos do AndroMDA (AndroMDA, 2009).

- Identificação e documentação de padrões para preencherem as lacunas identificadas pelo uso da matriz de interesses.

Referências Bibliográficas

ALEXANDER, C. et al. *A Pattern Language: Towns - Buildings - Construction*. [S.l.]: Oxford University Press, 1977.

ALEXANDER, I. Misuse cases: Use cases with hostile intent. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 20, n. 1, p. 58–66, 2003. ISSN 0740-7459.

ALLEN, J. H. et al. *Software Security Engineering: A Guide for Project Managers*. [S.l.]: Addison Wesley Professional, 2008.

Amzi! Inc. *Amzi! Prolog + Logic Server*. 2008. Http://www.amzi.com.

ANDERSON, J. M. Why we need a new definition of information security. *Computers & Security*, v. 22, n. 4, p. 308–313, 2003. Disponível em: <http://dx.doi.org/10.1016/S0167-4048(03)00407-3>. Acesso em: 31 de agosto de 2007.

AndroMDA. 2009. Http://andromda.org/. V. 3.03.

ARKIN, B.; STENDER, S.; MCGRAW, G. Software penetration testing. *IEEE Security & Privacy*, v. 3, n. 1, p. 84–87, 2005. Disponível em: <http://doi.ieeecomputersociety.org/10-.1109/MSP.2005.23>.

ASHBAUGH, D. A. Assessing information security risks in the software development life cycle. *CrossTalk - The Journal of Defense Software Engineering*, Sep 2006, p. 21–25, set. 2006. Disponível em: <http://www.stsc.hill.af.mil/crosstalk/2006/09/0609Ashbaugh.html>.

Babylon Translator. 2007. Disponível em: <http://www.babylon.com/>.

BARBOSA, M. A. A refinement calculus for software components and architectures. In: *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2005. p. 377–380. ISBN 1-59593-014-0.

BARMAN, S. *Writing Information Security Policies*. Indianapolis: SAMS, 2001. ISBN 157870264X.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2nd. ed. [S.l.]: Addison Wesley, 2003.

BECK, K.; CUNNINGHAM, W. *Using Pattern Languages for Object Oriented Programs*. [S.l.], 1987.

BERNSTEIN, D. *The qmail Security Guarantee*. December 1997. Disponível em: <http://cr.yp.to/qmail/guarantee.html>. Acesso em: 21 de setembro de 2007.

BISHOP, M. *Computer Security: Art and Science*. [S.l.]: Addison-Wesley, Boston, USA, 2003.

BISHOP, M.; FRINCKE, D. A. A human endeavor: Lessons from shakespeare and beyond. *IEEE Security & Privacy*, v. 3, n. 4, p. 49–51, 2005. Disponível em: <http://doi-.ieeecomputersociety.org/10.1109/MSP.2005.87>.

BISHOP, M.; FRINCKE, D. A. Teaching secure programming. *IEEE Security & Privacy*, v. 3, n. 5, p. 54–56, 2005. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MSP.2005-.133>.

BOOCH, G. *On Design , Handbook of Software Architecture*. March 2006. Blog. Disponível em: <http://www.booch.com/architecture/blog.jsp?archive=2006-03.html>.

BRAZ, F. A.; FERNANDEZ, E.; VANHILST, M. Eliciting software security requirements through misuse actions. In: *accepted for the Procedings 2nd International Workshop on Secure systems methodologies using patterns (SPattern'08)*. [S.l.: s.n.], 2008.

BROOKS, J. F. P. No silver bullet: essence and accidents of software engineering. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 4, p. 10–19, 1987. ISSN 0018-9162. Disponível em: <http://portal.acm.org/citation.cfm?id=26441>. Acesso em: 23 de julho de 2005.

BROWN, A. W. Model driven architecture: Principles and practice. *Software and System Modeling*, v. 3, n. 4, p. 314–327, 2004. Disponível em: <http://www.springerlink.com/index-/10.1007/s10270-004-0061-2>.

BROWN, G. *Laws of Form*. [S.l.]: George Allen & Unwin, 1971.

