Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Estratégia para Especificação e Geração de Casos de Teste a partir de Modelos UML

Karine de Pinho Peralta

Dissertação apresentada como requisito parcial à obtenção do grau de mestre em Ciência da Computação

Orientador: Prof. Dr. Avelino Francisco Zorzo

Porto Alegre
2009

Dados Internacionais de Catalogação na Publicação (CIP)

P426e Peralta, Karine de Pinho
Estratégia para especificação e geração de casos de teste a partir de modelos UML / Karine de Pinho Peralta. Porto Alegre, 2009.
86 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Avelino Francisco Zorzo

1. Informática. 2. Segurança – Computação. 3. Software. I. Zorzo, Avelino Francisco. II. Título.

CDD 005.8

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "**Estratégia para Especificação e Geração de Casos de Teste a Partir de Modelos UML**", apresentada por Karine de Pinho Peralta, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 08/01/09 pela Comissão Examinadora:

Prof. Dr. Avelino Francisco Zorzo – Orientador — PPGCC/PUCRS

Prof. Dr. Eduardo Augusto Bezerra - — PPGCC/PUCRS

Profa. Dra. Taisy Silva Weber – — UFRGS

Homologada em 03/03/09, conforme Ata No. 003/09 pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS

Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Para meus pais.

Agradecimentos

A minha família, pelo exemplo dado, apoio e auxílio sempre prestados. Ao professor Avelino Zorzo, pela orientação e trabalho realizado durante estes dois anos. À professora Ana Benso, pelo exemplo de profissionalismo e dedicação, mesmo quando não havia obrigação. Ao professor Flávio Oliveira, pelo auxílio e tempo dedicado ao trabalho. À HP P&D Brasil, pela oportunidade e bolsa concedida. E a todos aqueles, que de uma forma ou outra, fizeram parte da minha vida neste tempo, seja contribuíndo com alguma idéia, um artigo ou um momento de descontração.

Resumo

Com a expansão dos sistemas computacionais e com a popularização dos serviços providos pela Internet, é crescente a preocupação dos engenheiros de *software* com a segurança dos sistemas que desenvolvem. O volume de informações confidenciais que trafega pela Internet é cada vez maior, tornando essencial a avaliação de segurança destes sistemas antes de entregá-los a seus usuários. Entretanto, o tempo que dedicam em seus projetos à realização de testes para avaliar este aspecto é pequeno, fazendo com que *softwares* inseguros sejam liberados no mercado.

Verificar o nível de segurança de um *software* não é trivial. É preciso considerar este aspecto desde a fase de projeto do sistema, quando o modelo ainda está sendo elaborado. Uma limitação é a deficiência existente na área de segurança, seja em relação aos modelos, que provêem poucas estruturas para representar este aspecto, ou à pouca quantidade de ferramentas, documentos e *checklists* explicando como conduzir a execução de testes de segurança.

Neste contexto, vem se tornando popular uma técnica conhecida como teste baseado em modelos. Nesta, os testes a serem realizados são definidos automaticamente a partir do modelo da aplicação, de acordo com os aspectos desejados. Diversos trabalhos propõem modelos para especificar os mais variados requisitos, como funcionais e de desempenho, mas poucos se dedicam a definir uma forma de descrever aspectos de segurança.

Sendo assim, este trabalho propõe alguns estereótipos UML para especificar situações que podem comprometer a segurança de um software, além de um algoritmo que analisa estes e gera, automaticamente, casos de teste a partir do modelo. Desta forma, é possível assinalar, através da inserção de estereótipos no modelo, partes do sistema que podem conter vulnerabilidades, e, posteriormente, executar os casos gerados para verificar a ocorrência destas no software final. A elaboração deste trabalho tem dois objetivos principais: auxiliar durante a fase de implementação do *software*, prevendo situações que possam comprometer sua segurança e orientando os desenvolvedores, e permitir a geração automatizada de casos de teste de segurança a partir das informações inseridas.

Palavras-chave: Segurança. Teste Baseado em Modelos. UML. Geração de Casos de Teste.

Abstract

Due to the evolution of computer systems and the services provided by the Internet, software engineers are concerned about the security of the softwares they develop. The traffic of confidential data through the Internet is increasing, making essential the security evaluation of the systems before deploying them to the final users. However, a short period of time is dedicated to evaluate this characteristic during the test phase, resulting in an unsafe software.

It is not trivial to evaluate the security level of an application. This aspect must be considered since the design phase, when the model is still being elaborated. The problem is the lack of security information available, either in the models, which have limitations to represent this aspect, or in the documents, tools and checklists, which do not explain clearly how to conduct the security tests.

In this context, a new approach is becoming popular, known as model-based testing. The goal of this technique is to generate test cases by extracting specific information from a model, accordingly to the aspects that must be tested. Several works propose models to represent various aspects, such as functional or performance issues, but only a few of them are related to describing security characteristics.

Therefore, this work presents a set of UML stereotypes to specify some behaviors that may compromise software security, as well as an algorithm that analyzes the model and generates test cases. The use of the stereotypes allows the software engineer to annotate parts of the model that may contain vulnerabilities, and the test cases indicates to the tester the steps that must be performed to verify the occurrence of the vulnerabilities on the final software. This work has two main goals: to assist developers during the implementation process, emphasizing the functionalities that must be developed carefully; and, to allow the test case generation based on the security information provided by the model.

Keywords: Security. Model-Based Testing. UML. Test Case Generation.

Lista de Figuras

Figura 1 Relação entre falha, erro e defeito 16
Figura 2 Modelo de arquitetura utilizada pela TTCN-3 21
Figura 3 Formato das mensagens [22] 22
Figura 4 Execução do teste [22] 22
Figura 5 Modelo de autômatos finitos 23
Figura 6 Modelagem UML de um sistema de vendas [37] 28
Figura 7 Modelo U2TP para avaliar o sistema [37] 29
Figura 8 Taxonomia proposta por Taimur Aslam 34
Figura 9 Taxonomia proposta por Anil Bazaz *et al*. 35
Figura 10 Exemplo de uso do estereótipo «*fair exchange*» 42
Figura 11 Metamodelo do SecureUML [39] 43
Figura 12 Modelo de um sistema para agendamento de atividades [39] 44
Figura 13 Exemplos de códigos gerados a partir das definições com o SecureUML [39] 44
Figura 14 Tela do Apache JMeter para configuração de parâmetros 45
Figura 15 Resultados observados com o Apache JMeter 46
Figura 16 Modelagem de aplicação utilizando o *software* STAGE 47
Figura 17 Casos de teste gerados pelo STAGE a partir do modelo da aplicação . . 48
Figura 18 Lista de vulnerabilidades encontradas com o HP QAInspect 50
Figura 19 Mensagem HTTP *Request* com SQL *Injection* 50
Figura 20 Mensagem HTTP *Response* 51
Figura 21 Arquitetura geral proposta 56
Figura 22 Exemplo de diagrama de atividades 61
Figura 23 FSM equivalente 61
Figura 24 Inserção dos Estereótipos «BufferOverflow», «Encrypt» e «Expiration» 66
Figura 25 Inserção dos Estereótipos «SqlInjection» e «ByPassing» 66
Figura 26 Diagrama de Casos de Uso do OSCommerce 66
Figura 27 Inserção dos Estereótipos «BufferOverflow» e «Encrypt» 70
Figura 28 «SqlInjection» e «ByPassing» 70
Figura 29 Diagrama de Casos de Uso do TPC-W 70
Figura 30 Estereótipos de Segurança do Moodle 73
Figura 31 Diagrama de Casos de Uso do Moodle 73
Figura 32 Inserção dos Estereótipos «ByPassing», «Flooding» e «Expiration» . . 76
Figura 33 Diagrama de Casos de Uso do CesarFTP 76

Lista de Tabelas

Tabela 1 Ataques utilizados para avaliar a taxonomia DARPA 32
Tabela 2 Taxonomia de Weber 36
Tabela 3 Estereótipos, *tags* e restrições definidas pelo *profile* UMLSec 40
Tabela 4 *Top Ten List*, nas versões 2004 e 2007 58
Tabela 5 Formato de Geração dos Casos de Teste 62
Tabela 6 Mapeamento dos Requisitos de Segurança do OSCommerce 65
Tabela 7 Casos de Teste Gerados para o OSCommerce 67
Tabela 8 Resultados Obtidos com a Execução Manual dos Casos de Teste no OSCommerce 68
Tabela 9 Mapeamento dos Requisitos de Segurança do TPC-W 70
Tabela 10 Casos de Teste Gerados para o TPC-W 71
Tabela 11 Resultados Obtidos com a Execução Manual dos Casos de Teste no TPC-W 72
Tabela 12 Mapeamento dos Requisitos de Segurança do Moodle 73
Tabela 13 Casos de Teste Gerados para o Moodle 74
Tabela 14 Resultados Obtidos com a Execução Manual dos Casos de Teste no Moodle 75
Tabela 15 Mapeamento dos Requisitos de Segurança do CesarFTP 76
Tabela 16 Casos de Teste Gerados para o CesarFTP 77
Tabela 17 Resultados Obtidos com a Execução Manual dos Casos de Teste no CesarFTP 77

Lista de Siglas

DARPA	*Defense Advanced Research Projects Agency*
DoS	*Denial of Service*
ETSI	*European Telecommunication Standards Institute*
FSM	*Finite State Machine*
IPv6	*Internet Protocol version* 6
MONID	*Monitoring based Intrusion Detection tool*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OWASP	*The Open Web Application Security Project*
R2L	*Remote to Local*
RBAC	*Role-Based Acess Control*
SANs	*Stochastic Automata Networks*
SSL	*Secure Socket Layer*
STAGE	*STAte-based test GEnerator*
SUT	*System Under Test*
TTCN-3	*Testing and Test Control Notation version* 3
U2R	*User to Root*
U2TP	UML 2.0 *Testing Profile*
UIO	*Unique Input/Output*
UML	*Unified Modeling Language*
VFSM	*Variable Finite State Machine*
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*
XSS	*Cross-Site Scripting*

Sumário

1 Introdução 13

2 Teste de Software 15
2.1 Definições de falha, erro e defeito 15
2.2 Casos de Teste 16
2.3 Níveis de Teste 16
2.4 Técnicas de Teste 17
2.5 Teste de Segurança 17

3 Modelos Formais 19
3.1 Por que utilizar modelos formais? 20
3.2 Alguns Modelos com Aplicação na Área de Teste de Software 20
3.2.1 TTCN-3 21
3.2.2 Teste Estatístico baseado em Modelos 22
3.2.3 Notação Z 24
3.2.4 Lógica Temporal 26
3.3 UML 27
3.4 Considerações Finais 29

4 Taxonomias de Segurança 31
4.1 Taxonomia proposta pela DARPA 31
4.2 Taxonomia proposta por Aslam 33
4.3 Taxonomia proposta por Bazaz 35
4.4 Taxonomia proposta por Weber 36
4.5 Considerações Finais 38

5 Trabalhos Relacionados 39
5.1 Técnicas de Modelagem de Aspectos de Segurança 39
5.1.1 UMLSec 40
5.1.2 SecureUML 42
5.2 Ferramentas para Automatização de Teste 44
5.2.1 JMeter 45
5.2.2 STAGE 47
5.2.3 HP QAInspect 49
5.3 Considerações Finais 51

6 Especificação e Geração de Casos de Teste 54
6.1 Motivação e Objetivos 54
6.2 Modelagem dos Aspectos de Segurança 56
6.3 Geração dos Casos de Teste 60

6.4 Considerações Finais 61

7 Estudos de Caso 64
7.1 OSCommerce 64
7.2 TPC-W 69
7.3 Moodle 72
7.4 CesarFTP 75
7.5 Considerações Finais 78

8 Conclusão 79

Referências 81

1 Introdução

Com o crescente número de serviços providos pela Internet, é considerável o aumento na quantidade de dados trocados entre computadores e usuários. A expansão da Web é tamanha que superou seu objetivo inicial, o de ser apenas uma forma de conectar informações armazenadas em diferentes locais. Hoje em dia já é possível fazer compras, transações bancárias e inclusive declarar Imposto de Renda através da Internet, o que faz com que dados confidenciais sejam transmitidos pela rede. Devido a esta característica, a área de tecnologia de informação deparou-se com uma nova preocupação de seus usuários: a segurança e confidencialidade dos dados enviados [1].

Baseando-se nos princípios de teste de software, que visam encontrar problemas nos mais variados sistemas, foi criada uma área dedicada apenas à realização de testes de segurança em *softwares* [2]. Esta visa avaliar componentes que podem comprometer a segurança do sistema, como entradas de dados, interfaces e protocolos de comunicação. Entretanto, definir manualmente os testes que devem ser realizados demanda tempo e esforço dos engenheiros de teste, que devem conhecer detalhadamente a aplicação a ser testada, bem como o modo que ocorre a troca de dados. Caso contrário, o conjunto de casos de teste escolhido pode não encontrar falhas críticas no sistema, que serão descobertas apenas após a implantação deste em um ambiente real [3].

Desta forma, nos últimos anos vem sendo empregada uma abordagem diferente para a realização de testes, denominada teste baseado em modelos [4, 5]. Nesta, a partir do modelo formal de uma aplicação e da especificação dos critérios que se deseja testar, uma série de casos de teste são gerados, na maior parte das vezes de forma automatizada. Entretanto, apesar da popularização desta técnica, ela ainda não é amplamente empregada para analisar aspectos de segurança, área de estudo deste trabalho. O principal motivo se deve à dificuldade de especificar critérios de segurança utilizando modelos, devido a variedade de detalhes associados à área que devem ser modelados.

Para se definir um modelo que especifique segurança, é de suma importância conhecer diversos ataques e vulnerabilidades, a fim de propor um que permita descrever a maior parte dos comportamentos. O primeiro passo para se propor um novo modelo, ou mesmo para avaliar os existentes, é estudar algumas taxonomias de segurança [6]. Isto auxilia a identificar padrões de comportamentos entre ataques ou vulnerabilidades, permitindo a construção de modelos coerentes e precisos. Após a definição do modelo, é preciso extrair as informações relevantes deste e gerar, ao término da análise, os casos de teste que devem ser aplicados ao software quando este estiver pronto.

Sendo assim, este trabalho propõe uma estratégia para especificar aspectos de segurança durante a elaboração do modelo da aplicação, bem como automatizar o processo de teste de software provendo uma forma de gerar os casos de teste a partir deste modelo. O trabalho tem dois objetivos principais: auxiliar durante a fase de desenvolvimento do *software*, prevendo situações que possam comprometer sua segurança, e permitir a geração de casos de teste a partir das informações inseridas. Para descrever os aspectos de segurança, foram definidos alguns estereótipos UML representando vulnerabilidades de segurança. Para auxiliar este processo, utilizou-se como base uma lista com as dez vulnerabilidades mais críticas para aplicações Web, conhecida como Top Ten List e elaborada pelo projeto OWASP [7]. Já na parte de geração dos casos de teste, utilizou-se o método conhecido como *Unique Input Output* (UIO) [8], que utiliza uma máquina de estados finitos com entrada e saída cujo resultado são seqüências descrevendo os passos a serem seguidos pelo testador.

Este trabalho está estruturado da seguinte forma. O Capítulo 2 descreve aspectos relacionados à área de teste de software, tais como a sua importância e os tipos de teste existentes, enquanto que o Capítulo 3 apresenta alguns modelos formais e ressalta as características, vantagens e desvantagens de cada. O Capítulo 4 explica as taxonomias de segurança estudadas ao longo do trabalho e que auxiliaram no processo de especificação dos estereótipos de segurança, enquanto que o Capítulo 5 compara o trabalho aqui apresentado, com outros da área de modelagem e de automatização de teste. O Capítulo 6, por sua vez, descreve o trabalho desenvolvido, explicando a contribuição do trabalho, a motivação e objetivos dele, bem como os passos seguidos para especificar os estereótipos de segurança e gerar os casos de teste equivalentes. O Capítulo 7 explica os casos de teste executados em cada estudo de caso, demonstrando o processo de modelagem dos requisitos de segurança e os resultados obtidos quando aplicados aos softwares. Por fim, o Capítulo 8 apresenta as conclusões obtidas com este trabalho, bem como identifica situações não abordadas em seu escopo e as sugere como trabalhos futuros.

2 Teste de Software

Nos últimos anos, a busca pela excelência no desenvolvimento de software vem preocupando os mais diversos tipos de empresas da área de Informática. Visando produzir software com qualidade, as empresas têm implantado em suas dependências equipes voltadas exclusivamente ao teste de software [9, 10]. Em função do tamanho dos sistemas desenvolvidos ultimamente, bem como a complexidade destes e o número de pessoas envolvidas, dividir a etapa de produção de um software em fase de desenvolvimento e fase de testes torna-a mais eficaz, mais rápida e mais confiável [3].

A área de teste de software pode ser subdividida em diversas outras menores, como na execução de testes funcionais, de usabilidade ou de desempenho, por exemplo, aplicadas de acordo com as necessidades do usuário. Neste trabalho, será abordada a de segurança, escolhida por diversos motivos. Com a quantidade de serviços prestados via Internet e com o alto grau de confidencialidade das informações trocadas, um pré-requisito de algumas aplicações é que sejam seguras, e uma das formas de se verificar esta característica é realizando testes de segurança.

Sendo assim, este capítulo tem como objetivo abordar questões referentes a teste de software. Nas próximas seções serão apresentados alguns conceitos relacionados, as vantagens em se executar testes (e testes baseados em modelos), além de fundamentar a importância de se avaliar a segurança de um sistema antes de colocá-lo em fase de produção.

2.1 Definições de falha, erro e defeito

Existe uma série de conceitos semelhantes na área de teste de software, mas os que mais se confundem são as definições de falha, erro e defeito. Embora para alguns tenham o mesmo significado, eles possuem características que os diferem. Uma breve definição de cada um destes conceitos pode ser vista a seguir [11].

- Falha (*fault*): Uma falha é a causa hipotética de um erro. Quando provoca um erro é chamada de ativa (*active fault*); caso contrário, é chamada de adormecida (*dormant fault*).
- Erro (*error*): Um erro é o desvio que faz com que o serviço entregue seja diferente do especificado, ou seja, ele é a parte do estado total do sistema que pode levar a subseqüentes defeitos no serviço. Entretanto, é importante dizer que nem todo erro causa um defeito. Isto acontece apenas quando ele alcança algum estado externo do sistema.

- Defeito (*failure*): Um defeito é quando o serviço entregue é diferente do esperado. Ele pode ocorrer simplesmente porque o resultado não confere com o descrito na especificação ou porque a especificação não descreve adequadamente o funcionamento do sistema.

A Figura 1 mostra a relação existente entre falha, erro e defeito.

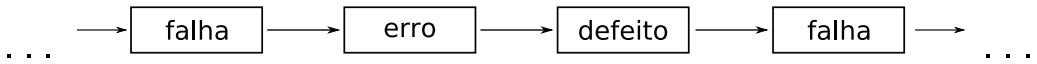

Figura 1 – Relação entre falha, erro e defeito

2.2 Casos de Teste

Quando se decide testar um software, a primeira tarefa é fazer uma análise do sistema em questão e definir quais itens serão testados. Feito isso, para cada item devem ser elaborados casos de teste. Casos de teste, ou *test cases*, servem para definir um teste e o que será testado com ele, como um roteiro a ser seguido pelo testador [12]. Para garantir o bom funcionamento de um software, inúmeros casos de teste devem ser criados, um complementando o outro, a fim de cobrir todas as situações possíveis de execução [13].

A documentação de um caso de teste deve compreender uma série de dados. Estão entre estes um identificador do caso de teste, um estado inicial, uma seqüência de operações de teste e os resultados esperados [14].

2.3 Níveis de Teste

Dependendo do estado em que se encontra a fase de desenvolvimento de um software, diferentes tipos de teste podem ser realizados nele. Os níveis de teste, como são chamados, podem ser classificados da seguite forma [11]:

- Teste Unitário: Este nível consiste em testar cada um dos módulos de maneira separada; tem por objetivo verificar a correta implementação de cada um;

- Teste de Integração: Consiste em testar um grupo de componentes ou módulos, visando validar tanto a interface de comunicação entre eles, como a capacidade de trabalharem juntos;

- Teste de Sistema: Consiste em realizar testes no sistema como um todo, comparando o funcionamento observado com o descrito na especificação;

- Teste de Aceitação: Consiste em realizar testes no sistema como um todo, mas comparando com os requisitos fornecidos pelo usuário a fim de verificar se o produto final atende às exigências deste.

2.4 Técnicas de Teste

Um teste pode ser aplicado a um sistema por diversos motivos: buscar por problemas, verificar a eficiência, o desempenho, etc. Dependendo da finalidade com que é executado, um teste pode ser classificado e, em categorias, algumas descritas a seguir [15]:

- Teste Funcional: Conhecido também como teste de caixa-preta, ou *black-box*, baseia-se na especificação do software e é aplicado ao sistema como um todo. O testador tem conhecimento apenas das entradas e saídas do sistema, e a partir de um oráculo[1] avalia se o software funciona de acordo com o esperado ou não;

- Teste Estrutural: Ao contrário do teste funcional, no teste estrutural o testador tem conhecimento do código-fonte do sistema testado. Por este motivo é conhecido como teste de caixa-branca, ou *white-box*. Pode ser realizado enquanto o software ainda está em desenvolvimento;

- Teste Baseado em Falhas: Esta técnica baseia-se em inserir, intencionalmente, um conjunto de falhas no *software*, com o objetivo de definir um critério de parada para a execução dos testes. Assume-se que este conjunto equivale a falhas típicas, ou seja, que sejam do mesmo tipo, ocorram na mesma freqüência e possuam as mesmas características que as falhas reais. Uma forma de selecionar este conjunto é utilizar dados históricos obtidos a partir de versões anteriores ou projetos similares.

2.5 Teste de Segurança

Com o grande número de aplicações comerciais na Internet, garantir que um software é seguro tornou-se essencial. A fim de buscar por possíveis vulnerabilidades nestes sistemas, criou-se um novo ramo na área de teste voltado exclusivamente para a realização de teste de segurança em software.

Teste de segurança é classificado como um teste de aspectos não funcionais (uma subdivisão do teste de sistema), assim como os testes de desempenho e *stress*. Especialistas deste ramo devem possuir um conhecimento aprofundado sobre redes de comunicação, arquitetura de computadores, protocolos, programação e vulnerabilidades. Além disso, devem ser persistentes, visto que muitas vezes um defeito só acontece em decorrência da combinação de uma

série de outros ataques menores. Uma técnica interessante para se explorar a segurança de um sistema é dividí-lo em áreas, focando em partes onde uma vulnerabilidade tem mais chances de ocorrer. Esta abordagem provê um nível mais alto de segurança do que o clássico modelo de teste de caixa-preta [17].

Dizer que uma aplicação é segura requer a execução de uma série de testes de forma exaustiva e repetitiva, buscando por conexões suscetíveis a roubo de dados e informações. Outra preocupação é com os ataques conhecidos como *Denial of Service*(DoS). Estes prejudicam o software de tal forma que podem resultar na negação de algum serviço ou até mesmo dar privilégios de acesso a usuários que não o possuem.

Softwares podem tornar-se vulneráveis em função de diversos fatores, tais como:

- Intenção criminosa de algum *hacker* em danificar o sistema, roubar informações, causar uma negação de serviço, invadir a privacidade do usuário;
- Erros cometidos por pessoas responsáveis pela manutenção do sistema que modificam, destroem ou comprometem dados devido a má informação ou pouco conhecimento.

Para identificar vulnerabilidades, a equipe de teste deve fazer-se passar por um *hacker* e atacar o sistema, explorando todas as brechas possíveis. Entretanto, é preciso seguir uma metodologia previamente definida, a fim de cobrir a maior parte de protocolos e funcionalidades do *software*. Uma estratégia possível é fazer um levantamento das camadas, protocolos e módulos presentes no sistema e, a seguir, estudar minuciosamente a especificação de cada um destes. Durante este estudo devem ser destacados mecanismos que possam ter sido mal implementados ou que possam apresentar problemas sob determinadas condições. Concluída esta etapa, é preciso implementar programas de teste para avaliar cada uma das possíveis vulnerabilidades, tendo como alvo o sistema em questão. Por fim, os testadores devem analisar o comportamento do sistema e concluir se o sistema foi atacado ou não.

É importante dizer que mesmo com a execução exaustiva de testes de segurança, não é possível assegurar que um software é totalmente seguro. Isto por que o tempo dedicado a realização de testes deste tipo não permite cobrir todas as possibilidades e combinações de ataques. Sabendo da quantidade de problemas de segurança que as aplicações apresentam, bem como da dificuldade em testá-las, este trabalho visa integrar as práticas de teste de software com a capacidade de representação de modelos formais, a fim de automatizar o processo de teste de segurança. Sendo assim, o próximo capítulo deste trabalho descreve alguns modelos formais com aplicação na área de teste, com o objetivo de auxiliar na escolha do mais indicado para representar os aspectos de segurança.

