Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Uma Abordagem para Avaliação da Eficácia de *Scanners* de Vulnerabilidades em Aplicações *Web*

Tania Basso

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Orientador: **Prof. Dr. Mario Jino**
Co-orientadora**: Prof. Dr. Regina L. O. Moraes**

Banca Examinadora:

Prof. Dr. Ivan Ricarte
Faculdade de Engenharia Elétrica e de Computação – UNICAMP

Prof. Dr. Ana Cristina Vieira de Melo
Instituto de Matemática e Estatística – USP

Campinas, SP
Agosto/2010

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

Basso, Tania
B295a Uma abordagem para avaliação da eficácia de scanners de vulnerabilidades em aplicações web / Tania Basso. --Campinas, SP: [s.n.], 2010.

Orientadores: Mario Jino, Regina Lúcia de Oliveira Moraes.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Scanner ótico. I. Jino, Mario. II. Moraes, Regina Lúcia de Oliveira. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Título em Inglês: An approach for evaluation of efficacy of vulnerability scanning tools in web applications
Palavras-chave em Inglês: Optical scanner
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Ivan Ricarte, Ana Cristina Vieira de Melo
Data da defesa: 25/08/2010
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidata: Tânia Basso

Data da Defesa: 25 de agosto de 2010

Título da Tese: "Uma Abordagem para Avaliação da Eficácia de Scanners de Vulnerabilidades em Aplicações WEB"

Prof. Dr. Mario Jino (Presidente): _____

Profa. Dra. Ana Cristina Vieira de Melo: _____

Prof. Dr. Ivan Luiz Marques Ricarte: _____

Resumo

Grande parte das aplicações Web é desenvolvida atualmente sob severas restrições de tempo e custo. A complexidade dos produtos de software é cada vez maior resultando em vulnerabilidades de segurança produzidas por má codificação. Ferramentas chamadas scanners de vulnerabilidade são utilizadas para auxiliar a detecção automática de vulnerabilidades de segurança em aplicações Web; portanto, poder confiar nos resultados da aplicação dessas ferramentas é essencial. Este trabalho propõe uma abordagem para avaliar a eficácia desses scanners. A abordagem proposta está baseada em técnicas de injeção de falhas e modelos de árvores de ataque; os resultados da aplicação de três scanners são avaliados na presença de falhas realistas de software responsáveis por vulnerabilidades de segurança em aplicações Web. As árvores de ataque representam os passos para se realizar um ataque, permitindo verificar se vulnerabilidades detectadas pelo scanner existem de fato na aplicação sob teste. A abordagem também pode ser utilizada para realizar testes de segurança, pois permite a detecção de vulnerabilidades pela execução de cenários de ataque.

Abstract

Nowadays, most web applications are developed under strict time and cost constraints. The complexity of software products is increasingly bigger leading to security vulnerabilities due to bad coding. Tools called vulnerability scanners are being applied to automatically detect security vulnerabilities in web applications; thus, trustworthiness of the results of application of these tools is essential. The present work proposes an approach to assess the efficacy of vulnerability scanner tools. The proposed approach is based on fault injection techniques and attack tree models; the results of the application of three scanners are assessed in the presence of realistic software faults responsible for security vulnerabilities in web applications. Attack trees represent the steps of performing an attack, allowing verifying whether security vulnerabilities detected by the scanner tool do exist in the application under test. The approach can also be used to perform security tests, as it permits the detection of vulnerabilities through the execution of attack scenarios.

Agradecimentos

A Deus, pelas bênçãos e oportunidades que me permitiram realizar este trabalho.

À minha co-orientadora Profa. Dra. Regina Lúcia de Oliveira Moraes, por acreditar em meu esforço e trabalho. Agradeço também por ser mais que uma co-orientadora – papel que aliás desempenhou de uma forma excepcional – mas por ter feito tanto por mim durante esta caminhada. Agradeço pela sua paciência e disponibilidade, pelo incentivo, pelos conselhos, pelo auxílio, pela companhia e amizade. A ela devo muito, muito mais do que ela possa imaginar.

Ao meu orientador Prof. Mario Jino, por me receber, por permitir que eu realizasse este sonhado trabalho. Agradeço não só pela oportunidade e pela valiosa orientação, mas também pela sua bondade e generosidade.

Ao meu esposo Leandro Piardi pelo apoio e compreensão nos momentos em que se viu privado de minha companhia. Agradeço pelo amor, amizade, companheirismo e pela ajuda nos momentos mais desgastantes e difíceis, os quais sempre me motivou a superar.

À minha mãe e familiares, por me darem estímulo ao longo do meu trabalho.

Ao amigo Plínio Cesar Simões Fernandes por me ceder parte de seu tempo e conhecimento para construção dos ambientes de testes e definição das formas de ataque. Sua ajuda foi fundamental para o desenvolvimento desta dissertação.

Ao amigo Bruno Pacheco Sanches, por me auxiliar na investigação de falhas de software em aplicações Java e construção do *faultload* utilizado nesta dissertação.

Aos amigos Danilo Andrade, Caio Santos e Ariane por trabalharem nas pesquisas sobre vulnerabilidades e ferramentas, que me auxiliaram na decisão sobre quais ferramentas utilizar nesse trabalho.

Aos demais amigos, que muito contribuíram com seus incentivos, e por terem proporcionado alguns momentos divertidos no decorrer desse trabalho.

À Universidade Estadual de Campinas (UNICAMP), que proporcionou todas as oportunidades para o meu desenvolvimento intelectual e profissional.

Ao projeto Harpia, pelo apoio financeiro e pela oportunidade de aprender de maneira prática como realizar as atividades de teste de software.

A CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, pelo apoio financeiro através de bolsa de estudo.

E, por fim, agradeço às pessoas que me desafiaram e que muitas vezes me decepcionaram, pois assim me ensinaram a me recuperar de cada tropeço na estrada da vida.

Índice

Capítulo 1 ..... **1**

Introdução **1**

1.1 Terminologia *2*

1.2 Scanners de vulnerabilidade *4*

1.3 Objetivos *6*

1.4 Abordagem Proposta *7*

1.5 Contribuições do trabalho *8*

1.6 Organização do texto *9*

Capítulo 2 ..... **11**

Trabalhos Relacionados **11**

2.1 Definição do faultload *11*

2.2 Análise da eficácia de scanners de vulnerabilidade *15*

2.3 Modelos de ataques e vulnerabilidades de segurança *17*

2.4 Resumo *20*

Capítulo 3 ..... **23**

Segurança de Sistemas Computacionais **23**

3.1 Segurança de Redes, Servidores e Protocolos de Comunicação *24*

3.2 Segurança de aplicações Web *26*

3.3 Resumo *31*

Capítulo 4 ..... **33**

Injeção de Falhas e Modelos de Ataques **33**

4.1 Injeção de falhas *33*

4.2 Modelos de Ataques *41*

4.3 Resumo *47*

Capítulo 5 ..... **49**

Abordagem, Experimentos e Resultados **49**

5.1 Descrição da Abordagem *49*

5.2 Estudo de caso *57*

5.3 Resultados *59*

5.4 Automatização *68*

5.5 Resumo *71*

Capítulo 6 ..... **73**

Conclusão **73**

Referências **77**

Apêndice A **85**

Apêndice B **101**

Apêndice C **111**

Índice de Tabelas

Tabela 2.1. Tipos de falhas mais frequentemente observadas em aplicações C. 13

Tabela 5.1. Falhas de segurança em Java mais frequentemente observadas..... 50

Tabela 5.2. Cenários e vulnerabilidades 60

Tabela 5.3. Tipos de falhas que afetaram resultados dos scanners..... 60

Tabela 5.4. Falta de cobertura e falsos positivos referentes às aplicações..... 62

Tabela 5.5. Falta de cobertura e falsos positivos referentes aos tipos de vulnerabilidades. 63

Tabela 5.6. Porcentagem de falta de cobertura e falsos positivos 64

Tabela 5.7. Falta de cobertura e falsos positivos referentes aos scanners. 66

Tabela 5.8. Falta de cobertura e falsos positivos referentes à aplicação AppX. 69

Tabela 5.9. Falta de cobertura e falsos positivos referentes aos tipos de vulnerabilidades. 69

Tabela 5.10. Falta de cobertura e falsos positivos referentes aos scanners. 71

Índice de Figuras

Figura 4.1. Ambiente básico de injeção de falhas 38

Figura 4.2. Representação gráfica e textual de relação do tipo AND entre nós da árvore. 42

Figura 4.3. Representação gráfica e textual de relação do tipo OR entre nós da árvore. 43

Figura 4.4. Árvore de ataques para abrir o cofre 44

Figura 4.5. Possíveis cenários de ataques selecionados a partir da árvore de ataques para abrir o cofre. 45

Figura 4.6. Padrão de ataque de buffer overflow 46

Figura 5.1. Árvore de ataques para vulnerabilidades CSRF 53

Figura 5.2. Exemplo de injeção de falhas do tipo MIFS. (a) Código fonte da aplicação original. (b) Código fonte com a falha injetada. 55

Figura 5.3. Procedimento para avaliação dos resultados do scanner de vulnerabilidade 56

Figura 5.4. Relação entre as detecções dos tipos de vulnerabilidades pelos scanners. 66

Figura 5.5. Relação entre as detecções dos tipos de vulnerabilidades pelos scanners. 70

1 Capítulo 1

Introdução

A sociedade moderna está cada vez mais dependente dos serviços prestados pelos computadores, e, consequentemente, dependente do software que está sendo executado para prover estes serviços. Assim, o bem estar e a segurança da sociedade estão baseados no pressuposto de que os sistemas de computadores apresentem alto nível de dependabilidade. O termo dependabilidade, do inglês *dependability*, é definido como a habilidade de um sistema prover serviços que, justificadamente, possam ser confiáveis, ou ainda, segundo Avizienis *et al.* [1], é a capacidade de evitar que ocorram defeitos frequentes ou com severidade maior que o esperado pelos usuários. É um conceito integrado que abrange os seguintes atributos: disponibilidade (prontidão para prover o serviço correto); confiabilidade (continuidade do serviço correto); inocuidade (do inglês, *safety*: a ausência de consequências catastróficas para os usuários ou ambiente); integridade (ausência de alterações indevidas); manutenibilidade (capacidade de sofrer modificações e reparos). No entanto, em muitos domínios de aplicação, a ocorrência de defeitos origina situações potencialmente sérias, especialmente em sistemas críticos, onde um defeito pode ser responsável pela perda de bens, grandes prejuízos financeiros, desastres ambientais e perda de vidas humanas. O longo histórico de casos documentados de consequências da ocorrência de defeitos de software demonstra que até mesmo os sistemas críticos exaustivamente testados são falíveis.

A ocorrência de defeitos em sistemas baseados em computadores em fase operacional continua sendo um acontecimento comum, apesar do enorme esforço despendido no desenvolvimento de sistemas capazes de tolerar falhas e do empenho na manutenção desses sistemas. Esses defeitos ocorrem devido a falhas residuais introduzidas no código. Essas falhas afetam o nível de dependabilidade do sistema de software e podem até mesmo ser responsáveis por tornar o software vulnerável do ponto de vista de segurança, principalmente quando se trata de aplicações *Web*.

O principal fator que contribui para que haja falhas residuais é a má codificação, que pode ter como principais causas as seguintes: complexidade crescente dos sistemas (sistemas mais complexos têm maior probabilidade de apresentar falhas residuais); pressão para reduzir o tempo de comercialização (leva à redução do tempo de desenvolvimento especialmente nas fases de teste de qualidade); utilização de componentes genéricos (os componentes genéricos não foram desenvolvidos especificamente para o cenário operacional em que são integrados e podem conter falhas ocultas que podem ser reveladas nestes novos cenários) e reutilização de código (o novo

ambiente operacional pode diferir bastante do anterior para o qual o software foi inicialmente previsto) [2].

Além disso, falhas residuais podem ser responsáveis pela segurança dos sistemas de software, pois estas falhas podem ser eventuais responsáveis por vulnerabilidades de segurança que, normalmente, são exploradas por pessoas externas, que têm interesse de invadir o sistema (*hackers*) e executar ações contrárias aos objetivos previstos nos requisitos [3]. Diante da atual demanda por computação ubíqua e a extrema popularidade das aplicações *Web,* um dos principais desafios neste contexto é manter a dependabilidade destes sistemas. Essas aplicações podem ser utilizadas por pessoas que estão geograficamente distribuídas e podem utilizar o sistema em qualquer lugar, usando qualquer computador que queiram, fazendo com que o impacto de falhas seja ainda maior, pois não se sabe a quem solicitar a correção dessas falhas.

Sabe-se que mecanismos tradicionais de segurança de redes como *firewalls*, criptografia e sistemas de detecção de intrusão podem proteger a rede, mas não mitigam ataques a aplicações *Web*. Dessa forma, *hackers* estão mudando seus focos da rede para essas aplicações, onde a codificação não segura representa maior risco [3]. Embora existam outras formas para potenciais causas de vulnerabilidades, o sucesso da maioria dos ataques é em decorrência da exploração de vulnerabilidades criadas por falhas de software [3][4].

Ferramentas chamadas scanners de vulnerabilidades são utilizadas por desenvolvedores, testadores e administradores de sistemas. Essas ferramentas analisam, de forma automática, aplicações Web em busca de vulnerabilidades de segurança. Confiar nos resultados que esse tipo de ferramenta provê é importante, pois fica a dúvida: vulnerabilidades não detectadas pelo scanner realmente não existem na aplicação sob teste? Então, a relevância dos resultados dessas ferramentas deve ser avaliada. Essa avaliação é feita a partir de dois aspectos: falta de cobertura (quando o scanner deixa de detectar vulnerabilidades que realmente existem na aplicação) e falsos positivos (o scanner detecta vulnerabilidades que não existem na aplicação). Falsos positivos implicam desperdício de tempo para correções que de fato não são necessárias.

A proposta do presente trabalho é estabelecer uma abordagem que permita avaliar a eficácia de scanners de vulnerabilidade e com isso permitir que diretrizes sejam fornecidas para aquisição ou utilização dessas ferramentas. A abordagem é baseada em técnicas de injeção de falhas e modelagem de árvores de ataques.

1.1 Terminologia

Falha, erro e defeito são termos bastante conhecidos nas áreas de teste e tolerância a falhas. No entanto, no Brasil, há uma divergência entre estas áreas, que utilizam os termos de formas diferentes, inclusive na literatura. Na área de testes normalmente utilizam-se os termos de acordo

com o padrão IEEE, número 610.12-1990 [5], onde, resumidamente, um defeito (*bug*) é um problema no programa, como por exemplo, uma instrução ou comando incorreto. Uma falha (*failure*) acontece quando uma saída incorreta é produzida com relação à especificação. Um erro (*error*) é uma diferença entre o valor correto e o valor esperado na execução do programa.

Na área de Tolerância a Falhas adotam-se, para os termos falha, erro e defeito, as seguintes definições (baseadas em Leite e Orlando[6] e Laprie [7]).

Falha (*fault*): causa suposta ou constatada de um erro do sistema.

Erro (*error*): uma falha ativada dentro de um dado processamento do sistema pode levar a um estado incorreto do sistema. Esse estado incorreto causado pela ativação da falha é denominado erro.

Defeito (*failure*): um erro provocado por uma falha, por sua vez, poderá fazer com que o sistema apresente um defeito, definido como um desvio da especificação do serviço.

Optou-se, neste trabalho, por adotar os termos utilizados na área de Tolerância a Falhas. Os demais termos necessários para o entendimento deste trabalho são definidos a seguir:

Verificação: é o processo de se avaliar um software a cada fase de sua construção para determinar se este software está sendo desenvolvido da maneira correta, ou seja, conforme o planejado.

Validação: a validação é um processo no qual se avalia um software durante ou após o desenvolvimento, para determinar se o produto satisfaz os seus requisitos.

Computação ubíqua: termo utilizado para descrever a onipresença dos recursos de computação no cotidiano das pessoas, onde dispositivos são conectados à rede com o fim de fornecer a informação ou serviços que os usuários requerem em qualquer lugar e a qualquer momento.

Benchmark: Um padrão pelo qual algo pode ser medido ou julgado, ou seja, uma referência ou um instrumento fixo que permite comparar um novo registro ("*mark*", uma marca) a um padrão preestabelecido, a partir de um ponto de observação ("*bench*", o banco onde os antigos exploradores ficavam sentados observando o fato acontecer).

Tolerância a falhas: capacidade do sistema de continuar a prover o serviço esperado mesmo na presença de falhas

Injeção de falhas: técnica de validação que usa a introdução deliberada de falha (ou erro) no sistema e acelera sua ativação para observar o comportamento do sistema na presença desta falha (ou erro).

***Faultload*:** especifica um conjunto de falhas que vão ser injetadas no sistema para simular a existência de falhas.

***Workload*:** é a carga de trabalho do sistema, ou seja, a tarefa ou conjunto de tarefas que são submetidas ao sistema e devem ser executadas por ele.

Representatividade das falhas: frequência com que estas falhas são identificadas em uma determinada classe de sistema.

Vulnerabilidade de segurança: é uma falha, uma brecha em um produto de software, que possibilita que um usuário mal intencionado (atacante) opere o sistema de forma indevida para obter privilégios, controlar as operações, comprometer dados do sistema e do usuário, entre outros objetivos.

Ataque: ação (ões) realizada(s) por usuário mal intencionado sobre uma aplicação alvo com o objetivo de explorar vulnerabilidades de segurança e operar o sistema de forma indevida.

***Scanner* de vulnerabilidade**: ferramenta que analisa automaticamente aplicações *Web* em busca de vulnerabilidades de segurança.

Falso positivo: quando o resultado da análise indica vulnerabilidade de segurança que efetivamente não existe na aplicação.

***Patch* de correção**: trechos de código disponibilizados para corrigir defeitos de determinada aplicação.

Árvore de ataques: estrutura de dados que descreve os possíveis ataques a um sistema de uma forma bem organizada para facilitar a análise de segurança. Representa os passos de um ataque e suas interdependências.

Cenários de ataque: conjunto de nós formado por um caminho percorrido em profundidade na árvore de ataques (caminho do nó raiz até o nó folha da árvore).

Oráculo: mecanismo capaz de determinar se o resultado de um teste ou experimento está ou não de acordo com os valores esperados.

1.2 Scanners de vulnerabilidade

Os scanners de vulnerabilidade são ferramentas utilizadas para testar aplicações em busca de vulnerabilidades de segurança. Eles são capazes de identificar automaticamente vulnerabilidades de segurança introduzidas no código e podem ser utilizados ao longo de todo o ciclo de vida do desenvolvimento do software. Isto envolve profissionais de segurança e auditoria, testadores e desenvolvedores.

Normalmente, estas ferramentas funcionam em três estágios: configuração, rastreamento e exploração. O estágio de configuração inclui a definição do endereço (URL - *Uniform Resource Locator*) da aplicação Web e a definição de demais parâmetros necessários, como, por exemplo, a utilização de *proxy* (servidor intermediário entre o cliente e a internet) ou certificados de segurança. Alguns scanners também apresentam, nesse estágio, recursos para autenticação (*login*) na

aplicação. É possível gravar macros contendo o nome do usuário e sua senha para que a ferramenta repita essa sequência durante o processo de exploração de forma que os formulários sejam preenchidos automaticamente e permitam o acesso em áreas restritas e testes sejam desenvolvidos nessas áreas.

No estágio de rastreamento, os scanners produzem um mapa da estrutura interna da aplicação web. Este estágio é importante porque a não identificação de alguma página da aplicação pode excluí-la dos testes nos estágios subsequentes. O scanner chama a primeira página web e então examina seu código procurando por *links*. Cada *link* encontrado é registrado e este procedimento é repetido várias vezes até que *links* e páginas não sejam mais encontrados.

O estágio de exploração é onde testes de penetração são desenvolvidos automaticamente contra a aplicação web simulando ações de um usuário (cliques em links e preenchimento de dados maliciosos em campos de entrada). Durante esse estágio, vários testes são executados de acordo com o algoritmo interno da ferramenta (não disponível) e as requisições e respostas são armazenadas e analisadas usando políticas de vulnerabilidades. As respostas também são analisadas usando dados coletados durante o estágio de rastreamento. Durante o estágio de exploração novos *links* podem ser encontrados e quando isso acontece esses *links* são adicionados ao resultado obtido no estágio de rastreamento para também serem escaneados. Ao final, os resultados são exibidos ao usuário e podem ser salvos para uma análise posterior. Muitos scanners também fornecem informações sobre as vulnerabilidades detectadas como, por exemplo, maneiras de evitá-las ou corrigi-las.

Os testes realizados no estágio de exploração são baseados em uma coleção de assinaturas (conjunto de dados, informações) sobre vulnerabilidades de segurança conhecidas que os scanners possuem. Essas assinaturas podem ser atualizadas regularmente de acordo com o surgimento de novas vulnerabilidades. Os scanners também possuem um conjunto de testes predefinidos para essas vulnerabilidades.

Os testes para identificar vulnerabilidades de segurança, bem como a maneira de executar os estágios de configuração, rastreamento e exploração variam de scanner para scanner. Então, os resultados obtidos por diferentes ferramentas podem variar bastante. Uma grande quantidade de scanners de vulnerabilidade tem surgido no mercado e, devido a essa variação de resultados, a principal dificuldade é saber quais deles são os mais adequados para ser utilizado em determinada empresa ou organização. Avaliar a qualidade e eficácia destas ferramentas em detectar, corretamente, vulnerabilidades de segurança é fundamental para que essas vulnerabilidades possam ser corrigidas e com isso a aplicação atingir maiores níveis de dependabilidade. Conhecendo o conjunto de vulnerabilidades detectado por cada ferramenta é possível fornecer diretrizes para a seleção de um ou mais scanners que cubram a detecção das falhas de software que representem as principais causas de ataques. São scanners disponíveis comercialmente: Acunetix [8], N-Stalker [9], Rational AppScan [10], Cenzic Hailstorm [11], HP Webinspect [12],

Sandcat [13]. Também existem scanners *freeware*, tais como, BurpSuite [14] e Gamja [15], estas últimas com ausência de algumas funcionalidades encontradas nas ferramentas comerciais. Os scanners comerciais normalmente apresentam interface intuitiva e de fácil utilização. A interação é feita somente durante o estágio de configuração. Os demais estágios são transparentes ao usuário, que só visualiza a situação (*status*), disponibilizada pelo scanner constantemente enquanto ele analisa a aplicação. Existem também, ao final da execução, opções de geração de relatórios e informações sobre as vulnerabilidades de segurança detectadas, como informações técnicas e recomendações para a correção dessas vulnerabilidades.

1.3 Objetivos

O principal objetivo deste trabalho para melhorar o cenário retratado pela falta de segurança em aplicações *Web* é estabelecer uma abordagem para avaliar a eficácia dos scanners de vulnerabilidade. Baseada no procedimento de injeção de falhas e modelos de árvores de ataques, essa abordagem deve permitir analisar, além da falta de cobertura, a quantidade de falsos positivos que as ferramentas apresentam, uma vez que este fator é tão importante quanto a cobertura, pois faz com que a organização responsável pelo desenvolvimento do sistema envide esforços para sanar um problema não existente.

Para estabelecer a abordagem é necessário definir alguns objetivos secundários. Primeiramente, pretende-se entender melhor como as falhas de software podem ser exploradas para burlar aspectos de segurança do software. Isto implica entender o(s) defeito(s) que ocorre(m) como consequência da exploração de determinada vulnerabilidade e identificar qual(is) tipo(s) de falha(s) é (são) responsável(eis) por esta vulnerabilidade, definindo um *faultload* representativo. Como as falhas de software normalmente refletem características específicas de cada linguagem de programação, optou-se, neste trabalho, por investigar falhas de software da linguagem de programação Java, pois é uma linguagem bastante utilizada para construir aplicações *Web* atualmente. Consequentemente, as aplicações a serem testadas pelos scanners também são desenvolvidas nesta linguagem.

A partir da identificação dessas falhas, pretende-se injetá-las, individualmente e de maneira controlável, no código fonte de aplicações alvo para investigar o efeito de sua ativação. O objetivo é entender, por meio da análise do contexto do código fonte onde a falha foi injetada, como esta falha pode afetar o comportamento das aplicações em relação a vulnerabilidades de segurança. Então, pretende-se analisar como isto afeta o comportamento das ferramentas detectoras de vulnerabilidades. O objetivo desses experimentos é verificar se os *scanners* de vulnerabilidade são eficientes para detectar os tipos de vulnerabilidades para os quais foram desenvolvidos.

Também é necessário, para estabelecer a abordagem, entender os modelos de árvores de ataques e vulnerabilidades de segurança que serão analisadas. Ao final, pretende-se fazer um estudo de caso para aplicar a abordagem estabelecida, validando a eficácia de scanners de vulnerabilidades comercialmente disponíveis e fornecendo diretrizes para aquisição ou utilização destes scanners.

1.4 Abordagem Proposta

Uma das maneiras de se garantir a dependabilidade de sistemas de software é aplicar procedimentos de remoção e previsão de falhas durante o processo de validação do software. Esses procedimentos consistem em, respectivamente, reduzir o número ou a severidade das falhas e avaliar a consequência da ativação de falhas que, porventura, permaneçam no produto de software. Técnicas de injeção de falhas têm sido consideradas muito úteis para contribuir com o processo de validação de software [16].

Injeção de Falhas é uma técnica de validação onde se procura produzir ou simular a presença de falhas (*fault* em inglês) e se observa o sistema para se verificar qual será sua resposta nessas condições [16]. Acelerando a ocorrência de erros e defeitos, essa técnica é uma valiosa abordagem para validar sistemas onde a segurança é um requisito indispensável. No entanto, os resultados são mais relevantes se a emulação das falhas for feita de maneira realista. Isto implica a necessidade de conhecer os tipos de falhas encontradas (característica) e suas respectivas distribuições no ambiente operacional (frequência), ou seja, implica a necessidade de se identificar a representatividade das falhas. Para identificar um conjunto realista de falhas foi desenvolvido um estudo de campo sobre aplicações Java que se encontram em fase operacional (isto é, aplicações que apresentam várias versões disponíveis). A linguagem de programação Java foi escolhida devido à sua extensa utilização para desenvolver aplicações modernas, principalmente aplicações *Web*. Neste estudo de campo são analisados *patches* de correções de vulnerabilidades de segurança para identificação das falhas de software e a frequência com que ocorrem.

Conhecendo a representatividade das falhas, estas são injetadas utilizando técnicas de injeção de falhas de software. Foram definidos *workloads* para diferentes famílias de aplicações *Web* do segmento de gestão de informações (gestão de aprendizagem para ensino à distância, *Enterprise Resource Planning* – ERP e *Customer Relationship Management* – CRM) e as falhas que constituem o *faultload* foram injetadas nessas aplicações para a realização dos experimentos. O contexto do código fonte da aplicação onde a falha é injetada deve ser analisado para avaliar a relação entre a falha e possíveis vulnerabilidades de segurança criadas, como também deve ser analisado o impacto da falha no funcionamento da aplicação e, consequentemente, no comportamento do *scanner*. Outro estudo importante e que está diretamente ligado à segurança de sistemas é a análise da eficiência de ferramentas detectoras de vulnerabilidades. É sabido que essas ferramentas divulgam ao

mercado suas capacidades em detectar vulnerabilidades específicas, porém, não se encontram com facilidade estudos que comprovem o grau de eficácia dessas ferramentas (qual a taxa de vulnerabilidades detectadas corretamente) e sua capacidade de minimizar os falsos positivos (qual a taxa de vulnerabilidades detectadas que, na realidade, não existem na aplicação sob teste).

A detecção de vulnerabilidades segue uma abordagem baseada em modelos de árvores para geração de ataques onde o ponto de vista do atacante é enfatizado. Ela consiste na realização de testes de segurança baseados em modelos. Isto permite fornecer, para os testadores, meios de realizar ataques que explorem determinadas vulnerabilidades, utilizando uma abordagem caixa preta. Com isto é possível prever quais ataques às vulnerabilidades produzidas por má codificação alteram o comportamento da aplicação *Web* em questão, possibilitando que sejam projetados procedimentos para eliminar ou minimizar o efeito desses ataques, aumentando o nível de dependabilidade da aplicação.

1.5 Contribuições do trabalho

Este trabalho contribui, primeiramente, para a definição de um *faultload* representativo para injeção de falhas específicas de segurança em aplicações Java. Este *faultload* também foi utilizado como base para o desenvolvimento de uma ferramenta injetora de falhas para aplicações desenvolvidas utilizando a linguagem Java, a J-SWFIT [17], que será abordada na seção 4.1.

Outra contribuição é a definição de árvores de ataques, que descrevem as diferentes possibilidades de ataques a determinados tipos de vulnerabilidades de segurança, auxiliando a realização de testes contra estas vulnerabilidades. Uma ferramenta injetora de ataques está sendo desenvolvida paralelamente a este trabalho, baseando-se na modelagem dos ataques.

A principal contribuição é o estabelecimento de uma abordagem com suas respectivas etapas e ferramentas de apoio para analisar a eficiência de *scanners* de vulnerabilidades. Esta abordagem também se mostra útil para realização de testes manuais de segurança. Sabe-se que as ferramentas automáticas demandam certo investimento financeiro e nem sempre empresas ou instituições podem adquiri-las. Utilizando os cenários de ataques modelados como parte da abordagem proposta é possível realizar testes manuais de segurança para detecção de vulnerabilidades, proporcionando maior nível de dependabilidade para a aplicação sob teste.

Essa abordagem pode também representar uma das dimensões necessárias para construção de um *benchmark* que, de acordo com o trabalho de Kanoun *et al.* [18], é definido como: a dimensão experimental, que inclui o ambiente operacional, o *workload*, o *faultload*, e os procedimentos e regras necessários para realizar um experimento. As demais dimensões que compõem um *benchmark*, como medidas (inclui a maneira que medidas são obtidas e o domínio - por exemplo, área de aplicação - para a qual estas medidas são válidas e significativas), categorização (inclui o uso do

benchmark, a fase do ciclo de vida, área da aplicação e a natureza do sistema alvo) e propriedades (representam os requisitos para que o *benchmark* seja válido e útil, definindo, por exemplo, que o *benchmark* deve ser portável, rentável, etc.) não são abordadas no presente trabalho.

Resultados da análise das ferramentas indicam qual o *scanner* que apresenta a maior cobertura de vulnerabilidades de segurança (do tipo que se diz capaz de detectar) e o menor número de falsos positivos. Este resultado serve como diretriz para utilização dessas ferramentas em sistemas reais, sugerindo até mesmo a opção de combinar essas ferramentas para obtenção de melhores resultados na detecção de vulnerabilidades.

1.6 Organização do texto

O presente trabalho está organizado da seguinte forma:

Capítulo 2 – Trabalhos Relacionados. Mostra os trabalhos que serviram de base para a presente dissertação, abrangendo a definição de *faultloads* realistas, análise da eficácia de scanners de vulnerabilidades, modelos de ataques e vulnerabilidades de segurança.

Capítulo 3 – Segurança de Sistemas Computacionais. Aborda aspectos de segurança em diferentes contextos, como segurança de redes, de servidores e de protocolos de comunicação. Também aborda a segurança de aplicações *Web* e suas principais vulnerabilidades.

Capítulo 4 – Injeção de Falhas e Modelos de Ataques. Este capítulo explica alguns conceitos de injeção de falhas e modelos de ataques, duas bases teóricas deste trabalho. Para injeção de falhas são apresentados ambientes, técnicas e ferramentas. Para modelos de ataques são apresentados a estrutura de árvore, seleção de cenários e padrões de ataques.

Capítulo 5 – Abordagem, Experimentos e Resultados. Relata a abordagem para validar a eficácia dos scanners de vulnerabilidades utilizando técnicas de injeção de falhas e modelos de ataques. A eficácia é avaliada em relação à cobertura na detecção de vulnerabilidades de segurança e na quantidade de falsos positivos encontrados. Neste capítulo também é mostrado um estudo de caso utilizando a metodologia e seus resultados.

Capítulo 6 – Conclusão. Apresenta as conclusões do estudo desenvolvido e algumas direções para trabalhos futuros.

Apêndice A – apresenta o estudo de caso desenvolvido para identificar um *faultload* para falhas específicas da linguagem Java e do paradigma de orientação a objetos.

Apêndice B – traz o trabalho desenvolvido para analisar o efeito destas falhas de software sobre vulnerabilidades de segurança e, consequentemente, na detecção das vulnerabilidades pelos scanners comerciais.

Apêndice C – apresenta a modelagem das árvores de ataques - referente às vulnerabilidades abordadas nesta dissertação - para testes de segurança em aplicações web.

2 Capítulo 2

Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos mais relevantes que serviram de base para o desenvolvimento desta dissertação. Os principais tópicos aqui discutidos foram divididos em três partes e trazem trabalhos relacionados a: (i) definição de *faultload* realista, incluindo falhas de segurança e esquema de classificação de falhas de software; (ii) análise da eficácia de scanners de vulnerabilidades; (iii) árvores de ataques e vulnerabilidades de segurança.

2.1 Definição do faultload

A definição de um *faultload* realista é fundamental para o sucesso da validação da dependabilidade e dos mecanismos de tolerância a falhas de sistemas de software. Isto significa que a maior preocupação ao utilizar ambientes e ferramentas de injeção de falhas é assegurar que as falhas injetadas representem falhas reais, pois é a condição necessária para obter resultados significativos. Esta preocupação se aplica principalmente a falhas de software, ainda um assunto pouco explorado, até mesmo devido à sua complexidade.

Diante da importância de definir *faultloads* realistas, alguns trabalhos investigam falhas reais de software e a acurácia da injeção destas em código alvo. O trabalho de Madeira *et al.* [19] apresenta um estudo experimental para analisar a viabilidade de se injetar falhas de software por meio de ferramentas SWIFI (*Software Implemented Fault Injection*). Para isto, os autores definem um conjunto de falhas reais de software, identificadas em diferentes aplicações. A identificação das falhas se dá por testes intensivos em programas, executando-os várias vezes com diferentes dados de entrada. Se em algum teste o sistema apresentar algum defeito, o código é analisado para identificar a falha. Estas falhas são então comparadas com a emulação realizada pela ferramenta Xception [20] e resultados revelam limitações da ferramenta para emular todos os tipos de falhas de software identificadas. Uma segunda investigação foi realizada sobre a possibilidade de emulação de todas as falhas de uma mesma classe. Resultados mostraram que o impacto das falhas emuladas sobre o sistema foi muito maior, de forma que somente uma pequena porcentagem das falhas permaneceu inativa e não afetaram o sistema. Os autores acreditam que isto se deve também às características específicas da ferramenta.

Christmansson e Chillarege [21] também investigam a acurácia ao emular falhas pela sua injeção no código executável. Para isso, os autores coletaram um conjunto de falhas reais de software a partir de defeitos coletados nos sistemas MVS, IMS e DB2, em fase operacional.

O trabalho de Durães e Madeira [22] apresenta um conjunto de falhas reais de software. Essas falhas são investigadas por meio de *patches* de correção de aplicações desenvolvidas utilizando a linguagem de programação C. Uma ferramenta *diff* (i.e, ferramenta que compara dois códigos e mostra o que foi mudado) é aplicada a diferentes versões do mesmo software e destaca as alterações, tornando possível a visualização da correção efetuada, possibilitando a identificação da falha. Uma nova técnica de injeção de falha é proposta pelos autores; para validar esta nova técnica as falhas são emuladas no código executável do sistema alvo.

Para auxiliar na definição de modelos de falhas realistas, alguns trabalhos foram desenvolvidos com o objetivo de investigar e propor esquemas de classificação de falhas de software, descritos a seguir.

Classificação de falhas de software. Para definir um *faultload* realista é necessário identificar a natureza e a frequência das falhas presentes nos produtos de software em fase operacional (i.e., sua representatividade) e uma forma de classificar essas falhas. Perry e Evangelist [23] apresentam um trabalho inicial sobre classificação das falhas de software. O trabalho utiliza falhas identificadas durante o desenvolvimento de vários programas desenvolvidos utilizando a linguagem de programação C, com enfoque em falhas de interface. No trabalho é utilizado um método de classificação de falhas que relaciona tipos de falhas com as correções efetuadas pelos programadores. No entanto, a classificação de falhas proposta pelos autores não é suficientemente detalhada para ser utilizada no procedimento de injeção de falhas.

O trabalho de Sullivan e Chillarege [24] apresenta um estudo sobre o impacto dos defeitos de software em sistemas desenvolvidos pela IBM baseando-se em informação sobre defeitos identificados ao longo de cinco anos. As falhas foram classificadas de acordo com o tipo de defeitos que causaram e, em uma segunda etapa, pelo conceito de enganos cometidos durante a etapa de codificação. No trabalho as falhas são descritas com um grau de detalhe próximo do necessário para a injeção de falhas.

Os mesmos autores estendem o estudo anterior [25] com a inclusão de mais falhas descobertas em sistemas denominados DM2 e IMS, apresentando a conclusão de que a distribuição de falhas, quando classificadas pelo tipo de defeito, é dependente do estágio de desenvolvimento do produto. Chillarege *et al.* [26] estendem o conceito e propõem o esquema de classificação ODC (*Orthogonal Defect Classification*). O esquema de classificação ODC foi desenvolvido com o propósito de identificar e classificar falhas de software de forma rápida para prover dados para análise e melhoria no desenvolvimento e teste de software. Neste esquema, as falhas de software são classificadas de acordo com a alteração do código necessária para sua correção. As falhas são classificadas de acordo com as seguintes classes:

- *assignment* (atribuição): valores são atribuídos incorretamente ou não são atribuídos;
- *checking* (teste): validação de dados se dá de forma incorreta ou expressões condicionais são incorretas;

- *interface* (interface): interação entre componentes é feita de forma incorreta;
- *algorithm* (algoritmo): implementação incorreta ou faltante que pode ser corrigida sem a necessidade de alteração do projeto do sistema;
- *function* (função): funcionalidade do sistema que não existe ou foi incorretamente implementada;
- *time/serialization* (temporização/serialização): coordenação incorreta do tempo de compartilhamento de recursos.