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. C. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. 5. ed. [S.l.]: John Wiley & Sons, 2007.

BUSCHMANN, F. et al. *Pattern-oriented Software Architecture: A System of Patterns*. [S.l.]: Wiley, 1996.

CERT-CC. *Full Statistics*. Jan 2008. Disponível em: <http://www.cert.org/stats/fullstats-.html>. Acesso em: 26 de Janeiro de 2008.

CHESS, B.; WEST, J. *Secure Programming with Static Analysis*. [S.l.]: Addison-Wesley, 2007.

CHUNG, L.; NIXON, B. A.; YU, E. An approach to building quality into software architecture. In: *CASCON '95: Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. [S.l.]: IBM Press, 1995. p. 13.

CLEMENTS, P. et al. *Documenting Software Architectures: Views and Beyond*. [S.l.]: Addison Wesley Professional, 2002.

CLEMENTS, P. et al. *A Practical Method for Documenting Software Architectures*. 2002.

COAD, P. Object-oriented patterns. *Commun. ACM*, ACM, New York, NY, USA, v. 35, n. 9, p. 152–159, 1992. ISSN 0001-0782.

COLE, E.; RING, S. *Insider threat: protecting the eterprise from sabotage, spying, and theft*. 1. ed. [S.l.]: Syngress, 2005. 424 p.

COMMON CRITERIA SPONSORING ORGANIZATIONS. *Common Criteria for Information Technology Security Evaluation Part 1: Introduction and General Model, Version 3.1 Rev 1*. [S.l.], 2006.

COMMON CRITERIA SPONSORING ORGANIZATIONS. *Common Criteria for Information Technology Security Evaluation Part 2: Security Functional Components, Version 3.1 Rev 1*. [S.l.], 2006.

COMMON CRITERIA SPONSORING ORGANIZATIONS. *Common Criteria for Information Technology Security Evaluation Part 3: Security assurance components, Version 3.1 Rev 1*. [S.l.], 2006.

Computer Security Institute. *Computer crime and security survey*. 2002. Disponível em: <http://www.gocsi.com/press/20020407.jhtml? requestid=195148>.

COPLIEN, J. O. *Advanced C++ programming styles and idioms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1992. ISBN 0-201-54855-0.

CRANOR, L. F.; GARFINKEL, S. *Security and Usability*. [S.l.]: O'Reilly, 2005.

DAVIS, N. et al. Processes for producing secure software: Summary of us national cybersecurity summit subgroup report. *IEEE Security & Privacy*, v. 2, n. 3, p. 18–25, 2004. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MSP.2004.21>.

DEMO, P. *Metodologia do Conhecimento Científico*. São Paulo: Atlas, 2000.

DEVANBU, P. T.; STUBBLEBINE, S. G. Software engineering for security: a roadmap. In: *ICSE - Future of SE Track*. [S.l.: s.n.], 2000. p. 227–239. Acesso em: 18 de dezembro de 2005.

DIMACS. *DIMACS Workshop on Design and Formal Verification of Security Protocols*.

ESSMAYR, W.; PERNUL, G.; TJOA, A. Access controls by object-oriented concepts. In: *11th IFIP WG 11.3 Working Conf. on Database Security*. [S.l.: s.n.], 1997.

FERNANDEZ, E. B.; FRANCE, R. B.; WEI, D. A formal specification of an authorization model for object-oriented databases. In: *Proceedings of the ninth annual IFIP TC11 WG11.3 working conference on Database security IX : status and prospects*. London, UK, UK: Chapman & Hall, Ltd., 1996. p. 95–110. ISBN 0-412-72920-2.

FERNANDEZ, E. B.; LARRONDO-PETRIE, M. M.; GUDES, E. A method-based authorization model for object-oriented databases. In: *OOPSLA 1993 Workshop on Security in Object-oriented Systems*. [S.l.: s.n.], 1993. p. 70–79.

