Universidade Federal do Rio de Janeiro

CÉLIO BORGES TAQUARY SEGUNDO

USO DE ÁRVORES DE ATAQUE E TÉCNICAS DE MUTAÇÃO DE CÓDIGO NA SEGURANÇA DE APLICAÇÕES WEB

DISSERTAÇÃO DE MESTRADO

RIO DE JANEIRO
2010

Instituto de Matemática

Núcleo de Computação Eletrônica

CÉLIO BORGES TAQUARY SEGUNDO

USO DE ÁRVORES DE ATAQUE E TÉCNICAS DE MUTAÇÃO DE CÓDIGO NA SEGURANÇA DE APLICAÇÕES WEB

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, do Instituto de Matemática e do Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientador: Luiz Fernando Rust da Costa Carmo

Rio de Janeiro

2010

T175 Taquary Segundo, Célio Borges.
Uso de árvores de ataque e técnicas de mutação de código na segurança de aplicações web / Célio Borges Taquary Segundo – 2010.
83 f: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Núcleo de Computação Eletrônica, 2010.

Orientador: Luiz Fernando Rust da Costa Carmo.

1. Segurança de Aplicações Web - Teses. 2. Árvore de Ataque – Teses. 3. Mutação de Código – Teses. I. Luiz Fernando Rust da Costa Carmo (Orient.). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática. Núcleo de Computação Eletrônica. III. Título.

CDD

CÉLIO BORGES TAQUARY SEGUNDO

USO DE ÁRVORES DE ATAQUE E TÉCNICAS DE MUTAÇÃO DE CÓDIGO NA SEGURANÇA DE APLICAÇÕES WEB

Dissertação submetida ao corpo docente do Programa de Pós-graduação em Informática do Instituto de Matemática e do Núcleo de Computação Eletrônica, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada em: Rio de Janeiro, 28 de abril de 2010.

_____

Profº. Luiz Fernando Rust da Costa Carmo, Docteur, PPGI/UFRJ (Orientador)

_____

Profª. Luci Pirmez, Docteur, PPGI/UFRJ

_____

Profº. Paulo Henrique de Aguiar Rodrigues, Ph.D, PPGI/UFRJ

_____

Prof. Artur Ziviani, Docteur, LNCC

DEDICATÓRIA

Dedico este *Mestrado* aos meus pais, Célio e Maria Aparecida, pelo incentivo e apoio em todas as minhas escolhas e decisões.

AGRADECIMENTOS

Agradeço a Deus pelo dom da vida e por estar sempre ao meu lado.

Aos meus pais por me proporcionarem um alicerce de valores necessários para o meu desenvolvimento o que permitiu aprimorar meus conhecimentos e me tornar uma pessoa melhor.

Ao meu amigo Paulo Melo pelo incentivo e dicas sobre o Ruby on Rails.

Aos meus amigos pela motivação e compreensão nos momentos de ausência.

Aos meus colegas de mestrado pelo apoio e companheirismo durante todo o curso.

Ao meu orientador, professor Luiz Fernando Rust, pela motivação, apoio, empenho, interesse e por sugerir acréscimos significativos na dissertação.

RESUMO

TAQUARY SEGUNDO, Célio Borges. **Uso de árvores de ataque e técnicas de mutação de código na segurança de aplicações web.** 2010. 83 f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2010.

A proliferação de aplicações baseadas em web tem aumentado a exposição das empresas a uma variedade de ameaças. Há várias etapas no ciclo de vida das aplicações que são destinadas a prevenir ou mitigar essas ameaças. Os testes de segurança são muito úteis, desde que sejam eficientes. Este trabalho foca a validação dos testes de segurança das aplicações web. Esta dissertação propõe uma metodologia para validação de ferramentas e testes de segurança de aplicações web, baseada em árvores de ataques, derivadas das vulnerabilidades conhecidas e divulgadas pelas comunidades de segurança afins. Para validar a eficácia dos testes derivados dessas árvores de ataque, inserem-se vulnerabilidades nas aplicações através de técnicas de Mutação de Código.

ABSTRACT

TAQUARY SEGUNDO, Célio Borges. **Uso de árvores de ataque e técnicas de mutação de código na segurança de aplicações web.** 2010. 83 f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2010.

The proliferation of web-based applications has increased the exposure of companies to a variety of threats. There are several stages in the life cycle of the applications that are designed to prevent or mitigate those threats. The safety tests are very useful, provided they are efficient. This work focuses on the validation of security testing of web applications. It proposes a methodology for validation of tools and security testing of web applications, based on attack trees, derived from known vulnerabilities disseminated by related security communities. To validate the effectiveness of tests derived from these attack trees, security vulnerabilities are inserted into applications through Mutation Code techniques.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta uma introdução da pesquisa documentada nesta dissertação, abordando a caracterização do problema, o estado da arte, os objetivos e a estrutura da dissertação.

1.1 CARACTERIZAÇÃO DO PROBLEMA

Nos últimos anos, as aplicações web se tornaram extremamente populares, tanto para grandes organizações como para indivíduos comuns. Quase tudo é comercializado, armazenado ou disponibilizado na web, através de aplicações como sites pessoais, comércio eletrônico, bancos, webmails, redes sociais, blogs, fóruns etc. Elas se tornaram tão importantes no nosso dia a dia que não surpreende que se tornem alvo de pessoas mal intencionadas. Um exemplo desse fato é o crescimento da percentagem de ataques baseados na web, que pulou de 25% em 2000 para 61% em 2006, segundo os dados da base de dados de vulnerabilidades da Common Vulnerabilities and Exposures (CVE, 2006).

Dessa forma, uma grande parte das aplicações web exige soluções seguras, seja pelo fato de lidarem com ativos de alto valor ou manipularem informações sigilosas ou mesmo por ser objeto de regulamentação por parte do governo ou entidades reguladoras. Embora exista uma crescente preocupação com a segurança das aplicações web, evitar as vulnerabilidades de segurança é um grande desafio: são vários os fatores que contribuem para tornar essa tarefa ainda mais difícil, conforme veremos.

As diversas tecnologias existentes para o ambiente web trouxeram muitos benefícios e facilidades para os usuários, mas também aumentaram em muito a complexidade das aplicações web (Hoglund et al., 2006). Essa diversidade possibilita uma grande quantidade de escolhas quanto a modelos, serviços, algoritmos, parâmetros e estrutura de dados que podem ser empregados no desenvolvimento, tornando as aplicações mais suscetíveis a falhas e, consequentemente, bem mais expostas a ataques.

A pressão do mercado sobre as empresas de Tecnologia de Informação fazem com que as mesmas almejem sempre lançar novos produtos antes da concorrência de forma a obter vantagem competitiva. Essa corrida contra o tempo faz com que muitas empresas de TI acabem colocando no mercado produtos/aplicações não seguros, uma vez que garantir a segurança é uma tarefa demorada e os resultados nem sempre são evidentes, o que lhes

permite obter uma vantagem competitiva injusta sobre aquelas que respeitam as normas de segurança.

Outro fator que concorre para dificuldade de prevenir essas vulnerabilidades é o fato de que é comum encontrar desenvolvedores e administradores de aplicações web sem o conhecimento necessário ou experiência na área de segurança (Fonseca, 2009).

Complementarmente, a maioria dos mecanismos existentes de defesa específicos para aplicações web atuam de modo reativo, restringindo o acesso não autorizado e/ou não permitindo o uso de determinadas portas. A desvantagem da abordagem reativa é que essa não atinge diretamente o cerne do problema: a existência de software de má qualidade. Torna-se, portanto, imprescindível o desenvolvimento de técnicas mais acuradas (seguras) que permitam a efetiva verificação de pontos vulneráveis dos softwares, e a sua posterior eliminação.

Com intuito de cobrir esta lacuna, várias ferramentas são lançadas no mercado com propósito de realizar testes de segurança em aplicações web. No entanto, essas ferramentas também são softwares e portanto podem conter os mesmos problemas. Então, como podemos garantir que essas ferramentas são eficazes para encontrar as vulnerabilidades das aplicações e garantir a segurança das mesmas?

Esta dissertação trata sobre a validação de testes de segurança e ferramentas que se determinam para tal, através da proposição de uma metodologia de validação de testes de segurança. Uma aplicação web foi utilizada como estudo de caso para validação experimental do uso da metodologia proposta.

1.2 CONTEXTUALIZAÇÃO

Uma abordagem interessante e bastante explorada na validação dos testes funcionais de uma aplicação é a utilização de técnicas de injeção de falhas. Uma grande parte desses estudos baseia-se na inserção de erros através de pequenas modificações no código fonte, com intuito de verificar se os testes irão, posteriormente, identificar essas modificações. No entanto, essa técnica não é totalmente adequada para testes de segurança, uma vez que essas pequenas modificações não são capazes de inserir vulnerabilidades de segurança para a validação de testes de segurança.

Embora existam várias abordagens para validação de testes funcionais em aplicações, encontramos na literatura poucas referências sobre a validação de testes de segurança.

Dentre as abordagens relacionadas à segurança de aplicações web, podemos destacar o trabalho de Fonseca (2009) quanto ao uso de um injetor de vulnerabilidades para aplicações web. A ideia subjacente é a de que, através do ataque controlado a uma aplicação web, pode-se avaliar o comportamento de uma série de mecanismos de segurança (por exemplo, IDS, scanners de vulnerabilidades, firewall etc.), cujo objetivo é proteger uma aplicação web. Para emular com precisão as vulnerabilidades do mundo real na Internet, Fonseca (2008) utilizou um estudo de campo sobre as vulnerabilidades de segurança real e usou uma ferramenta de injeção de vulnerabilidades. No presente trabalho a injeção de vulnerabilidades é feita seletivamente por técnicas de mutação de código.

Huang et al (2003) apresentam uma proposta que se aproxima, em termos funcionais, da proposta de injeção de vulnerabilidades para validação de mecanismos de segurança apresentada por Fonseca (2009). Ela se baseia explicitamente na injeção de erros seguida de monitoração comportamental. A diferença básica é que sua abordagem com injeção de erros amplifica muito o escopo de validação da aplicação, pois tem que lidar com erros imprevisíveis que não geram vulnerabilidades.

Outro estudo relacionado à segurança, mas com uma abordagem voltada apenas para detecção de vulnerabilidades, diz respeito à aplicabilidade estática ou dinâmica dos testes de segurança. Apesar das abordagens estáticas já serem atualmente usadas para a identificação de vulnerabilidade em programas do mundo Unix com relativo sucesso, existe um consenso quanto ao comprometimento da sua eficiência em aplicações web, dado o massivo número de interações em tempo de execução entre os seus diversos componentes (Tevis e Hamilton, 2006). Essas interações geram uma nova gama de vulnerabilidades.

1.3 OBJETIVOS

Nesse contexto, esta dissertação tem como objetivo geral apresentar uma proposta de metodologia de análise e validação de ferramentas de testes de segurança de aplicações web, que cobrem um determinado escopo de vulnerabilidades e que usam tanto técnicas estáticas como dinâmicas na detecção dessas vulnerabilidades.

Na busca de uma melhor representação e uma modelagem estruturada desse conjunto de vulnerabilidades, das formas de ataques aplicáveis e das respectivas técnicas utilizadas para detectá-las, optou-se pelo uso de árvores de ataque. O encaminhamento intuitivo provido

pelas árvores de ataque facilita o entendimento e a escolha das ferramentas apropriadas para identificar uma determinada vulnerabilidade.

A metodologia proposta compreende basicamente a geração de uma árvore de ataque para determinar o escopo de validação e, em seguida, a execução de ferramentas de testes de segurança em uma aplicação, de forma a cobrir todas as vulnerabilidades determinadas pela árvore de ataque. Caso as ferramentas utilizadas consigam identificar as vulnerabilidades as quais se propõem, temos um indicativo de que elas são eficazes para aquele contexto. Caso contrário, é preciso validar essas ferramentas antes de determinar que uma aplicação seja segura.

Para validar a eficácia das ferramentas de teste de segurança que não identificaram vulnerabilidades, inserem-se intencionalmente vulnerabilidades na aplicação alvo, através de técnicas de Mutação de Código (injeção de falhas). Em seguida, aplicam-se os mesmos testes.

O resultado da repetição dos testes indica quão adequado são as ferramentas de testes de segurança usadas na validação da segurança de uma aplicação web.

Com intuito de garantir que as mutações realizadas no código da aplicação realmente incluíssem vulnerabilidades de segurança, pesquisamos pelas vulnerabilidades conhecidas e divulgadas pelas comunidades de segurança afins e a forma como elas são corrigidas.

Essa metodologia foi validada experimentalmente através do estudo de um caso bem representativo, fornecendo resultados surpreendentes.

O escopo do conjunto de dados de testes, utilizados para testar a aplicação alvo do estudo de caso, inclui ferramentas automáticas e técnicas manuais de *ethical hacking*[1] para encontrar vulnerabilidades de segurança em aplicações web.

Dentre as contribuições desta dissertação, destaca-se a elaboração de uma metodologia inovadora que gera subsídios para uma real verificação da eficácia de uma ferramenta de avaliação de segurança de aplicativos para cenários específicos, bem como para a validação de segurança da própria aplicação web.

Embora o trabalho realizado nesta dissertação esteja focado em aplicações web, nada impede que ele seja empregado em outras áreas.

---

[1] Prática de invadir computadores sem más intenções, simplesmente para encontrar os riscos de segurança e comunicá-las as pessoas responsáveis.

1.4 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em cinco capítulos, sendo a introdução o primeiro capítulo.

O capítulo 2 faz uma revisão dos conceitos teóricos necessários para compreensão deste trabalho, expondo as principais referências na literatura sobre o tema que serviram de base para o seu desenvolvimento.

O capítulo 3 apresenta uma proposta de metodologia para a validação de testes de segurança em aplicações web.

O capítulo 4 descreve a utilização da metodologia proposta, através de validação experimental, em uma aplicação web escolhida para o estudo de caso, e descreve a avaliação dos resultados obtidos.

O capítulo 5 descreve as considerações, limitações e dificuldades encontradas e sugestões para continuidade desta pesquisa.

2 CONCEITOS BÁSICOS

2.1 INTRODUÇÃO

É realizada, neste capítulo, a exposição dos conceitos básicos sobre a representação, geração e estrutura de uma Árvore de Ataque. Em seguida são apresentados os conceitos sobre Testes de Software e o detalhamento de uma das técnicas de teste de software, a Mutação de Código. Por fim, são apresentadas algumas características específicas dos Testes de Segurança, que são o foco principal desta dissertação, além das principais vulnerabilidades que podem ocorrer em aplicações web e suas formas de verificação.

2.2 ÁRVORE DE ATAQUE

2.2.1 Definição

O conceito de árvores de ataque é derivado do conceito de árvores de erro. Árvores de erro são utilizadas para descrever como os erros se propagam em sistemas de software, e a sua análise pode ser utilizada para testar o software (Viega e McGraw, 2002). Enquanto as árvores de erro são mais usadas para modelar o modo como os problemas ocorrem em sistemas críticos, dado que são construídas com foco na propagação de erros, as árvores de ataque modelam o modo como um ataque pode ser realizado ou como uma ameaça pode ser concretizada. A introdução de um atacante ou grupo de atacantes torna possível criar um modelo de ameaças para o sistema incluindo o aspecto do alvo dos ataques. Considerações sobre a probabilidade de concretização de um ataque com base nos custos e nas ferramentas disponíveis ou na motivação do atacante dão uma imagem mais fundamentada do nível de risco do sistema.

A árvore de ataque é uma forma de identificar e documentar os possíveis ataques em um sistema de uma forma estruturada e hierarquizada. A estrutura em árvore fornece uma imagem detalhada de vários ataques que podem ser empregados para comprometer um sistema. Com o uso de árvores de ataque, cria-se uma representação reutilizável de problemas de segurança que ajudará a focar os esforços de mitigação de ameaças. Uma equipe de testes pode usar as árvores para criar planos de teste que validem o projeto de segurança. Arquitetos

podem usar as árvores para avaliar o custo da segurança de abordagens alternativas. Os desenvolvedores podem usar as árvores para tomar decisões durante a codificação.

2.2.2 Contextualização

Schneier (1999) introduziu o conceito de árvores de ataque como uma forma de modelo de ameaças contra os sistemas computacionais. A ideia do conceito é compreender as ameaças para um sistema, descrevendo em uma estrutura de árvore as diferentes formas pelas quais o sistema pode ser atacado e comprometido. O nó raiz da árvore representa o objetivo de um atacante, ou o ativo/bem a ser protegido. Os nós folhas representam as diferentes formas de alcançar esse objetivo. Com base na estrutura da árvore e na informação de quem são os atacantes e das capacidades que eles possuem, seremos capazes de estabelecer as contramedidas adequadas e lidar com as verdadeiras ameaças.

Moore, Ellison e Linger (2001) descrevem um meio para documentar ataques de segurança de informações de uma forma estruturada e reutilizável, usando árvores de ataque. O trabalho aborda diversas questões sobre o uso de árvores de ataque para a documentação das vulnerabilidades. O foco foi documentar e organizar padrões genéricos de ataques para poder reutilizá-los na construção da árvore de ataque, e assim aperfeiçoar o modelo de árvore de ataque com a reutilização de padrões genéricos de ataque.

Mauw e Oostdijk (2006) apresentam uma sugestão para formalizar os conceitos de árvore de ataque. É descrito o conceito de conjunto de ataque como um nível de abstração de uma árvore de ataque. É sugerida uma semântica formal para descrição de como as árvores de ataque podem ser manipuladas durante a sua construção e análise. Os autores argumentam que a execução dos trabalhos de semântica e formalização do método é necessária para se tornar possível a construção de ferramentas automatizadas.

Khand (2009) descreve que os tipos de nós das árvores de ataque convencionais não são suficientes para representar adequadamente os ataques para sistemas tolerantes à intrusão. São apresentados novos tipos de nós, como nós prioritários, nós que representam a quantidade de subobjetivos que devem ser alcançados para os ataques obterem sucesso, nós que possuem condições para ocorrerem, nós que forçam uma determinada sequência e nós que são usados para modelar ataques que mudam com o tempo. São apresentados três modelos de árvores de ataque para sistemas de segurança: Servidor Seguro Distribuído tolerante a intrusão, Servidor de Dados e Servidor de Aplicação com arquitetura tolerante a intrusão.

Além dos trabalhos citados anteriormente, destacamos um trabalho relacionado à validação de segurança de aplicações utilizando árvore de ataque. Em Li & He (2008), é usado um modelo de árvore de ameaças estendido, baseado em dados estatísticos históricos, para tratar dos aspectos de segurança durante o projeto de desenvolvimento da aplicação web.

Nesta dissertação as árvores são usadas para representar as vulnerabilidades em aplicações web e as formas para explorá-las, diferente da proposta de Li & Hi que é dedicada à mitigação das ameaças ainda no estágio de desenvolvimento da aplicação. O modelo de árvore que propomos também se difere dos demais em relação a profundidade da árvore, que relaciona também as formas de ataque a algumas das principais vulnerabilidades de aplicações web bem como uma forma de ataque de "baixo nível", que pode ser verificada por ferramentas de testes de segurança. A árvore é ainda divida em faixas que facilitam a identificação de qual é o objetivo de ataque (onde), o local onde realizar o ataque (onde) e a forma de realizar o ataque (como).

2.2.3 Representação gráfica e semântica

Pode-se observar que várias semânticas foram propostas, desde a ideia inicial lançada por Schneier. O modelo básico descreve a árvore de ataque com dois tipos diferentes de nós, nós AND e nós OR. Nos nós do tipo OR, pelo menos um sub-objetivo deverá ser satisfeito para atingir o objetivo do nó raiz. Nos nós do tipo AND, o objetivo do nó só é atingido quando cada sub-objetivo for satisfeito. Valores podem ser atribuídos aos nós folha, como declarações booleanas do tipo "possível/impossível" ou "exige equipamentos especiais/não exige equipamentos especiais". Para se obter o resultado de um nó, cálculos booleanos podem ser realizados sobre os valores dos seus nós sub-objetivos baseado na expressão booleana do nó (AND ou OR).

Os princípios básicos são mostrados no exemplo de árvore da Figura 1. O exemplo ilustra uma árvore de ataque contra um cofre trancado. Expressões OR são aplicadas na árvore através da ligação dos sub-objetivos ao nó resultante. Expressões AND são representadas graficamente por um arco ligando os dois sub-objetivos, acrescido da palavra "AND" no interior do arco.