3 Modelos Formais

Com a evolução dos sistemas de computadores e a quantidade de funcionalidades providas por estes, os softwares tornaram-se estruturas tão complexas que mesmo testá-los tornou-se difícil. Como conseqüência, versões instáveis têm sido liberadas para os usuários finais, contendo inúmeros problemas de implementação. Diversos motivos justificam este comportamento, tais como o curto prazo dispendido com a fase de testes e a má especificação da arquitetura. O curto prazo faz com que as aplicações sejam desenvolvidas e testadas com pressa, fazendo com que uma série de falhas sejam ignoradas. Já os problemas com a especificação, como por exemplo uma especificação confusa, superficial ou ambígua, pode resultar em divergências que serão descobertas apenas ao final da fase de desenvolvimento, quando corrigí-las custa caro. A fim de resolver problemas como estes, algumas empresas têm mudado a forma como descrevem os requisitos e a arquitetura de seus sistemas. Eles vêm substituindo a descrição em linguagem natural, comumente usada, por uma abordagem mais consistente, utilizando modelos formais [5, 18].

Um modelo formal nada mais é do que uma forma de representar, através de uma linguagem formal específica, uma outra entidade, podendo esta ser um objeto físico ou mesmo outro modelo. São utilizados para comunicar percepções e transferir conhecimento a outras pessoas, sendo que esta formalização pode ser representada através de desenhos, gráficos e até mesmo equações matemáticas. Quando aplicados ao desenvolvimento de software, servem para descrever propriedades e para assegurar a integridade do sistema, caso a implementação baseie-se na modelagem previamente realizada. Mais do que isto, a partir da descrição de um modelo formal é possível simular o comportamento de um sistema, identificar padrões e necessidades deste, além de poder detectar possíveis falhas na especificação [10].

Um modelo formal é descrito através de uma notação própria, também conhecida como linguagem formal. Com o número de notações existentes, a escolha de uma que atenda aos requisitos do projeto e consiga representar o que se deseja é crucial. Uma observação a ser feita é que a grande variedade de notações dificulta a popularização dos modelos formais, já que, por não haver uma padronização, é necessário realizar um estudo prévio dos existentes antes de optar por qual será utilizado.

Sabendo-se da importância do uso de modelos formais, este capítulo tem como objetivo ressaltar as vantagens de se empregá-los durante o ciclo de desenvolvimento de software, bem como apresentar como eles podem auxiliar as áreas de teste de software e de segurança, foco deste trabalho.

3.1 Por que utilizar modelos formais?

Conforme mencionado, modelos formais são metodologias utilizadas para especificar determinadas características de um sistema, podendo representar, por exemplo, o comportamento esperado do software, as estruturas existentes neste ou até mesmo como ele deve ser testado. Por serem precisos, podem ser utilizados em diversas etapas do processo de desenvolvimento de software, desde sua especificação até o levantamento dos casos a serem testados [14].

A principal vantagem de aplicá-los é que, como são fortemente baseados em matemática e lógica formal, permitem validar, por exemplo, se uma especificação foi elaborada corretamente ou se apresenta alguma ambigüidade. Outras vantagens obtidas com o uso de modelos formais são que eles ajudam a melhorar a qualidade do software, tornando-o mais robusto se aplicados de maneira correta, e levam a uma redução nos custos do projeto, já que detectam uma grande quantidade de erros já na fase de especificação.

A aplicação de modelos formais pode ser muito útil para diversas áreas da computação, mas uma das mais importantes, sem dúvidas, é a de segurança de redes [5]. Hoje em dia, a maior parte das empresas presta serviços via Internet, sendo que a quantidade de dados confidenciais trocados é muito grande. Para poder lidar com esta situação, as aplicações e protocolos responsáveis pela comunicação devem ser os mais robustos possíveis, evitando assim o roubo de informações e diversos outros ataques. Como dito anteriormente, o uso de modelos formais garante que não existem ambigüidades em especificações. Desta forma, utilizá-los como base para descrever aplicações, ou mesmo requisitos de segurança, auxilia na identificação de muitas vulnerabilidades antes da implementação, quando corrigí-las é mais rápido e barato [14].

3.2 Alguns Modelos com Aplicação na Área de Teste de Software

A construção de um modelo formal do sistema desejado durante a fase de projeto de software pode melhorar a qualidade do produto final, pois fornece uma estrutura passível de análise e que permite a detecção de alguns defeitos. Ele descreve características do sistema através de uma linguagem própria, conhecida como linguagem formal ou notação formal.

Existem diversas notações para se descrever um modelo formal, cada uma com suas peculiaridades. Considerando a complexidade e a limitação de algumas destas, a escolha da linguagem apropriada é fundamental para o sucesso do projeto, sendo preciso avaliar diversos antes de decidir qual será utilizado. Inúmeros fatores devem ser considerados durante a escolha de uma notação, tais como o vocabulário proposto, a capacidade de representação da linguagem e a clareza com que esta expõe as informações [10]. Sendo assim, o objetivo desta seção é apresentar algumas notações já conhecidas e que podem, de alguma forma, representar características de segurança, bem como auxiliar durante o processo de teste de software.

3.2.1 TTCN-3

Testing and Test Control Notation version 3 (TTCN-3) [19] é uma linguagem de teste de padrão internacional desenvolvida pelo *European Telecommunication Standards Institute* (ETSI) [20]. Foi projetada com o objetivo de permitir a execução de diversos tipos de testes para uma série de aplicações, independente de plataforma. Para avaliar os sistemas, baseia-se na troca de mensagens (requisições e respostas), o que faz com que seja amplamente utilizada na área de telecomunicações no teste de dispositivos de rede e de protocolos de comunicação. Um exemplo é o teste do recente protocolo *Internet Protocol version* 6 (IPv6), cujos resultados podem ser vistos no trabalho de Sabiguero *et al.* [21]. Outras características da TTCN-3 são a modularidade, a facilidade de descrição de testes e a semelhança com as demais linguagens de programação.

Para realizar um teste utilizando TTCN-3, o usuário deve seguir corretamente as especificações de uso da linguagem. O modelo básico de teste empregado é composto por duas entidades: uma representando o sistema testado (*System Under Test*, SUT) e outra representando uma suíte de teste abstrata, que contém as informações que serão extraídas do SUT, os casos de teste e o controle da execução dos mesmos. A Figura 2 apresenta o modelo de arquitetura utilizada pela TTCN-3.

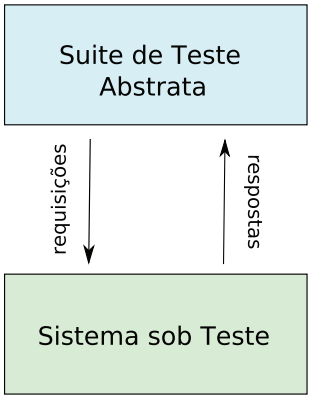

Figura 2 – Modelo de arquitetura utilizada pela TTCN-3

Um exemplo de uso da linguagem TTCN-3, apresentado por Stepien [22], descreve o teste de um serviço meteorológico. Conforme mencionado, o sistema é avaliado de acordo com as informações fornecidas em resposta às requisições feitas pela suite de teste. Desta forma, quando a suite de teste é elaborada, inicialmente devem ser descritas as estruturas de dados utilizadas, bem como as regras de validação dos parâmetros. A seguir, deve-se especificar os casos de teste que serão executados, indicando as seqüências de mensagens que devem ser enviadas para e recebidas do SUT. Por último define-se a ordem de execução dos vários casos de teste.

No sistema em questão (serviço meteorológico), o testador deve enviar uma requisição

contendo a localidade, a data e o tipo de relatório desejado, e receber como resposta a confirmação dos parâmetros enviados acrescidos da temperatura no local, a velocidade do vento e as condições do tempo. A Figura 3 apresenta o formato, os dados inseridos nas mensagens e as regras para avaliação das respostas, enquanto a Figura 4 descreve a seqüência de execução das mensagens.

```
module SimpleWeather
{
   type record weatherRequest
   {
      charstring location,
      charstring date,
      charstring kind
   }

   type record weatherResponse
   {
      charstring location,
      charstring date,
      charstring kind,
      integer temperature,
      integer windVelocity,
      charstring conditions
   }

   template weatherRequest ParisWeatherRequest :=
   {
      location := "Paris",
      date := "15/06/2006",
      kind := "actual"
   }

   template weatherResponse ParisResponse :=
   {
      location := "Paris",
      date := "15/06/2006",
      kind := "actual",
      temperature := (15..30),
      windVelocity := (0..20),
      conditions := "sunny"
   }

   . . .

}
```

Figura 3 – Formato das mensagens [22]

```
module SimpleWeather
{

   . . .

   type port weatherPort message
   {
      in weatherResponse;
      out weatherRequest;
   }

   type component MTCType
   {
      port weatherPort weatherOffice;
   }

   testcase testWeather() runs on MTCType
   {
      weatherOffice.send(ParisWeatherRequest);
      alt
      {
         [] weatherOffice.receive(ParisResponse)
         {
            setverdict(pass)
         }

         [] weatherOffice.receive
         {
            setverdict(fail)
         }
      }
   }

   control
   {
      execute (testWeather())
   }

}
```

Figura 4 – Execução do teste [22]

Após a execução dos testes, é gerado um arquivo contendo os resultados obtidos, informando se eles foram executados com sucesso, se falharam ou se algum erro ocorreu durante o processo.

3.2.2 Teste Estatístico baseado em Modelos

Com a quantidade de funcionalidades providas, os softwares tornaram-se estruturas tão complexas que mesmo testá-las tornou-se difícil. Esta situação é agravada pelo pouco tempo normalmente dedicado às atividades de teste, já que não é suficiente prever apenas alguns casos e testá-los. É preciso analisar o sistema e concluir quais são as partes mais utilizadas e mais

críticas, a fim de pelo menos explorá-las mais do que módulos não tão comprometedores.

A fim de permitir este tipo de análise, foi desenvolvida a técnica de teste de software estatístico baseado em modelos. Com esta estratégia é possível descrever um modelo do sistema utilizando, por exemplo, Cadeias de Markov ou Redes de Autômatos Estocásticos (*Stochastic Automata Networks*, SANs), e atribuir probabilidades às transições de um estado para outro. A partir destas probabilidades são gerados automaticamente casos de teste para os caminhos mais utilizados do software, tornando-o assim mais robusto. O teste estatístico pode ser considerado um guia sobre como o software será testado, baseado em distribuições de probabilidades, populações e amostras. A Figura 5 apresenta um exemplo de descrição utilizando autômatos finitos.

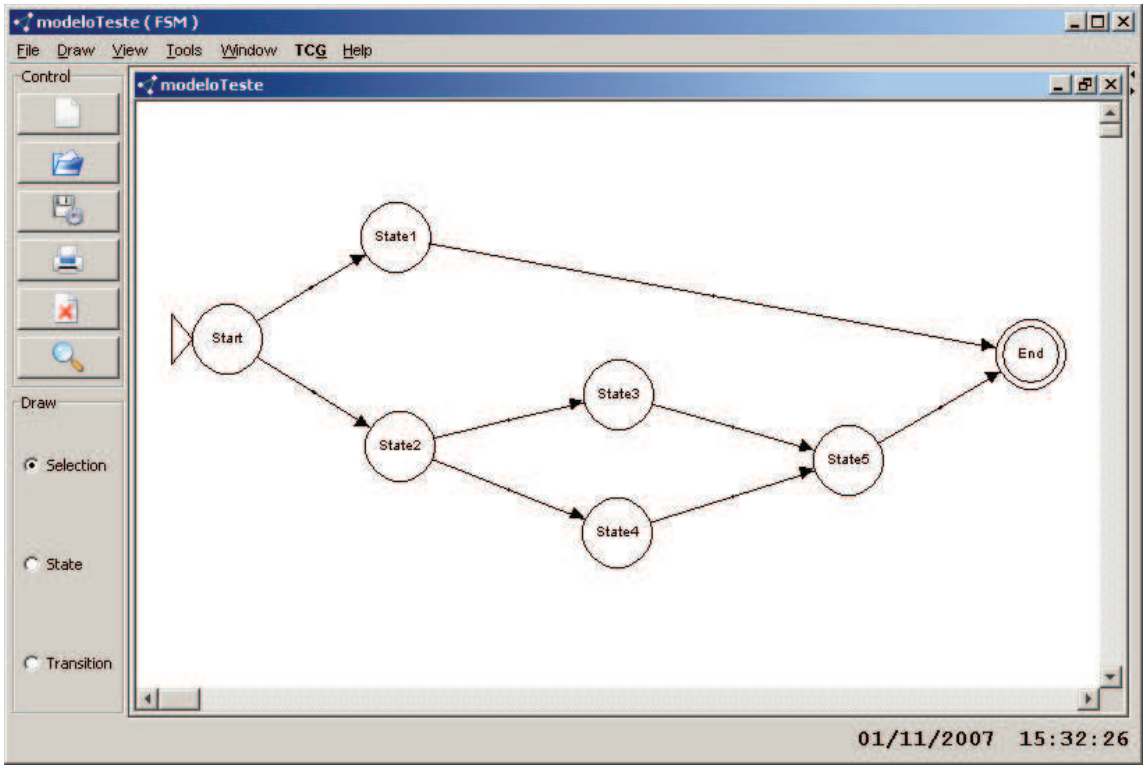

Figura 5 – Modelo de autômatos finitos

Uma ferramenta desenvolvida para auxiliar no processo de teste estatístico é o *STAte-based test GEnerator* (STAGE) [23, 24]. O objetivo desta é gerar casos de teste funcionais e *scripts* a partir do modelo da aplicação, descrito através de autômatos finitos. Com base neste modelo e nas probabilidades associadas às transições entre os estados, o STAGE identifica os caminhos que podem ser percorridos pelo software e determina que combinações devem ser testadas.

Conforme mencionado, a geração dos casos de teste é feita de acordo com os caminhos que podem ser percorridos pelo software. A ferramenta verifica, desde o estado inicial, todas as formas de se chegar ao estado final, garantindo assim que todos os caminhos tenham sido cobertos. Cada caminho equivale a um caso de teste diferente. Por fim, a ferramenta ainda permite

gerar *scripts* de teste para determinados *software*, visando automatizar também o processo de execução dos testes. Maiores detalhes sobre a ferramenta STAGE estão no Capítulo 5.

3.2.3 Notação Z

A notação Z [25] é uma especificação formal utilizada para descrever sistemas de computadores baseada na teoria dos conjuntos proposta por Zermelo-Frankel e em lógica de primeira ordem. O principal problema associado ao uso de especificações formais é que, quanto maior é o sistema que se quer representar, mais complexas e incompreensíveis elas se tornam. A fim de resolver este problema, a notação Z utiliza um modelo previamente definido que auxilia na construção da especificação, pois permite combinar diferentes seções de esquemas matemáticos através de descrições formais de texto.

Este modelo que auxiliará na descrição através da notação Z pode ser construído de diversas maneiras. A mais conveniente utiliza modelos baseados em estados, devido a facilidade de descrição e compreensão. Uma descrição aceitável de um modelo, por exemplo, é a presença de um estado abstrato e uma seqüência de transições representando as operações dentro dele. Mesmo que pareça simples demais, é possível especificar o comportamento de uma série de sistemas utilizando apenas esta notação. O modelo não deve ser influenciado por questões referentes à implementação, e nem deve ser formal demais que comprometa o seu entendimento. Ele deve ser simples o bastante para representar apenas o funcionamento e as restrições desejadas do sistema [26].

Um exemplo simples apresentado no manual de referência da Notação Z [27] propõe a implementação de um sistema para armazenar as datas de aniversário de algumas pessoas. O primeiro passo para modelar este sistema é definir os tipos básicos a serem utilizados (conjuntos), bem como definir as variáveis e condições de operabilidade. Neste caso, temos dois conjuntos principais de dados, os nomes dos aniversariantes e suas respectivas datas de aniversário. Além disso, é preciso definir os tipos de cada variável e estabelecer que o número de nomes da lista deve ser igual ao número de aniversários armazenados. Caso contrário, o sistema será considerado inconsistente. A Equação 3.1 apresenta a descrição dos tipos básicos de dados, enquanto que a Equação 3.2 demonstra a declaração de um conjunto (*known*), uma função de mapeamento utilizando este conjunto (*birthday*) e a relação de igualdade entre estes (o tamanho do conjunto *known* é igual ao domínio da função de mapeamento *birthday*).

$$[NAME, DATE] \tag{3.1}$$

$$\begin{array}{l} known : NAME \\ birthday : NAME \mapsto DATE \\ \hline known = \mathrm{dom}\ birthday \end{array} \tag{3.2}$$

Um possível preenchimento deste sistema poderia ser feito com os registros John - 25/03, Mike - 20/12 e Susan, 20/12. O formato de declaração dos registros pode ser visto na Equação 3.3.

$$\begin{aligned} known &= \{John, Mike, Susan\} \\ birthday &= \{John \mapsto 25/03, \\ &\quad\ Mike \mapsto 20/12, \\ &\quad\ Susan \mapsto 20/12\} \end{aligned} \tag{3.3}$$

Usando a notação Z, é possível também formalizar o comportamento das operações do sistema, como por exemplo o de adicionar um novo aniversariante ao catálogo. Para descrever esta atividade, o primeiro passo é estabelecer uma pré-condição determinando que o novo nome não pode existir na base de dados. Caso contrário teríamos uma ambiguidade nesta, pois para uma mesma pessoa seriam atribuídas duas datas de aniversário diferentes. A partir de então, caso esta pré-condição seja satisfeita, a base de dados deve ser expandida através da união deste novo registro. A Equação 3.4 apresenta a descrição desta etapa do processo.

$$\begin{array}{l} name? : NAME \\ date? : DATE \\ \hline name? \notin known \\ birthday' = birthday \cup \{name? \mapsto date?\} \end{array} \tag{3.4}$$

Como é possível perceber, a notação Z não foi desenvolvida com o intuito de descrever testes, foco deste trabalho, mas de modelar formalmente os sistemas em si. Entretanto, diversos trabalhos já experimentaram utilizar a notação Z para formalizar os critérios de teste de software. Um exemplo é o trabalho de Vilkomir e Bowen [28], que propõe algumas estruturas descritas usando esta notação para formalizar os critérios de teste que avaliam o fluxo de controle do software. A principal vantagem do uso da notação Z para formalização de critérios

de teste é a unicidade, já que o nível de detalhamento permitido evita que uma descrição seja compreendida de diferentes formas, como acontece com a linguagem natural.

3.2.4 Lógica Temporal

Em lógica, o termo lógica temporal significa representar proposições qualificadas em relação ao tempo, utilizando-se de regras e simbolismos. Com esta abordagem, é possível expressar sentenças tais como "determinado fato sempre acontece", "determinado fato acontece eventualmente" ou "determinado fato acontece até que tal medida seja tomada" [29].

Na área da computação, a lógica temporal possui uma grande aplicação na descrição de requisitos de hardware ou software. Através dela é possível especificar comportamentos do software de forma precisa, estabelecendo restrições e pré-condições para que determinada operação ocorra. Desta forma, com a descrição em lógica temporal pode-se descrever situações como "nunca haverá dois dispositivos acessando o mesmo recurso simultaneamente" e "sempre se passa por um estado A antes de ir para um estado B".

Devido a capacidade de representar casos como estes, lógica temporal vem sendo amplamente utilizada não apenas durante o desenvolvimento de software, mas também durante a fase de teste de software [30]. O processo de teste com lógica temporal pode ser dividido em duas etapas: na primeira, é elaborado um documento contendo a descrição funcional do sistema; na segunda, cria-se um outro documento definindo as restrições de uso, ou seja, especificando como ocorre a transição dos estados e em que situações estas devem acontecer. Terminada a parte de descrição, estes dois arquivos são analisados por um verificador formal que realizará uma série de testes e indicará em que situações o comportamento do sistema viola as restrições impostas.

Um exemplo do uso de lógica temporal na área de segurança é o *framework Monitoring based Intrusion Detection tool* (MONID) [31]. Seu objetivo é detectar intrusões através da comparação do comportamento do sistema real com o do esperado, previamente descrito através de lógica temporal. Para tanto, foi implementado um algoritmo que captura o comportamento do sistema, o transforma em uma fórmula e a compara com a fórmula previamente feita da especificação. Sempre que a especificação for violada (isso acontece quando a fórmula lógica retorna valor falso), deve ser disparado um "alarme" indicando desvio de comportamento.

Suponha uma restrição que estabeleça que sempre que um usuário conecta-se ao sistema, ele demora até 100 unidades de tempo para se desconectar. A Equação 3.5 demonstra como seria a especificação desta regra que se deseja monitorar.

$$\begin{aligned} \min\ & EvTimedLogout(string\ k, double\ t, double\ \delta) = (time - t \leq \delta) \\ & \wedge ((action = logout \wedge userId = k) \vee \bigcirc EvTimedLogout(k, t, \delta)) \\ \text{mon}\ & M_1 = \text{Always}((action = login) \rightarrow EvTimedLogout(userId, time, 100)) \end{aligned} \tag{3.5}$$

Após a descrição da regra, é preciso analisar o comportamento do sistema real. Para tanto, será considerada a seguinte sequência de eventos *e1* e *e2*:

$$e1 = userId = \text{“}Bob'', action = login, time = 17.0$$
$$e2 = userId = \text{“}Bob'', action = logout, time = 150.0$$

Para as seqüências *e1* e *e2*, a fórmula final M1 ficaria como descrito pela Equação 3.6.

$$\begin{aligned} eval(M_1, e1) = & EvTimedLogout(\text{“}Bob'', 17.0, 100) \wedge \\ & \text{Always}((action = login) \rightarrow EvTimedLogout(userId, time, 100)) \end{aligned} \tag{3.6}$$

Analisando-se a Equação 3.6, percebe-se que os parâmetros informados durante o evento *e1* foram substituídos na fórmula original. Durante a execução do segundo evento *e2*, o predicado $(time - t \leq \delta)$ foi instanciado como (150.0 - 17.0 <= 100), o que é falso. Desta forma, toda a sentença é considerada falsa, e o comportamento considerado uma tentativa de intrusão.

Por fim, o uso de lógica temporal não garante que a implementação do sistema está de acordo com o modelo, mas provê uma forma de avaliar se a especificação deste não é ambígua ou se não fere as restrições impostas. Sendo assim, após a implementação do sistema, é preciso executar novos testes na aplicação e comparar com os resultados previamente obtidos, a fim de validar se o sistema implementado é vulnerável ou não.

3.3 UML

A *Unified Modeling Language*[1] (UML) [32] é uma linguagem padrão de modelagem mantida pelo *Object Management Group* (OMG) [33], amplamente utilizada para especificar sistemas orientados a objeto, seu comportamento e interação. Ela define notações para construir diferentes diagramas, cada qual representando uma visão particular de um artefato a ser modelado. Devido a sua simplicidade, abrangência e flexibilidade, já que é possível expandir a linguagem com os chamados *profiles*, ela vem sendo amplamente utilizada como suporte para testes baseados em modelo [34]. Isto porque, além de descrever o comportamento do sistema, a UML também permite inserir restrições ao modelo através de uma linguagem própria para

este fim, chamada *Object Constraint Language* (OCL) [35]. Com esta, é possível especificar invariantes, pré e pós-condições, definindo precisamente como devem ser alguns estados do sistema.

Originalmente, a UML foi desenvolvida com o objetivo de ser apenas um instrumento para representar características do sistema. Entretanto, com o aumento da complexidade destes, surgiu a necessidade de integrar-se à modelagem o conceito de teste de software. Neste contexto, o OMG propôs a criação do UML 2.0 *Testing Profile* (U2TP) [36], que provê mecanismos para especificar testes de acordo com os aspectos estruturais e comportamentais do sistema. O objetivo com este *profile* é realizar testes efetivos, eficientes e, quando possível, automatizados, seguindo o modelo UML.

O U2TP define uma linguagem para projeto, visualização, especificação, análise, construção e documentação das características do sistema testado. Pode ser utilizado sozinho, para especificar apenas os pontos do sistema que serão testados, ou de forma integrada com a UML, para descrever tanto o sistema, como os casos de teste.

Um exemplo do uso do U2TP pode ser visto nas Figuras 6 e 7, extraídas do trabalho apresentado por Biasi e Becker [37]. A Figura 6 apresenta a modelagem UML de um sistema de vendas, com a inserção do estereótipo «sut», do U2TP, destacando as classes do software que serão testadas. Já a Figura 7 ilustra o modelo U2TP que descreve como o sistema será avaliado, incluindo os estereótipos «testcontext», que representa a definição do pacote de teste, «testcase», que indica a instância de cada um dos casos de teste a serem executados, e «setup», que permite associar à operações definidas por um diagrama de seqüência externo.