No entanto, visto que o objetivo dos autores [26] é melhorar o processo de desenvolvimento do software e não o processo de injeção de falhas, as classes que definem o esquema de classificação ODC são genéricas e abrangentes, não oferecendo o nível de detalhe de informação necessário para a reprodução das falhas.

No trabalho de Durães e Madeira [22], a classificação ODC é estendida e minuciosamente representada a tal ponto que é possível descrever falhas de maneira exata em nível de máquina e, desta forma, podem ser injetadas em sistemas baseados em computadores para que seu comportamento seja observado.

Para propor a extensão da classificação ODC, os autores analisaram falhas dentro do contexto do programa no qual elas se inserem e relacionaram estas falhas com construções da linguagem de programação. As falhas são então classificadas de acordo com sua natureza. A Tabela 2.1 apresenta a classificação dos tipos de falhas mais frequentemente observadas em aplicações escritas na linguagem C e suas respectivas classificações ODC.

Tabela 2.1. Tipos de falhas mais frequentemente observadas em aplicações C.

Tipo de falha	**Descrição**	**Tipo ODC**
MFC	Missing function call	Algorithm
MVIV	Missing variable initialization using a value	Assignment
MIA	Missing if construct around statements	Checking
MIFS	Missing if construct plus statements	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	Checking
MLOC	Missing "OR EXPR" in expression used as branch condition	Checking
WVAV	Wrong value assigned to variable	Assignment
WPFV	Wrong variable used in parameter of function call	Interface
WFCS	Wrong function called with same parameters	Algorithm
ELOC	Extraneous "OR EXPR" in expression used as branch condition	Checking
EFC	Extraneous function call	Algorithm

Definição de *faultload* com falhas de software que levam a vulnerabilidades de segurança. Existe muito pouco trabalho relativo à definição de *faultloads* baseados em falhas de software que levam a vulnerabilidades de segurança. O trabalho apresentado por Fonseca e Vieira [3] utiliza o esquema de classificação proposto por Durães e Madeira [22] e investiga falhas de segurança em aplicações desenvolvidas utilizando a linguagem de programação PHP (*Hypertext Preprocessor*) para entender a potencial relação entre certos tipos de falhas de software e as vulnerabilidades de segurança.

Para entender qual código é responsável por esses problemas de segurança o estudo foi baseado em *patches* que corrigem as vulnerabilidades acima citadas. Comparar estes *patches* de correção com versões anteriores (i.e., vulneráveis) torna possível identificar e classificar falhas reais que levam a vulnerabilidades de segurança. Os autores também definem algumas regras para uma análise coerente desses *patches* para evitar enganos durante a classificação das falhas. Um novo tipo de falha é identificado em função de características específicas da linguagem de programação e uma nova classe de falhas é criada, seguindo a nomenclatura e estendendo a classificação proposta por Durães e Madeira [22]. A distribuição de falhas de segurança é comparada com falhas de software genéricas apresentadas em outros estudos [21] [22] e conclui-se que os tipos de enganos que levam a vulnerabilidades de segurança têm uma distribuição diferente dos tipos de falhas genéricas. Esta diferença também é afetada pelo tipo de linguagem de programação e tecnologia envolvida ao analisar falhas de segurança.

O problema é que o estudo apresentado [3] é específico para aplicações desenvolvidas utilizando a linguagem PHP e as falhas não podem ser facilmente generalizadas. Seixas *et al.* [27] analisaram falhas de software que levam a vulnerabilidades de segurança do ponto de vista do tipo de linguagem, ou seja, fracamente tipadas ou fortemente tipadas. Em linguagens fracamente tipadas um valor de um tipo pode ser tratado como outro tipo (por exemplo, uma string pode ser tratada como um número). Linguagens fortemente tipadas previnem esta situação e a tentativa de usar um valor errado para determinado tipo, resulta em uma exceção. Exemplos de linguagens fracamente tipadas: PHP, PERL, CGI; exemplos de linguagens fortemente tipadas: Java, C# e VB.NET. As três últimas foram utilizadas no estudo.

O objetivo de Seixas *et al.* [27] é investigar se os tipos de falhas de software que levam a vulnerabilidades são independentes da linguagem de programação. Os autores utilizaram a mesma metodologia e compararam os resultados com os de Fonseca e Vieira [3], que utilizam uma linguagem fracamente tipada (PHP). Observaram que aplicações escritas nessas linguagens apresentaram um menor número de vulnerabilidades relatadas e que os tipos de falhas mais frequentemente encontrados são independentes do tipo da linguagem utilizada.

No entanto, o trabalho de Seixas *et al.* [27] agrupa várias linguagens em uma mesma classificação (fortemente tipadas) e não apresenta falhas específicas de acordo com as características de cada linguagem. Em ambos os estudos ([3] e [27]) somente são investigadas

vulnerabilidades do tipo *Cross Site Scripting* (XSS) e injeção de SQL, o que restringe as falhas a estes tipos de vulnerabilidades.

O estudo de campo desenvolvido por Basso *et al.* [28] pode ser visto no Apêndice A . O estudo de campo foi realizado sobre falhas reais de aplicações desenvolvidas em Java, incluindo falhas de segurança. Com base na análise de *patches* de correção disponíveis em repositórios de livre acesso, mais de 550 falhas foram analisadas e classificadas, determinando a sua representatividade. Os autores também definem novos operadores, específicos para esta estrutura de linguagem de programação, guiando a definição de *faultload* com falhas específicas da linguagem Java. Este *faultload* foi utilizado para os experimentos que avaliam a eficácia dos scanners de vulnerabilidades; os trabalhos relacionados a estes são apresentados a seguir.

2.2 Análise da eficácia de scanners de vulnerabilidade

Poucos trabalhos foram desenvolvidos para avaliar a eficácia dessas ferramentas. A eficácia de um scanner pode ser medida em relação à cobertura na detecção de vulnerabilidades de segurança e em relação aos falsos positivos, ou seja, scanners com maior eficácia apresentam maior taxa de de cobertura e menor taxa de falsos positivos.

Bau et al. [29] desenvolveram um trabalho para avaliar o estado da arte dos scanners de vulnerabilidade. Os autores avaliaram a eficácia de oito scanners comerciais em relação à detecção de vulnerabilidades do tipo *Cross Site Scripting*, Injeção de SQL e *Cross Site Request Forgery*. A validação dos resultados foi baseada em vulnerabilidades já conhecidas nas aplicações sob teste. O conhecimento dessas vulnerabilidades se deu a partir do conhecimento de patches de correções para corrigi-las, ou seja, se determinada versão de uma aplicação tem um patch de correção para determinada vulnerabilidade, ela de fato apresenta essa vulnerabilidade e o scanner deveria detectá-la. No entanto, Bau et al. [29] não desenvolvem um estudo comparativo avaliando as ferramentas individualmente e não informam dados comparativos nem provêm diretrizes para aquisição ou utilização dos scanners. Os resultados mostram que, como um grupo, os scanners apresentaram baixa cobertura e vários casos de falsos positivos.

Fonseca *et al.* [30] propuseram um método para avaliar e produzir um *benchmark* para *scanners* de vulnerabilidade. Este *benchmark* é produzido por meio de técnicas de injeção de falhas, em que falhas classificadas de acordo com o modelo proposto por Durães e Madeira [22] são injetadas no código de aplicações *Web*. As falhas são injetadas utilizando a técnica G-SWFIT [22] e a aplicação é analisada pelo *scanner* para verificar se este é capaz de identificar a potencial vulnerabilidade criada pela falha injetada. Então, uma inspeção manual do código é realizada para garantir a consistência dos resultados.

Três ferramentas comerciais consideradas líderes de mercado foram avaliadas e os resultados foram analisados individualmente para cada ferramenta, mostrando que, em geral, a cobertura das ferramentas é baixa e a porcentagem de falsos positivos é bastante alta. No entanto, este estudo teve seu enfoque em aplicações *Web* desenvolvidas utilizando a linguagem de programação PHP e considerando dois tipos de vulnerabilidades: *Cross Site Scripting* (XSS) e injeção de SQL. Isto faz com que os resultados obtidos não sejam facilmente generalizados, considerando tanto as especificidades de diferentes linguagens de programação quanto as especificidades de outros tipos de vulnerabilidades comuns a aplicações *Web*.

Outro trabalho que avalia a eficácia de *scanners* de vulnerabilidade foi desenvolvido por Vieira *et al.* [31]. Os autores apresentam uma avaliação experimental de vulnerabilidades de segurança em 300 *Web services* disponíveis publicamente. Quatro diferentes *scanners* de uso comercial foram utilizados com o objetivo de identificar tipos comuns de vulnerabilidades em ambientes de *Web services*. Os resultados mostram, primeiramente, que muitos dos serviços testados foram desenvolvidos sem muita preocupação com segurança, já que uma grande quantidade de vulnerabilidades foi detectada. Também, ao analisar o desempenho dos *scanners*, os autores observaram baixa cobertura e alto índice de falsos positivos. Embora a análise de *Web services* contribua para reforçar o estudo da falta de segurança em aplicações e da baixa eficácia das ferramentas disponíveis, os tipos de aplicações analisadas ainda formam um grupo restrito, e um grupo mais abrangente mostra-se necessário para obtenção de resultados mais gerais.

Outra observação importante é que ambos os trabalhos [29][31] não apresentam claramente uma abordagem para validar as vulnerabilidades detectadas pelos *scanners*. Apenas mencionam que uma análise manual foi realizada, mas não apresentam detalhes da realização dessa análise.

O Apêndice B descreve o trabalho desenvolvido por Basso *et al.* [32]. No trabalho, as falhas de segurança mais frequentemente observadas por Basso *et al.* [28] são injetadas em aplicações web para analisar o efeito das falhas de software sobre vulnerabilidades de segurança e, consequentemente, sobre os resultados do scanner de vulnerabilidades. O objetivo é entender, pela análise do contexto do código fonte e das tecnologias usadas para a construção das aplicações alvo, como as falhas de software podem afetar a eficácia dos resultados providos pelos scanners; e, a partir do conhecimento obtido deste estudo, estabelecer uma abordagem para validar as vulnerabilidades detectadas pelos *scanners.* Técnicas de injeção de falhas e modelos de árvores de ataques foram usados nos experimentos e como resultado pôde-se observar que as falhas injetadas realmente afetaram o comportamento da aplicação e do scanner. Um alto índice de falsos positivos e falhas de cobertura resultam da aplicação do scanner utilizado no experimento.

.

2.3 Modelos de ataques e vulnerabilidades de segurança

A modelagem de ataques é usada para descrever as etapas de um ataque bem-sucedido e pode utilizar desde modelos simples de árvores [33] a métodos baseados em redes de Petri [34] ou modelos baseados em UML [35]. O que distingue as abordagens de modelagem de ataques de outros métodos de modelagem de aspectos funcionais é que na modelagem de ataques o ponto de vista de um atacante é enfatizado, fornecendo, para os testadores, meios de injetar ataques que explorem determinadas vulnerabilidades de segurança presentes no sistema sob teste. É importante que o modelo represente ataques reais e permita detectar o máximo possível de vulnerabilidades a partir dos cenários de ataque injetados. A seguir são apresentados dois trabalhos mais recentes que caminham nesse sentido.

O trabalho de Wang *et al.* [36] propõe uma abordagem de testes de segurança dirigida por modelo, onde as ameaças (ataques) são modeladas usando diagramas de sequência UML. Basicamente, os testes de segurança seguem uma sequência de passos onde, primeiramente, os cenários de ataques são modelados no diagrama. A partir do modelo é extraído um conjunto de eventos, composto por uma sequência de mensagens que representa o comportamento do ataque e que não deveriam ocorrer durante a execução normal do sistema. O código fonte é instrumentado para coletar informação em tempo de execução, tais como, nome de métodos e classes. Isto permite gerar uma sequência de eventos que representa o comportamento do sistema durante sua execução. Os dados de teste são gerados aleatoriamente e um grande número de entradas é necessário para ativar uma vulnerabilidade, o que compromete a eficiência da abordagem e a duração dos experimentos. Estes dados de teste permitem que a aplicação, ao ser executada, colete traços que representam o comportamento do sistema. Este passo é importante para a identificação da vulnerabilidade. Esta identificação se dá pela comparação entre os traços da execução da aplicação e os traços do modelo de ameaças. Se ambos os traços forem correspondentes conclui-se que uma vulnerabilidade foi detectada. Embora uma aplicação simples seja usada como estudo de caso e apenas um ataque simples de violação de autorização foi realizado, a abordagem poderia ser aplicada a outras classes de ataques reais já que depende exclusivamente do modelo. O uso da abordagem ainda se mostra bastante restrito.

O trabalho de Martins *et al.* [37] tem como objetivo detectar vulnerabilidades de segurança de um protocolo usado em redes sem fio (protocolo WAP – *Wireless Application Protocol*). Para isto os autores propuseram um método que utiliza geração de árvores de ataques para representar ataques relatados e conhecidos do protocolo sob teste. Os ataques, previamente modelados, variam o mínimo possível de parâmetros, e, assim, diminui-se drasticamente a carga de falhas e aumenta-se a eficiência da metodologia. A carga de falhas é gerada a partir de cenários de ataques reais, que foram previamente selecionados de uma base de dados de ataques catalogados. Para emular os ataques foram utilizadas técnicas e ferramentas de injeção de falhas, de forma que cenários de

ataques foram gerados em um formato independente do injetor de falhas utilizado. O método permite também a criação de variantes de ataques conhecidos, e, consequentemente, a emulação de novos ataques e detecção de possíveis novas vulnerabilidades. Experimentos foram realizados e vulnerabilidades foram identificadas. A exploração dessas vulnerabilidades por meio de ataques violou propriedades de integridade e disponibilidade do protocolo WAP. A utilização de modelagem de ataques reais para realização dos testes de segurança se mostrou bastante adequada para a geração de casos de teste e detecção de vulnerabilidades.

Árvores de ataques. Além do trabalho de Martins *et al.* [37], vários trabalhos tem utilizado modelagem de árvore de ataques para analisar possíveis ameaças à segurança de sistemas, redes, implementações de protocolos, entre outras aplicações. Porém, poucos trabalhos utilizam essa modelagem para analisar vulnerabilidades em aplicações web.

Árvores de ataques foram primeiramente utilizadas por Schneier [33] para prover uma maneira formal de descrever a segurança de um sistema. O autor propõe representar ataques contra um sistema em uma estrutura de árvores onde o objetivo é o nó raiz e as diferentes maneiras de se atingir o objetivo são os nós folhas.

Byres *et al.* [38] descrevem a aplicação de árvores de ataques para avaliar a segurança de protocolos de comunicação para troca e controle de mensagens em redes industriais denominados SCADA (*Supervisory Controls and Data Acquisition*). Os autores identificaram onze possíveis objetivos de atacantes e com isto identificaram vulnerabilidades de segurança inerentes à especificação e depuração deste protocolo.

Edge *et al.* [39], com a finalidade de avaliar a segurança de um sistema bancário online, utilizaram árvores de ataques para identificar como atacantes comprometem as contas bancárias e com isso desenvolver e aplicar contra-medidas para protegê-las.

O trabalho de Fung *et al.* [40] utiliza árvore de ataques para analisar a “sobrevivência” (do inglês *survivability,* que significa a capacidade do sistema de cumprir suas missões e objetivos mesmo na presença de ataques ou defeitos) em sistemas baseados em uma arquitetura orientada a serviços (*service-oriented architectures* - SOA). Ataques a componentes que possam ser comprometidos foram identificados e modelados na árvore de ataques. A partir desta modelagem, cenários de intrusões foram identificados e isto permitiu aos autores sugerir uma medida quantitativa e prover sugestões para melhorar a “sobrevivência” do sistema analisado.

O trabalho de Lin *et al.* [41] apresenta um modelo de árvores de ataques para explorar vulnerabilidades de aplicações web do tipo *Cross Site Request Forgery* (CSRF). Esta vulnerabilidade permite que, diante de um ataque, o navegador da vítima (autenticado) envie requisições para a aplicação web vulnerável, realizando a ação desejada em nome da vítima, sem o conhecimento ou consentimento dela. O objetivo dos autores é apresentar a visão de um atacante para auxiliar pesquisadores a desenvolver defesas para este tipo de ataque. Os autores construíram a árvore a partir de um exemplo real de ataque a uma aplicação bastante popular, mas não realizaram nenhum

estudo de caso para avaliar a utilização da árvore para desenvolver ataques e identificar vulnerabilidades em aplicações. Além disso, o trabalho é restrito a apenas um tipo de vulnerabilidade.

No trabalho de Fernandes *et al.* [42], os autores apresentam a modelagem de árvores de ataques para explorar três diferentes tipos de vulnerabilidades de segurança: injeção de SQL, C*ross Site Scripting* (XSS) e C*ross Site Request Forgery* (CSRF). Vulnerabilidades do tipo injeção de SQL permitem que atacantes consigam, por meio de manipulação de entrada de dados de uma aplicação, interagir com as respectivas bases de dados. Vulnerabilidades XSS permitem que trechos de código chamados scripts sejam executados pela aplicação e com isso danos sejam causados. Vulnerabilidades CSRF, como mencionado no parágrafo anterior, permitem que o navegador da vítima (autenticado) envie requisições para a aplicação web vulnerável. Os autores também apresentam exemplos de casos de testes obtidos a partir das árvores de ataques e um estudo de caso foi desenvolvido para realizar testes de segurança em aplicações Web. Como resultado, os modelos de árvores de ataques se mostraram bastantes úteis para testes de segurança e, em vários casos, os testes baseados nestes modelos apresentaram resultados mais eficazes que resultados apresentados por scanners de vulnerabilidades executados sobre a mesma aplicação. Este trabalho pode ser visto no Apêndice C da presente dissertação.

Vulnerabilidades de segurança. Para modelar os ataques a vulnerabilidades de segurança é necessário ter conhecimento de suas causas, das consequências de sua exploração e principalmente os passos para realizar um ataque que a explore. Em busca deste conhecimento, utilizamos inúmeras fontes da internet, como *fóruns*, *blogs*, e *websites* sobre vulnerabilidades e invasões [43][44][45]. Contudo, alguns trabalhos principais serviram como base e são citados a seguir.

O banco de vulnerabilidades do *National Vulnerability Database* (NVD) [46], pertencente ao *National Institute of Standards and Technology* (NIST) [47] é considerado, atualmente, um dos repositórios mais abrangentes e completos de informação pública sobre vulnerabilidades em sistemas computacionais e redes. Este fornece, além de outras informações: **i)** informações gerais de uma vulnerabilidade: data de publicação, data da última revisão, descrição, versões de software vulneráveis, *patches* e recomendações fornecidas por empresas; **ii)** métricas que descrevem características inerentes de uma vulnerabilidade: complexidade do ataque (baixa, média ou alta), nível de autenticação necessário (nenhuma, única ou múltiplas), propriedades de segurança violadas (*confidencialidade*, *integridade* ou *disponibilidade*); **iii)** métricas que descrevem o efeito da vulnerabilidade dentro do ambiente de uma organização: potencial das perdas (baixa , média ou alta), porcentagem de sistemas vulneráveis (baixa , média ou alta), nível dos requisitos de segurança impactados (baixa , média ou alta). A partir deste banco de vulnerabilidades podem ser extraídas as informações necessárias para a obtenção dos cenários de ataques a serem injetados. Assim, é possível escolher diversos critérios para a seleção desses cenários, como por exemplo, pela severidade do ataque ou sua complexidade.

A OWASP (*Open Web Application Security Project*) [48] é uma comunidade de livre acesso cujo enfoque é melhorar a segurança de aplicações *Web.* Sua missão é divulgar aspectos de segurança para este tipo de aplicação, de forma que pessoas e organizações possam se manter informadas e com isso tomar as melhores decisões diante de riscos de segurança. Nesta comunidade é possível encontrar informações sobre as vulnerabilidades; explicações sobre como funcionam os ataques a estas; exemplos de testes caixa preta para identificar a vulnerabilidade; entre outros assuntos relacionados.

No trabalho de Uto e Melo [49] os autores apresentam, além de outras vulnerabilidades, uma breve descrição dos três tipos de vulnerabilidades de interesse (injeção de SQL, XSS e CSRF) e mostra, por meio de diversos cenários, como elas podem ser exploradas por usuários maliciosos.

Halfond e Viegas [50] desenvolveram uma comparação entre técnicas para detectar e prevenir injeção de SQL. Para isto os autores fazem uma revisão dos diferentes ataques conhecidos a vulnerabilidades do tipo injeção de SQL, descrevendo e exemplificando como esses ataques são desenvolvidos. Entre as técnicas avaliadas estão teste de caixa preta, análise estática do código, sistemas de detecção de intrusão, entre outras. Elas foram avaliadas em termos de capacidade de detectar ou prevenir este tipo de ataque e os resultados mostraram que muitas apresentaram problemas ao detectar ataques que explorem a má codificação.

O trabalho de Jovanovic *et al.* [51] propõe uma solução que provê proteção automática para ataques a vulnerabilidades do tipo *Cross Site Request Forgery* (CSRF). Para desenvolver esta solução os autores introduzem, em detalhes, os conceitos e mecanismos sobre estes ataques. Com base neste conhecimento e utilizando uma abordagem baseada em Proxy junto ao servidor, desenvolveram um protótipo que detecta e previne ataques CSRF de maneira transparente para os usuários e para as aplicações *Web*. Experimentos foram realizados e demonstraram que a solução é viável. Os autores chamam a atenção para este tipo de vulnerabilidade, que recebe pouca atenção porque é, relativamente, desconhecida por desenvolvedores e atacantes em busca de alvos fáceis, mas que pode causar grandes danos quando explorada.

2.4 Resumo

Neste capítulo foram apresentados os trabalhos que serviram de base para a presente dissertação, abrangendo a definição de *faultload* realista, análise da eficácia de *scanners* de vulnerabilidades, árvores de ataques e vulnerabilidades de segurança.

Para a definição de *faultload* realista foram descritos trabalhos que investigam falhas reais de software pela análise do código fonte de aplicações reais, investigando, falhas que levam a vulnerabilidades de segurança. Também foram descritos trabalhos que propõem modelos de

classificação das falhas de software e um estudo de campo que foi desenvolvido, sobre falhas reais em Java, incluindo falhas de segurança.

Em seguida, foram citados trabalhos que avaliam a eficácia de *scanners* de vulnerabilidades, bem como suas limitações, especialmente no que diz respeito à metodologia utilizada nestes trabalhos. A partir da metodologia proposta, apresentamos também trabalhos que utilizam modelos de árvores de ataques para realização de testes de segurança. Estes trabalhos avaliam, mais especificamente, aspectos de segurança de protocolos de comunicação.

Finalmente, mostraram-se repositórios e comunidades que tornam públicas informações sobre vulnerabilidades de segurança e alguns trabalhos que descrevem e exemplificam como ataques a estas vulnerabilidades são desenvolvidos. Isto é importante para que a modelagem das árvores represente, da melhor forma possível, os ataques a determinadas vulnerabilidades.

3 Capítulo 3

Segurança de Sistemas Computacionais

Quando se fala sobre segurança de sistemas computacionais, há uma tendência em imediatamente pensar em atacantes danificando páginas e aplicações *Web*, roubando números de cartões de crédito e bombardeando servidores com ataques que causam negação de serviço (este ataque é conhecido como DoS – *Denial of Service*). Também se pensa em vírus, cavalos de tróia e vermes (*worms*). Mas estes são só alguns problemas, que recebem maior fama, porque representam algumas das ameaças mais significativas às aplicações *Web* atualmente. Outros problemas que também apresentam risco normalmente são ignorados, como administradores desonestos, funcionários descontentes, e usuários que utilizam dados sigilosos de forma equivocada. A solução para segurança de aplicações *Web* é um processo contínuo envolvendo pessoas e práticas [52].

Segundo Meier *et al.* [52], a segurança de sistemas computacionais baseia-se nos seguintes elementos:

- **Autenticação**: é o processo de identificar, de forma única, os clientes que utilizam determinados serviços e aplicações. Estes clientes podem ser usuários finais, outros serviços, processos ou computadores.

- **Autorização**: é o processo que gerencia os recursos e operações que um cliente autenticado pode ou não acessar, dependendo das restrições de acesso impostas a este cliente.

- **Auditoria**: auditoria e registro de informações sobre eventos e operações realizados a partir da aplicação (*log*) são processos importantes para garantir o não-repúdio, ou seja, garantir que um usuário não negue que desenvolveu ou iniciou alguma transação.

- **Confidencialidade**: também conhecido como privacidade, é o processo de assegurar que dados se mantenham privados e confidenciais, não sendo vistos por usuários não autorizados ou acessados durante tráfego na rede. Criptografia e listas de controles de acesso (ACLs) são recursos normalmente utilizados para auxiliar na garantia da confidencialidade.

- **Integridade**: é a garantia que os dados são protegidos de modificações acidentais ou propositais, de forma maliciosa. Assim como a confidencialidade, a integridade deve ter seu enfoque para dados que trafegam pela rede. Técnicas *hash* e códigos de autenticação de mensagens são os mecanismos mais comumente utilizados para auxiliar na garantia da integridade dos dados.

- **Disponibilidade**: do ponto de vista de segurança, disponibilidade significa que sistemas devem se manter disponíveis para usuários legítimos. O objetivo de muitos atacantes com ataques de negação de serviço (DoS) é derrubar uma aplicação ou torná-la sobrecarregada de forma que outros usuários não possam acessá-la.

Ainda segundo Meier *et al.* [52], a segurança tem sido considerada, tradicionalmente, uma questão que envolve principalmente rede de computadores. Mas para tornar uma aplicação segura, é necessária uma abordagem para prover segurança para redes, servidores e aplicação. Estas abordagens são discutidas a seguir.

3.1 Segurança de Redes, Servidores e Protocolos de Comunicação

Segurança de redes. Uma aplicação Web segura requer uma infra-estrutura de rede segura. Então, uma possível abordagem para a segurança de redes é a proteção da infra-estrutura e transmissão de dados. A infra-estrutura da rede é formada por dispositivos como roteadores, *firewalls* e *switches*. Roteadores são responsáveis por transmitir pacotes pela rede à qual estão conectados e devem ser usados para bloquear tráfego não autorizado ou não desejado. *Firewalls* servem para bloquear todas as portas desnecessárias e permitir tráfego somente em portas conhecidas, além de prevenir ataques pelo monitoramento da chegada de requisições. Já os *Switches* são responsáveis por encaminhar pacotes diretamente a um servidor ou segmento de rede.

Normalmente, um atacante procura por dispositivos de redes mal configurados para explorar. Vulnerabilidades comuns residem na rede em função de configurações padrão de instalação, controle de acesso com poucas restrições, e dispositivos com *patches* de correção desatualizados. Estas vulnerabilidades podem ser exploradas por ataques como [52]:

- ***Information gathering*** (coleta de informação). Consiste em um atacante obter informações (tais como topologia da rede, configurações do sistema e dispositivos de rede) e utilizar essas informações para explorar possíveis vulnerabilidades.

- **Sniffing**: neste tipo de ataque, o atacante monitora o tráfego da rede em busca de dados como senhas em claro (i.e., sem cifragem) ou informações sobre configuração. Além disso, dados cifrados que utilizam algoritmos mais simples podem ser facilmente decifrados.

- ***Spoofing***: consiste em esconder a verdadeira identidade de determinado usuário da rede. Um endereço de rede falso é utilizado de forma a não representar a real fonte de um ataque, por exemplo.

- ***Session hijacking*** (sequestro de sessão): também conhecido como ataque *man in the middle,* o atacante utiliza uma aplicação que mascara tanto o cliente quanto o servidor, fazendo com que ambos acreditem que estão se comunicando legitimamente, enquanto estão se comunicando com a aplicação do atacante. Isto possibilita a captura de informações confidenciais.

- ***Denial of service*** (negação de serviço): consiste em impedir que usuários legítimos utilizem determinados serviços. Para isto, atacantes sobrecarregam o tráfego da rede, consumindo banda e recursos.

O conhecimento de possíveis ataques que podem explorar vulnerabilidades e afetar a segurança da rede é fundamental para que contra medidas sejam aplicadas a fim de evitar ou amenizar as consequências destes ataques. Para os ataques citados acima, são indicadas algumas contra medidas como: uso de *firewalls* para proteger serviços que não devem ser expostos; forte segurança física que impeça que dispositivos maliciosos sejam inseridos na rede; uso de criptografia para cifrar dados de sessão e informações que trafeguem pela rede; filtragem de requisições *broadcast* (i.e, requisições que se espalham para todos os computadores da rede).

Segurança de servidores. A segurança de servidores é necessária para que a aplicação seja hospedada em uma base segura. Basicamente, esta segurança é provida por meio de definições de configurações. Estas configurações permitem reduzir vulnerabilidades que possam ser exploradas por ataques. Diretórios virtuais mal configurados, por exemplo, podem permitir acessos não autorizados; contas de usuários negligenciadas podem permitir que atacantes passem pelas defesas do servidor despercebidamente; uma porta ignorada pode ser a porta de entrada para um atacante.

Embora configurações estejam relacionadas ao sistema operacional do servidor, Meier *et al.* [52] definem uma abordagem sistemática e repetível para configurar um servidor *Web*. A metodologia proposta é reutilizável para diferentes tecnologias e um exemplo de configuração utilizando o sistema operacional Windows 2000 é apresentada. Os principais tipos de ataques a servidores *Web* são [52]:

- **Profiling**: processo exploratório onde o atacante obtém informações sobre a aplicação e, com isto, ataca possíveis vulnerabilidades. Este ataque pode se dar por meio de protocolos desnecessários, portas abertas e servidores que provêm facilmente informações de configurações.

- **Acesso não autorizado**: ocorre quando um usuário obtém, de forma maliciosa, permissão para acesso a informações ou realização de operações restritas.

- **Execução de código arbitrário**: ocorre quando um atacante executa código malicioso no servidor para comprometer recursos ou reproduzir ataques contra outros sistemas.

- **Elevação de privilégios**: atacante executa código por meio de uma conta de usuário privilegiado.

- **Vírus, vermes (*worms*) e cavalos de Tróia**: códigos maliciosos que muitas vezes são mascarados para simular códigos úteis, mas, na verdade, são prejudiciais, causando alto consumo de recursos, roubo de informações, etc.

Algumas medidas auxiliam no monitoramento do status da segurança do servidor para atualizá-lo regularmente e prevenir que novas vulnerabilidades sejam descobertas e exploradas. Algumas destas medidas são: acompanhar as ações dos grupos de usuários, especialmente usuários com maiores privilégios; analisar arquivos de *log* regularmente; analisar o software do servidor e atualizar *patches* de correção disponíveis regularmente.

Segurança de Protocolos de comunicação. Como parte da segurança de redes tem que se considerar a segurança de protocolos de comunicação. A disseminação de conhecimento

relacionado a protocolos de redes com fio (por ex. TCP/IP) e sem fio (por ex. padrão IEEE 802.11[1]) e habilidades de intrusão avançadas vêm fazendo destes protocolos de comunicação alvo de interesse para atacantes e definindo uma classe de vulnerabilidades que vem crescendo nos últimos anos. O *National Vulnerability Database* (NVD) [46] relatou um aumento de 138% de vulnerabilidades de autenticação e um aumento de 48% de vulnerabilidades de criptografia em 2008 em relação ao ano de 2007.

Um dos principais desafios para auxiliar a garantia da segurança dos protocolos de comunicação é, além do conhecimento dos tipos de ataques, a definição de técnicas de validação para identificar vulnerabilidades nestes protocolos. Estas técnicas têm sido estudadas como mostram os trabalhos de Martins *et al.* [37] e Xiao *et a*l. [53] onde, respectivamente, os autores testam protocolos de redes sem fio (WAP – *Wireless Protocol Application*) e com fio (IGMP - *Internet Group Management Protocol* e PIMDM - *Protocol Independent Multicast - Dense Mode*).

3.2 Segurança de aplicações Web

Vários pontos de fragilidades são identificados quando se trata de segurança de aplicações *Web*. Organizar estes pontos de fragilidades em categorias torna possível entender os enganos comuns que afetam a segurança e permite lidar com estes enganos de forma sistemática, a fim de aplicar contra medidas para evitá-los.

Tsipenyuk *et al.* [54] definem um conjunto de categorias que representam os pontos de fragilidades mais comuns. Estas categorias são listadas a seguir.

- **Validação e representação de entrada de dados.** Metacaracteres, codificações alternativas, representações numéricas podem causar problemas de representação e validação de entrada. Quando a validação dos dados de entrada é insuficiente ou inexistente, ataques do tipo XSS (*Cross-site scripting*), injeção de SQL, entre outros, podem ser realizados com sucesso.

- **Exploração de API.** Uma API (*Application Program Interface*) é um contrato entre um método solicitante e um provedor. Uma das formas mais comuns de exploração de APIs ocorre quando o solicitante não cumpre o seu final do contrato. Por exemplo, se um programa deixa de chamar o método *chdir( )* (responsável por mudar o diretório de trabalho atual para outro diretório especificado) depois de chamar o comando *chroot( )* (altera o diretório raiz para o diretótio especficado) ele viola o contrato que especifica como alterar, com segurança, o diretório raiz ativo.

[1] http://www.ieee802.org/11/

▪ **Recursos de Segurança**. Em um sistema é possível combinar diferentes recursos de segurança como, por exemplo, autenticação, controle de acesso, confidencialidade, criptografia, gerenciamento de privilégios, entre outros. Todo este conjunto traz dificuldades de gerenciamento e vulnerabilidades de segurança podem ser geradas neste contexto, armazenamentos inseguros e quebra do controle de acesso.

▪ **Comunicação**. A computação distribuída precisa compartilhar estados de alguma forma e isto requer tempo. A comunicação em um espaço de tempo abre possibilidades para que ataques sejam aplicados. Computadores modernos trabalham com várias *threads*, executando tarefas distintas simultaneamente. Defeitos relacionados a interações inesperadas entre *threads*, processos, tempos e troca de informações mascaram a maneira como a execução é feita na realidade. Quebra de autenticação e gerenciamento falho de sessão são vulnerabilidades encontradas neste contexto.

▪ **Erros**. Tratamento de erros representa um tipo especial de contrato tal como a exploração de APIs que pode facilitar ataques quando tratados de maneira imprópria.

▪ **Qualidade do Código**. Má qualidade do código leva a um comportamento imprevisível e, para o atacante, a má qualidade do software fornece uma oportunidade de ataque e exploração do sistema.

▪ **Encapsulamento**. Implica desenvolver proteção ao redor de recursos. Considerando navegadores *Web*, isso pode significar que não se pode ter acesso ao disco rígido de maneira arbitrária, ou seja, autenticação é necessária para acessar recursos protegidos.

▪ **Ambiente**. O ambiente é responsável por abrigar o produto de software e conectá-lo com o ambiente externo, sendo uma parte crítica para a segurança do sistema. Pode causar o gerenciamento falho da configuração do sistema.

Segundo McGraw [55], os primeiros livros e artigos acadêmicos relativos ao assunto de desenvolvimento de software seguro surgiram em 2001. Isto demonstra que desenvolvedores, arquitetos e cientistas da computação começaram a estudar este assunto em um período relativamente recente.

Existe uma diferença entre o estudo da segurança de aplicações *Web* em fase operacional e o estudo da segurança de software em desenvolvimento. Segundo McGraw [55], segurança do software em desenvolvimento envolve conceitos de construção de software seguro, como projetar e assegurar que o software é seguro; educar arquitetos, desenvolvedores e usuários sobre como construir um sistema com aspectos de segurança. Já a segurança em aplicações, aborda maneiras de proteger sistemas de software que estão prontos, depois que o desenvolvimento está completo. Isto inclui verificar e proteger as aplicações contra códigos maliciosos, prevenir exibição de informações que possam denunciar dados da aplicação (como mensagens de erro que informam campos do banco de dados, por exemplo), monitorar dados de entrada das aplicações e arquivos executáveis.

O desenvolvimento de software seguro visa contribuir com melhorias ao cenário da falta de segurança em aplicações *Web* e reduzir a quantidade de falhas residuais no código, que podem levar a vulnerabilidades de segurança. Desenvolvedores são incentivados a seguir melhores práticas de codificação, e isto inclui procedimentos como revisar aspectos de segurança no código da aplicação, realizar auditorias de segurança regularmente, utilizar ferramentas analisadoras de vulnerabilidades, entre outros. No entanto, em função de restrições de tempo e custo, desenvolvedores, muitas vezes, mantêm o enfoque principal do desenvolvimento da aplicação na funcionalidade e na satisfação dos requisitos do cliente, negligenciando aspectos de segurança. Por isso, a validação da segurança de aplicações *Web* em fase operacional, ou seja, com desenvolvimento completo, é uma atividade de grande importância. É o que mostra o *site* do ICAT/NIST (*National Institute of Standards and Technology*) [4], onde estatísticas informam que mais de 5.600 vulnerabilidades que incluem somente falhas de software foram relatadas em 2008.

Também, uma pesquisa realizada pelo CSI (Computer Security Institute) [56] mostra que, em 2006, 40% das organizações entrevistadas sofreram de 1 a 5 incidentes em suas aplicações Web [57]. Em 2007 este número subiu para 47% [58]. Uma auditoria realizada pela empresa Acunetix em 2007 [59] revela que 70% das 3.200 aplicações web analisadas nos 3 anos anteriores contêm vulnerabilidades de segurança. A pesquisa mais recente realizada pelo CSI [60] mostra que, em 2008, houve um considerável aumento em casos de ataques que prejudicam *websites* (13,5% em relação a 6% do ano anterior). Uma nova classificação, referente à exploração de navegadores clientes, foi incluída e 11% das organizações entrevistadas relataram ter sofrido esse tipo de ataque. Outro estudo provido pelo Gartner Group revela que 75% dos ataques são baseados em aplicações Web [61].