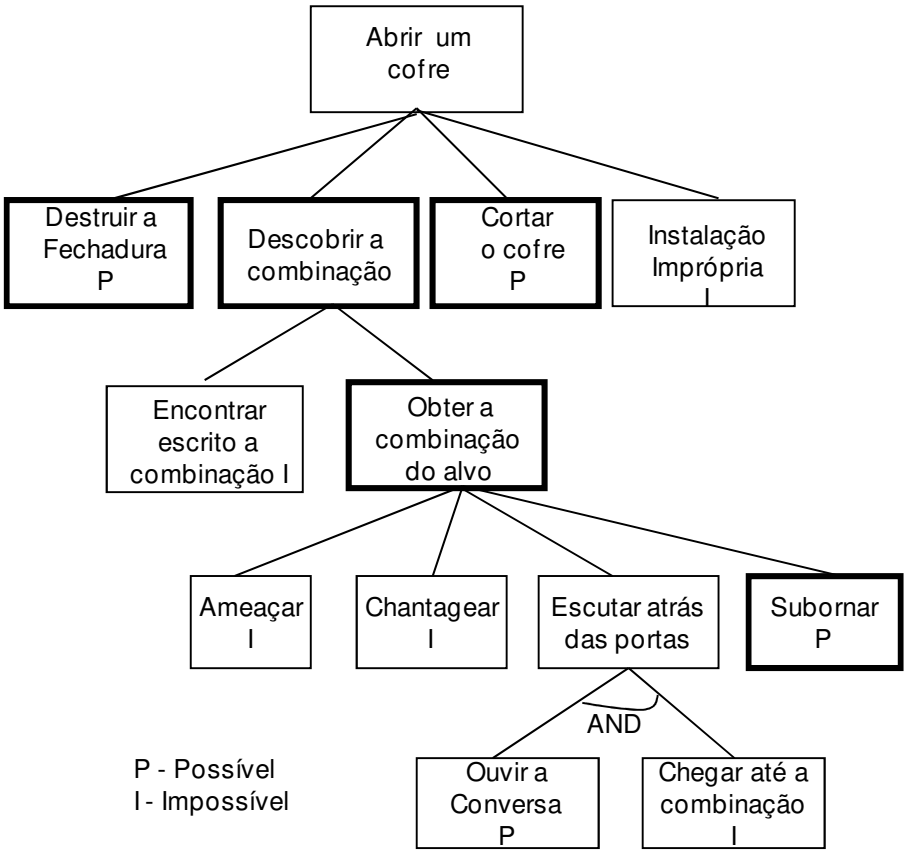

Figura 1 - Exemplo de árvore de ataque

Realizando os cálculos booleanos, podemos concluir que as formas possíveis de "Abrir um cofre" seriam as representadas pelos caminhos em que os nós folhas estão destacados. A opção do nó "Descobrir a combinação", por exemplo, só seria possível através do nó folha "Subornar", visto que "Ameaçar" e "Chantagear" são impossíveis e apesar de "Ouvir a Conversa" ser possível, "Chegar até a combinação" é impossível, tornando o nó "Escutar atrás da porta" impossível para se alcançar a "Combinação do alvo".

No modelo de Schneier, valores não booleanos indicando o esforço de um ataque (custo ou tempo) também podem ser atribuídos a um nó. Entretanto, os cálculos realizados sobre esses valores são limitados ao: (i) somatório dos valores dos sub-nós no caso de nós AND, e (ii) valor do sub-nó com custo mais baixo no caso de nós OR.

2.3 TESTE DE SOFTWARE

2.3.1 Definição

Teste de software é o processo de investigação do software com o objetivo de encontrar seus defeitos. Segundo Myers (1979), teste é o processo de executar um programa com a intenção de descobrir defeitos.

Para descrever os testes, são criados os Casos de Testes, que atendem a um determinado critério relacionado a alguma técnica de teste de software. Os casos de testes são formados por um conjunto de dados de entrada, válidos e inválidos dentro de um escopo definido, e pelas respectivas respostas esperadas.

Caso a saída de uma determinada entrada seja diferente da resposta esperada, o caso de teste identificou um defeito no software.

O teste de software é um processo crítico para a qualidade do software. À medida que os resultados dos testes são acumulados e analisados, começa a surgir uma indicação da confiabilidade e qualidade do software.

Os critérios de teste definem os requisitos a serem atendidos e ajudam a orientar a escolha dos dados de teste, a avaliação da qualidade dos testes e a seleção dos requisitos de atendimento para conclusão das atividades de teste. Atender um critério de teste significa atender todos os requisitos definidos no critério.

Os critérios de teste utilizados em conjunto com as técnicas de teste têm aumentado a probabilidade de encontrar defeitos. O objetivo é selecionar melhor os dados que serão utilizados nos testes. Três técnicas de teste de softwares podem ser associadas à definição dos critérios de teste de software. A técnica Funcional, a Estrutural e a Baseada em Defeitos. A diferença entre essas três técnicas está no tipo de informação utilizada para derivar os requisitos de teste de cada critério (Guidetti, 2005). Elas serão detalhadas abaixo.

2.3.2 Técnica Funcional

Os critérios de teste associados à Técnica Funcional, também conhecida como caixa preta, avaliam o comportamento externo do software. Esses critérios não consideram os detalhes de implementação do software, preocupando-se apenas com os aspectos funcionais

do sistema. Os casos de testes são baseados na especificação do programa. Dados de entrada são fornecidos e a resposta obtida é comparada com a resposta esperada.

Quanto mais entradas são fornecidas, mais rico será o teste. Numa situação ideal todas as entradas possíveis seriam testadas, mas na ampla maioria dos casos isso é impossível (Myers, 2004). Uma abordagem mais realista é escolher um subconjunto de entradas que maximize a eficiência do teste.

O Particionamento de Equivalência (Pressman, 2004) é uma delas. Nesse critério, o testador divide o domínio de entrada de dados em classes. A partir das classes de equivalência, constroem-se casos de teste que atuam nos limites superiores e inferiores das classes, de forma que um número mínimo de casos de teste permita a maior cobertura de teste possível.

Outro critério utilizado é a Análise do Valor Limite (Pressman, 2004). Nesse critério, os casos de teste utilizam valores limites, derivados das classes de equivalência. Opta-se por esses valores porque o maior número de erros ocorre com valores de testes próximos das extremidades do domínio de entrada, e não com os centrais.

O critério Grafo de Causa-Efeito (Pressman, 2004) é um critério funcional que obtém as possíveis condições de entrada (CAUSAS) e as possíveis ações (EFEITOS) do programa, utilizando essas informações para a construção de um grafo, que é então convertido em uma tabela de decisão utilizada na construção dos casos de teste.

2.3.3 Técnica Estrutural

Os critérios de teste associados à Técnica Estrutural**,** conhecida como caixa branca, analisa o comportamento interno do software. Esses critérios trabalham diretamente sobre o código fonte, levando em consideração os detalhes da implementação.

Esse tipo de teste é desenvolvido a partir da análise do código fonte e da elaboração de casos de teste que cubram todas as possibilidades do componente de software. Dessa maneira, todas as variações relevantes originadas por estruturas de condições são testadas. Conforme afirma Bueno:

> "Os critérios de teste estrutural estabelecem componentes estruturais do programa a serem exercitados. Satisfazer um critério de teste estrutural significa exercitar todos os componentes requeridos pelo critério" (Bueno, 1999, pg.10)

Os critérios baseados em Fluxo de Dados usam informações do fluxo de dados do programa para derivar os requisitos de teste. Esses critérios analisam o uso das variáveis, determinando associações entre os pontos onde um valor é atribuído a uma variável, chamados de Definição da variável, e os pontos onde esse valor é utilizado, chamados de Uso da variável. Esses critérios exigem a execução do caminho do ponto onde a variável foi definida até o ponto onde foi utilizada sem passar por uma redefinição (Guidetti, 2005).

Os critérios baseados no Fluxo de Controle usam as características de controle da execução do programa para determinar quais estruturas são requeridas. Esses critérios utilizam uma representação gráfica do programa chamada Grafo de Fluxo de Controle (Rapps e Weyuker, 1985) para determinar as estruturas que devem ser cobertas pelos casos de teste. Esses critérios exigem que cada nó, cada ramo e cada caminho do grafo de programa sejam executados pelo menos uma vez.

2.3.4 Técnica Baseada em Defeitos

Os critérios de teste associados à Técnica Baseada em Defeitos**,** conhecida também como Técnica Baseada em Erros, baseiam-se em informações sobre erros que os programadores costumam cometer durante a codificação do software, na fase de desenvolvimento do sistema.

O critério **Mutação de Código** é um exemplo de critério da técnica baseada em defeitos e será mais bem detalhado em uma seção própria, por se tratar da técnica que é foco deste trabalho.

2.4 MUTAÇÃO DE CÓDIGO

2.4.1 Definição

Testes por Mutação de Código ou Análise de Mutantes consistem num método de teste de software que realiza a injeção de falhas em um programa, através da modificação de pequenos trechos do código fonte, gerando, consequentemente, mutantes desse programa. As mudanças, chamadas de mutações, baseiam-se em operadores de mutação bem definidos que imitam erros típicos de programação (tais como o uso errado de operador ou nome de variável), ou forçam a criação de testes (tais como levar cada expressão a zero).

O objetivo é ajudar o testador a: (i) desenvolver testes eficazes e (ii) localizar falhas nos dados utilizados para testar o programa, ou em seções de código que são raramente ou nunca acessadas durante a execução.

2.4.2 Contextualização

O critério Análise de Mutantes foi originalmente proposto por DeMillo, Lipton e Sayward (1978) e é um exemplo da técnica de teste de software Baseada em Defeitos.

Com a enorme disponibilidade de capacidade computacional, tem havido um ressurgimento de análise por mutação na comunidade científica da computação (Guidetti, 2005). As principais iniciativas buscam definir testes por mutação para programas em linguagem orientada a objeto e linguagens não procedurais, como XML, SMV e máquinas de estado finito.

Em Shahriar (2008) é apresentado um trabalho que faz uso da técnica de mutação para testar *bugs* que causam negação de serviço em aplicações. Foi desenvolvido um protótipo de ferramenta de teste que gera 8 diferentes mutantes e os analisam em quatro programas de código fonte aberto.

Giacometti et al. (2002) apresentam um conjunto operadores de mutação para validação de aplicações paralelas que utilizam o ambiente de passagem de mensagens PVM – *Parallel Virtual Machine*. A nossa proposta de difere destas anteriores por usar operadores de mutação concebidos de forma a sempre inserir uma vulnerabilidade.

Na mesma linha, Vicenzi et al. (1999) propõem a redução dos custos na realização de teste por mutação através da criação de operadores essenciais de mutação. A aplicação dos testes é apoiada por uma ferramenta para seleção do conjunto de operadores essenciais.

2.4.3 Análise de Mutantes

Para descrever como funciona o critério de Análise de Mutantes iremos supor um programa *P* qualquer. O programa *P* possui um domínio de entrada de dados *D*. A partir desse domínio temos um subconjunto próprio denominado *T*, que é o conjunto de dados de teste. Esse conjunto possui dados *t*, pertencentes ao conjunto *T*, que são fornecidos pelo profissional de teste para avaliar a qualidade de *P*. O programa *P* é executado e testado com os casos de

teste *T*. Quando *P* é executado, podem ocorrer falhas. Caso alguma falha seja encontrada em *P(t)*, o defeito é corrigido.

No entanto, *P* pode conter falhas que o conjunto de dados de teste *T* não consiga revelar. Para identificar o problema são criados programas mutantes de *P*.

2.4.4 Geração de Mutantes

Considere um programa *m* sintaticamente correto, gerado através de uma única mudança sintática em *P*. A mudança em *P* será determinada por uma regra *r*. A nova versão de *P* é denominada mutante ou código mutante. A regra *r* é denominada Operador de Mutação.

Para exemplo de um operador de mutação, considere o seguinte fragmento de código C++:

```
if (a && b)
   c = 1;
else
    c = 0;
```

Um operador de mutação simples poderia conter a seguinte regra: substituir em todo o código os caracteres '& &' por '| |'. A aplicação desse operador de mutação no código apresentado produziria o seguinte código mutante:

```
if (a || b)
   c = 1;
else
   c = 0;
```

Nesse exemplo, para que algum teste possa detectar diferença entre os resultados produzidos pelo código original e pelo código modificado pelo operador de mutação, as seguintes condições devem ser satisfeitas:

a) o teste de entrada de dados deverá causar diferentes estados no programa para o mutante e o programa original; por exemplo, um teste com entradas *a = 1* e *b = 0* faria isso; retornando *c=0* para o código original e *c=1* para o código mutante;
b) o valor de *c* deve ser propagado para a saída do programa e checado pelo teste.

Cada vez que um operador de mutação é aplicado num programa *P*, um novo mutante *m* é gerado. Assim, um conjunto de operadores de mutação *R* aplicado em um programa *P* gera um conjunto de códigos mutantes *m1*, *m2*, ..., *mn*, onde $n \geq 1$. Uma observação importante é que cada mutante possui apenas uma modificação sintática em relação ao programa *P*.

2.4.5 Testes no Código Mutante

Depois de criado os mutantes, eles são testados com o mesmo conjunto de dados de teste *T* de entrada. Se um dado de teste *t* aplicado em um mutante m_i produzir um resultado diferente do obtido quando aplicado no programa original *P*, temos que esse dado de teste *t* foi capaz de distinguir o mutante do programa original. Nesse caso, o código mutante é considerado Morto, pois os testes conseguiram identificar o defeito inserido no mutante, mostrando a diferença entre *P* e m_i.

Um mutante é dito morto se pelo menos um caso de teste produzir resultados diferentes para o código mutante e o código original, ou seja, o conjunto de teste foi capaz de identificar que o mutante gerou um resultado diferente do esperado.

No entanto, nem sempre o conjunto de dados de teste *T* consegue diferenciar um programa *P* de um mutante *m*. Caso os dados de testes *t* apresentem resultantes semelhantes para o mutante e o programa original, o mutante é dito vivo.

Se nenhum caso de teste pôde matar o mutante, temos que isto ocorreu por um de dois motivos: (i) ambos os programas possuem as mesmas funções e, nesse caso, nenhum dado de teste conseguirá mostrar a diferença de *P* e *m*, ou seja, o mutante é Equivalente, ou (ii) o conjunto de dados de testes *T* não foi capaz de distinguir *P* de m_i, ou seja, os testes não foram capazes de detectar o defeito inserido. Nesse caso, o mutante vivo é Não-Equivalente, sendo necessária a geração de novos casos de teste ou a correção da ferramenta utilizada, implicando um aumento do conjunto de testes.

A figura 2 ilustra o processo tradicional de aplicação de Teste por Mutação de Código.

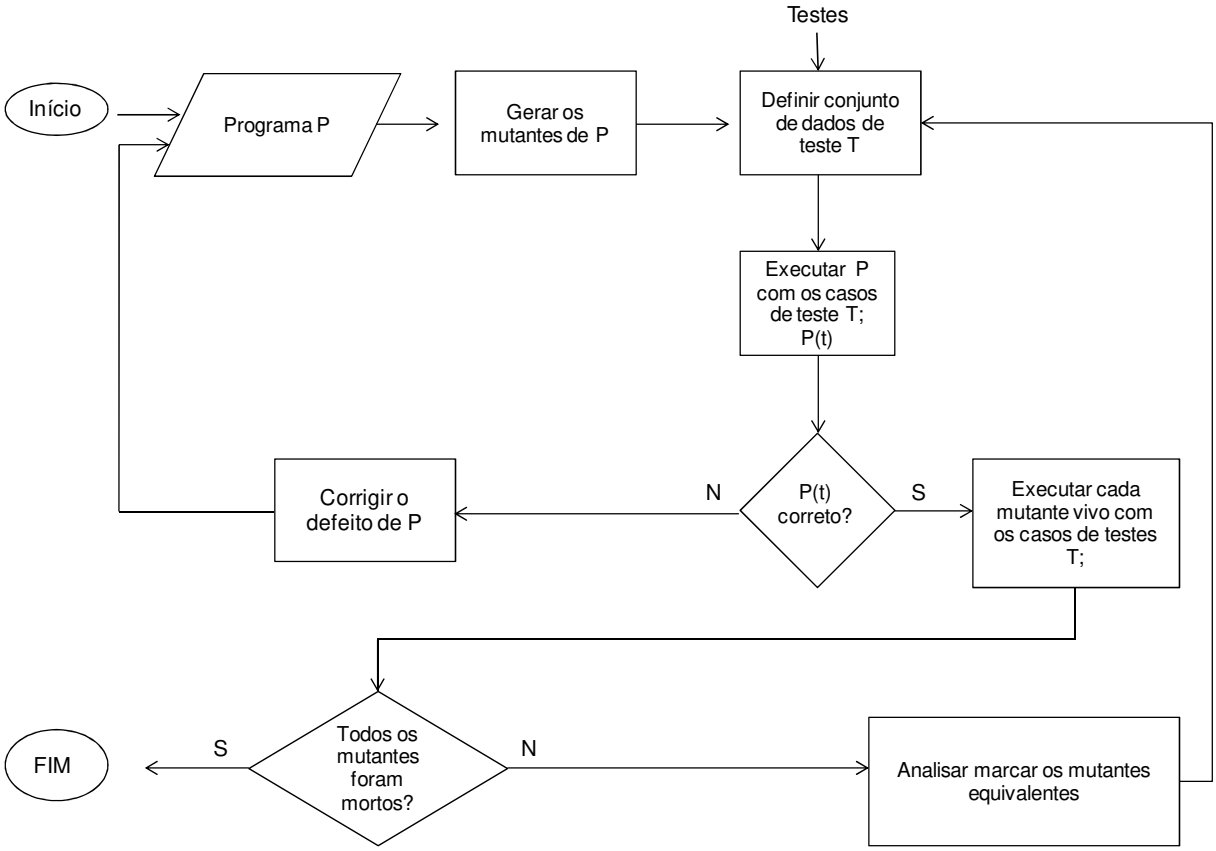

Figura 2 - Processo tradicional de teste por mutação

2.4.6 Mutantes Equivalentes

Muitos operadores de mutação podem produzir Mutantes Equivalentes. Por exemplo, considere o seguinte fragmento de código C++:

```
int index=0;
while (...)
{    . . .;
     index++;
     if (index==10)
          break;
}
```

O operador de mutação substituirá "==" por "> =", produzindo o seguinte mutante:

```
int index=0;
while (...)
{    . . .;
     index++;
     if (index>=10)
          break;
}
```

Nesse exemplo, não é possível encontrar qual caso de teste poderia obter resultados diferentes, portanto o programa resultante é equivalente ao original.

2.4.7 Indicador de Cobertura dos Testes

O objetivo do critério de teste por mutação é encontrar um conjunto de dados de teste *T* que consiga matar o maior número possível de Mutantes m_i Não Equivalentes, tornando assim o conjunto de dados de teste *T* adequado para o teste do programa *P*.

A adequação de um conjunto de teste pode ser medida pelo índice de mutação (MS – Mutation Score), que é dado pela razão entre o número de mutantes mortos e o número total de mutantes Não-Equivalentes (Shahriar e Zulkernine, 2008). O cálculo é feito da seguinte forma:

$$MS(P,T) = \frac{K}{M - E}$$

Onde:

P = programa testado

T = conjunto de dados teste definidos para P

K = quantidade de Mutantes Mortos

M = quantidade de Mutantes Gerados

E = quantidade de Mutantes Equivalentes

O índice de mutação pode variar de 0 a 1. Quanto maior for o seu valor, mais adequado será o conjunto de dados de teste utilizados para matar os mutantes não equivalentes gerados.

2.4.8 Critérios de Mutação de Código

Segundo Offutt (1991), podemos classificar os Testes por Mutação ou Análise de Mutantes em Teste por Mutação Fraca e Testes por Mutação Forte.

Considere o programa *P* constituído de vários componentes. *C* é um simples componente de *P*, *C'* é uma versão mutante de *C* e *m* é a versão mutante de *P* contendo *C'*. Esse componente pode ser uma expressão aritmética, uma expressão booleana, uma referência a uma variável ou uma atribuição de variável etc.

O critério definido como **Mutação Fraca** requer que um dado de teste *t* pertencente a *T* seja aplicado no componente modificado *C'* e em seu correspondente não modificado *C*, produzindo resultados diferentes. No entanto, a comparação dos resultados deve acontecer imediatamente após a execução do componente em *P* e em *m*.

Um dos maiores problemas da mutação fraca é que mesmo que o componente *C'* produza um resultado diferente de *C*, ao final da execução do programa, *m* pode produzir os mesmos resultados que *P*. Sendo assim, o conjunto de dados de teste *T* adequado à mutação fraca pode não ser adequado à análise de mutantes tradicional.

O critério definido como **Mutação Forte** considera a avaliação dos resultados ocorridos no final da execução dos programas *P* e *m*. Esse critério requer que a aplicação de um dado de teste *t* em *P* e em *m*, produza resultados diferente na saída dos programas. A mutação forte requer testes mais rigorosos que a mutação fraca.

Testes por mutação fraca exigem que apenas uma condição seja satisfeita. Testes de mutação forte exigem que ambas as condições sejam satisfeitas. A Mutação fraca está mais próxima dos métodos de detecção de vulnerabilidades através de varredura de código fonte, os chamados testes de caixa branca. Consequentemente, pode-se dizer que a análise pelo critério de mutação forte é mais robusta, pois assegura que o conjunto de testes pode realmente identificar os problemas.