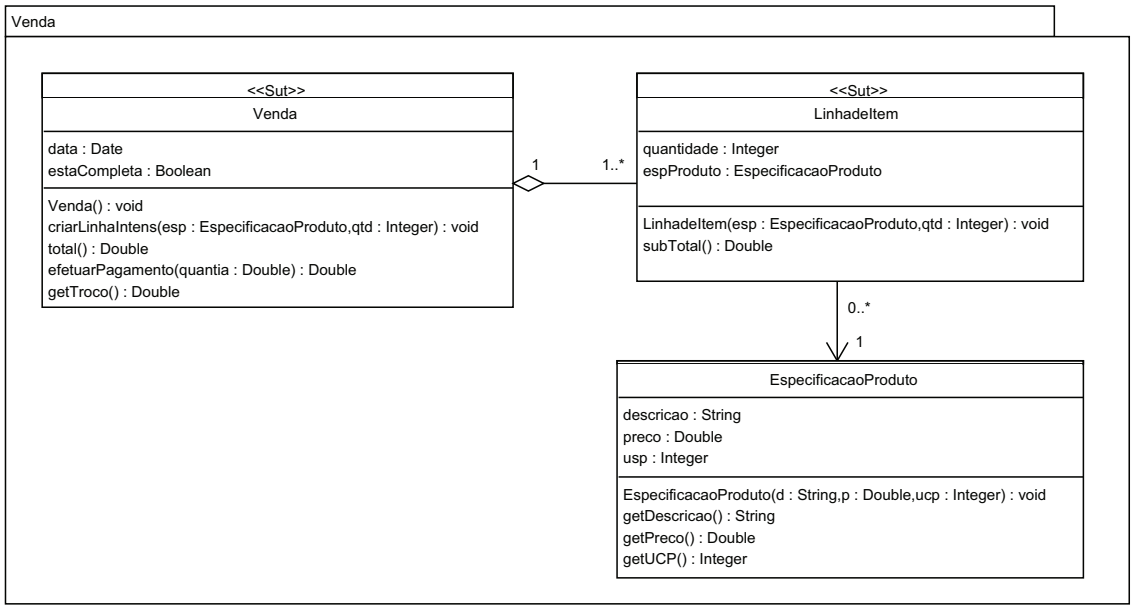

Figura 6 – Modelagem UML de um sistema de vendas [37]

Apesar de ser voltada à área de teste, a U2TP não define nenhum *profile* para segurança, foco deste trabalho. Como alternativa a esta limitação, pesquisadores da área preferem especificar

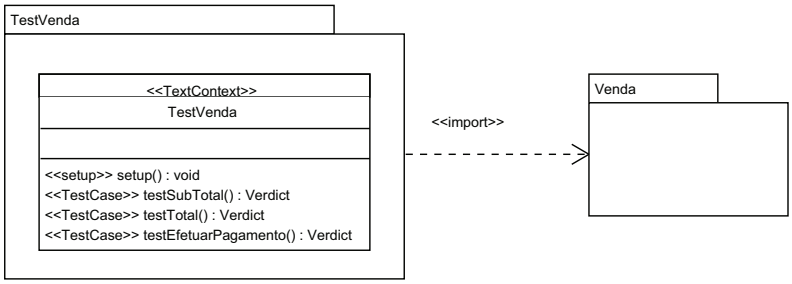

Figura 7 – Modelo U2TP para avaliar o sistema [37]

requisitos de segurança através de restrições OCL ou utilizar-se da flexibilidade de expansão provida pela linguagem, definindo estereótipos próprios. É o que acontece, por exemplo, nos trabalhos UMLSec [38] e SecureUML [39], explicados no Capítulo 5.

3.4 Considerações Finais

Em função da quantidade de funcionalidades providas pelos sistemas, e do curto período de tempo dedicado à fase de testes dentro do ciclo de desenvolvimento, o comportamento dos softwares tem sido analisado de forma cada vez mais deficiente. Normalmente, apenas um pequeno conjunto de testes é aplicado ao sistema, em sua maioria testes funcionais. Desta forma, muitos outros aspectos deixam de ser avaliados, como por exemplo, segurança.

Para auxiliar neste processo, a técnica de teste baseado em modelos vem tornando-se popular. Com esta, é possível gerar casos de teste a partir da extração de informações específicas do modelo, de acordo com o aspecto que deseja-se avaliar. É importante salientar que existe outra técnica de teste utilizando modelos, conhecida como *model-checking*. Diferente da técnica de teste baseado em modelos, cujo objetivo é automatizar o processo de geração de casos de teste, a técnica de *model-checking* visa validar o modelo. Para tanto, algumas regras são aplicadas ao modelo, que são posteriormente verificadas através de um analisador formal. A maior vantagem desta abordagem é a possibilidade de se detectar algumas inconsistências ainda na fase de projeto do software, quando a correção dos problemas é mais rápida e barata do que se realizada após a etapa de implementação. Entretanto, apesar das vantagens relacionadas ao uso de *model-checking*, a técnica utilizada neste trabalho é a de teste baseado em modelos.

Um dos modelos mais utilizados para auxiliar no processo de teste baseado em modelos é a linguagem UML, e isto se deve a diversos motivos. Ela é uma notação fácil de utilizar, simples e grande parte dos engenheiros de *software* possui algum tipo de treinamento a respeito dela. Além disso, UML é flexível, sendo possível estendê-la para modelar aspectos específicos, o que contribuiu para sua popularização e posição de modelagem padrão da indústria [40].

Por estes motivos, apesar do estudo de outros modelos formais e do reconhecimento de que estes também podem ser aplicados à área de teste, optou-se por utilizar a linguagem UML

como base para este trabalho. Considerando que o objetivo deste é automatizar o processo de geração de casos de teste de segurança a partir de modelos formais, é preciso primeiramente inserir informações de segurança no modelo, o que será feito através do uso de estereótipos. Na seqüência, é preciso interpretar o modelo e extrair as informações relevantes à geração dos casos de teste. Isto é feito através da análise do arquivo *XML Metadata Interchange* (XMI), um documento que descreve tanto a organização dos diagramas do sistema, como as informações inseridas através dos estereótipos.

4 Taxonomias de Segurança

Conforme mencionado no Capítulo 2, realizar testes de segurança em uma aplicação ou protocolo implica em analisar o *software*, fazer o levantamento das possíveis vulnerabilidades que podem afetá-lo e executar os testes. Entretanto, devido à variedade de ataques existentes, este processo torna-se demorado, extrapolando o curto prazo que normalmente é dedicado à fase de testes. Além do tempo, outro fator que dificulta a execução de testes de segurança é a falta de experiência dos testadores, que precisam conhecer a maioria dos ataques (ou pelo menos o comportamento padrão destes), bem como estruturas e protocolos que podem ser explorados.

Considerando que o conhecimento sobre ataques e vulnerabilidades é essencial para qualquer estudo relacionado a teste de segurança, este capítulo tem como objetivo apresentar algumas taxonomias de segurança propostas por pesquisadores da área. Elaborar uma taxonomia significa dividir um conjunto de objetos em classes, onde os objetos pertencentes a uma mesma classe possuem características em comum se analisado determinado critério [41–43]. Desta forma, o estudo de taxonomias de segurança é importante pois permite analisar e compreender o comportamento dos ataques, identificando similaridades entre eles. Além disso, ele auxilia na escolha de um modelo que permita representar adequadamente os padrões dos ataques, permitindo que os requisitos de segurança sejam considerados desde a fase de projeto do *software*.

Sendo assim, durante este capítulo serão apresentadas as características e os critérios de classificação de algumas taxonomias de segurança, extraídas da compilação feita por Undercoffer [44]. Ao final, será feita uma comparação entre elas, visando concluir a que melhor descreve os tipos de falha que comprometem a segurança de um *software* e que servirá como guia para o resto do trabalho.

4.1 Taxonomia proposta pela DARPA

Visando auxiliar o processo de detecção de intrusão em sistemas, a *Defense Advanced Research Projects Agency* (DARPA) [45] desenvolveu uma taxonomia de ataques relativamente simples, composta por apenas quatro classes. Para elaborá-la, o primeiro passo foi selecionar que ataques os sistemas de detecção de intrusão conseguiam identificar (objetivo final do trabalho), e então agrupá-los de acordo com as características em comum. Ao todo foram escolhidos 38 ataques, divididos entre as seguintes categorias:

- Negação de serviço (*Denial of Service*);

- Usuário remoto para local (*Remote to local*);
- Usuário comum para super usuário (*User to root*);
- Ataques de vigilância (*Surveillance/probing*).

Conforme mencionado, a DARPA baseou-se em uma série de ataques conhecidos, divididos de acordo com os sistemas operacionais que afetavam. A Tabela 1 apresenta que ataques são estes e em que categorias eles foram classificados.

Tabela 1 – Ataques utilizados para avaliar a taxonomia DARPA

	Solaris	*SunOS*	*Linux*	*Cisco Router*
Denial of Service	apache2 back mailbomb neptune ping of death process table smurf syslogd udp-storm	apache2 back land mailbomb neptune ping of death process table smurf udp-storm	apache2 back mailbomb neptune ping of death process table smurf teardrop udp-storm	
Remote to Local	dictionary ftp-write guest http-tunel phf xlock xsnoop	dictionary ftp-write guest pfh xlock xsnoop	dictionary ftp-write guest imap named phf sendmail xlock xsnoop	snmp-get
User to Root	at eject ffbconfig fdformat ps	loadmodule	perl xterm	
Surveillance/Probing	ip sweet mscan nmap saint satan	ip sweet mscan nmap saint satan	ip sweet mscan nmap saint satan	

Classificam-se como ataques de *Denial of Service* (DoS) aqueles que, como resultado, tornam indisponível um serviço provido por um *host* ou pela rede. A segunda categoria refere-se aos ataques conhecidos como *Remote to Local* (R2L), ou seja, aqueles cujo objetivo é prover acesso local a um usuário malicioso que está conectado à rede remotamente. A terceira classe

definida pela taxonomia abrange os ataques que garantem privilégios de super usuário a um usuário local da máquina, conhecidos como *User to Root* (U2R). A última categoria apresentada pela taxonomia refere-se aos ataques de vigilância (*surveillance*). Ataques desta classe aproveitam-se das informações coletadas pelos *softwares* de varredura da rede (*scans*) para explorar as vulnerabilidades apontadas por estes.

Como é possível perceber, a taxonomia proposta pela DARPA é relativamente genérica, tendo como critério de classificação o objetivo ou resultado do ataque. Devido a sua superficialidade, ela não consegue representar diversos tipos de ataque, como os que aproveitam-se de problemas existentes em protocolos de comunicação ou em algoritmos ruins para criptografia. Por este motivo, definir um modelo para segurança baseado nesta classificação torna-se inviável, pois o resultado final não abrangerá diversos ataques.

4.2 Taxonomia proposta por Aslam

A segunda taxonomia abordada é de autoria de Taimur Aslam [46, 47]. Assim como a proposta pela DARPA, a construção desta taxonomia também foi baseada nas falhas de segurança que afetavam sistemas operacionais. Entretanto, ao invés de considerar falhas de diversos sistemas operacionais, Aslam analisou apenas as falhas que afetavam o sistema operacional Unix. Após identificá-las, ele buscou informações sobre como cada falha era explorada, que área funcional do sistema que ela afetava, em que versões estavam presentes e as conseqüências atribuídas a cada uma. Deste estudo, concluiu-se que as falhas poderiam ser classificadas em três categorias maiores. São elas:

- Falhas de codificação;
- Falhas operacionais;
- Falhas de ambiente.

É importante ressaltar que esta taxonomia aborda apenas as falhas que afetam diretamente o *software*, desde a sua implementação até a sua instalação. Neste caso, não são previstos os casos de exposição de informações confidenciais por parte dos funcionários, problemas na comunicação, problemas de acesso físico e uso de *softwares* para captura de informações.

A primeira categoria proposta refere-se às falhas de codificação, que ocorrem durante o ciclo de desenvolvimento do *software*. Visando refinar esta categoria, foi feito um estudo aprofundado das falhas que normalmente comprometem o funcionamento do *software*, para que fosse possível identificar que parte específica no código provocava a vulnerabilidade. A segunda classe engloba as vulnerabilidades provenientes de falhas operacionais, resultantes de problemas com a instalação do *software*. O mesmo processo de detalhamento foi usado, ou seja, tentou-se identificar que operações levavam a aquela falha e então utilizar esta informações

para criar subcategorias. A terceira e última categoria refere-se às falhas de ambiente. Falhas deste tipo podem ocorrer devido a limitações no sistema onde o *software* está sendo executado, a problemas no compilador ou no sistema operacional utilizado ou em função de problemas de comunicação existentes entre os módulos do sistema.

A estrutura final da taxonomia pode ser vista na Figura 8.

Figura 8 – Taxonomia proposta por Taimur Aslam

Pelos fatores considerados e pela forma como foram divididos, a taxonomia é abrangente e não-ambígua. Além disso, a estrutura em árvore permite identificar aos poucos as características do problema existente, chegando-se a um nodo folha com maior precisão. Para facilitar este

processo, Aslam ainda define uma árvore de critérios de seleção, que indica aos poucos como as características da falha devem ser analisadas. Concluindo, outra vantagem desta taxonomia é que ela é expansível, ou seja, ela pode ser estendida para abranger outros sistemas operacionais ou mesmo incluir novas categorias de falhas.

4.3 Taxonomia proposta por Bazaz

A terceira taxonomia aqui apresentada foi proposta por Anil Bazaz *et al.* [48], e possui como objetivo identificar recursos do *software* que podem ser afetados por ataques. A partir desta identificação, os pesquisadores elaboraram uma série de restrições e cuidados que os programadores devem ter durante o desenvolvimento do *software*.

Através da análise de *exploits* de segurança, obtidos a partir de livros, listas de *e-mail* e *websites*, foram identificados os recursos que são os alvos mais comuns de ataques de segurança. São eles:

- Memória principal;
- Entrada/saída de dados;
- Recursos de criptografia.

Considerando estes três recursos como os de maior necessidade para um *software*, eles tornaram-se a base da taxonomia, sendo convertidos em classes genéricas. A partir de então as características particulares de cada ataque foram sendo identificadas, refinando a taxonomia e criando subclasses. A árvore final da taxonomia pode ser vista na Figura 9.

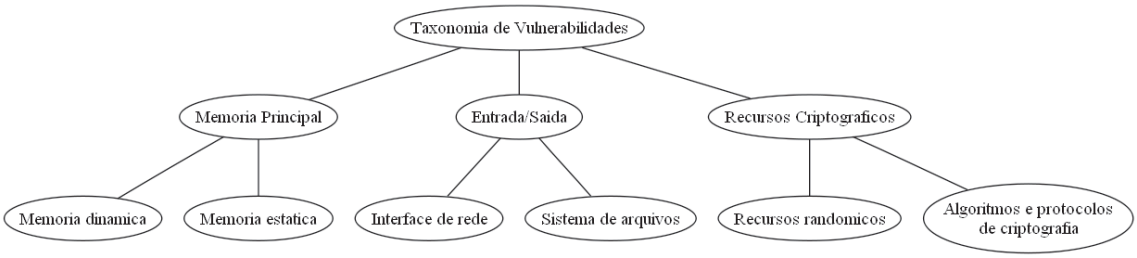

Figura 9 – Taxonomia proposta por Anil Bazaz *et al*.

Como a classificação é baseada nos recursos utilizados, o desenvolvedor pode utilizá-la para identificar os que ele utiliza e, então, prever que ataques podem ocorrer e empregar políticas para evitá-los. A elaboração destas práticas é o objetivo final dos autores da taxonomia, mas até o momento foram apresentadas apenas boas práticas associadas à memória dinâmica.

Enfim, a abordagem difere das outras pois ela não visa identificar a causa ou a conseqüência de um ataque, mas sim o recurso que será utilizado pelo *software*. Portanto, é mais sistemática,

pois permite visualizar previamente os possíveis pontos de vulnerabilidade e desenvolvê-los com cuidado, evitando assim que futuras vulnerabilidades afetem o *software* sob implementação.

4.4 Taxonomia proposta por Weber

A taxonomia proposta por Weber [49] utiliza como critério de classificação os tipos de falhas que podem afetar um *software*. Esta abordagem é útil para engenheiros de *software* e desenvolvedores, que devem prever e evitar problemas de projeto e implementação. A Tabela 2 demonstra como ela está organizada.

Tabela 2 – Taxonomia de Weber

Intencionais	Maliciosas	Trapdoor	
		Logic/Time Bomb	
	Não-Maliciosas	Canal	Armazenamento
			Tempo
		Caminhos Inconsistentes de Acesso	
Acidentais	Erros de Validação	Erros de Endereçamento	
		Erros de Validação de Parâmetros	
		Ordem Incorreta de Verificação	
		Identificação/Autenticação Inadequada	
	Erros de Abstração	Reuso de Objetos	
		Exposição da Representação Interna	
	Falhas Assíncronas	Falhas de Concorrência	
		Inconsistência de Nomenclatura	
	Mau uso/defeito de Subcomponentes	Estouro de Recursos	
		Falhas de Responsabilidade	
	Erros de Funcionalidade	Defeito na Manipulação de Erros	
		Outros Erros de Funcionalidade	

A primeira divisão utiliza, como critério de classificação da taxonomia, a forma como a falha foi inserida no *software*. As falhas podem ser inseridas de forma **intencional**, quando deseja-se acrescentar alguma funcionalidade ao *software*, ou de forma **acidental**, que ocorre durante o desenvolvimento normal do *software*. Segundo a taxonomia, as falhas intencionais podem ser de duas formas: as inseridas de forma maliciosa, como as que abrem portas indevidas no sistema, e as inseridas de forma não-maliciosa, como as que alteram a estrutura de diretórios, tornando inválidos os caminhos de acesso aos arquivos.

Em relação às falhas acidentais, a taxonomia identifica diversas categorias de problemas de implementação do *software*. A primeira refere-se aos erros de validação dos dados, ou seja,

falhas que ocorrem em função dos dados de entrada não terem sido validados corretamente. Alguns ataques que utilizam-se das vulnerabilidades classificadas nesta categoria são: *buffer overflow* e acesso a posições inexistentes (erros de endereçamento), atribuição de valores inexistentes a uma variável (erros de validação de parâmetros) e execução indevida de alguma operação (identificação/autenticação inadequada).

A segunda classe corresponde aos erros de abstração, também conhecidos por erros de domínio. Ocorrem em sistemas onde um módulo, que deveria visualizar apenas um modelo abstrato dos outros módulos, consegue manipular ou acessar informações internas destes. Vulnerabilidades deste tipo permitem, por exemplo, que o atacante recupere senhas ou documentos (para o caso de reuso de objetos) ou consiga privilégios através da modificação de algum bloco de controle crítico (exposição da representação interna).

A terceira categoria refere-se às falhas assíncronas, ou seja, que permitem que o atacante utilize-se de problemas existentes no controle dos processos para danificar o sistema. Encontram-se nesta categoria as falhas de concorrência, como *deadlocks* e ataques de *race condition* [50], e as falhas de inconsistência de nomenclatura, como usar diferentes nomes para referir-se ao mesmo objeto.

A quarta classe de falhas corresponde ao mau uso ou defeito dos subcomponentes. Acontece quando o sistema não gerencia de forma adequada os recursos, como por exemplo, quando a memória alocada pelo sistema não é liberada corretamente. Este tipo de falha é classificado na taxonomia como estouro de recursos, e pode ser explorado, por exemplo, com um ataque de *flooding* de conexões. A outra subdivisão desta categoria chama-se falhas de responsabilidade. Nesta, devido a inconsistências presentes nas interfaces dos subcomponentes, o sistema torna-se vulnerável. Assim, de acordo com a operação executada, é possível, por exemplo, elevar os privilégios de um usuário após um ataque de *buffer overflow*.

A quinta e última divisão corresponde aos erros de funcionalidade, ou seja, que estão diretamente relacionados à implementação das funcionalidades do sistema. Divide-se em duas categorias: defeito na manipulação de erros, que ocorre quando as exceções do sistema são manipuladas de forma incorreta, e em outros erros de funcionalidade, onde podem ser classificados os outros problemas de implementação, tais como o armazenamento inseguro de informações.

Sendo assim, como é possível perceber, esta taxonomia é mais completa que as demais, pois permite classificar qualquer tipo de falha ocorrida durante o processo de desenvolvimento do *software*. Apesar de ser ambígua, já que permite classificar o mesmo tipo de vulnerabilidade em diferentes categorias, a taxonomia de Weber é a mais atual e mais citada pelos autores, pois orienta para uma implementação segura e tem a capacidade de representar falhas atuais, como por exemplo *Cross-Site Scripting* (XSS) [49].

4.5 Considerações Finais

Quando um analista define que casos de teste serão aplicados em um *software*, ele deve seguir um plano que permita avaliar a maior parte do sistema, na tentativa de identificar o máximo de defeitos possível. Entretanto, diferentemente dos outros tipos de teste que possuem técnicas consolidadas para critérios de cobertura, não existem métricas para avaliar a eficácia dos testes de segurança. Desta forma, quando se deseja determinar cientificamente que vulnerabilidades ou ataques podem afetar um sistema, utilizam-se taxonomias de segurança [49].

A existência destas taxonomias permite ter um panorama geral dos aspectos de segurança que se deseja analisar. Normalmente, o critério de classificação utilizado é o de vulnerabilidades conhecidas, mas também existem trabalhos que baseiam-se em tipos de ataques ou recursos afetados do sistema. Após definir a taxonomia que será utilizada como guia, resta ao analista avaliar que casos podem afetar o sistema e definir que testes serão realizados para verificar a existência destes.

Diversas taxonomias de segurança foram estudadas durante este trabalho [41–48]. Entretanto, grande parte destas foi elaborada há bastante tempo, estando desatualizada. Sendo assim, a taxonomia escolhida para ser utilizada como base foi a proposta por Weber [49]. Como critério de classificação, ela utiliza os tipos de falhas que podem afetar um *software*, o que a torna simples e abrangente, já que consegue representar a maior parte das vulnerabilidades.

Para complementar o estudo sobre falhas e vulnerabilidades, avaliou-se, além da taxonomia, a base de dados desenvolvida pelo projeto *The Open Web Application Security Project* (OWASP) [7], que mantém uma lista com as 10 vulnerabilidades mais críticas para aplicações Web. Associada à taxonomia, esta lista foi utilizada para destacar os problemas mais comuns de segurança, permitindo, inclusive, concluir sobre a viabilidade de representar estes através de estereótipos UML. Sendo assim, devido a sua simplicidade e capacidade de representação, estas foram as estruturas escolhidas para inserir as informações relevantes aos casos de teste no modelo do sistema a ser testado.

5 Trabalhos Relacionados

A dinâmica do mercado de Tecnologia da Informação exige a liberação constante de novas versões de *software*, estipulando prazos cada vez menores para isso. Esta pressa reduz o tempo dedicado à execução de testes no *software*, comprometendo não apenas o funcionamento deste, mas também outros aspectos importantes, como segurança. Neste contexto foi proposta, nos últimos anos, uma nova técnica de teste, denominada teste baseado em modelos [4, 5].

O objetivo desta técnica é gerar casos de teste automatizados a partir do modelo do sistema, de acordo com o aspecto que se deseja testar. O processo para utilizá-la pode ser dividido em duas etapas: a primeira consiste em especificar no modelo as informações relevantes sobre o aspecto testado; a segunda, em desenvolver um algoritmo de geração de casos de teste a partir destas informações.

Considerando que o contexto deste trabalho é automatizar o processo de geração de casos de teste voltados à segurança, foram estudados alguns trabalhos já existentes na área a fim de verificar a eficiência e as limitações destes. Sendo assim, este capítulo está dividido em duas seções: a Seção 5.1 refere-se aos trabalhos focados na área de modelagem de aspectos de segurança, enquanto que a Seção 5.2 apresenta algumas ferramentas e algoritmos para geração de casos de teste.

5.1 Técnicas de Modelagem de Aspectos de Segurança

Conforme mencionado anteriormente, o primeiro passo para viabilizar a técnica de teste baseado em modelos é inserir, no modelo, as informações relevantes sobre o aspecto que se deseja avaliar. Devido a sua simplicidade e capacidade de representação, optou-se por utilizar como base deste trabalho o modelo UML.

Embora a inserção de restrições OCL nos diagramas da UML permita descrever, de forma precisa, diferentes características do sistema, ela é uma linguagem de difícil compreensão. Este fato fez com que os engenheiros de *software* estendessem a UML utilizando uma terceira representação, baseada em estereótipos e *tags*.

Na área de segurança, diversos trabalhos propõem *profiles* e *frameworks* para especificar e modelar características deste aspecto. Para tanto, diversas notações foram utilizadas, como é possível ver nas seções que seguem.

5.1.1 UMLSec

Um dos principais trabalhos relacionados à área de modelagem de segurança é o *profile* UMLSec [38]. Ele foi criado para modelar e avaliar aspectos de segurança, visando garantir alguns princípios básicos da área, tais como descrever operações confiáveis e marcar *links* seguros. Através deste *profile* é possível encapsular conhecimentos sobre engenharia de segurança e disponibilizá-las aos desenvolvedores, que podem não estar familiarizados com práticas seguras.

O *profile* UMLSec utiliza três estruturas para expandir a capacidade de representação da linguagem UML: estereótipos, *tags* (valores marcados) e restrições. A Tabela 3 apresenta os estereótipos propostos pelo trabalho, bem como as *tags* e restrições equivalentes.