FERNANDEZ, E. B. et al. A methodology to develop secure systems using patterns. In: MOURATIDIS, H.; GIORGINI, P. (Ed.). *Integrating security and software engineering: Advances and future vision*. Hershey, Pennsylvania, USA: Idea Group, 2006. cap. V, p. 107–126.

FERNANDEZ, E. B. et al. Defining security requirements through misuse actions. In: OCHOA, S. F.; ROMAN, G.-C. (Ed.). *Advanced Software Engineering: Expanding the Frontiers of Software Technology*. [S.l.]: Springer Boston, 2006. (IFIP International Federation for Information Processing, v. 219), p. 123–137.

FERNANDEZ, E. B. et al. Classifying security patterns. In: *accepted for the Procedings of 10th Asia-Pacific Web Conference (APWEB'08)*. [S.l.: s.n.], 2008.

FLICK, U. *Introdução a Pesquisa Qualitativa*. 3a.. ed. [S.l.]: Artmed, 2009. 405 p.

FOWLER, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd. ed. [S.l.]: Addison Wesley, 2003. 208 p.

GALLAGHER, T.; JEFFRIES, B.; LANDAUER, L. *Hunting Security Bugs*. [S.l.]: Microsoft Press, 2006. 559 p. (Secure Software Development Series). ISBN 0-7356-2187-X.

GAMMA et al. *Design Patterns Elements of Reusable Object-Oriented Software*. Massachusetts: Addison-Wesley, 1994. ISBN 0-201-63361-2.

GARLAN, D. Software architecture: a roadmap. In: *ICSE - Future of SE Track*. [s.n.], 2000. p. 91–101. Disponível em: <http://doi.acm.org/10.1145/336512.336537>. Acesso em: 18 de julho de 2007.

GARLAN, D.; SHAW, M. An introduction to software architecture. In: AMBRIOLA, V.; TORTORA, G. (Ed.). *Advances in Software Engineering and Knowledge Engineering*. Singapore: World Scientific Publishing Company, 1993. p. 1–39.

GERMAN, D.; COWAN, D. Towards a unified catalog of hypermedia design patterns. In: *Proceedings of 33rd Hawaii International Conference on System Sciences*. [S.l.: s.n.], 2001.

GIARRATANO, J. C.; RILEY, G. D. *Expert Systems: Principles and Programming*. Pacific Grove, CA, USA: Brooks/Cole Publishing Co., 2005. ISBN 0534384471.

GOERTZEL, K. M. et al. *Security in the Software Life Cycle*. [S.l.], August 2006. Disponível em: <https://buildsecurity.us-cert.gov>. Acesso em: 5 de setembro de 2006.

GOLLMANN, D. *Computer security*. 2nd. ed. New York, NY, USA: John Wiley & Sons, Inc., 2005. ISBN 0-471-97844-2.

GORTON, I. *Essential Software Architecture*. [S.l.]: Springer, 2006.

GRAFF, M. G.; WYK, K. R. van. *Secure Coding: Principles & Practices*. O'Reilly & Associates, Inc., 2003. Disponível em: <http://www.securecoding.org/>.

GRAHAM, D. *Introduction to the CLASP Process*. November 2006. Disponível em: <https://buildsecurityin.us-cert.gov/daisy/bsi/>. Acesso em: 30 de agosto de 2007.

GREGOIRE, J. et al. On the secure software development process: Clasp and sdl compared. In: *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*. Washington, DC, USA: IEEE Computer Society, 2007. p. 1. ISBN 0-7695-2952-6. Acesso em: 2 de julho de 2007.

GROSSMAN, J. et al. *Cross Site Scripting Attacks: XSS Exploits and Defense*. [S.l.]: Syngress, 2007.

HAFIZ, M.; ADAMCZYK, P.; JOHNSON, R. E. Organizing security patterns. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 24, n. 4, p. 52–60, 2007. ISSN 0740-7459.

HALEY, B. et al. Using trust assumptions with security requirements. *Requir. Eng.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 11, n. 2, p. 138–151, 2006. ISSN 0947-3602.

HALEY, C. et al. Security requirements engineering: A framework for representation and analysis. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 34, n. 1, p. 133–153, 2008. ISSN 0098-5589.