2.5 TESTES DE SEGURANÇA

Os testes de segurança de software têm o objetivo de encontrar vulnerabilidades que possam comprometer a segurança das informações manipuladas pelo software. Eles são essenciais para confirmar se os requisitos de segurança do software foram garantidos mesmo quando esses requisitos não fazem parte das especificações funcionais.

Os testes de segurança se diferenciam dos testes "tradicionais", os quais buscam avaliar apenas as funcionalidades do software. Os testes de segurança buscam garantir a Confidencialidade, Integridade e Disponibilidade das informações que serão manipuladas pelo software, bem como um controle de acesso eficaz e o não repúdio. Para isso, os testes devem abranger tanto o próprio software como a infraestrutura que o suporta. No entanto, o Gartner estima que 75% dos ataques em aplicações web ocorrem ao nível da camada de aplicação (Gartner *apud* Ruby on Rails Security Guide, 2009). Dessa forma, os testes de segurança em software devem acompanhar cada vez mais o ciclo de vida das aplicações web.

Assim como os testes tradicionais de softwares, os testes de segurança podem ser realizados através de várias abordagens.

A abordagem do tipo **Caixa Branca** parte do pressuposto de que o testador possui amplo conhecimento do software e acesso ao código fonte da aplicação (Huang et al., 2004). Ela permite um teste completo de todas as características do software, no entanto, o teste pode-se tornar inviável devido ao grande número de linhas de código, sendo necessário o uso de alguma ferramenta automática.

A abordagem do tipo **Caixa Preta** parte do pressuposto que o testador possui total desconhecimento de como a aplicação foi desenvolvida, e de que ele tenha os mesmos conhecimentos e privilégios de um usuário comum. Os testes são realizados sem acesso ao código fonte, possibilitando executar os testes em tempo de execução. No entanto, esse tipo de teste exige que o testador tenha conhecimentos avançados em segurança de software, podendo ser necessário até mesmo que ele faça uma engenharia reversa no software. Apesar de a abordagem Caixa Preta apresentar um maior espectro de aplicabilidade, normalmente é bem menos eficiente que a abordagem Caixa Branca, dada a natural limitação quanto ao número de elementos analisáveis.

A abordagem do tipo **Fuzzing** é uma técnica que consiste em submeter a aplicação a vários tipos de entrada de dados de uma forma estruturada, em busca de vulnerabilidades causadas por entradas indevidas.

Os testes de segurança podem ser realizados de forma manual ou através de ferramentas que automatizam os testes.

A técnica de testes de segurança com abordagem **manual** é um dos métodos mais antigos utilizados para descobrir vulnerabilidades em aplicações. Eles são realizados geralmente por especialistas e consiste basicamente em encontrar e explorar problemas de segurança em aplicações web.

As ferramentas **automatizadas** são capazes de cruzar diversas informações do ambiente, analisá-las e testar as aplicações de uma maneira mais eficiente do que o teste de invasão manual.

Tanto os testes manuais quanto os automatizados são métodos importantes para identificação de vulnerabilidades em aplicações web. Cada método tem suas forças e fraquezas inerentes e ambos podem ser úteis para descobrir vulnerabilidades críticas de segurança.

Embora as ferramentas automatizadas e testadores qualificados possam navegar em uma aplicação web, somente o testador é capaz de compreender a lógica por trás do fluxo de trabalho da aplicação. Esse entendimento permite ao teste manual subverter a lógica do processo. Por exemplo, um aplicativo pode direcionar o usuário a partir do ponto A ao ponto B e posteriormente ao ponto C, onde o ponto B é uma validação de segurança. A revisão manual do aplicativo pode detectar que é possível ir diretamente ao ponto C, a partir do ponto A, ignorando totalmente a validação de segurança.

Como os aplicativos web continuam a crescer em tamanho, o uso apenas de testes manuais está ficando cada vez mais difícil. Em muitas organizações ele se torna praticamente impossível de ser aplicado isoladamente, pois seria necessária muita dedicação de tempo, esforço físico e dinheiro.

As ferramentas automatizadas não devem substituir completamente os testes manuais. No entanto, se usadas corretamente, estas ferramentas podem ser usadas para encontrar uma ampla gama de vulnerabilidades, reduzindo o esforço de teste. O ideal é utilizar uma combinação adequada de ferramentas automatizadas e ensaios de invasão manual para proporcionar uma melhor segurança do aplicativo web.

2.6 PRINCIPAIS VULNERABILIDADES EM APLICAÇÕES WEB

As vulnerabilidades são falhas na implementação ou configuração de um sistema computacional, que quando exploradas por um atacante pode gerar algum dano ou prejuízo. A Fundação OWASP (Open Web Application Security Project) levantou as dez vulnerabilidades mais críticas encontradas em aplicações web (OWASP Top Ten, 2007).

Apresentaremos três dessas vulnerabilidades, por estarem presentes diretamente no escopo da validação experimental, necessitando um melhor entendimento das mesmas. Nas próximas subseções serão descritas as principais características, os objetivos de teste e as formas de identificação e proteção dessas vulnerabilidades.

Apesar das outras vulnerabilidades não estarem presentes diretamente no escopo da validação experimental, instanciada nesta dissertação, elas são importantes para uma validação prática das ferramentas e testes de segurança, por isso elas estão descritas no anexo A desta dissertação.

2.6.1 Falhas de Injeção

Falhas de Injeção, particularmente injeção de comandos SQL, são comuns em aplicações web. Existem muitos tipos de injeção: SQL, LDAP, XPath, XSLT, HTML, XML, comando de sistema operacional e muitas outras (OWASP Top Ten, 2007).

As falhas de Injeção acontecem quando os dados que o usuário fornece de entrada são enviados como parte de um comando ou consulta. Os atacantes confundem o interpretador para que este mesmo execute comandos manipulados, enviando dados modificados. As falhas de Injeção habilitam o atacante a criar, ler, atualizar ou apagar arbitrariamente qualquer dado disponível para a aplicação. No pior cenário, essas falhas permitem ao atacante comprometer completamente a aplicação e os sistemas relacionados, até mesmo dos ambientes contornados por firewall.

Todos os *frameworks* de aplicação web que usem interpretadores ou invoquem outros processos são vulneráveis a ataques por injeção. Isso inclui quaisquer componentes do *framework* que possam usar interpretadores como *back-end*. Caso uma entrada de usuário seja fornecida a um interpretador sem validação ou codificação, a aplicação é vulnerável (OWASP Testing Guide, 2007).

O objetivo da verificação de segurança é averiguar se os dados fornecidos pelo usuário não podem modificar os comandos e *queries* enviadas aos interpretadores invocados para a aplicação.

A verificação de segurança pode ser realizada de forma manual ou automática. Na **abordagem automatizada,** as ferramentas de varredura de vulnerabilidades localizam as falhas de Injeção, particularmente Injeção SQL. Ferramentas de análise estática que localizam o uso de APIs de interpretadores não seguras são úteis, mas frequentemente não podem verificar se uma validação ou codificação está adequada para proteger contra esse tipo de ameaça. Por exemplo, caso a aplicação gerencie erros internos de servidor de banco de dados detalhados, isso pode impedir a detecção da vulnerabilidade por meio das ferramentas automatizadas, no entanto o código ainda continuará em risco. As ferramentas automatizadas são capazes de detectar também injeções LDAP, XML e XPath.

As **abordagens manuais** são feitas através da verificação do código que invoca os interpretadores. Elas são mais precisas e eficientes. O revisor deve verificar o uso de uma API segura ou verificar se há uma validação e/ou codificação apropriada. O teste pode ser extremamente demorado, com baixa cobertura, devido ao fato de a superfície de ataque, na maioria das aplicações, ser extensa.

A proteção para essa vulnerabilidade consiste em evitar o uso de interpretadores, quando possível. Caso invoque um interpretador, o método chave para evitar injeções está no uso de APIs seguras, como por exemplo, *queries* parametrizadas e bibliotecas de mapeamento de objeto relacional (ORM).

Essas interfaces tratam todas as entradas de dados, mesmo aquelas que não demandam tratamento. Note que enquanto interfaces seguras resolvem o problema, a validação é ainda recomendada para conter ataques (OWASP Top Ten, 2007).

2.6.2 Cross Site Scripting

O Cross Site Scripting, mais conhecido como XSS, é de fato um subconjunto de injeções de código HTML em uma aplicação web. As vulnerabilidades de XSS ocorrem em quaisquer aplicações que recebam dados originados do usuário e os envie ao navegador do usuário sem primeiramente validar ou tratar aquele conteúdo. O XSS é uma das questões de segurança em aplicações web mais prevalente e perniciosa (OWASP Top Ten, 2007).

O XSS permite que atacantes executem scripts maliciosos (malware) no navegador da vítima, podendo sequestrar sessões de usuários, desfigurar web sites, inserir conteúdo hostil, conduzir ataques de roubo de informações pessoais (*phishing*) e obter o controle do navegador do usuário. O script malicioso é frequentemente codificado em Java Script, mas qualquer linguagem de script suportada pelo navegador da vítima é um alvo potencial para esse ataque. O uso do Java Script habilita o atacante a manipular aspectos de uma página a ser apresentada, incluindo a adição de novos elementos (como um espaço para *login* que encaminha credenciais para um site hostil), a manipulação de qualquer aspecto interno do DOM (*Document Object Model*) e a remoção ou modificação da forma de apresentação da página. Os *frameworks* de aplicações web são vulneráveis a XSS.

Em uma verificação de segurança, o objetivo é averiguar se todos os parâmetros da aplicação são validados e/ou recodificados antes de serem incluídos em páginas HTML.

As verificações de segurança com **abordagens automatizadas** são compostas de ferramentas de teste de penetração, capazes de detectar XSS de reflexão a partir de injeção de parâmetro. A falha na identificação de XSS é frequente, particularmente se a saída do vetor de injeção XSS é tratada através de verificação de autorização (como, por exemplo, se um usuário fornece dados de entrada maliciosos que são vistos posteriormente apenas pelos administradores).

Assim, a maneira mais eficiente de verificação de segurança é verificar o código, principalmente quando houver um mecanismo centralizado de validação e recodificação dos dados de entrada. Se for utilizada uma implementação distribuída, a verificação demandará um esforço adicional considerável, tornando as **abordagens manuais** impraticáveis, visto que a superfície de ataque (ou seja, a quantidade de campos de entrada de dados) da maioria das aplicações é muito grande.

A melhor proteção contra XSS está na combinação de validação de "lista branca" de todos os dados de entrada e recodificação apropriada de todos os dados de entrada. A validação habilita a detecção de ataques e a recodificação previne que qualquer injeção de script seja bem sucedida de ser executada no navegador. A prevenção de XSS ao longo da aplicação como um todo requer uma abordagem arquitetural consistente.

2.6.3 Furo de Autenticação e Gerência de Sessão

Autenticação e gerência de sessões apropriadas são críticas para a segurança na web. Falhas nessa área geralmente envolvem a falha na proteção de credenciais e nos *tokens* da sessão durante seu tempo de vida. Essas falhas podem estar ligadas a roubo de contas de usuários ou administradores, contornando controles de autorização e de responsabilização, causando violações de privacidade (OWASP Top Ten, 2007).

Todos os frameworks de aplicações web estão vulneráveis a furos de autenticação e de gerência de sessão. Furos no mecanismo principal de autenticação não são incomuns, mas as falhas são geralmente introduzidas a partir de funções menos importantes de autenticação como *logout*, gerência de senhas, *timeout*, recordação de dados de *logon*, pergunta secreta e atualização de conta.

O objetivo em um teste de segurança é verificar se a aplicação autentica corretamente os usuários e protege as identidades das credenciais associadas.

Nas verificações de segurança com **abordagem automatizada,** as ferramentas de localização de vulnerabilidade têm dificuldade em esquemas de autenticação e de sessão personalizados, por exemplo, quando não usam mecanismos disponibilizados pela infra-estrutura de um web Server. As ferramentas de análise estáticas provavelmente também não detectarão problemas em códigos personalizados para autenticação e gerência de sessão.

A **abordagem manual** inclui a revisão de código e testes que, especialmente combinados, são muito efetivos para verificar se a autenticação, gerência de sessão e funções secundárias estão todas implementadas corretamente.

A autenticação segura depende da comunicação e do armazenamento das credenciais. Para a proteção contra essa vulnerabilidade, primeiramente deve-se assegurar que o protocolo SSL é a única opção de acesso para todas as partes autenticadas do aplicativo e que todas as credenciais estão guardadas de uma forma encriptada ou em hash.

Prevenir falhas de autenticação requer um planejamento cuidadoso, normalmente feito durante a fase de planejamento de arquitetura das aplicações. Algumas das considerações são importantes (OWASP Top Ten, 2007):

• usar somente mecanismos padrão para gerência de sessão. Não escrever ou usar gerenciadores secundários de sessão em qualquer situação;

• não aceitar novos identificadores de sessão, pré-configurados ou inválidos na URL ou em requisições. Isso é chamado de ataque de sessão fixada;

• limitar e ou limpar o código de *cookies* personalizados com propósito de autenticação de gerência de sessão, como funções 'lembrar do usuário' ou funções domésticas de autenticação centralizadas como o *Single Sign-On* (SSO). Isto não se aplica às soluções de autenticação federadas robustas ou SSO reconhecidas;

• usar um mecanismo único de autenticação com dimensão e número de fatores apropriados. Certificar-se que este mecanismo não estará facilmente sujeito a ataques ou fraudes;

• não permitir que o processo de *login* comece de uma página não encriptada. Sempre iniciar o processo de *login* de uma segunda página encriptada ou de um novo código de sessão, para prevenir o roubo de credenciais ou da sessão, *phishing* e ataques de fixação de sessão;

• considerar a geração de uma nova sessão após uma autenticação que obteve sucesso ou mudança do nível de privilégio;

• assegurar que todas as páginas tenham um link de *logout*. O *logout* deve destruir todas as sessões e *cookies* de sessão;

• usar períodos de expiração de prazo que automaticamente dão *logout* em sessões inativas;

• usar somente funções de proteção secundárias eficientes (perguntas e respostas, *reset* de senha), pois essas credenciais são como senhas, nomes de usuários e *tokens*. Aplicar *one-way hash* nas respostas para prevenir ataques nos quais a informação pode ser descoberta;

• não expor nenhum identificador de sessão ou qualquer parte válida das credenciais em URLs e logs (não regravar ou armazenar informações de senhas de usuários em logs);

• verificar a senha antiga do usuário quando ele desejar mudar a senha;

• não confiar em credenciais falsificáveis como forma de autenticação, como endereços de IP ou máscaras de rede, endereço de DNS ou verificação reversa de DNS, cabeçalhos da origem ou similares;

• cuidar do envio de segredos para endereços de e-mail com um mecanismo de *reset* de password. Usar números randômicos para fazer o reset de acesso e enviar um e-mail de retorno assim que a senha for reconfigurada. Cuidar para que, quando permitir que usuários registrados mudem seus endereços de e-mail, seja enviada uma mensagem para o e-mail anterior antes de efetuar a mudança.

2.7 CONCLUSÃO

Este capítulo apresentou os conceitos básicos dos assuntos abordados nesta dissertação. Foram apresentados os conceitos sobre a representação, geração e estrutura de uma Árvore de Ataque, bem como os conceitos sobre Testes de Software e a técnica de teste por Mutação de Código. Foram descritas as características específicas dos Testes de Segurança, as principais vulnerabilidades que podem ocorrer em aplicações web e suas respectivas formas de verificação.

Foram apresentados também os principais trabalhos encontrados na literatura científica das diversas áreas relacionadas, e que serviram de base e inspiração para o desenvolvimento da metodologia que será proposta no capítulo seguinte.

3 METODOLOGIA DE VALIDAÇAO DE TESTES

3.1 INTRODUÇÃO

Este capítulo descreve a proposta de metodologia para a validação de ferramentas e testes de segurança em aplicações web, apresentando as características das etapas do fluxo de validação.

Em seguida são apresentados os detalhes da metodologia para a geração da árvore de ataque, a qual será utilizada para representar o levantamento e a lista das vulnerabilidades e as possíveis formas de ataque a uma aplicação. São apresentados também os detalhes da metodologia utilizada na geração do código mutante, que é uma das etapas mais importantes do processo de validação dos testes e ferramentas selecionados para verificação.

3.2 METODOLOGIA PROPOSTA

A metodologia proposta neste trabalho para a validação de ferramentas de testes de segurança para aplicações web envolve: (i) a criação da árvore de ataque com as principais vulnerabilidades em aplicações web, (ii) a definição e escolha das ferramentas de testes que serão utilizadas para cobrir as vulnerabilidades derivados das árvores de ataque, (iii) a execução dos testes e (iv) a validação dos testes através de técnicas de mutação de código. A figura 3 apresenta as etapas do fluxo de validação proposto pela metodologia.

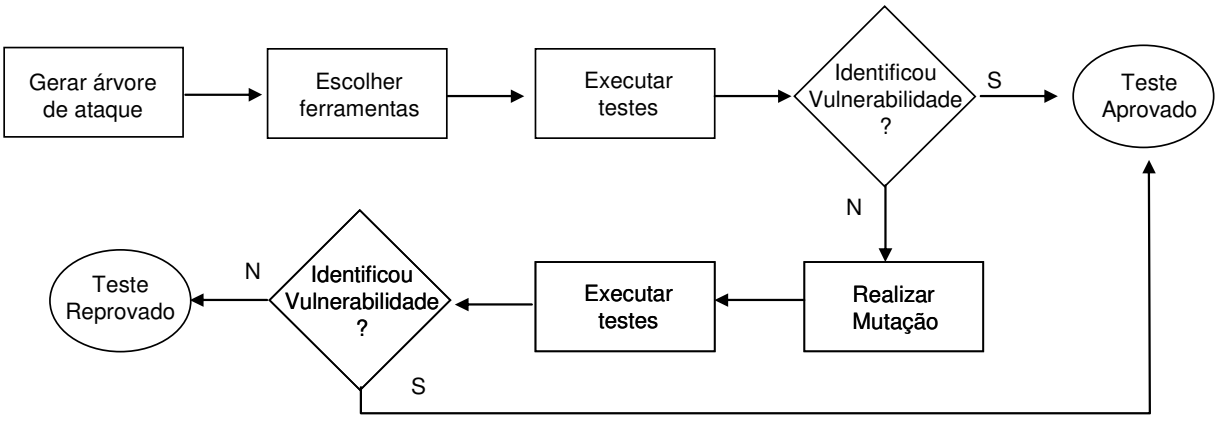

Figura 3 - Fluxo de Validação dos Testes de Segurança

Segundo essa metodologia, gera-se inicialmente uma árvore de ataque específica para um escopo de aplicações. Esse escopo é baseado: no objetivo de ataque, na linguagem empregada na programação, no banco de dados utilizado, na infraestrutura necessária para suportar a execução, no ambiente de produção etc. A árvore gerada para esse escopo é derivada de uma árvore de ataque genérica para aplicações web e está associada aos objetivos de ataque e às vulnerabilidades e possibilidades de ataques referentes ao escopo da aplicação. Essa árvore de ataque consiste, de fato, em um roteiro para a realização de testes de vulnerabilidades, baseado nas vulnerabilidades potenciais dessa aplicação. Basicamente, as árvores de ataque são utilizadas para representar o levantamento e a lista das vulnerabilidades e formas de ataque às aplicações web.

Em seguida, considerando de um lado o roteiro gerado pela árvore de ataque, e por outro, as ferramentas de testes de segurança disponíveis, é efetuada a escolha das ferramentas de teste mais apropriadas. As ferramentas são escolhidas de forma a cobrir todas as vulnerabilidades descritas pela árvore.

Após a escolha das ferramentas (ou mesmo scripts de testes) que serão utilizados, os testes são executados na aplicação original.

Caso sejam encontradas vulnerabilidades na aplicação testada, obtém-se um indicativo de que os testes de vulnerabilidades cobrem as vulnerabilidades para as quais se propõem. Em caso contrário, ou seja, se após a execução desses testes não forem encontradas as vulnerabilidades listadas pela árvore de ataque do escopo da aplicação, é necessário efetuar a validação dos testes executados, inserindo intencionalmente vulnerabilidades no código fonte da aplicação e executando os testes novamente.

A inserção de vulnerabilidades no código fonte é feita através da técnica de mutação de código. Após a geração do código mutante, executam-se novamente os testes selecionados no código modificado, de forma a verificar se as vulnerabilidades incluídas propositalmente pela mutação de código serão agora detectadas pelos testes de segurança.

Caso os testes identifiquem as vulnerabilidades inseridas, tem-se um indicativo de que as ferramentas de teste de segurança empregadas na cobertura das vulnerabilidades representadas pela árvore de ataque estão aptas a serem utilizados. Caso os testes não identifiquem as vulnerabilidades incluídas pela mutação de código, temos um indicativo de que as ferramentas não cobrem adequadamente as técnicas de ataque mapeadas pela árvore de ataque do escopo da aplicação

3.2.1 Geração da Árvore de Ataque

A elaboração da árvore de ataque para um escopo de aplicações é composta de duas etapas: i) geração de uma árvore de ataque genérica para aplicações web, ii) e adequação dessa árvore aos objetivos específicos da validação, através da redução dos nós e caminhos que não são necessários ou estão fora do escopo da validação.