Tabela 3 – Estereótipos, *tags* e restrições definidas pelo *profile* UMLSec

Estereótipo	*Classe Base*	*Tags*	*Restrições*	*Descrição*
Internet	link			Conexão à Internet
encrypted	link			Conexão criptografada
LAN	link			Conexão LAN
secure links	subsystem		assinalado em links	Reforça a segurança dos links de comunicação
secrecy	dependency			Sigilo de informações
secure dependency	subsystem		«call» e «send»	Interação estrutural sobre segurança de dados
critical	object	{secret}		Objetos críticos
no down-flow	subsystem			Fluxo de informações
data security	subsystem			Requisitos básicos de segurança de dados
fair exchange	package	{start/stop}	após o estado *start*, eventualmente alcanço estado stop	Reforçar operação de troca justa

A explicação de cada estereótipo pode ser visto a seguir.

- Os estereótipos «*Internet*», «*encrypted*» e «*LAN*» são utilizados nos *links* do diagrama de implantação. Denotam o tipo de *link* de comunicação;

- O estereótipo «*secure links*» é usado para marcar susbsistemas onde é preciso garantir que os requisitos de segurança da comunicação estão sendo atendidos pela camada física;
- O emprego do estereótipo «*secrecy*» assinala objetos ou diagramas de componentes em que as dependências devem prover o sigilo dos dados enviados como argumento ou retorno de operações;
- O estereótipo «*secure dependency*» pode ser inserido em subsistemas contendo diagramas de objetos ou estruturais. Visa garantir que as dependências «*call*» ou «*send*» respeitem os requisitos de segurança dos dados enviados através deles;
- O estereótipo «*critical*» marca objetos cujas instâncias são consideradas críticas para o sistema;
- O estereótipo «*no down-flow*» é utilizado para assinalar subsistemas e é usado para melhorar a segurança do fluxo de informações do sistema;
- O estereótipo «*data security*» marca subsistemas cujo comportamento deve respeitar os requisitos de segurança de dados destacados pelo estereótipo «*critical*» e *tags* associadas;
- O estereótipo «*fair exchange*» é utilizado para formalizar os requisitos de troca justa. Com as *tags* «*start*» e «*stop*» é possível definir um comportamento em que, se uma função alcança o estado S1 marcado por «*start*», então ele chega ao estado S2 marcado por «*stop*».

A Figura 10 apresenta um exemplo de uso do estereótipo «*fair exchange*», do *profile* UMLSec. No exemplo é descrita a atividade de compra de um produto. Considerando que este estereótipo define que, uma vez que o estado assinalado pela *tag start* seja alcançado, um dos estados marcados pela *tag stop* também será executado, o exemplo demonstra que a partir do estado **Pagar**, ou o pedido será entregue corretamente (estado **Entregar**) ou o cliente reclamará o produto que não chegou no prazo (estado **Reclamar**).

Como é possível perceber, o *profile* UMLSec não visa modelar partes do sistema que podem tornar-se vulneráveis, mas especificar uma forma de garantir um comportamento seguro do sistema. Considerando ainda que as partes comportamentais do *profile* podem ser descritas através de formalismos (OCL), ainda é possível fazer uma verificação no modelo, buscando por requisitos de segurança que não tenham sido descritos corretamente. Sendo assim, o UMLSec é um *profile* de segurança que alia a facilidade de descrição provida pela linguagem UML, com a precisão das restrições OCL, o que permite avaliar formalmente os requisitos de segurança definidos durante o projeto do sistema.

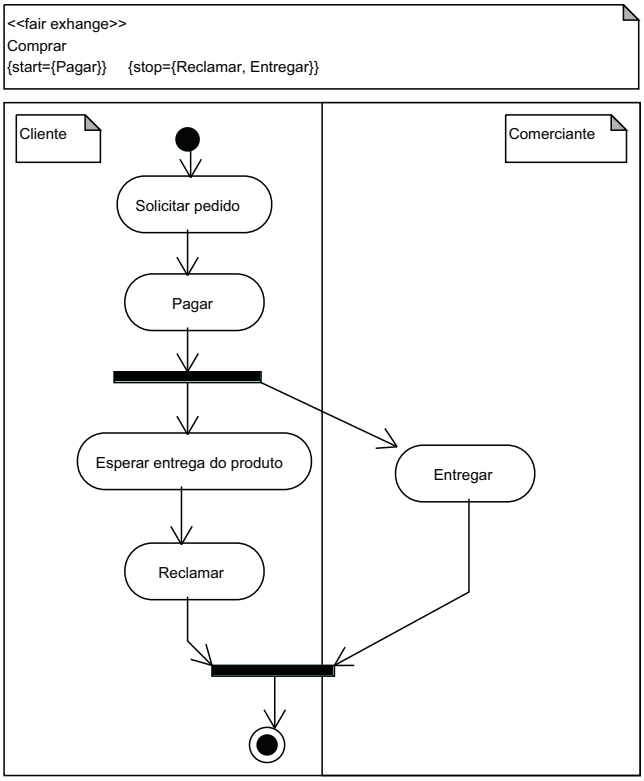

Figura 10 – Exemplo de uso do estereótipo «*fair exchange*»

5.1.2 SecureUML

SecureUML [39] propõe uma extensão da linguagem UML para especificar políticas de acesso baseadas em papéis, conhecidas como *Role-based Access Control* (RBAC). O objetivo desta é descrever estruturas de controle de acesso ao sistema, ainda no modelo UML, agregando a simplicidade da notação gráfica utilizada pelo modelo RBAC ao formalismo associado as restrições lógicas especificadas com OCL.

Como mencionado anteriormente, a linguagem SecureUML é baseada em uma extensão do modelo RBAC. Mesmo que este modelo de controle de acesso seja bem estabelecido, com inúmeras vantagens reconhecidas e suportado por diversas plataformas, ele possui algumas limitações. Um exemplo é a deficiência em expressar condições de acesso referentes aos estados do sistema, tais como o acesso à recursos protegidos, hora ou data. A fim de solucionar estes problemas, introduziu-se, no SecureUML, o conceito de restrições de autorização, ou seja, pré-condições que garantem acesso a algumas operações. A definição destas restrições é feita através da linguagem OCL. Com esta estrutura, é possível especificar as políticas de acesso simples através de permissões baseadas em papel, através do modelo RBAC, enquanto que as mais complexas podem ser descritas adicionando-se restrições de autorização. A Figura 11 apresenta o metamodelo que define a sintaxe abstrata do SecureUML.

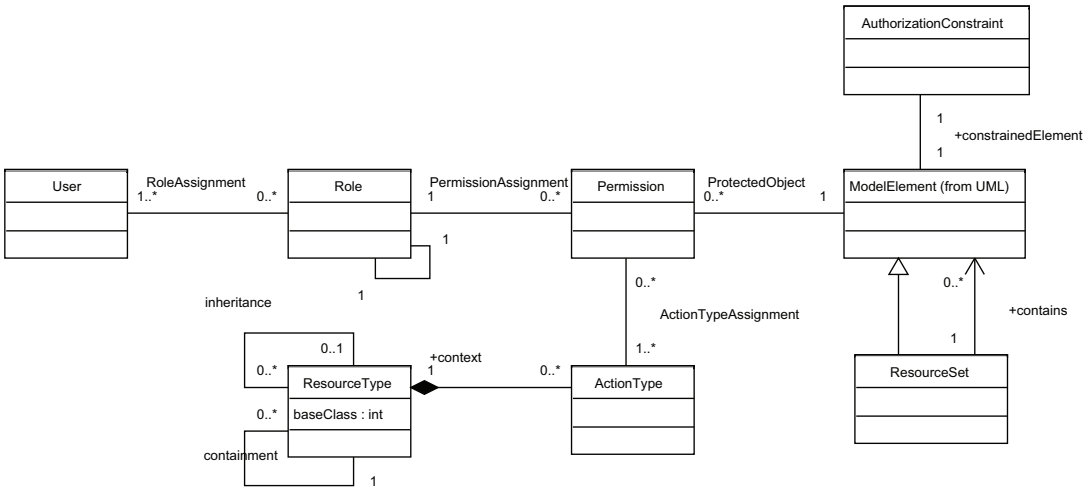

Figura 11 – Metamodelo do SecureUML [39]

A fim de provar que é possível gerar estruturas de segurança para controle de acesso usando o modelo proposto, foi implementado um gerador de componentes para a arquitetura EJB [51] utilizando os estereótipos propostos pelo SecureUML. A Figura 12 demonstra um exemplo de aplicação do SecureUML em um sistema com dois componentes: um Calendário e um Registro. Um calendário pode conter diversos registros, todos contendo uma data de início, término, localização e pertencer a um dono. As demais estruturas são usadas para expressar informações sobre controle de acesso através do SecureUML.

Após a análise do modelo, foram obtidas aplicações EJB com controles de acesso já configurados, incluindo definição de papéis, permissões e restrições de autorização. Cada elemento modelado é convertido em políticas de segurança para um componente EJB. Alguns exemplos de código gerado a partir destas definições podem ser vistos na Figura 13.

A partir do que foi descrito, é possível concluir que o uso do SecureUML permite especificar políticas de controle de acesso de forma precisa, ainda na fase de projeto de *software*, através de diagramas UML. Por ser genérico, pode ser utilizado com qualquer arquitetura de segurança que suporte o acesso baseado em papéis, entretanto não consegue especificar outros aspectos que não estejam relacionados ao controle de acesso. Sendo assim, esta proposta pode ser utilizada, por exemplo, quando é preciso configurar aplicações responsáveis pelo acesso ao *software*, mas não para descrever outros aspectos de segurança, tais como estouro de *buffer* ou uso de protocolos inseguros.

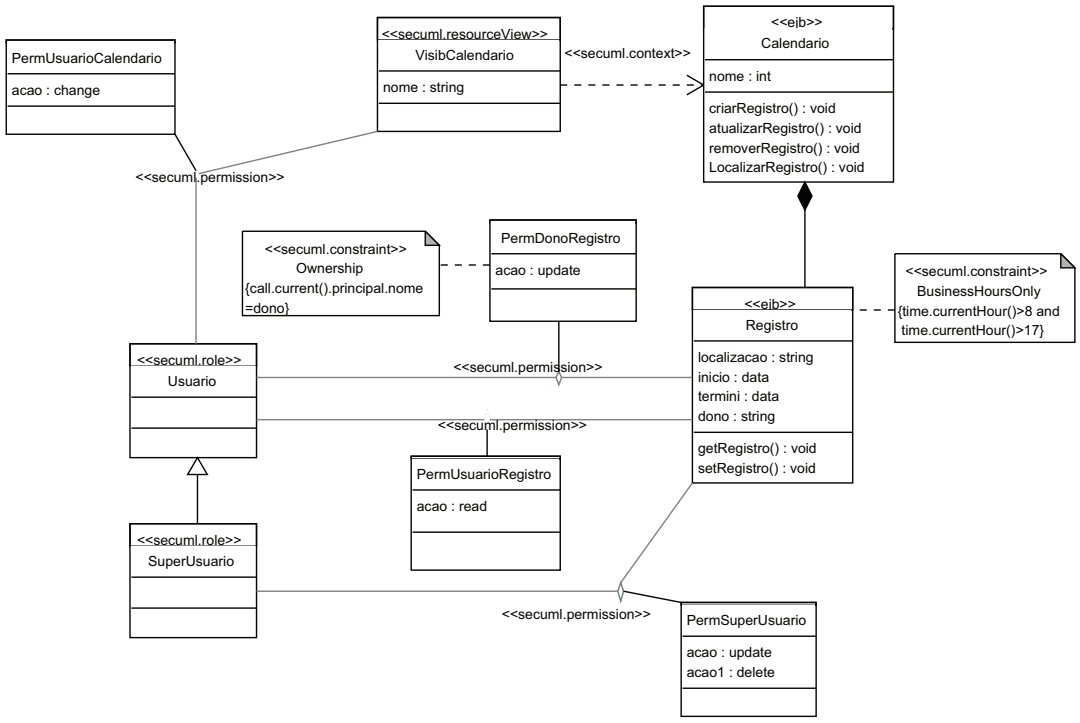

Figura 12 – Modelo de um sistema para agendamento de atividades [39]

```
<security-role>
    <role-name> User </role-name>
</security-role>

<security-role-assignment>
    <role-name> User </role-name>
    <principal-name> Smith <principal-name>
<security-role-assignment>

Descrição OCL:
    context Registro::getRegistro():EntryInfo
    pre: time.currentHour() > 8 and time.currentHour() < 17
```

Figura 13 – Exemplos de códigos gerados a partir das definições com o SecureUML [39]

5.2 Ferramentas para Automatização de Teste

Como parte da técnica de teste baseado em modelos, além da especificação de aspectos de segurança, também é preciso automatizar o processo de teste de *software*. Este pode ser dividido em duas etapas: a de geração dos casos de teste a partir do modelo e a de execução dos testes, seja através de *drivers* de teste ou *scripts* próprios.

Existem diversos trabalhos e ferramentas cujo objetivo é auxiliar no processo de teste. A descrição de alguns pode ser vista nas seções seguintes.

5.2.1 JMeter

O Apache JMeter [52] é uma ferramenta projetada para executar testes funcionais e de desempenho de forma automatizada. Originalmente foi desenvolvida para avaliar sistemas Web, mas devido a sua eficiência e simplicidade foi expandida para suportar outros tipos de aplicações, tais como o teste de servidores FTP, arquivos e *scripts* em Perl. Dentre suas funcionalidades, está a capacidade de simular grandes quantidades de tráfego para servidores e redes, a fim de avaliar o desempenho destes. Ele também disponibiliza diversos tipos de gráficos como saída, o que facilita a análise dos resultados obtidos com os testes.

Diferentemente de outras ferramentas voltadas a área de teste, o Apache JMeter exige poucas configurações para executar um caso de teste. Por sua arquitetura ser baseada em *plug-ins*, ele possui um núcleo básico que contém algumas funções já implementadas, tais como o envio de pacotes HTTP e FTP, restando ao usuário apenas definir alguns parâmetros. Além disso, a ferramenta é extensível, sendo possível adicionar novas funcionalidades a ela desenvolvendo *plug-ins* específicos para as necessidades dos usuários.

A Figura 14 apresenta como seria a definição dos parâmetros para executar um teste de requisições FTP.

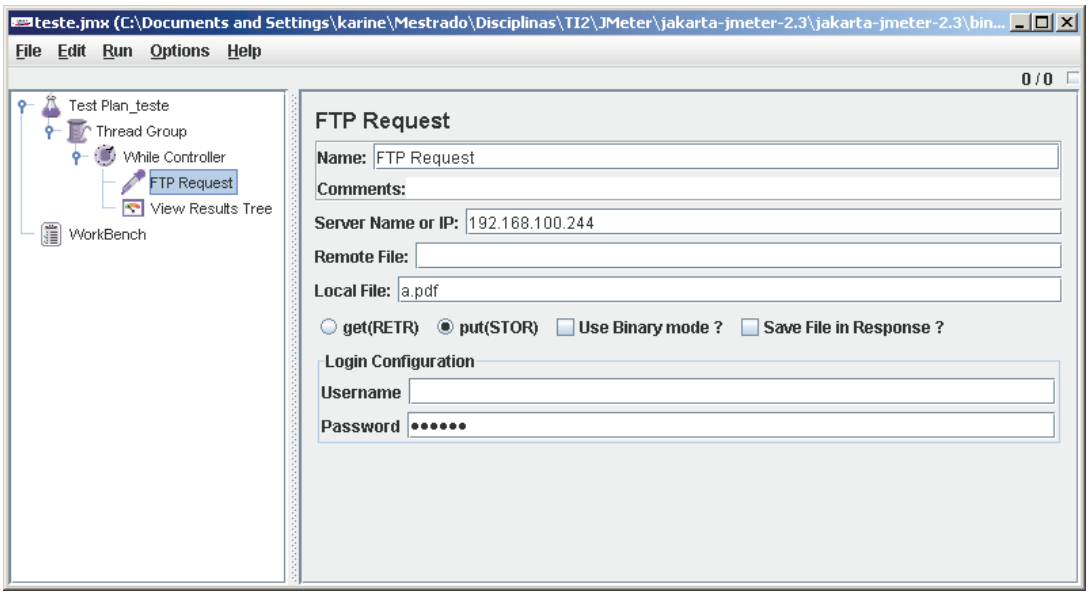

Figura 14 – Tela do Apache JMeter para configuração de parâmetros

Analisando-se a estrutura do teste, é possível perceber que o caso a ser executado aparece na árvore a esquerda da tela, abaixo do nodo *Thread Group*. O Apache JMeter possui uma série de elementos que permitem estruturar os casos de teste, e *Thread Group* é um deles. É este elemento que define a seqüência na qual os testes serão executados. A execução de uma *Thread Group* dispara todos os testes associados a ela, independente da existência de outras *threads*. Outros elementos amplamente utilizados são:

- *Samplers*, que definem o tipo de teste que vai ser realizado (ex.: HTTP *Request* e FTP *Request*);
- *Logical Controllers*, que permitem configurar a lógica utilizada no envio de requisições (ex.: *While Controller*);
- *Listeners*, que permitem visualizar os resultados obtidos com a execução dos testes (ex.: *View Results Tree*);
- *Assertions*, que permitem verificar se os valores retornados pela aplicação sob teste são realmente válidos.

A Figura 15 utiliza alguns destes elementos para demonstrar os resultados obtidos com o envio de uma mensagem FTP *Request*.

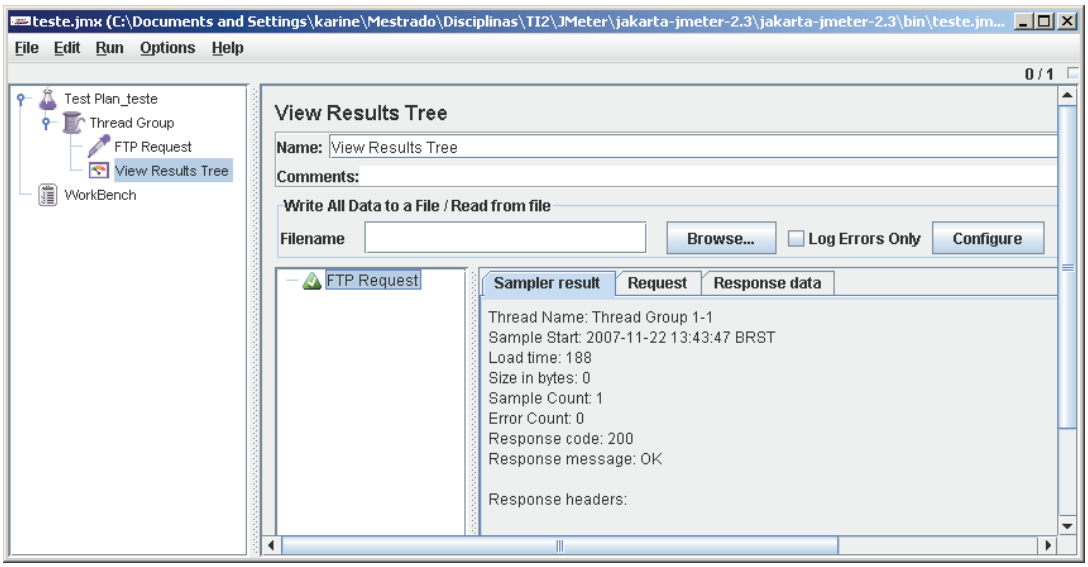

Figura 15 – Resultados observados com o Apache JMeter

Apesar das inúmeras funcionalidades providas pela ferramenta Apache JMeter, ela não possui suporte a teste de segurança. Considerando que um de seus objetivos é realizar testes de desempenho, uma das poucas aplicações na área de segurança seria o envio excessivo de requisições, simulando um ataque de *flooding*. Neste, um usuário malicioso envia um número maior de mensagens do que a máquina destino pode processar, resultando no esgotamento de recursos de memória, processador ou até mesmo em uma negação de serviço. Outros ataques relacionados à conexão e envio de dados também poderiam ser executados, desde que fossem incluídas algumas funcionalidades extras na ferramenta. Entram nesta categoria os ataques de adivinhação de senhas, inconsistência no formato dos dados, bem como os de tentativa de roubo ou dano à conexão. Desta forma, apesar de não ser voltado ao teste de segurança, a ferramenta Apache JMeter pode ser utilizada para executar alguns casos, desde que incluídas as devidas funcionalidades.

5.2.2 STAGE

Diferentemente da ferramenta Apache JMeter, descrita na Seção 5.2.1, o *STAte-based Test GEnerator*(STAGE) [23] é um gerador de casos de teste e *scripts* funcionais a partir do modelo da aplicação a ser testada, descrita através de autômatos finitos. Ela permite utilizar um dos seguintes modelos de autômatos: máquina de estados finitos (*Finite State Machine*, FSM), máquina de estados finitos virtual (*Variable Finite State Machine*, VFSM) ou cadeias de Markov. Com base neste modelo, a ferramenta identifica os caminhos que podem ser percorridos pelo *software* e determina que combinações devem ser testadas. A Figura 16 apresenta um exemplo de modelo de aplicação através de autômatos finitos.

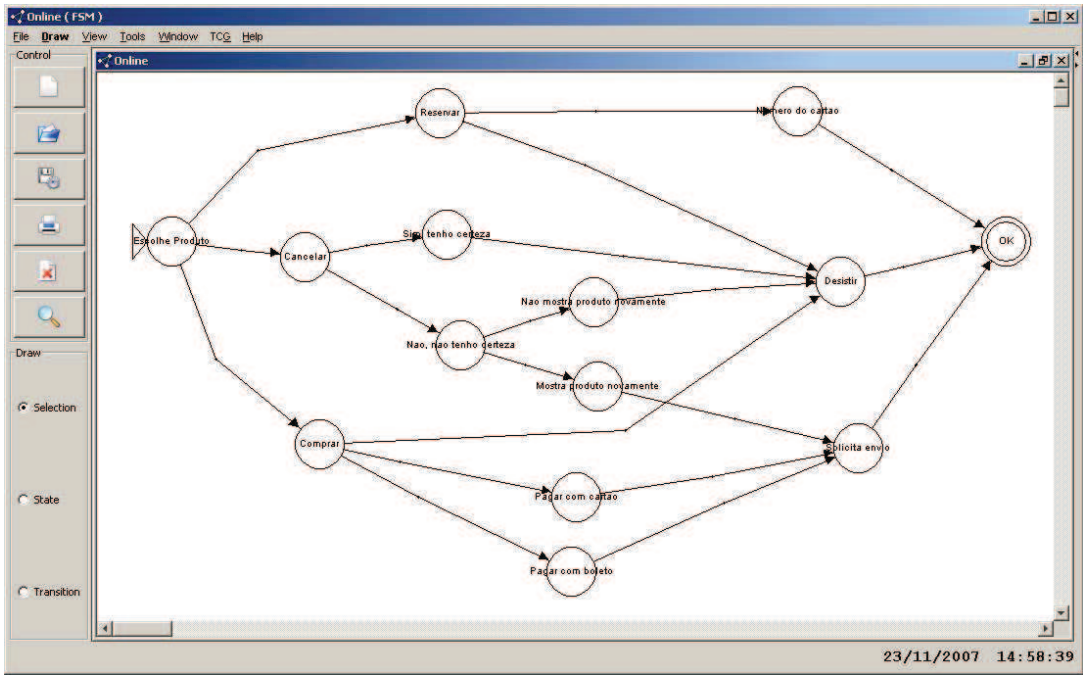

Figura 16 – Modelagem de aplicação utilizando o *software* STAGE

Como mencionado anteriormente, a ferramenta identifica, a partir do modelo da aplicação, quais são os caminhos que podem ser percorridos durante a execução da ferramenta. Cada caminho representa um caso de teste, e a geração destes é feita por um módulo denominado *Test Case Generator*, ou simplesmente TCG. Ele verifica, desde o estado inicial, todas as formas de se chegar ao estado final, garantindo assim que todos os caminhos do *software* tenham sido cobertos. A forma como os casos de teste são descritos pode ser vista na Figura 17.