HATEBUR, D.; HEISEL, M.; SCHMIDT, H. Analysis and component-based realization of security requirements. In: *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*. [S.l.]: IEEE, 2008. (IEEE Transactions). To be published.

HOFMEISTER, C. et al. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, v. 80, n. 1, p. 106–126, 2007. Disponível em: <http://dblp.uni-trier.de/db/journals/jss/jss80.html>.

HOFMEISTER, C.; NORD, R. L.; SONI, D. Describing software architecture with UML. In: DONOHOE, P. (Ed.). *WICSA*. [S.l.]: Kluwer, 1999. (IFIP Conference Proceedings, v. 140), p. 145–160. ISBN 0-7923-8453-9.

HOFMEISTER, C.; NORD, R. L.; SONI, D. *Applied Software Architecture*. [S.l.]: Addison Wesley, 2000.

HOFSTADER, J. We don't need no architects. *The Architecture Journal*, v. 15, p. 2–6, 2008.

HOGLUND, G.; MCGRAW, G. *Exploiting Software: How to Break Code*. [S.l.]: Addison-Wesley, 2004.

HOO, K. S. *How Much Is Enough? A Risk-Management Approach to Computer Security*. June 2000. Disponível em: <citeseer.ist.psu.edu/505332.html>. Acesso em: 20 de maio de 2006.

HOPE, P.; LAVENHAR, S.; PETERSON, G. *Architectural Risk Analysis*. Oct 2005. Https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture/10-BSI.html.

HOPE, P.; MCGRAW, G.; ANTON, A. I. Misuse and abuse cases: Getting past the positive. *IEEE Security and Privacy*, IEEE Computer Society, Los Alamitos, CA, USA, v. 02, n. 3, p. 90–92, 2004. ISSN 1540-7993.

HOWARD, M. Expert tips for finding security defects in your code. *MSDN Magazine*, November 2003. Disponível em: <http://msdn.microsoft.com/msdnmag/issues/03/11-/SecurityCodeReview/>. Acesso em: 10 de fevereiro de 2006.

HOWARD, M. Building more secure software with improved development processes. *IEEE Security & Privacy*, v. 2, n. 6, p. 63–65, 2004.

HOWARD, M. A process for performing security code reviews. *IEEE Security and Privacy*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 4, n. 4, p. 74–79, 2006. ISSN 1540-7993.

HOWARD, M.; LEBLANC, D. *Writing Secure Code.* 2nd. ed. Redmond, WA: Microsoft Press, 2002. 650 p.

HOWARD, M.; LIPNER, S. Inside the windows security push. *IEEE Security & Privacy*, v. 1, n. 1, p. 57–61, 2003. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/MSECP-.2003.1176996>. Acesso em: 17 de janeiro de 2006.

HOWARD, M.; LIPNER, S. *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. [S.l.]: Microsoft Press, 2006. 304 p.

IEEE. *IEEE Standards Description: 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems-Description*. [S.l.], 2000. Disponível em: <http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html>.

INFORMATION technology - Systems Security Engineering - Capability Maturity Model (SSE-CMM). Geneva, 2002.

IONITA, M. T.; AMERICA, P.; HAMMER, D. K. A method for strategic scenario-based architecting. In: *HICSS*. IEEE Computer Society, 2005. ISBN 0-7695-2268-8. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/HICSS.2005.24>.

ISO/IEC 17799:2005. Information Technology Security Techniques Code of Practice for Information Security Management. Geneva, 2005.

JAATUN, M. G.; TøNDEL, I. A. Covering your assets in software engineering. *ARES*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1172–1179, 2008.

JACKSON, M. *Problem frames: analyzing and structuring software development problems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-59627-X.

JAFERIAN, P. et al. RUPSec: Extending business modeling and requirements disciplines of RUP for developing secure systems. In: *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2005. p. 232–239. ISBN 0-7695-2431-1.

JüRJENS, J. Using UMLsec and goal trees for secure systems development. In: *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2002. p. 1026–1030. ISBN 1-58113-445-2.