Diante deste problema de segurança de aplicações Web a indústria vem se dedicando ao desenvolvimento de soluções. Ferramentas denominadas scanners automáticos de vulnerabilidade fazem parte deste conjunto de soluções e são abordados na Seção 1.2.

Vulnerabilidades e ataques a aplicações *Web*. Para investigar falhas no código de aplicações *Web* que levam a vulnerabilidades de segurança é necessário identificar possíveis vulnerabilidades que possam ser exploradas por ataques. Um dos documentos produzidos pela OWASP [48] é o *OWASP top 10* [62]. Este documento representa um consenso sobre as 10 vulnerabilidades de segurança mais críticas presentes em aplicações *Web* e tem como objetivo educar desenvolvedores, arquitetos e organizações a respeito das consequências de ataques que explorem estas vulnerabilidades. O *OWASP top 10* mais recente é de 2010 [62] e, de acordo com este trabalho, as principais vulnerabilidades e ataques a aplicações *Web* são as descritas a seguir.

- **Injeção de SQL**. É a vulnerabilidade mais crítica em 2010 segundo a OWASP top 10 [62]. Ataques a este tipo de vulnerabilidade consistem em confundir o interpretador de SQL para que este execute comandos manipulados. Isto é feito a partir dos dados de entrada do usuário, que são enviados como parte de um comando ou consulta. Assim, um atacante pode ter acesso à base de dados da

aplicação e criar, ler, apagar ou atualizar arbitrariamente qualquer dado disponível para a aplicação. Aplicações se tornam vulneráveis a ataques de injeção de SQL quando entradas de usuários fornecidas a um interpretador não são validadas ou recodificadas. Os atacantes também se beneficiam de mensagens de erro exibidas pela aplicação. Essas mensagens normalmente trazem informações sobre nome da base de dados e nome de tabelas definidas na base de dados, facilitando a construção de comandos válidos para explorar a vulnerabilidade.

Uma variação do ataque a vulnerabilidades do tipo injeção de SQL é chamada *Blind SQL injection* – injeção de SQL às cegas. Este tipo de ataque é idêntico à injeção de SQL exceto pelo fato de, quando um atacante tentar explorar uma aplicação, ao invés de obter mensagens de erro que sejam úteis para construir comandos válidos para o ataque, ele obtém uma página genérica especificada pelo desenvolvedor. Isto torna a exploração de uma vulnerabilidade mais difícil, mas não impossível. Utilizando o ataque às cegas, ainda é possível construir comandos válidos para explorar a vulnerabilidade fazendo uma série de consultas com resultados do tipo verdadeiro ou falso e analisando esses resultados.

▪ ***Cross Site Scripting*** **(XSS)**. Vulnerabilidades do tipo XSS permitem que trechos de código denominados *scripts* sejam executados pela aplicação através do navegador da vítima, permitindo sequestro de sessões de usuário, inserção de conteúdo hostil, roubo de informações pessoais, entre outros danos. Ataques a vulnerabilidades XSS podem ser realizados com sucesso em aplicações que recebam dados originados do usuário e os envie ao navegador sem primeiramente validar estes dados ou recodificar o seu conteúdo de maneira apropriada.

Os ataques que exploram vulnerabilidades XSS podem ser classificados em três tipos: armazenado, refletido ou baseado em DOM (*Document Object Model* - Modelo de Objetos do Documento). Em ataques XSS do tipo armazenado o *script* é injetado na aplicação e armazenado por ela. Então, quando a aplicação é utilizada posteriormente, o *script* é executado. Em ataques XSS do tipo refletido o *script* injetado é imediatamente exibido pela aplicação, ou seja, imediatamente executado (não é armazenado). Em ataques baseados em DOM, o *script* modifica o *Document Object Model,* que é uma interface para acessar, e manipular objetos que representam documentos do tipo HTML e XML. A modificação do DOM é feita no navegador da vítima e o *script* é executado dinamicamente. Ataques desse tipo não requerem que a aplicação seja modificada.

▪ **Falha de autenticação e gerência de sessão**. Ataques que roubam contas ou senha de usuários e administradores, violando a privacidade da aplicação podem ser bem sucedidos quando a aplicação apresenta falhas no procedimento de autenticação e gerência de sessão. Estas falhas normalmente são introduzidas a partir de funções de *logout*, gerência de senhas, *timeout*, recordação de dados de *login*, pergunta secreta, atualização de conta, proteção de credenciais e gerenciamento de sessão.

▪ **Referência direta a objeto de forma insegura.** Este tipo de vulnerabilidade existe quando a aplicação em questão expõe a referência a um objeto implementado internamente ao usuário, como,

por exemplo, arquivos, diretórios, registros da base de dados ou chaves primárias, URLs ou parâmetros de formulários. Os atacantes podem manipular estas referências para acessar outros objetos sem autorização.

▪ ***Cross Site Request Forgery* (CSRF).** Um ataque CSRF força o navegador da vítima, que esteja autenticado em uma aplicação, a enviar uma requisição para uma aplicação *Web* vulnerável. Como a vítima possui uma sessão autenticada, a aplicação *Web* vulnerável, por sua vez, realiza a ação desejada em nome da vítima, conforme requisição recebida. Este tipo de ataque é possível principalmente pelo fato de aplicações *Web* autorizarem requisições baseadas em credenciais submetidas automaticamente, como *cookies* de sessão, endereço de IP de origem, certificados SSL, entre outros. Em algumas literaturas também é conhecido por diversos outros nomes, como XSRF, *Session Riding*, ataque *One-Click* e *Cross Site Reference Forgery* [49].

▪ **Má Configuração de segurança.** Bons níveis de segurança exigem uma configuração segura para a aplicação, servidores (de aplicação, de banco de dados), plataforma, *frameworks*. Todas estas configurações devem ser definidas, implementadas e mantidas, uma vez que muitas não são distribuídas com os padrões seguros. Isso inclui manter todos os softwares em dia, incluindo todas as bibliotecas de código usadas pelo aplicativo.

▪ **Armazenamento criptográfico inseguro**. Ataques para violar integridade e confidencialidade podem ser efetivos em aplicações que não criptografam informações e credenciais ou criptografam estes dados com algoritmos ineficientes, denominados, na literatura, como algoritmos “fracos”.

▪ **Comunicações inseguras.** Semelhante à vulnerabilidade anterior, comunicações inseguras também permitem ataques que violam integridade e confidencialidade dos dados de determinada aplicação. Neste caso, porém, a criptografia se aplica ao tráfego da rede.

▪ **Falha ao restringir acesso a URLs.** Frequentemente, uma aplicação protege suas funcionalidades críticas somente pela supressão de informações como *links* ou URLs para usuários não autorizados. Atacantes podem fazer uso desta fragilidade para acessar e realizar operações não autorizadas por meio do acesso direto às URLs.

▪ **Proteção insuficiente na camada de transporte**. Aplicações frequentemente falham ao autenticar, criptografar e proteger a confidencialidade e integridade do tráfego de uma rede sensível. Quando isso acontece, essas aplicações normalmente suportam algoritmos fracos de criptografia, usam certificados expirados ou inválidos ou não os usa de maneira correta.

▪ **Redirecionamentos e encaminhamentos inválidos**. Aplicações Web frequentemente redirecionam e encaminham seus usuários para outras páginas e usam dados não confiáveis para determinar as páginas de destino. Sem validação apropriada, atacantes podem redirecionar vítimas a páginas com vírus ou usar os encaminhamentos para acessar páginas não autorizadas.

3.3 Resumo

Este capítulo apresentou uma visão geral sobre segurança, abordando seus aspectos em diferentes contextos, como segurança de redes, de servidores e de protocolos de comunicação. Foram descritos os principais aspectos nos quais a segurança, de um modo geral, se baseia (autenticação, autorização, auditoria, confidencialidade, integridade e disponibilidade). Também foram descritas as principais vulnerabilidades de segurança de redes e servidores que podem ser exploradas por ataques, bem como algumas contra medidas para evitar ou amenizar as consequências destes ataques. Citaram-se trabalhos em que protocolos de redes com e sem fio estão sendo testados contra vulnerabilidades de segurança.

Para segurança de aplicações *Web*, que é o enfoque desta dissertação, foram descritas categorias que representam os principais pontos de fragilidade. Dentre estes pontos de fragilidades selecionou-se, de acordo com o objetivo do trabalho, a qualidade do código, pois falhas residuais presentes no código em função de uma má codificação podem deixar a aplicação vulnerável do ponto de vista de segurança. Em seguida, foram relatadas pesquisas que demonstram a importância da validação da segurança de aplicações *Web* em fase operacional. Ao final foram descritas as principais vulnerabilidades que afetam aplicações *Web* atualmente.

4 Capítulo 4

Injeção de Falhas e Modelos de Ataques

Este capítulo descreve duas técnicas de apoio utilizadas para estabelecer a abordagem que é proposta na presente dissertação: injeção de falhas e modelos de ataques. Técnicas de injeção de falhas tem se mostrado uma poderosa aliada na avaliação do comportamento de um sistema na presença de falhas, dando ao desenvolvedor maior agilidade nos testes e uma visão mais segura do correto funcionamento do sistema, antes que esse seja colocado em produção. Modelagens de ataques têm sido utilizadas para avaliar e verificar a segurança de sistemas, redes, aplicações e protocolos de comunicação.

4.1 Injeção de falhas

Sabe-se que atualmente usuários de sistemas computacionais exigem níveis de dependabilidade cada vez maiores. Pode-se dizer que a importância de sistemas confiáveis tem crescido e, com isso, cresceu a importância de testar esses sistemas. Assim, algumas técnicas de teste de sistemas confiáveis foram desenvolvidas. Uma técnica que é bastante utilizada é a Injeção de Falhas [16]. Esta técnica permite observar o comportamento do sistema na presença de falhas para verificar se continua a operar como esperado [63]. Dessa forma, procura-se entender como as diferentes categorias de falhas afetam o sistema. Esse conhecimento pode ser usado para prevenir falhas como também para avaliar e aprimorar os mecanismos de tolerância a falhas, medindo sua eficiência (cobertura). Sendo assim, essa técnica pode proporcionar maior confiabilidade com respeito ao funcionamento do sistema.

Numa época em que uma falha de computador é cada vez menos tolerada pelos usuários, a confiabilidade que a validação por injeção de falhas pode trazer é vital. O desenvolvedor pode ter maior confiança na correta operação de seu software e, portanto, o software pode ser mais rapidamente validado e entregue. Por esse motivo, muitos desenvolvedores têm voltado sua atenção à Injeção de Falhas, em particular a uma de suas principais implementações, a injeção de falhas por software, isto é, injeção de falhas realizada em trechos especiais de código [63]. A seguir são apresentadas definições de falhas de hardware e falhas de software, técnicas e métodos de injeção de falhas, ambiente de injeção de falhas e ferramentas que implementam essa técnica.

Modelo de falhas. Uma falha é um fenômeno que pode levar a um desvio de comportamento de um componente de software ou de hardware em relação à sua função original [64]. Um dos aspectos mais importantes em relação à técnica de injeção de falhas é o modelo de falhas, que

especifica a natureza exata das falhas a serem injetadas. Outro aspecto a se considerar é a classe das falhas, que se refere à origem e a forma pela qual a falha se manifesta. Assim, um modelo de falhas inclui a informação a respeito das ações orientadas à reprodução de falhas de uma determinada classe. Carreira *et al.* [65] enumeram as classes mais relevantes para sistemas de computadores: falhas de operador, falhas de ambiente, falhas de hardware e falhas de software.

Falhas de operador são ações incorretas por parte do usuário ou administrador do sistema. **Falhas de ambiente** estão relacionadas com acontecimentos no ambiente externo no qual o sistema está inserido (por exemplo, falta de energia, incêndios, etc). Uma **falha de hardware** é causada por componentes mal construídos ou por condições ambientais que o degradem, impedindo seu funcionamento correto. Segundo Carreira *et al.* [65], as falhas de hardware podem ser classificadas como permanentes, transientes ou intermitentes. Falhas **permanentes** são resultados de danos irreversíveis ao componente de hardware; falhas **transientes** são disparadas por alterações temporárias em condições ambientais; e falhas **intermitentes** são causadas por instabilidade do hardware. As **falhas de software**, por sua vez, são causadas por especificação, projeto ou construções incorretas de um sistema, levando a falhas de codificação e incompatibilidade com outros componentes ou com o ambiente de execução [66]. Recentemente, tem havido preocupação com as consequências de falhas de software, pois são reconhecidas como a principal causa de defeitos computacionais, como mostram os trabalhos de Sullivan e Chillarege [25], Kalyanakrishnam *et al*. [67], Lee e Yier [68]. Resumidamente, Kalyanakrishnam *et al*. [67] apresentam resultados de uma análise onde constatam que falhas de software se propagam em uma rede local WindowsNT e fazem com que as máquinas sejam reiniciadas. Lee e Yier [68] fazem uma avaliação de dependabilidade do sistema operacional *Tanden's Guardian*; falhas de software foram consideradas a principal causa de defeitos que invocam o processo de backup. Devido à grande complexidade dos produtos de software atuais, a quantidade de falhas tende a aumentar. Estas falhas podem até mesmo tornar a aplicação vulnerável do ponto de vista de segurança [3].

A completa eliminação das falhas em um software é uma tarefa difícil ou mesmo impossível de ser atingida [69][70]. Em muitos casos, esperar pela ativação natural destas falhas não é viável, uma vez que essa ativação raramente acontece (caso contrário, as falhas seriam facilmente encontradas e eliminadas na fase de teste). Identificar falhas de software que levam a vulnerabilidades é uma ajuda muito grande para validar sistemas onde a segurança é um requisito indispensável, pois assim é possível acelerar a ocorrência de erros e defeitos pela ativação dessas falhas e analisar o comportamento da aplicação diante de ataques que possam vir a explorar a vulnerabilidade.

Técnicas de injeção de falhas podem ser usadas para avaliar tanto o efeito de falhas de hardware como de falhas de software [71]. A seguir são descritas três categorias nas quais se podem dividir as técnicas de injeção de falhas: injeção de falhas por simulação (utilizam um modelo do sistema alvo), injeção de falhas por hardware (injetam falhas físicas no sistema alvo) e injeção de falhas por software (permite emular falhas - tanto falhas de hardware quanto de software – por software).

Injeção de Falhas por simulação. Esta técnica consiste na introdução de falhas em um modelo do sistema alvo e é útil para avaliar a dependabilidade do sistema nas primeiras fases do desenvolvimento. Injeção de falhas por simulação permite injetar falhas bastante precisas e coletar informações detalhadas sobre seus efeitos.

No entanto, este tipo de injeção apresenta algumas desvantagens. Como o teste é sobre um modelo do sistema, os resultados obtidos serão sobre este modelo e não sobre o sistema final. Então, um dos problemas nesse tipo de injeção é a construção de um modelo que seja fiel à estrutura e/ou comportamento do sistema. Outro problema é a aplicação de modelos de falhas representativos. Em se tratando de componentes com larga escala de integração, nem sempre é possível uma descrição detalhada de sua estrutura interna para que se possa obter a precisão necessária.

Injetar falhas em um protótipo ou versão operacional de um sistema (falhas no hardware ou no software) é útil em complementação à injeção de falhas por simulação.

Injeção de Falhas por hardware. Esta técnica é aplicável nas fases finais do desenvolvimento de um sistema e consiste em introduzir falhas em um sistema físico, permitindo verificar a eficácia de mecanismos de tolerância a falhas implementados em hardware. Para injetar falhas, são necessários equipamentos extras, como produtos de hardware específicos, projetados especialmente para este propósito. Segundo Hsueh *et al.* [63], o método de injeção de falhas por hardware pode ser dividido em duas categorias: injeção de falhas por hardware **com contato** e injeção de falhas por hardware **sem contato**. Na primeira categoria, o injetor tem contato físico direto com o sistema alvo produzindo, por exemplo, alterações de corrente e voltagem. Na segunda categoria, o injetor não possui contato físico direto com o sistema alvo. Uma fonte externa produz algum fenômeno físico, como interferência eletromagnética, por exemplo.

As principais vantagens deste método são: permitir a validação de um protótipo que é próximo do produto final; permitir a observação do efeito de falhas reais sobre o sistema e permitir a validação global do sistema, integrando tanto hardware quanto software.

Uma das principais desvantagens é o não-determinismo, no sentido que não se pode determinar nem o número e nem a localização dos erros que são gerados, impossibilitando a capacidade de repetição dos experimentos. Outra desvantagem é que a injeção pode danificar componentes e equipamentos, de forma que os custos dos testes podem ser altos e a obtenção de resultados nem sempre é possível.

Injeção de Falhas por software. Esta técnica procura injetar falhas lógicas em aplicações e sistemas operacionais. O objetivo é alterar o estado do sistema alvo pelo controle de um software, emulando, dessa forma, tanto as consequências de falhas de hardware quanto de falhas de software. Para isso, não é preciso hardware especial e os testes podem ser mais facilmente controlados e observados. Devido a essas vantagens, Injeção de Falhas por software tem se tornado mais popular entre os desenvolvedores de sistemas tolerantes a falhas.

A injeção de falhas por software pode ser usada para simular tanto falhas internas (falhas de software), como falhas externas (falhas de sistemas externos ao software, mas conectados a este através de interfaces) [72]. Falhas **internas** representam falhas de projeto e implementação, tais como, variáveis ou parâmetros que estão errados ou não inicializados, atribuições incorretas ou verificações incorretas de condições. Falhas **externas** representam todos os fatores externos que não estão relacionados com falhas no código alvo, mas alteram o estado do software de algum modo. O software de injeção da falha interrompe ou altera de modo controlado a execução do sistema alvo por meio de algum mecanismo e executa o código de injeção, que emula falhas em diferentes componentes de acordo com o modelo de falhas usado; por exemplo, falhas de memória (alteram o conteúdo de posições de memória – programa ou dados), falhas de processador (afetam o conteúdo de registradores, resultado de cálculos, fluxo de controle ou código de instruções), falhas de comunicação (afetam mensagens que são transmitidas através de um canal de comunicação que podem ser perdidas, alteradas, duplicadas ou entregues com atraso).

A principal desvantagem da técnica de injeção de falhas por software é em relação ao fato do injetor ser um código a ser executado. A execução do injetor pode fazer com que o sistema se comporte de forma diferente, ou seja, injetores podem ser intrusivos e alterar, muitas vezes de maneira bastante significativa, o desempenho do sistema alvo. Dessa forma, os resultados obtidos podem ser afetados, devido ao *overhead* causado pela inserção do código do injetor de falhas. As ferramentas de injeção, além de interferirem no sistema para introduzir falhas desejadas, precisam monitorar o sistema e determinar se a falha e os mecanismos de tolerância a falhas foram ativados, coletando dados que auxiliem no diagnóstico dos erros apresentados. Uma maneira de se reduzir essa interferência é usar técnicas híbridas, empregando ferramentas de software e de hardware.

A **G-SWFIT** (*Generic Software Fault Injection Technique*) [22] é uma técnica de injeção de falhas por software que permite inserir diretamente no código executável de um programa uma sequência de instruções que emulam falhas de software que se quer injetar. As modificações inseridas no código são idênticas àquelas que seriam geradas pelo compilador, caso o código fonte tivesse uma falha cometida pelo programador. A ferramenta localiza no código de máquina a tradução das estruturas de programação e, dessa forma, modifica estas estruturas emulando falhas apropriadas para cada um dos tipos.

Para injetar as falhas, a ferramenta utiliza um método de dois passos; o primeiro identifica o local onde as falhas devem ser injetadas; o segundo efetivamente injeta as falhas. Para identificar o local de injeção é feita uma varredura (*scan*) automática no código executável. O resultado dessa varredura é um mapeamento que identifica os pontos de injeção que são apropriados para emular cada falha específica. O processo de varredura é guiado por uma biblioteca de operadores que são previamente definidos de acordo com o modelo de falhas. Cada operador descreve um tipo de falha específico e compreende dois componentes: um padrão de busca e uma definição de código em baixo nível. A injeção de falhas no passo dois é muito simples e muito pouco intrusiva, uma vez que

cada local de injeção já foi identificado no passo um. Como a G-SWFIT trabalha no nível do código de máquina ela é independente do código fonte.

Métodos de injeção de falhas. Diferentes métodos são usados para injetar falhas por software. Uma maneira de categorizar essas formas é definir quando as falhas são injetadas, se em tempo de compilação ou em tempo de execução [63].

Na **injeção em tempo de compilação**, falhas são introduzidas no código fonte ou *assembly* do programa alvo, e consiste na alteração de instruções do programa. Na execução, quando uma instrução modificada é executada, a falha é ativada. Este método não precisa de software extra durante a execução para injetar falhas. Como resultado, não causa perturbação no sistema alvo durante a execução, mas exige a criação de várias versões do software.

Em **injetores em tempo de execução**, código extra é necessário para injetar falhas e monitorar seus efeitos, respectivamente, um injetor de falhas e um monitor. Além disso, também é requerido um mecanismo para disparar a injeção de falhas. Pode ser mais intrusivo e alterar o desempenho do sistema alvo, além de apresentar resultados alterados.

A escolha do método mais adequado depende de vários fatores, entre eles: acessibilidade dos componentes nos quais as falhas serão injetadas, o nível aceitável de perturbação do programa alvo, recursos oferecidos pelo ambiente de execução e objetivo dos experimentos.

Ambiente de Injeção de Falhas. A sequência de trabalhos de Arlat *et al.* [73][16][74] apresenta metodologias de validação de dependabilidade e mecanismos de tolerância por procedimento de injeção de falhas. Arlat *et al.* [73] abordam a validação da dependabilidade de sistemas tolerantes a falhas por meio da validação de mecanismos de tolerância a falhas. Falhas são injetadas no nível físico de um protótipo composto por hardware e software e o comportamento do mecanismo de tolerância a falhas é analisado. Arlat *et al.* [16] é uma extensão do trabalho inicial [73] para prover um maior conjunto de resultados. Dois sistemas são avaliados: um subsistema computadorizado para controle de bloqueio de transporte ferroviário e um sistema distribuído de comunicação *multicast.* Arlat *et al.* [74] descrevem um método baseado em injeção de falhas para avaliar dependabilidade. Esse método estabelece uma ligação entre a avaliação experimental do processo de tolerância a falhas e o processo de ocorrência das falhas, obtendo medidas experimentais e analisando-as. Nos três trabalhos [73][16][74], os autores utilizam o modelo FARM para caracterizar a injeção de falhas. Este modelo é constituído por 4 conjuntos: O conjunto *F,* que define o conjunto de falhas a serem injetadas; o conjunto *A,* que define o domínio da ativação do sistema e especifica a funcionalidade que deve ser exercitada; o conjunto *R,* que especifica as observações a serem efetuadas e quais as características do sistema mais importantes para serem observadas; e o conjunto *M* que especifica o modelo a ser aplicado às observações do conjunto *R* para obter um conjunto de medidas, como por exemplo, tempo médio entre defeitos. O conjunto *F* pode ser definido como *faultload*, e nestes trabalhos os autores já demonstram a preocupação em definir um *faultload* independente, onde a definição do conjunto de falhas seja mais realista possível.

A definição do conjunto *F* [16] parte do princípio que, para remoção e previsão de falhas o conjunto *F* deve representar falhas específicas identificadas de acordo com a especificação do sistema e identificar uma grande quantidade de falhas que correspondam a uma distribuição estatística representativa.

Uma visão mais recente da experiência da injeção de falhas pode ser descrita pelos componentes: injetor de falhas, gerador de carga (*workload*), monitor e controlador. O trabalho de Hsueh *et al.* [63] apresenta um ambiente básico para injeção de falhas com estes componentes, incluindo um componente para coletar e analisar dados. A Figura 4.1 mostra como é este ambiente.

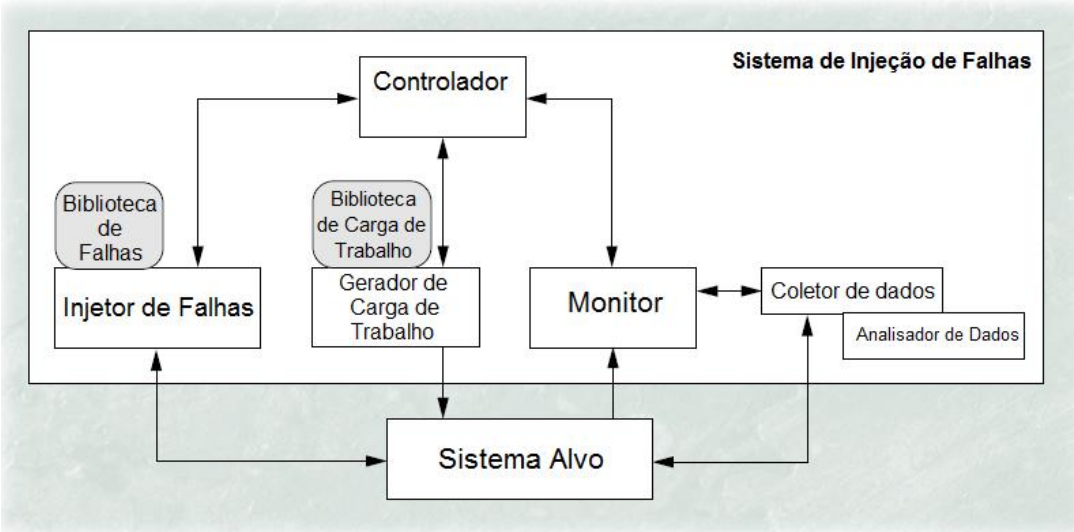


Figura 4.1. Ambiente básico de injeção de falhas (tradução da autora) [58]

O funcionamento do ambiente ilustrado na Figura 4.1 ocorre da seguinte forma: enquanto o sistema alvo executa os trabalhos provenientes do componente gerador de carga de trabalho, o injetor de falhas insere a carga de falhas no sistema alvo. A carga de trabalho é um conjunto de tarefas que o sistema alvo executaria no ambiente de trabalho normal. E a carga de falhas é um conjunto de falhas que o sistema pode sofrer durante a execução do trabalho. O monitor tem a função de coletar todos os dados gerados no teste para que possam ser analisados pelo componente analisador de dados. Por fim, o controlador serve para controlar o experimento [63].

Tanto a biblioteca de falhas quanto a biblioteca de carga de trabalho são componentes distintos e separados dos seus injetores pois, assim, é garantida a flexibilidade e portabilidade dos conjuntos que serão utilizados. O injetor de falhas pode suportar diferentes tipos de falhas vindos da biblioteca de falhas, ou seja, diferentes *faultloads*. Reforça-se então a idéia de que a definição de um *faultload* independente e realista é fundamental para o sucesso da validação do sistema em questão.

O modelo FARM e o sistema de injeção de falhas apresentado serviram como base para construção de várias ferramentas de injeção de falhas. Algumas são descritas na seção seguinte.

Ferramentas de injeção de falhas. Técnicas e ferramentas injetoras de falhas são elementos importantes para análise de dependabilidade e validação de sistemas, especialmente para sistemas em que a segurança é um requisito indispensável. Existem vários injetores de falhas, inclusive para falhas de hardware. Denominadas HWIFI (*Hardware Implemented Fault Injection*), estas ferramentas tornaram-se sucessivamente mais difíceis de construir em função do aumento crescente da tecnologia e diminuição do tamanho dos componentes de hardware. Então, os injetores de falhas de software SWIFI (*Software Implemented Fault Injection*), que permitem modelos de falhas mais complexos, se mostraram cada vez mais vantajosos, versáteis e de menor custo. Como o enfoque desta dissertação é a injeção de falhas por software, algumas técnicas e ferramentas nesta linha são abordadas nesta seção.

ORCHESTRA [75] é uma ferramenta desenvolvida para testar aplicações distribuídas e protocolos de comunicação. Ela permite inserir uma camada de injeção de falhas denominada PFI (*Protocol Fault Injection Layer*) entre a camada do protocolo testado e as camadas inferiores da pilha do protocolo, possibilitando a filtragem e manipulação das mensagens trocadas entre os participantes. Mensagens podem ser atrasadas, perdidas, reordenadas, duplicadas, modificadas; e novas mensagens podem ser introduzidas espontaneamente no sistema testado para conduzi-lo a um determinado estado desejado.

ComFIRM (*Communication Fault Injection through OS Resources Modification*) [76] permite modificar o sistema operacional sobre o qual o sistema sob teste está sendo executado para injetar falhas. A ferramenta foi desenvolvida para ser usada com o sistema operacional Linux e seu objetivo é testar mecanismos de tolerância a falhas relacionados com a comunicação dentro de um sistema distribuído. Isto é feito pela troca ou remoção de algumas mensagens trocadas entre nós do sistema, injetando falhas de comunicação para testar mecanismos de comunicação tolerantes a falhas. A ComFIRM injeta falhas a partir do sistema operacional e é capaz de fazê-lo com baixa perturbação do sistema sob teste. É dependente do sistema operacional Linux, podendo ser usada com sistemas alvo que sejam executados sobre esse sistema operacional, ficando dependente da portabilidade do Linux. A ferramenta não trata falhas que não sejam referentes a transmissão de mensagens.

As primeiras versões da ComFIRM exigiam a alteração do código do núcleo do sistema operacional hospedeiro. Apesar de eficiente, essa estratégia é intrusiva e compromete a portabilidade do injetor. A sua última versão já adotava o conceito de módulo carregável no núcleo. Esse conceito foi aproveitado e Drebes *et al.* [77] propuseram a evolução da ComFIRM, provendo um melhor mecanismo para a descrição de cargas de falhas. Os autores apresentam a ferramenta **FIRMAMENT** (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*), que é um injetor localizado no núcleo do sistema operacional Linux e que

permite injeção por software de falhas de comunicação no protocolo IP ou em qualquer outro protocolo construído sobre IP.

A FIRMAMENT emprega o conceito de *faultlet,* que é usado para especificação de cenários de falhas. Um *faultlet* é executado em cada pacote que cruza o fluxo de comunicação, podendo atrasá-lo, alterar seu conteúdo, duplicá-lo, descartá-lo ou aceitá-lo. Além das ações sobre pacotes, um *faultlet* opera sobre variáveis de estado do fluxo, que podem ser usadas para alterar o valor dos dados do pacote.

Os tipos de falhas injetáveis por FIRMAMENT são: omissão de envio e recebimento de mensagens, colapso de canal, atraso de mensagens e falhas bizantinas (i.e., falhas arbitrárias). Duas idéias básicas norteiam o injetor: possibilitar a modificação da configuração e da descrição dos cenários de falhas de um experimento durante sua execução e representar cargas de falhas por meio de regras simples, que não sobrecarreguem o processamento de mensagens.

XCEPTION *(Software Fault Injection and Monitoring in Processor Functional Units*) [20] é uma ferramenta de injeção de falhas por software que permite até mesmo emular falhas de hardware como, por exemplo, falhas de barramento de endereço, falhas de ponteiro de instrução, entre outras. Para isso, a Xception utiliza características especiais dos processadores para injetar falhas e monitora a ativação destas falhas, bem como seus impactos no sistema sob teste. A simulação das falhas se dá por instruções destinadas a realizar depuração e testes de desempenho dos chips, sendo disparadas pela ocorrência de algum evento específico e, consequentemente, causando uma exceção de hardware.

A principal vantagem desta ferramenta é em relação à perturbação no sistema alvo, que é desprezível. Outra vantagem é que não é necessário nenhum código extra para fazer a injeção de falhas e o monitoramento.

A ferramenta **JACA** [78] se baseia em reflexão computacional para injetar falhas e monitorar o sistema. Foi projetada para seguir um padrão de implementação definido para ferramentas de injeção de falhas. Oferece mecanismos para se injetar falhas de interfaces em sistemas orientados a objetos escritos na linguagem Java e é bastante adequada para testes de robustez. A ferramenta não necessita do código fonte da aplicação para realizar as injeções e esta independência do código fonte é um aspecto importante, pois permite a validação de sistemas compostos por vários componentes de terceiros mesmo quando o código fonte não está disponível.

Além do injetor de falhas propriamente dito, a Jaca possui uma interface gráfica cujo objetivo é facilitar a especificação das características do sistema e das falhas a serem injetadas. Para se especificar as falhas é preciso informar os pontos de injeção, o modo de ativação, o padrão de repetição das injeções e quando as injeções devem ser iniciadas. Pode alterar valores de parâmetros e retornos de métodos, bem como atributos públicos.

O projeto inicial da Jaca previa a injeção em todos os tipos básicos da linguagem Java. Na maioria dos sistemas atuais quase todos os métodos apresentam objetos como parâmetros, valores de retorno e atributos. Nesses sistemas a eficiência da Jaca era quase nula, pois quase nenhuma injeção podia ser feita. Para contornar essa limitação a Jaca foi estendida para ser capaz de alterar objetos. Para se alterar objetos pode-se criar um novo objeto ou chamar algum método daquele objeto. Sendo assim, o usuário pode criar variáveis, chamar construtores, chamar métodos que utilizam essas variáveis e chamar métodos estáticos.

Outra extensão feita na Jaca foi a versão criada para se injetar falhas de comunicação, o que permitiu a validação de sistemas distribuídos [79]. Nessa versão, a injeção de falhas é feita em um nível acima da pilha de protocolos de comunicação do sistema operacional, o que preserva a portabilidade da ferramenta. O modelo de falhas para o protocolo UDP *(User Datagram Protocol*) contempla as falhas de omissão, temporização e colapso. Para o protocolo TCP *(Transfer Control Protocol)* inclui apenas falhas de colapso, pois as demais já são mascaradas pelo protocolo e não atinge a aplicação, alvo da ferramenta.

As vantagens da Jaca são em relação à intrusão sobre o sistema alvo e à portabilidade. A intrusão do sistema alvo é desprezível e a ferramenta apresenta boa portabilidade, pois pode ser utilizada em qualquer ambiente que tenha a máquina virtual Java (JVM) padrão instalada. A principal desvantagem é a limitação quanto ao tipo de falhas que pode emular, pois seu enfoque em falhas de interface a torna limitada.

J-SWFIT (*Java Software Fault Injection Tool)* [17]. A G-SWFIT teve seu primeiro estudo desenvolvido utilizando a linguagem de programação C, com algumas considerações sobre uma extensão da técnica para qualquer outra linguagem. A ferramenta J-SWFIT estende a técnica G-SWFIT a fim de englobar particularidades da programação orientada a objetos, adaptando-a para códigos escritos em Java e mantendo-se a independência da disponibilidade do código fonte para realizar a emulação das falhas de software. Os operadores de emulação de falhas já conhecidos são mantidos e novos operadores específicos do paradigma de orientação a objetos e da linguagem Java são incluídos. O funcionamento da J-SWFIT consiste em: encontrar locais onde falhas específicas possam existir; injetar cada falha independentemente; executar o sistema com a falha presente; monitorar os resultados; e, ao final, comparar o comportamento na presença e na ausência de cada falha. A arquitetura foi proposta em um nível de abstração que pode ser facilmente entendida e estendida.

4.2 Modelos de Ataques

Os objetivos e os métodos de um atacante possuem um papel central em praticamente todas as considerações de segurança. Modelos de ataques permitem representar de maneira clara e

organizada as ações que podem ser desenvolvidas por um atacante para realizar um ataque com sucesso. Segundo Steffan e Schumacher [80], modelos de ataques podem ser úteis em: descobrir vulnerabilidades em sistemas novos; evitar vulnerabilidades durante o desenvolvimento do software; avaliar implementações existentes para vulnerabilidades conhecidas. Estes modelos podem representar: os métodos de ataque gerais e específicos contra um sistema, as propriedades de um sistema e as pré-condições para que um ataque tenha sucesso. Também, devido a sua representação gráfica compacta, modelos de ataques não precisam de explicações detalhadas e não exigem conhecimento prévio em métodos de modelagem de ataques.

Árvore de ataques [33] é uma estrutura que descreve os possíveis ataques a um sistema de forma organizada para facilitar a análise de segurança. A árvore de ataques representa os passos para se realizar um ataque, seus objetivos e interdependências, e também pode ser usada para representar e calcular probabilidades, riscos, custos ou outras ponderações. Elas utilizam lógica booleana (E/OU) para descrever possíveis relações entre os passos de um ataque.

Nas árvores de ataques o nó raiz representa o objetivo final do atacante. Cada nó filho representa sub-objetivos que precisam ser realizados para que o objetivo do nó pai tenha sucesso. Nós pais podem estar relacionados com seus filhos por uma relação de tipo OR ou de tipo AND. Em uma relação do tipo OR, se qualquer um dos sub-objetivos dos nós filhos é realizado então o nó pai é bem-sucedido. Com uma relação do tipo AND, todos os sub-objetivos dos nós filhos devem ser realizados para que o nó pai seja bem sucedido. As folhas da árvore, ou seja, os nós que não são mais decompostos, representam ações do atacante.

Árvores de ataques podem ser representadas graficamente ou textualmente. A Figura 4.2 apresenta uma relação do tipo AND. Isto significa que para que o objetivo G0 seja atingido é necessário que o atacante atinja todos os sub-objetivos, de G1 até Gn.