Após a descrição dos casos de teste, a ferramenta ainda permite a geração de *scripts* representando o comportamento destes, possibilitando a execução automatizada dos testes na aplicação. O STAGE gera código no formato de entrada de duas ferramentas de teste, o Rational XDE Tester [53] e o HttpUnit [54], responsáveis por aplicar testes em aplicações Java e Web, respectivamente. Uma funcionalidade interessante do STAGE é que ele também permite

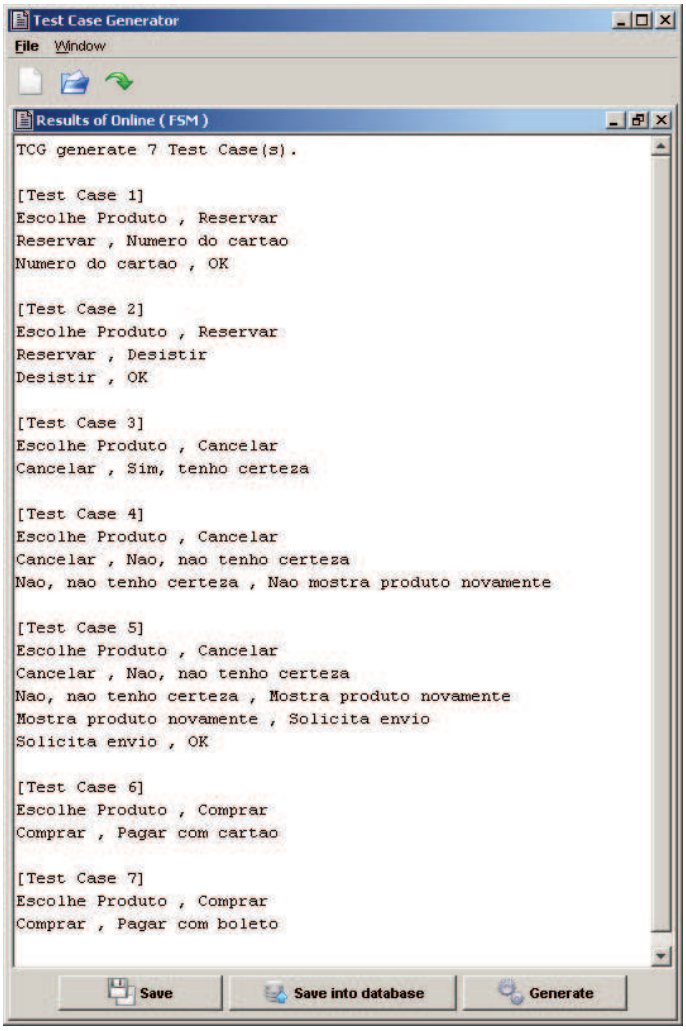

Figura 17 – Casos de teste gerados pelo STAGE a partir do modelo da aplicação

definir testes para as interfaces das aplicações. Desta forma, é possível fazer um mapeamento dos componentes que constituem a interface e integrá-los ao comportamento descrito pelos autômatos. Sendo assim, o usuário descreve os componentes de cada interface a ser testada, como botões, campos de texto, *checkbox* e *links*, e associa a estes os eventos mapeados pelo modelo comportamental. Com esta funcionalidade é possível, por exemplo, utilizar os testes gerados para verificar inconsistências nos tamanhos dos *buffers*, a presença de *links* mal-redirecionados ou de botões que disparam outras funções diferentes da esperada.

Como o objetivo do STAGE é gerar os possíveis caminhos que podem ser executados pelo *software*, ele não se preocupa em considerar outros aspectos, como segurança. Entretanto, é possível adaptar os testes aplicados na interface e utilizá-los para avaliar alguns casos desta área. Um exemplo é o mapeamento dos componentes para entradas de dados, que se mal implementados podem comprometer a segurança do aplicativo devido a um problema de estouro de *buffer* ou inconsistência de dados. Outro exemplo são os casos de falhas de autenticação

(*By Passing*), onde a partir de um estado chega-se a outro sem passar por um intermediário, responsável por validar a identidade do usuário. Portanto, apesar de não possuir nenhuma estrutura especialmente dedicada a segurança, é possível utilizar alguns dos casos de teste de interface gerados, para analisar este aspecto. O teste de alguns casos, como o roubo de conexão entre duas máquinas, por exemplo, é inviável, mas com algumas modificações é possível ao menos ampliar o número de situações que podem ser transformadas em casos de teste de segurança mapeadas pelo STAGE.

5.2.3 HP QAInspect

A HP QAInspect [55] é uma ferramenta de uso simples e que oferece uma solução totalmente automatizada para a realização de testes de segurança em aplicações Web. Ela foi desenvolvida com o objetivo de auxiliar os profissionais de teste de *software* a avaliar aspectos de segurança, tenham eles conhecimento desta área ou não. A ferramenta é integrada ao HP Quality Center, um ambiente de gerenciamento que permite integrar diferentes tipos de testes, como funcionais e de desempenho. Para avaliar uma aplicação utilizando o *software* QAInspect, é preciso ter uma licença de uso deste e do Quality Center. Entretanto, é disponibilizada uma licença de avaliação cuja duração é de 15 dias e que permite realizar testes sobre uma aplicação única.

Os testes realizados pelo QAInspect podem ter caráter unitário, sendo voltados a um cenário ou página, ou integrados, e realizados sobre a aplicação como um conjunto de páginas. Também podem ser do tipo *Crawl* que gera e estuda a estrutura tanto das informações como da plataforma da aplicação, e/ou do tipo *Audit* que verifica a existência de uma vulnerabilidade e determina uma prioridade para a mesma.

Antes de executar um teste, é preciso configurar uma série de parâmetros, tais como o endereço e tipo de aplicação Web, e a política de segurança utilizada. O QAInspect possui uma série de testes já definidos, e a escolha dos casos que serão testados acontece de acordo com a política de segurança selecionada. Entre as políticas disponíveis encontram-se a Sarbanes-Oxley [56], a ISO 17799 [57], a PCI *Data Security Standard* [58] e a OWASP *Top Ten* [7], um projeto que mantém uma lista com as 10 vulnerabilidades mais críticas para aplicações Web.

Após a execução dos testes, é possível analisar todas as vulnerabilidades encontradas pelo QAInspect na aplicação testada. Uma vantagem deste *software* é que os detalhes dos testes são exibidos de forma detalhada, apresentando dados sobre o nome da página avaliada, o tipo de irregularidade encontrado e o quão crítico é o problema. Ela também apresenta uma descrição da vulnerabilidade encontrada, além de dicas sobre como executar o teste manualmente e como evitar que a vulnerabilidade aconteça.

A Figura 18 apresenta a lista de vulnerabilidades encontradas no *site* padrão do QAInspect (http://zero.webappsecurity.com), quando aplicada a política de segurança OWASP Top Ten.

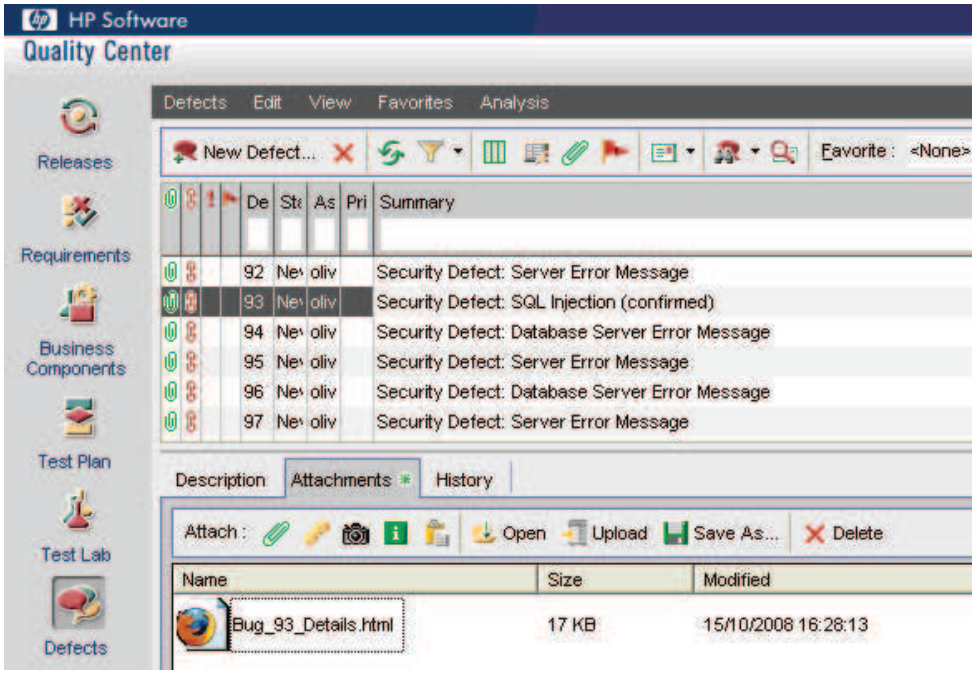

Figura 18 – Lista de vulnerabilidades encontradas com o HP QAInspect

Como exemplo, vamos analisar a vulnerabilidade de número 93, que refere-se à execução de um ataque de SQL Injection. Este consiste na execução de comandos SQL no banco de dados através de caixas de texto. É um falha de implementação que extrai a informação diretamente das caixas de texto e a preenche na consulta SQL.

Analisando-se os resultados apresentados para o caso de SQL Injection, é possível ver como o QAInspect executou o teste. O primeiro passo consiste em enviar uma consulta SQL no campo de *login* da página login.asp, através de uma mensagem HTTP *Request*. Na seqüência, é preciso analisar a mensagem HTTP *Response* obtida como resposta ao código malicioso enviado para a aplicação Web. As Figuras 19 e 20 apresentam o conteúdo das mensagens HTTP *Request* e HTTP *Response*, respectivamente.

```
1. POST /login1.asp HTTP/1.1
2. Referer: http://zero.webappsecurity.com:80/login.asp
3. Content-Type: application/x-www-form-urlencoded
4. Content-Length: 77
5. User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
6. Accept: */*
7. Pragma: no-cache
8. Host: zero.webappsecurity.com
9. Connection: Keep-Alive
10. Cookie: CustomCookie=WebInspect39122ZX636BCC9CFDF249598418C224C84B1074YAB6A;
11. ASPSESSIONIDAASCDDQB=PPLDBJFBDNGPHJBJBANOEDHC;sessionid=;state=;username=;
12. userid=;status=yes;

13. login=foo'+and++(select+count(*)+from+spitable)+%3d1+or+'1'%3d'0&password=foo
```

Figura 19 – Mensagem HTTP *Request* com SQL *Injection*

```
1. HTTP/1.0 500 Internal Server Error
2. Server: Microsoft-IIS/5.0
3. Date: Wed, 15 Oct 2008 20:28:28 GMT
4. X-Powered-By: ASP.NET
5. Content-Length: 445
6. Content-Type: text/html
7. Cache-Control: private
8. X-Cache: MISS from proxy4.pucrs.br
9. X-Cache: MISS from firewall01.cpph.pucrs.br
10. Via: 1.0 firewall01.cpph.pucrs.br:800 (squid/2.6.STABLE17)
11. Connection: close
12. <font face="Arial" size=2>
13. <p>Microsoft OLE DB Provider for ODBC Drivers</font> <font face="Arial"
14. size=2>error '80040e37'</font>
15. <p>
16. <font face="Arial" size=2>[Microsoft][ODBC Microsoft Access Driver]
17. The Microsoft Jet database engine cannot find the input table or query 'spitable'.
18. Make sure it exists and that its name is spelled correctly.
19. </font>
20. <p>
21. <font face="Arial" size=2>/login1.asp</font><font face="Arial" size=2>,
22. line 10</font>
```

Figura 20 – Mensagem HTTP *Response*

Analisando-se a mensagem de resposta, é possível perceber que o teste não foi executado com sucesso (linha 13 da Figura 19), já que a tabela "spitable" não foi encontrada (linhas 17 e 18 da Figura 20). Entretanto, apesar do ocorrido, é possível concluir que a aplicação aceitou o código SQL enviado, já que avaliou os parâmetros enviados. Se o nome da tabela estivesse correto, seria possível consultá-la, alterá-la ou mesmo deletá-la do sistema.

Sendo assim, o uso do QAInspect auxilia no processo de teste, pois executa, de forma automatizada, testes que exigiriam pessoas qualificadas para executá-los. Além disso, o tempo despendido com a execução dos testes é menor do que se estes fossem realizados de forma manual, já que diversos casos podem ser verificados paralelamente. Uma limitação do *software* é que ele consegue executar apenas os casos pré-descritos pelas políticas de segurança, não sendo possível expandí-lo para suportar outros testes. Desta forma, se algum usuário malicioso criar um novo tipo de ataque, será preciso esperar até que uma nova versão do *software* seja disponibilizada, já que ele é comercial e de código fechado.

5.3 Considerações Finais

Mesmo sendo possível descrever características de segurança usando apenas UML e OCL, diversos trabalhos propõem *profiles* e *frameworks* de segurança para expandir o modelo UML. Jürjens propos o UMLSec, um *profile* UML que permite modelar e avaliar aspectos de segurança com o objetivo de garantir princípios básicos ao sistema, como marcar *links* seguros e descrever

a operação de troca justa. Lodderstedt *et al.* propuseram o SecureUML, uma extensão da UML para especificar políticas para controle de acesso através de notações gráficas e restrições OCL.

Outros trabalhos também propõem a expansão do modelo UML para representar políticas de controle de acesso, como os apresentados por Ahn and Hu [59] e Ray *et al.* [40]. O primeiro propôs um *framework* para integrar políticas de segurança e de validação de acesso no modelo, com o objetivo de orientar os desenvolvedores. O segundo utiliza técnicas de visualização para representar a violação de restrições de controle de acesso, ao invés de descrevê-las usando OCL.

Para auxiliar no processo de geração de casos de teste de segurança, objetivo deste travalho, é preciso um modelo que represente as vulnerabilidades que podem afetar o *software*, bem como em que partes deste elas podem ocorrer. Além disso, deve conter informações necessárias ao processo de teste, para que seja possível definir os casos que devem ser executados e qual o comportamento esperado. Apesar da quantidade de trabalhos encontrados sobre modelagem de aspectos de segurança, nenhum representa estas informações. Sendo assim, optou-se por descrever um conjunto de estereótipos próprios para representar estas características. A descrição de cada estereótipo, bem como a forma de uso, podem ser vistos no Capítulo 6.

Além das técnicas de modelagem de aspectos de segurança, também foram estudadas algumas ferramentas de geração e execução de casos de teste. Para tanto, foram analisadas as ferramentas Apache JMeter, STAGE e HP QAInspect. O objetivo da ferramenta Apache JMeter é realizar testes de desempenho e funcional em aplicações Web. Ela implementa uma série de funcionalidades que permitem fazer requisições para uma aplicação, ou mesmo enviar grandes quantidades de conteúdo para ela. Apesar de ser possível utilizá-la para executar ataques de segurança baseados em requisições, ela não foi desenvolvida com este propósito, apresentando limitações em executar casos simples, como estouro de *buffer*. Ela também não possui a funcionalidade de gerar casos de teste a partir de modelos, outra área de interesse deste trabalho.

A segunda ferramenta estudada, o STAGE, tem como objetivo gerar e executar casos de testes funcionais. Baseando-se no modelo da aplicação, descrito através de autômatos, a ferramenta retorna todas as combinações de caminhos possíveis e ainda gera a implementação destes no formato de entrada das ferramentas Rational XDE Tester e HttpUnit, ambas para automação de teste. A proposta da ferramenta é interessante, pois permite a geração de casos de teste a partir do modelo, mas possui duas limitações. A primeira é o tipo de modelo utilizado, autômatos finitos ao invés de UML, base deste trabalho. A segunda é o tipo de teste executado. Apesar de ser possível executar alguns casos de segurança como se estes fossem funcionais, grande parte dos testes exige informações adicionais, como os ataques de *flooding* e roubo de informações.

A última ferramenta avaliada foi a HP QAInspect. Apesar desta ser voltada à área de segurança, ela possui algumas limitações. Uma delas é o fato dela não pode ser expandida, o que significa que apenas os casos já existentes podem ser executados. Além disso, ela não trabalha com o modelo do sistema, mas diretamente na aplicação real. O principal problema associado a este fato é a falta de garantia de que todo o *software* foi coberto pelos casos executados, o que é facilmente visualizado através de modelos.

Considerando as limitações encontradas pelas ferramentas, optou-se por implementar um algoritmo que interprete o modelo UML com os estereótipos de segurança e gere os casos de teste automaticamente. Com os estereótipos é possível assinalar as partes do sistema que devem ser desenvolvidas cuidadosamente, caso contrário podem tornar-se vulneráveis. Já os casos de teste servem para indicar os passos que os testadores devem seguir para verificar a existência da vulnerabilidade. A descrição destas duas etapas do trabalho podem ser vistas no Capítulo 6.

6 Especificação e Geração de Casos de Teste

Devido a evolução dos sistemas de computadores e à quantidade de funcionalidades que estes provêem, os *softwares* adquiriram tamanha complexidade que mesmo testá-los tornou-se uma tarefa complicada. A dinâmica do mercado de tecnologia da informação exige a liberação de novas versões constantemente, estipulando prazos cada vez menores para isso. Esta pressa reduz o tempo dedicado à execução de testes no *software*, comprometendo não apenas o funcionamento deste, mas também outros aspectos importantes como segurança. Neste contexto, uma outra abordagem de teste vem se tornando popular, denominada de teste baseado em modelos.

O uso desta técnica auxilia o processo de teste de *software* pois permite gerar, automaticamente, casos de teste a partir do modelo da aplicação. Dois elementos são necessários para utilizá-la: um modelo do *software*, contendo as informações sobre o aspecto que será testado, e um gerador de casos de teste, que extrairá estas informações do modelo e as converterá em etapas do teste. Além de permitir a geração dos casos de teste, o modelo também serve para orientar os desenvolvedores, destacando as partes do *software* que podem tornar-se vulneráveis se não implementadas cuidadosamente. Ao mesmo tempo, os casos de teste gerados indicam os passos que os testadores devem executar para verificar se o uso do modelo foi efetivo e se as vulnerabilidades demarcadas realmente não afetam o *software*.

O objetivo deste trabalho não é validar o modelo, técnica conhecida como *model-checking*, mas utilizá-lo como base para gerar casos de teste e aplicá-los ao produto final, processo denominado de teste baseado em modelos. Para elaborá-lo, ele foi dividido em duas partes: a primeira consiste na definição de estereótipos para representar informações de segurança em modelos UML. A segunda é um algoritmo que analisa estes estereótipos, extrai as informações relevantes do modelo e gera os casos de teste. Sendo assim, este capítulo apresenta a motivação e objetivos deste trabalho, bem como descreve o funcionamento das etapas de especificação de aspectos de segurança e geração de casos de teste.

6.1 Motivação e Objetivos

Durante a elaboração deste trabalho, diversas áreas correlatas foram estudadas, como teste de *software*, no Capítulo 2, modelos formais, no Capítulo 3 e taxonomias de segurança, no Capítulo 4. A partir destas, foi possível obter as seguintes conclusões:

- Dentre os modelos formais estudados no Capítulo 3, o mais popular é o modelo UML

[32]. Isto se deve ao fato desta notação ser simples, abrangente e flexível, pois pode ser expandida através de *profiles*. Além disso, grande parte dos profissionais de informática possui algum tipo de treinamento nesta linguagem, o que facilita o seu uso. A UML também permite inserir restrições ao modelo através de uma linguagem própria para isto, chamada OCL [35]. Embora com esta seja possível representar diferentes características de um sistema, sua compreensão é difícil, o que faz com que os engenheiros de *software* estendam a UML através de estereótipos próprios, que representam as informações que se queira adicionar ao modelo.

- Sobre as três primeiras taxonomias estudadas (DARPA [45], Aslam [46,47] e Bazaz [48]), devido ao fato de abordarem características amplas dos objetos estudados, não contribuem com a identificação de um padrão de comportamento dos ataques e nem de estruturas que devam ser representadas pelos modelos. Neste sentido, escolheu-se como guia a taxonomia proposta por Sam Weber [49]. Ela é atual e utiliza como critério de classificação os tipos de falhas existentes, auxiliando a identificar que ataques podem derivar de uma falha específica. Para complementar as informações providas pela taxonomia de Weber, considerou-se a lista de vulnerabilidades disponibilizada pela OWASP [7]. Denominada *Top Ten List*, esta apresenta as 10 vulnerabilidades mais críticas para aplicações Web. Ela foi escolhida por ser a mesma base de dados de segurança utilizada por Weber para verificar a capacidade de classificação e abrangência de sua taxonomia [49].

- Sobre os *profiles* de segurança estudados, UMLSec [38] e SecureUML [39], ambos possuem limitações em relação à representação de vulnerabilidades de segurança, objetivo deste trabalho. O primeiro é um *profile* UML para modelar e avaliar aspectos de segurança, visando garantir princípios básicos no comportamento do sistema. O segundo é uma extensão da UML para descrever apenas políticas de segurança para controle de acesso ao sistema (RBAC), não sendo possível representar outros tipos de requisito.

Desta forma, com o auxílio da taxonomia de Weber e com as informações disponibilizadas pela base de dados da OWASP, foram especificados alguns estereótipos para representar as vulnerabilidades que podem comprometer a segurança do sistema. A lista foi utilizada tanto para fornecer informações a respeito das vulnerabilidades, como para identificar quais delas seria possível representar através de estereótipos e quais iriam requerer uma estrutura mais complexa.

O emprego destes estereótipos possui dois objetivos: orientar os desenvolvedores durante o processo de concepção do *software*, destacando partes do modelo que podem tornar-se vulneráveis se não implementados corretamente, e prover informações necessárias à geração automatizada de casos de teste. Este último indica os passos que devem ser executados pelo testador, a fim de verificar se a vulnerabilidade previamente assinalada no modelo afeta o *software* ou não. Para analisar o modelo e extrair as informações relevantes, utilizou-se o algoritmo *Unique*

Input/Output (UIO), que utiliza uma máquina de estados finitos com entradas e saídas para montar a seqüência de execução do *software* e gerar os casos de teste. A Figura 21 apresenta a arquitetura geral deste trabalho, descrevendo os passos seguidos desde a fase de modelagem dos estereótipos (destacados pelo quadro da esquerda), até a de geração dos casos de teste (destacados pelo quadro da direita).

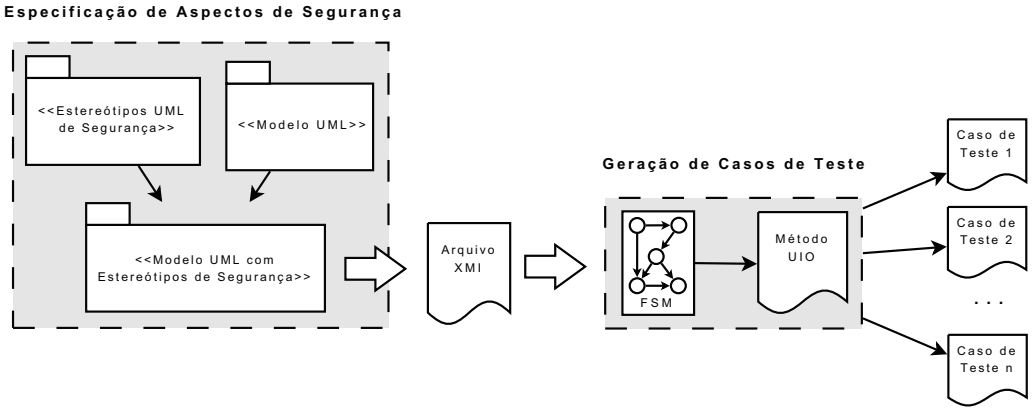

Figura 21 – Arquitetura geral proposta

Após a inserção dos estereótipos propostos no modelo UML da aplicação (quadro da esquerda da Figura 21), é gerado um arquivo XMI correspondente, descrevendo a estrutura do sistema, atividades, casos de uso e estereótipos. Este arquivo é utilizado como entrada para a segunda parte do trabalho, demarcada pelo quadro da direita da Figura. O arquivo XMI é analisado e convertido em uma máquina de estados finitos com entradas e saídas. Nesta, por sua vez, é aplicado o método UIO, cujas saídas são seqüências de passos que, após serem manipuladas, equivalem a casos de teste em linguagem natural. Além disso, é preciso ressaltar que o processo de especificação dos estereótipos associou a flexibilidade provida pelos estereótipos UML, ao conhecimento proveniente da base de dados do projeto OWASP. Sendo assim, para expandir o conjunto proposto e adicionar estereótipos capazes de representar outras vulnerabilidades, basta analisar uma base de segurança, extrair as informações necessárias ao teste e propôr o caso.

6.2 Modelagem dos Aspectos de Segurança

A partir da análise das taxonomias de segurança foi possível concluir que existem diversos tipos de vulnerabilidades que podem afetar uma aplicação. Ao mesmo tempo, o estudo dos trabalhos relacionados revelou que a maioria dos autores que define estratégias para especificar aspectos de segurança durante a fase de modelo estão focados em características específicas, como por exemplo controle de acesso. Visando representar características mais amplas de segurança, este trabalho propõe alguns estereótipos que permitem identificar partes do modelos que podem tornar-se vulneráveis caso sejam mal implementados, ou especificar alguns requi-

sitos de segurança, como criptografia. A inserção destes estereótipos no modelo tem como objetivos orientar os desenvolvedores para que estes evitem algumas falhas, bem como prover informações necessárias à geração automatizada de casos de teste.