KAZMAN, R. et al. Scenario-based analysis of software architecture. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 13, n. 6, p. 47–55, 1996. ISSN 0740-7459.

KAZMAN, R.; ASUNDI, J.; KLEIN, M. *Making Architecture Design Decisions: An Economic Approach*. [S.l.], jul. 02 2002. Disponível em: <http://citeseer.ist.psu.edu/593655.html; ftp://ftp.sei.cmu.edu/pub/documents/02.reports/pdf/02tr035.pdf>.

KAZMAN, R. et al. Experience with performing architecture tradeoff analysis. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999. p. 54–63. ISBN 1-58113-074-0.

KAZMAN, R.; BASS, L.; KLEIN, M. The essential components of software architecture design and analysis. *Journal of Systems and Software*, v. 79, n. Issue 8, p. 1207–1216, August 2006. Disponível em: <http://www.sciencedirect.com/science/article/B6V0N-4K8S5HK-1/2-/ab63330bb982ddea0c9eb176a135e919>.

KAZMAN, R.; KLEIN, M.; CLEMENTS, P. *ATAM: Method for Architecture Evaluation*. [S.l.], set. 2000. Disponível em: <http://www.sei.cmu.edu/pub/documents/00.reports/pdf-/00tr004.pdf>.

KELLY, G. *The Psychology of Personal Constructs*. [S.l.]: Norton, 1955.

KIS, M. Information security antipatterns in software requirements engineering. In: *Proceedings of the 9th Pattern Languages of Programs Conference (PLoP2002)*. [S.l.: s.n.], 2002.

KLEIN, M.; KAZMAN, R. *Attribute-Based Architectural Styles*. Pittsburgh, PA, 1999. Disponível em: <http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr022.pdf>.

KONHEIM, A. G. *Computer Security and Cryptography*. [S.l.]: John Wiley & Sons, 2007.

KRASNER, G. E.; POPE, S. T. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, v. 1, n. 3, p. 26–49, ago./set. 1988.

KREBS, B. *Cyber Incident Blamed for Nuclear Power Plant Shutdown*. June 2008. WashingtonPost.com. Disponível em: <http://www.washingtonpost.com/wp-dyn/content-/article/2008/06/05/AR2008060501958.html>. Acesso em: 05 de junho de 2008.

KRUCHTEN, P. *The Rational Unified Process: And Introduction*. Third. [S.l.]: Addison-Wesley, 2003. ISBN 0321197704.

KRUCHTEN, P. B. The 4+1 view model of architecture. *IEEE Software*, v. 12, n. 6, p. 42–50, nov. 1995.

LEVESON, N. A new accident model for engineering safer systems. *Safety Science*, v. 42, n. 4, p. 237–270, April 2004.

LIMA-MARQUES, M. Introdução à linguagem PROLOG. Universidade de Brasília - UnB, Departamento de Ciência da Informação. 1997.

MARCIANO, J. L. P. *Segurança da Informação - uma abordagem social*. Tese (Doutorado) — CID/FACE-UNB, Brasília, Setembro 2006.

MARTIMIANO, L. A. F. *Sobre a estruturação de informação em sistemas de segurança computacional: o uso de ontologia*. Tese (Doutorado) — Universidade de São Paulo, São Carlos, August 2006.

MCGRAW, G. Software security. *Security & Privacy Magazine, IEEE*, v. 2, n. 2, p. 80–83, 2004.

MCGRAW, G. *Software Security: Building Security In*. [S.l.]: Addison Wesley Professional, 2006. 448 p.

MEDVIDOVIC, N. *Formal Definition of the Chiron-2 Software Architectural Style*. Irvine, CA, USA, August 1995. Disponível em: <citeseer.ist.psu.edu/article/medvidovic00formal.html>.

Microsoft Corporation. *Microsoft Threat Analysis and Modeling Tool*. 2006. Version 2.1.2. Disponível em: <http://www.microsoft.com/downloads/details.aspx?familyid=59888078-9daf-4e96-b7d1-944703479451>.