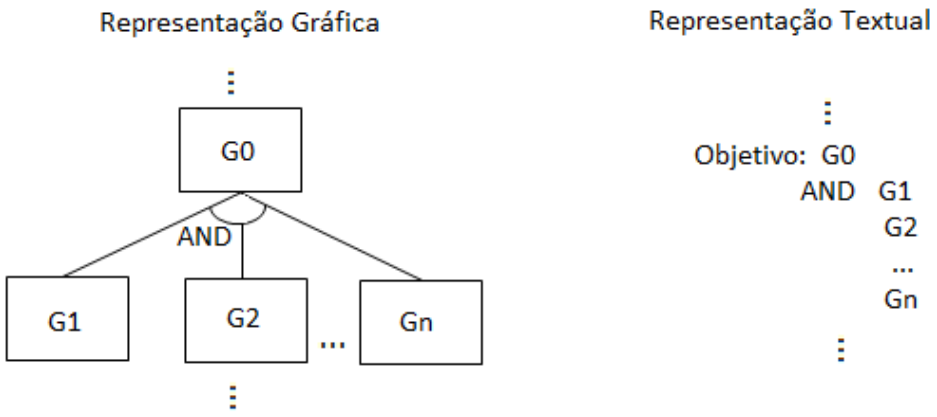


Figura 4.2. Representação gráfica e textual de relação do tipo AND entre nós da árvore.

De forma similar, a Figura 4.3 representa uma relação do tipo OR. Significa que para atingir o objetivo G0 o atacante necessita atingir pelo menos um dos sub-objetivos entre G1 e Gn.

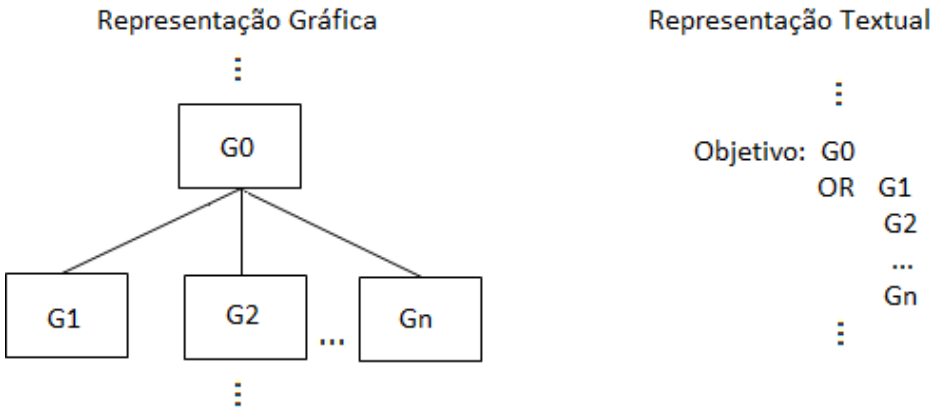


Figura 4.3. Representação gráfica e textual de relação do tipo OR entre nós da árvore.

Cenários de ataque. Cenários de ataque individuais podem ser gerados percorrendo a árvore em profundidade, ou seja, um cenário de ataque é um caminho da raiz até uma folha da árvore, assim gerando uma combinação mínima de eventos das folhas, no sentido que se qualquer nó for omitido do cenário de ataque então o objetivo da raiz não é alcançado. O conjunto completo de cenários de ataque de uma árvore mostra todos os ataques que estão disponíveis para um atacante.

É possível associar atributos aos nós, como custo, probabilidade ou valores lógicos. Assim, é possível selecionar cenários de ataques que são mais prováveis de ocorrer ou de menor custo, por exemplo. Para obter estes cenários, é necessário propagar os valores dos atributos a partir dos nós folhas até o nó raiz da árvore.

Para criar árvores de ataques, primeiro deve-se identificar o objetivo do ataque. Cada objetivo forma uma única árvore, embora estas possam compartilhar sub-árvores e nós. Então, deve-se pensar em todas as possíveis maneiras de se atacar para atingir o objetivo e adicionar os passos à árvore, bem como os atributos (por exemplo, custo do ataque em Reais (R$)). Esta árvore pode ser utilizada para avaliar se determinado sistema é vulnerável ao tipo de ataque representado e também para tomada de decisões sobre aspectos de segurança.

A Figura 4.4 traz uma ilustração para o uso de árvore de ataques. É uma versão simplificada do exemplo dado no trabalho de Schneier [33]. O exemplo considera um ataque a um cofre. Neste exemplo, para alcançar o objetivo final de “Abrir o cofre”, o atacante deveria “Forçar a fechadura” OR “Aprender a combinação do cofre”. Para “Aprender a combinação do cofre” ele deveria “Achar a combinação escrita” OR “Obter a combinação do cofre”. Uma maneira de “Obter a combinação do cofre” é por “Escuta”, o qual requer um equipamento especial. A “Escuta” deve ser feita de modo que o atacante possa “Ouvir uma conversa” AND “Escutar o dono do cofre dizer a combinação”. Atacantes não podem alcançar o objetivo a menos que os sub-objetivos sejam satisfeitos.

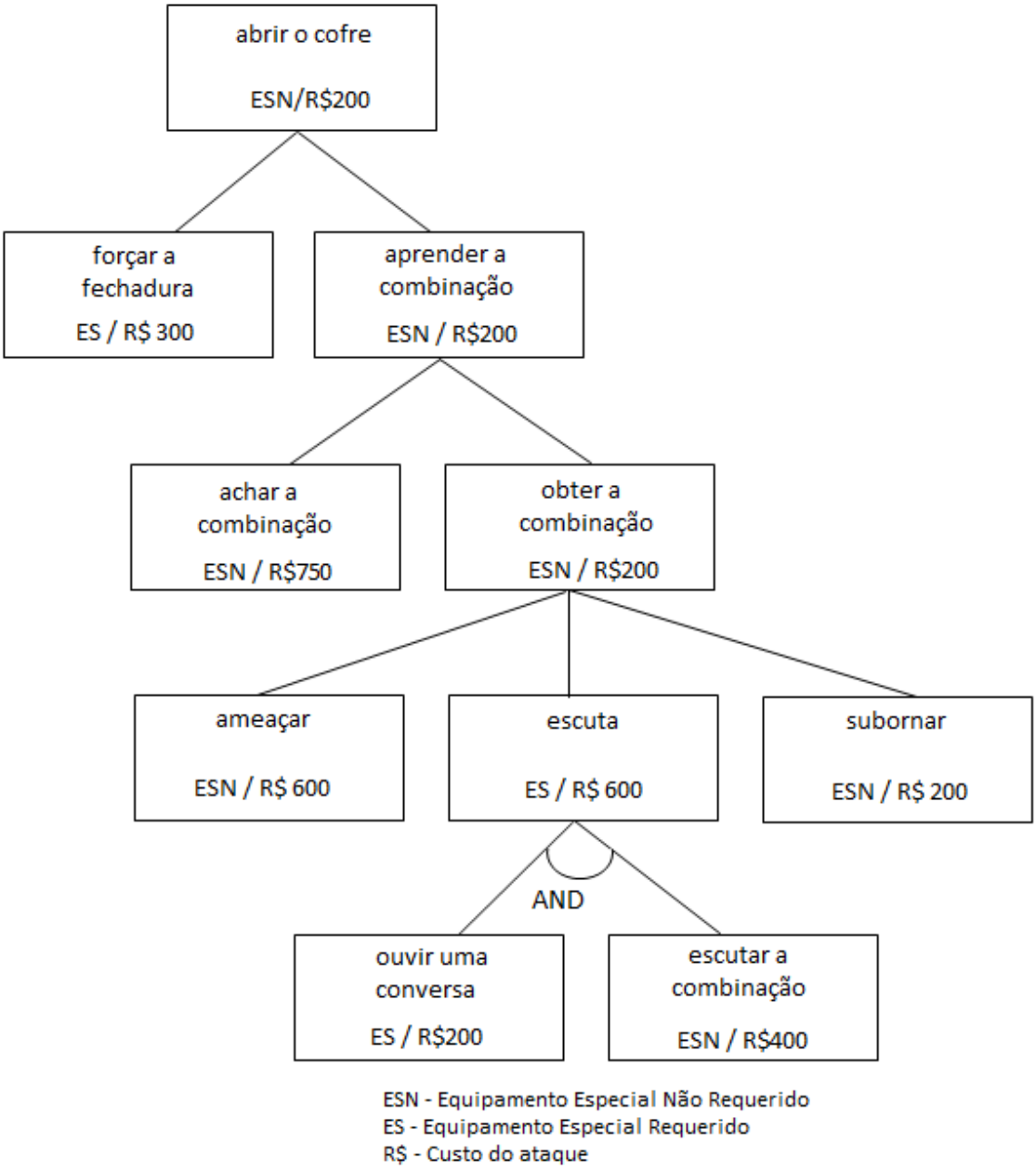


Figura 4.4. Árvore de ataques para abrir o cofre (tradução de Morais [81])

Na árvore de ataques da Figura 4.4 o cenário de ataque < “Subornar” > é o ataque mais barato (R$ 200) que não requer nenhum equipamento especial. Outro possível cenário é o que já foi explicado <“Ouvir a conversa”, “Escutar o dono do cofre dizer a combinação”>, que possui um custo médio (R$ 600) e requer equipamento especial.

A Figura 4.5 ilustra quatro dos vários possíveis diferentes cenários de ataques, onde, para cada cenário, os nós de cor cinza representam o objetivo e sub-objetivos que devem ser cumpridos.

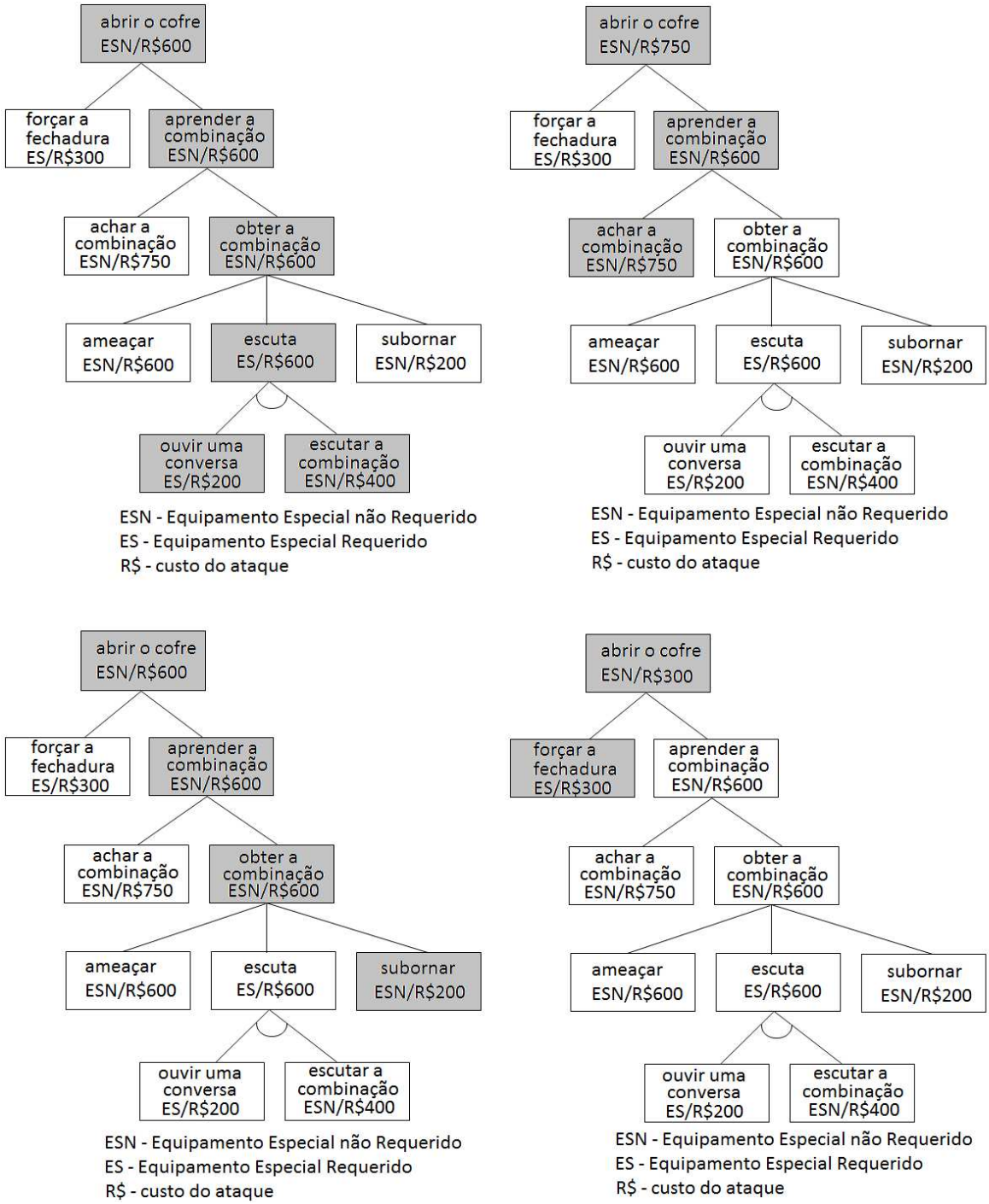


Figura 4.5. Possíveis cenários de ataques selecionados a partir da árvore de ataques para abrir o cofre.

Esse modelo de ataques é bastante útil para gerar cenários de ataque para testes de segurança, pois os cenários que correspondem a ataques reais podem ser gerados de acordo com os custos ou a frequência dos ataques na vida real.

Padrões de ataque. Segundo Moore *et al.* [82], padrão de ataque (*attack pattern* em inglês) é uma representação genérica de um ataque malicioso e intencional que geralmente ocorre em contextos específicos. Cada padrão de ataque contém: i) o objetivo do ataque especificado pelo padrão; ii) uma lista de pré-condições para que aconteça; iii) os passos para executar o ataque; iv) uma lista de pós-condições se o ataque foi bem-sucedido. As precondições incluem suposições feitas sobre o atacante ou o estado do sistema que são necessárias para que o ataque tenha sucesso, como por exemplo, habilidades, recursos, acesso, ou conhecimento que um atacante deve possuir, e o nível de risco que ele deve estar disposto a tolerar. As pós-condições incluem o conhecimento adquirido pelo atacante e alterações do estado do sistema que resultaram dos passos do ataque bem-sucedido.

A Figura 4.6 traz um exemplo de padrão de ataque para vulnerabilidade de segurança do tipo buffer overflow, que é uma condição anômala onde a aplicação tenta armazenar dados além dos limites de um buffer de tamanho fixo.

Uma árvore de ataques pode ser refinada desde o nó raiz usando uma combinação de extensões manuais e aplicação de padrões de ataque. Extensões manuais dependem enormemente do conhecimento de segurança da pessoa que está construindo a árvore de ataques. A aplicação de padrões de ataque também depende de tal conhecimento, mas em menor proporção, pois muito desse conhecimento está implementado na biblioteca de padrões de ataque.

Padrão de Ataque de *Buffer Overflow*:

Objetivo: Explorar a vulnerabilidade de *buffer overflow* para executar uma função maliciosa no sistema alvo.

Precondição: Atacante pode executar programas no sistema alvo.

Ataque:

AND 1. Identificar um programa executável no sistema alvo suscetível a vulnerabilidade de *buffer overflow*.

2. Identificar o código que irá executar a função maliciosa quando ele for executado com privilégio de programa.

3. Construir o valor de entrada que forçará com que o código esteja no espaço de endereço do programa.

4. Executar o programa de uma maneira que ele pule para o endereço onde o código reside.

Pós-condição: O sistema alvo executa a função maliciosa.

Figura 4.6. Padrão de ataque de buffer overflow (tradução de Morais [81])

4.3 Resumo

O presente capítulo descreve os principais conceitos das duas técnicas de apoio utilizadas para estabelecer a abordagem: injeção de falhas e modelos de ataques.

As diferentes técnicas para injeção de falhas foram descritas, como injeção de falhas por simulação, por hardware e por software, podendo-se concluir que, apesar das vantagens e desvantagens de cada método, a injeção de falhas por software é mais controlável e mais barata do que a injeção por hardware, e mais próxima das reais condições de execução, quando comparada com simulação. Também foram apresentadas ferramentas para injeção de falhas por software.

O principal objetivo de modelos de ataques é permitir a representação clara e organizada das ações de um atacante para realizar um ataque com sucesso. Esses modelos têm sido utilizados para avaliar a segurança de sistemas reais e também para realizar testes de segurança.

5 Capítulo 5

Abordagem, Experimentos e Resultados

Este capítulo apresenta a abordagem proposta para validar a eficácia dos *scanners* de vulnerabilidade. A eficácia é avaliada em relação à cobertura na detecção de vulnerabilidades de segurança e em relação aos falsos positivos, ou seja, scanners com maior eficácia apresentam menor taxa de *falta* de cobertura e menor taxa de falsos positivos. O objetivo da abordagem é fornecer diretrizes para a aquisição e utilização dessas ferramentas e com isso proporcionar maiores níveis de dependabilidade para aplicações *Web*, neste caso, desenvolvidas em Java. Esta abordagem consiste na realização de testes de segurança baseada em modelos e também se estende a empresas e organizações que não possuem ou não podem adquirir este tipo de ferramenta. Isto porque permite que os testes de segurança sejam realizados mesmo na presença de falhas de software que, quando ativadas, podem ser responsáveis por vulnerabilidades de segurança. E também permite utilizar uma abordagem caixa preta em casos em que o código fonte da aplicação a ser testada não esteja disponível.

Um estudo de caso sobre aplicações reais foi desenvolvido, utilizando a abordagem para validar scanners líderes de mercado. Os resultados são apresentados na Seção 5.3.

5.1 Descrição da Abordagem

A abordagem proposta na presente dissertação teve como base o desenvolvimento de dois principais estudos prévios: o primeiro é um estudo de campo para identificar falhas de software que levam a vulnerabilidades de segurança em aplicações Java [28]; o segundo é um estudo da análise do efeito das falhas de software sobre vulnerabilidades de segurança e, consequentemente, sobre a eficácia dos resultados providos pelos scanners de vulnerabilidades [32].

Falhas de software são consideradas a principal causa de vulnerabilidades de segurança exploradas com sucesso por de ataques. Para testar aplicações contra vulnerabilidades de segurança seria importante conhecer as falhas de software presentes nessa aplicação e, assim, poder explorar as possíveis vulnerabilidades causadas por essa falha. Porém, as falhas dificilmente são encontradas, motivo pelo qual permanecem no software sem que sejam corrigidas. Dessa forma, é preciso usar técnicas de emulação de falhas no produto sob teste e é para isso que a injeção de falhas é utilizada. A injeção de falhas permite que a aplicação sob teste seja avaliada na presença de falhas controladas, permitindo a avaliação do impacto da detecção de vulnerabilidades e a aplicação de contramedidas para corrigi-la ou amenizar seus efeitos quando explorada.

O primeiro estudo de campo que serviu de base para estabelecer a abordagem foi importante para identificar o *faultload* que é utilizado para que as falhas injetadas nas aplicações alvo sejam realistas e representem as falhas que realmente ocorrem no ambiente operacional. Ele permite que falhas de software características da linguagem de programação Java sejam identificadas e injetadas posteriormente. Para determinação do *faultload* utilizou-se a metodologia proposta por Fonseca e Vieira [3], que investiga *patches* de correção de vulnerabilidades de segurança disponíveis em aplicações desenvolvidas utilizando a linguagem PHP. Basicamente, essa metodologia consiste em analisar a diferença (i.e., as alterações introduzidas no código fonte) entre o *patch* de correção e a versão da aplicação à qual o *patch* se refere. Isto torna possível a visualização da correção efetuada e, consequentemente, a identificação da falha.

As falhas identificadas na linguagem Java são então classificadas de acordo com o trabalho de Durães e Madeira [22] para a composição do *faultload*. Este estudo gerou uma publicação intitulada "*An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults*", publicada no *Workshop* de Testes e Tolerância a Falhas – WTF 2009, em conjunto com o *Latin American Dependable Computing* – LADC 2009 [28] e se encontra no Apêndice A desta dissertação. A Tabela 5.1 apresenta as falhas de software responsáveis por vulnerabilidades de segurança mais frequentemente observadas, constituindo o *faultload* utilizado neste trabalho.

Tabela 5.1. Falhas de segurança em Java mais frequentemente observadas [28]

Natureza da Falha	**Classificação da falha de acordo com tipo específico**	**Quantidade de Falhas**
Missing construct	Missing function call (MFC)	14
	Missing If construct plus statements (MIFS)	11
Wrong construct	Wrong variable used in parameter of function call (WPFV)	8
	Wrong value used in variable initialization (WVIV)	7
	Wrong value used in parameter of function call (WPFL)	5
	Wrong algorithm - code was misplaced(WALR)	4
Total de falhas		**49**

Na Tabela 5.1, as falhas de software apresentadas foram identificadas, em diferentes contextos, como responsáveis por vulnerabilidades de segurança em aplicações Web. Essas aplicações Web são disponíveis e amplamente utilizadas. Os dois tipos de falhas mais frequentemente observados são MFC e MIFS com, respectivamente, 14 e 11 falhas identificadas. Falhas do tipo MFC se referem a uma função ou chamada a um método que estava faltando na implementação, ou seja, a aplicação vulnerável não apresentava no código fonte essa chamada à função ou método; para corrigir a vulnerabilidade, essa chamada foi acrescentada. Falhas do tipo

MIFS representam a omissão de um bloco de código com uma construção “IF” e suas instruções associadas que são executadas somente se a condição do “IF” for verdadeira.

O segundo estudo que serviu de base para estabelecer a abordagem consiste em injetar as falhas de segurança mais frequentemente observadas no *faultload* [28] em aplicações reais. Para injetar as falhas de maneira controlada foi utilizada a técnica G-SWFIT [22]. Então, o contexto do código fonte é analisado, bem como as tecnologias empregadas para a construção da aplicação alvo. O objetivo é entender como estas falhas de software afetam o comportamento da aplicação e consequentemente, do *scanner*. Com base nesse conhecimento é possível entender como falsos positivos e falta de cobertura afetam a eficácia dos resultados gerados pela ferramenta. Isso é importante para auxiliar no estabelecimento da abordagem proposta. Os resultados mostraram que falhas injetadas afetaram o comportamento das aplicações em relação a vulnerabilidades de segurança, e algumas tecnologias utilizadas podem tornar a aplicação mais segura contra determinados tipos de ataques. Este estudo gerou uma publicação intitulada “*Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool”,* publicada no *IEEE DSN Workshops* (DSNW) - *4th Workshop on Recent Advances in Intrusion-Tolerant Systems* – WRAITS 2010, em conjunto com o *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2010) [32] e se encontra no Apêndice B desta dissertação.

A abordagem em si é composta por sete etapas principais: i) estudo das formas de ataques às vulnerabilidades consideradas; ii) modelagem de árvores para representação dos ataques; iii) análise da aplicação alvo usando o *scanner* sob análise e armazenamento dos resultados; iv) injeção das falhas de software na aplicação alvo; v) reanálise da aplicação alvo (com falha injetada) utilizando o mesmo *scanner*; vi) comparação dos resultados da aplicação original (sem falha injetada) e da aplicação na presença de falhas; vii) validação dos resultados do scanner utilizando testes de segurança a partir das árvores de ataques modeladas. A descrição de cada uma destas etapas é dada a seguir.

i) Estudo das formas de ataques às vulnerabilidades consideradas. Para testar uma aplicação contra vulnerabilidades de segurança é necessário, inicialmente, identificar e entender as maneiras pelas quais esta aplicação pode ser atacada. Se necessário, pode-se analisar até mesmo a probabilidade desses ataques e o impacto (danos) que podem causar e, com isto, fornecer diretrizes para tomada de decisões. Fontes de informação sobre vulnerabilidades e formas de ataques a aplicações podem ser encontradas em páginas na Internet como *National Vulnerability Database* (NVD) [46], *Open Web Application Security Project* (OWASP) [48], CERT *Coordination Center* [83].

Para a realização do presente trabalho foram selecionadas três vulnerabilidades que se pretende investigar: Cross Site Scripting (XSS), injeção de SQL e Cross Site Request Forgery (CSRF). Esta seleção se deve ao fato de serem vulnerabilidades críticas (primeiro, segundo e quinto

lugar na classificação da OWASP Top 10 [62]) e podem causar grandes prejuízos quando exploradas. Embora vulnerabilidades CSRF representem, nessa classificação, o quinto tipo mais crítico, ela foi selecionada porque os ataques que exploram essas vulnerabilidades são difíceis de detectar e normalmente a remediação é possível somente depois do incidente ocorrido. Além disso, segundo Lin *et al.* [41], esta é uma área em que as pesquisas ainda são limitadas.

ii) Modelagem de árvores para representação de ataques. Para organizar e descrever os ataques identificados na primeira etapa escolheu-se utilizar a estrutura de árvores de ataques devido a uma série de vantagens que apresenta. Algumas dessas vantagens são [81]: i) árvores de ataques apresentam foco na análise dos objetivos de atacantes que podem ser traduzidos em testes específicos contra aplicações Web; ii) árvores de ataques permitem descrever, de uma maneira mais estruturada do que linguagem natural, as ações executadas por um ataque bem-sucedido; iii) o modelo é fácil de compreender para pessoas com pouca prática em modelos formais; também é conciso e possibilita que várias pessoas contribuam para sua manutenção; iv) A estrutura modular permite o reuso da árvore de ataques para representar outros ataques maiores, ou seja, uma nova árvore de ataques pode usar módulos menores na sua composição; v) A estrutura hierárquica, na qual objetivos em níveis mais altos são divididos em sub-objetivos até que o nível de refinamento (detalhes) desejado seja alcançado, simplifica a navegação e possibilita que se trabalhe em diferentes ramos em paralelo; vi) o modelo é fácil de estender e possibilita a atualização em caso de descoberta de novos ataques.

As árvores deste presente trabalho foram modeladas para os ataques conhecidos às vulnerabilidades de segurança analisadas. Novas formas de ataques podem surgir e estes ataques podem ser representados na mesma árvore, acrescentando novos nós que representem os respectivos passos para explorar a vulnerabilidade. As árvores de ataques são estruturas modulares que permitem essa expansão.

A Figura 5.1 traz o exemplo de árvore de ataques para a vulnerabilidade *Cross Site Request Forgery* - CSRF. Conforme descrito na Seção 3.2, um ataque CSRF aproveita-se de uma sessão de usuário já estabelecida na aplicação vulnerável para realizar ações automatizadas sem o conhecimento e consentimento da vítima. Nesta árvore foram representados os passos para realizar o ataque a partir da aceitação de requisições vindas de outras fontes. Os passos relativos aos meios utilizados para enganar o usuário para ativar requisições maliciosas não serão representados, pois estão além das barreiras defensivas que uma aplicação pode ter contra ataques. Para efeito de teste de vulnerabilidade, considera-se que o usuário foi levado a ativar o ataque de CSRF, por exemplo, acessando um *link* malicioso em outra aplicação.

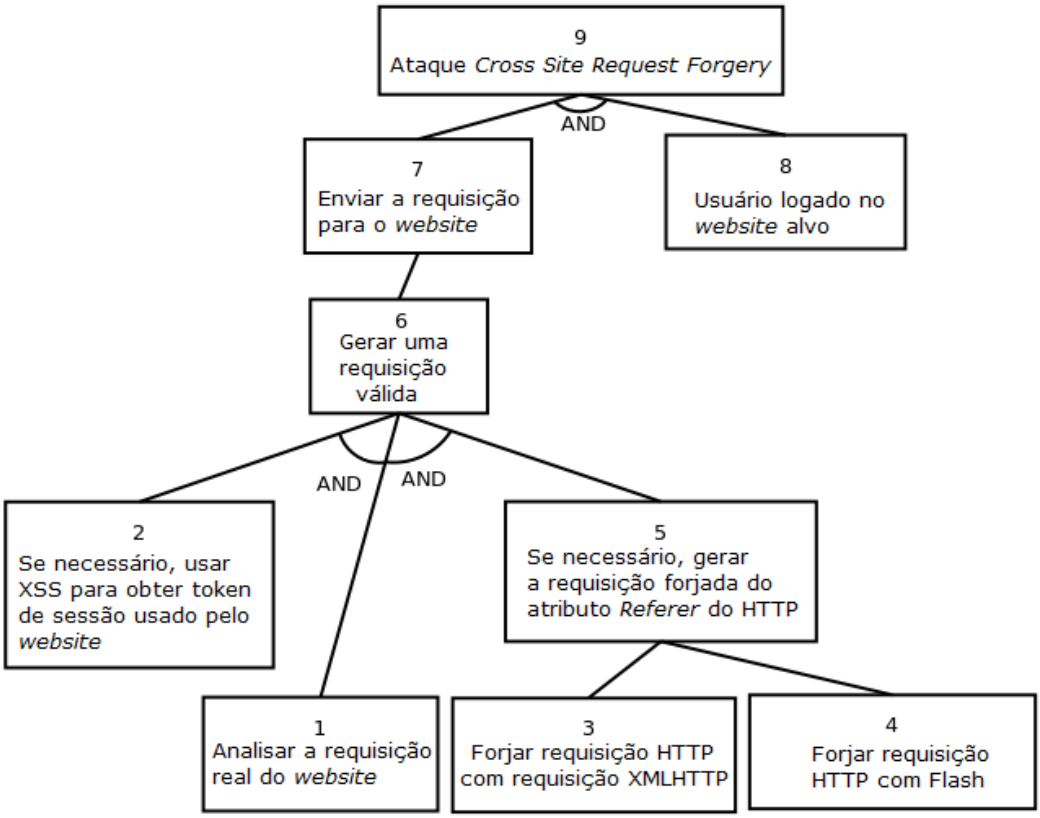


Figura 5.1. Árvore de ataques para vulnerabilidades CSRF

Na Figura 5.1 a principal condição para realizar um ataque CSRF é ter o usuário autenticado pela aplicação (representado pelo nó 8), pois o ataque abusa da confiança que a aplicação tem no usuário já autenticado. Então, se este passo não for satisfeito, o ataque não pode ser realizado. O primeiro passo, representado pelo nó 1, é analisar a requisição da aplicação que será alvo do ataque para poder forjá-la. Se a aplicação alvo não tiver nenhuma contramedida para ataques CSRF, este passo levará ao próximo, representado pelo nó 6, pois a requisição forjada será considerada válida e terá efeito no site.

Se a aplicação alvo utilizar alguma medida defensiva será necessário tomar ações adicionais, além de analisar as requisições. Um método defensivo conhecido consiste em adicionar diferentes *tokens* (marcações) em cada requisição; contudo, mas essa abordagem defensiva pode se tornar ineficaz caso a aplicação seja vulnerável a ataques XSS, pois por esse tipo de ataque seria possível obter um *token* de sessão válido da aplicação alvo e utilizá-lo na requisição forjada. Essa situação é representada pela estrutura formada pelos nós 1 AND 2.

Os demais caminhos, representados pelas estruturas formadas pelos nós 3, 1 AND 5 ou pelos nós 4,1 AND 5 da árvore, mostram como atacar aplicações que usam a verificação do atributo

Referer do HTTP (*Hypertext Transfer Protocol*), que é o atributo que indica a aplicação da qual a requisição se origina. No entanto, esta não é uma medida defensiva recomendada.

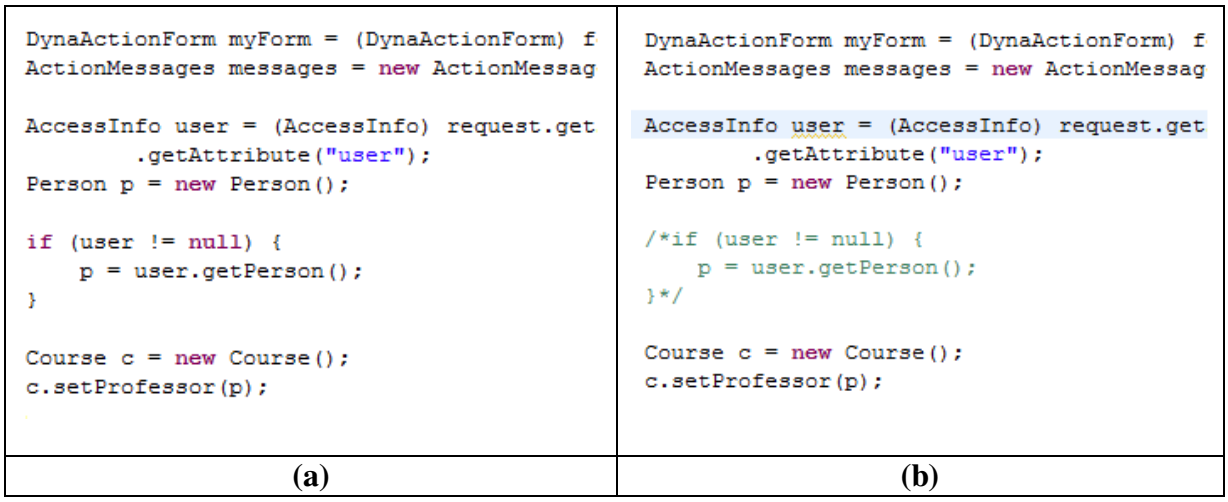
As modelagens das demais árvores bem como suas respectivas explicações e um estudo de caso da realização dos ataques estão no Apêndice C. Este trabalho gerou um artigo intitulado "A*ttack Trees Modeling for Security Tests in Web Applications*", que foi submetido para *The 12th IEEE International High Assurance Systems Engineering Symposium - HASE 2010* [42].

iii) Análise da aplicação alvo usando o *scanner*. Nesta etapa deve-se, primeiramente, utilizar a aplicação original, ou seja, a aplicação sem falhas injetadas. Então executa-se um *scan* com a ferramenta (*scanner*) a ser avaliada sobre a aplicação original – chamamos esta execução de "*gold run* "- e os resultados obtidos devem ser armazenados de maneira apropriada para que sejam utilizados como base de comparação com os resultados dos experimentos na presença de falhas. A aplicação originalmente já pode ter algumas vulnerabilidades e este primeiro *scan* deve ser capaz de detectá-las, de forma que se assegure que as novas vulnerabilidades, identificadas nos experimentos, são decorrentes das falhas injetadas. Este primeiro *scan* deve servir de referência (oráculo) para se verificar o efeito das novas falhas injetadas, especialmente na criação de novas vulnerabilidades.

iv) Injeção das falhas de software na aplicação alvo. A partir do *faultload*, as falhas devem ser injetadas na aplicação que será testada. Esta etapa é importante para validar a eficácia dos scanners, porque permite que haja um controle sobre as falhas de software presentes no código, sabendo que as falhas estão no software e onde estão localizadas. Se novas vulnerabilidades de segurança forem identificadas em função da injeção das falhas, impactando o funcionamento da aplicação, é possível verificar a eficácia do scanner em detectar a nova vulnerabilidade.

Para injetar as falhas pode-se utilizar a técnica G-SWFIT [22] ou a ferramenta J-SWFIT [17]. Neste trabalho utilizou-se, em uma primeira fase, aplicações alvo cujo código fonte está disponível. Então, localizam-se os possíveis locais para injeção do tipo de falha em questão e uma falha é injetada de cada vez, para que os efeitos de sua ativação sejam analisados isoladamente. Caso o código fonte da aplicação a ser testada não esteja disponível, a G-SWFIT permite que as falhas sejam emuladas diretamente no código binário (compilado). Para isso é necessária a definição de operadores que representem estas falhas também em código binário. Em uma segunda fase do trabalho, cujo objetivo é escalar a solução para aplicações maiores e mais complexas, sem o código fonte disponível, utilizou-se a ferramenta de injeção de falhas J-SWFIT, descrita na Seção 4.1.

Para os experimentos optou-se por injetar as falhas mais frequentemente observadas na Tabela 5.1. A Figura 5.2 traz um exemplo de injeção de falha do tipo MIFS.



```
DynaActionForm myForm = (DynaActionForm) f
ActionMessages messages = new ActionMessag

AccessInfo user = (AccessInfo) request.get
        .getAttribute("user");
Person p = new Person();

if (user != null) {
    p = user.getPerson();
}

Course c = new Course();
c.setProfessor(p);
```

(a)

```
DynaActionForm myForm = (DynaActionForm) f
ActionMessages messages = new ActionMessag

AccessInfo user = (AccessInfo) request.get
        .getAttribute("user");
Person p = new Person();

/*if (user != null) {
    p = user.getPerson();
}*/

Course c = new Course();
c.setProfessor(p);
```

(b)

Figura 5.2. Exemplo de injeção de falhas do tipo MIFS. (a) Código fonte da aplicação original. (b) Código fonte com a falha injetada.

v) Utilização do scanner na aplicação alvo (com falha injetada). Neste passo deve-se executar o *scan* na aplicação com a falha injetada. Várias execuções são feitas, uma para cada falha que se pretende injetar. Os resultados obtidos dessas execuções devem ser armazenados de maneira similar ao efetuado na etapa iii, facilitando uma automatização da comparação dos resultados obtidos nessa etapa com os resultados da "*gold run*".

vi) Comparação dos resultados. Os resultados dos *scans* originais ("*gold run*" - *scan* executado na aplicação sem falhas injetadas) devem ser comparados com os resultados de cada *scan* executado na aplicação com falha injetada para avaliar se houve alterações. Neste caso, as diferenças entre os resultados devem ser investigadas, pois podem indicar falsos positivos, falta de cobertura por parte da ferramenta ou novas vulnerabilidades de segurança em decorrência da presença de falha de software. É importante que os resultados do *scan* na aplicação original também sejam investigados. Esta investigação é descrita na etapa seguinte.

vii) Validação dos resultados do *scanner* utilizando testes de segurança a partir das árvores de ataques modeladas. Caso haja diferenças entre os resultados dos *scans* da aplicação original e da aplicação com falha injetada, deve-se realizar ataques à(s) vulnerabilidade(s) que se deseja investigar. Estes ataques são guiados pelas árvores de ataques modeladas na segunda etapa desta metodologia. Se o ataque obtiver sucesso é porque a ferramenta detectou a vulnerabilidade corretamente. Neste caso, deve-se repetir o teste também na aplicação original para verificar se a vulnerabilidade já existia antes da falha ser injetada e não foi detectada pela ferramenta, indicando falta de cobertura. Por outro lado, se o ataque a determinada vulnerabilidade não obtiver sucesso, a ferramenta detectou um falso positivo.