Sendo assim, os estereótipos definidos correspondem ao conjunto de testes que será aplicado no *software* sob avaliação. Entretanto, antes de se definir os testes, é preciso considerar o tipo de aplicação que está sendo testada. Isto por que o conjunto utilizado para avaliar um sistema Web, por exemplo, é diferente do aplicado em um sistema operacional ou em um *software* para gerenciamento de banco de dados. Para tanto, existem diversas bases de dados de segurança que elaboram, regularmente, listas com as vulnerabilidades mais críticas para cada tipo de aplicação. Exemplos destas bases são as do projeto OWASP [7], do Instituto NIST [60], do Instituto SANS [61] e do grupo SecurityFocus [62]. A associação das informações disponibilizadas por estas bases de dados, com a flexibilidade provida pelos estereótipos UML, nos permitiu definir estereótipos específicos para representar as informações de segurança.

Considerando a quantidade de serviços disponibilizados pela Internet e o volume de informações enviados através desta, os estereótipos propostos neste trabalho estão voltados à aplicações Web. Obviamente alguns casos podem ser aplicados em outros tipos de *software*, mas o foco abordado por este trabalho, bem como os estudos de caso utilizados, serão aplicações Web. Para auxiliar o processo de definição dos estereótipos, a base de dados utilizada foi a disponibilizada pelo projeto OWASP [7]. Este projeto mantém uma lista com as 10 vulnerabilidades mais críticas que afetam aplicações Web, conhecida como *Top Ten List* (a última versão foi liberada em 2007). Além de indicar as vulnerabilidades, a lista também apresenta uma breve descrição de cada uma e explica como verificar a existência destas, o que foi útil tanto para a geração dos casos de teste, como para definir que vulnerabilidades seria possível especificar através de estereótipos e quais iriam requerer uma estrutura mais complexa. A Tabela 4 apresenta as vulnerabilidades descritas por esta lista nas duas versões existentes, a de 2004 e de 2007.

De acordo com as categorias apresentadas pela taxonomia de Weber e com as vulnerabilidades extraídas da base de dados do projeto OWASP, apresentados na Tabela 4, os seguintes estereótipos foram definidos:

- **Caso 1 - *Buffer Overflow*:** esta vulnerabilidade consiste em atribuir a uma variável um dado cujo comprimento é maior do que a estrutura pode receber. Neste caso, serão considerados apenas os campos de texto que exigem interação do usuário, como nome e senha, por exemplo. O estereótipo deve ser inserido na atividade do diagrama onde ocorre a entrada dos dados do usuário.
  Estereótipo: «BufferOverflow»
  Tag: {BOFlow = {<field>, <size>}}
  onde <field> é o nome do campo para inserção dos dados e <size> é o tamanho máximo de caracteres permitido para este campo.

- **Caso 2 - *Flooding* de Conexões**: esta vulnerabilidade permite iniciar, simultaneamente,

Tabela 4 – *Top Ten List*, nas versões 2004 e 2007

	Top 10 2004	**Top 10 2007**
1	Entrada sem Validação	*Cross Site Scripting*
2	Quebra do Controle de Acesso	Injeção de Falhas
3	Quebra da Autenticação e Gerenciamento de Sessão	Execução Maliciosa de Arquivos
4	*Cross Site Scripting*	Referência Insegura Direta a Objeto
5	Estouro de *Buffer*	*Cross Site Request Forgery*
6	Injeção de Falhas	Vazamento de Informações e Manipulação Incorreta de Erros
7	Manipulação Incorreta de Erros	Quebra da Autenticação e Gerenciamento de Sessão
8	Armazenamento Inseguro	Armazenamento Criptográfico Inseguro
9	Negação de Serviço	Comunicação Insegura
10	Comunicação Insegura	Falha ao Restringir o Acesso à URLs

mais conexões do que o suportado pelo provedor do serviço. O estereótipo deve ser inserido na atividade do diagrama que representa a situação de conexão estabelecida.
Estereótipo: «Flooding»
Tag: {FDMaxConn = {<max_connection>}}
onde <max_connection> é o número máximo de conexões simultâneas permitidas.

- **Caso 3 - Criptografia (Conexões):** esta vulnerabilidade consiste em enviar dados sensíveis através da rede, sem criptografar a conexão. Neste caso, a fim de auxiliar no processo de geração de casos de teste, devem ser indicados os campos que necessitam de criptografia. O estereótipo deve ser inserido na atividade do diagrama responsável pelo envio dos dados pela rede.
  Estereótipo: «Encrypt»
  Tag: {CRField = {<field>}}
  onde <field> é o nome do campo a ser criptografado.

- **Caso 4 - Controle de Acesso (*By Passing*):** esta vulnerabilidade consiste em solicitar os dados de *login* do usuário apenas na página inicial, não sendo requisitado caso o usuário acesse um *link* interno do sistema diretamente. Para auxiliar na geração de casos de teste, é preciso inserir três tipos de *tags* nos diagramas, todas integrantes do estereótipo «By-Passing». Em primeiro lugar deve ser feita a identificação dos usuários do sistema. Para tanto, é preciso inserir, nos atores do diagrama de caso de uso, o estereótipo «ByPassing» com a *tag* <BPRole>. O próximo passo é mapear os *links* das páginas protegidas pelo sistema de *login*. Este procedimento é feito através da inserção do mesmo estereótipo, mas usando a *tag* <BPLink>. Por último, é preciso identificar a partir de que ponto os demais usuários não podem acessar o sistema. Para tanto, pode-se optar pelo uso da *tag* <BPAl-

lowed>, para indicar que usuários podem acessar o sistema, ou da *tag* <BPDenied>, para indicar que usuários não podem acessar o sistema.

Estereótipo: «ByPassing»
Tag: {BPRole}
sem valores, pois é apenas para marcar o ator.

Estereótipo: «ByPassing»
Tag: {BPLink = {<link>}}
onde <link> é o endereço a ser acessado.

Estereótipo: «ByPassing»
Tag: {BPAllowed = {<role>}}
onde <role> é o ator que tem acesso ao sistema.

Estereótipo: «ByPassing»
Tag: {BPDenied = {<role>}}
onde <role> é o ator que não tem acesso ao sistema.

- **Caso 5 - Controle de Acesso (Expiração de sessão):** esta vulnerabilidade acontece quando a sessão do usuário não expira após determinado tempo sem interação deste. Pode ocorrer também caso o tempo de expiração seja relativamente grande que possa comprometer a segurança do usuário, mas este tempo deve ser previsto pelo engenheiro do *software*. O estereótipo deve ser inserido após a atividade do diagrama que representa o *login* bem sucedido do usuário.
  Estereótipo: «Expiration»
  Tag: {ExpTime = {<time>, <unit>}}
  onde <time> indica o intervalo de tempo que a sessão deve expirar e <unit> a unidade de medida, que pode ser ‘s’ (segundo), ‘m’ (minuto), ‘h’ (hora) ou ‘d’ (dia).

- **Caso 6 - SQL *Injection*:** esta vulnerabilidade acontece quando o sistema não faz validação dos dados de entrada antes de atribuí-los a uma query SQL. Desta forma, o usuário malicioso insere no campo de dados uma outra query SQL, ao invés do nome de usuário ou senha, por exemplo. A execução deste caso de teste exige mais conhecimento do testador, que precisa avaliar os resultados obtidos do sistema para elaborar novos casos de teste. Para auxiliá-lo, serão fornecidos junto com o caso algumas sugestões de querys de busca. A execução deste teste tem como objetivo detectar duas vulnerabilidades: a susceptibilidade do sistema a execução de códigos SQL (SQL Injection) e a exposição de informações sensíveis, que podem ser senhas de outros usuários, informações pessoais

ou mesmo nomes das estruturas internas do sistema. O estereótipo deve ser inserido nas atividades do diagrama que possuem captura de dados provenientes do usuário.
Estereótipo: «SqlInjection»
Tag: {SQLField = {<field>}}
onde <field> é o campo para inserção de dados.

Considerando a quantidade de vulnerabilidades existentes, apenas algumas foram mapeadas. Para simplificar a modelagem, foram utilizados apenas diagramas de atividades para inserir as informações (exceto no caso do estereótipo «ByPassing», que exige a marcação também no diagrama de casos de uso). Por último, é importante ressaltar que os estereótipos apresentados neste trabalho foram publicados na forma de um artigo estendido [63].

Após a modelagem, deve-se gerar o arquivo XMI. Este será utilizado como entrada do gerador de casos de teste, que irá interpretá-lo e extrair as informações necessárias aos testes. A descrição do funcionamento do gerador de casos de teste pode ser vista na seção seguinte.

6.3 Geração dos Casos de Teste

Conforme mencionado, a segunda parte deste trabalho consiste na geração de casos de teste de segurança a partir dos estereótipos adicionados ao modelo. O objetivo destes casos é indicar ao testador os passos que este deve executar para alcançar a parte da aplicação que poderia tornar-se vulnerável caso não fosse implementada cuidadosamente. Sendo assim, é preciso adicionar os estereótipos propostos a algum diagrama da UML que descreva o modelo comportamental do *software*, tais como diagramas de atividades, estados ou de seqüência. De posse desta representação, resta apenas interpretar o modelo e gerar os casos de teste.

Uma das vantagens em se utilizar modelos comportamentais que descrevam uma seqüência de passos, é a facilidade em convertê-los em máquinas de estados finitos. Também existem inúmeras técnicas para percorrer estas estruturas e gerar, automaticamente, casos de teste. Uma destas utiliza o algoritmo *Unique Input/Output* (UIO) [8], que permite identificar o estado particular em que a máquina de estados finitos se encontra baseando-se apenas nas entradas e saídas deste. Com esta técnica, é possível definir uma seqüência de passos que conduza a máquina desde o estado inicial *s0* até determinado estado *si*, aplicar o símbolo de entrada para aquele estado e, por fim, calcular o estado de saída que deveria ser atingido. Após o cálculo de todas as seqüências de entradas e saídas, é preciso gerar os casos de teste e aplicar estes ao *software*, verificando se a implementação é fiel ou não ao comportamento descrito pela máquina.

Para implementar o gerador da casos de teste, o primeiro passo foi desenvolver um *parser* que avaliasse o arquivo XMI gerado a partir do modelo UML com os estereótipos, extraísse as informações necessárias e elaborasse uma máquina de estados finitos com entradas e saídas. Considerando que o método UIO utiliza as transições da máquina para determinar o estado que

deve ser alcançado, o conteúdo dos estereótipos deve ser inserido nestas, como entrada dos estados, dando origem assim a uma máquina de Mealy [64]. A Figura 22 apresenta um exemplo de diagrama de estados com estereótipo, enquanto que a Figura 23 demonstra a máquina de estados finitos equivalente ao diagrama. É importante ressaltar que, para o correto funcionamento da máquina, é preciso incluir um estado inicial e conectar todos os outros nodos a ele, permitindo assim que ela seja reiniciada.

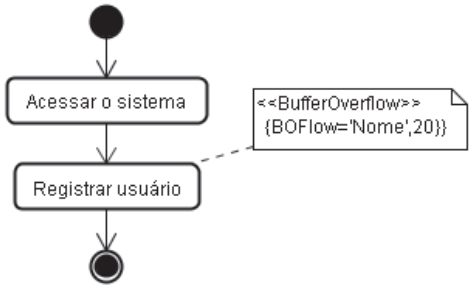

Figura 22 – Exemplo de diagrama de atividades

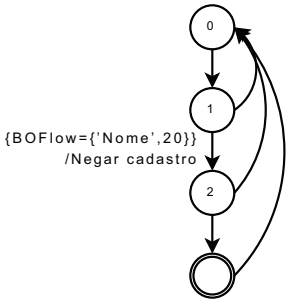

Figura 23 – FSM equivalente

A partir desta máquina, serão geradas as seqüências UIO para cada estereótipo, que indicam o caminho para acessar os estados. Por fim, com base nestas seqüências é que são gerados os casos de teste, que no escopo deste trabalho estão descritos em linguagem natural, mas que podem ser adaptados e utilizados como entrada de alguma ferramenta de execução de *scripts* de teste. É importante salientar que, para cada estereótipo adicionado ao modelo, é gerado um caso de teste, com exceção dos casos de «BufferOverflow» e «Flooding», que geram três casos, um para o limite inferior, um para o limite exato e um para o limite superior.

O formato de geração dos casos de teste, para cada estereótipo, pode ser visto na Tabela 5.

6.4 Considerações Finais

Conforme mencionado, o objetivo deste trabalho é especificar aspectos de segurança em modelos UML e permitir a geração de casos de teste a partir deste. Para tanto, ele foi dividido em duas partes: a primeira consiste em definir um conjunto de estereótipos UML capazes de indicar vulnerabilidades de segurança que podem afetar o *software*, enquanto que a segunda visa extrair informações do modelo contendo estes estereótipos e gerar casos de teste automaticamente, permitindo que um testador verifique a existência das vulnerabilidades assinaladas.

Para auxiliar no processo de definição dos estereótipos, foi utilizada a base de dados de segurança disponibilizada pelo projeto OWASP, conhecida como *Top Ten List*. Esta lista relata as 10 vulnerabilidades mais críticas de segurança que afetam aplicações Web, tendo sido liberada em duas versões: uma em 2004 e outra, mais recente, em 2007. Considerando as vulnerabilidades destacadas pelas duas listas, apresentadas na Tabela 4, é possível concluir que o uso

Tabela 5 – Formato de Geração dos Casos de Teste

Estereótipo	Caso de Teste
«BufferOverflow» (3 casos de teste)	1. Executar as atividades até a atividade que contém o estereótipo. 2. Preencher o campo com tamanhos equivalentes aos valores limite. 3. Resultado esperado: aceito para os limites inferior e exato, e negado para o limite superior.
«Flooding» (3 casos de teste)	1. Executar as atividades até a atividade que contém o estereótipo. 2. Repetir o procedimento descrito na atividade 1, avaliando o sucesso de conexão do número limite *n* de usuários (*n*-1, *n* e *n*+1). 3. Resultado Esperado: logins aceitos para *n*-1 e *n*, e negado para *n*+1.
«Encrypt»	1. Executar a seqüência de atividades até a atividade anterior da que contém o estereótipos. 2. Solicitar que o usuário execute um analisador de tráfego. 3. Executar a atividade assinalada com o estereótipo. 4. Solicitar que o usuário pare a execução do analisar de tráfego. 5. Resultado Esperado: o valor enviado no campo marcado pelo estereótipo não pode ser visto com o analisador de tráfego.
«Expiration»	1. Executar a seqüência de atividades até a atividade anterior da que contém o estereótipos. 2. Esperar o tempo indicado pelo estereótipo sem interagir com o *software*. 3. Executar a atividade assinalada com o estereótipo. 4. Resultado Esperado: o *software* deve negar o acesso e solicitar o login novamente.
«ByPassing»	1. Fazer *login* usando o nome de usuário indicado pelo estereótipos. 2. Percorrer as atividades até a assinalada pelo estereótipos. 3. Copiar o *link* da barra de endereços do *browser*. 4. Fazer *logoff*. 5. Tentar acessar o *link* copiado na atividade 3. 6. Resultado Esperado: o acesso ao *link* deve ser negado.
«SqlInjection»	1. Percorrer as atividades até a que contém o estereótipo. 2. Inserir código SQL malicioso no campo indicado pelo estereótipo. 3. Resultado Esperado: os comandos SQL devem ser rejeitados.

dos estereótipos consegue representar grande parte destas, auxiliando a prever casos como o de Injeção de Falhas, considerado a segunda vulnerabilidade mais crítica da lista de 2007. Com os estereótipos apresentados na Seção 6.2, foi possível representar os casos de número 1, 2, 3, 5, 6, 7 e 9, da lista de 2004, e os de número 2, 6, 7, 9 e 10 da lista de 2007. No total, dos 16 casos apresentados por ambas as listas (4 são casos repetidos), 9 puderam ser modelados com os estereótipos, o que significa que mais de 50% das vulnerabilidades consideradas críticas foram representadas. Concluída esta etapa, é preciso analisar os demais casos, identificar as características essenciais aos testes e, por fim, expandir o conjunto de estereótipos, sendo possível representar também casos como os de *Cross Site Scripting*(XSS), Execução Maliciosa

de Arquivos e Armazenamento Inseguro, por exemplo.

Em relação à segunda etapa do trabalho, existem diversos outros métodos para geração de seqüências de teste a partir de máquinas de estados finitos, tais como o Método *Transition Tour* (Método TT) [8], Método *Distinguishing Sequence* (Método DS) [8] e o Método *Characterizing Sets* (Método W) [8]. Entretanto, comparado às outras técnicas, o método UIO vem se tornando popular por dois motivos principais: ele auxilia na detecção de falhas na transição de estados, mesmo de máquinas parcialmente especificadas, e produz as seqüências de teste mais curtas que os outros métodos [65].

A fim de verificar a capacidade dos estereótipos em representar aspectos de segurança, e a clareza com que os casos de teste especificam os passos a serem seguidos, foram selecionados quatro estudos de caso. A descrição dos sistemas, bem como a modelagem de cada um, os casos gerados para cada requisito e os resultados obtidos com a execução manual destes podem ser vistos no capítulo a seguir.

7 Estudos de Caso

Para verificar a eficiência de representação dos estereótipos propostos, bem como avaliar se os passos descritos pelo gerador de casos de teste permitem identificar vulnerabilidades em um sistema, é preciso aplicar a estratégia proposta em alguns estudos de caso. Considerando que a base de dados utilizada como guia é voltada a aplicações Web, os estudos de caso escolhidos devem ser *softwares* deste tipo, como por exemplo sistemas de *e-commerce*. Além disso, a aplicação avaliada deve ser *open-source*, para que se possa identificar as causas das vulnerabilidades, e conter a modelagem UML do sistema, onde serão adicionados os estereótipos.

Quatro aplicações foram selecionadas como estudo de caso. As duas primeiras, OSCommerce [66] e TPC-W [67], reproduzem o comportamento de um *site* de *e-commerce*. A terceira, conhecida como Moodle [68], é um *software* de apoio à aprendizagem que permite a criação de cursos on-line e disponibiliza uma série de recursos a serem utilizados por professores e alunos. A última, denominada CesarFTP [69], é um servidor de FTP gratuito e fácil de utilizar e de configurar. Para avaliá-los, inicialmente foi realizada uma análise dos requisitos dos *softwares* e identificados os que poderiam ser representados pelos estereótipos propostos. Após, estes estereótipos foram inseridos nos respectivos modelos UML, permitindo que a partir destes fossem gerados os casos de teste correspondentes. Por fim, executou-se os testes sugeridos para verificar se nenhuma das vulnerabilidades assinaladas no modelo existiam de fato, ou seja, se os requisitos de segurança foram respeitados.

É importante ressaltar ainda que, para fins de validação, foram utilizados apenas diagramas de atividades (exceto durante a representação do estereótipo «ByPassing» que exige um diagrama de casos de uso para identificação dos atores do sistema). As seções seguintes apresentam uma breve descrição de cada uma destas ferramentas, bem como demonstram a inserção dos estereótipos de segurança na modelagem dos sistemas, os casos de teste gerados a partir destes e, ao final, os resultados obtidos quando executou-se manualmente os casos.

7.1 OSCommerce

O OSCommerce, ou *Open-Source Commerce* [66], é um *software* que agrega funções tanto de comércio eletrônico, como de gerenciamento de lojas *online*. Ele provê um conjunto básico de funcionalidades que vão ao encontro das necessidades da maioria dos comerciantes *online*, sendo possível configurar, executar e atualizar lojas *online* com o mínimo de esforço, sem cus-

tos, taxas ou limitações. Entre os recursos disponibilizados pelo *software* estão o cadastro de clientes, o carrinho de compras, opções de pagamento (boleto, cartão, depósito bancário, contra entrega), opções de entrega (encomenda normal, registrada e Sedex) e página administrativa.

Considerando o vasto conjunto de funcionalidades providas pelo *software*, bem como a importância de se avaliar aplicações deste tipo (*e-commerce*), o OSCommerce foi escolhido como um dos estudos de caso deste trabalho. Para tanto, inicialmente foi realizada uma análise dos requisitos do *software* e identificados os que poderiam ser representados pelos estereótipos. A seguir, foram gerados e executados os casos de teste correspondentes, a fim de identificar o atendimento ou não dos requisitos previamente estabelecidos.

Considerando que a configuração de alguns parâmetros da aplicação é feita pelo administrador do sistema, foram definidas as seguintes características:

- **Requisito 1:** O nome dos fabricantes, adicionado pelo administrador, deve possuir no máximo 15 caracteres.
- **Requisito 2:** O envio de senhas, durante o processo de autenticação do usuário, deve ser criptografado.
- **Requisito 3:** O sistema deve fazer logoff do usuário quando não houver interação deste com o sistema em um período de 180 segundos.
- **Requisito 4:** As informações inseridas no campo de busca por produtos devem ser analisadas previamente a fim de evitar ataques do tipo SQL *Injection*.
- **Requisito 5:** O acesso ao carrinho de compras deve ser feito apenas por clientes cadastrados.

Os requisitos acima podem ser representados pelos estereótipos descritos na Tabela 6.

Tabela 6 – Mapeamento dos Requisitos de Segurança do OSCommerce

Requisito	Estereótipo	*Tag*
Requisito 1	«BufferOverflow»	{BOFlow={‘ManufacturersName’,15}}
Requisito 2	«Encrypt»	{CRField={password}}
Requisito 3	«Expiration»	{ExpTime={180,‘s’}}
Requisito 4	«SqlInjection»	{SQLField={SearchField}}
Requisito 5	«ByPassing» «ByPassing»	{BPLink={/cart.php}} {BPAllowed={Cliente}}

A inserção dos estereótipos nos diagramas de atividades equivalentes pode ser vista na Figura 24, que apresenta como é feita a inserção dos estereótipos «BufferOverflow», «Encrypt»

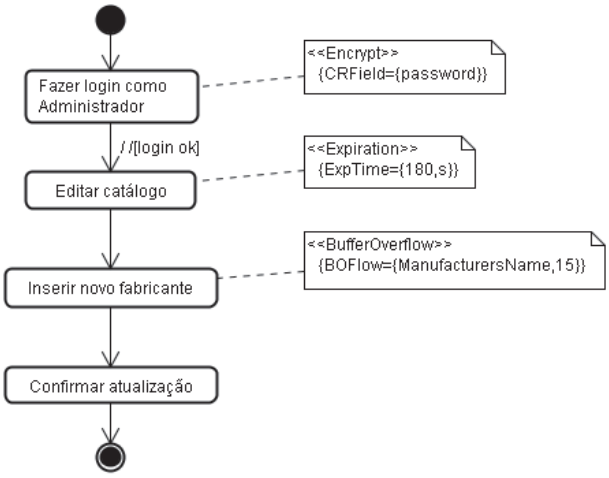

Figura 24 – Inserção dos Estereótipos «BufferOverflow», «Encrypt» e «Expiration»

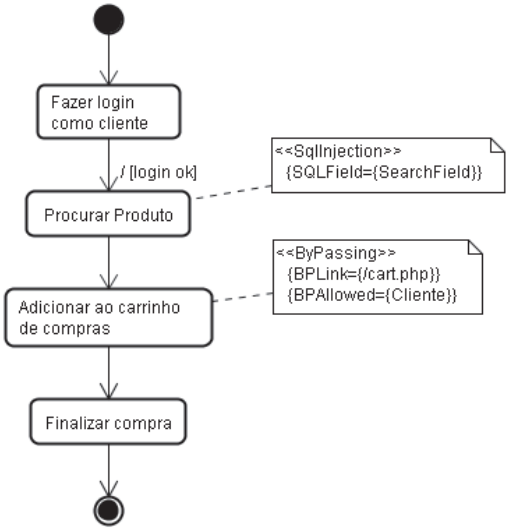

Figura 25 – Inserção dos Estereótipos «SqlInjection» e «ByPassing»

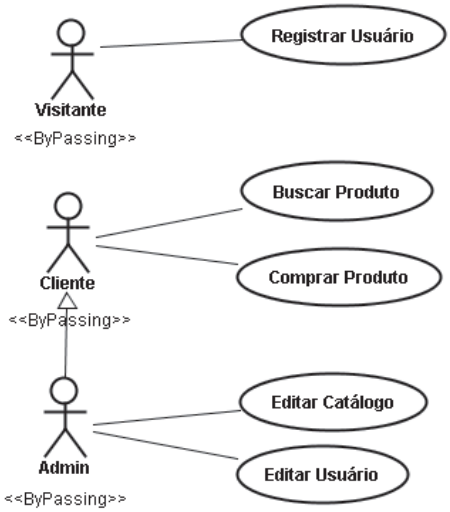

Figura 26 – Diagrama de Casos de Uso do OSCommerce

e «Expiration», na Figura 25, que demonstra o uso dos estereótipos «SqlInjection» e «ByPassing», e na Figura 26, onde são marcados os atores nos casos de uso.

Após a modelagem dos requisitos, gerou-se o arquivo XMI correspondente, o qual foi utilizado como entrada para o algoritmo de geração de casos de teste. Como resultado obteve-se os passos necessários para verificar a parte do sistema que pode conter uma vulnerabilidade, e então avaliá-lo. Para cada estereótipo inserido foi gerado um caso de teste equivalente (exceto nos casos de *BufferOverflow* e *Flooding* que realizam testes com os valores limites).

Desta forma, os casos de teste gerados para o *software* OSCommerce podem ser vistos na Tabela 7.