A árvore de ataque genérica auxiliará a elaboração da árvore de ataque específica para o escopo da aplicação que será utilizada na validação dos testes de segurança.

3.2.1.1 Semântica adotada

A semântica adotada segue a proposta de Schneier (1999), descrita e exemplificada na subseção 2.2.3 desta dissertação.

3.2.1.2 Preenchimento

Para geração da árvore, inicialmente são identificados os ativos mais importantes da aplicação e, em seguida, é efetuada uma análise crítica sobre eles. Assim, buscam-se definir quais são os ativos essenciais e quais são as principais interações envolvidas. Um exemplo prático na seleção de ativos é analisar os processos de negócio e verificar os serviços fornecidos pelo sistema.

Algumas perguntas auxiliam na definição dos ativos que serão os objetivos de ataque, por exemplo: (i) quais são as principais preocupações dos clientes que vão utilizar o serviço e o que os deixa confiantes em relação à segurança? (ii) quais são os principais ativos e informações do sistema? (iii) quais ativos são cruciais para que o sistema possa oferecer o serviço proposto?

Os ativos identificados nas respostas dessas perguntas serão os objetivos de ataques dos testes, (o que?). Para cada objetivo de ataque selecionado é necessário identificar também quais as técnicas de ataques podem ser utilizadas para alcançá-los (onde?) e quais as formas como eles podem ser atingidos (como?).

Na árvore de ataque de um ativo escolhido, a raiz da árvore, representa o objetivo de ataque (o que?), identificados como mais importantes para uma aplicação. As folhas da árvore representam as vulnerabilidades que podem ser exploradas para alcançar o objetivo de ataque, bem como as possíveis formas de ataque. Elas são preenchidas com base nas principais

vulnerabilidades divulgadas pelas comunidades de segurança em aplicações web. Uma fonte importante a ser considerada é o OWASP Top Ten (2007) e o OWASP Testing Guide (2007).

As árvores são desenhadas de cima para baixo, começando pelo objetivo de ataque selecionado. Os nós que representam os sub- objetivos são colocados hierarquicamente sob o nó superior, e conectados com linhas para o nó anterior, com a descrição do evento, em uma cadeia de acontecimentos que conduzem a um ataque. Dessa forma, nós folhas ligados em uma cadeia até o objetivo de ataque de mais alto nível representam cadeias de eventos que conduzem a um possível ataque.

Quando dois ou mais nós estão associados a um nó ascendente na árvore, dois procedimentos lógicos diferentes podem ocorrer.

O primeiro procedimento lógico ocorre quando dois ou mais nós do tipo folhas têm links que levam ao mesmo nó, um nível acima na hierarquia. Nesse caso, aplica-se entre os elementos o operador lógico OR. Por exemplo, na árvore de ataque da figura 4, “Acessar o banco de dados diretamente” ou “Acessar a aplicação como usuário privilegiado” resultam de “Acessar os dados da aplicação”.

O segundo procedimento lógico ocorre quando dois ou mais nós que não são do tipo folha têm links que levam ao mesmo nó, um nível acima na hierarquia, e um arco liga as duas linhas com uma notificação "AND". Nesse caso, aplica-se entre os elementos o operador lógico AND; por exemplo, na árvore de ataque da figura 4, tanto “Manter um ID válido” como “Modificar o ID do *cookie* do usuário” precisam ser realizados para “Fixar uma sessão”, caso tenha-se optado por seguir esse caminho de ataque.

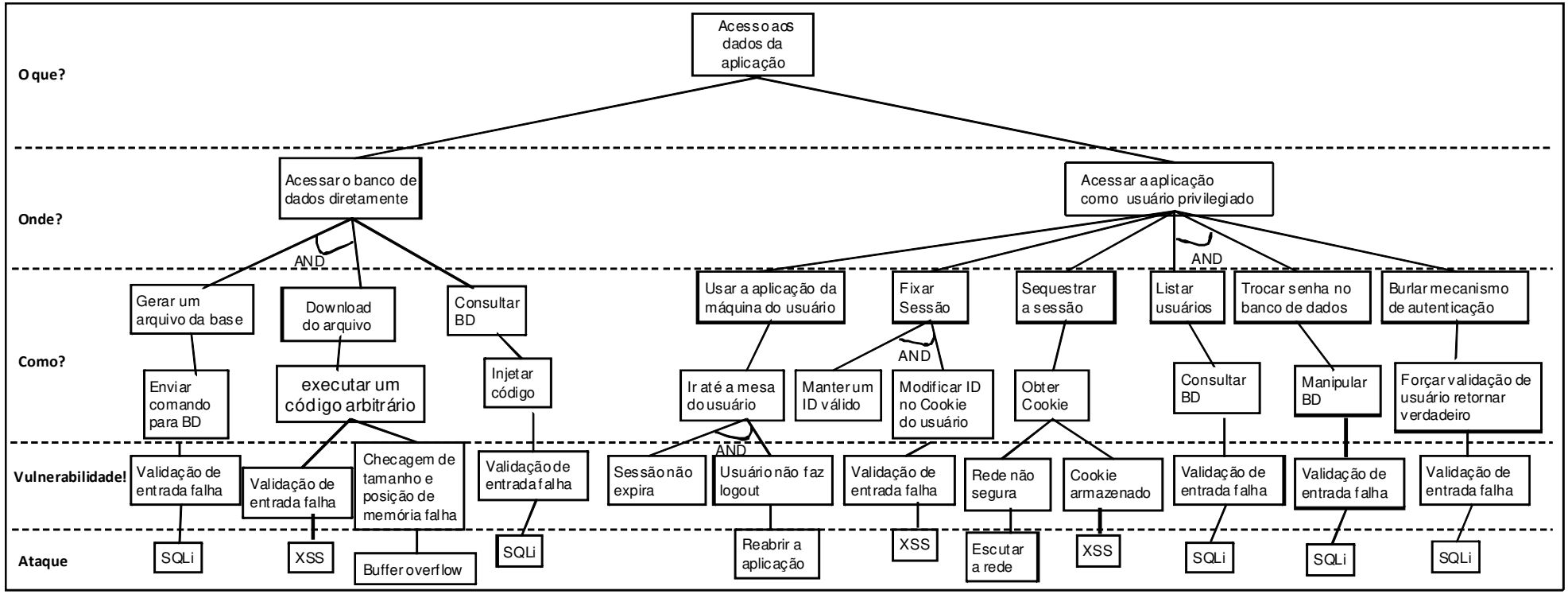

Figura 4 - Árvore de ataque genérica para o ativo "Dados da aplicação"

Através de uma árvore, é possível também realizar a análise de um sistema ou sub-sistema em situações e ambientes específicos. Isso torna possível dividir os ataques descritos em um conjunto de possíveis caminhos, tornando um objetivo de ataque de alto nível em vários sub-ataques, que são direcionados aos seus sub-objetivos. Dessa forma, a geração da árvore de ataque genérica fornecerá uma ferramenta estrutural que poderá ser utilizada para descobrir eventuais vulnerabilidades em uma aplicação específica.

A possibilidade de projetar e analisar as árvores de ataque em diferentes níveis de profundidade e em áreas selecionadas de interesse e criticidade permite a escolha dos testes que serão executados e a definição de quais caminhos serão seguidos para alcançar o objetivo de ataque. Cada aplicação específica que utilizar a árvore genérica como base, deverá considerar o escopo da aplicação na seleção das alternativas de caminhos da árvore para alcançar o objetivo de ataque.

3.2.2 Escolha das Ferramentas de Teste

A escolha das ferramentas que serão utilizadas nos testes é feita a partir da análise das vulnerabilidades listadas na árvore de ataque específica do escopo da validação. Após listar as vulnerabilidades citadas e as possíveis formas de ataque, selecionam-se as ferramentas e os possíveis scripts maliciosos que se propõe a detectar essas vulnerabilidades.

Algumas ferramentas possuem parâmetros e itens de configuração que precisam ser ajustados de acordo com o contexto. Elas devem ser configuradas conforme as orientações dos fabricantes e devem ser as mesmas tanto nos testes executados no código original como no código mutante.

Após a escolha e configuração das ferramentas são executados os testes no código original. Se os testes identificarem no código original as vulnerabilidades pesquisadas, temos um indicativo de que o teste é valido para identificá-las dentro escopo definido. Caso não seja identificado as vulnerabilidades pesquisadas é preciso validar a eficácia dos testes. Essa validação é feita através da mutação de código.

3.2.3 Execução dos Testes

Os testes para detectar as vulnerabilidades podem ser feitos manualmente ou por ferramentas que se propõe a testar os tipos de vulnerabilidades específicos. Se efetuadas por

ferramentas, a validação dos testes de vulnerabilidades servirá também como validação dessas ferramentas de testes.

A aplicação dos testes manuais consiste na execução da aplicação simulando tanto um usuário que não possui credenciais de acesso, como usuários comuns que possuem apenas privilégios básicos de um usuário padrão, ou seja, sem os privilégios de administrador. Para ambos os casos são utilizadas entradas maliciosas que tentam burlar os mecanismos de validação ou da lógica da aplicação.

A aplicação dos testes automáticos consiste na execução exaustiva das técnicas utilizadas nos testes manuais, através de ferramentas automatizadas. Essas ferramentas são capazes de analisar as informações da aplicação e testá-la de um modo mais eficiente do que os testes manuais.

Após a escolha das ferramentas e/ou scripts que serão utilizados, os testes são executados na aplicação original.

3.2.4 Geração dos Mutantes

Utilizando-se da técnica de testes por análise de mutantes, ou mutação de código, pode-se validar os testes de segurança que foram executados no código original e que não conseguiram identificar as possíveis vulnerabilidades pesquisadas.

A validação inicia-se com o mapeamento das vulnerabilidades definidas na árvore de ataque e em seguida com uma pesquisa em busca de operadores de mutação que irão realizar a inserção dessas vulnerabilidades no código. Para cada vulnerabilidade ou um conjunto delas, teremos um ou mais operadores de mutação.

As regras dos operadores de mutação indicam as mudanças que devem ser realizadas no código original para inserir uma determinada falha. No entanto, os geradores automáticos de mutação de código conhecidos possuem operadores de mutação bastante simples, como por exemplo, trocar ‘&&’ por ‘||’ ou ‘==’ por ‘>=’. Esses tipos de operadores dificilmente vão incluir uma vulnerabilidade de segurança em uma aplicação.

Para que os operadores de mutação gerados realmente incluam vulnerabilidades de segurança no código, é preciso pesquisar pelas vulnerabilidades conhecidas e divulgadas pelas comunidades de segurança afins, bem como a forma como elas são corrigidas. Dessa forma é possível gerar operadores de mutação capazes de localizar e modificar trechos no código

original que possuem as características de códigos escritos de forma correta, de modo a torná-los vulneráveis.

Isso torna esses operadores fortemente dependentes do contexto da aplicação. Espera-se, em longo prazo, que da mesma forma que existem hoje comunidades e sites que publicam as principais vulnerabilidades em aplicações web, também possa ser criado comunidades e sites que divulguem operadores de mutação provocadores das principais vulnerabilidades.

Um dos grandes problemas dos testes por mutação é saber quando um mutante é equivalente (não altera o funcionamento de um código). Segundo Offutt (1994), a detecção de mutantes equivalentes é um dos maiores obstáculos para a prática de utilização de testes por mutação. O esforço necessário para verificar se os mutantes são equivalentes pode ser muito elevado, mesmo para pequenos programas.

Este dissertação assume o uso de operadores de mutação que gerem apenas mutantes não equivalentes, ou seja, que produzem vulnerabilidades reais, eliminando a necessidade de análise dos mutantes para identificar os equivalentes.

3.2.5 Execução dos Testes nos códigos Mutantes

Após a geração dos códigos mutantes, cada um deles é testado com o mesmo conjunto de ferramentas de testes a que o código original foi submetido.

Se algum dos testes produzir um resultado diferente do resultado no código original, temos que este teste foi capaz de distinguir o mutante do código original, ou seja, foi capaz de identificar a vulnerabilidade inserida. Neste caso, a ferramenta pode ser validada como eficaz para o escopo definido.

Se nenhum dos testes pôde matar o mutante, ou seja, se não conseguiram identificar a vulnerabilidade inserida, o mutante é dito vivo, indicando que o teste executado pela ferramenta (ou pelo script) não é adequado para identificar a vulnerabilidade mapeada naquele escopo.

3.3 CONCLUSÃO

Este capítulo apresentou uma proposta de metodologia para validação de ferramentas e testes de segurança em aplicações web, baseada na i) geração de uma árvore de ataque para

representação dos testes necessários para validação de segurança da aplicação e ii) em mutações de código com inserção intencional de vulnerabilidades para checagem da eficácia dos testes.

A metodologia proposta será validada através de uma validação experimental, empregando o seu uso na verificação dos testes de segurança em uma aplicação web, e será apresentada no próximo capítulo.

4 VALIDAÇÃO EXPERIMENTAL

4.1 INTRODUÇÃO

Neste capítulo é apresentada a validação experimental da metodologia proposta no capítulo 3. A validação é feita através da aplicação prática da metodologia na verificação dos testes de segurança de uma aplicação web. É apresentada uma breve descrição da aplicação que será alvo dos testes de segurança. Em seguida é elaborada a árvore de ataque específica para essa aplicação, gerada a partir da árvore genérica apresentada no capítulo anterior e a partir do levantamento das vulnerabilidades específicas do escopo da aplicação. É apresentado o resultado da aplicação dos testes no código original e nas versões mutantes. E por fim é feita uma análise dos resultados obtidos.

4.2 DESCRIÇÃO DO CASO TURMA CIDADÃ

Para validação experimental da metodologia proposta, escolhemos uma aplicação web como estudo de caso. A aplicação *Engajar* faz parte do projeto *Turma Cidadã,* patrocinado pelo CEFET-RJ, e foi construída por desenvolvedores voluntários da Petrobras. O sistema implementa o controle e gerenciamento de turmas de treinamento ministradas por instrutores voluntários. Um usuário disposto a ministrar um curso como instrutor voluntário pode-se cadastrar no sistema e registrar suas respectivas áreas de interesse e disponibilidade. Os interessados em participar de um curso também fazem um cadastro e o sistema se encarrega de casar os interesses e montar as turmas. Ela está acessível através do endereço eletrônico http://turmacidadaweb.cefet-rj.br .

A aplicação *Engajar* foi desenvolvida na linguagem de programação *Ruby on Rails*. Para guardar os dados por ela manipulados, é utilizado um sistema de banco de dados MySql. Nas próximas subseções será descrito algumas das vulnerabilidades referentes ao escopo da aplicação *Engajar* e que serão determinantes para a geração da árvore de ataque específica desta aplicação.

4.2.1 Vulnerabilidades existentes no escopo da aplicação

Uma das ameaças mais comuns e devastadoras para aplicações web, que pode prejudicar mesmo aquelas elaboradas na linguagem *Ruby on Rails*, é a injeção, categoria de ataque que introduz códigos maliciosos ou parâmetros em uma aplicação web, de forma a executar esses códigos dentro do contexto de segurança da aplicação (OWASP Ruby Guide, 2009). Os exemplos mais proeminentes de injeção são o *Cross Site Scripting* (XSS) e o *SQL injection*, apresentados na seção 2.6.

Conforme descrito na figura 4, podemos observar na árvore de ataque genérica que a vulnerabilidade de injeção de comandos SQL pode ser utilizada para alcançar vários subobjetivos de ataque. Por este motivo, serão detalhados a seguir alguns ataques referentes a esse tipo de vulnerabilidade e relacionados ao escopo da aplicação.

4.2.1.1 Acesso ao servidor de Banco de dados através de código injetado

Apesar de o *Ruby on Rails* possuir métodos inteligentes, as aplicações construídas nessa linguagem também podem ter problemas com *SQL Injection,* caso os desenvolvedores não tomem certos cuidados (OWASP Ruby Guide, 2009).

Os ataques de injeção de SQL visam alterar as consultas realizadas ao banco de dados, manipulando os parâmetros de uma aplicação web. Um objetivo muito conhecido de ataques de *SQL Injection* é realizar uma manipulação ou leitura arbitrária de dados.

Abaixo temos um exemplo de como não utilizar a entrada de dados de usuário em uma consulta:

```
Project.find(:all, :conditions => "name = '#{params(:name)}'")
```

Provavelmente esse código é de uma pesquisa em que o usuário pode digitar o nome do projeto que pretende encontrar. Se um usuário malicioso digitar `'OR 1 = 1 --`, a consulta SQL resultante será:

```
SELECT * FROM projects WHERE name = ' ' OR 1=1 --'
```

Os dois traços iniciam um comentário, ignorando todos os comandos que viriam após o código injetado. Portanto, a consulta retorna todos os registros da tabela “projetos”, incluindo os ocultos para o usuário. Isso porque a condição é válida para todos os registros.

4.2.1.2 Burlando o mecanismo de autorização

Outro objetivo de ataque muito conhecido é burlar os mecanismos de autorização. Normalmente, uma aplicação web possui um controle de acesso. O usuário digita suas credenciais de *login* e a aplicações tentam localizar o correspondente registro na tabela usuários. A aplicação concede o acesso quando ela encontra o registro. No entanto, um invasor pode burlar esta verificação com a injeção de comando SQL (OWASP Ruby Guide, 2009). O código abaixo, típico de aplicações Rails, mostra uma consulta à base de dados para encontrar o primeiro registro na tabela usuários que coincide com as credenciais de *logins* fornecidas pelo usuário.

```
User.find(:first, "login = '#{params(:name)}' AND password =
'#{params(:password)}'")
```

Se um usuário mal intencionado entrar com `'OR'1' ='1` no campo *nome* e `'OR'2'>'1` no campo *senha*, a consulta que a aplicação enviará para o banco ficaria assim:

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR
'2' > '1' LIMIT 1
```

Esta consulta localiza o primeiro registro no banco de dados, concedendo o acesso a este usuário.

4.2.1.3 Usando campo limit ou offset

O método `User.find()` obtém a lista de usuários, começando por um *`offset`* e limitado a um número fixo de entradas. Por exemplo:

```
User.find (: all,: limit => 10,: offset => 5)
```

Essa função utiliza uma consulta SQL semelhante a:

*"select * from user limit 10 offset 5".*

Num contexto web, os parâmetros *`:limit`* e *`:offset`* geralmente vem da url. No entanto, os valores de *`:limit`* e *`:offset`* não são checados. Um atacante pode usar, por exemplo,