MORANA, M. M. Producing secure software with security enhanced software development process. *(IN)SECURE Magazine*, v. 16, p. 57–67, 2008. Disponível em: <www.insecure-.com>.

MS03-007: Memória intermédia não verificada num componente do Windows pode comprometer o servidor da Web. [S.l.], 2006. Disponível em: <http://support.microsoft.com-/kb/815021/pt>. Acesso em: 23 de Setembro de 2006.

MYAGMAR, S.; LEE, A. J.; YURCIK, W. Threat Modeling as a Basis for Security Requirements. In: *SREIS*. [S.l.: s.n.], 2005.

National Cyber Security Partnership Strategy. *Improving Security Across The Software Development Lifecycle*. [S.l.], April 2004.

National Security Agency. *National Training Standard for Information Systems Security (INFOSEC) Professionals*. 1994. Http://www.nsa.gov/ia/academia/cnsstesstandards.cfm. NSTISSI-4011.

NOVOSOFT. *Novosoft UML Library for Java*. April 2002. Http://nsuml.sourceforge.net/.

NUML. *nUml 0.5*. Dec 2007. Http://numl.sourceforge.net/.

Object Management Group. *Unified Modeling Language: Superstructure*. August 2005. Http://www.omg.org/docs/formal/05-07-04.pdf. Version 2.0.

Open Web Application Security Project. *The OWASP Testing Project*. 2008. Http://www.modsecurity.org/archive/OWASPTesting_PhaseOne.pdf.

PAES, C. E. de B.; HIRATA, C. M. RUP extension for the development of secure systems. *ITNG*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 643–652, 2007. Acesso em: 16 de julho 2007.

PAYMENT Card Industry (PCI) Data Security Standard. [S.l.], 2006.

PELTIER, T. R. *Information Security Risk Analysis*. [S.l.]: AUERBACH, 2001. 296 p.

PELTIER, T. R.; PELTIER, J.; BLACKLEY, J. *Information Security Fundamentals*. N.W. Corporate Blvd., Boca Raton, Florida 33431: CRC Press, 2000.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, ACM Press, New York, NY, USA, v. 17, n. 4, p. 40–52, 1992. ISSN 0163-5948.

PFLEEGER, C. P. The fundamentals of information security. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 14, n. 1, p. 15–16,60, 1997. ISSN 0740-7459.

RANDAZZO, M. R. et al. *Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector*. [S.l.], 2004.

REDWINE, S. T. *Software Assurance - A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software*. [S.l.], October 2007.

REN, J. *A Connector-Centric Approach to Architectural Access Control*. Tese (Doutorado) — Information and Computer Science - University of California, 2006.

RESCORLA, E. Security holes ... who cares? In: *Proc. 12th Usenix Security Conf.* [S.l.]: Usenix Assoc., 2003.

SCHUMACHER, M. *Security Engineering with Patterns: Origins, Theoretical Models, and New Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2003. ISBN 3540407316.

SCHUMACHER, M.; ACKERMANN, r.; ATEINMETZ, R. Towards security at all stages of a systems life cycle. In: *Proceedings of . Int. Conf. on Software, Telecommunications, and Computer Networks (Softcom)*. [S.l.: s.n.], 2000.

SCHUMACHER, M. et al. *Security Patterns: Integrating Security and Systems Engineering*. [S.l.]: John Wiley & Sons Ltd, 2006.

SHAW, M.; CLEMENTS, P. The golden age of software architecture. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 23, n. 2, p. 31–39, 2006. ISSN 0740-7459.

SHAW M. L. G., . G. B. R. Kelly's 'geometry of psychological space' and its significance for psychological modeling. *New Psychologist*, v. 10, p. 23–31, 1992.

SHIREY, R. *(RFC) 2828 - Internet Security Glossary*. [S.l.], May 2000. Disponível em: <http://www.faqs.org/rfcs/rfc2828.html>.

SHUMBA, R. et al. Teaching the secure development lifecycle: Challenges and experiences. In: UNIVERSITY OF MARYLAND, UNIVERSITY COLLEGE. *Proceedings of the 10th Colloquium for Information Systems Security Education*. Adelphi, MD, 2006.

SINDRE, G.; OPDAHL, L. Eliciting security requirements with misuse cases. *Requir. Eng.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 10, n. 1, p. 34–44, 2005. ISSN 0947-3602.

STEEL, C.; NAGAPPAN, R.; LAI, R. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. [S.l.]: Prentice Hall, 2005.

STONEBURNER, G.; GOGUEN, A.; FERINGA, A. *Risk Management Guide for Information Technology Systems*. July 2002. NIST Special Publication 800-30. U.S. Department of Commerce. Disponível em: <www.csrc.nist.gov/publications/nistpubs/800-30/sp800-30-.pdf>.

SUTTON, M.; GREENE, A.; AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. [S.l.]: Addison-Wesley Professional, 2007. ISBN 0321446119.

SWIDERSKI, F.; SNYDER, W. *Threat Modeling*. [S.l.]: Microsoft Press, 2004. 288 p.

SYMANTEC Internet Security Threat Report - Trends for July 05 to December 05. [S.l.], March 2006. Disponível em: <http://eval.symantec.com/mktginfo/enterprise/white_papers-/ent-whitepaper_symantec_internet_security_threat_report_ix.pdf>. Acesso em: 14 de maio de 2007.

SYSTEMS Security Engineering Capability Maturity Model. [S.l.], 2003.

TØNDEL, I. A.; JAATUN, M. G.; MELAND, P. H. Security requirements for the rest of us: A survey. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 25, n. 1, p. 20–27, 2008. ISSN 0740-7459.

TRAVASSOS, G.; GUROV, D.; AMARAL, E. *Introdução à Engenharia de Software Experimental.* [S.l.], Abril 2002.

TROWBRIDGE, D. et al. *Describing the Enterprise Architectural Space*. June 2004. Http://msdn.microsoft.com/en-us/library/ms978655.aspx.

VANHILST, M.; FERNANDEZ, E. B.; BRAZ, F. A multi-dimensional classification for users of security patterns. *Journal of Research and Practice in Information Technology*, v. 41, n. 2, p. 87–97, May 2009.

VIEGA, J.; MCGRAW, G. *Building Secure Software: How to Avoid Security Problems the Right Way*. Indianoplis, IN: Addison-Wesley Publishing Co., 2001.

WEBER, S.; KARGER, P. A.; PARADKAR, A. A software flaw taxonomy: aiming tools at security. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 4, p. 1–7, 2005. ISSN 0163-5948.

WILLIAMS, A. T.; MACDONALD, N. *Integrate Security Best Practices and Tools Into Software Development Life Cycle*. [S.l.], February 2006.

WING, J. M. A symbiotic relationship between formal methods and security. In: *CSDA '98: Proceedings of the Conference on Computer Security, Dependability, and Assurance*. Washington, DC, USA: IEEE Computer Society, 1998. p. 26. ISBN 0-7695-0337-3.

WOJCIK, R. et al. *Attribute-Driven Design (ADD), Version 2.0*. [S.l.], 2006. Disponível em: <http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tr023.pdf>.

WYSOPAL, C. et al. *The Art os Software Security Testing*. [S.l.]: Symantec Press, 2007.

YODER, J.; BARCALOW, J. Architectural patterns for enabling application security. In: *PLOP'97*. [s.n.], 1997. Disponível em: <http://jerry.cs.uiuc.edu/~plop/plop97>.

ZHANG et al. Computer vulnerability evaluation using fault tree analysis. In: *International Conference on Information Security Practice and Experience (ISPEC), LNCS*. [S.l.: s.n.], 2005. v. 1.

ZUCCATO, A. Holistic security requirement engineering for electronic commerce. *Computers & Security*, v. 23, p. 63–76, February 2004. Disponível em: <http://www.sciencedirect.com-/science/article/B6V8G-4BS48S3-6/2/4334618f07d9d69a99238cb76ab9d073>.