Tabela 7 – Casos de Teste Gerados para o OSCommerce

Estereótipo	Caso de Teste
«Encrypt»	1. Executar analisador de tráfego. 2. Fazer login como administrador. 3. Parar execução do analisador de tráfego. 4. Resultado Esperado: o valor enviado no campo ‘password’ não pode ser visualizado.
«Expiration»	1. Fazer login como administrador. 2. Esperar durante 180 segundos sem interação com o *software*. 3. Editar catálogo. 4. Resultado Esperado: acesso negado.
«BufferOverflow»	1. Fazer login como administrador. 2. Editar catálogo. 3. Inserir novo fabricante. 4. Inserir no campo ‘ManufacturersName’ um valor com comprimento menor que 15 caracteres. 5. Resultado Esperado: valor aceito.
	1. Fazer login como administrador. 2. Editar catálogo. 3. Inserir novo fabricante. 4. Inserir no campo ‘ManufacturersName’ um valor com comprimento igual a 15 caracteres. 5. Resultado Esperado: valor aceito.
	1. Fazer login como administrador. 2. Editar catálogo. 3. Inserir novo fabricante. 4. Inserir no campo ‘ManufacturersName’ um valor com comprimento maior que 15 caracteres. 5. Resultado Esperado: valor rejeitado.
«SqlInjection»	1. Fazer login como cliente. 2. Procurar produto. 3. Inserir código SQL malicioso no campo ‘searchField’. 4. Resultado Esperado: código rejeitado.
«ByPassing»	1. Fazer login usando o nome de usuário ‘Cliente’. 2. Procurar produto. 3. Adicionar ao carrinho de compras. 4. Copiar o *link* shopping_cart.php?osCid=<...> 5. Logoff. 6. Acessar o *link* copiado shopping_cart.php?osCid=<...> 7. Resultado Esperado: acesso negado.

A Tabela 8 descreve os resultados obtidos com a execução manual dos casos de teste gerados.

Tabela 8 – Resultados Obtidos com a Execução Manual dos Casos de Teste no OSCommerce

Estereótipo	Resultado do Teste
«BufferOverflow»	O sistema aceitou o cadastro de fabricantes quando o comprimento do nome foi maior que 15 caracteres. Entretanto, nenhuma instabilidade foi detectada no comportamento do *software*.
«Encrypt»	Apesar do uso do protocolo SSL estar habilitado, o envio das informações de *login* do usuário foi feita de forma aberta pela rede, sendo possível capturar a senha de administrador usando um analisador de tráfego.
«Expiration»	Mesmo configurando a aplicação para expirar a sessão do usuário após 3 minutos de inatividade, esta não comportou-se como esperado. Entretanto, vale ressaltar que após aproximadamente 15 minutos de inatividade o *software* solicitou novamente os dados de *login* do usuário.
«SqlInjection»	A aplicação não foi vulnerável a ataques de SQL, ignorando os comandos enviados no campo de busca por produtos.
«ByPassing»	O sistema mostrou-se vulnerável apenas quando o usuário real está conectado ao sistema em outra máquina. Neste caso foi possível acessar o *link* interno do carrinho de compras a partir de outra máquina. Entretanto, quando o usuário real desconecta-se do sistema, o acesso direto ao *link* é negado, sendo solicitados os dados de login novamente.

Sendo assim, a avaliação do *software* OSCommerce expôs uma série de vulnerabilidades. Dos cinco casos de teste aplicados, quatro revelaram defeitos na aplicação, representando um risco ao sistema e aos seus usuários. Considerando que a aplicação é um *site* de *e-commerce*, um dos casos mais importantes é o de criptografia, que mesmo com o uso do protocolo SSL habilitado, enviou as informações de usuário e senha em texto puro pela rede. Em um primeiro momento, existem duas possibilidades para a existência desta vulnerabilidade: ou o protocolo SSL ainda não está implementado na versão testada (versão 2.2 M2), ou existe um problema na utilização deste com o *software*.

Outra vulnerabilidade crítica do sistema é a deficiência na expiração da sessão do usuário, que pode expor informações confidenciais do usuário, como número de cartão de crédito, ou permitir que um usuário malicioso assuma a identidade de um verídico. Diversos motivos podem estar provocando este problema, desde o uso de um valor padrão de tempo, desconsiderando a configuração do administrador, até a incapacidade em alterar o tempo de inatividade no servidor PHP, que pode controlar as sessões. Por último, vale ressaltar a vulnerabilidade de *By Passing*, que permite que um *link* interno seja acessado a partir de outra máquina se o usuário em questão estiver conectado ao servidor. Provavelmente a permissão de acesso do usuário é armazenada diretamente no servidor, e não na máquina do usuário. Sendo assim, enquanto o

usuário estiver autorizado no servidor, seu identificador (ID) é considerado válido, podendo ser acessado independentemente da máquina onde esteja.

7.2 TPC-W

O TPC-W [67] é um *software* desenvolvido para realizar testes de desempenho em infra-estruturas. Ele pode ser utilizado, por exemplo, para verificar o número máximo de usuários que um ambiente pode suportar, ou se este dispõe da quantidade de recursos necessária para prover um serviço. Para desempenhar esta função, ele simula o comportamento de um *site* de *e-commerce*, disponibilizando operações como buscar produtos, adicionar ao carrinho de compras e conectar-se ao sistema.

De acordo com Sopitkamol e Menascè [70], é muito difícil, se não impossível, conduzir experimentos em *sites* de *e-commerce* reais, o que faz do uso do TPC-W uma boa opção. Além disso, existe uma grande quantidade de informações sobre este *software*, o que inclui um documento de especificação completo e alguns diagramas UML construídos pelos desenvolvedores. Apesar de ser considerado um padrão para fazer avaliação de desempenho, sua capacidade em simular o comportamento de uma aplicação Web comum permite que outros tipos de teste sejam executados nele, como segurança. Por este motivo, o TPC-W não será utilizado neste trabalho como um "*benchmark*", mas como uma aplicação Web comercial onde serão avaliados alguns aspectos de segurança.

Da mesma forma que o OSCommerce, o primeiro passo foi fazer um levantamento dos requisitos de segurança do TPC-W e modelá-los através dos estereótipos propostos na Seção 6.2, para então gerar os casos de teste a partir do modelo UML. Conforme mencionado, o TPC-W provê uma série de ações ao usuário, reproduzindo as principais operações realizadas durante a visita a um *site* de *e-commerce*. No contexto, é simulado um *site* para compra de livros. Analisando-se seu documento de especificação, foram extraídos quatro requisitos de segurança, que se não implementados podem comprometer a segurança do sistema. São eles:

- **Requisito 1:** O campo de primeiro nome do cliente, preenchido durante o cadastro deste, deve ter no máximo 15 caracteres.
- **Requisito 2:** O envio da página que contém as informações da compra deve criptografar o número do cartão de crédito do cliente.
- **Requisito 3:** A alteração de preço dos livros deve ser feita por um usuário administrador conectado ao sistema.
- **Requisito 4:** As bases de dados devem ser protegidas de usuários não autorizados, como no caso de atualização de preços.

A Tabela 9 apresenta a modelagem destes requisitos utilizando os estereótipos, enquanto que as Figuras 27, 28 e 29 demonstram a inserção destes no modelo UML.

Tabela 9 – Mapeamento dos Requisitos de Segurança do TPC-W

Requisito	Estereótipo	*Tag*
Requisito 1	«BufferOverflow»	{BOFlow={'FirstName',15}}
Requisito 2	«Encrypt»	{CRField={CX_NUM}}
Requisito 3	«ByPassing» «ByPassing»	{BPLink={/Update.html}} {BPAllowed={Admin}}
Requisito 4	«SqlInjection»	{SQLField={PRECO}}

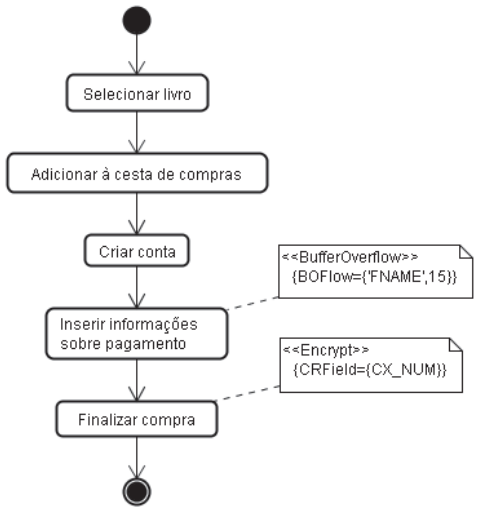

Figura 27 – Inserção dos Estereótipos «BufferOverflow» e «Encrypt»

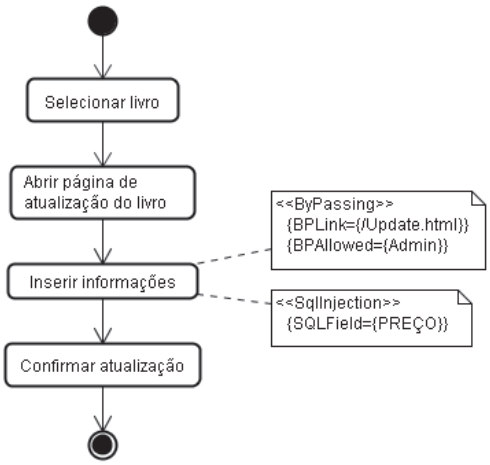

Figura 28 – «SqlInjection» e «ByPassing»

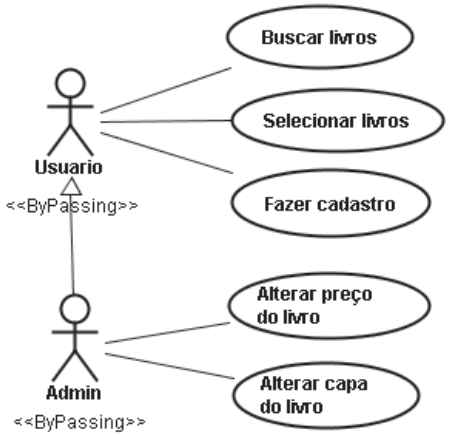

Figura 29 – Diagrama de Casos de Uso do TPC-W

Assim como no OSCommerce, após a modelagem dos requisitos foram gerados os casos de teste. Para cada estereótipo inserido foi gerado um caso de teste, descritos na Tabela 10.

Tabela 10 – Casos de Teste Gerados para o TPC-W

Estereótipo	Caso de Teste
«BufferOverflow»	1. Selecionar livro. 2. Adicionar à cesta de compras. 3. Criar conta. 4. Inserir no campo ‘FNAME’ um valor com comprimento menor que 15 caracteres. 5. Resultado Esperado: valor aceito.
	1. Selecionar livro. 2. Adicionar à cesta de compras. 3. Criar conta. 4. Inserir no campo ‘FNAME’ um valor com comprimento igual a 15 caracteres. 5. Resultado Esperado: valor aceito.
	1. Selecionar livro. 2. Adicionar à cesta de compras. 3. Criar conta. 4. Inserir no campo ‘FNAME’ um valor com comprimento maior que 15 caracteres. 5. Resultado Esperado: valor rejeitado.
«Encrypt»	1. Selecionar livro. 2. Adicionar à cesta de compras. 3. Criar conta. 4. Inserir informações sobre pagamento. 5. Executar analisador de tráfego. 6. Finalizar compra. 7. Parar execução do analisador de tráfego. 8. Resultado Esperado: o valor enviado no campo ‘CX_NUM’ não pode ser visualizado.
«ByPassing»	1. Fazer login usando o nome de usuário ‘Admin’. 2. Selecionar livro. 3. Abrir página de atualização do livro. 4. Copiar o *link* Update.htm 5. Logoff. 6. Acessar o *link* copiado Update.htm 7. Resultado Esperado: acesso negado.
«SqlInjection»	1. Selecionar livro. 2. Abrir página de atualização do livro. 3. Inserir código SQL malicioso no campo ‘PREÇO’. 4. Resultado Esperado: código rejeitado.

Os resultados obtidos com a execução manual dos casos de teste gerados podem ser vistos na Tabela 11.

Tabela 11 – Resultados Obtidos com a Execução Manual dos Casos de Teste no TPC-W

Estereótipo	Resultado do Teste
«BufferOverflow»	O TPC-W aceitou o cadastro do usuário com o primeiro nome sendo de comprimento maior que 15 caracteres, mas nenhum comportamento anormal foi detectado no *software*.
«Encrypt»	O *software* não criptografou o número do cartão de crédito do usuário antes de enviá-lo pela rede, sendo possível roubá-lo usando um analisador de tráfego.
«SqlInjection»	O *software* não foi vulnerável aos ataques de SQL.
«ByPassing»	O sistema não solicitou o login do usuário antes de atualizar o preço dos livros, aceitando o novo valor.

Analizando-se a Tabela 11, é possível concluir que o TPC-W não é uma aplicação segura. Se fosse um sistema de *e-commerce* real, informações pessoais dos usuários seriam enviadas sem criptografia e usuários maliciosos poderiam modificar os preços dos produtos. Uma observação a ser feita é que o *software* utiliza consultas SQL pré-compiladas, evitando assim que a estrutura da consulta original seja modificada e impedindo que ataques de SQL *Injection* ocorram.

7.3 Moodle

O Moodle, ou *Modular Object-Oriented Dynamic Learning Environment* [68], é um sistema de gerenciamento de cursos projetado para auxiliar os educadores a criar comunidades efetivas de aprendizado *online*. Ele constitui-se em um sistema de administração de atividades educacionais, destinado à criação de comunidades em ambientes virtuais voltados à aprendizagem colaborativa. Para tanto, utiliza-se de diversos recursos, tais como áreas para *download*, *chats*, fóruns de discussão, calendário e questionários.

A plataforma Moodle vem sendo aplicada em diversas instituições de ensino, tanto nas que utilizam o sistema de educação a distância, como as de cursos presenciais. Além disso, já vem sendo utilizada por outros setores não ligados à educação, como empresas privadas e ONGs. Por ser uma aplicação Web e conter informações pessoais de seus usuários, é essencial que implemente algumas diretrizes básicas de segurança. Sendo assim, considerando a sua complexidade e ampla utilização, o Moodle também foi utilizado como um dos estudo de caso deste trabalho. Da mesma forma como nos outros casos, o primeiro passo para avaliar a segurança do *software* foi fazer um levantamento dos requisitos de segurança deste, para então representá-los através dos estereótipos de segurança propostos. Como no OSCommerce, as restrições do *software* são configuradas pelo administrador. Foram especificados cinco requisitos de segurança:

- **Requisito 1:** A senha do usuário deve ter no máximo 32 caracteres de comprimento.
- **Requisito 2:** Durante o *login*, o envio da senha do usuário deve ser criptografado.
- **Requisito 3:** Iniciada uma sessão, o usuário pode ficar no máximo 5 minutos sem interagir com o sistema. Após este tempo, a sessão deve expirar.
- **Requisito 4:** Para selecionar um curso, o usuário deve ser cadastrado no sistema.
- **Requisito 5:** As bases de dados devem ser protegidas de comandos SQL.

A modelagem dos requisitos descritos acima pode ser feita conforme a Tabela 12.

Tabela 12 – Mapeamento dos Requisitos de Segurança do Moodle

Requisito	Estereótipo	*Tag*
Requisito 1	«BufferOverflow»	{BOFlow={'password',32}}
Requisito 2	«Encrypt»	{CRField={password}}
Requisito 3	«Expiration»	{ExpTime={5,'m'}}
Requisito 4	«ByPassing» «ByPassing»	{BPLink={/category.php}} {BPAllowed={UsuarioCadastrado}}
Requisito 5	«SqlInjection»	{SQLField={searchField}}

A Figura 30 demonstra a inserção dos estereótipos de «BufferOverflow», «Expiration», «Encrypt», «SqlInjection» e «ByPassing», enquanto que a Figura 31 apresenta o diagrama de casos de uso identificando os atores do sistema.

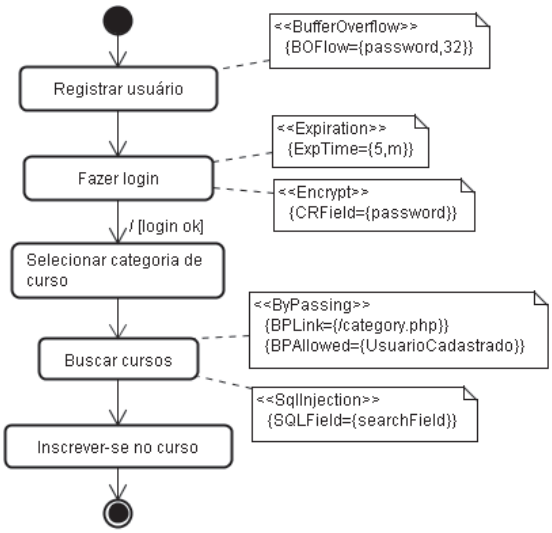

Figura 30 – Estereótipos de Segurança do Moodle

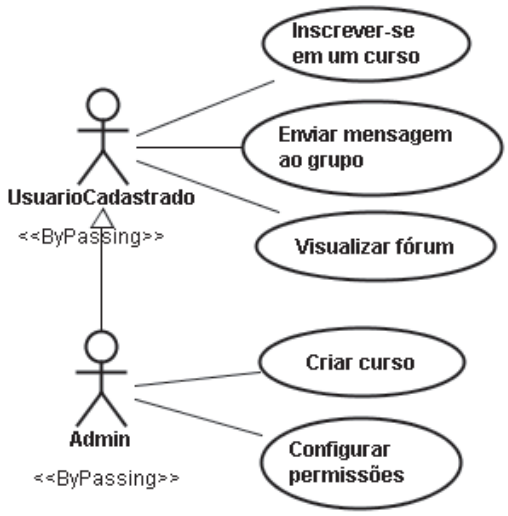

Figura 31 – Diagrama de Casos de Uso do Moodle

Assim como nos estudos de caso apresentados anteriormente, após a modelagem dos requisitos foram gerados os casos de teste referentes a cada um dos estereótipos inseridos nos

diagramas do sistema. Desta forma, os casos gerados para avaliar a segurança da aplicação Moodle podem ser vistos na Tabela 13.

Tabela 13 – Casos de Teste Gerados para o Moodle

Estereótipo	Caso de Teste
«BufferOverflow»	1. Registrar usuário. 2. Inserir no campo ‘password’ um valor com comprimento menor que 32 caracteres. 3. Resultado Esperado: valor aceito.
	1. Registrar usuário. 2. Inserir no campo ‘password’ um valor com comprimento igual a 32 caracteres. 3. Resultado Esperado: valor aceito.
	1. Registrar usuário. 2. Inserir no campo ‘password’ um valor com comprimento maior que 32 caracteres. 3. Resultado Esperado: valor rejeitado.
«Expiration»	1. Registrar usuário. 2. Fazer login como administrador. 3. Esperar durante 5 minutos sem interação com o *software*. 4. Selecionar categoria de curso. 5. Resultado Esperado: acesso negado.
«Encrypt»	1. Registrar usuário. 2. Executar analisador de tráfego. 3. Fazer login. 7. Parar execução do analisador de tráfego. 8. Resultado Esperado: o valor enviado no campo ‘password’ não pode ser visualizado.
«ByPassing»	1. Fazer login usando o nome de usuário ‘UsuarioCadastrado’. 2. Selecionar categoria de curso. 3. Buscar cursos. 4. Copiar o *link* course/index.php 5. Logoff. 6. Acessar o *link* copiado course/index.php 7. Resultado Esperado: acesso negado.
«SqlInjection»	1. Registrar usuário. 2. Fazer login. 3. Selecionar categoria de curso. 4. Buscar cursos. 5. Inserir código SQL malicioso no campo ‘searchField’. 4. Resultado Esperado: código rejeitado.

Após a execução manual dos testes descritos anteriormente, foram observados os seguintes resultados, apresentados na Tabela 14:

Tabela 14 – Resultados Obtidos com a Execução Manual dos Casos de Teste no Moodle

Estereótipo	Resultado do Teste
«BufferOverflow»	O Moodle aceita uma senha com comprimento maior que 32 caracteres apenas se a *checkbox* "mostrar" estiver marcada. Caso contrário, o *software* não permite configurar senhas cujo comprimento é maior que 32 símbolos. Entretanto, apesar da vulnerabilidade, nenhum comportamento anormal foi detectado.
«Encrypt»	O *software* criptografou corretamente as informações do usuário, não sendo vulnerável a roubos de senha através de analisadores de pacotes.
«Expiration»	A aplicação não expirou a sessão do usuário após 5 minutos de inatividade, como configurado pelo administrador.
«ByPassing»	O sistema não foi vulnerável aos ataques de *By Passing*, solicitando as informações de login do usuário quando os *links* internos foram acessados diretamente.
«SqlInjection»	O *software* não foi vulnerável a ataques de SQL *Injection*, ignorando os comandos inseridos no campo de busca.

Analisando-se os resultados obtidos, é possível concluir que o *software* contém alguns problemas de interface, como no caso de *buffer overflow*, onde ao marcar a *checkbox* "mostrar" torna-se possível adicionar senhas de comprimento maior que o permitido. Além desta vulnerabilidade, a aplicação não expirou a sessão do usuário corretamente, comprometendo a identidade deste. Se considerarmos o uso do *software* em computadores compartilhados, como por exemplo em laboratórios de uma Universidade, um aluno mal intencionado consegue facilmente fazer uso da conta de outro aluno, desde que este não tenha encerrado a sessão através do *link* "Sair". A expiração de sessão ainda é um dos métodos mais eficentes de se adicionar segurança à sistemas que utilizem esta estrutura para identificar seus usuários, pois apesar de aumentar o tráfego no servidor, é uma forma de corrigir alguns erros cometidos pelo usuário, como por exemplo esquecer de desconectar-se do sistema antes de liberar o computador para outro usuário.

7.4 CesarFTP

O CesarFTP [69] é um servidor FTP leve, gratuito, fácil de utilizar e simples de configurar. Ele suporta um sistema virtual de arquivos completo, o que preserva a estrutura do disco no servidor. Outras configurações deste *software* permitem negar o acesso a determinados usuários, limitar a velocidade de *downloads*, limitar o número de conexões ao *site* e expirar sessões. Considerando estas últimas características, o CesarFTP foi considerado um bom exemplo para

verificar a aplicabilidade do estereótipo «Flooding», não considerado nas outras aplicações, já que estas não permitiam configurar o número máximo de usuários que poderiam estar conectados ao sistema simultaneamente.

Sendo assim, após configurar o *software*, foram definidos três requisitos de segurança:

- **Requisito 1:** O número máximo de usuários conectados ao sistema simultaneamente é 5.
- **Requisito 2:** A sessão do usuário, quando inativa, deve expirar em 20 segundos.
- **Requisito 3:** O acesso a um arquivo só pode ser feito por usuários autorizados. Caso contrário o servidor deve negar o acesso do visitante.

A modelagem dos requisitos acima pode ser feita como apresentado na Tabela 15.

Tabela 15 – Mapeamento dos Requisitos de Segurança do CesarFTP

Requisito	Estereótipo	*Tag*
Requisito 1	«Flooding»	{FDMaxConn={5}}
Requisito 2	«Expiration»	{ExpTime={20,'s'}}
Requisito 3	«ByPassing» «ByPassing»	{BPLink={/arquivo1.txt}} {BPAllowed={UsuarioCadastrado}}

As Figuras 32 e 33 apresentam a modelagem do sistema e a inserção dos estereótipos referentes aos casos descritos na Tabela 15. A Figura 32 demonstra a inserção dos estereótipos de «ByPassing», «Flooding» e «Expiration», enquanto que a Figura 33 apresenta o diagrama de casos de uso identificando os atores do sistema.

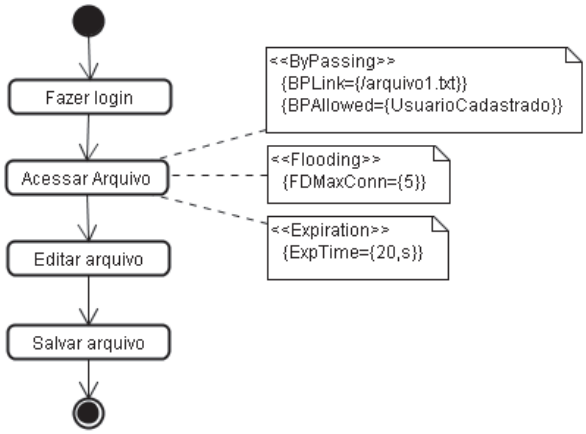

Figura 32 – Inserção dos Estereótipos «ByPassing», «Flooding» e «Expiration»

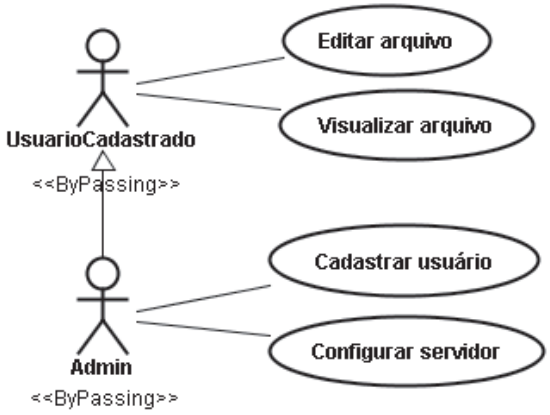

Figura 33 – Diagrama de Casos de Uso do CesarFTP

Após a modelagem dos requisitos, foi gerado o arquivo XMI, que deu origem aos casos de teste relacionados a cada estereótipo. Os casos de teste gerados para avaliar os aspectos de segurança do servidor de FTP CesarFTP podem ser vistos a seguir, na Tabela 16.