A Figura 5.3 traz o fluxograma que representa o procedimento de validação dos resultados do scanner sob avaliação.

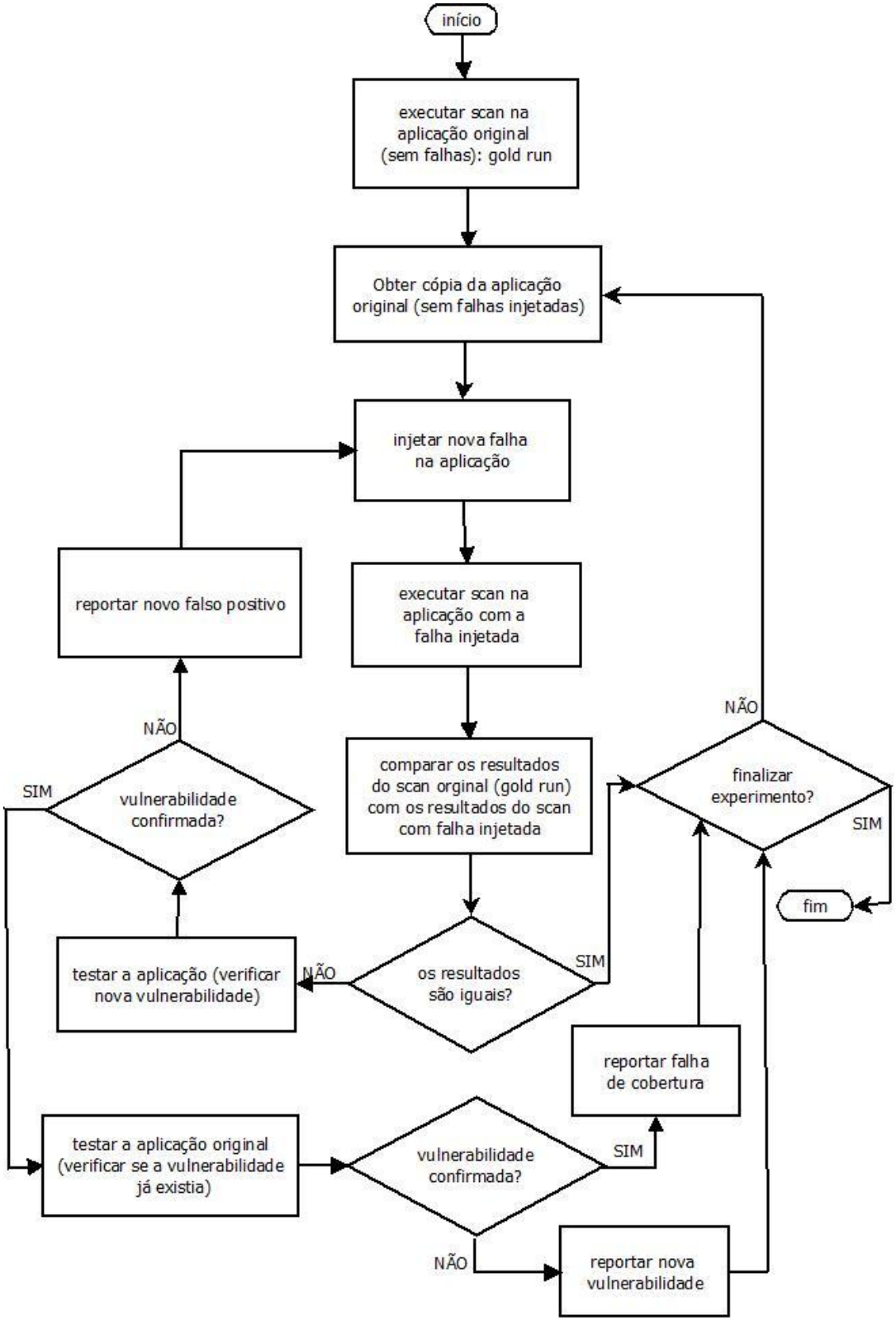


Figura 5.3. Procedimento para avaliação dos resultados do scanner de vulnerabilidade

As etapas descritas para a presente abordagem podem ser adaptadas para a realização de testes de segurança quando não se possui este tipo de ferramenta (*scanners* de vulnerabilidades). Esses testes, apesar de demandarem mais tempo do que os testes com apoio de ferramentas automáticas, são uma maneira de garantir maior nível de segurança à aplicação em questão quando, por exemplo, por restrições de orçamento não se possa contar com o apoio destas ferramentas. Para realizá-los, é necessário seguir as etapas **i** (estudo das formas de ataques às vulnerabilidades consideradas), **ii** (modelagem de árvores para representação destes ataques) e **iv** (injeção das falhas de software na aplicação alvo). Como última etapa, para este contexto, deve-se realizar os testes de segurança utilizando as árvores de ataques. Os ataques devem ser realizados em cada caso de uso da aplicação a fim de identificar vulnerabilidades específicas.

5.2 Estudo de caso

Esta seção apresenta um exemplo do uso da abordagem para mostrar sua aplicabilidade na avaliação da eficácia de scanners de vulnerabilidades comercialmente disponíveis e nos testes de segurança em aplicações reais. São descritos os passos para a preparação e execução dos experimentos, bem como os resultados observados.

Na preparação selecionaram-se as aplicações *Web* desenvolvidas em Java, selecionaram-se os scanners de vulnerabilidades e definiram-se as vulnerabilidades de segurança que serão investigadas. Na execução a abordagem foi aplicada, de forma que se identificou o *faultload*; modelaram-se as árvores de ataques; injetaram-se as falhas; os *scans* foram executados e os resultados foram investigados por ataques manuais baseados nas árvores modeladas. Além da aplicação da abordagem, uma análise do contexto do código fonte foi realizada para verificar qual o impacto que a ativação da falha causa no comportamento da aplicação e consequentemente no comportamento da ferramenta. Também se investigou, com base no contexto do código fonte, a relação entre a falha injetada e possíveis novas vulnerabilidades de segurança criadas.

Preparação. Foram selecionadas duas aplicações *Web* desenvolvidas em Java. Os códigos fonte estavam disponíveis (*open source*) e a primeira aplicação possui 137 classes e aproximadamente 27.048 linhas de código; a segunda possui 262 classes e aproximadamente 45.800 linhas de código. Uma das aplicações, chamada Eberom [84], versão 1.0, foi desenvolvida para o gerenciamento de projetos e relacionamento com clientes (*Customer Relationship Manager - CRM*). Ela possui como principais funcionalidades: cadastro de funcionários e consultores; planilha de controle de horas de trabalho para cada funcionário, que inclui férias, ausências por motivos de saúde, entre outros fatores; cadastro de produtos e empresas; realização de cotação de produtos para determinada empresa; geração de vários tipos de relatórios. A aplicação utiliza o banco de dados *MySQL* versão 5.0, ambiente Java (*Java Runtime Environment – JRE*) versão 5.0 update 7 e servidor de aplicação Tomcat versão 5.5.28. Também utiliza tecnologias como Hibernate [85] para

persistência objeto/relacional, *framework* de desenvolvimento Struts [86] e JasperReports [87] para apoio à geração de relatórios.

A segunda aplicação é um sistema de gestão de aprendizagem para Ensino a Distância chamada Amadeus [88], versão 0.6. Ela permite: cadastrar alunos e professores; gerenciar cursos e respectivas turmas de alunos; gerenciar material de apoio utilizado nos cursos; controlar tarefas dos alunos. A aplicação utiliza o banco de dados PostgreSQL versão 8.4, ambiente Java (*Java Runtime Environment – JRE*) versão 6.0 e servidor de aplicação Tomcat versão 5.5.28. Tecnologias como Hibernate [85] e AJAX (*Asynchronous Javascript And XML***)** são utilizadas nesta aplicação.

As duas aplicações foram selecionadas por terem o código fonte disponíveis, por representarem um mesmo segmento de aplicações (gestão de dados) e por utilizar algumas tecnologias que provavelmente impactem nos resultados dos *scanners* (por exemplo, Hibernate, que está ligada diretamente à construção de consultas (*queries*) e afeta a detecção de vulnerabilidades do tipo injeção de SQL). Além disso, são aplicações cujos serviços se aplicam às áreas comercial e acadêmica; especialmente esta última, para gerenciamento de cursos, atividades, material didático tem sido bastante utilizada ultimamente no meio acadêmico. Para a injeção das falhas de software foram selecionados casos de uso semelhantes entre as duas aplicações: “manter produto” (incluir, excluir, alterar produto) e “manter curso” (incluir, excluir e alterar curso), respectivamente.

Os scanners de vulnerabilidades selecionados foram Acunetix versão 6.0 [8], Rational AppScan [10] versão 7.8 e HP WebInspect [12] versão 8.0. Estas ferramentas foram selecionadas devido à grande inserção que têm no mercado.

Como já mencionado, os três tipos de vulnerabilidades de segurança que foram considerados são C*ross Site Scripting* (XSS), injeção de SQL e C*ross Site Request Forgery* (CSRF). Esta seleção foi devido à alta criticidade e os danos que estas vulnerabilidades podem causar quando exploradas. A explicação detalhada destas vulnerabilidades e respectivas árvores de ataques encontram-se no Apêndice C.

Execução. Os tipos de falhas a serem injetadas foram selecionadas com base no *faultload* definido por Basso *et al.* [28]. O tipo *Missing Function Call* (MFC) e *Missing If Construct plus Statement* (MIFS) foram selecionados por serem os tipos de falhas de segurança mais frequentemente observados. O tipo *Wrong Extended Class* (WEC) também foi selecionado por ser o tipo de falha específico da linguagem Java mais frequentemente observado. Este tipo de falha consiste na especificação errada da super classe durante a declaração de uma classe. Esta falha e seu respectivo *faultload* (composto por falhas específicas da linguagem Java) pode ser observada no Apêndice A, *Table 7.* A estrutura da falha pode ser observada neste mesmo apêndice, na *Table 8.*

Os experimentos iniciam com a execução de um *scan* inicial, a “*gold run*”, onde a aplicação é testada pela primeira vez pela ferramenta sem qualquer falha injetada. É esperado que a aplicação já tenha algumas vulnerabilidades e o objetivo desta execução é encontrá-las para que não se atribua vulnerabilidades existentes a priori como sendo uma consequência da injeção de falhas

efetuada. Depois da "*gold run*", todas as vulnerabilidades apontadas são verificadas, ou seja, ataques são planejados de forma a confirmar que cada vulnerabilidade apontada realmente existe e indica pontos passíveis de serem atacados. Nesta verificação identificam-se falsos positivos, quando uma vulnerabilidade é apontada e os ataques possíveis, de acordo com o tipo da vulnerabilidade apontada, não resultam em um sucesso, significando que esta vulnerabilidade na realidade não existe.

A seguir, uma falha é injetada. O contexto do código fonte onde a falha foi injetada é analisado para investigar o efeito desta falha no comportamento da aplicação. Se necessário, dados são inseridos, removidos ou alterados na base de dados para se garantir a ativação da falha injetada. Então, o código e a base de dados são versionados, definindo um cenário a ser testado.

O *scanner* de vulnerabilidade é executado e observa-se se novas vulnerabilidades do interesse deste trabalho foram identificadas. Em caso afirmativo, ataques são realizados no atual cenário da aplicação *Web* utilizando-se as árvores de ataques, com o objetivo de verificar se a nova vulnerabilidade detectada realmente existe ou é um falso positivo. Os mesmos ataques são realizados no cenário da aplicação original (sem qualquer falha injetada) para verificar se a vulnerabilidade já existia anteriormente à injeção da falha e não foi identificada pela ferramenta na "*gold run*", caracterizando, nesse caso, uma falha de cobertura já na análise inicial feita pela ferramenta na aplicação original.

O procedimento descrito acima é repetido para cada possível local onde a falha possa ser injetada no código fonte do cenário selecionado. Os locais possíveis para injeção da falha é determinado pela técnica G-SWFIT. Os experimentos são repetidos para cada uma das três ferramentas, analisando as duas aplicações.

5.3 Resultados

Esta seção apresenta a análise dos resultados obtidos nos experimentos realizados no estudo de caso. O objetivo é verificar a aplicabilidade da abordagem proposta e avaliar a eficácia dos scanners de vulnerabilidades.

Para a apresentação dos resultados decidiu-se não associar marcas e versões dos scanners, bem como das aplicações, para assegurar neutralidade e também porque fornecedores de ferramentas comerciais geralmente não permitem a publicação de resultados de avaliações desse tipo. Assim, os scanners utilizados nos experimentos serão referidos, neste trabalho, a partir deste ponto, como S1, S2 e S3, sem qualquer ordem em especial. As aplicações serão chamadas App1 e App2, também sem qualquer ordem em especial.

Efeito da injeção de falhas sobre os resultados. Para ambas as aplicações *Web* foram analisados, respectivamente, 31 e 35 diferentes cenários. A segunda linha da Tabela 5.2 mostra o

total de cenários cujos resultados foram afetados pela falha injetada. Estes resultados incluem alterações na quantidade de vulnerabilidades detectadas, especialmente em função de novas vulnerabilidades causadas pela injeção de falhas.

Tabela 5.2. Cenários e vulnerabilidades

	App1	App2	Total
Total de cenários analisados	31	35	66
Cenários que apresentaram resultados afetados pela falha injetada	22	20	42
% de falhas que afetaram os resultados dos scanners	70,9%	57,1%	63,6%

De acordo com a Tabela 5.2, em média 63,6% das falhas de software injetadas afetaram os resultados dos *scanners*. Em aproximadamente 20% dos 66 cenários, as alterações dos resultados foram verificadas em pelo menos dois dos *scanners*. A Tabela 5.3 apresenta a quantidade de falhas que afetaram os resultados dos *scanners* de acordo com cada tipo de falha. Também apresenta a quantidade de alterações que a falha injetada produziu para cada tipo de vulnerabilidade considerada neste trabalho. Em vários casos, um único cenário, ou seja, a injeção de uma única falha afetou os resultados de forma que uma ou mais novas vulnerabilidades surgiram e/ou uma ou mais vulnerabilidades deixaram de ser detectadas, divergindo dos resultados originais. Todas estas divergências foram contabilizadas considerando todos os cenários e seus totais são apresentados, nas três últimas colunas da Tabela 5.3.

Tabela 5.3. Tipos de falhas que afetaram resultados dos scanners.

			Alterações de vulnerabilidades criadas pela injeção da falha			
Tipo da falha	Cenários	Falhas que afetaram *scans*	XSS	Injeção de SQL	CSRF	Total
MFC	33	20 (60,6%)	1	18	32	51
MIFS	26	17 (65,3%)	0	5	38	43
WEC	7	5 (71,5%)	0	3	11	14
Total	66	42 (63,6%)	1	26	81	108

Percebe-se, na Tabela 5.3, que MFC foi o tipo de falha mais injetado, contabilizando 33 cenários. Este tipo de falha foi o que mais afetou os resultados referentes às vulnerabilidades do tipo Injeção de SQL, com 18 variações de resultados para este tipo de vulnerabilidade. Também foi o único tipo de falha que afetou resultados referentes às vulnerabilidades do tipo XSS. A vulnerabilidade CSRF foi a que mais apresentou alterações nos resultados, com 81 alterações em

um total de 108 alterações observadas. Os tipos de falha MFC e MIFS produziram uma quantidade próxima de alterações nos resultados para as vulnerabilidades do tipo CSRF: 32 e 38 alterações respectivamente.

Ainda na Tabela 5.3, as porcentagens de falhas que afetaram os *scans* também são próximas: 60,6% para MFC e 65,3% para MIFS. O tipo de falha WEC foi o menos encontrado, porém, das 7 falhas injetadas, 5 afetaram os resultados dos scanners, caracterizando 71,5% de efeito sobre a detecção das vulnerabilidades analisadas, sendo este o maior impacto entre os tipos de falhas.

Uma análise detalhada do contexto do código fonte da aplicação onde as falhas foram injetadas foi importante para avaliar o efeito da falha no comportamento da aplicação. Esta análise do contexto juntamente com a estrutura dos ataques permitiu avaliar a influência da falha injetada sobre potenciais novas vulnerabilidades de segurança detectadas pelas ferramentas. Consequentemente, este procedimento permite avaliar corretamente a eficácia do *scanner* em questão, pela identificação de falta de cobertura e existência de falsos positivos.

A partir da análise notou-se que as falhas em si não foram o foco da criação de novas vulnerabilidades, ou seja, os locais onde as novas vulnerabilidades surgiram não correspondem exatamente aos locais onde as falhas foram injetadas. As falhas injetadas afetaram o comportamento da aplicação e consequentemente o comportamento das ferramentas por meio do contexto da aplicação e dos procedimentos necessários para ativar a falha. Por exemplo, muitas falhas do tipo MIFS [28] foram injetadas em locais no código fonte onde é feita uma verificação de um dado de entrada cujo valor seja nulo. Ativando esta falha, a aplicação modifica seu comportamento, não verificando e permitindo que dados de entrada nulos sejam aceitos pela aplicação, levando-as a exibir páginas de erro (normalmente valores nulos não seriam aceitos pela aplicação). Também, de acordo com a estrutura de ataque presente na Figura 5.1, a verificação de dados de entrada nulos não é explorada, ou seja, esta permissão de valores nulos não cria uma potencial vulnerabilidade de segurança que possa ser explorada por atacantes (*hackers*).

Ainda que as falhas de software injetadas não estejam relacionadas aos mesmos locais onde novas vulnerabilidades surgiram, elas de fato afetaram os resultados das ferramentas.

Falta de cobertura e falsos positivos. A Tabela 5.4 mostra, para cada scanner e para cada aplicação, a quantidade de vulnerabilidades não detectadas (falta de cobertura) e de falsos positivos. A falta de cobertura foi contabilizada com base no número total de vulnerabilidades detectadas corretamente por meio dos testes baseados nas árvores de ataques.

Tabela 5.4. Falta de cobertura e falsos positivos referentes às aplicações

	Vulnerabilidades não detectadas (falta de cobertura)			Falsos positivos		
	App1	App2	Total	App1	App2	Total
S1	19	3	22	13	1	14
S2	54	17	71	1	0	1
S3	54	10	64	18	1	19

Na Tabela 5.4 é possível observar que *scans* executados na App1 apresentaram mais casos de falta de cobertura e falsos positivos do que os executados na App2. Pela análise do contexto do código fonte acredita-se que o fato da App2 ser mais modularizada, com menos acoplamento entre módulos e menos casos de uso implementados tenha sido responsável por este resultado.

A Tabela 5.5 apresenta a falta de cobertura e falsos positivos de acordo com os tipos de vulnerabilidades analisadas. Os testes executados de acordo com as árvores de ataques permitiram confirmar que existem, de fato, entre os 66 cenários analisados envolvendo ambas as aplicações, 2 vulnerabilidades XSS, nenhuma vulnerabilidade do tipo injeção de SQL e 71 vulnerabilidades CSRF. A partir destes resultados serão analisados os índices de falso positivo e falta de cobertura dos scanners.

Observa-se que a segunda linha da Tabela 5.5 traz a quantidade de vulnerabilidades analisadas. Esse termo foi utilizado porque existem casos em que vulnerabilidades detectadas em um cenário com falha injetada já existiam na aplicação original e, naquela ocasião ("*gold run*") não foram detectadas pela ferramenta, caracterizando falta de cobertura na aplicação original. Então, não só as novas vulnerabilidades foram analisadas, mas também a presença delas na aplicação original. Nesse caso, todas as análises são contabilizadas (das vulnerabilidades detectadas e dos casos de falta de cobertura).

Tabela 5.5. Falta de cobertura e falsos positivos referentes aos tipos de vulnerabilidades.

	S1			S2			S3		
	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF
Vulnerabilidades existentes nas aplicações	2	0	71	2	0	71	2	0	71
Vulnerabilidades analisadas	2	2	83	3	0	71	2	18	72
Detectadas corretamente	2	0	49	2	0	0	1	0	8
Falta de cobertura	0	0	22	0	0	71	1	0	63
Falso positivo	0	2	12	1	0	0	0	18	1

Ainda na Tabela 5.5 observa-se que o scanner S1 identificou corretamente as 2 vulnerabilidades do tipo XSS existentes. O S2 também identificou as 2 vulnerabilidades corretamente, porém, apresentou 1 falso positivo. E o scanner S3 identificou somente uma das vulnerabilidades corretamente, apresentando um caso de falta de cobertura para este tipo de vulnerabilidade de segurança. Nenhuma vulnerabilidade do tipo injeção de SQL foi confirmada, e isso foi detectado corretamente por S2. S1 e S3 apresentaram, respectivamente, 2 e 18 falsos positivos. Para vulnerabilidades do tipo CSRF, S1 apresentou 22 casos de falta de cobertura e 12 falsos positivos; S2 apresentou total falta de cobertura para vulnerabilidades desse tipo e S3 apresentou alto índice de falha de cobertura (63 vulnerabilidades não detectadas), porém baixo índice de falso positivo (somente 1 detectado).

Também é possível observar na Tabela 5.5 que vulnerabilidades do tipo CSRF são encontradas com uma frequência muito maior do que as dos tipos XSS e injeção de SQL. Isso se deve ao fato das aplicações utilizarem *framework* de desenvolvimento e tecnologia de persistência de dados, tais como *Hibernate*. *Frameworks* de desenvolvimento como *Struts* ou *Java Server Faces* possuem, como padrão, um filtro para execução de *scripts*. Então, a menos que o desenvolvedor, ao construir o código da aplicação, especifique que a aplicação deve executar scripts – e isso não é comum porque a idéia é acrescentar aspectos de segurança à aplicação e não removê-los – a aplicação vai tratar qualquer *script* como uma string, ou seja, marcações do tipo <script></script> não são executadas e sim adicionadas à aplicação como um texto comum. Já o uso do Hibernate

permite que a aplicação tenha suas consultas (*queries*) totalmente encapsuladas em objetos e APIS (*Application Programming Interfaces*), não permitindo que trechos da consulta sejam concatenados.

A Tabela 5.6 mostra a porcentagem de falta de cobertura e falsos positivos de cada scanner. As respectivas porcentagens foram calculadas em relação à quantidade de vulnerabilidades analisadas neste estudo de caso, representadas na segunda linha da Tabela 5.5.

Tabela 5.6. Porcentagem de falta de cobertura e falsos positivos

	S1			S2			S3		
	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF
Detectadas corretamente	100%	0%	59%	66,6%	100%	0%	50%	0%	11,2%
Falta de cobertura	0%	0%	26,5%	0 %	0%	100%	50%	0%	87,5%
Falso positivo	0%	100%	14,5%	33,4%	0 %	0%	0%	100%	1,3%

Na Tabela 5.6 é possível observar que o maior índice de falta de cobertura se refere a vulnerabilidades do tipo CSRF e abrange os três scanners S1, S2, e S3 com respectivamente 26,5%, 100% e 87,5%. As falhas de cobertura foram investigadas tanto na aplicação original (sem falha injetada) e nas aplicações com falhas injetadas. Em alguns casos, vulnerabilidades identificadas na aplicação original deixaram de ser detectadas nas aplicações com falhas injetadas, caracterizando falha de cobertura na aplicação com falha injetada. No entanto, para a maioria dos casos, ao executar o scanner em uma aplicação com falhas injetadas, uma nova vulnerabilidade foi detectada e, ao analisar a aplicação original, esta vulnerabilidade já existia e não foi identificada na *"gold run*".

Ainda na Tabela 5.6, os falsos positivos abrangem os três tipos de vulnerabilidades, representando 33% para XSS (detectados pelo scanner S2), 100% para injeção de SQL (detectados pelos scanners S1 e S3) e aproximadamente 16% para vulnerabilidades CSRF (14,5% detectados pelo scanner S1 e 1,3% detectados pelo S3). Os falsos positivos associados às vulnerabilidades do tipo XSS ocorreram porque a ferramenta integra um navegador interno que corresponde a uma versão ultrapassada de navegadores para acesso à Internet. Um ataque bem sucedido executado pela ferramenta, quando executado manualmente, utilizando versões mais recentes de navegadores para acesso à Internet, não surte efeito, porque estas versões mais recentes implementam características que não permitem a execução de ataques comuns a vulnerabilidades XSS.

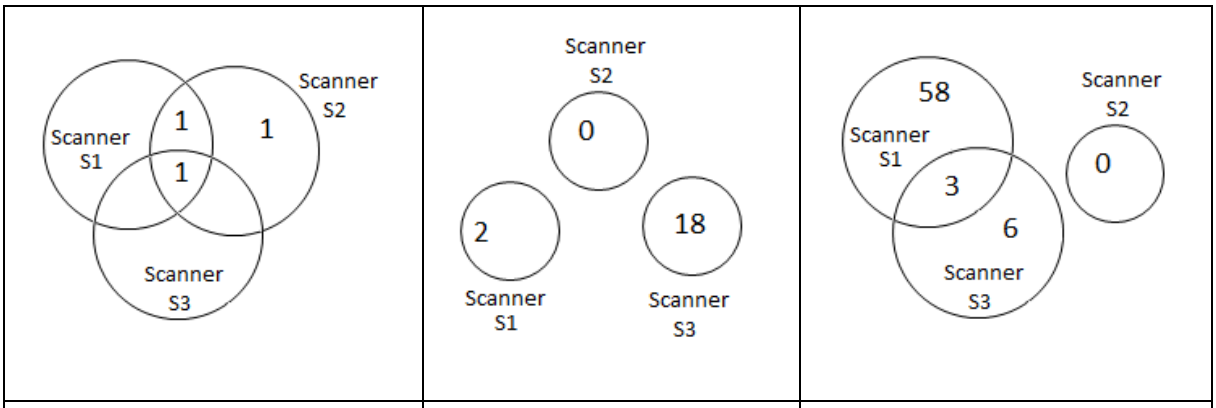
Os falsos positivos associados à vulnerabilidade do tipo injeção de SQL foram identificados por ataques e pela análise do código fonte. Ambas as aplicações utilizam a tecnologia Hibernate, que é um serviço de consulta e persistência objeto/relacional [85]. Hibernate permite encapsular as consultas (*queries*) e enviar objetos para a base de dados através de métodos e classes

predefinidos, descartando a necessidade de construções explícitas de consultas SQL. De acordo com alguns fóruns e informações técnicas disponíveis em páginas na Internet [88][90], em códigos construídos com Hibernate é mais difícil – mas não impossível – encontrar vulnerabilidades que sejam exploradas por ataques do tipo injeção de SQL, pois esta tecnologia também permite, em alguns casos, construções de consultas menos encapsuladas, ou seja, é possível encontrar construções de consultas SQL com parte explícita em código que utiliza Hibernate. No entanto, a maneira que as aplicações App1 e App2 foram codificadas, isto é, extremamente encapsulada, não oferece oportunidade para desenvolver ataques bem sucedidos. Até mesmo a ferramenta que indicou este tipo de vulnerabilidade não garante a eficácia da detecção, informando que requer uma verificação por parte do usuário para confirmar a existência da vulnerabilidade.

A maioria dos casos onde foram identificados os falsos positivos referentes à vulnerabilidade CSRF é referente a páginas de erro. Um ataque que acessa uma página de erro pode ser perigoso se esta página de erro apresenta *links* ou botões que possibilitem o acesso direto à aplicação (como, por exemplo, botões ou *links* do tipo “voltar”, que apresentam a última página da aplicação acessada pelo usuário). Também pode ser perigoso se a página de erro apresentar informações relevantes ou privadas sobre o sistema (como, por exemplo, nomes de tabelas do banco de dados). Para ambas as aplicações analisadas, as páginas de erro não apresentam maneiras de acessar funcionalidades da aplicação ou informações privadas. Então, consideraram-se estes casos como falsos positivos porque um ataque a uma vulnerabilidade CSRF, quando resulta no acesso a estas páginas de erros, não será bem sucedido.

Na Tabela 5.6 é possível observar que, apesar do tipo de vulnerabilidade dominante ser CSRF, dois dos três *scanners* (S2 e S3) apresentaram alta porcentagem de falha de cobertura para vulnerabilidades CSRF. O único tipo de vulnerabilidade que foi detectada pelos três scanners, com quantidades próximas, foi o XSS.

Eficácia dos scanners avaliados. A Figura 5.4 ilustra, para cada tipo de vulnerabilidade analisada, a relação entre as detecções pelos *scanners* S1, S2 e S3. A intersecção entre as áreas dos círculos representa a quantidade de mesmas vulnerabilidades detectadas por mais de um scanner. Os casos de falta de cobertura não foram considerados para a ilustração, ou seja, foram consideradas somente as vulnerabilidades detectadas, inclusive os falsos positivos.



XSS	Injeção de SQL	CSRF

Figura 5.4. Relação entre as detecções dos tipos de vulnerabilidades pelos scanners.

Na Figura 5.4 observa-se que existem poucas vulnerabilidades em comum entre os três scanners. Essa diferença é um indicador de que estas ferramentas implementam diferentes formas de executar seus testes de intrusão e que os resultados de diferentes *scanners* podem se diferenciar bastante. Isto sugere que, para ter boa cobertura dos testes de segurança, o usuário pode combinar vários *scanners* ao invés de confiar nos resultados de somente um deles. Uma análise da eficácia destes scanners pode fornecer diretrizes para a seleção destas ferramentas. A Tabela 5.7 mostra a porcentagem de falhas de cobertura e falsos positivos de cada scanner analisado.

Tabela 5.7. Falta de cobertura e falsos positivos referentes aos scanners.

	S1	S2	S3
Vulnerabilidades analisadas	87	74	92
Vulnerabilidades detectadas corretamente	51 (58,6 %)	2 (2,7 %)	9 (9,8%)
Falta de cobertura	22 (25,3 %)	71 (95,9 %)	64 (69,6%)
Falso positivo	14 (16,1 %)	1 (1,4 %)	19 (20,6%)

Ao realizar uma comparação dos aspectos de cobertura e falsos positivos que as ferramentas apresentaram, fica claro, observando a Tabela 5.7, que o *scanner* S1 é o melhor em termos de cobertura apresentando o menor índice de casos de falta de cobertura (25,3%), seguido dos *scanners* S3 e S2, com, respectivamente, 69,6% e 95,9%. O scanner S2 apresentou o maior índice de falta de cobertura, e, no entanto, apresentou porcentagem muito mais baixa de falsos positivos quando comparada aos *scanners* S1 e S3 (1,4%). A porcentagem de falsos positivos dos *scanners* S1 e S3 são próximas (16,1% e 20,6% respectivamente), considerando-se que uma quantidade maior de vulnerabilidades foi analisada para o *scanner* S3.

Também é possível, observando a ., avaliar a eficácia dos scanners de acordo com o tipo de vulnerabilidade que se deseja testar. O scanner S1 apresentou melhores resultados para testes de vulnerabilidades do tipo XSS, pois não apresentou falsos positivos ou falta de cobertura para este tipo de vulnerabilidade. O Scanner S2 apresentou melhores resultados para vulnerabilidades do tipo injeção de SQL, também não detectando nenhum falso positivo ou falta de cobertura. Já para vulnerabilidades CSRF, os três scanners apresentaram resultados distintos, onde a quantidade de casos de falta de cobertura e de falso positivo são inversamente proporcionais (quanto menor o índice de falhas de cobertura, maior o índice de falsos positivos e vice-versa). Nesse caso, deve-se utilizar, de forma complementar, outros critérios de seleção, como por exemplo, uma segunda vulnerabilidade mais importante, para determinar a melhor ferramenta a ser adotada.

Como esperado, o índice de falsos positivos tende a ser diretamente proporcional à capacidade da ferramenta de detectar vulnerabilidades. Estes resultados, que apresentaram altos índices de falta de cobertura e falso positivo, mostram as limitações dos *scanners* de vulnerabilidades. Para aplicações *Web* críticas, vários *scanners* devem ser utilizados e testes complementares não devem ser descartados do procedimento.

Síntese dos resultados. A abordagem proposta na presente dissertação foi utilizada em um estudo de caso para avaliar a eficácia de três scanners de vulnerabilidades de grande inserção no mercado. Para isso utilizou-se duas aplicações web que fazem parte do segmento de gestão de informações. Os resultados do estudo de caso mostraram que:

- em média 63,6% das falhas de software injetadas nas aplicações afetaram os resultados dos *scanners*. Em aproximadamente 20% dos casos, a alteração dos resultados foram verificadas, ao mesmo tempo, em pelo menos dois dos *scanners;*

- *scans* executados na App1 apresentaram mais casos de falta de cobertura e falsos positivos do que os executados na App2 devido ao fato da App2 ser menor e mais modularizada;

- tecnologias empregadas para o desenvolvimento da aplicação (*Hibernate, Struts, Java Server Faces*) implementam recursos que reduzem o índice de vulnerabilidades de segurança. Mesmo assim, algumas vulnerabilidades foram detectadas nas aplicações em fase operacional, reforçando a importância dos testes de segurança;

- poucas vulnerabilidades em comum entre os três scanners foram detectadas. Essa diferença é um indicador de que estas ferramentas implementam diferentes formas de executar seus testes de intrusão e que os resultados de diferentes *scanners* podem ser bastante diferentes;

- S1 mostrou-se o melhor scanner em termos de cobertura apresentando o menor índice de casos de falta de cobertura (25,3%), seguido dos *scanners* S3 e S2, com, respectivamente, 69,6% e 95,9%. O scanner S2 apresentou o maior índice de falta de cobertura, e, no entanto, apresentou porcentagem muito mais baixa de falsos positivos quando comparada aos *scanners* S1 e S3 (1,4%). A porcentagem de falsos positivos dos *scanners* S1 e S3 são próximas (16,1% e 20,6%

respectivamente), considerando-se que uma quantidade maior de vulnerabilidades foi analisada para o *scanner* S3;

- o scanner S1 apresentou melhores resultados para testes de vulnerabilidades do tipo XSS, pois não apresentou falsos positivos ou falta de cobertura para este tipo de vulnerabilidade. O scanner S2 apresentou melhores resultados para vulnerabilidades do tipo injeção de SQL, também não detectando nenhum falso positivo ou falta de cobertura. Já para vulnerabilidades CSRF, os três scanners apresentaram resultados distintos, onde a quantidade de casos de falta de cobertura e de falso positivo são inversamente proporcionais (quanto menor o índice de falhas de cobertura, maior o índice de falsos positivos e vice-versa);

- o índice de falsos positivos tende a ser diretamente proporcional à capacidade da ferramenta de detectar vulnerabilidades.

5.4 Automatização

A metodologia proposta é escalável e pode ser utilizada em aplicações maiores e mais complexas. Também pode ser automatizada por meio de ferramentas que realizem, de forma automática, os procedimentos de injeção de falhas e realização de ataques. Para verificar essa escalabilidade e automatização, foi realizado um estudo onde uma aplicação utilizada comercialmente foi testada contra vulnerabilidades e os resultados dos scanners de vulnerabilidade foram analisados para verificar sua eficácia. O procedimento de injeção de falhas foi automatizado por meio da ferramenta J-SWFIT [17].

A aplicação testada é utilizada para controlar chamadas a um centro de atendimento de uma grande empresa, gerenciando a agenda e procedimentos de profissionais na área da saúde. Por ser comercialmente disponível, o nome da aplicação não será divulgado e esta será referenciada como “AppX”. Ela foi desenvolvida na linguagem Java e utiliza tecnologias como Hibernate*,* Ajax e o *framework* Java Server Faces [91]. Possui aproximadamente 45 mil linhas de código e quatro bases de dados integradas no MySQL.

Os scanners de vulnerabilidades utilizados no presente estudo foram os mesmos dos estudos anteriores e serão chamados S1, S2 e S3. A injeção de falhas foi realizada através da ferramenta J-SWFIT [17], que permite a injeção de falhas quando o código fonte da aplicação não está disponível. Como a ferramenta para realizar ataques de forma automática ainda está em processo de desenvolvimento, os ataques foram baseados nas árvores modeladas (também

considerada como base para o desenvolvimento da ferramenta para ataques) e utilizadas nos estudos prévios [42].

Os resultados do estudo de caso sobre a AppX estão na Tabela 5.8. Somente um caso de uso foi selecionado e 38 falhas do tipo MFC (Tabela 5.1 desta dissertação) foram injetadas entre classes de persistência e de modelo de negócios. Neste estudo, as falhas injetadas não afetaram os resultados dos scanners.

Tabela 5.8. Falta de cobertura e falsos positivos referentes à aplicação AppX.

	Vulnerabilidades não detectadas (falta de cobertura)	Falsos positivos
S1	1	2
S2	2	0
S3	2	0

A baixa quantidade de vulnerabilidades apresentada na Tabela 5.8 se deve ao uso das tecnologias como Java Server Faces e Hibernate, que, respectivamente, implementam recursos de filtros contra execução de *scripts* e encapsulamento de consultas à base de dados. Também se deve ao fato das falhas de software injetadas não terem sido responsáveis por novas vulnerabilidades de segurança, e, consequentemente, não alteraram os resultados dos *scanners*.

A Tabela 5.9 apresenta os casos de falta de cobertura e falsos positivos de acordo com os tipos de vulnerabilidades analisadas.

Tabela 5.9. Falta de cobertura e falsos positivos referentes aos tipos de vulnerabilidades.

	S1			S2			S3		
	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF	XSS	Inj. SQL	CSRF
Vulnerabilidades existentes nas aplicações	1	0	2	1	0	2	1	0	2
Vulnerabilidades analisadas	1	2	1	1	0	0	0	0	1
Detectadas corretamente	1	0	1	1	0	0	0	0	1
Falha de cobertura	0	0	1	0	0	2	1	0	1
Falso positivo	0	2	0	0	0	0	0	0	0

Neste estudo de caso foram confirmadas a presença de 1 vulnerabilidade XSS, nenhuma vulnerabilidade do tipo injeção de SQL e 2 vulnerabilidades CSRF na aplicação. Na Tabela 5.9 observa-se que os scanners S1 e S2 identificaram corretamente a vulnerabilidade do tipo XSS existente. Em relação a vulnerabilidades do tipo injeção de SQL, o Scanner S1 detectou dois falsos positivos. E, para vulnerabilidades CSRF, o scanner S1 detectou apenas uma delas. O mesmo ocorreu com o scanner S3. Para este tipo de falha, todos os scanners apresentaram falha de cobertura.

De acordo com a análise de falta de cobertura e falsos positivos referentes aos tipos de vulnerabilidades, observa-se que os scanners S1 e S2 se mostraram mais eficientes para detectar vulnerabilidades do tipo XSS. Os scanners S2 e S3 se mostraram mais eficientes para detectar vulnerabilidades do tipo injeção de SQL, pois não detectaram falsos positivos. Os scanners S1 e S3, apesar de apresentarem uma falha de cobertura cada, foram mais eficazes para a detecção de vulnerabilidades CSRF, pois detectaram uma das falhas corretamente.

A Figura 5.5 ilustra, para cada tipo de vulnerabilidade analisada, a relação entre as detecções pelos *scanners* S1, S2 e S3. A intersecção entre as áreas dos círculos representa a quantidade de mesmas vulnerabilidades detectadas por mais de um scanner. Os casos de falta de cobertura não foram considerados para a ilustração, ou seja, foram consideradas somente as vulnerabilidades detectadas, inclusive os falsos positivos.

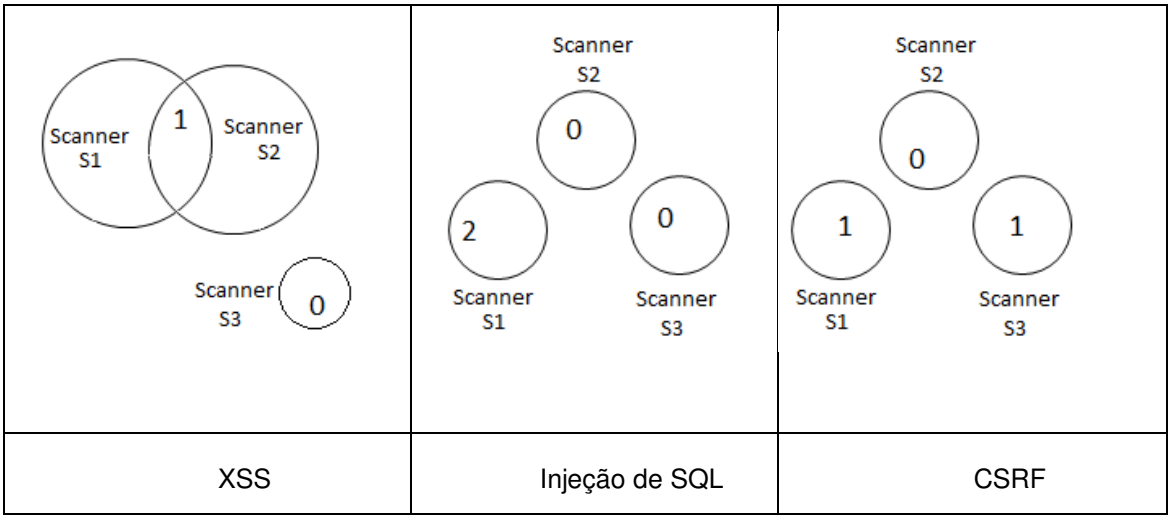


Figura 5.5. Relação entre as detecções dos tipos de vulnerabilidades pelos scanners.

Como é possível observar na Figura 5.5, somente uma vulnerabilidade do tipo XSS foi identificada por dois scanners diferentes. Novamente, essa diferença reforça a sugestão de combinar vários scanners para obtenção de melhores resultados ao testar aplicações *Web*.

A Tabela 5.10 mostra a porcentagem de falta de cobertura e falsos positivos de cada scanner analisado.

Tabela 5.10. Falta de cobertura e falsos positivos referentes aos scanners.

	S1	S2	S3
Vulnerabilidades analisadas	5	3	3
Vulnerabilidades detectadas corretamente	2 (40%)	1 (33,5%)	1 (33,5%)
Falta de cobertura	1 (20%)	2 (66,5%)	2 (66,5%)
Falso positivo	2 (40%)	0 (0%)	0 (0%)

Como o índice de falsos positivos tende a ser diretamente proporcional à capacidade da ferramenta de detectar vulnerabilidades, o scanner S1 apresentou maior porcentagem de vulnerabilidades detectadas corretamente (40%), mas também apresentou maior índice de falso positivo (40%). Os scanners S2 e S3 tiveram resultados semelhantes, apresentando baixo índice de falso positivo (0%), mas com alto índice de falta de cobertura (66,5%).

Dentre os resultados do estudo de caso, em termos de escalabilidade e automatização, a metodologia, aliada à ferramenta de injeção de falhas se mostrou adequada para ser utilizada em aplicações *Web* de grande porte, permitindo a análise da eficácia dos scanners de vulnerabilidades e também realizar testes de segurança nestas aplicações. Dentre as vantagens da utilização do conjunto composto pela metodologia e ferramentas, estão a redução no tempo de realização dos testes (pois é feito de maneira automática e transparente ao testador) e a possibilidade de realizar estes testes em aplicações cujo código fonte não esteja disponível. Futuramente, integrando a ferramenta para automatizar os ataques, pode-se reduzir ainda mais o tempo de execução dos testes e otimizar a obtenção dos resultados.

5.5 Resumo

Neste capítulo foi apresentada uma metodologia para avaliar a eficácia de *scanners* de vulnerabilidades. Esta metodologia foi descrita em sete etapas e, diante de uma pequena adaptação, pode também ser utilizada para testes de segurança em aplicações *Web* cuja aplicação de um *scanner* não seja possível, normalmente por motivos financeiros.

Um estudo de caso em aplicações do segmento de gestão de informações permitiu mostrar a aplicabilidade da metodologia para avaliar a eficácia de três *scanners* de vulnerabilidades comercialmente disponíveis e líderes de mercado. Técnicas de injeção de falhas permitiram que a ativação de falhas fosse acelerada de forma a criar potenciais vulnerabilidades que devem ser

detectadas e, com isso, contribuir com maiores níveis de dependabilidade das aplicações sob teste. Uma análise do contexto do código fonte permitiu avaliar o efeito das falhas sobre o comportamento da aplicação e consequentemente sobre o comportamento da ferramenta. Modelos de árvores de ataques permitiram que ataques fossem realizados com o objetivo de avaliar se determinada vulnerabilidade detectada pelo *scanner* é um falso positivo ou se é uma falta de cobertura na detecção de vulnerabilidades dessa ferramenta.

Outro estudo de caso foi desenvolvido para verificar a escalabilidade da metodologia. Com auxílio de uma ferramenta de injeção de falhas, uma aplicação de uso comercial foi testada e os resultados dos scanners foram avaliados, apresentando resultados coerentes com os da análise prévia.

Os resultados, de modo geral, mostraram que boa parte das falhas de software injetadas afetou os resultados dos *scanners*, mostrando a importância desta técnica ao se validar segurança de aplicações *Web*. Outro resultado importante foi o alto índice de falta de cobertura e falsos positivos apresentados pelas ferramentas, indicando as limitações deste segmento para o estudo realizado.

6 Capítulo 6

Conclusão

Síntese do trabalho. Diante do atual cenário que retrata a falta de segurança de aplicações *Web* – especialmente no segmento de gestão de informações – e as limitações dos scanners de vulnerabilidades que apresentam, comprovados por estudos prévios, resultados de baixa eficácia, a presente dissertação definiu uma abordagem para validar a eficácia destes *scanners* e, com isso, contribuir para o aumento do nível de dependabilidade das aplicações.

A abordagem proposta é baseada em técnicas de injeção de falhas e modelos de árvores de ataques. Para realizar a injeção de falhas foi necessário desenvolver previamente um estudo de campo para identificar falhas de software específicas do paradigma de orientação a objetos e da linguagem Java que, atualmente, é utilizada para desenvolver grande parte das aplicações *Web* [28]. Entre as falhas de software investigadas foram identificadas falhas responsáveis por vulnerabilidades de segurança. Este estudo de campo compôs o *faultload* necessário para que as falhas fossem injetadas de maneira realista nos experimentos que utilizam a abordagem. O *faultload* também foi utilizado para o desenvolvimento da ferramenta de injeção de falhas J-SWFIT [17]. Outro estudo prévio que foi importante para a definição da abordagem analisou o efeito de falhas de software sobre o comportamento da aplicação – e consequentemente do *scanner* – em relação a vulnerabilidades de segurança [32]. Nesse estudo, falhas de software injetadas foram responsáveis por novas vulnerabilidades nas aplicações. O contexto do código fonte e as tecnologias atualmente utilizadas para a construção das aplicações foram analisados a fim de entender como as falhas injetadas afetam a aplicação, auxiliando a composição da metodologia.

Para a modelagem de árvores de ataques, que consiste em um dos passos da abordagem, foi desenvolvido um estudo para identificar diferentes maneiras de explorar vulnerabilidades em aplicações *Web* consideradas críticas [42]. A partir desse estudo as árvores foram modeladas e casos de testes foram obtidos a partir dessas árvores. Estas árvores podem até mesmo ser utilizadas para testes de segurança de forma independente. Também servem de base para a construção de uma ferramenta para injeção de ataques.

A abordagem foi proposta em sete etapas e um estudo de caso foi realizado para validar a eficácia de três diferentes scanners comercialmente disponíveis. Os resultados proporcionaram diretrizes para a seleção de ferramentas, indicando que, para ter boa cobertura dos testes de segurança, o usuário pode combinar vários *scanners* ao invés de confiar nos resultados de somente um deles. Ou então selecionar a ferramenta mais conveniente de acordo com prioridades pré-estabelecidas, como por exemplo, a prioridade em detectar determinado tipo de vulnerabilidade (alguns scanners apresentaram melhores resultados ao detectar determinados tipos de

vulnerabilidades). Testes complementares também não devem ser descartados do procedimento, principalmente para aplicações *Web* críticas.

Discussão dos resultados. Embora a quantidade de aplicações utilizada no estudo de caso seja relativamente pequena e essas aplicações representem um único segmento dentre os vários existentes, a abordagem se mostrou adequada e apresentou resultados satisfatórios, cumprindo os objetivos a que se propôs, que é avaliar a eficácia de scanners de vulnerabilidades.

A injeção de falhas nas aplicações sob teste fez com que falhas de software fossem ativadas e vulnerabilidades de segurança fossem previstas. Este procedimento foi importante para assegurar que vulnerabilidades de segurança a serem detectadas pelos scanners estivessem presentes nas aplicações. Também é bastante relevante para testes de segurança realizados de maneira independente, pois permite prever vulnerabilidades e aplicar contramedidas para corrigi-las ou reduzir a severidade dos danos causados quando exploradas por ataques.

O estudo das vulnerabilidades de segurança e a modelagem de árvores de ataques proporcionaram o conhecimento sobre essas vulnerabilidades e sobre os procedimentos que um atacante pode utilizar para explorá-las. As árvores de ataque permitiram que as diferentes formas possíveis de exploração de determinada vulnerabilidade de segurança fossem representadas, assegurando bons níveis de cobertura em relação aos testes contra a vulnerabilidade em questão.

A metodologia é escalável e pode ser utilizada até mesmo em aplicações *Web* de grande porte com o auxílio de ferramentas de injeção de falhas e realização de ataques. O conjunto (metodologia e ferramentas) apresenta uma boa técnica para testar aplicações contra vulnerabilidades de segurança, principalmente quando o código fonte da aplicação sob teste não está disponível.

A principal contribuição desta dissertação é a abordagem para validar a eficácia dos scanners e a sua aplicação em um estudo de caso. Dentre as demais contribuições, estão: i) a definição do *faultload;* ii) o estudo da relação entre as falhas de software e vulnerabilidades de segurança; iii) os modelos de árvores de ataques. Para a construção de um futuro *benchmark* para análise de *scanners* de vulnerabilidades, o conjunto experimental se mostra adequado para formar uma das dimensões necessárias, denominada dimensão experimental.

Limitações. Há duas limitações no estudo realizado:

i) Os experimentos, em uma primeira fase, foram realizados manualmente em função da análise do código fonte e também porque ainda não se dispõe de ferramentas para automatizá-los completamente. Ou seja, a quantidade de aplicações e de vulnerabilidades investigadas foi limitada neste estudo. As aplicações também representam um único segmento que, apesar de bastante representativo, não permite a generalização dos resultados.

ii) Como os testes de segurança são direcionados para validar os resultados dos scanners, não foi adotado um critério de cobertura da aplicação sob teste. Novas vulnerabilidades que

caracterizam falta de cobertura foram detectadas em função da diferença dos resultados dos scanners; ou seja, se em uma versão da aplicação (com falha injetada) um scanner detectou determinada vulnerabilidade, mas essa mesma vulnerabilidade já existia na aplicação original (sem falha injetada) e não foi detectada, ela é caracterizada como falta de cobertura na aplicação original. Então, a aplicação pode apresentar outras vulnerabilidades que não tenham sido detectadas pelo scanner em momento algum e, consequentemente, não foram validadas ou identificadas pelos testes.

Trabalhos Futuros. Diante das limitações apresentadas, são levantadas as seguintes perspectivas para este trabalho:

- Ampliar a quantidade de vulnerabilidades analisadas, estudando e modelando árvores de ataques para outras vulnerabilidades de segurança. Isto permite que os testes verifiquem mais tipos de vulnerabilidades presentes na aplicação, aumentando o nível de dependabilidade da aplicação sob teste.

- Automatizar o procedimento proposto para a abordagem. Para isso, foi desenvolvido um trabalho paralelo a este, resultando em uma ferramenta para injeção de falhas em aplicações Java, a J-SWFIT [17]. Outro trabalho está sendo desenvolvido paralelamente a este e deve resultar em uma ferramenta para injeção automática de ataques.

- Realizar estudos de casos utilizando aplicações web que representem outros segmentos e avaliar o uso da abordagem nesses segmentos de forma que, se necessário, por razões de tecnologias e estruturas utilizadas, adaptações na abordagem sejam desenvolvidas.

- Medir a cobertura de código alcançada pela realização dos ataques. O objetivo é realizar ataques que alcancem boa cobertura de código esperando que isso resulte em melhor cobertura de vulnerabilidades.

Referências

[1] Avizienis, A.; Laprie, J. C.; Randell, B.; Landwehr, C. “Basic concepts and taxonomy of dependable and secure computing”. IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, Jan.-Mar 2004. Páginas 11 – 33.

[2] Durães, J.; “Faultloads baseadas em falhas de software para testes padronizados de cofiabilidade.”. Tese de Doutorado. Faculdade de Ciências e Tecnologia, Universidade de Coimbra, Portugal, 2005.

[3] Fonseca, J.; Vieira, M. “Mapping software faults with web security vulnerability”. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2008), Anchorage, USA, junho/2008. Páginas 257-266

[4] ICAT/NIST. National Institute of Standards and Technology. Disponível em www.icat.nist.gov. Último acesso em março/2009.

[5] IEEE – Institute of Electrical and Electronics Engineers. Standard Glossary of Software Engineering Terminology, Document Number: IEEE 610.12-1990, Maio/1990.

[6] Leite, J.; Orlando, G. “Software”. In II SCTF, cap. 4 do mini-curso intitulado: Introdução à Tolerância a Falhas, Campinas, SP, Brasil, 1987.

[7] Laprie, J.C. “Dependability – Its Attributes, Impairments and Means”. In Predictability Dependable Computing Systems, B. Randell, J.-C. Laprie, H. Kopetz, B. Littlewood, Editors, Springer, Berlim, Alemanha, 1995, pp. 3-18.

[8] Acunetix. Disponível em www.acunetix.com. Último acesso em março/2009.

[9] N-Stalker. Disponível em www.nstalker.com. Último acesso em março/2009.

[10] Rational AppScan. Disponível em www.ibm.com/developerworks/rational/products/appscan . Último acesso em março/2009.

[11] Cenzic Hailstorm. Disponível em www.cenzic.com. Último acesso em março/2009.

[12] HP WebInspect. Disponível em https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__. Último acesso em março/2009.

[13] Sandcat. Disponível em http://www.syhunt.com/?section=sandcat. Último acesso em janeiro/2010.

[14] Burp Suite. Disponível em http://www.portswigger.net/suite/. Último acesso em novembro/2009.

[15] Gamja. Disponível em http://sourceforge.net/projects/gamja/. Último acesso em novembro/2009.

[16]Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.C.; Laprie, J.C.; Martins, E.; Powell, D. "*Fault Injection for Dependability Validation – A Methodology and some Applications*". In IEEE Transactions on Software Engineering, 16 (2), 1990. Páginas 166-182.

[17]Sanches, B., Moraes, R. Desenvolvimento de Ferramenta para Injeção de Falhas de Software - J-SWFIT. In Anais do XVII Congresso Interno de Iniciação Científica da UNICAMP, 2009.

[18]Kanoun, K.; Madeira, H.; Moreira, F.; Cin, M.; García, J. "DBench - Dependability Benchmarking". Fifth European Dependable Computing Conference (EDCC-5), Budapeste, Hungria, Abril/2005.

[19]Madeira, H.; Costa, D.; Vieira, M. "On the emulation of software faults by software fault injection". In Proceedings of the International Conference on Dependable Systems and Networks, 2000, pp. 417 – 426.

[20]Carreira, J., Madeira, H., Silva, J. "Xception: Software Fault Injection and Monitoring in Processor Functional Units." In Proceedings of 5th IFIP International Working Conference on Dependable Computing for Critical Applications, Urbana-Champaign, EUA, 1995, pp. 135-149.

[21]Christmansson, J.; Chillarege, R. "Generation of an Error Set that Emulates Software Faults-Based on Fields Data." In Proceedings of 26th International Symposium on Fault-Tolerant Computing, Sendai, Japan, 1996, pp. 304-13.

[22]Durães, J.; Madeira, H. "Emulation of Software Faults: A Field Data Study and Practical Approach". IEEE Transactions on Software Engineering, volume 32, n. 11, Nov. 2006, pp. 849-867.

[23]Perry, D. E.; Evangelist, W. M. "An empirical study of software interface faults". In Proceedings of the IEEE International Symposium on New Directions in Computing, 1985, pp. 32-38.

[24]Sullivan, M.; Chillarege, R. "Software defects and their impact on systems availability – A study of field failures on operating systems". In Proceedings of the 21st IEEE Fault Tolerant Computing Symposium, Sendai, Japan, 1991, pp.2-9.

[25]Sullivan, M.; Chillarege, R. "A Comparison of Software Defects in Database Management Systems and Operating Systems". In Proceedings of the 22nd IEEE International Symposium on Fault Tolerant Computing, Jul. 1992, pp. 475-484.

[26]Chillarege, R.; Bhandari, I. S.; Chaar, J. K.; Halliday, M. J., Moebus, D.S.; Ray, B. K.; Wong, M. Y. "Orthogonal Defect Classification – A Concept for In-Process Measurements" IEEE Transactions On Software Engineering. Vol. 18, n. 11, 1992, pp. 943-956.

[27]Seixas, N.; Fonseca, J.; Vieira, M.; Madeira, H. "Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study." In 20th International Symposium on Software Reliability – ISSRE. Nov. 2009, pp. 129-135.

[28]Basso, T.; Moraes, R. Sanches, B.P.; Jino, M. "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults." In Workshop de Testes e Tolerância a Falhas – WTF 2009, João Pessoa, Brasil, 2009. PP. 1-13.

[29]Bau, J.; Bursztein, E.; Gupta, D; Mitchell, J. “State of the Art: Automated Black-Box Web Application Vulnerability Testing”. IEEE Symposium on Security and Privacy, Oakland, USA. Maio/2010. Páginas 332-345.

[30]Fonseca, J.; Vieira, M.; Madeira, H.; “Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks”, 13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007), Melbourne, Victoria, Australia, Dezembro/2007.

[31]Vieira, M.; Antunes, N.; Madeira, H. "Using Web Security Scanners to Detect Vulnerabilities in Web Services". Practical Experience Report. DSN 2009, Lisboa, Portugal.

[32]Basso, T., Fernandes, P.C.S., Jino, M., Moraes, R., “Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool”. In Proceedings of IEEE DSN Workshops (DSNW), Chicago, EUA, junho/2010

[33]Schneier, B., “Attack Trees: Modeling Security Threats”, Dr. Dobb’s Journal, dezembro/1999.

[34]McDermott, J. “Attack Net Penetration Testing”. In The 2000 New Security Paradigms Workshop, 2001, pp. 15-22.

[35]Hussein, M., Zulkernine, M.: “UMLintr: a UML profile for specifying intrusions”. In Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, Tucson, AZ, USA, 2006, pp. 279–288.

[36]Wang, L.; Wong, E.; Xu, D. “A Threat Model Driven Approach for Security Testing”. In: Proceedings of the Third International Workshop on Software Engineering for Secure Systems, Minneapolis, MN, USA, maio/2007.

[37]Martins, E.; Morais, A.; Cavalli, A. “Generating attack scenarios for the validation of security protocol implementations.” 2nd. Brazilian Workshop on Systematic and Automated Software Testing (SAST). Campinas, Brasil, Maio/2008.

[38]Byres, E. J., Franz, M.; Miller, D., “The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems”. International Infrastructure Survivability Workshop (IISW '04), IEEE, Lisboa, Portugal, Dezembro/2004

[39]Edge, K., Raines, R., Grimaila, M., Baldwin, R., Bennington, R., and Reuter. “The Use of Attack and Protection Trees to Analyze Security for an Online Banking System.” In Proceedings of the 40th Annual Hawaii international Conference on System Sciences. Janeiro/2007, pp 144b.

[40]Fung, C., Chen Y. L., Wang, X., Lee, J. Tarquini, R., Anderson, M. Linger, R. “Survivability analysis of distributed systems using attack tree methodology”. Military Communications Conference – MILCOM. Outubro/2005, pp 583-589

[41]Lin, X., Zavarsky, P., Ruhl, R., and Lindskog, D. “Threat Modeling for CSRF Attacks.” Proceedings of the 2009 international Conference on Computational Science and Engineering, Vancouver, Canadá, outubro/2009, pp 486-491.

[42]Fernandes, P.C.S, Basso, T., Moraes, R., Jino, M. “Attack Trees Modeling to Security Tests in Web Applications”. Artigo submetido para The 12th IEEE International High Assurance Systems Engineering Symposium - HASE 2010.

[43]Java Place. Disponível em http://javaplace.blogspot.com/. Último acesso em dezembro/2009.

[44]GUJ. “Notícias, artigos e o maior fórum brasileiro sobre Java”. Disponível em http://www.guj.com.br/home.index.logic. Último acesso em dezembro/2009.

[45]Forum invaders. “Seu portal de conhecimentos na internet”. Disponível em http://www.forum-invaders.com.br/phpBB/. Último acesso em dezembro/2009.

[46]National Vulnerability Database. Disponível em http://nvd.nist.gov/. Último acesso em janeiro/2010.

[47]National Institute of Standards and Technology. Disponível em http://www.nist.gov/index.html. Último acesso em janeiro/2010.

[48]OWASP – “The Free and Open Application Security Community”. Disponível em http://www.owasp.org/index.php/Main_Page. Último acesso em janeiro/2010.

[49]Uto, N.; Melo, S. P. “Vulnerabilidades em Aplicações Web e Mecanismos de Proteção”. In IX Simpósio Brasileiro de Segurança da informação e de sistemas computacionais – SBSEG, cap. 6 dos mini-cursos SBSEG, Campinas, Brasil, outubro/2009.

[50]Halfond, W. G.; Viegas, J.; Orso, A.“A Classification of SQL-Injection Attacks and Countermeasures”, In Proceedings of the International Symposium on Secure Software Engineering - ISSSE 2006, Arlington, Virginia, Março/2006.

[51]Jovanovic, N.; Kirda, E.; Kruegel, C. “Preventing Cross Site Request Forgery Attacks.” In Securecomm and Workshops, Agosto/2006, pp. 1–10.

[52]Meier, J. D.; Mackman, A.; Vasireddy, S.; Dunner, M. Escamilla, R.; Murukan, A. “Improving Web Application Security – Threats and Countermeasures”. Microsoft Corporation, jun/2003.

[53]Xiao, S.; Deng, L.; Li, S.; Wang, X. “Integrated TCP/IP Protocol Software Testing for 122 Vulnerability Detection”. International Conference on Computer Networks and Mobile Computing, Shanghai, Outubro/2003.

[54]Tsipenyuk, K.; Chess, B.; McGraw, G. “Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors”. IEEE Security & Privacy, IEEE Computer Society, 2005.

[55]McGraw, G. “Software Security”. IEEE Security & Privacy, IEEE Computer Society, 2004.

[56]CSI. Computer Security Institute. Disponível em http://www.gocsi.com/. Último acesso em março/2009.

[57]L. Gordon, M. Loeb, W. Lucyshyn, R. Richardson. “Computer crime and security survey”. Computer Security Institute, 2007. Disponível em http://i.cmpnet.com/v2.gocsi.com/pdf/CSISurvey2007.pdf. Último acesso em agosto/2010.

[58]R. Richardson. “CSI Computer crime and security survey”. Computer Security Institute, 2008. Disponível em http://www.cse.msstate.edu/~cse6243/readings/CSIsurvey2008.pdf. Último acesso em agosto/2010.

[59]Acunetix News. Disponível em http://www.acunetix.com/news/security-audit-results.htm. Último acesso em março/2009.

[60]2009 CSI Computer Crime and Security Survey – Executive Summary. Computer Security Institute, 2009. Disponível em http://pathmaker.biz/whitepapers/CSISurvey2009.pdf. Último acesso em agosto/2010.

[61]Gartner Group. Disponível em www.gartner.com. Último acesso em março/2009.

[62]OWASP Top 10. Disponível em http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, Fevereiro/2010.

[63]Hsueh, M.C.; Tsai, T.; Iyer. R.; “Fault Injection Techniques and Tools.” In IEEE Computer, 1997, pp. 75-82.

[64]Clark, J.A.; Pradhan, D.K. “Fault injection: a method for validating computer-system dependability”. IEEE Computer, Volume 28, Jun 1995, pp. 47 - 56.

[65]Carreira, J.V.; Costa, D.; Silva, J.G. “Fault injection spot-checks computer system dependability”. IEEE Spectrum, Volume 36, Aug 1999, pp. 50 - 55.

[66]Voas, J.; “A tutorial on software fault injection”. Disponível em http://www.cigital.com/papers/download/spectrum2000.pdf. Último acesso em janeiro/2010.

[67]Kalyanakrishnam, M.; Kalbarczyk, Z.; Iyer, R. “Failure Data Analysis of a LAN of Windows NT Based Computers”, Symposium on Reliable Distributed Database Systems, SRDS-18, October, Switzerland, 1999. Páginas 178-187.

[68]Lee, I.; Iyer, R.K.. “Software Dependability in the Tandem GUARDIAN System”, IEEE Trans. on Software Engineering, vol. 21, no. 5, May 1995. Páginas 455-467.

[69]Lyu, M. “Handbook of Software Reliability Engineering”, IEEE Computer Society Press, MCGraw-Hill, 1996.

[70]Musa, J. “Software Reliability Engineering”, McGraw-Hill, 1996.

[71]Voas, J. “Fault injection for the masses”. IEEE Computer. Volume 30, Issue 12, Dezembro 1997. Páginas 129-130.

[72]Voas, J.; “Software Fault Injection – Inoculating Programs Against Errors”. Wiley Computer Publishing, 1998.

[73]Arlat, J.; Crouzet, Y.; Laprie, J.-C. “Fault injection for dependability validation of fault-tolerant computing systems”. In Proceedings of the 19th International Symposium on Fault-Tolerant Computing, Jun 1989, pp. 348 – 355.

[74]Arlat, J.; Costes, A.; Crouzet, Y.; Laprie, J.C.; Powell, D. “Fault injection and dependability evaluation of fault-tolerant systems”. IEEE Transactions on Computers, Volume 42, Aug 1993, pp. 913 – 923

[75]Dawson, S.; Jahanian, F.; Mitton, T. “Orchestra: A fault injection environment for distributed systems”. In Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS), Sendai, Japan, Jun 1996, pp 404-414.

[76]Barcelos, P. P. O.; Leite, F. O.; Weber, Taisy Silva. “Implementação de um Injetor de Falhas de Comunicação”. Anais do SCTF ’99 – VIII Simpósio de Computação Tolerante a Falhas. Campinas, Brasil, Jul 1999, pp. 225-239.

[77]Drebes, R.J.; Jacques-Silva, G.; Trindade, J. F.; Weber, T. S. “A Kernel-based Communication Fault Injector for Dependability Testing of Distributed Systems”. In Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD-3), 2005, Haifa, Israel.

[78]Martins, E., Rubira, C., Leme, N. “Jaca: A reflective fault injection tool based on patterns.” In: Proceedings of the 2002 International Conference on Dependable Systems & Networks, Washington D.C. USA, 2002, pp. 483-487.

[79]Silva, G., Moraes, R., Weber, T. “Validando Sistemas Distribuídos desenvolvidos em Java utilizando Injeção de Falhas de Comunicação por Software.” In Proceedings of V Workshop de Testes e Tolerância a Falhas – WTF2004, Gramado, Brasil, 2004.

[80]Steffan, J.; Schumacher, M. “Collaborative Attack Modeling”. In Proceedings of the 2002 ACM Symposium on Applied Computing, 2002, pp. 253–259.

[81]Morais, A.N.P.; “Injeção de Ataques Baseado em Modelo para Teste de Protocolos de Segurança”. Dissertação de Mestrado. Instituto de Computação, Universidade Estadual de Campinas, Brasil, 2009.

[82]Moore, A.P.; Ellison, R.J.; Linger, R.C. “Attack Modeling for Information Security and Survivability”. Technical Note CMU/SEI-2001-TN-001, março/2001.

[83]CERT – “Coordination Center at Carnegie Mellon University’s Software Engineering Institute”. Disponível em http://www.cert.org/cert/. Último acesso em janeiro/2010.

[84]Eberom. “A CRM and Project Management Tool”. Disponível em http://sourceforge.net/projects/eberom/. Último acesso em janeiro/2010.

[85]Hibernate. Disponível em https://www.hibernate.org/. Último acesso em Dezembro/2009.

[86]Struts. Disponível em http://struts.apache.org/. Último acesso em janeiro/2010.

[87]JasperReports. Disponível em http://jasperforge.org/plugins/project/project_home.php?projectname=jasperreports. Último acesso em janeiro/2010.

[88]Portal do software público brasileiro – Amadeus. Disponível em http://www.softwarepublico.gov.br/ver-comunidade?community_id=9677539. Último acesso em janeiro/dezembro/2009.

[89]OWASP – Preventing SQL injection in Java. Disponível em http://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java#Hibernate, Dezembro/2009.

[90]CWE – Common Weakness Enumeration. Disponível em http://cwe.mitre.org/data/definitions/564.html, Dezembro/2009.

[91]Java Server Faces. Disponível em http://java.sun.com/javaee/javaserverfaces, Junho/2010.

Apêndice A

Publicação

Basso, T.; Moraes, R.; Sanches, B.; Jino, M. "***An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults***". Workshop de Testes e Tolerância a Falhas (WTF 2009), realizado em conjunto com o *Fourth Latin-American Symposium on Dependable Computing*, em João Pessoa – PB, Brasil em Setembro de 2009.

An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults

Tania Basso[1], Regina L. O. Moraes[2], Bruno P. Sanches[2], Mario Jino[1]

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP, Brazil

[2]Faculdade de Tecnologia
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP, Brazil

{taniabasso,brunopsanches}@gmail.com, regina@ceset.unicamp.br, jino@dca.fee.unicamp.br

Abstract. *The knowledge of real software faults representativeness is important to allow the emulation of software faults in a more accurate way through software fault injection techniques. This paper presents a field data study to analyze the representativeness of Java software faults, including security faults. The faults are classified according to a previous field study of C faults representativeness and new types of faults are identified due to the specific characteristics of the Java language structure. Results are compared and show that the mistakes most commonly made by programmers follow a pattern, independently of the programming language.*

1. Introduction

Modern society is increasingly dependent on computer services and, consequently, on the software executed to provide these services. According to Avizienis *et al* [2004], dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable. Assuring that software systems are dependable is important especially in critical systems, where faults can cause major damages or loss of human lives [Gage and McCormick 2004],[Pfleeger 2000].

The current scenario of software production requires more complex elements involved in software production, frequent changes and cost constraints within a short limit of development time. These factors can lead to the insertion of faults into the software. Software faults are recognized as the main cause of computational defects [Kalyanakrishnam *et al* 1999], [Lee and Iyer 1995], [Sullivan and Chillarege 1992] and one of the main challenges in this context is to achieve dependability in these systems. Another demand on dependable systems arises from ubiquitous computation. They need to be universally accessed by users around the world (e.g., online trading and home banking). Most of these systems are web applications. In this scenario, the users may find it difficult to report the failures and the unsolved problems can lead them to put in check the organization credibility or allow hackers to attack the systems, which may have a highly negative impact on users. Although other

potential causes for vulnerability do exist, the root cause of most security attacks are vulnerabilities created by software faults [Fonseca and Vieira 2008].

One way to ensure software dependability is to apply procedures for fault removal and fault prediction during the software validation process. These procedures enable, respectively, the reduction of the number of faults or the fault severity and the evaluation of the consequences of faults remaining in software products. Software fault injection techniques (i.e., deliberate faults insertion) have been very useful to help the software validation process [Durães and Madeira 2006]. Besides the well known usefulness of the technique for this purpose, the results are even more important if the fault emulation is done in a realistic way, which implies the knowledge of the kind of faults found and their distribution in the operational environment. This means that it is necessary to identify the fault representativeness.

This paper presents a field data study on Java systems in production phase (i.e., systems that have gone through several releases), aiming to identify Java specific fault representativeness. The Java programming language was selected for this field data study because it is widely used in current software applications, including web-based systems. The representativeness of a subset of security faults was also analyzed. 574 faults were analyzed and 67 of them are recognized as security faults. The results are compared with those from a previous field data study [Durães and Madeira 2006].

The proposal of this paper is to evaluate if the fault representativeness of Java applications (based on object-oriented paradigm) follows a pattern similar to the software fault representativeness of C applications (based on structural paradigm); it is an important step for the development of a fault injection tool for Java applications.

Some injection tools have been developed [Barcelos *et al* 1999], [Carreira *et al* 1995], [Martins *et al* 2002], [Neves *et al* 2006], and, however, there is no available injection tool to inject specific java software faults, it means, to change Java code structure. Evaluating fault representativeness of Java applications would enable the emulation of software faults in an accurate and realistic way, leading to useful results in the injection Java faults and, consequently, in the validation process.

In summary, we present a software fault classification based on a set of faults found in Java applications and propose an extension to the classification presented by Durães and Madeira [2006], specific to the object-oriented programming paradigm and the Java language.

The structure of the paper is as follows: Section 2 presents background on software faults; Section 3 describes the approach for analyzing and classifying the software faults; Section 4 contains the comparison with the other field study and presents new types of faults due to the Java language characteristics; Section 5 presents our conclusions and future work.

2. Software faults

Software faults are caused by mistakes made by software product programmers and remain in the program source code. The complete elimination of software faults is a difficult or even impossible goal to be achieved [Lyu 1996], [Musa 1996]. In many cases, waiting for the

activation of these faults through normal use is not practicable because it rarely happens (otherwise, the faults would be identified and eliminated by software tests). Fault injection techniques are one option for activation of these faults in order to evaluate the software behavior during the software validation process.

Knowledge of the software fault representativeness is essential for realistic fault injection and to ensure that results obtained during the experiments are very close to those of an operational environment.

To establish real software faults representativeness it is necessary to classify the most frequent software faults, to understand their origins and the way human mistakes occur in the programming task.

A well known classification of software faults is ODC (Orthogonal Defect Classification) [Chillarege 1995], [Chillarege *et al* 1992]. Durães and Madeira [Durães and Madeira 2006] extended this classification and refined their representation to a point that they can be emulated. The generic technique created by [Durães and Madeira 2006] is the G-SWFIT (Generic Software Fault Injection Technique). This technique makes possible the emulation of the most frequently found software faults. It consists of modifying the compiled binary code of software modules by introducing specific changes which correspond to the code that would be generated by the compiler if the intended software fault were in the high level source code. The emulation is done by means of an emulation operator's library. Each operator in this library consists of two binary code instructions. The first one represents a pattern corresponding to a specific fault and the second one represents the necessary changes for the adequate injection of the fault [Durães and Madeira 2006]. Although this technique is generic, the study was developed using the C language and additional investigation is necessary to extend it to other programming languages.

As software faults are strongly dependent on the paradigm and the programming language structure of the application, not all types of faults found in the present field study are represented in Durães and Madeira study [Durães and Madeira 2006]. Thus, it is necessary to define a new classification and a distribution including the new types of software faults which represent characteristics specific of the Java language.

2.1. Java security software faults

Nowadays, due to the widespread use of web applications, the security vulnerabilities in these systems are being more intensely explored by hackers. The work proposed by Fonseca and Vieira [2008] aims to improve this scenario by investigating security faults in applications developed using the PHP (Hypertext Preprocessor) programming language. They want to achieve an understanding of the relationship between certain software defects and security vulnerabilities.

The vulnerabilities analyzed were Cross Site Scripting (XSS) and SQL injection (see [Fonseca and Vieira 2008] for more details). To understand which code is responsible for the security problems the study was based on vulnerabilities correction patches. Comparison of these patches makes it possible to identify and classify real software faults that lead to security vulnerabilities. The authors also define rules to make the patches analysis coherent and to avoid mistakes during fault classification.

The methodology and patches analysis rules proposed by Fonseca and Vieira [2008] are used in our field data study as a basis for security faults identification.

3. Application analysis and fault identification

The field data study reported in this paper aims to investigate software faults in Java applications in order to define faultloads. To determine these faultloads, Java applications source codes are analyzed; an essential condition for the study to be conducted is the availability of the source code of these applications. We also need previous releases of these applications to compare them and to analyze the corrections made and the programming structures which were used to perform these corrections. The analysis of these structures will indicate the types of faults and, consequently, the types of operators that can be used to emulate these faults making use of the G-SWFIT technique.

To identify faults that lead to security vulnerabilities additional information is necessary to link the vulnerability correction with the specific application release (e.g., information on which files or source code fragments were modified exclusively to correct a particular vulnerability).

In accordance with the necessary conditions above and considering their popularity, six open source applications were selected for the field study: Azureus (Vuze) [Vuze 2009], FreeMind [FreeMind 2009], JEdit [JEdit 2009], Phex [Phex 2009], Struts [Struts 2009] and Tomcat [Tomcat 2009].

Azureus (Vuze) and Phex are client applications for sharing files; they use protocols that, briefly, allow content distribution through the web, optimizing the bandwidth consumption under traffic limitations. FreeMind helps on storing and organizing ideas, contributing, for example, to keep personal knowledge bases, recording information of meetings, brainstorms, presentations, planning, and so on. JEdit is a text editor for software systems programmers. Struts is a framework used to create Java web applications. And, finally, Tomcat is a very popular web application server.

To analyze the differences between the selected applications, a diff tool (i.e., a tool that compares two codes and shows their differences) is applied to the different versions of the same application; the changes introduced in the source code are highlighted. It makes possible to visualize the correction, and, consequently, to identify the software fault. The diff tool used is WinMerge 2.10.0.0 [WinMerge 2009]. A total of 14 versions of applications were analyzed. Table 1 shows the analyzed versions and the faults found through ODC type classification for each selected application.

The applications were selected considering the size (lines of code) and number of downloads. Azureus and Tomcat are in a higher maturity level; they are more popular, have more lines of code and more released versions available. So, they have more number of reported bugs and it implies in more number of faults.

The faults are identified through the comparison between bugs correction code of the current release and the source code of the immediately preceding released version. The bugs' correction codes are identified trough changelog files and it is assumed that different code fragments do correct faults.

Table 1. Applications and analyzed versions

Application	Analyzed versions	ODC type					Total Faults	Size (LOC)
		ALG	ASG	CHK	FUN	INT		
Azureus (Vuze)	3.0.5.2 and 3.1.1.0	46	19	14	36	10	125	576958
FreeMind	0.7.1 and 0.8.0	42	10	2	26	10	90	19397
JEdit	4.2pre15 and 4.3pre16	26	10	8	18	9	71	173798
Phex	3.2.6.106 and 3.4.0.110	12	3	1	2	2	20	159206
Struts	1.2.7, 1.2.8 and 1.2.9	48	18	9	4	20	99	158718
Tomcat	6.0.14, 6.0.16 and 6.0.18	98	21	23	4	23	169	279298

Declarations of new structures (i.e., new classes, new variable/object declaration including the "get" and "set" methods for them) were not identified as faults, because a simple structure declaration does not imply a fault correction. If they are associated with a fault correction, this will be seen in the code, and the fault would be properly classified.

For each fault, we attempt to understand the language construction and the program context around it. Then, we try to correlate the fault with a fault type observed in the previous study [Durães and Madeira 2006]. When it is not possible to classify a fault according to the previous study [Durães and Madeira 2006], a new fault type is created, in accordance with the methodology proposed [Durães and Madeira 2006]. Changes in configuration files (i.e., files with "xml" and "properties" extensions) were not considered.

Under the decisions and assumptions above, we were able to identify and classify all the faults found in this field data study. The final classification should be seen as a complementary extension to the previous classification [Durães and Madeira 2006] and can be used to define specific fault emulation operators to inject faults in Java applications.

3.1. Security vulnerabilities patches analysis

Due to the importance of knowing the faults that lead to security vulnerabilities, a subset of security faults was identified. To identify this fault subset, additional decisions were made as explained in the next paragraph. These decisions are based on the methodology proposed by Fonseca and Vieira [2008].

The Struts and Tomcat applications were selected to identify security faults because they are very popular and have many security vulnerabilities reported. For the other applications there was not security vulnerabilities reported, and, consequently, security faults were not analyzed for them. The vulnerability corrections are available in correction patches. Normally, these patches also present many corrections, including faults not linked to security. Thus, it is necessary to identify files and pieces of source code that were modified exclusively to correct a particular vulnerability. This detailed information was found through an internal control number called "number revision" in the patches of the applications under

analysis. From this number it was possible to identify the files that were modified exclusively to correct the particular vulnerability. It is understood that all the modified source code fragments in these files are connected to the present security faults and it does correct the corresponding security vulnerabilities.

4. Results and discussions

A set of 574 Java faults were analyzed. Table 2 presents the most frequent faults found considering the representative set of Java application shown in Table 1. These most frequent faults represent 78.4% of the total Java faults found.

Table 2. Most frequent Java faults and corresponding ODC fault types

Fault nature	Specific fault types	Faults	ODC type
Missing construct	Missing functionality (MFCT)	87	Function
	Missing If construct plus statements (MIFS)	85	Algorithm
	Missing function call (MFC)	60	Algorithm
	Missing if construct plus else plus statements around statements (MIEA)	20	Algorithm
	Missing if construct around statements (MIA)	17	Checking
	Missing parameter in function call (MPFC)	11	Interface
Wrong construct	Wrong function called with same parameters (WFCS)	30	Algorithm
	Wrong value used in variable initialization (WVIV)	23	Assignment
	Wrong data types or conversion used (WSUT)	19	Assignment
	Wrong value used in parameter of function call (WPFL)	18	Interface
	Wrong variable used in parameter of function call (WPFV)	17	Interface
	Wrong logical expression used as branch condition (WLEC)	15	Checking
	Wrong algorithm - small sparse modifications (WALD)	13	Algorithm
	Wrong return value (WRV)	12	Interface
	Wrong algorithm - code was misplaced(WALR)	11	Algorithm
Extraneous construct	Extraneous function call (EFC)	12	Algorithm
Total faults		450	

The most frequent fault types found in our field study are *Missing Functionality (MFCT)* and *Missing if construct plus statements (MIFS)*. Both types present a percentage much higher than that of other fault types, accounting for 30% of the total of analyzed faults. The third most frequent is the *Missing function call (MFC)*, representing 10.5% of the total

faults. This rank is similar to that of a previous field study [Durães and Madeira 2006], where: MIFS is the most frequent type representing 10.7% of the total of faults found in the C language; MFC is the third one, representing 6.7%; and MFCT represents 3.2%. These results indicate that, despite some different frequencies, the mistakes most commonly made by programmers are common to both programming languages (C and Java).

4.1. General Java Faults

From the most frequent fault types in Table 2, seven are also found as the most frequent fault types in Durães and Madeira´s study [Durães and Madeira 2006]. They are MIFS, MFC, MFCT, MIA, WSUT, WPFV and WLEC. Table 3 presents a comparison between the distribution of this field study and the distributions of Durães and Madeira´s field study [Durães and Madeira 2006].

Table 3. General fault distribution across ODC defect-type distribution and comparison with Durães and Madeira´s field study [Durães and Madeira 2006].

ODC defect-type	General Java Faults (GJF)	GJF ODC defect-type distribution (%)	C faults distribution [Durães and Madeira 2006] (%)
Algorithm	272	47.4	40.1
Assignment	81	14.1	21.4
Checking	57	9.9	25
Function	90	15.7	6.1
Interface	74	12.9	7.3
total	574		

Results show that general Java faults distribution is similar from the tendency found in the study by Durães and Madeira [2006]. We observe that Assignment, Interface and Function faults have roughly the same frequency and Checking faults are the least frequent ones. There was a considerable increase of Function faults when compared to the study by Durães and Madeira [2006]. It is our belief that these differences are due to the specific characteristics of the programming language and paradigm; Java programming code tends to be more modularized and encapsulated, and requires more methods constructions to make the corrections. There was also a considerable increase of Interface faults. As Interface faults are faults that produce errors in the interaction among components, modules, device drivers, call statements or parameter lists, this increase may be due to the stronger importance assigned to the interfaces in Java applications and to the security vulnerabilities corrections, since attacks can be done especially through variables and values inputs.

4.2. Security Java Faults

From 574 investigated faults, 67 are found to lead to software vulnerabilities. Table 4 shows the most frequent faults observed in security vulnerability correcting patches. These faults represent 73.2% of the total security faults observed and follow a pattern similar to that of the general Java faults, where the fault types MFC and MIFS are the most frequent ones. Table 5

presents the security Java faults distribution and a comparison with the distribution of Fonseca and Vieira´s field study [Fonseca and Vieira 2008].

Table 4. Most frequent security Java faults and corresponding ODC fault types

Fault Nature	Specific fault types	Faults	ODC Type
Missing construct	Missing function call (MFC)	14	Algorithm
	Missing If construct plus statements (MIFS)	11	Algorithm
Wrong construct	Wrong variable used in parameter of function call (WPFV)	8	Interface
	Wrong value used in variable initialization (WVIV)	7	Assignment
	Wrong value used in parameter of function call (WPFL)	5	Interface
	Wrong algorithm - code was misplaced(WALR)	4	Algorithm
Total faults		49	

Table 5. Security fault distribution across ODC defect-type distribution and comparison with Fonseca and Vieira´s field study [Fonseca and Vieira 2008].

ODC defect-type	Security Java Faults (SJF)	SJF ODC defect-type distribution (%)	PHP faults distribution (%) [Fonseca and Vieira 2008]
Algorithm	36	53.7	86.01
Assignment	11	16.4	6.04
Checking	5	7.5	2.36
Function	1	1.5	0
Interface	14	20.9	5.6
total	67		

The security fault distribution follows a distribution pattern similar to that of general Java faults, except for Function faults. However, significant differences arise when we compare our security fault distribution with that presented by Fonseca and Vieira [2008]. Again, we believe these differences are due to the programming language specific characteristics (PHP faults were investigated in that study).

Table 6 shows, for each fault type, the security fault percentage relative to general Java faults. Interface faults have the highest percentage: 14 of the 74 Interface faults make the software vulnerable, meaning that 18.9% of the Interface faults are security faults.

Table 6. Security fault percentage relative to general Java faults.

ODC defect-type	GJF	SJF	SJF / GJF (%)
Algorithm	272	36	13.2
Assignment	81	11	13.6
Checking	57	5	8.8
Function	90	1	1.1
Interface	74	14	18.9
Total	574	67	11.7

4.3. New fault types

In our study new fault types were found due to the Java language specific characteristics and the object-oriented paradigm, representing 7.1% of the total Java faults. As they arise due to the Java language specificity, they must be considered in the creation of new operators and, consequently, in the development of a realistic Java injection fault tool. Also, some of these new fault types are security faults. Table 7 presents a new Java fault types summary with the corresponding frequencies and the ODC type. A brief description for each new fault type follows.

Table 7. New Java fault types

Fault nature	Fault specific type	Faults	ODC type
Missing contruct	Missing interface implementation (MII)	6	Interface
	Missing throw statement (MTS)	4	Algorithm
	Missing try/catch/finally statement around statements (MTCFAS)	4	Algorithm
	Missing try/catch/finally statement plus statements (MTCFS)	4	Algorithm
	Missing Synchronized statement around statements (MSAS)	2	Algorithm
	Missing throws specification in method (MTSM)	2	Algorithm
	Missing extended class (MEC)	1	Interface
	Missing Synchronized statement (MSS)	1	Algorithm
Wrong construct	Wrong extended class (WEC)	9	Interface
	Wrong parameter passed to an object constructor (WPOC)	5	Assignment
Extraneous construct	Extraneous try/catch/finally statement (ETCFS)	2	Algorithm
	Extraneous Synchronized statement (ESS)	1	Algorithm
Total faults		41	

Interface faults

Missing interface implementation (MII): the omission of the interface specification that will be implemented by the class or even the *implements* clause.

Missing Extended class (MEC): the omission of the super class specification or even the *extends* clause.

Wrong extended class (WEC): the super class specified during the class declaration is wrong.

Assignment faults

Wrong parameter passed to an object constructor (WPOC): a wrong parameter passed to a constructor method when an object is created by the *new* clause.

Algorithm faults

Missing synchronized statement (MSS): the omission of a code fragment with synchronize instruction.

Missing synchronized statement around statements (MSAS): the omission of a synchronize instruction in the existing code fragments.

Missing throws specification in method (MTSM): the omission of the exceptions specification, which will be thrown by particular methods.

Missing throw statement (MTS): the omission of exception handling.

Missing try/catch/finally statement plus statements (MTCFS): the omission of a code fragment with the corresponding exception handling

Missing try/catch/finally statement around statements (MTCFAS): the omission of exception handling in existing code fragments

Extraneous synchronized statement (ESS): a synchronized instruction used mistakenly.

Extraneous try/catch/finally statement (ETCFS): exception handling used in a wrong way.

Fault types relative to the try/catch/finally statements were classified in a generic way and some characteristics of the try/catch/finally block syntax must be explained. It is known that every try block requires at least one catch block or one finally block. For each try block several catch blocks are permitted. The *missing construct* faults relative to the try/catch/finally block can be the omission of one or more catch blocks corresponding to an existent try block; it can also be the omission of one finally block. The extraneous construct faults can be the wrong use of catch and finally blocks even separately.

To classify these new fault types the nomenclature proposed by Durães and Madeira [2006] was followed. Some structures are related to well known Java specific faults presented by Reilly [2009] (Top Ten Errors), but the majority of the new fault types do not

represent that kind of mistakes. This can be due to the maturity level of the applications, where more stable versions do not present those common errors.

Most representative faults

The most representative new fault types are WEC, MII and WPOC, accounting for, respectively, 22%, 14.6% and 12.2% of the total new fault types. Their structures are exemplified in Table 8, where the missing or wrong code is marked with gray background.

Table 8. Fault structure of the most representative new fault types

MII	…code before public class Class1 implements Interface1, Interface2 … code after
WEC	…code before public class Class1 extends SuperClass1 … code after
WPOC	…code before Class1 object1 = new Class1 (parameter) …code after

In the object-oriented paradigm an interface establishes a kind of contract which is fulfilled by a class that implements this interface. When a class implements an interface, it is ensured that all the functionalities specified by the interface will be offered by the class. Interfaces define only method and constants variables definitions and represent higher level abstraction than the classes do, i.e., it is possible to design explicitly all the interfaces of an application before deciding on the more adequate implementation form. A class can implement several interfaces.

The new fault type MII occurs when the programmer misses the *implements* clause or misses the specification of the implementation of some interface. This fact can lead to missing some functionality necessary to the correct behavior of the application. This functionality will be included when the correction of this fault type is done. It also contributes to the increase of the MFCT fault type in this field study, because some interfaces may have several methods definitions that are called in some part of the code.

The WEC fault type occurs due to the concept of inheritance provided by the object-oriented paradigm. This concept permits characteristics common to various classes to be concentrated in only one class (superclass). Extending a wrong class means inheriting wrong characteristics and wrong functionalities to be used by the subclass.

The WPOC fault type is related to the constructor method. This method determines which actions must be executed when an object is created. In the Java language, the constructor is defined as a method whose name is the class name and that does not have a

return type, neither *void*. The constructor is invoked only at the moment that the object is created by the *new* clause. The constructor can receive arguments and the same constructor can be defined with different numbers of arguments, using the overloading mechanism.

All the classes have at least one predefined constructor. If no one constructor is defined explicitly by the programmer, a default constructor, which does not receive arguments, is included in the class by the Java compiler. Thus, for the WPOC fault type, if the called constructor has an argument, it means that the programmer wants to write a piece of code to be executed when an object is created (he does not want to use the default constructor). Passing a wrong parameter as an argument means that the code to be executed in this constructor can create an object with wrong initialization values, for example.

Three security faults identified among the new fault types were found. They correspond respectively to the MTSM, MTCFAS and MTCFS fault types. These three fault types are related to the exceptions concept. The Java language offers mechanisms to detect and to treat exceptions.

The existence of an exception means that some exceptional condition has occurred during the code execution. Thus, exceptions are associated with faults that were not identified during the compilation. Catching the exceptional situation is necessary to treat them. For each exception that can occur during the code execution, an exception handler must be specified. The Java compiler verifies and enforces that exceptions have an associated treatment block.

The MSTM, MTCFAS, MTCFS fault types occur due to missing treatment blocks and missing statements related to them. It means that the exceptions can propagate and cause errors in the applications.

5. Conclusions

The field data study reported in this paper analyzed 574 Java software faults. Among them, 67 lead to security vulnerabilities. The results show that the fault types found correspond largely to the fault types found in the field study using the C programming language. The fault representativeness also follows this tendency, meaning that the most frequent Java faults correspond to the most frequent C faults; this shows that mistakes made by computer systems programmers follow a similar pattern independently of these programming languages.

Concerning fault distribution, the Java field study shows a considerable increase of Function and Interface fault types; this increase can be due to Java language characteristics and object-oriented paradigm, security vulnerabilities corrections and a stronger importance assigned to the interfaces in the Java applications analyzed.

New fault types were identified according to Java language specific characteristics and the object-oriented paradigm. These new fault types are important for the validation process of the Java applications, mainly because some of them are security faults that must be considered.

As future work the authors' aim is to develop a fault injection tool to automate the software fault injection process in Java applications. This tool will use the fault

representativeness found in the present field study to create faultloads to be used during fault injection experiments, to obtain more realistic and useful results in the software validation process.

References

Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C. (2004) "Basic concepts and taxonomy of dependable and secure computing". IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, p. 11 – 33.

Azureus (Vuze). (2009). Available in www.azureus.sourceforge.net. Last access on February.

Barcelos, P. P., Leite, F., Silva, T. W. (1999) "Implementação de um Injetor de Falhas de Comunicação". SCTF '99 – VIII Simpósio de Computação Tolerante a Falhas. Campinas, Brazil, p. 225-239.

Carreira, J., Madeira, H. and Silva, J. G. (1995) "Xception: Software Fault Injection and Monitoring in Processor Functional Units". 5º IFIP International Working Conference on Dependable Computing for Critical Applications. Urbana-Champaign, EUA, p. 135-149.

Chillarege, R. (1995) "Orthogonal Defect Classification", Chapter 9 of "Handbook of Software Reliability Engineering", Michael R. Lyu Ed., IEEE Computer Society Press, McGraw-Hill.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D., Ray, B. and Wong, M. (1992) "Orthogonal Defect Classification – A Concept for In-Process Measurement". IEEE Transactions on Software Engineering, vol. 18, n. 11, p. 943-956.

Durães, J. and Madeira, H. (2006) "Emulation of Software Faults: A Field Data Study and Practical Approach". IEEE Trans. on Software Engineering, vol. 32, n. 11, p. 849-867.

Fonseca, J. and Vieira, M. (2008) "Mapping software faults with web security vulnerability". IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), Anchorage, USA, p. 257-266

FreeMind. (2009). Available in www.freemind.sourceforge.net. Last access on February.

Gage, D. and McCormick, J. (2004) "Why Software Quality Matters", Baseline Magazine, p. 32-59.

JEdit. (2009). Available in www.jedit.org. Last access on February.

Kalyanakrishnam, M., Kalbarczyk, Z. and Iyer, R. (1999) "Failure Data Analysis of a LAN of Windows NT Based Computers", Symposium on Reliable Distributed Database Systems, SRDS-18, Switzerland, p. 178-187.

Lee, I. and Iyer, R. K. (1995) "Software Dependability in the Tandem GUARDIAN System", IEEE Trans. on Software Engineering, vol. 21, no. 5, p. 455-467

Lyu, M. (1996) “Handbook of Software Reliability Engineering”, IEEE Computer Society Press, MCGraw-Hill.

Martins, E., Rubira, C. and Leme, N. (2002) “Jaca: A reflective fault injection tool based on patterns”. Proc of the 2002 International Conference on Dependable Systems & Networks, Washington D.C. USA, p. 483-487.

Musa, J. (1996) “Software Reliability Engineering”, McGraw-Hill.

Neves, N., Antunes, J., Correia, M., Veríssimo, P., Neves, R. “Using Attack Injection to Discover New Vulnerabilities”. Proc. of the International Conference on Dependable Systems and Networks (DSN), 2006, p. 457-466.

Pfleeger, S. (2000) "Risky Business: what we have yet to learn about risk management", The Journal of Systems and Software, 53, p. 265-273.

Phex. (2009). Available in www.phex.org. Last access on February.

Reilly, D. (2009). Top Ten Errors Java Programmers Make. Available in WWW.javacoffebreak.com/articles/toptenerrors.html. Last access on May.

Struts.(2009). Available in www.struts.apache.org. Last access on February.

Sullivan, M. and Chillarege, R. (1992) “Comparison of Software Defects in DataDatabase Management Systems and Operating Systems”, Proc. of the 22nd IEEE Fault Tolerant Computing Symposium, FTCS-22, p. 475-484.

Tomcat. (2009). Available in www.tomcat.apache.org. Last access on February.

WinMerge. (2009). Avaliable in www.winmerge.org. Last access on February.

Apêndice B

Publicação

Basso, T.; Fernandes, P.C.S.; Jino, M.; Moraes, R. "***Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool***". *4th Workshop on Recent Advances in Intrusion-Tolerant Systems* (WRAITS 2010), realizado em conjunto com o *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2010).

Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool

Tânia Basso Plínio César Simões Fernandes Mario Jino Regina Moraes
State University of Campinas, UNICAMP, Brazil
{taniabasso, pliniocsfernandes}@gmail.com {jino@dca.fee, regina@ft}.unicamp.br

Abstract

Most software systems developed nowadays are highly complex and subject to strict time constraints, and are often deployed with critical software faults. In many cases, software faults are responsible for security vulnerabilities which are exploited by hackers. Automatic web vulnerability scanners can help to locate these vulnerabilities. Trustworthiness of the results that these tools provide is important; hence, relevance of the results must be assessed. We analyze the effect on security vulnerabilities of Java software faults injected on source code of Web applications. We assess how these faults affect the behavior of the scanner vulnerability tool, to validate the results of its application. Software fault injection techniques and attack trees models were used to support the experiments. The injected software faults influenced the application behavior and, consequently, the behavior of the scanner tool. High percentage of uncovered vulnerabilities as well as false positives points out the limitations of the tool.

1. Introduction

Web applications are extremely popular nowadays. From single individuals up to large organizations, there is an increasing dependency on this technology. Information and data are stored, traded and made available on the Web. This type of application is becoming increasingly exposed as any security vulnerability can be exploited by hackers.

Automatic vulnerability scanner tools are often used by developers and system administrators to test Web applications against security vulnerabilities. Reliable results from vulnerability scanners are essential and the analysis of the scanners' effectiveness is important to guide the selection as well as the use of these tools. Effectiveness may be assessed by two main aspects: vulnerability coverage and false positive rate. The vulnerability coverage is associated to the reliability of the tool; high reliability means that the tool is able to detect correctly all security vulnerabilities in the application. (it is doubtful whether undetected vulnerabilities do not really exist in the application or the scanner was not able to detect it). It is important to minimize the rate of false positives because when a non-existent vulnerability is reported, the development team may spend a lot of time trying to correct it before realizing that the false vulnerability does not really exist.

Previous research [31][29] shows that, in general, Web vulnerability scanners present a high number of false-positives and low coverage, highlighting the limitations of this kind of tool. Although other potential causes for vulnerability do exist, the root cause of most security attacks are vulnerabilities created by software faults [3][4].

Our goal is to investigate the effect that injected Java software faults may have on security vulnerabilities. The proposal is to understand, through the analysis of the context of the source code of the applications where the faults were injected, how these faults affects the behavior of the applications with respect to security vulnerabilities. This is important to speed up the detection of security vulnerabilities, allowing that countermeasures are applied to eliminate them or to reduce the severity of their exploitation, contributing to higher levels of dependability for the application under test. Then, we want to analyze how it affects the behavior of the scanner vulnerability tool. In order to validate the scanner results it is necessary to assess its effectiveness. Based on this knowledge, we intend to extend the experiments to other scanner tools and investigate how to scale the results to more complex applications in an automatic way. Then we want to propose a methodology to analyze vulnerability scanners effectiveness based on fault injection and attack injection techniques.

The paper describes a method based on attack trees modeling to perform security tests. The approach consists of injecting software faults into

small Java applications. They have to be small because the context of the source code should be analyzed to get accurate measures of the detection coverage and false positives rate, the reason why we want to have the experiments under control. Once the faults are injected, the scan is run to check if it can detect potential vulnerabilities caused by the injected fault. Creation of vulnerabilities is confirmed through manual attacks, guided by the attack trees.

Unlike other studies, where lots of faults are injected only to validate the scanner tool results, we want to investigate, through application's source code construction and attack models, the relationship between the fault injected and the potential security vulnerabilities created. The method described in this paper will eventually lead to the development of an attack injection tool for Java applications. Later, we intend to reproduce these experiments, automatically, on larger and more complex applications, applying the knowledge acquired through these controlled experiments.

The structure of this paper is as follows: Section 2 presents the background on software faults; Section 3 describes the related work on analysis of scanner tools effectiveness; Section 4 describes the attack trees modeling approach; Section 5 shows the steps and the methodology applied to the experimental study; Section 6 presents the results and discussions on the study; and Section 7 presents our conclusions and future work.

2. Software fault injection

Few works address the relationship between software faults and security vulnerabilities. A study by Fonseca and Vieira [3] analyzed security patches of web applications developed in PHP. The types of faults that are most likely to lead to security vulnerabilities are characterized.

The work by Basso *et al* [4] presents a field data study on real Java software faults, including security faults. The field study was based on security correction patches analysis available in open source repositories. More than 550 faults were analyzed and classified, determining the representativeness of these faults. The authors also define new operators, specific to this programming language structure, guiding the definition of a Java faultload. The software fault injection technique used in this paper is the G-SWFIT 0. This technique focuses on the emulation of just the most frequent types of faults. It is based on a set of fault injection operators that reproduce directly in the target executable code the instruction sequences that represent the most common types of high-level software faults.

To inject the faults, a use case of the application was selected, including all classes in the source code that implements this use case. Then, the locations in this piece of the target code where the injection is performed are selected by the G-SWFIT to inject representative software faults. Each fault was injected in all possible locations of this specific use case, one at time, forming different scenarios to be analyzed.

3. Vulnerability scanner tools effectiveness

Web vulnerability scanners are regarded as an easy way to test applications against vulnerabilities. Most of these scanners are commercial tools (e.g., Acunetix [6], IBM Rational AppScan [7], N-Stalker [8] and HP WebInspect [9]); there are also free ones (e.g., Burp Suite [10] and Gamja [11]), but with limited use, not fully automatic as their commercial equivalent.

Vieira *et al* [31] present an experimental evaluation of security vulnerabilities in publicly available web services. Four well known vulnerability scanners have been used to identify security flaws in web services implementations. A large amount of differences in vulnerabilities detected and a high number of false-positives and low coverage were observed.

Fonseca *et al* [29] propose a method to evaluate and benchmark automatic Web vulnerability scanners using software fault injection techniques. Three leading commercial scanning tools were evaluated and the results also have shown that in general the coverage is low and the percentage of false positives is very high.

However, these studies were focused on a specific family of applications: web services and PHP applications, respectively. Thus, the results obtained cannot be easily generalized and our intention is to complement the results obtained previously, providing Java applications results, aiming to increase the amount of different programming languages applications to obtain the sufficient requirements to generalize the results. Furthermore, their previous study does not present a clear methodology to validate the vulnerabilities detected by scanner tools.

We investigate the behavior of scanner tool in the presence of injected Java faults, show a method using attack trees to model the possible ways to

perform attacks to specific vulnerabilities, and validate the results obtained by the scanner. This is addressed in the next sections.

4. Attack trees and security vulnerabilities

Attack trees provide a structural way of describing the security of systems, based on several attacks types [12]. In attack trees, the root node represents the achievement of the ultimate goal of the attack. Each child node represents sub-goals that have to be accomplished for the parent goal to succeed. Parent nodes can have their child nodes related by an OR or an AND relationship. In an OR relationship, if any of the sub-goals are accomplished then the parent node is successful. With an AND relationship, all of the sub-goals must be accomplished for the parent node to be successful. Individual intrusion scenarios are generated by traversing the tree in a depth-first manner. The objective is to cover all actions represented in the leaves.

In our work the attack trees were used to describe the possibilities of attacking a specific type of security vulnerability. We consider three types of security vulnerabilities: SQL Injection , Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). They were selected because of their criticality, occupying the first, second and fifth place in the OWASP Top 10 [13]. These vulnerabilities are widely spread and dangerous vulnerabilities, and may cause major damage to the victims.

XSS occurs when a web application gathers malicious data from a user. The data is usually gathered in the form of a hyperlink which contains malicious content within it. The user will most likely click on this link from another website, instant message, or simply just reading a web board or e-mail message. After the data is collected by the Web application, it creates an output page for the user, containing the malicious data that was originally sent to it, but in a manner to make it appear as valid content from the website [14]. The malicious code can also be permanently stored on the target servers, such as in a database or it can be generated dynamically through the Document Object Model of the browser. SQL injection refers to a class of code-injection attacks in which data provided by the user is included in an SQL query in such a way that part of the user's input is treated as SQL code. By leveraging these vulnerabilities, an attacker can submit SQL commands directly to the database [15].

CSRF works by exploiting the trust a site has for the user. Site tasks are usually linked to specific URLs (e.g: http://site/stocks?buy=100&stock=ebay) allowing specific actions to be performed when requested. If a user is logged into the site and an attacker tricks their browser into making a request to one of these task URLs, then the task is performed and logged as the logged in user [16].

For each of these three types of vulnerability an attack tree was created. Figure 1 presents the attack tree for validation of CSRF vulnerabilities. This tree was chosen because it is the vulnerability that appeared more frequently in the results. Due to space restrictions, the other trees are not presented, but they can be seen elsewhere [17]. The "OR" labels are omitted to improve the tree simplicity.

For the CSRF tree we covered the part of the CSRF attack relative to the acceptance of the requests coming from another source. The part relative to the means used to lure the user to activate the request is not covered as they are out of the defensive bounds that an application can have against CSRF.

In Figure 1, the first step to perform a CSRF attack is to have the user logged in the site because the attack will use its trust in the user authentication. If this step is not fulfilled the attack could not be realized. The next step is to analyze the request from the site that the attack will target in order to be able to reproduce it. If the site does not have CSRF countermeasures this step will lead to the next one because the request will be considered valid and will take effect on the site.

If the site uses any defensive measure it will be necessary to analyze the request and take additional actions. A known defensive method consists in appending different tokens to each request, but this approach can be bypassed if the application is vulnerable to XSS attacks. This is possible because XSS attacks permit to get valid session tokens from the application. The last path (the three remaining leaf nodes) of the tree shows how to overcome applications that use verification of the HTTP (Hypertext Transfer Protocol) Referrer attribute, although this is not a recommended defensive measure.

5. The experimental study

Two open source Web applications developed in Java were selected to carry out the experiment. The first one, which we call App1, is a Customer Relationship Manager (CRM) and Project and Management Tool. It uses the MySQL database

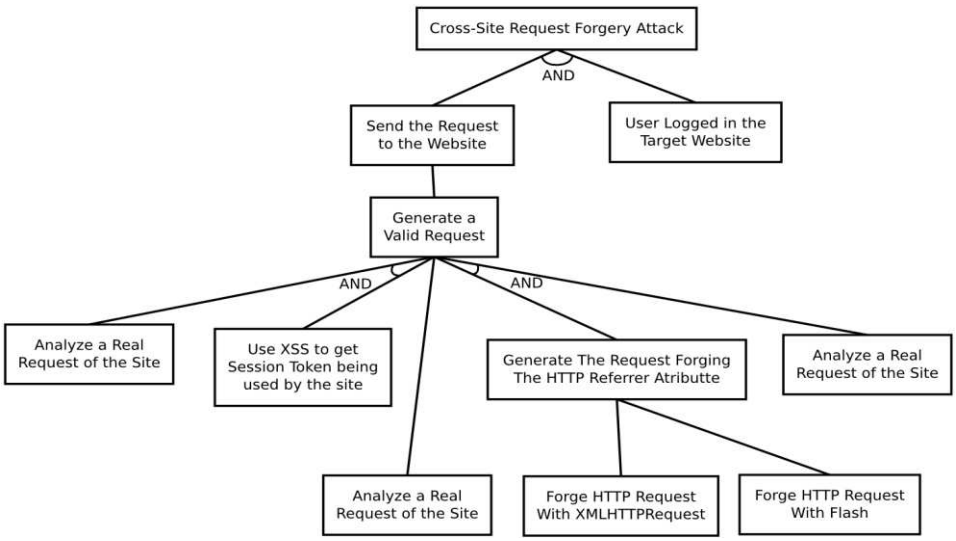


Figure 1. CSRF attack tree

and technologies such as Hibernate, Struts framework, and Jasper Reports. The second Web application, App2, is a management system for Distance Education, developed by the Brazilian federal government. It uses the Postgres database and technologies such as Hibernate and Ajax. We have chosen similar use cases from both applications to be the target piece of code of injected faults.

The types of fault to be injected were selected from the faultload of Basso *et al* [4]. We selected the two most frequent types of faults observed: the Missing Function Call (MFC) and the Missing If construct plus the Statement (MIFS) for this first set of experiments. They represent the most common types of fault that are responsible for security vulnerabilities. The MFC is the fault type which relates to function or method call that was missing from implementation. The MIFS is a fault type that represents the omission of a block of code made of an "if" construct and its associated statements which are executed only if the "if" condition is true 0

The security vulnerability scanner was selected because of its great market insertion and availability. We do not mention its brand because commercial licenses do not allow in general the publication of tool evaluation results. Basically, the operation of the scanner consists of two stages: explore and test. During the explore stage, requests are send to the application and the responses are analyzed, looking for indication of potential vulnerabilities. In the test stage, the tool sends thousands of custom tests to identify security problems (based on the results of the first stage) and rank their level of security risk.

The three security vulnerabilities considered for this study are discussed in Section 4

5.1. Injecting faults, executing the scans and validating the results

The tests start with a "Gold Run", where the application is tested once by the scanner tool without any fault injected. The web application may already have some vulnerabilities and this run should be able to find most of them. The results of "Gold Run" execution are collected to compare other results when the faults are injected.

After the "Gold Run", one fault is injected. The context of the code where the fault is injected is analyzed manually to understand the effect of this fault in the applications behavior. If necessary, data are inserted, removed or changed in the database of the application under test to guarantee the activation of the fault. Next, the code and database are versioned, defining a scenario to be tested.

The scanner application is run and verification for new vulnerabilities is made, i.e., we identify new vulnerabilities when comparing the vulnerabilities detected in the original application (without any fault injected) . In some cases, one fault injected can be responsible for many security vulnerabilities. If new vulnerabilities are detected, attacks are performed in the current scenario using the attack trees. To exploit the new vulnerabilities all possibilities detected through the attack tree are experimented. This aims to verify if the new vulnerability actually exists or if it is a false positive. Then, the same attacks are performed in the original application scenario (without any fault injected) in order to verify if the vulnerability existed before the fault injection and was not identified by the tool (lack of coverage).

The procedure is done for each possible location in the source code where faults can be injected in accordance with G-SWFIT technique (for the selected use case).

6. Results and discussions

For both Web applications, we analyzed, respectively, 11 and 23 different scenarios. Table 1 shows the total of scenarios that presented new security vulnerabilities detected by the scanner due to the fault injection.

Table 1. Applications scenarios and vulnerabilities

	App1	App2
Total scenarios analyzed	11	23
Scenarios with new vulnerabilities	5	7
% of faults that affected the scan	46%	30%

According to Table 1, about 40% of the injected software faults affected the scanner results. A detailed analysis of the context of application's source code where the faults were injected is important to assess the effect of the fault in the application behavior. The context analysis in conjunction with the structure of the attacks permits to assess the influence of the injected fault on potential new security vulnerabilities detected by the scanner tool. Consequently, this procedure permits to assess correctly the effectiveness of the scanner tool, through the identification of lack of coverage and false positives.

We noticed that the injected faults were not in the same locations the new vulnerabilities arose. The injected faults affected the applications behavior and, consequently, the scanner tool behavior, due to the context of the application and the procedures necessary to activate the fault. For example, many faults were injected in locations where a null entry point is verified in the source code. Activating this fault, the application modifies its behavior by not verifying the null entry point and forcing the application to display error pages. Also, according to the attack structure in Figure 1, the verification of null entry points is not explored, i.e., it doesn't create a security vulnerability.

Even though the injected software faults are unrelated to the location of new vulnerabilities, they affected the results of the scanner tool. Table 2 shows the lack of coverage and the number of false positives obtained in the experiments. The lack of coverage is about vulnerabilities that do exist in the web applications, confirmed through successful manual attacks. The number of false positives is related to vulnerabilities indicated by the tool that were not confirmed by the manual attacks. Table 3 shows the percentage of lack of coverage and false positives according to each type of security vulnerability, where the last column resume the percentage of the total of experiments including all types of vulnerabilities.

Table 2. Applications lack of coverage and false positives

	App1	App2	Total
Vulnerabilities not detected (lack of coverage)	8	1	9
False positives	5	3	8

Table 3. Percentage of security vulnerabilities: lack of coverage and false positives

	XSS	SQL inject	CSRF	Total
Vulnerabilities	2	2	15	19
Lack of coverage (%)	0%	0%	60%	47%
False positive (%)	50%	100%	34%	43%

Based on Table 2 the App1 presented worse lack of coverage and more false positives than App2. By the analysis of the context of the source code, we believe that the App2 code is more modularized, with less coupling with other modules and fewer use cases. It is also smaller, i.e., has fewer lines of code (LOC). Thus, it is easier to activate the injected faults and easier to control the application's behavior. Similarly, it is easier for the tool to analyze the application and detect the vulnerabilities in a correct way.

All 9 undetected vulnerabilities are about CSRF and are part of the vulnerabilities presented in Table 3. They represent 60% of the lack of coverage. These lacks of coverage were identified in the original applications (without any fault injected) and in the applications with faults injected. In most of the cases, when scanning the application with faults injected, a new vulnerability detected by the tool was one already present in the original application, not identified in the "Gold Run".

Also in Table 3, the false positives come from the three types of security vulnerabilities: XSS, SQL injection and CSRF, representing, respectively, 50%, 100% and 34% of the vulnerabilities detected. The false positive associated to the XSS vulnerabilities is

considered because the scanner tool integrates outdated version of internet browsers. An attack successfully executed by the tool, when executed in the later versions of internet browsers, has no effect, because these versions implement features that do not permit the execution of common XSS attacks.

The SQL injection false positives were identified through the attacks and the analysis of the source code. Both applications use the Hibernate technology, which is an object/relational persistence and query service [18]. It permits to encapsulate the queries and send objects to the database through predefined classes and methods, discarding the necessity of explicit SQL queries constructions. According to forums and some information available in technical websites [88][90], in code constructed with Hibernate it is more difficult – but not impossible – to have vulnerability to SQL injection attacks. However, the way that the application was coded, i.e., extremely encapsulated, do not open opportunities to develop successful attacks. Even the scanner tool provides no assurance about its detection result, and it informs that this detected vulnerability requires user verification.

Most of cases where CSRF false positives were identified, they happened in error pages. An attacker performing a CSRF attack to access an error page can be dangerous if the error page presents links or buttons which permit access to the application (as "back" buttons which bring back the user to the last page he/she accessed) or if the error page displays private information about the system (such as database name or table names). For both applications, the error pages do not present any way of accessing application functionalities or private information. Hence, we considered theses cases as false positives because a CSRF attack when accessing the error pages is useless.

The last column of Table 3 shows the total percentage of lack of coverage and false positives. From the 19 vulnerabilities investigated, 42% are false positives and 47% were not identified by the scanner tool. It indicates the limitations of this tool we found in this study.

7.Conclusions

In this paper we present an experimental study where we analyzed the effect that Java software faults, injected on the source code of Web applications, can have on security vulnerabilities. Also, we analyzed the influence of these faults on the security vulnerabilities detection by a well known commercial web security vulnerability scanner tool. These analyses were performed based on a method that uses attack trees modeling in order to verify the results obtained by the scanner tool.

Fault injection techniques were used to support the experiments and software faults were injected, one at time, in a controlled way, into target Java codes of two small Web applications.

The context of the application code where the fault was injected is analyzed in order to understand the relationship between the fault and potential new vulnerabilities. Manual attacks were performed guided by attack tree models to confirm the existence of vulnerability.

Results show that, according to the context of both Java target code applications and considered security vulnerabilities structure, the location of the injected faults were not where new vulnerabilities arose. However, the injected faults did affect the behavior of the application and, consequently, the behavior of the scanner tool in detecting new vulnerabilities..

Results of the scanner tool were validated through manual attacks based on attack trees. It showed high percentage of lack of coverage and many false positives, showing its limitations. Some factors that influenced this percentage are, in addition to the activation of the faults injected into the source code of the applications, the use of different development technologies (such as Hibernate) and some outdated features of the tool (as the internal internet browser).

As future work we intend to extend this experiment analyzing the effect of other types of faults and the effectiveness of other vulnerability scanner tools. We also intend to develop a tool to perform the attacks (based on attack trees) automatically.

References

[1] M. Vieira, N. Antunes, H. Madeira. "Using Web Security Scanners to Detect Vulnerabilities in Web Services". *IEEE/IFIP Intl Conf. on Dependable Systems and Networks, DSN 2009*, Lisboa, Portugal, June 2009.

[2] J. Fonseca, M. Vieira, H. Madeira. "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks", *13º IEEE Pacific Rim Dependable Computing Conference (PRDC 2007),* Melbourne, Victoria, Australia, December 2007.

[3] J. Fonseca, M. Vieira. "Mapping software faults with web security vulnerability". *IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2008),* Anchorage, USA, 2008.

[4] T. Basso, R. Moraes, B. P. Sanches, M. Jino. "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults." *In: Workshop*

de Testes e Tolerância a Falhas - WTF2009, João Pessoa, Brazil, 2009, pp. 1-13.
[5] J. Durães, H. Madeira. "Emulation of Software Faults: A Field Data Study and Practical Approach". *IEEE Trans. on Software Engineering*, vol. 32, n. 11, Nov. 2006, pp.849-867.
[6] Acunetix Web Appication Security. Available in http://www.acunetix.com, November/2009.
[7] IBM Rational AppScan. Available in http://www-01.ibm.com/software/awdtools/appscan/, November/2009.
[8] N-Stalker The Web Security Specialists. Available in http://www.nstalker.com/, November/2009.
[9] HP WebInspect. Available in https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__, November/2009.
[10] Burp Suite. Available in http://www.portswigger.net/suite/, November/2009.
[11] Gamja. Available in http://sourceforge.net/projects/gamja/, November/2009.
[12] B. Schneir. "Attack Trees: Modeling Security Threats", Dr. Dobb's Journal, December, 1999.
[13] OWASP Top 10 Project. Available in http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, February/2010.
[14] CGISecurity.com. "The Cross Site Scripting FAQ." Available in http://www.cgisecurity.com/xss-faq.html, November/2009.
[15] W. G. Halfond, J. Viegas, A. Orso, "A classification of SQL injection attacks and countermeasures". In Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, Virginia, March/2006.
[16] R. Auger. "The Cross-Site Request Forgery (CSRF/XSRF) FAQ". Available in http://www.cgisecurity.com/csrf-faq.html, November/2009.
[17] Research Test Group. Available in http://www.ceset.unicamp.br/docentes/regina/projeto/, December/2009.
[18] Hibernate. Available in https://www.hibernate.org/, December/2009.
[19] OWASP – Preventing SQL injection in Java. Available inhttp://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java#Hibernate, December/2009.
[20] CWE – Common Weakness Enumeration. Available in http://cwe.mitre.org/data/definitions/564.html, December/2009.

Apêndice C

Artigo Submetido para Publicação

Fernandes, P.C.S.; Basso, T.; Moraes, R.; Jino, M.; "***Attack Trees Modeling For Security Tests in Web Applications***". *Artigo submetido para The 12th IEEE International High Assurance Systems Engineering Symposium - HASE 2010.*

Attack Trees Modeling for Security Tests in Web Applications

Plínio Cesar Simões Fernandes Tania Basso Regina Moraes Mario Jino
State University of Campinas, UNICAMP, Brazil
{pliniocsfernandes, taniabasso}@gmail.com {regina@ft, jino@dca.fee}.unicamp.br

Abstract— **Security vulnerabilities in Web applications, when exploited by attackers, can cause large damages. Testing this kind of applications against security vulnerabilities is an important activity and, in this case, it is necessary for the tester to have the vision of an attacker. To provide this attacker vision, this paper brings the attack trees modeling to explore three types of security vulnerabilities that are, nowadays, considered critical. The attack trees are also useful to help researchers that want to develop defenses mechanisms against these kinds of attacks. A case study was performed, in which real applications are tested against the security vulnerabilities based on test cases derived from the attack trees. Results show that modeling using attack trees is very useful to security tests and can provide, many times, more effective results than some automatic Web vulnerabilities scanner tools can.**

Security Vulnerability, Attack Tree Modeling, Security Test (key words)

I. INTRODUCTION

Software security vulnerabilities have been largely exploited by attackers to steal confidential information and invade corporate networks. Currently, the number of cases of exploited security vulnerabilities in Web applications has increased. This fact occurs because most Web applications are developed with software faults due to factors as high complexity and strict time and cost constraints. This software faults can lead to security vulnerabilities. Also, usually, a secure software development cycle is not adopted.

Even though the software developers are encouraged to follow best coding practices – and it includes security aspects – most times, due to time restrictions, the developers are focused on developing the main functionalities and satisfying the client requirements, neglecting security aspects. It is known that traditional network security mechanisms as firewalls, cryptography and intrusion detection systems can protect the network but not mitigate Web application attacks. Then, the attackers are changing their focus from the network to the Web applications, where the insecure codification represents major risks [3]. Therefore, the security validation of Web applications that are in operational phase is an important activity.

The goal of this paper is to present most common security vulnerabilities that affect Web applications and show, through attack trees modeling, how these vulnerabilities can be exploited by malicious users. This modeling is important to assist security tests in Web applications and researches to develop defenses countermeasures against the vulnerabilities exploitations. It is known that security tests are fundamentally different from functional tests. Functional tests are described in test cases that define a roadmap of the steps to be followed and the expected result of a behavior not defective. In security tests, the tester follows a line of reasoning different from the traditional one, by thinking like the malicious user, who tries to find not forecasted flows that can compromise the application [2]. Thus, the attack tree modeling is useful because it permits to define the steps of a attacker to exploit a particular security vulnerability and to guide security tests to find these vulnerabilities. If it is possible to understand the steps by which applications are attacked, it is possible to develop countermeasures to thwart the attacks or to reduce the severity of their consequences.

Three different types of security vulnerabilities were investigated in this work: SQL Injection [3], Cross-Site Scripting (XSS) [4], and Cross-Site Request Forgery (CSRF) [5]. This selection is due to the high criticality of these vulnerabilities. They occupy, respectively, the first, second and fifth places of the OWASP top 10 rank [6].

Figure 1(a) shows a statistical summary from *National Vulnerability Database* (NVD) web site [7], which belonged to the *National Institute of Standards and Technology* (NIST) [8]. In 2009, from more than 5700 vulnerabilities reported due to the exploitation of software faults, 14,37% were about XSS type, 16,54% were about SQL injection and 1,95% were about CSRF. Figure 1(b) shows that, in 2010, until march, this percentage is very similar, where, from more than 700 reported vulnerabilities, XSS type represents 13,38%, SQL Injection represents 15,21% and CSRF represents 1,41%. Although the CSRF vulnerability type has a lower number of reported cases, testing Web applications against this type of vulnerability is important because the attacks are difficult to detect and the remediation usually is possible only after the

incidents. Moreover, this is an area in which limited research has been done [9].

Erro! Não é possível criar objetos a partir de códigos de campo de edição. **(a)**	Erro! Não é possível criar objetos a partir de códigos de campo de edição. **(b)**

Figure 1 (a). Security vulnerabilities reported in 2009. (b). Security vulnerabilities reported until march, 2010.

The numbers presented in Figures 1(a) and 1(b) reinforces the importance to test web applications, especially in operational phase, to identify security vulnerabilities. Previous studies [10][11][12] show that even vulnerability scanners (i.e., automatic tools that analyze web applications to find security vulnerabilities) present high rate of lack of coverage, i.e., it cannot detect vulnerabilities that do exist in the applications.

The structure of this paper is as follow: Section 2 presents the related works that were the basis of this study. Section 3 describes the security vulnerabilities and the modeled attack trees. Section 4 presents a case study about testing real Web applications using test cases derived from the attack trees. Section 5 shows the results of the study case and section 6 presents our conclusions and future works.

II. RELATED WORK

Security tests based on models to identify security vulnerabilities are still a challenge. To perform this kind of test it is important that the model represents real attacks and permits to detect the maximum of vulnerabilities from the attack scenarios. Attack tree [13] is a structure that can describe the possible attacks to a system in an organized way, to facilitate the security analysis. The attack tree represents the steps to realize an attack, its goals and interdependencies, and also can be used to represent or calculate probabilities, risks, costs or other weights. They use Boolean logic (AND/OR) to describe relationships between the steps to an attack.

The attack tree models have been used to analyze possible threats to systems, networks or protocol security implementations, and other applications. However, few works refers to attack trees modeling in order to attack security vulnerabilities in Web applications. Byres *et al* [14], for example, describe the use of attack trees to assess the security of a communication protocol called SCADA (Supervisory Controls and Data Acquisition), used to message exchange and control in industrial networks. The authors identified eleven possible attackers goal and then identified security vulnerabilities inherent in the specification and debugging of this protocol.

The work of Martins *et al* [15] aims to detect security vulnerabilities in a wireless network protocol called WAP (Wireless Application Protocol). For this purpose, the authors proposed a method that uses attack trees to represent known attacks in the protocol under test. Thus, test cases are derived from the attack trees and can be mapped to faults to be supported by fault injection tools. As the attacks were previously modeled, the variation of parameters is minimal and it drastically reduces the faultload, increasing the efficiency of the methodology. Experiments were realized and security vulnerabilities were identified. The exploitation of these vulnerabilities through attacks has violated integrity and availability of the WAP protocol.

Fung *et al* [16] uses attack trees to analyze the systems survivability based on service-oriented architectures (SOA). Attacks to components that can be damaged were identified and modeled as the attack trees. Intrusions scenarios were derived from this modeling, which has permitted to the authors to suggest a quantitative measure and provide suggestions to improve the survivability of the analyzed systems.

Edge *et al* [17], in order to evaluate the security of an online banking system, has used attack trees to identify how attackers damage banks accounts and develop countermeasures to protect them. The countermeasures are also represented in tree structure, called protection trees. However, the attacks represented in the attack trees do not refer to specific vulnerabilities of web applications as in the present work. The approach of Edge *et al* [17] refers to the ways of stealing passwords, for example: stealing password through telephone conversation, through network attacks, (main- in- the middle attack), through viruses' contamination (trojans), brute force attacks, and other.

The work of Lin *et al* [9] presents an attack tree modeling to exploit Cross Site Request Forgery (CSRF) vulnerabilities. Their goal is to represent the attacker vision to assist researchers to develop ways to stand up for to this kind of attack. The authors has modeled the trees from a real attack of a popular application, but they did not perform a case study to assess the tree application concerning to the attacks performing and security vulnerabilities identified. In addition, the work is restricted to modeling attacks to only one type of vulnerability.

In our paper, three types of security vulnerabilities were modeled through the trees to represent attacks. A case study using real Web applications was developed to assess the use of the trees. The attacks and respective attack tree modeling are described in the next section.

III. SECURITY VULNERABILITIES AND ATTACK TREES MODELING

This section brings a brief explanation of the three types of security vulnerabilities addressed in this paper. Also, it brings the model of the attack trees to exploit the vulnerabilities. In the attack trees the root node represents the achievement of the ultimate goal of the attack. Each child node represents sub-goals that have to be accomplished in order to reach the parent's goal to succeed. Parent nodes can establish relationships with their child nodes using an "OR" or an "AND" relationship. In an "OR" relationship, if any of the child nodes sub-goals are accomplished then the parent node is successful. With an "AND" relationship, all of the child node sub-goals must be accomplished in order to the parent node to be successful. The leaves of the tree (i.e., nodes that are no longer decomposed) represent attacker's actions. Individual intrusion scenarios are generated by traversing the tree in a depth-first manner. The goal is to cover all actions represented in the leaves.

In this paper, the attack trees were built considering the knowledge about the security vulnerabilities and the attacker capabilities. This is necessary to determine whether an attack is feasible or not, verifying the possibilities of he or she can exploit the web applications over the Internet executing scripts, manipulating input queries or tricking the victim's browser. Then, knowledge about well succeeded attacks is necessary to define attacks to be performed. The attack goals are identified and the attack tree can be constructed. The scenarios are selected so as to cover the leaves to achieve the attack goal, i.e., the root node. To improve simplicity, the "OR" labels are omitted in the models presented.

A. SQL Injection

Exploiting SQL injection vulnerabilities consists of confusing the SQL interpreter so that it executes manipulated commands. This is done from the user data input, which are send as part of a request. Thus, an attacker may gain access to the applications database and create, read, update or delete, in an arbitrary way, any available data to the application. Web applications become vulnerable to this type of attacks when user data input provided to an interpreter are not validated or recoded.

Figure 2 shows the attack tree modeled to represent the steps to exploit SQL injection vulnerabilities. The steps 1 to 4 show the different ways of injecting malicious code (i.e., different inputs) [3]. The step represented by the node 6 consists of finding website data submissions mechanisms to identify an appropriate way to inject information in the web application, identifying the inputs that transmit information to the applications' database. After identifying possible input data to inject, it is necessary to create the SQL code snippet that will be used in the attack (represented by the node 5). The SQL code is constructed according to the intention of the attack and should consider some techniques to avoid mechanisms like signature evasion, if necessary.

If it is noticed that the application performs some type of validation or filtering data, is necessary to modify the SQL code so that it is accepted by the application and, in this way, reaches the database. After creating the code to be injected, it can be tested in different input data (node 7). Sometimes, validations are done in the applications client-side, but not in the applications Server-side. In this case the attacker should bypass this validation (node 8). Nodes 9 and 10 represent the steps to verify the response of the attack and, consequently, if it was successful.

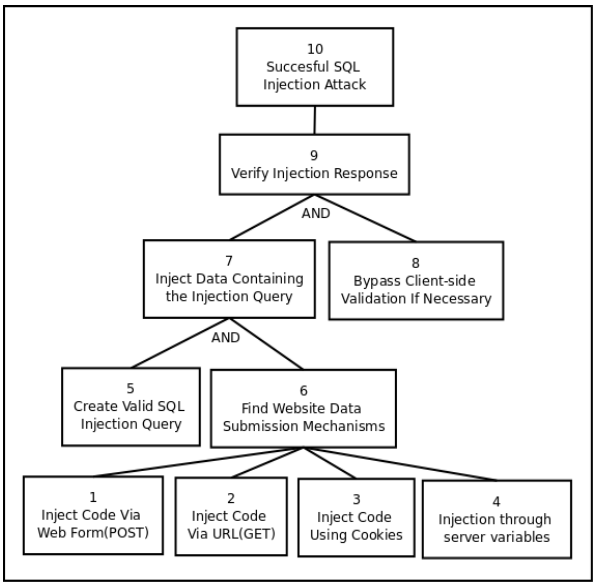


Figure 2. SQL injection attack tree

B. Cross Site Scripting (XSS)

The attack to Cross Site Scripting (XSS) vulnerabilities consists of executing scripts through the victims' browser, allowing session hijacking, insertion of hostile content, theft of personal information, and other damages. This kind of attack can be successfully performed when applications receive user data and send these data to the browser without validating or recoding their content in an appropriate manner.

The XSS attacks can be classified in three types [6]: stored, reflected and DOM-based XSS attacks. In the stored XSS attack the script data are injected into the application and stored by it. So, when the application is accessed later, the script is executed. In the reflected XSS attack, the injected script is immediately shown in the application, i.e.,

immediately executed. In the DOM-based XSS attack, the script modifies the Document Object Model of the victims' browser and is executed dynamically, even when the application was not modified.

Figure 3 shows the attack tree to XSS attacks. The first step to perform a XSS attack, is to find the XSS holes (node 1), which are indications that the application is vulnerable. A XSS hole can be described as a part of the application where it is possible to inject information to be displayed by the application (for example, a user data input that, later, is displayed on a table).

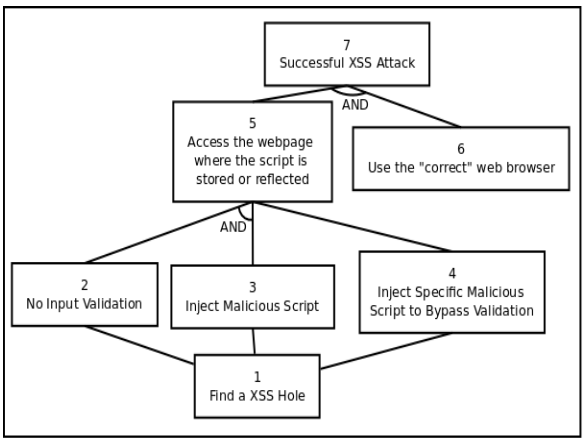


Figure 3. XSS attack tree

The next step is to test the scripts injection into the application. To do this, it is necessary to verify if the application validates special characters in the inserted data. If there is no input validation (node 2), the malicious script can be injected (node 3).

A basic way to test if the application is vulnerable following the steps 1, 2 and 3, is to inject, into each form input data field and variables passed to the URL, a simple script like *<script>alert('XSS')</script>, &{alert('XSS')};* or *<img csstest=javascript:alert('XSS')>*[18].

If the application validates the inserted data, is still possible to perform the attack with a specific script for filter evasion [18] and access the target application using a browser compatible with the script syntax. This step is represented in the node 4.

The final steps to perform the XSS attack consists of accessing the part of the application where the malicious script is executed (node 5), because this part may not be the next one to be accessed after injecting the script, especially when the stored XSS attack is performed. Node 6 represents the correct browser because different browsers process the scripts in some different ways and some of them implements security mechanism against XSS attacks [19]. This step is important to identify in which browsers it is possible to perform the attack.

C. Cross Site Request Forgery (CSRF)

A Cross Site Request Forgery (CSRF) attack forces the victims' browser, which is authenticated in the target application, to send a request to the vulnerable Web application. As the victim has an authenticated session, the vulnerable Web application performs the desired action in the name of the victim, according to the received request. This type of attack is possible because Web applications authorize requests based on credentials that are submitted automatically, as session cookies, IP address, SSL certificates, among others [5].

For the CSRF tree we covered the part of the CSRF attack that is related to the acceptance of the requests coming from another source. The other part related to the means used to lure the user to activate the request will not be covered as they are out of the defensive bounds that an application can have against CSRF. To test the application against vulnerabilities we consider that the user was tricked to activate the attack, for example, clicking on a link in another Web site.

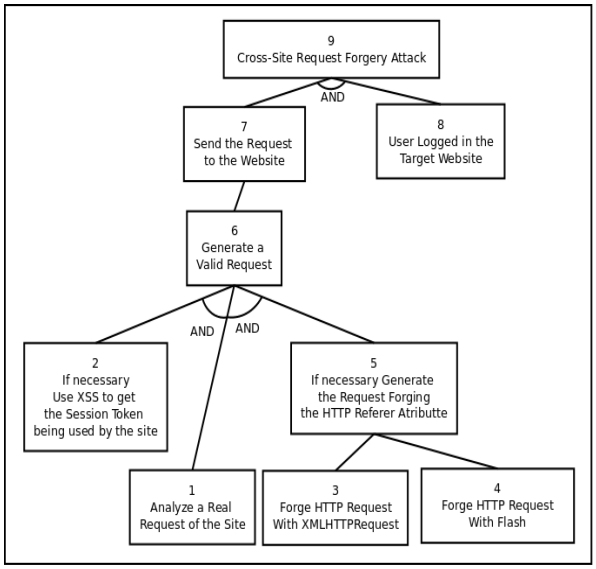


Figure 4. CSRF attack tree

In Figure 4, the main condition to perform a CSRF attack is to have the user logged in the site (node 8) because the attack will use its trust in the user authentication. If this condition is not satisfied, the attack could not be realized. The first step, represented by the node 1, is to analyze the request from the site that the attack will target in order to be able to reproduce it. If the site does not have CSRF countermeasures this step will lead to the next one, represented by node 6, because the request will be considered valid and will take effect on the site.

If the site uses any defensive measure it will be necessary to analyze the request and take additional

actions. A known defensive method consists in append different tokens in each request, but this approach can be bypassed if the application is vulnerable to XSS attacks (nodes 1 AND 2). It is possible because XSS attacks permit to get valid session tokens from the target application. The other scenarios of the tree (nodes 3, 1 AND 5 or nodes 4, 1 AND 5) shows how to overcome applications that uses verification of the HTTP (Hypertext Transfer Protocol) Referrer attribute, although this is not a recommended defensive measure.

IV. CASE STUDY

To assess the applicability of the tests based on attack trees, we developed a case study where three Web applications were tested against security vulnerabilities and the results are compared with the results of a commercial scanner tool, aiming to validate its effectiveness. Test cases were derived from the attack trees to cover the attack scenarios. Since the attack tree express attack goals in the same abstract notation as in security requirements, it is necessary to make some refinements, describing the possible useful data and code to be used in the tests, including their variations. Again, it can be done according to the attacker's capability and information available, for example, on Internet attackers' forums and blogs, as in the http://ha.ckers.org/xss.html website [18] .

In this paper, we present one test case example for each security vulnerability. Figure 5 shows the test case to SQL injection vulnerabilities, derived from the steps 5 AND 6, 7, 9 and 10 from the tree in Figure 2. It includes some examples to better illustrate the way the tests can be performed, once it can vary a lot.

1. Find data submission mechanisms (e.g., a search field).
2. Put specific special character (e.g., the "'" character) and submit the request.
3. Observe the normal response to a request. If the application returns syntax error message maybe it is vulnerable.
4. Inject a SQL query with an extra true condition to be concatenated to the original query. It will make the final query return a manipulated result. (e.g., ' union select load_file ('/etc/passwd'),1 #)
5. Observe the response to verify if the query is executed and brings the expected results.

Figure 5. Test case to SQL injection vulnerability

1. Find data submission mechanisms. (e.g., a login form).
2. Observe the normal response to a request.
3. Inject a SQL query with an extra true condition that will be concatenated to the original query and make the final query return a manipulated result. (e.g., 1 AND 1=1 or a" AND 0=0)
4. Observe the response to verify if the additional condition injected has been executed as SQL code (same response returned) and the site is vulnerable or if it has been treated as text input (different response)

Figure 6. Test case to Blind SQL Injection vulnerability

In Figure 5, the step 4 shows an example to inject a query to read the file "/etc/passwd", which content is included in the report displayed to the attacker. The symbol # is to comment the rest of the line being concatenated. Figure 6 shows a test case to an extension of the SQL injection vulnerability exploitation, called Blind SQL injection.

Blind SQL injection is identical to normal SQL injection, except that, when an attacker attempts to exploit an application, rather than getting a useful error message they get a generic page specified by the developer instead. This makes exploiting a potential SQL Injection attack more difficult but not impossible. An attacker can still steal data by asking a series of True and False questions through SQL statements [4].

1. Identify an user input data that are displayed in any of the applications screen (e.g. data tables, screen reports).
2. Insert javascript code (e.g. <script>alert('vulnerable to XSS'</script>) and submit the request.
3. Verify if the javascript code is executed (if the message popups).

Figure 7. Test case to XSS vulnerability

1. Log in the application (authenticate) and go through several protected areas.
2. Analyze the applications source code to construct a HTML attack page to perform the same requests than the application under test.
3. Access the attack page from a different server than the application under test.
4. Verify if the attack page action was executed by the target application, automatically.

Figure 8. Test case to CSRF vulnerability.

Figure 7 shows the test case to exploit XSS vulnerabilities. It corresponds to the steps 1, 2 AND 3, 5 AND 6 and 7 from the tree in the Figure 3. Next, a test case to CSRF vulnerabilities can be seen on Figure 8. This test case was derived from the steps 1, 6, 7 AND 8 and 9 from the tree in Figure 4.

The three Web applications selected were constructed using the Java programming language. Their source code is available to the tests. The first application is a Customer Relationship Manager – CRM application. It uses the MySQL Server 5.0 database and Tomcat 5.5.28 application server. Also, it uses technologies as Hibernate to object/relational persistence, framework Struts and JasperReports to support report generation. The second application is a Learning Management System for Distance Education. It uses the PostgreSQL 8.4 database and Tomcat 5.5.28 application server. It also uses Hibernate and AJAX (Asynchronous Javascript And XML). The third application is to call center and doctors schedule management of a big health treat company. It uses the framework Java Server Faces, the application server JBoss 4.2, MySQL Server 5.0 database, Hibernate and AJAX. These three web applications were selected because their services are applied to commercial and academic areas and they use recent technologies that can impact on the presence of security vulnerabilities. We do not mention their names because commercial licenses do not allow the publication of evaluation results.

The experiments were performed using a commercial Web vulnerability scanner as an aid. First, the commercial scanner was executed and some vulnerability were found. Then, we performed tests using the test cases derived from the attack trees. From these results, we compared and validated the results of the commercial scanner tool, classifying them as correct, false positive or lack of coverage. False positives are vulnerabilities detected by the scanner tool that could not be confirmed by any kind of known attack. Lack of coverage is vulnerabilities that the scanner tool was not able to detect.

The scanner tool brand is not mentioned because it is a commercial license. The applications, for the same reason, will be called App1, App2 and App3, without any special order.

V. RESULTS AND DISCUSSIONS

The applications were tested against XSS, SQL injection and CSRF vulnerabilities. Table 1 presents the results by vulnerability type. Considering the three applications scanned, the scanner tool detected respectively, 3 XSS, 4 SQL injection and 15 CSRF vulnerabilities. The results were validated through the tests guided by the attack trees and 2 XSS and 12 CSRF vulnerabilities were detected correctly. Still in Table 1, 8 vulnerabilities detected by the scanner tool was confirmed as false positive, being respectively 1 XSS, 4 SQL injection and 3 CSRF vulnerabilities. Performing more tests to exploit the application it was found 9 new vulnerabilities, i.e., vulnerabilities that already exist in the application and was not detected by the scanner tool. Due to time restrictions, the tests to find new vulnerabilities were not performed to the App3. So, these new vulnerabilities correspond only to App1 and App2.

TABLE 1. TESTS RESULTS BY VULNERABILITY TYPE

	XSS	SQL	CSRF	Total
Detected by the commercial scanner tool	3	4	15	22
Detected correctly	2	0	12	14
False positive	1	4	3	8
New vulnerabilities found (lack of coverage)	0	0	9	9

Table 2 shows the results of the experiments performed on Web applications. In the App1 it was detected 1 false positive and 7 new vulnerabilities. App2 have 3 false positive and 2 new vulnerabilities, while all vulnerabilities detected in App3 were false positives. As we have analyzed the context of the source code, we believe that App1 presented more vulnerability because it is less modularized than the other ones, i.e., App1 were constructed with more use cases and less coupling between them.

TABLE 2. TESTS RESULTS BY WEB APPLICATION

	App1	App2	App3	Total
Detected by the commercial scanner tool	13	5	4	22
Detected correctly	12	2	0	14
False positive	1	3	4	8
New vulnerabilities found (lack of coverage)	7	2	-	9

According to Table 1 and 2, from the 22 vulnerabilities found by the scanner tool, 14 were detected correctly, 8 were considered false positives

and 9 new vulnerabilities were found. Thus, in addition to validate the scanner results, testing the applications using the attack tress can bring better results to identify security vulnerabilities and to avoid wasting time and resources when a non-existent vulnerability (false positive) is reported (in this case, the development team may spend a lot of time trying to correct a false reported vulnerability before realizing that it does not really exists).

The false positive related to the XSS vulnerabilities is considered because the scanner tool integrates deprecated version of internet browsers. The same attack that was successfully executed by the tool, when executed in latest versions of internet browsers, has no effect, because these latest versions implement features that do not permit the execution of commons XSS attacks.

The SQL injection false positives were identified through the attacks and the analysis of the source code. All the applications use the Hibernate technology, which is an object/relational persistence and query service [20]. It consists, briefly, in express queries in its own portable SQL extension, as well as in native SQL, or with an object-oriented Criteria and Example API, i.e., it permits to encapsulate the queries and send objects to the database through predefined classes and methods, discarding the necessity of explicit SQL queries constructions. According to forums and some information available in technical websites [21]**Erro! Fonte de referência não encontrada.**, the code constructed with Hibernate is more difficult – but not impossible – to have the SQL injection vulnerability. However, the way that the application was coded, i.e., extremely encapsulated, do not open opportunities to develop successful attacks.

Most cases where CSRF false positives were identified, they happened in error pages. An attacker performing a CSRF attack to access an error page can be dangerous if the error page presents links or buttons that permit access to the application (as "back" buttons which bring back the user to the last page he/she accessed) or if the error page displays private information about the system (such as database name or table names). For both applications, the error pages do not present any way of accessing application functionalities or private information. Hence, we considered theses cases as false positives because a CSRF attack when accessing the error pages is useless.

The applications were constructed using frameworks like Struts or Java Server Faces. These frameworks, by default, have a filter to XSS attacks. Unless the developer clearly specifies in the source code that it is permitted to execute XSS scripts, the framework treats the scripts (identified, for example, by <script></script>) as strings. This explains the low index of XSS vulnerabilities in the applications. However, even with the use of the frameworks, two XSS vulnerabilities were detected correctly, what reinforces the importance of testing the applications against security vulnerabilities.

VI. CONCLUSIONS

In the present paper were presented the attack trees modeling to SQL injection, XSS and CSRF vulnerabilities. These attack trees as useful structures to perform security tests because it permits that the tester analyze the application with an attacker vision. Thus, test cases can be derived from the trees, considering the necessary steps to perform a successful attack.

A case study was developed and the attack trees were used to guide the tests in real commercial management Web applications. Three Web applications were tested with the support of a scanner tool, and the tests were useful to validate the scanner results and to find new security vulnerabilities that the tool was not able to detect. Results show that the tests could identify many false positives and lack of coverage, showing the tool limitations and identifying the reason why the scanner tools results are incorrect, as, for example, the use of different development technologies (Hibernate) and some deprecated features of the tool (the internal internet browser). Also, the tests provided better results to identify security vulnerabilities and to avoid wasting time and resources.

As future work we intend, based on the modeled attack trees, to construct an automatic tool to inject the attacks in the applications in order to find the respective vulnerabilities.

REFERENCES

[1] J. Fonseca, M. Vieira. "Mapping software faults with web security vulnerability". IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2008), Anchorage, USA, 2008.

[2] Uto, N. and Melo, S. P. "Vulnerabilidades em Aplicações Web e Mecanismos de Proteção". In IX Simpósio Brasileiro de Segurança da informação e de sistemas computacionais – SBSEG, cap. 6 dos mini-cursos SBSEG, Campinas, Brasil, 2009.

[3] Halfond, W. G.; Viegas, J.; Orso, A. "A Classification of SQL-Injection Attacks and Countermeasures", In Proceedings of the International Symposium on Secure Software Engineering - ISSSE 2006, Arlington, Virginia, 2006.

[4] CGISecurity.com. "Web applications news and more". Available in http://www.cgisecurity.com. Last access on March/2010.

[5] Auger, R. "The Cross-Site Request Forgery (CSRF/XSRF) FAQ". Available in http://www.cgisecurity.com/csrf-faq.html. Last access on march/2010.

[6] OWASP Top 10 Project. Available in http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, Last access on February/2010.

[7] NVD (2010) “National Vulnerability Database”. Available in http://nvd.nist.gov/. Last Access on March/2010.

[8] NIST (2010) “National Institute of Standards and Technology”. Available in http://www.nist.gov/index.html. Last access on March/2010.

[9] Lin, X., Zavarsky, P., Ruhl, R., and Lindskog, D. “Threat Modeling for CSRF Attacks.” Proceedings of the 2009 international Conference on Computational Science and Engineering, 2009, pp 486-491.

[10] Fonseca, J.; Vieira, M.; Madeira, H. “Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks”, 13º IEEE Pacific Rim Dependable Computing Conference, Melbourne, Victoria, Australia, 2007.

[11] Vieira, M.; Antunes, N.; Madeira, H. "Using Web Security Scanners to Detect Vulnerabilities in Web Services". In Proceedings of IEEE/IFIP Intl Conf. on Dependable Systems and Networks, DSN 2009, Lisbon, Portugal, June 2009..

[12] Basso, T., Fernandes, P.C.S., Jino, M., Moraes, R., “Analysis of the Effect of Java Software Faults on Security Vulnerabilities and Their Detection by Commercial Web Vulnerability Scanner Tool”. In Proceedings of IEEE DSN Workshops (DSNW), Chicago, USA, June/2010.

[13] Schneier, B. “Attack Trees: Modeling Security Threats”, Dr. Dobb’s Journal, 1999.

[14] Byres, E. J., Franz, M.; Miller, D. “The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems”. International Infrastructure Survivability Workshop (IISW '04), IEEE, Lisboa, Portugal, 2004.

[15] Martins, E.; Morais, A.; Cavalli, A. “Generating attack scenarios for the validation of security protocol implementations.” 2nd. Brazilian Workshop on Systematic and Automated Software Testing (SAST). Campinas, Brasil, 2008.

[16] Fung, C., Chen Y. L., Wang, X., Lee, J. Tarquini, R., Anderson, M. Linger, R. “Survivability analysis of distributed systems using attack tree methodology”. Military Communications Conference – MILCOM, 2005, pp 583-589

[17] Edge, K., Raines, R., Grimaila, M., Baldwin, R., Bennington, R., and Reuter, C. “The Use of Attack and Protection Trees to Analyze Security for an Online Banking System.” In Proceedings of the 40th Annual Hawaii international Conference on System Sciences, 2007, pp 144b.

[18] XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion. Available in http://ha.ckers.org/xss.html. Last access on May/2010

[19] Browser Scope Project – Security Test Results. Available in http://browserscope.org/?category=security. Last access on May/2010.

[20] Hibernate. Available in https://www.hibernate.org/. Last Access on January/2010.

[21] OWASP – Preventing SQL injection in Java. Available in http://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java#Hibernate. Last access on January/2010.

[22] CWE – Common Weakness Enumeration. Available in http://cwe.mitre.org/data/definitions/564.html. Last access on January/2010