```
User.find (:all, :limit => "10; SQL_query;", : offset => 5)
```

para executar um *"SELECT * FROM USER LIMIT 10; SQL query; OFFSET 5".*

A partir daí, pode-se então utilizar os parâmetros *`:limit`* ou *`:offset`* para injetar uma consulta SQL através do *Ruby on Rails*.

4.3 GERAÇÃO DA ÁRVORE DE ATAQUE

Para geração da árvore de ataque específica da aplicação que será utilizada como estudo de caso na validação dos testes de segurança, o ativo considerado crítico selecionado foi os “Dados Cadastrais dos Interessados”. Esses dados são armazenados no banco de dados da aplicação e foram considerados críticos porque são informações pessoais dos usuários do sistema, e estes não autorizaram a divulgação dos mesmos. Esse ativo será o objeto de ataque da árvore.

A partir do ativo crítico selecionado, definimos um objetivo de ataque para esse ativo e iniciamos o trabalho de listar as formas de como seria possível alcançá-lo. O objetivo é acessar a base de cadastro dos interessados.

Baseado na árvore de ataque genérica para aplicações web, nas vulnerabilidades conhecidas da linguagem *Ruby on Rails* e nas características referentes ao banco de dados, levantamos as possíveis vulnerabilidades e as suas formas de serem exploradas, listando os respectivos ataques. Para cada forma diferente, um nó folha no subnível abaixo foi criado. A árvore de ataque do escopo de validação das ferramentas de testes deste estudo de caso foi resumida à árvore da figura 5.

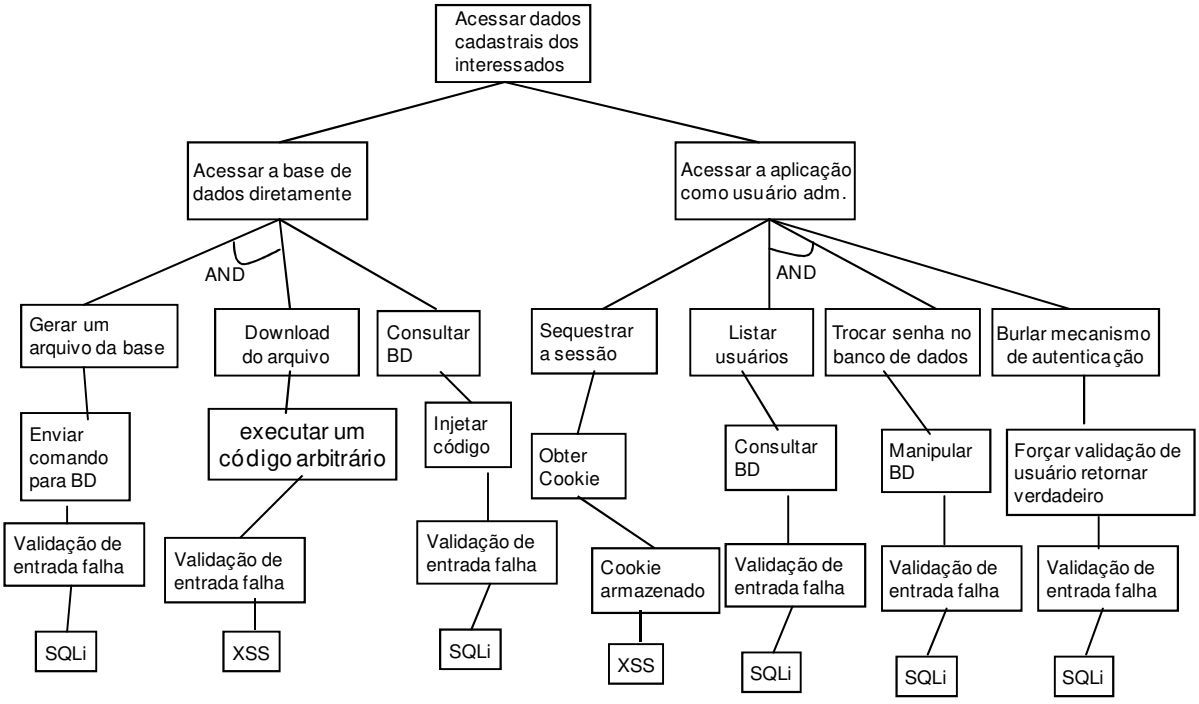

Figura 5 - Árvore de ataque do ativo "Dados cadastrais dos interessados”

.

4.4 ESCOLHA DAS FERRAMENTAS

Considerando, de um lado, o roteiro gerado pela árvore de ataque, e de outro, as vulnerabilidades cobertas e declaradas pelas ferramentas de testes de segurança disponíveis, foi efetuada a escolha das ferramentas de testes que serão utilizadas.

Para cobrirmos os ramos da árvore de ataque, escolhemos três ferramentas para testes de segurança. Elas realizam testes que cobrem as vulnerabilidades de injeção de dados, por exemplo, ataques de SQL *Injection* e XSS.

Duas das ferramentas escolhidas são de uso livre. A ferramenta Pblind se propõe a testar vulnerabilidades de SQL *Injection*. Ela faz parte de um "Kit" de ferramentas para teste de invasão de uso livre, chamado BackTrack 4. A segunda ferramenta, chamada w3af, também de uso livre, é integrante do "Kit" de ferramentas da fundação OWASP (Owasp Live CD, 2009), e se propõe a testar as principais vulnerabilidades em aplicações web, inclusive as de SQL *Injection* e XSS. A terceira ferramenta, chamada NStalker, é uma ferramenta comercial que se propõe a testar tanto vulnerabilidades de SQL *Injection* e XSS como outras vulnerabilidades em aplicações web e na infra-estrutura que as suportam. As ferramentas foram escolhidos de forma a cobrir todas as vulnerabilidades descritas pela árvore.

Apesar de as ferramentas terem sido instaladas no mesmo equipamento em que está instalado o servidor de aplicação e o banco de dados da aplicação, elas estavam rodando em uma máquina virtual. Dessa forma, os ataques foram realizados simulando uma aplicação disponível na Internet.

As ferramentas foram configuradas para procurar por qualquer tipo de vulnerabilidade, no entanto o escopo deste estudo de caso é a validação dos testes para as vulnerabilidades de injeção de dados, o que não impede o uso da metodologia na validação dos demais testes.

4.5 TESTES INICIAIS

4.5.1 Aplicação dos testes iniciais

Após a definição e configuração das ferramentas e da seleção dos testes manuais que serão executados para tentar identificar as vulnerabilidades descritas na árvore de ataque, iniciamos a aplicação dos testes.

Inicialmente, foi executado cada teste na aplicação original. As ferramentas foram configuradas conforme orientação dos fabricantes. Os testes automatizados foram executados conforme a configuração descrita na seção 4.4.

4.5.2 Resultados dos testes iniciais

A ferramenta de uso comercial, apesar de não ter identificado vulnerabilidades de falhas de injeção, informou haver outras vulnerabilidades fora do escopo dessa validação. As ferramentas de uso livre executaram a varredura e também não identificaram vulnerabilidades relacionadas a falhas de injeção.

Analisando as variáveis da aplicação no momento da execução dos testes, podemos verificar um dos valores utilizados como tentativa de injeção de código, conforme a figura 6.

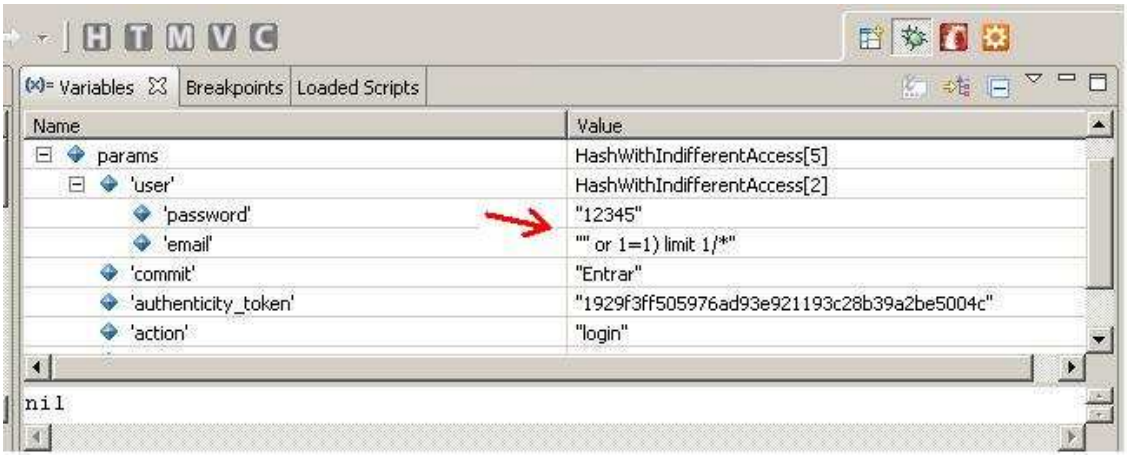

Figura 6 - Variáveis da aplicação em tempo de execução dos testes

Nesse caso de teste, a aplicação gerou a seguinte consulta para o banco de dados:

```
SELECT * FROM `users` WHERE (email = '\" or 1=1) limit 1/*' AND
password_hashed = '8cb2237d0679ca88db6464eac60da96345513964') LIMIT 1
```

A aplicação não aceitou o caractere *aspas* como um comando e sim como um caractere normal, não permitindo a injeção de código seguinte. Dessa forma, se o teste fosse executado manualmente, o resultado dessa tentativa seria uma mensagem da aplicação informando que o usuário não foi encontrado. Provavelmente a aplicação está utilizando

algum recurso da linguagem que filtra caracteres especiais, tratando o conteúdo das variáveis antes de montar as consultas ao banco de dados.

Na próxima fase do nosso estudo de caso tentaremos inibir o uso desses recursos pela aplicação, procurando por essas características no código e modificando o uso desses recursos.

4.6 GERAÇÃO DO CÓDIGO MUTANTE

Seguindo os passos da metodologia, vamos inserir intencionalmente as vulnerabilidades que estão sendo objeto dos testes, ou seja, aquelas indicadas pela árvore de ataque. Para isso foi identificado as vulnerabilidades mapeadas e elaborado uma lista de operadores de mutação para alterar o código original, com intuito de incluir essas vulnerabilidades. Foi pesquisado pelas vulnerabilidades relativas ao escopo da aplicação e a forma como elas são corrigidas. Assim geramos operadores que procuram no código fonte por características de código escrito de forma correta e os modificam, tornando-o vulnerável. Dessa forma, foi possível gerar operadores que realmente incluíssem vulnerabilidades na aplicação e não apenas erros comuns.

Foram criados três operadores de mutação que se propõem a inserir de forma automática vulnerabilidades relacionadas à falha de injeção, removendo as funções de validação ou proteção dos dados de entrada. A vulnerabilidade de "validação de entrada falha" permite ataques de injeção de comando no banco de dados, *SQL injection*, e ataques de execução de *script* na aplicação, *Cross Site Scripting* (XSS). Esses ataques possibilitam percorrer os possíveis caminhos de ataque representados pela árvore de ataque específica gerada para a validação experimental. A tabela 1 apresenta os operadores de mutação que serão utilizados.

Tabela 1 - Operadores de mutação

m	Cod.Operador	Descrição do Operador
1	Mod_rec_param	Moficar a forma como algumas variáveis recebem dados fornecido por usuários
2	Rem_AttrProtected	Comentar as linhas que possuem funções para ativar proteção de variáveis importantes
3	Inibe_Validates	Desabilita as funções de validação de valores de variáveis

Os operadores de mutação apresentados atuam, respectivamente, da seguinte forma:

(1) Procuram no código fonte original por variáveis que tem seus valores atribuídos através do caractere "?" em vez de ser atribuído diretamente da string digitada pelo usuário, e modificam o código para que elas tenham seus valores atribuídos diretamente;

(2) Procuram no código fonte original as variáveis que possuem algum tipo de proteção pela função "*Attr_Protected*", como por exemplo, *update_attributes* ou *new(attributes)*. O operador comenta as linhas que ativam essa proteção;

(3) Procuram no código fonte original as funções que fazem a validação de entrada dos dados, inseridos pelos usuários na aplicação, e as desabilita, por exemplo, inserindo uma função que determina se a validação será executada ou não, e forçando o seu resultado negativo;

Para este conjunto de operadores de mutação foi gerado um mutante *m*. Iremos aplicar em cada mutante os mesmos testes executados inicialmente e analisar o resultado da aplicação dos mesmos.

4.6.1 Geração do código mutante 1

Este operador de mutação modifica o trecho de código responsável por fazer a leitura do campo digitado pelo usuário no *browser* e por montar a consulta SQL a ser executada no banco. O código original utiliza-se o caractere ' ? ' para indicar onde o dado digitado pelo usuário deve ser inserido. Essa característica da linguagem *Ruby on Rails* cria uma espécie de filtro para caracteres especiais, ou seja, a consulta ao banco de dados é montada de forma que qualquer caractere digitado pelo usuário seja interpretado como um caractere normal. Portanto, as aspas, ponto e vírgula, traços não serão interpretados pelo banco de dados como comandos da linguagem, e sim como um caractere normal. Como resultado da aplicação desse operador de mutação, o código original que era:

```
user = User.find :first, :conditions => ('email = ? AND password_hashed =
?', params(:user)(:email), encrypted_password(params(:user)(:password)))
```

torna-se o código mutante abaixo:

```
user = User.find :first, :conditions => ('email = "' +
params(:user)(:email) + '" AND password_hashed = "' +
encrypted_password(params(:user)(:password)) + '"')
```

4.6.2 Geração do código mutante 2

O segundo operador de mutação modifica no programa original os trechos de código que protegem os atributos de variáveis importantes da aplicação de um conjunto de atribuições. Essa proteção ignora qualquer atribuição aos atributos da variável protegida, protegendo-as de serem sobrescritos por usuários mal intencionados através da adulteração de formulários.

Esse operador é bastante simples. Ele percorre o código fonte procurando pela *string* "Attr_Protected". Quando ele encontra a *string* procurada, o operador insere um "#" antes da *string*. Com esse símbolo, toda a linha será interpretada pelo compilador como um comentário de código, ou seja, a função de proteção da variável foi desabilitada.

Como exemplo da aplicação desse operador de mutação, uma linha no código original que era:

```
attr_protected :admin
```

após a mutação torna-se:

```
#attr_protected :admin
```

4.6.3 Geração do código mutante 3

O terceiro operador de mutação modifica no programa original os trechos de código que fazem as validações dos campos de entrada através da função "validates". Uma das características desse método de validação nativo da linguagem *Ruby on Rails* é um parâmetro opcional que determina se aquela validação vai ser executada ou não. O parâmetro funciona de forma que se uma determinada função, definida no próprio parâmetro, for verdadeira, a validação é executada; se for falsa, a validação não é executada. A ideia é inserir este parâmetro em todas as chamadas para os métodos de validação com uma função que sempre terá o resultado falso, por exemplo, testar se 1 é menor que 0. Dessa forma, as validações nunca serão executadas, permitindo, por exemplo, que seja injetado caracteres especiais maliciosos nos campos que serão processados pela aplicação ou pelo banco de dados.

Uma validação de dados que no código original invocava um método da seguinte forma:

```
validates_numericality_of :value
```

passou a invocá-lo assim:

```
validates_numericality_of :value, :if => Proc.new { |x| 1 < 0 }
```

Este operador de mutação atua varrendo o código em busca das chamadas para validação de valores de uma variável. Por exemplo, quando ele encontra "validates_numericality_of", ele insere o parâmetro opcional com uma função que sempre retornará falsa ("`:if => Proc.new { |x| 1 < 0 },` "), desabilitando a validação feita pelo método.

4.7 TESTES NOS CÓDIGOS MUTANTES

Após a realização das mutações de código, objetivando inserir as vulnerabilidades mapeadas na árvore de ataques, repetimos os testes realizados antes das mutações. Foram executados os mesmos testes e as configurações das ferramentas foram mantidas para podermos comparar os resultados obtidos.

4.7.1 Resultados dos testes no código mutante 1

Nos testes realizados no mutante 1, das três ferramentas escolhidas para executar os testes, apenas duas identificaram as vulnerabilidades inserida pelo operador de mutação 1. A ferramenta Pblind identificou que o mutante estava vulnerável a *SQL Injection*, conforme mostrado na figura 7.

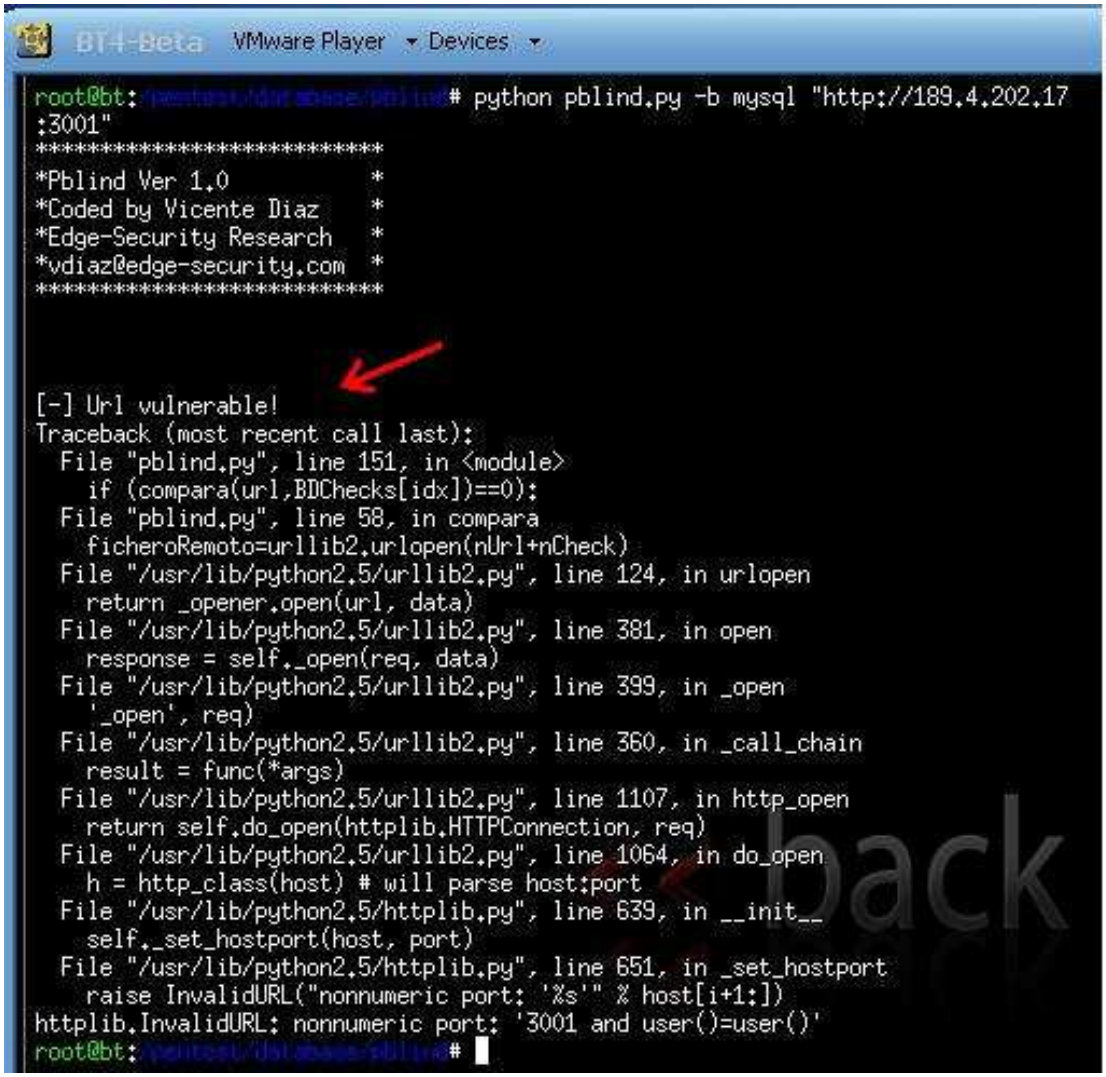

Figura 7 - Resultado do teste realizado com a ferramenta Pblind no mutante 1

Já a ferramenta w3af não identificou qualquer vulnerabilidade presente no código, conforme apresentado na figura 8. Esse resultado nos leva a concluir que ela não é adequada para o escopo do teste ou então, ela não foi configurada adequadamente, ou seja, o conjunto de teste precisa ser revisto.

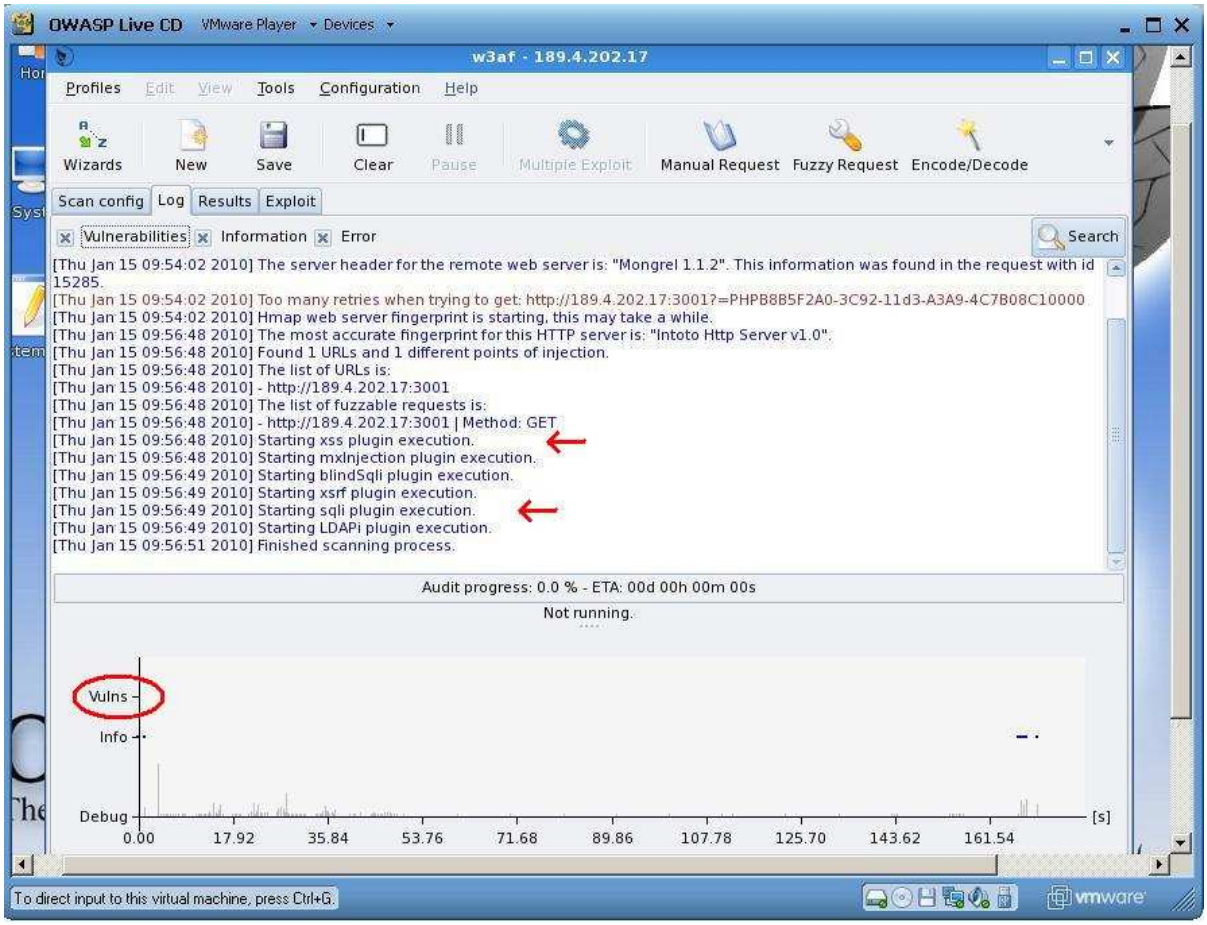

Figura 8 - Resultado do teste realizado com a ferramenta w3af no mutante 1

Apesar de a ferramenta NStalker cobrir todas as vulnerabilidades de falha de injeção do escopo dos testes, e ter sido configurada para tal, ela foi capaz de identificar apenas a vulnerabilidade de *Cross Site Scripting* (XSS). Ele detectou que a aplicação está vulnerável em três locais diferentes, conforme podemos verificar na figura 9.

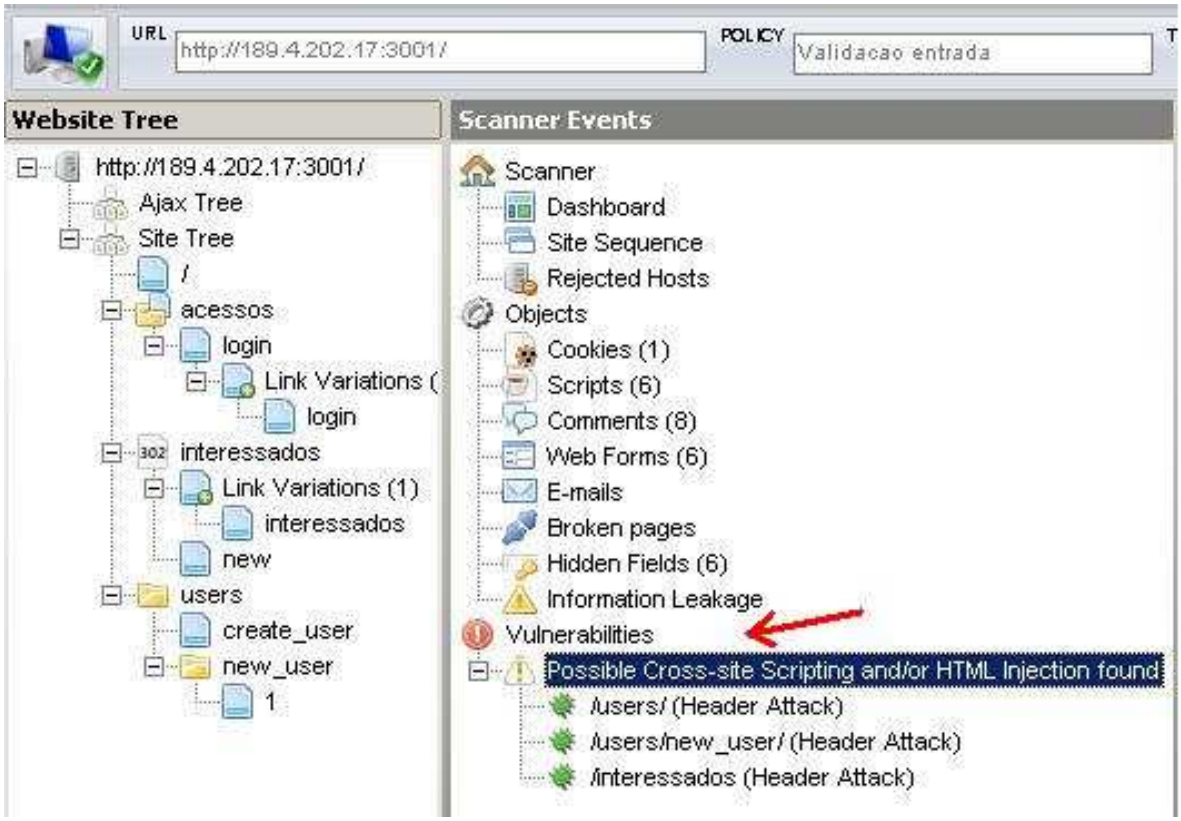

Figura 9 - Resultado do teste realizado com a ferramenta NStalker no mutante 1

4.7.2 Resultados dos testes no código mutante 2

Os resultados da execução dos testes no mutante 2 com as ferramentas escolhidas foram semelhantes aos do mutante 1: das três ferramentas escolhidas, duas identificaram as vulnerabilidades inserida pelo operador de mutação 2. A ferramenta Pblind, indicou que a aplicação estava vulnerável a *SQL Injection*, de acordo com resultado apresentado na figura 10.

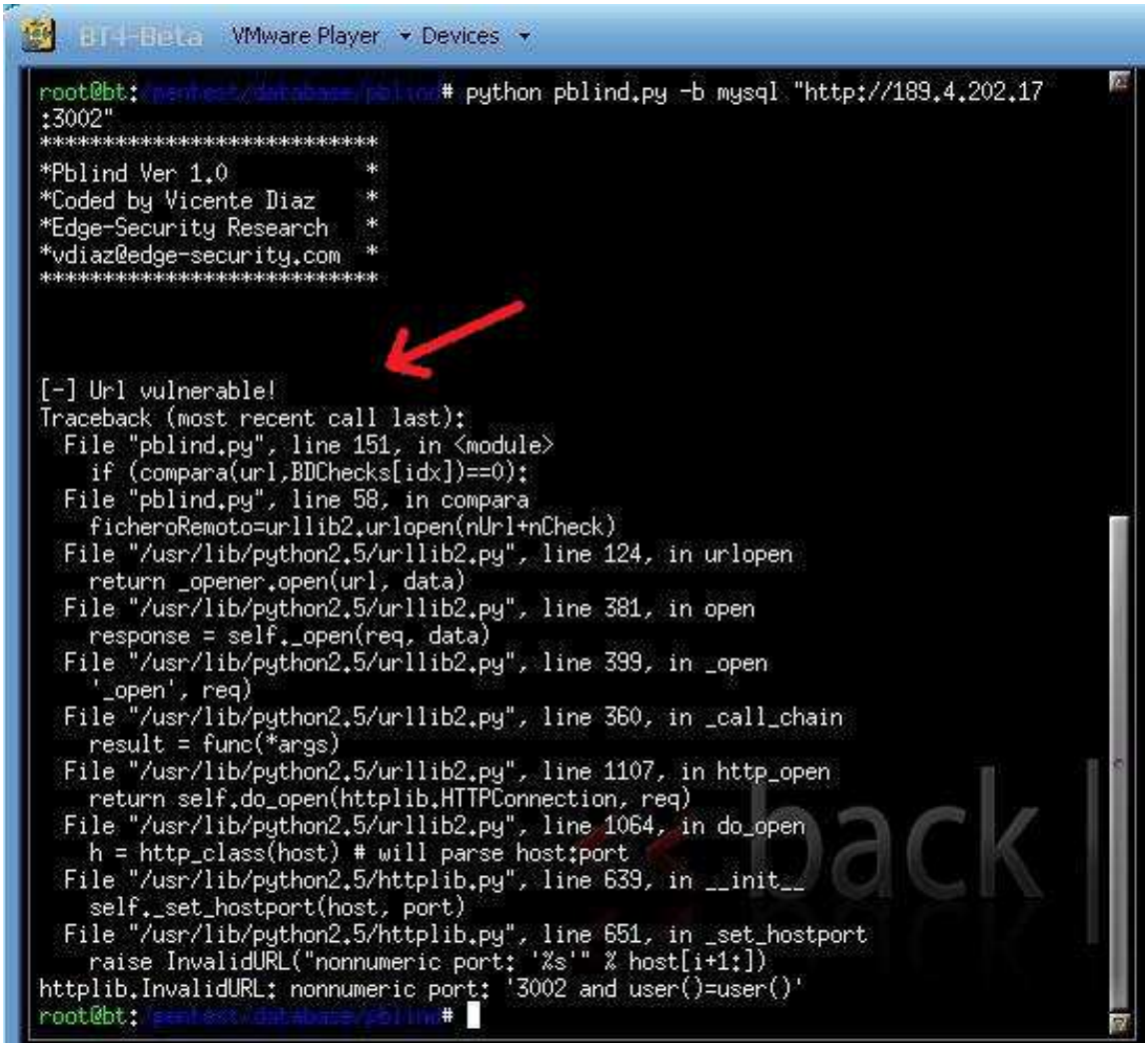

Figura 10 - Resultado do teste realizado com a ferramenta Pblind no mutante 2

A ferramenta w3af, novamente não foi capaz de identificar vulnerabilidades no código. A figura 11 apresenta o resultados da execução dessa ferramenta no mutante 2.

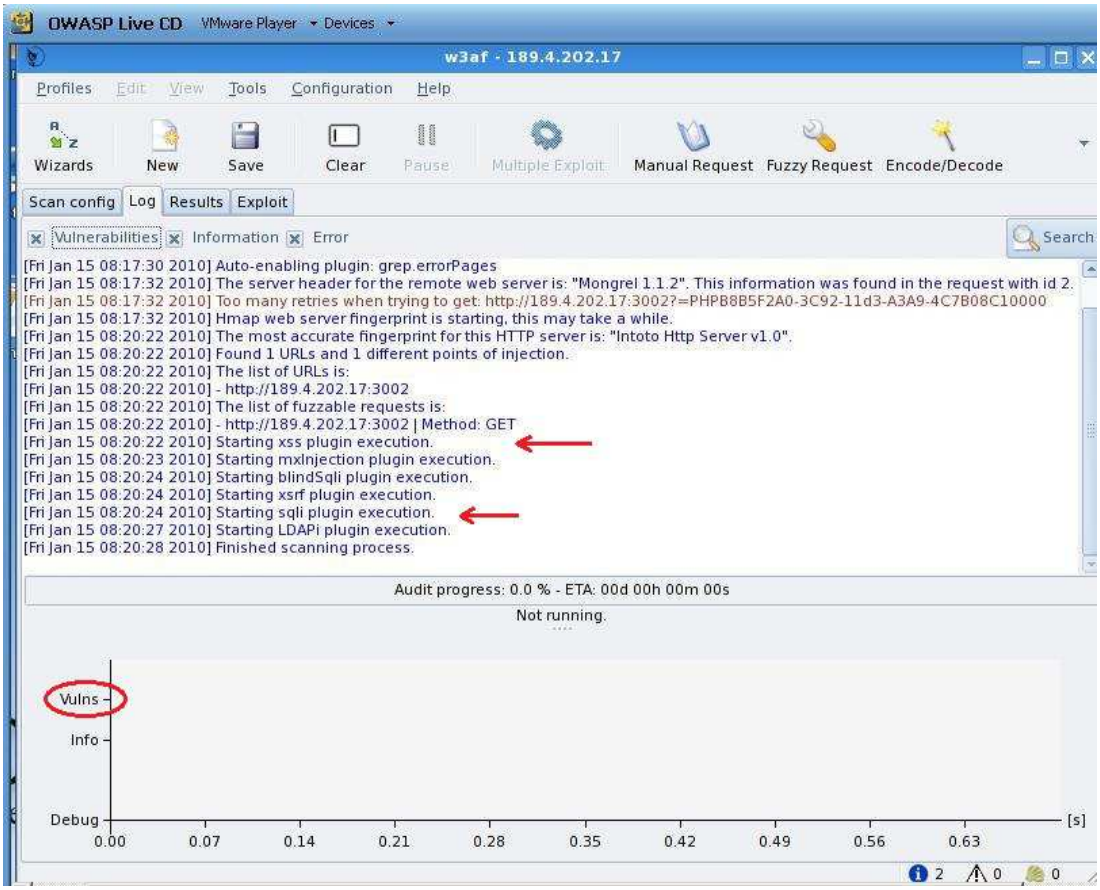

Figura 11 - Resultado do teste realizado com a ferramenta w3af no mutante 2

Já a ferramenta comercial, NStalker, conseguiu identificar apenas as vulnerabilidades do tipo *Cross Site Scripting* (XSS), encontradas nas mesmas funções que apresentaram a vulnerabilidade quando da execução dos testes no mutante 1. No entanto a ferramenta não detectou nenhuma vulnerabilidade do tipo *SQL Injection*. O resultado pode ser verificado na figura 12.

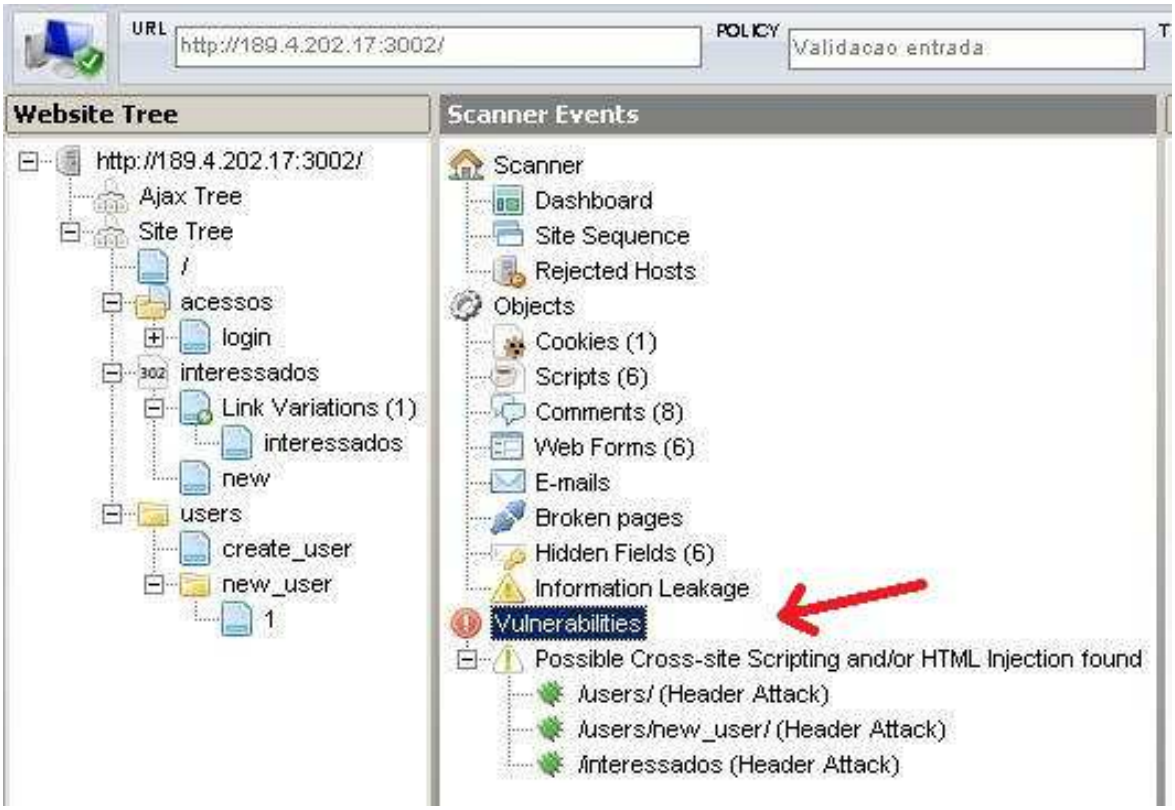

Figura 12 - Resultado do teste realizado com a ferramenta NStalker no mutante 2

4.7.3 Resultados dos testes no código mutante 3

O resultado da execução dos testes no mutante 3, assim como nos mutantes 1 e 2, apenas duas das três foram capazes de identificar a vulnerabilidade inserida pelo operador de mutação 3. Apesar de obtermos o mesmo quantitativo de ferramentas que identificaram vulnerabilidades no mutante 3, a ferramenta NStalker foi capaz de explorar mais de um tipo vulnerabilidade.

A ferramenta Pblind identificou apenas que o código estava vulnerável a *SQL Injection*, conforme mostra a figura 13.

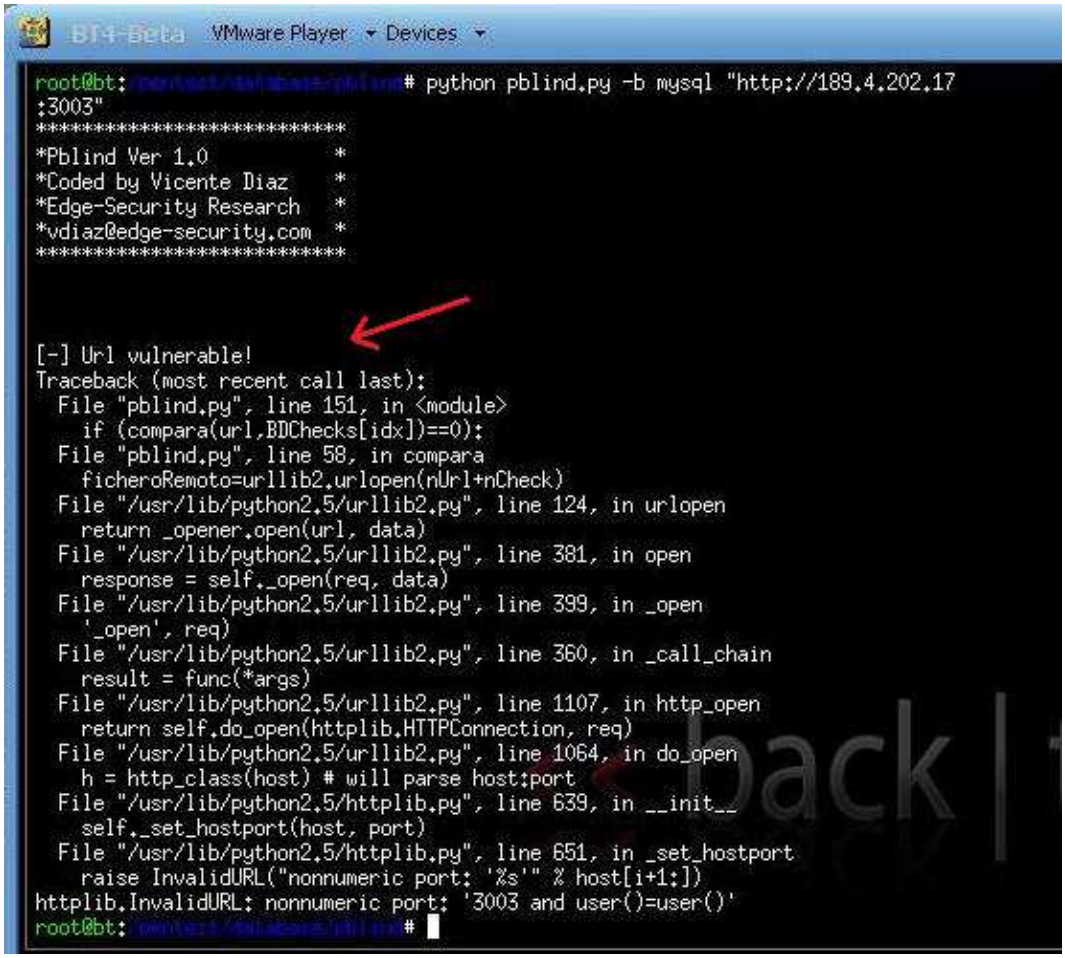

Figura 13 - Resultado do teste realizado com a ferramenta Pblind no mutante 3

A ferramenta w3af não identificou qualquer vulnerabilidade no código do mutante 3. O resultado pode ser verificado na figura 14.

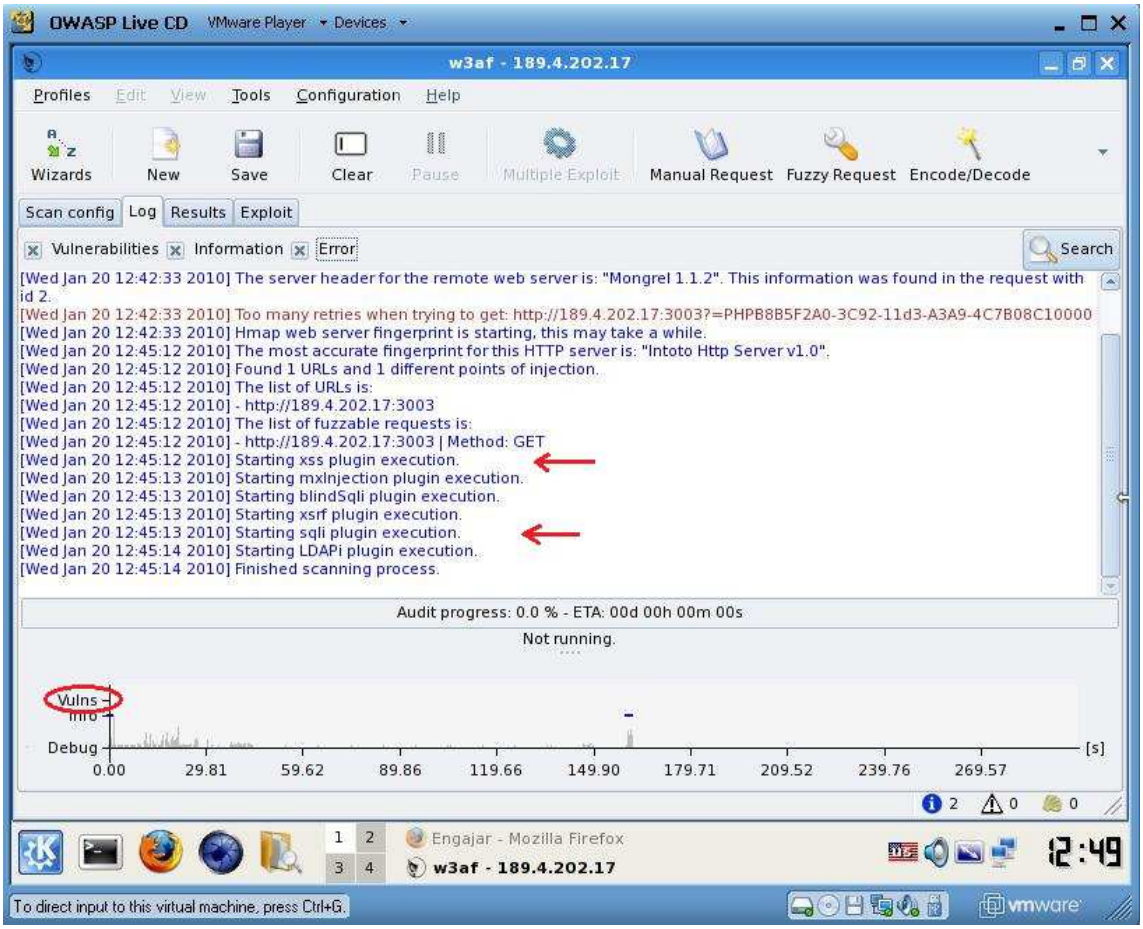

Figura 14 - Resultado do teste realizado com a ferramenta w3af no mutante 3

Assim como as outras ferramenta, a NStalker apresentou para o mutante 3, resultado semelhante ao apresentado para os mutantes 1 e 2, ou seja detectou apenas as vulnerabilidades do tipo *Cross Site Scripting* (XSS). O resultado pode ser verificado na figura 15.

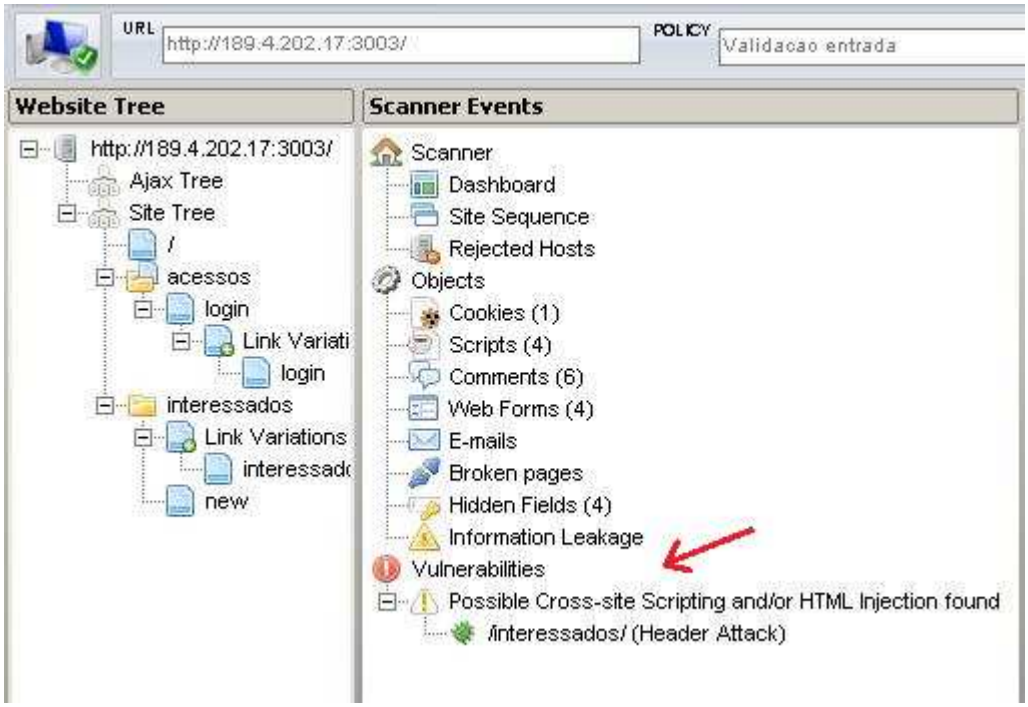

Figura 15 - Resultado do teste realizado com a ferramenta NStalker no mutante 3

Posteriormente, identificamos que apesar da ferramenta NStalker ter apresentado no resultado de teste do mutante 3 apenas o tipo de vulnerabilidade relacionada a *Cross Site Scripting*, durante a execução dos testes no mutante 3 ela conseguiu de algum modo inserir informações na base de dados, indicando que provavelmente o código estava vulnerável a *SQL Injection* também, no entanto esta vulnerabilidade não foi reportada. A figura 16 apresenta os dados que foram inseridos no cadastro de Interessados.

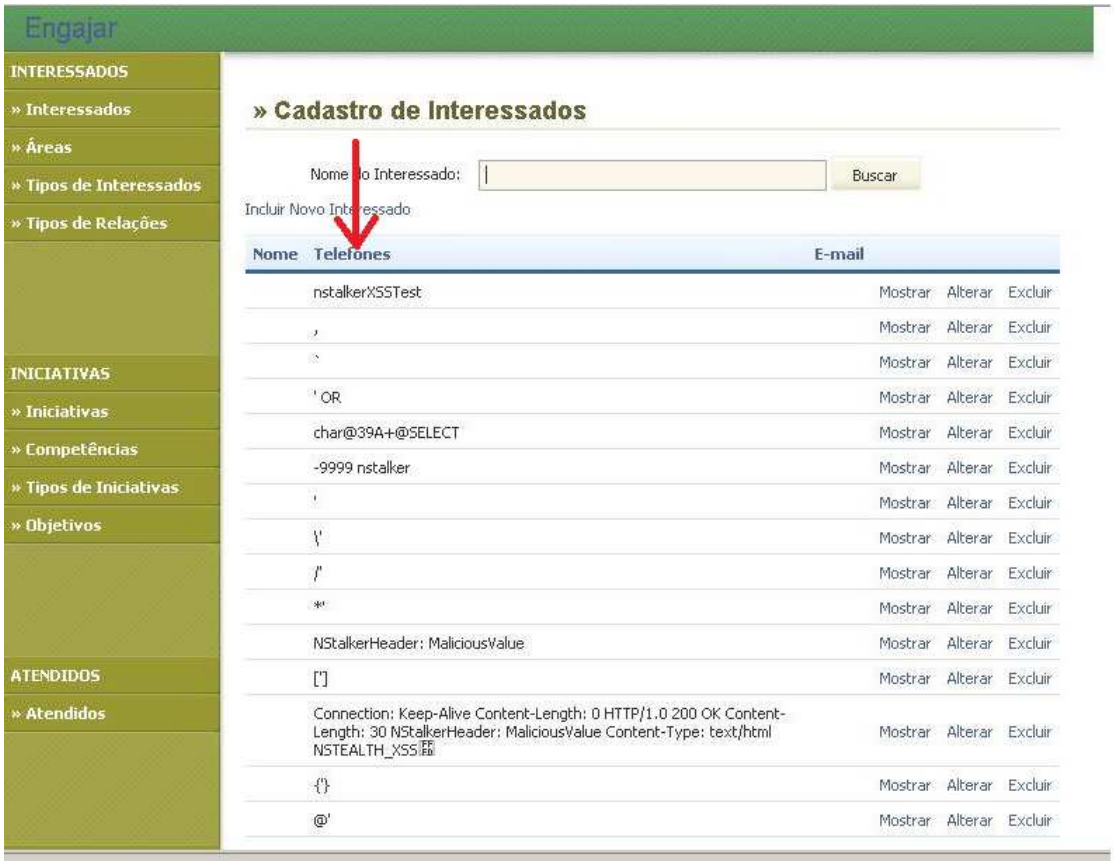

Figura 16 - Base de dados do cadastro de Interessados, após teste no mutante 3

4.8 CONCLUSÃO

Os resultados da massa de testes, no código original e posteriormente no código mutante, nos permitem afirmar que os testes executados por duas das ferramentas escolhidas, quando aplicados no código original, foram eficazes para identificar as vulnerabilidades dos ramos da árvore de ataque selecionados para atingir o objetivo de ataque descrito na raiz da árvore (“Acessar Dados Cadastrais dos Interessados”). No entanto, essa validação de ferramentas e testes está condicionada ao escopo das tecnologias empregadas na aplicação testada, ou seja, um determinado teste ou ferramenta será validado para o escopo delimitado pela árvore de ataque utilizada. Nesse caso, o escopo da árvore foi determinado com base na linguagem de programação utilizada no desenvolvimento da aplicação, banco de dados usado no armazenamento dos dados manipulados e no servidor de aplicação.

Ao analisarmos cada teste ou ferramenta individualmente, podemos perceber que nem todos foram capazes de matar todos os mutantes, conforme podemos observar na tabela 2.

Tabela 2 - Resultado apresentado por cada ferramenta na detecção de vulnerabilidades em cada mutante

	Ferramentas de Testes de Segurança				
	Pblind (SqlI)	w3af (SqlI)	w3af (XSS)	Nstalker (SqlI)	Nstalker (XSS)
Mutante 1	S	N	N	N	S
Mutante 2	S	N	N	N	S
Mutante 3	S	N	N	N	S
Qtde Mutantes Mortos por ferramenta/testes	3	0	0	0	3

Considerando todos os testes e ferramentas utilizadas como um conjunto de teste único, a validação para este conjunto de teste, na identificação das vulnerabilidades de "validação de entrada falha", que podem ser explorada pelos ataques do tipo SQL *Injection* e *Cross Site Scripting,* foi garantida. A validação foi garantida uma vez que ao aplicar o conjunto de teste no código original não foi detectada nenhuma vulnerabilidade e ao aplicar o mesmo conjunto de teste nos códigos que sofreram mutação para inserção de vulnerabilidades, o conjunto de teste demonstrou que a aplicação está vulnerável a esses tipos de ataque.

Nos casos dos testes das ferramentas que tiveram a quantidade de mutantes mortos igual a zero, ou seja, não foram capazes identificar nenhuma vulnerabilidade inserida, esses testes precisam ser redefinidos e/ou as ferramentas re-configuradas ou corrigidas, de forma a poderem identificar as vulnerabilidades para as quais eles se propõem.

5 CONSIDERAÇÕES FINAIS

É apresentado, neste capítulo, o resumo da pesquisa descrita nesta dissertação, expondo as principais contribuições e sugestões para prosseguimento do trabalho.

5.1 RESUMO DO TRABALHO

Este trabalho buscou a criação de uma metodologia para a avaliação dos testes de segurança em aplicações web. O emprego de árvore de ataques permite um verdadeiro mapeamento estruturado do universo de possíveis vulnerabilidades, de forma a construir uma análise sistemática dos elementos de maior criticidade. Na realização do estudo de caso ficou evidente a importância de definir os cenários de avaliação do teste, de forma a efetivamente cobrir os pontos mais vulneráveis. O uso de técnicas de mutação de código é sem dúvida um potente aliado para realizar, de forma automatizada, a inserção de vulnerabilidades em uma aplicação. É patente que para cada vulnerabilidade e para cada ambiente de desenvolvimento será necessário construir um conjunto de operadores de mutação específico. Porém, uma vez criado um conjunto de operadores para um dado ambiente, é relativamente fácil instanciá-lo para outro ambiente.

Os resultados obtidos surpreenderam: de um conjunto de três ferramentas analisadas, uma não cobriu totalmente e outra, parcialmente, os pontos definidos como críticos, o que reforça a importância de uma contribuição no sentido de averiguar a real eficácia das ferramentas de avaliação de segurança de aplicativos. A repetição sistemática (repetibilidade) dessa abordagem para outros pontos críticos permite a posterior construção de um perfil de abrangência de cada ferramenta de teste.

5.2 PRINCIPAIS CONTRIBUIÇÕES

Uma das contribuições desta pesquisa foi a elaboração de uma metodologia que gera subsídios para uma real verificação da eficácia de testes e ferramentas de avaliação de segurança de aplicativos para cenários específicos, bem como a própria validação de segurança de uma aplicação web. Cremos que a metodologia proposta possa auxiliar os profissionais responsáveis por testes de segurança, tanto na elaboração dos seus próprios testes como na escolha de ferramentas automatizadas.

Mesmo tendo ciência de que a elaboração de uma árvore de ataque genérica para “acessar os dados de uma aplicação web” representa apenas uma pequena amostra dos vários objetivos de ataque que uma aplicação pode sofrer, a árvore genérica gerada é uma contribuição não menos importante, uma vez que os dados manipulados pela aplicação, geralmente, são alguns dos ativos críticos de qualquer aplicação.

A apresentação dos resultados, a partir da proposta de avaliação experimental, contribuiu para verificar a viabilidade do uso da metodologia proposta. Os resultados gerados foram considerados úteis para determinar o sucesso da metodologia proposta.

Embora o trabalho realizado nesta dissertação esteja focado em aplicações web, nada impede que ele seja empregado em outras áreas.

5.3 TRABALHOS FUTUROS

Alguns aspectos relacionados aos assuntos abordados foram identificados durante o desenvolvimento do trabalho, mas não foram tratados com a profundidade necessária. Esses aspectos são apresentados como trabalhos futuros, importantes para a continuidade do trabalho, tais como:

- estender o escopo de validação dos testes, ampliando a árvore de ataque genérica para outros ativos críticos de aplicações web;
- elaboração de novos operadores de mutação e testes para cobrir toda a nova árvore de ataque;
- desenvolvimento de uma ferramenta para realizar as mutações de código, com intuito de inserir, de forma automática, as vulnerabilidades de segurança no código da aplicação a ser testada. Esse trabalho exigiria um levantamento mais extensivo das vulnerabilidades e o mapeamento de padrões de códigos corretos que poderiam ser modificados para se tornarem códigos vulneráveis. Com base neste mapeamento, a ferramenta faria a varredura do código original em busca destes padrões e os operadores de mutação realizariam as modificações, gerando códigos mutantes com vulnerabilidades de segurança.

6 REFERÊNCIAS

ACREE, A. T. et al. **Mutation analysis**. Atlanta: Georgia Institute of Technology, 1979. (Relatório Técnico GIT-ICS-79/08).

BUENO, P. M. S. **Geração automática de dados e tratamento de não executabilidade no teste estrutural de software.** 1999. Dissertação. (Mestrado em Engenharia Elétrica) – Faculdade de Engenharia Elétrica e Computação, Universidade Estadual de Campinas, Campinas, 1999.

CVE. **Common vulnerabilities and exposures**. 2006. Disponível em: http://www.cve.mitre.org, Acesso em: 23 ago. 2009.

DEMILLO, R. A. ; LIPTON, R. J. ; SAYWARD, F.G. **Hints on test data selection**: help for the practicing programmer. **IEEE Computer**, New York, v. 11, n. 4, p. 34-41, Apr. 1978.

EDWARDS, D. ; SIMMONS, S. ; WILDE, N. **Prevention, detection and recovery from cyber-attacks using a multilevel agent architecture**. Pensacola: Department of Computer Science, University of West Florida, 2007.

ESPEDALEN, J. **Attack trees describing security in distributed internet-enabled metrology**. 2007. 91 f. Dissertação (Mestrado em Informática) - Department of Computer Science and Media Technology, Gjøvik University College, Gjøvik, Noruega, 2007.

FONSECA, J. ; VIEIRA, M. Mapping software faults with web security vulnerabilities. In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 38., 2008, Anchorage, Alasca. **Proceedings …** .Anchorage: IEEE/IFIP, 2008.

FONSECA, J. et al. Vulnerability & attack injection for web applications, In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 39., 2009, Lisboa. **Proceedings ….** Lisboa: IEEE/IFIP, 2009.

GIACOMETTI, C. et al. Teste de mutação para a validação de aplicações concorrentes usando PVM. **REIC – Revista Eletrônica de Iniciação Científica**, Porto Alegre, v.2, n.3, set. 2002.

GUIDETTI, S. A. **Aplicação de análise de mutantes à geração de dados de teste para detecção de vulnerabilidade do tipo buffer overflow**. Campinas:Universidade Estadual de Campinas, 2005.

HARRIS S. et al. **Gray hat hacking:** the ethical hacker's handbook. New York: McGraw-Hill Osborne Media, 2004.

HOGLUND, G. ; MCGRAW, G. **Exploiting software:** how to break code. Reading: Addison Wesley, 2006.

HUANG, Y. W. et al. Securing web application code by static analysis and runtime protection**.** In: INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, 13., 2004, New York. **Proceedings …** New York: ACM, 2004.

HUANG, Y. W.et al. Web application security assessment by fault injection and behavior monitoring. In: INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, 12., 2003. Budapest. **Proceedings …** New York: ACM, 2003. p. 148-159.

KHAND, P. A. System level security modeling using attack trees**.** In: INTERNATIONAL CONFERENCE ON COMPUTER, CONTROL AND COMMUNICATION, 2., 2009.Karachi, Paquistan. **Proceedings …** Piscataway: IEEE, 2009

LI, X. ; HE, K. A unified threat model for assessing threat in web applications, In: INTERNATIONAL CONFERENCE ON INFORMATION SECURITY AND ASSURANCE, 2., 2008, Busan, Korea. **Proceedings …** Piscataway: IEEE, 2008. p. 142-145.

LONG, J. **Google hacking for penetration testers.** Rockland, MA: Syngress Publishing, 2005.

MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing, In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 15., 1991. Tóquio, **Proceedings …** Piscataway: IEEE , 1991, p. 604-605.

MAUW, S. ; OOSTDIJK, M. Foundations of attack trees**.** In: WON, D. ; KIM, S. (Ed). **Information security cryptology,** ICISC 2005. Berlin: Springer, 2006. p. 186–198

MOORE, A. P. ; ELLISON, R. J. ; LINGER, R. C. **Attack modeling for information security and survivability.** Carnegie Mellon: Carnegie Mellon University, 2001. (Technical Report, CMU/SEI-2001-TN-001).

MYERS, G. J. **The Art of software testing**, New York: John Wiley & Sons, 1979.

MYERS, G. J. **The Art of software testing, 2 ed.** New York: Wiley & Sons, 2004.

OFFUTT, A. J. A practical system for mutation testing: help for the common programmer, IEEE INTERNATIONAL TEST CONFERENCE ON TEXT:THE NEXT 25 YEARS. 1994. Charlotte, NC. **Proceedings …** .Waskington: IEEE, 1994.

OFFUTT, A. J. ; LEE, S. D. How strong is weak mutation? SYMPOSIUM ON TESTING, ANALYSIS, AND VERIFICATION. 1991, Victoria, Canada. **Proceedings …** New York: ACM, 1991.

OWASP**. Open web application security project. live CD project,** 2009. Disponível em: http://www.owasp.org/index.php/Category:OWASP_Live_CD_Project. Acesso em: 15 dez. 2009.

OWASP. **Open web application security project. ruby on rails security.** Guide V. 2**,** 2009. Disponível em: http://www.owasp.org/index.php/Category:OWASP_Ruby_on_Rails_Security_Guide_V2 Acesso em: 15 dez. 2009.

OWASP. **Open web application security project.** Testing Guide v. 2**,** 2007. Disponível em: <http://www.owasp.org/index.php/Category:OWASP_Testing_Project> Acesso em: 15 dez. 2009.

OWASP. **Open web application security project.** Top Ten**,** 2007. Disponível em: <http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project> Acesso em: 15 dez. 2009.

PRESSMAN, R. S. **Software engineering**: a practitioner s approach, 6. ed. New York: McGraw-Hill, 2004.

RAPPS, S. ; WEYUKER, E. J., Selecting software test data using data flow information. **IEEE Transaction on Software Engineering**, Los Alamitos, v. 11, n. 4, p. 367-375, Apr. 1985.

SCHNEIER, B. **Attack trees**, Dr. Dobb's Journal. 1999. Disponível em: http://www.schneier.com/paper-attacktrees-ddj-ft.html. Acesso em: 13 mai. 2009.

SHAHRIAR, H. ; ZULKERNINE, M. Mutation-based testing of format string bugs. IEEE HIGH ASSURANCE SYSTEMS ENGINEERING SYMPOSIUM, 11., 2008. Najing. **Proceedings …** Los Alamitos: IEEE, 2009.

SHEMA, M. **Hack notes**: segurança na web: referência rápida. Rio de Janeiro: Campus, 2003.

TEVIS, J. E. J. ; HAMILTON JR, J. A. **Static analysis of anomalies and security vulnerabilities in executable files.** 44th ANNUAL SOUTHEAST REGIONAL CONFERENCE, 44., 2006, Melbourne, FL **Proceedings …** New York: ACM, 2006.

THURAISINGHAM, B. **Building trustworthy semantic webs.** New York: Auerbach Publications, EUA, 2008.

VICENZI, A. M. R et al; **Operadores essenciais de interface: um estudo de caso**. São Paulo: USP/UEM, 1999. Disponível em: < http://www.inf.ufsc.br/~sbes99/anais/SBES-Completo/34.pdf>., Acesso em: 12 ago 2009.

VIEGA, J. ; MCGRAW, G. **Building secures software**: How to avoid security problems the right way. Reading: Addison-Wesley, 2002.

WASC – Web Application Security Consortium (2004) "WSTC – Web Security Threat Classification", Disponível em: http://www.webappsec.org/projects/threat. Acesso em: 13 mai. 2009.

APÊNDICES

APÊNDICE A - VULNERBILIDADES EM APLICAÇÕES WEB

Além das vulnerabilidades que foram descritas na seção 2.6, descreveremos agora as demais vulnerabilidades encontradas em aplicações web e que foram consideradas como as mais críticas pela Fundação OWASP. Apesar delas não estarem presentes diretamente no escopo da validação experimental, instanciada nessa dissertação, elas são importantes para uma validação prática das ferramentas e testes de segurança.

1 – *Cross Site Request Forgery* (CSRF)

Cross site request forgery não é um tipo novo ataque, e embora simples é devastador. Um ataque CSRF força o navegador logado da vítima a enviar uma requisição para uma aplicação web vulnerável, que realiza a ação desejada em nome da vítima (OWASP Top Ten, 2007).

Essa vulnerabilidade pode ser facilmente disseminada, caso a aplicação web:

- não faça verificação de autorização para ações vulneráveis;
- execute alguma ação caso um *login* padrão seja enviado na requisição (ex. http://www.example.com/admin/doSomething.ctl?username=admin&passwd=admin)
- autorize requisições baseadas somente em credenciais que são automaticamente submetidas, como, por exemplo, *cookie* de sessão (se logada corretamente na aplicação) ou a funcionalidade "Relembrar-me" (se não logada na aplicação), ou ainda, um *token* Kerberos (se parte de uma Intranet que tenha o logon integrado com o Active Directory).

Infelizmente, hoje, a maioria das aplicações web confia exclusivamente em credenciais submetidas automaticamente, como por exemplo, *cookies* de seção, credenciais de autenticação básica, endereço de IP de origem, certificados SSL ou credenciais de um domínio Windows. Esse tipo de aplicação estará em risco.

Essa vulnerabilidade é também conhecida por outros nomes, como *Session Riding*, Ataques *One-Click*, *Cross Site Reference Forgery*, *Hostile Linking* e *Automation Attack*. O acrônimo XSRF é frequentemente usado. Tanto a OWASP quanto o MITRE padronizaram o uso do termo *Cross Site Request Forgery* e do acrônimo CSRF.

Todos os *frameworks* de aplicações web estão vulneráveis à CSRF (OWASP Top Ten, 2007).

O objetivo de um teste de segurança é verificar se a aplicação está protegida contra ataques CSRF pela geração e posterior requisição de algum tipo de *token* de autorização que não seja automaticamente submetido pelo *browser*.

Abordagens automatizadas: hoje poucos scanners podem detectar CSRF, mesmo que a detecção de CSRF seja possível para a inteligência dos scanners de aplicação. Entretanto, caso seu scanner de vulnerabilidade localize uma vulnerabilidade XSS e não haja proteções anti-CSRF, é muito provável que esteja em risco de ataques CSRF encubados.

Abordagens manuais: teste de penetração é uma maneira rápida de verificar que uma proteção CSRF esteja em operação. A verificação de código é a maneira mais eficiente para se verificar se o mecanismo está seguro e implementado apropriadamente.

A forma de proteção é certificar-se de que a aplicação não está se baseando em credenciais ou *tokens* que são automaticamente submetidos pelos navegadores. A única solução é utilizar um *token* personalizado de forma que o navegador não se "lembre" dele para incluí-lo automaticamente em um ataque de CSRF.

As seguintes estratégias devem estar em todas as aplicações web:

• garantir que não exista vulnerabilidades do tipo XSS na aplicação;

• inserir *tokens* randômicos personalizados em todos os formulários e URL que não sejam automaticamente submetidos pelo *browser* e verificar se o *token* submetido é correto para o usuário corrente;

• para dados sensíveis ou transações de valores, re-autenticar ou usar assinatura de transação para garantir que a requisição é genuína;

• não usar requisições do tipo GET (URLs) para dados sensíveis ou para realizar transações de valores. Usar somente métodos do tipo POST quando processar dados sensíveis do usuário;

• um único POST é insuficiente para proteção. Deve-se também combinar com *tokens* randômicos, autenticação por outros meios ou re-autenticação para proteger apropriadamente contra CSRF;

• usar *tokens* únicos e eliminar todas as vulnerabilidades XSS da aplicação.

2 – Falhas ao Restringir Acesso à URLs

Comumente, a única proteção para uma URL é não mostrar o link para usuários não autorizados. No entanto, um atacante motivado, hábil ou apenas um sortudo pode ser capaz de achar e acessar essas páginas, executar funções e visualizar dados. Segurança por obscuridade não é suficiente para proteger dados e funções sensíveis em uma aplicação. Verificações de controles de acesso devem ser executadas antes de se permitir uma solicitação a uma função sensível, garantindo que somente o usuário autorizado acesse a respectiva função (OWASP Top Ten, 2007).

Grande parte dos *frameworks* de aplicações web estão vulneráveis a falhas de restrição de acesso a URLs. O principal método de ataque para essa vulnerabilidade é chamado de "navegação forçada" (*forced browsing*), a qual envolve técnicas de adivinhação de links (*guessing*) e de força bruta (*brute force)* para achar páginas desprotegidas.

É comum que aplicações utilizem códigos de controle de acesso por toda a aplicação, resultando em um modelo complexo que dificulta a compreensão para desenvolvedores e especialistas em segurança. Essa complexidade torna prováveis a ocorrência de erros e a não validação de algumas páginas, deixando a aplicação vulnerável.

Alguns exemplos dessas falhas incluem:

• URLs "escondidas" e "especiais", mostradas apenas para administradores ou usuários privilegiados na camada de apresentação, porém acessível a todos os usuários caso tenham conhecimento de que esta URL existe, como /admin/adduser.php ou /approveTransfer.do. Estas são particularmente comuns em códigos de menus.

• aplicações geralmente permitem acesso a arquivos "escondidos", como arquivos XML estáticos ou relatórios gerados por sistemas, confiando toda segurança na obscuridade, escondendo-os.

O objetivo desse teste de segurança é verificar se o controle está forçado constantemente na camada de apresentação e nas regras de negócio para todas as URLs da aplicação.

Abordagens automatizadas: tanto scanners de vulnerabilidades e quanto ferramentas de análise manual têm dificuldades em verificar o controle de acesso na URL por diferentes razões. Scanners de vulnerabilidades possuem dificuldade em adivinhar páginas escondidas e determinar qual página deveria ser permitida para cada usuário, enquanto mecanismos de análise estática tentam identificar controles de acessos personalizados no código e ligam a camada de apresentação com as regras de negócios.

Abordagens manuais: a abordagem mais eficiente e precisa está em utilizar a combinação da revisão do código e dos testes de segurança para verificar os mecanismos de controle de acesso. Se o mecanismo é centralizado, a verificação pode ser bastante eficiente. Se o mecanismo é distribuído através de uma completa base de código, a verificação pode se tornar dispendiosa. Se o mecanismo está forçado externamente, a configuração deve ser examinada e testada.

O passo primordial para alcançar a proteção contra acessos não autorizados é planejar a autorização, criando uma matriz para mapear as regras e as funções da aplicação. Aplicações web devem garantir controle de acesso em cada URL e funções de negócio. Não é suficiente colocar o controle de acesso na camada de apresentação e deixar a regra de negócio desprotegida. Também não é suficiente verificar uma vez o usuário autorizado e não verificar novamente nos passos seguintes. De outra forma, um atacante pode simplesmente burlar o passo em que a autorização é verificada e forjar o valor do parâmetro necessário e continuar no passo seguinte.

3 – Referência Insegura Direta a Objetos

Uma referência direta a um objeto acontece quando um desenvolvedor expõe uma referência a um objeto de implementação interna, como por exemplo, um arquivo, diretório, registro na base de dados ou chave, uma URL ou um parâmetro de um formulário. Um atacante pode manipular diretamente referências a objetos para acessar outros objetos sem autorização, a não ser que exista um mecanismo de controle de acesso (OWASP Top Ten, 2007).

Por exemplo, em aplicações de Internet Banking é comum o uso do número da conta como a chave primária. Consequentemente, pode ser tentador usar o número da conta diretamente na interface web. Mesmo que os desenvolvedores tenham usado *queries* SQL parametrizadas para prevenir inserções de comandos SQL (SQL injection), e caso não exista uma verificação adicional para garantir que o usuário é o proprietário da conta e que está autorizado a ver a conta, um atacante pode manipular a partir do parâmetro do número da conta e possivelmente pode ver e modificar todas as contas.

Esse tipo de ataque aconteceu no site da Australian Taxation Office's GST Start Up Assistance em 2000, em que um usuário legítimo, mas hostil, simplesmente modificou o ABN (identificador da empresa) presente na URL. O usuário se apossou de cerca de 17.000

registros de empresas cadastrados no sistema e então enviou para cada uma delas os detalhes do ataque. Esse tipo de vulnerabilidade é muito comum, porém não é testado amplamente em muitas aplicações.

Todos os *frameworks* de aplicações web estão vulneráveis a ataques de referência insegura direta a objeto (OWASP Top Ten, 2007).

O objetivo nesta verificação de segurança consiste em verificar se a aplicação não permite que referências diretas a objetos sejam manipuladas por atacantes.

Abordagens automatizadas: scanners de vulnerabilidades possuem dificuldades de identificar os parâmetros susceptíveis a manipulação ou se a manipulação aconteceu. As ferramentas de análise estática não podem saber quais parâmetros devem ter uma verificação de controle de acesso antes de ser usado.

Abordagens manuais: uma revisão de código pode localizar parâmetros críticos e identificar se eles são susceptíveis a manipulação em muitos casos. Testes de penetração podem verificar também quando tal manipulação é possível. Entretanto, ambas as técnicas são dispendiosas e podem não ser suficientes

A melhor proteção é evitar a exposição direta aos usuários das referências a objetos, usando um índice, um mapa de referência indireta ou outro método indireto que seja fácil de validar. Caso uma referência direta a objeto seja utilizada, é necessário garantir que o usuário seja autorizado antes do uso.

4 - Execução Maliciosa de Arquivo

As vulnerabilidades de execução de arquivos são encontradas em muitas aplicações. Os desenvolvedores têm por hábito usar diretamente ou concatenar entradas potencialmente hostis com funções de arquivo ou *stream*, ou confiar de maneira imprópria em arquivos de entrada. Em muitas plataformas, *frameworks* permitem o uso de referências a objetos externos, como referências a URLs ou a arquivos de sistema. Quando um dado não é verificado adequadamente, pode ocorrer uma inclusão arbitrária remota, resultando em um processamento ou invocação de um conteúdo hostil pelo servidor web (OWASP Top Ten, 2007).

Isso permite ao atacante realizar execução de código remoto, instalação remota de *rootkit* e comprometimento total do sistema. No Windows, o comprometimento interno do sistema pode ser possível a partir do uso de PHP's SMB file *wrappers*.

Esse ataque é particularmente prevalecente em PHP. Um cuidado extremo deve ser tomado com qualquer sistema ou função de arquivo para garantir que a entrada fornecida pelo usuário não influencie os nomes de arquivos.

Todos os *frameworks* de aplicação web são vulneráveis a execução maliciosa de arquivo se eles aceitam nomes de arquivos ou arquivos do usuário. Exemplos típicos incluem: assemblies .NET que permitem argumentos de nome de arquivos ou código que aceita a escolha do arquivo pelo usuário de modo a incluir arquivos locais (OWASP Testing Guide, 2007).

Abordagens automatizadas: ferramentas de localização de vulnerabilidades possuem dificuldades em identificar os parâmetros que são usados para entrada do arquivo ou a sintaxe que faça isso. As ferramentas de análise estática podem localizar o uso de APIs perigosas, mas não podem verificar se a validação ou codificação apropriada foi implementada para proteger contra a vulnerabilidade.

Abordagens manuais: uma revisão de código pode identificar trechos no código que possam permitir que um arquivo seja incluído na aplicação. Os testes podem detectar tais vulnerabilidades, mas identificar parâmetros particulares e a sintaxe correta pode ser difícil.

A prevenção a falhas de inclusão de arquivos remotos exige um planejamento cuidadoso nas fases de arquitetura e design, avançando para os testes. Em geral, uma aplicação bem desenvolvida não usa entrada de usuário para nomes de arquivos para nenhum recurso de servidor (como imagens, documentos XML e XSL ou inclusão de script), e terá regras de firewall estabelecidas para prevenir conexões de saída para a Internet ou internamente como retorno a qualquer outro servidor.

5 - Vazamentos de Informações e Tratamento de Erros Inapropriado

Aplicações podem, sem intenção, expor informações sobre suas configurações, funcionamento interno ou violação de privacidade através de diversas formas. Elas podem expor o funcionamento interno através de informações sobre o tempo de resposta para executar determinados processos ou respostas diferentes para entradas diversas, ou mesmo exibindo uma mesma mensagem de erro com código de erros diferentes (OWASP Top Ten, 2007).

Aplicações web frequentemente vão expor informações sobre seu funcionamento interno através de mensagens de erros detalhadas ou debug. Frequentemente, essas

informações podem ser um caminho para ataques ou para o uso de ferramentas automáticas mais poderosas.

Todos os *frameworks* de aplicações web estão vulneráveis ao vazamento de informações e tratamento de erros inapropriado, uma vez que as aplicações frequentemente geram mensagens de erros e as mostram para os usuários. Muitas vezes essas informações são úteis para os atacantes, visto que elas revelam detalhes de implementações ou informações úteis para explorar uma vulnerabilidade (OWASP Top Ten, 2007).

O objetivo de um teste de segurança é verificar se a aplicação não vaza informações via mensagens de erros ou outros meios.

Abordagens automatizadas: ferramentas de scanner de vulnerabilidades geralmente ocasionarão mensagens de erros. Ferramentas de análise estática podem procurar pelo uso de API que vazam informações, mas não terão capacidade de verificar o significado dessas mensagens.

Abordagens manuais: em uma revisão de código, podemos procurar por manipulações inapropriadas de erros e outros fatores que vazam informações, mas isso demandaria um consumo de tempo elevado.

Para proteger as aplicações, os desenvolvedores devem testar suas aplicações fazendo-as gerar erros. As aplicações deveriam também incluir um padrão de exceção de manipulação para prevenir que informações desnecessárias vazem para os atacantes.

6 - Armazenamento Criptográfico Inseguro

Proteger dados sensíveis com criptografia tem sido parte chave da maioria das aplicações web. Simplesmente não criptografar dados sensíveis é muito comum. Além disso, aplicações que adotam criptografia frequentemente possuem algoritmos mal concebidos, usam mecanismos de cifragem inapropriados ou cometem sérios erros usando cifragem forte.

Todos os *frameworks* de aplicações web estão vulneráveis ao armazenamento criptográfico inseguro (OWASP Top Ten, 2007). Os problemas mais comuns são:

- não criptografar dados sensíveis;
- uso inseguro de algoritmos fortes;
- uso de algoritmos caseiros;
- continuar usando algoritmos que provadamente são fracos (MD5, SHA-1, RC3, RC4, etc.);

• difícil codificação de chaves;

• guardar chaves em sistemas de armazenamento desprotegidos.

O objetivo desse teste de segurança é verificar se as aplicações armazenam corretamente as informações sensíveis nos sistemas de armazenamento.

Abordagens automatizadas: ferramentas de pesquisas de vulnerabilidades não verificam a criptografia em sistemas de armazenamento em geral. Ferramentas de pesquisa de códigos podem detectar o uso de APIs de criptografias conhecidas, mas não podem detectar se o uso está sendo feito corretamente ou se a criptografia é realizada em um mecanismo externo.

Abordagem manual: como ferramentas de pesquisas, testes não podem verificar o armazenamento criptografado. Revisão de código é a melhor forma de verificar que a aplicação criptografa dados sensíveis e tem mecanismos de armazenamento de chaves corretamente implementados. Em alguns casos isso pode envolver o exame das configurações de sistemas externos.

O aspecto mais importante para a proteção é assegurar que tudo que deve ser criptografado está realmente criptografado. Deve-se assegurar também que a criptografia esteja corretamente implementada. Como existem várias formas de usar a criptografia incorretamente, as seguintes recomendações devem ser seguidas como parte dos testes para assegurar o manuseamento seguro de mecanismos de criptografia:

• não criar algoritmos de criptografia. Usar somente os algoritmos aprovados publicamente como, AES, Criptografia de chaves publicas RSA, SHA-256 ou melhores para *hash*;

• não usar algoritmos fracos, como MD5/SHA1. Usar mecanismos mais seguros como SHA-256 ou melhores;

• criar chaves *offline* e armazenar as chaves privadas com extremo cuidado. Nunca transmitir chaves privadas em canais inseguros;

• assegurar que credenciais de infraestrutura, como credenciais de banco de dados ou detalhes de filas de acessos MQ, estão corretamente seguras (por meio de rígidos sistemas de arquivos e controles), criptografadas de forma adequada e que não podem ser decriptografadas por usuários locais ou remotos;

• assegurar que dados armazenados criptografados no disco não sejam fáceis de descriptografar. Por exemplo, criptografia de banco de dados é inútil se a conexão de banco de dados permite acessos não criptografados;

• não armazenar dados desnecessários.

7 - Comunicações Inseguras

Aplicações podem falhar na hora de encriptar o tráfego de rede de comunicações sensíveis. A encriptação (geralmente SSL) deve ser usada em todas as conexões autenticadas, especialmente páginas web com acesso via Internet, mas também conexões com o *backend*, pois neste caso o aplicativo irá expor uma autenticação ou o *token* de sessão. Adicionalmente, a autenticação deve ser usada sempre que dados sensíveis, como aqueles relativos a cartões de crédito ou informações de saúde, são transmitidos. Os padrões PCI, por exemplo, requerem que todas as informações de cartões de crédito que são transmitidas pela internet sejam encriptadas. Aplicações cujo modo de encriptação possa ser subvertido são alvos de ataques (OWASP Top Ten, 2007).

Todos os *frameworks* de aplicações web são vulneráveis às comunicações inseguras. Falha na hora de encriptar informações sensíveis significa que um invasor, que possa escutar o tráfego da rede, poderá ter acesso à conversa, incluindo quaisquer credenciais ou informações sensíveis transmitidas. Considerando que redes diferentes terão mais ou menos suscetibilidade à escuta. Entretanto, é importante notar que eventualmente um servidor será comprometido em praticamente qualquer rede, e que invasores instalarão rapidamente uma escuta para capturar as credenciais de outros sistemas.

O uso de SSL para comunicação com usuários finais é crítico, pois é muito provável que eles utilizem formas inseguras de acessar os aplicativos. Como HTTP inclui credenciais de autenticação ou um *token* de sessão para cada pedido, toda autenticação do tráfego deve ir para o SSL, não só os pedidos de *login*.

A encriptação de informações com servidores de *backend* também é importante. Mesmo que estes servidores sejam naturalmente mais seguros, as informações e as credenciais que elas carregam são mais sensíveis e mais impactantes. Portanto, usar SSL no *backend* também é muito importante.

A encriptação de informação sensível, assim como cartões de crédito e informações de previdência, se tornou um regulamento financeiro e de privacidade para várias empresas. Negligenciar o uso de SSL para o manuseio de conexões de informações cria um risco de não conformidade.

O objetivo em um teste de segurança é verificar se as aplicações encriptam corretamente toda a comunicação autenticada e sensível.

Abordagens automatizadas: ferramentas de localização de vulnerabilidade podem verificar se o SSL é utilizado na interface do sistema e pode localizar muitas falhas relacionadas à SSL. Entretanto, elas não têm acesso às conexões no *backend* e não podem verificar se elas são seguras. As ferramentas de análise estática podem ajudar com análises de algumas ligações no *backend*, mas provavelmente não entenderão a lógica customizada requerida para todos os tipos de sistemas.

Abordagens manuais: testes podem verificar se o SSL é utilizado e localizar muitas falhas relacionadas à SSL na interface, mas as abordagens automatizadas são provavelmente mais eficientes. A revisão de código é muito eficiente para verificar o uso de SSL em conexões no *backend*.

Uma das formas de proteção é usar o SSL em todas as conexões autenticadas ou em qualquer informação sensível em transmissão.