Tabela 16 – Casos de Teste Gerados para o CesarFTP

Estereótipo	Caso de Teste
«ByPassing»	1. Fazer login usando o nome de usuário 'UsuarioCadastrado'. 2. Acessar arquivo1.txt. 3. Copiar o *link* arquivo1.txt. 4. Logoff. 5. Acessar o *link* copiado arquivo1.txt. 6. Resultado Esperado: acesso negado.
«Flooding»	1. Fazer login. 2. Repetir o procedimento de login até que 4 usuários estejam logados. 3. Resultado Esperado: logins aceitos.
	1. Fazer login. 2. Repetir o procedimento de login até que 5 usuários estejam logados. 3. Resultado Esperado: logins aceitos.
	1. Fazer login. 2. Repetir o procedimento de login até que 6 usuários estejam logados. 3. Resultado Esperado: último login rejeitado.
«Expiration»	1. Fazer login 2. Esperar durante 20 segundos sem interação com o *software*. 4. Acessar arquivo. 5. Resultado Esperado: acesso negado.

A Tabela 17 apresenta os resultados obtidos após a execução manual dos casos de teste gerados:

Tabela 17 – Resultados Obtidos com a Execução Manual dos Casos de Teste no CesarFTP

Estereótipo	Resultado do Teste
«ByPassing»	O CesarFTP negou qualquer acesso de usuário não conectado ao sistema, não sendo vulnerável a ataques de *By Passing*.
«Flooding»	O *software* impediu, corretamente, a conexão de um sexto usuário, considerando que o limite imposto no *software* eram de 5 usuários.
«Expiration»	Após 20 segundos de inatividade, a aplicação expirou a sessão do usuário, funcionamento de acordo com a restrição imposta durante a configuração do servidor.

Analisando-se os resultados obtidos, é possível concluir que o servidor CesarFTP não apresentou nenhuma vulnerabilidade de segurança para os casos avaliados. Em princípio, o *software* comporta-se de forma adequada, negando o acesso de usuários além do número máximo suportado, expirando as sessões inativas e protegendo os arquivos hospedados de usuários não-autorizados. É importante ressaltar que os valores utilizados durante os testes foram baixos devido à praticidade na execução dos testes, que ainda estão sendo feitos manualmente. Para o

caso de *Flooding*, por exemplo, foi preciso instanciar seis usuários manualmente e conectá-los ao sistema, controlando o comportamento deste durante o processo de conexão do último usuário, que deveria ser impedido. Entretanto, considerando que um dos próximos passos para dar continuidade a este trabalho é promover a automatização dos testes, será possível avaliar aplicações como o CesarFTP com um número maior de usuários, permitindo inclusive determinar o máximo de usuários que o sistema comporta.

7.5 Considerações Finais

Conforme visto durante o Capítulo 7, a execução dos testes revelou uma série de vulnerabilidades. Das aplicações avaliadas, apenas o CesarFTP não foi vulnerável a nenhum dos casos executados, seguindo corretamente as configurações especificadas no servidor. Os *softwares* OSCommerce e Moodle, cujas restrições de segurança são especificadas pelo administrador do *software*, apresentaram algumas vulnerabilidades críticas. O OSCommerce, por exemplo, mesmo estando com o protocolo SSL habilitado, faz o envio das informações do usuário de forma aberta pela rede, permitindo que um usuário malicioso roube estas informações utilizando um analisador de tráfego. Já o Moodle obteve melhores resultados diante dos testes, pois não foi vulnerável aos casos considerados mais críticos. Entretanto, apresentou problemas durante o teste de expiração da sessão do usuário, que deveria ser encerrada após 5 minutos de inatividade. Por fim, o TPC-W, apesar de ser uma aplicação de *e-commerce* de teste, também apresentou vulnerabilidades, como a que permite alterar o valor de um produto sem estar conectado ao sistema como administrador. Se este *software* fosse uma aplicação real, um usuário qualquer poderia modificar o valor de um produto na hora da compra, escolhendo o valor que deseja pagar por ele.

Sendo assim, a solução para os problemas encontrados nos *softwares* é aparentemente simples, pois consiste em corrigir erros de implementação no gerenciamento de sessões, no uso de protocolos de segurança (*Secure Socket Layer*, SSL) e no controle de acesso dos usuários. Esta é outra vantagem no uso dos estereótipos de segurança no modelo, que além de permitir a geração automatizada dos casos de teste, ainda servem como guia para os desenvolvedores, destacando as partes do modelo que podem conter vulnerabilidades e que devem ser implementados cautelosamente.

8 Conclusão

Durante o ciclo de desenvolvimento de um *software*, uma das etapas é avaliar se este atende aos requisitos especificados no início do projeto. Esta análise é realizada durante a fase de teste de *software*, responsável por verificar tanto aspectos funcionais, como não-funcionais. Nesta última categoria encontra-se o teste de segurança, que vêm tornando-se importante a medida que o número de serviços prestados via Internet cresce. Entretanto, avaliar a segurança de um *software* apenas quando ele já está concluído não é eficiente. É preciso mapear estes requisitos antes de implementá-los, orientando todo o processo de desenvolvimento. Ao final deste, aplica-se o teste como uma rotina complementar, com o objetivo de verificar se os requisitos de segurança foram considerados ou não.

Apesar da simplicidade desta abordagem, colocá-la em prática é uma tarefa difícil. Poucas referências utilizam modelos UML para descrever aspectos de segurança, o que dificultou a condução desta pesquisa. Isto se deve, principalmente, ao fato de as vulnerabilidades de segurança possuírem características próprias, dificultando a elaboração de um modelo genérico que represente todas. Por este motivo, a maioria dos trabalhos relacionados está focada em um determinado aspecto, normalmente sobre políticas de controle de acesso (RBAC).

Sendo assim, o objetivo deste trabalho é disponibilizar uma forma de descrever aspectos de segurança ainda na fase de projetos, e adicioná-los ao modelo UML. A representação deste aspecto desde a fase de modelagem possui duas vantagens principais: orientar o trabalho dos desenvolvedores, destacando as partes que podem se tornar vulneráveis se mal implementadas; e possibilitar a geração automatizada de casos de teste de segurança a partir do modelo, o que permite verificar se estas vulnerabilidades realmente não afetam o *software* final.

O desenvolvimento deste trabalho foi dividido em duas etapas, uma para definição de alguns estereótipos de segurança, e outra para análise destas informações e geração dos casos de teste. Com o auxílio da taxonomia de Weber [49] e da lista de vulnerabilidades provida pelo projeto OWASP [7], estipulou-se um conjunto de estereótipos capazes de representar os casos mais comuns de segurança [63]. Apesar de alguns trabalhos proporem formas de especificar aspectos de segurança, nenhum consegue representar as vulnerabilidades aqui apresentadas. Com estes estereótipos é possível representar casos de estouro de *buffer*, necessidade de criptografia, controle no número de conexões aberta, acesso a *links* internos sem permissão, gerenciamento de sessões e prevenção a ataques de SQL *Injection*. Após a definição destes, construiu-se um algoritmo para analisar o arquivo XMI que descreve a estrutura do modelo UML do sistema, e extrair as informações relevantes deste. Este é outro diferencial deste trabalho, já que nenhum dos trabalhos relacionados modela características de segurança com o objetivo de permitir a

automatização do processo de teste. Durante a geração dos casos de teste, aplicou-se o método conhecido como UIO, que utiliza uma máquina de estados finitos com entradas e saídas para identificar os possíveis caminhos de execução. Por fim, a interpretação das seqüências geradas pelo método deu origem aos casos de teste em linguagem natural, descrevendo os passos que o testador deve executar para verificar a existência da vulnerabilidade ou não.

Entretanto, é importante ressaltar que nem sempre a execução de um caso básico é suficiente para atacar um sistema ou revelar uma vulnerabilidade. As vezes é preciso elaborar, ou mesmo unir ataques, para realizar o ataque com sucesso. Sendo assim, um dos trabalhos futuros relacionados a esta pesquisa refere-se à geração de casos de teste mais complexos, juntamente com a representação dos demais casos apresentados pela lista de vulnerabilidades do projeto OWASP, como XSS, Execução Maliciosa de Arquivos e Armazenamento Inseguro de Informações. Outro aspecto que se deseja incluir no trabalho é a geração de *scripts* de teste no formato de alguma ferramenta, ao invés de ser em linguagem natural. Desta forma é possível automatizar todo o processo de teste, desde a fase de geração a partir do modelo, até a parte de execução do teste, utilizando ferramentas como o JUnit [71], o JMeter [52] e o WebLOAD [72].

Por último, é preciso mencionar que este trabalho não foi conduzido individualmente. Ele está integrado a uma linha de produto de teste, descrita por Orozco *et al.* [73], cujo objetivo é gerar ferramentas capazes de aplicar testes específicos em um *software*, de acordo com a necessidade do usuário. Nesta ferramenta, o usuário pode escolher se quer executar apenas testes funcionais em seu *software*, ou se pretende associá-lo a algum outro tipo de teste, como segurança (apresentado neste trabalho) ou desempenho (apresentado por Rodrigues *et al.* [74]). Após concluir a etapa de configuração, a linha de produto de teste produzirá como resultado uma segunda ferramenta, contendo apenas as funcionalidades escolhidas pelo usuário e os módulos necessários para executar os tipos de teste escolhidos. Neste contexto, a ferramenta de teste de segurança descrita neste trabalho é um dos módulos da linha de produto de teste, colaborando com as atividades de avaliação do modelo UML, extração das informações relevantes e geração de casos de teste em linguagem natural. Posteriormente, com a construção de um processo de geração de *scripts*, será possível adicionar mais um produto à linha, permitindo que o usuário escolha se quer gerar os casos em linguagem natural ou como entrada para alguma ferramenta de automação de teste.

Referências

[1] YOSHIHAMA S., URAMOTO, N., MAKINO, S., ISHIDA, A., KAWANAKA, S. e KEUKELAERE, F. D. *"Security Model for the Client-Side Web Application Environments"*. In: Workshop Web 2.0 Security and Privacy, 2007. Disponível em <http://w2spconf.com/2007/papers/paper-159-z_7167.pdf>. Acesso em dezembro de 2008.

[2] BLACKBURN, M., CHANDRAMOULI, R., BUSSER, R. e NAUMAN, A. *"Model-based Approach to Security Test Automation"*. In: 13th International Symposium on Software Reliability Engineering, Industry Track, 2002. Disponível em <http://csrc.nist.gov/groups/SNS/asft/documents/Issre_2002.pdf> Acesso em dezembro de 2008.

[3] McDERMID, J. e SHI, Q. *"A Formal models of Security Dependency for Analysis and Testing of Secure Systems"*. In: Proceedings of Computer Security Foundations Workshop IV, páginas 188 - 200, 1991.

[4] MARICK, B. *"New Models for Test Development"*. In: 12th International Software Quality Week, Keynote Talk, 1999. Disponível em <http://www.exampler.com/testing-com/writings/new-models.pdf> Acesso em dezembro de 2008.

[5] HOLLOWAY, C. *"Why Engineers should consider Formal Methods"*. In: Proceedings of the 16th AIAA/IEEE Digital Avionics Systems Conference, vol. 1, páginas 16 - 22, 1997.

[6] JANZEN, D. e SAIEDIAN, H. *"Test-Driven Development: Concepts, Taxonomy, and Future Direction"*. In: Computer Magazine, vol. 8, nº 9, páginas 43 - 50. IEEE Computer Security Press, 2005.

[7] The Open Web Application Security Project. *"The Ten most Critical Web Application Security Vulnerabilities"*. Disponível em <http://www.owasp.org>. Acesso em dezembro de 2008.

[8] DELAMARO, M., MALDONADO, J. C. e JINO, M. *"Introdução ao Teste de Software"*. Editora Elsevir, São Paulo, 2007.

[9] HEITMEYER, C. *"On the need for Practical Formal Methods"*. In: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, páginas 18 - 26, 1998.

[10] BOWEN, J. P. e HINCHEY, M. G. *"Ten Commandments of Formal Methods"*. In: ACM Digital Library, vol. 28, nº 4, páginas 56 - 63. IEEE Computer Society Press, 1995.

[11] AVIZIENIS, A., LAPRIE, J.C., RANDELL, B. e LANDWEHR, C. *"Basic Concepts and Taxonomy of Dependable and Secure Computing"*. In: IEEE Transactions on Dependable and Secure Computing, vol. 1, nº 1, páginas 11 - 33, IEEE Computer Society Press, 2004.

[12] HAILPERN, B. e SANTHANAM, P. *"Software debugging, testing and verification"*. In: IBM Systems Journal, vol. 41, nº 1, páginas 4 - 12, Allen Press, Inc, 2002.

[13] KAPFHAMMER, G. M. *"Software Testing"*. The Computer Science and Engineering Handbook, CRC Press, 2004.

[14] KHENDEK, F. e BOCHMANN, G. *"Formal Specifications Design, Evolution and Reuse"*. In: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: Software Engineering, páginas 184 - 193, 1993.

[15] BURNSTEIN, I. *"Practical Software Testing: a Process-oriented Approach"*. Editora Springer, 2002.

[16] STOCKS, P. A. *"Applying Formal Methods to Software Testing"*. Tese (Doutorado) - Universidade de Queesland, 1993.

[17] JORGENSEN, P. C. *"Software Testing - A Craftsman's Approach (2nd Edition)"*. Editora CRC Press LLC, 2002.

[18] POPOVIC, M. e VELIKIC, I. *"A Generic Model-Based Test Case Generator"*. In: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, páginas 221 - 228, 2005.

[19] *"Testing and Test Control Notation version 3"*. Disponível em <http://www.ttcn-3.org/WhatisT3.html>. Acesso em dezembro de 2008.

[20] *"European Telecommunication Standards Institute (ETSI)"*. Disponível em <http://www.etsi.org>. Acesso em dezembro de 2008.

[21] SABIGUERO, A., BAIRE, A., DESMOULIN, A., FLOCH, A., ROUDAUT, F. e VIHO, C. *"Towards an IP-oriented Testing Framework: the IPv6 Testing Toolkit"*. In: 3rd TTCN-3 User Conference, 2006. Disponível em <http://www.irisa.fr/armor/lesmembres/Desmoulin/TowardsIPv6TestingFramework.pdf>. Acesso em dezembro de 2008.

[22] STEPIEN, B. *"TTCN-3 in a Nutshell - TTCN-3 Tutorial"*. Disponível em <http://site.uottawa.ca/ bernard/ttcn.html>. Acesso em dezembro de 2008.

[23] OLIVEIRA, F. M. e COPSTEIN, B. *"STAGE: an Integrated Environment for Statistical Test Script Generation"*. In: Anais do V Workshop de Testes e Tolerância a Falhas, páginas 77 - 78, 2004.

[24] BERTOLINI, C., FERNANDES, P. e OLIVEIRA, F. M. *"Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis"*. In: Proceedings of the Second International Conference on Software Engineering and Formal Methods, páginas 251 - 260, 2004.

[25] *"Padrão ISO/IEC 13568:2002 - Z Formal Specification Notation"*. Disponível em <http://www.iso.org>. Acesso em dezembro de 2008.

[26] BOWEN, J. P. *"Formal Specification and Documentation using Z: A Case Study Approach"*. International Thomson Computer Press, 1996.

[27] SPIVEY, J. M. *"The Z Notation: A Reference Manual (2nd Edition)"*. Prentice Hall International, 1998.

[28] VILKOMIR, S. A. e BOWEN, J. P. *"Formalization of Software Testing Criteria using the Z Notation"*. In: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, páginas 351 - 356, ACM Press, 2001.

[29] LAMPORT, L. *"The Temporal Logic of Actions"*. In: ACM Transactions on Programming Languages and Systems, vol. 16, nº 3, páginas 872 - 923, 1994.

[30] TAN, L., SOKOLSKY, O. e LEE, I. *"Specification-Based Testing with Linear Temporal Logic"*. In: Proceedings of the IEEE International Conference on Information Reuse and Integration, páginas 483 - 498, 2004.

[31] NALDURG, P., SEN, K. e THATI, P. *"A Temporal Logic Based Framework for Intrusion Detection"*. In: Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems, LNCS 3235, páginas 359 - 376, Springer Berlin/Heidelberg, 2004.

[32] *"Unified Modeling Language (UML)"*. Disponível em <http://www.omg.org/gettingstarted/what_is_uml.htm>. Acesso em dezembro de 2008.

[33] *"The Object Management Group (OMG)"*. Disponível em <http://www.omg.org>. Acesso em dezembro de 2008.

[34] BOUQUET, F., GRANDPIERRE, C., LEGEARD, B., PEUREUX, F., VACELET, N. e UTTING, M. *"A Subset of Precise UML for Model-based Testing"*. In: Proceedings of the 3rd International Workshop on Advances in Model-based Testing, páginas 95 - 104, 2004.

[35] *"Object Constraint Language Specification (OCL)"*. In: OMG Unified Modeling Language Specification, versão 1.3, Junho 1999.

[36] Object Management Group. *"UML 2.0 Testing Profile Specification"*. 2004.

[37] BIASI, L. e BECKER, K. *"Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP"*. In: Anais do 10º Simpósio Brasileiro de Engenharia de Software, páginas 33 - 48, 2006.

[38] JÜRJENS, J. *"UMLsec: Extending UML for Secure Systems Development"*. In: Proceedings of the 5th International Conference on the Unified Modeling Language, páginas 412 - 425, 2002.

[39] LODDERSTEDT, T., BASIN, D. A. Basin e DOSER, J. *"SecureUML: A UML-based Modeling Language for Model-Driven Security"*. In: Proceedings of the 5th International Conference on the Unified Modeling Language, páginas 426 - 441, 2002.

[40] RAY, I., LI, N., KIM, D. e FRANCE, R. *"Using UML to Visualize Role-based Access Control Constraints"*. In: Proceedings of the 9th ACM Symposium on Access control Models and Technologies, páginas 115 - 124, 2004.

[41] LANDWEHR, C. E., BULL, A. R., McDERMOTT, J. P. e CHOI, W. S. *"A Taxonomy of Computer Program Security Flaws, with Examples"*. In: ACM Computer Surveys, vol. 26, nº 3, ACM Press, 1994.

[42] SYVERSON, P. *"A Taxonomy of Replay Attacks"*. In: Proceedings of 7th IEEE Computer Security Foundations Workshop, páginas 187 - 191, 1994.

[43] WUN, A., CHEUNG, A. e JACOBSEN, H. A. *"A taxonomy for denial of service attacks in content-based publish/subscribe systems"*. In: Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems, páginas 116 - 127, 2007

[44] UNDERCOFFER, J. L. *"Intrusion Detection: Modeling System State to Detect and Classify Aberrant Behavior"*. Tese (Doutorado) - Universidade de Maryland, 2004.

[45] LIPPMANN, R. P., FRIED, D. J., GRAF, I., HAINES, J. W., KENDALL, K. R., McCLUNG, D., WEBER, D., WEBSTER, S. E., WYSCHOGROD, D., CUNNINGHAM, R. K. e ZISSMAN, M. A. *"Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation"*. In: Proceedings of DARPA Information Survivability Conference and Exposition, vol. 2, páginas 12 - 26, 2000.

[46] ASLAM, T. *"A Taxonomy of Security Faults in the Unix Operating Systems"*. Dissertação (Mestrado), Universidade de Purdue, 1995.

[47] ASLAM, T., KRSUL, I. e SPAFFORD, E. H. *"Use of a Taxonomy of Security Faults"*. In: Proceedings of the 19th National Information Systems Security Conference, páginas 551 - 560, 1996.

[48] BAZAZ, A., ARTHUR, J. D. e TRONT, J. G. *"Modeling Security Vulnerabilities: A Constraints and Assumptions Perspective"*. In: Proceedings of the 2nd IEEE International Sysmposium on Dependable, Autonomic and Secure Computing, páginas 95 - 102, 2006.

[49] WEBER, S., KARGER, P. A. e PARADKAR, A. *"A Software Flaw Taxonomy: Aiming Tools at Security"*. In: Proceedings of the 2005 Workshop on Software Engineering for Secure Systems, páginas 1 - 7, 2005.

[50] BOSWORTH, S. e KABAY, M. E. *"Computer Security Handbook (4th Edition)"*. J. Wiley & Sons, Inc., 2002.

[51] Sun Microsystems, Inc. *"Enterprise JavaBeans Specification, Version 3.0, 2006"*. Disponível em <http://java.sun.com/ejb/docs.html. Acesso em dezembro de 2008.

[52] The Apache Jakarta Project. *"Apache JMeter"*. Disponível em <http://jakarta.apache.org/jmeter>. Acesso em dezembro de 2008.

[53] *"Rational XDE Tester"*. Disponível em <http://www.ibm.com/developerworks/lotus/library/lsxde-tester>. Acesso em dezembro de 2008.

[54] *"HttpUnit"*. Disponível em <http://httpunit.sourceforge.net/>. Acesso em dezembro de 2008.

[55] *"HP QAInspect Software"*. Disponível em <http://www.hp.com>. Acesso em dezembro de 2008.

[56] *"Lei Sarbanes-Oxley"*. Lei Federal Norte-americana, assinada no 117º Congresso dos Estados Unidos da América, Estatuto 745, em 30 de Julho de 2002. Disponível em <http://frwebgate.access.gpo.gov/cgi-bin/getdoc.cgi?dbname=107_cong_bills&docid=f:h3763enr.tst.pdf>. Acesso em dezembro de 2008.

[57] *"Padrão ISO/IEC 17799:2005 - Code of practice for information security management"*. Disponível em <http://www.iso.org>. Acesso em dezembro de 2008.

[58] *"Payment Card Industry (PCI) Data Security Standard - Requirements and Security Assessment Procedures"*. Disponível em <https://www.pcisecuritystandards.org>. Acesso em dezembro de 2008.

[59] AHN, G. e SANDHU, R. *"Role-based Authorization Constraints Specification using Object Constraint Language"*. In: Proceedings of the 10th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, páginas 157 - 162, 2001.

[60] National Institute of Standards and Technologies. *"National Vulnerability Database"*. Disponível em <http://nvd.nist.gov>. Acesso em dezembro de 2008.

[61] Sans Institute. *"SANS Top 20 List"*. Disponível em <http://www.sans.org/top20>. Acesso em dezembro de 2008.

[62] Security Focus Project. *"Security Focus Vulnerability List"*. Disponível em <http://www.securityfocus.com/vulnerabilities>. Acesso em dezembro de 2008.

[63] PERALTA, K., OROZCO, A., ZORZO, A. e OLIVEIRA, F.M. *"Specifying Security Aspects in UML Models"*. In: Workshop on Modeling Security, 2008. Disponível em <http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-413/paper11.pdf>. Acesso em dezembro de 2008.

[64] MEALY, G. H. *"A Method to Synthesizing Sequential Circuits"*. Bell System Technical Journal, vol. 34, nº 5, páginas 1045 - 1079, Bell Labs, 1955.

[65] DERDERIAN, K., HIERONS, R. M., HARMAN, M. e GUO, Q. *"Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs"*. The Computer Journal, vol. 49, nº 3, páginas 331 - 344, Oxford University Press, 2006.

[66] Open Source E-Commerce Solutions. *"OSCommerce"*. Disponível em <http://www.oscommerce.com>. Acesso em dezembro de 2008.

[67] *"TPC Benchmark-W (TPC-W)"*. Disponível em <http://www.tpc.org/tpcw>. Acesso em dezembro de 2008.

[68] *"Moodle"*. Disponível em <http://www.moodle.org>. Acesso em dezembro de 2008.

[69] *"CesarFTP 0.99g"*. Disponível em <http://www.aclogic.com>. Acesso em dezembro de 2008.

[70] SOPITKAMOL, M e MENASCÈ, D. *"A Method for Evaluating the Impact of Software Configuration Parameters on e-commerce Sites"*. In: Proceedings of the 5th International Workshop on Software Performance, páginas 53 - 64, 2005.

[71] *"JUnit.org"*. Disponível em <http://www.junit.org>. Acesso em dezembro de 2008.

[72] *"WebLOAD Open Source Load Testing"*. Disponível em <http://www.webload.org>. Acesso em dezembro de 2008.

[73] OROZCO, A. *"Linha de Produtos de Testes Baseados em Modelos"*. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio Grande do Sul, 2008.

[74] RODRIGUES, E. *"Realocação de Recursos em Ambientes Virtualizados"*. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio Grande do Sul, 2008.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo