J-Attack - Injetor de Ataques para Avaliação de Segurança de Aplicações Web

Plinio C. S. Fernandes, Tânia Basso, Regina Moraes

Faculdade de Tecnologia – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 456 – 13484-332 - Campus I – Limeira – SP – Brasil

{pliniocsfernandes,taniabasso}@gmail.com, regina@ft.unicamp.br

Abstract. *The security of software systems is mandatory in current software development scenario. It requires expertise in several areas of knowledge and includes the study of malicious faults, in order to know them and use this knowledge to allow providing computer systems ability to detect and avoid them without compromising the security required by the users. This paper presents the design and development process of a tool that is able to inject attacks in Web applications aims to use it to evaluate the security of the applications.*

Resumo. *A busca por sistemas seguros é uma necessidade no cenário de software atual. A segurança de sistemas computacionais exige conhecimentos específicos em diversas áreas de conhecimento e inclui o estudo de falhas maliciosas, para que, conhecendo-as, seja possível prover aos sistemas computacionais capacidade de detectá-las e evitá-las sem comprometer a segurança exigida pelos usuários. Este artigo apresenta o projeto e o processo de desenvolvimento de uma ferramenta capaz de injetar ataques em aplicações Web para utilização em avaliações de segurança de aplicações.*

1. Introdução

Vulnerabilidades de segurança de software têm sido amplamente exploradas com o intuito de obter informações confidenciais e invadir redes corporativas. Isto ocorre devido à complexidade dos produtos de software, que é cada vez maior, e resulta em vulnerabilidades de segurança produzidas por má codificação. Também ocorre devido restrições de tempo e custo, pois, diante desses fatores, desenvolvedores, muitas vezes, mantêm o enfoque principal do desenvolvimento da aplicação na funcionalidade e na satisfação dos requisitos do cliente, negligenciando aspectos de segurança. Assim, normalmente, não é adotado um ciclo de desenvolvimento seguro para o software. O desenvolvimento de software seguro visa contribuir com melhorias ao cenário da falta de segurança em aplicações Web e reduzir a quantidade de falhas residuais no código, que podem levar a vulnerabilidades de segurança.

Sabe-se que mecanismos tradicionais de segurança de redes como *firewalls*, criptografia e sistemas de detecção de intrusão podem proteger a rede, mas não evitam ataques às aplicações. Por este motivo, o foco de ataques tem mudado da rede para as aplicações *Web*, onde codificação insegura representa grande risco [Singhal *et al* 2007].

Sabendo da importância da segurança de software para as organizações e da expressiva quantidade de vulnerabilidades de segurança que a grande maioria das aplicações *Web* apresenta, provedores de soluções voltadas para o desenvolvimento de software colocam no mercado um número significativo de ferramentas para detectar as vulnerabilidades dessas aplicações, os *scanners* de vulnerabilidades. No entanto, estas ferramentas apresentam baixa cobertura (o scanner deixa de detectar vulnerabilidades que realmente existem na aplicação) e alta taxa de falsos positivos (o resultado da análise indica vulnerabilidade de segurança que efetivamente não existe na aplicação).

Visando avaliar as ferramentas de vulnerabilidades (*scanners*) e sistematizar a validação de aplicações *Web* quanto às vulnerabilidades de segurança, foi desenvolvido em um trabalho prévio [Fernandes *et al* 2010] uma metodologia com base em árvores de ataque. As árvores de ataque foram usadas para descrever cenários que pudessem revelar vulnerabilidades nas aplicações e para guiar a injeção de ataques para avaliar se os scanners eram eficazes na detecção de vulnerabilidades (cobertura) sem apresentar uma alta taxa de falsos positivos.

Durante o desenvolvimento do trabalho foi percebido que poderia ser criada uma ferramenta para automatizar, ao menos em parte, a injeção dos ataques através do modelo das árvores. Dessa forma, o objetivo deste artigo é relatar o projeto e desenvolvimento de um Injetor de Ataques (*J-Attack*) para avaliação de segurança de aplicações *Web*. O objetivo a ser atingido é a extensão do estudo de vulnerabilidades de segurança de sistemas computacionais, procurando prover uma maneira automática de emular ataques de segurança.

Depois da Introdução que esta Seção apresentou, o restante deste artigo apresenta na Seção 2 os trabalhos relacionados que serviram de embasamento teórico para o desenvolvimento da pesquisa relatada neste artigo; a Seção 3 apresenta as Árvores de Ataque que serviram de base para o desenvolvimento do Injetor de Ataque "*J-Attack*" que, por sua vez, é apresentado na Seção 4; um Estudo de Caso é apresentado na Seção 5 e a Seção 6 apresenta os Resultados e Discussões; finalmente a Seção 7 apresenta algumas Considerações e Trabalhos Futuros.

2. Trabalhos Relacionados

A interação do usuário com uma aplicação *Web* é feita através de um navegador que é responsável por interpretar arquivos contendo a linguagem de marcação *Hypertext Markup Language* (HTML) e *Cascading Stylesheets* (CSS), como também códigos de script em algumas linguagens, sendo a mais conhecida e utilizada o *Javascript*. O *Hypertext Transfer Protocol* (HTTP) é o protocolo de comunicação mais comumente usado por aplicações Web, sendo que as requisições contêm seu método, o *Uniform Resource Identifier* (URI) e a versão do protocolo, seguidos por uma mensagem do tipo *Multipurpose Internet Mail Extensions* (MIME) com os modificadores da requisição, informação do cliente e um conteúdo. [Fielding *et al*. 1999].

Aplicações *Web* são alvos constantes de ataques, pois são aplicações com alta exposição na rede mundial. Segundo o *Open Web Application Security Project* [OWASP 2010], uma vulnerabilidade é uma brecha ou uma fraqueza na aplicação, que pode ser uma falha de *design* ou um *bug* de implementação, que permite que um atacante cause problemas para seus *stakeholders*, ou seja, que ataque a aplicação vulnerável. Devido à

alta exposição, qualquer vulnerabilidade nas aplicações *Web* será inevitavelmente atacada em algum momento. Estes ataques não podem ser protegidos por mecanismos tradicionais de segurança de redes, tais como *firewalls*, criptografia e sistemas de detecção de intrusão. A razão é que esses ataques são feitos através de portas que são usadas pelo tráfego regular da rede [Singhal *et al* 2007] e para prover a proteção necessária, grande conhecimento sobre o contexto de negócio é necessário [OWASP 2010]. Por este motivo, o foco de ataques tem mudado da rede para as aplicações *Web*, onde codificação insegura representa grande risco [Singhal *et al* 2007].

Muitos trabalhos publicados mostram que aplicações *Web* apresentam falhas graves de segurança [Christey and Martin 2007, Acunetix 2007, NT Monitor 2008]. Acunetix (2007) apresentou o resultado de um experimento que inspecionou (*scan*) 3200 *websites* durante um ano. De acordo com esses resultados, 70% dos *websites* apresentaram vulnerabilidades de alto ou médio risco para as organizações. O relatório anual sobre segurança da NTA Monitor (2008) relata que 25% das organizações que participaram dos testes, apresentaram uma ou mais vulnerabilidades de alto risco e 78% delas contêm vulnerabilidades de médio risco. Esses fatos mostram que as aplicações *Web* são entregues sem o devido cuidado em relação à segurança e ressalta a importância da pesquisa na área de segurança para esse segmento de aplicações.

Uma maneira que as organizações provedoras de soluções para o desenvolvimento de software acharam para amenizar o problema foi o desenvolvimento de ferramentas para detectar as vulnerabilidades dessas aplicações, os *scanners* de vulnerabilidades [HP 2011, Acunetix 2011, IBM 2011]. Porém, como experimentalmente comprovado por diversos trabalhos científicos, estas ferramentas apresentam baixa cobertura e alta taxa de falsos positivos [Fonseca *et al* 2007, Vieira *et al* 2009, Bau *et al* 2010, Basso *et al* 2010].

Diversas ferramentas foram desenvolvidas para injetar ataques, normalmente voltadas para um determinado tipo de ataque, por exemplo, SQL Injection [SqlMap (2011), Havij (2011), SqlHelper (2011)]. Porém, a ferramenta que está sendo proposta procura ser flexível para que se possa acoplar injetores para diferentes vulnerabilidades além de implementar a visão do atacante através da árvore de ataques. Em Fernandes *et al* (2010), a utilidade das árvores de ataques foram investigadas e modeladas para três principais vulnerabilidades. Este trabalho complementa o anterior, implementando a ferramenta que usa os modelos desenvolvidos no primeiro trabalho.

Na próxima seção árvores de ataque são abordadas para introduzir a maneira como o injetor de ataques consegue automatizar os ataques injetados.

3. Árvores de Ataque

O modelo de árvores de ataque foi desenvolvido por SCHNEIER (1999) e tem como vantagem permitir a visualização da aplicação do ponto de vista de um atacante. Neste modelo o nó raiz representa o sucesso em alcançar a meta final de um ataque. Cada nó filho representa submetas que devem ser alcançadas para que a meta do nó pai seja alcançada. Nós pai podem estabelecer uma relação entre seus filhos utilizando um relacionamento “OU” ou “E”. Em um relacionamento OU, se qualquer meta de um nó filho for alcançada a meta do nó pai também é alcançada. Já em um relacionamento “E”, as metas de todos os nós filhos devem ser alcançadas para que a meta do nó pai seja

alcançada. Cenários individuais de invasão são gerados ao se navegar na árvore de um modo *depth-first* (busca em profundidade).

Não existem orientações específicas para a construção das árvores de ataque e no trabalho de Fernandes *et al* (2010) foi utilizada uma abordagem *bottom up*, onde os passos para alcançar a meta principal são identificados e representados de acordo com a ordem na qual eles devem ser executados (dos nós folha para o nó raiz). Para não sobrecarregar as árvores, as marcações de relacionamentos do tipo "OU" foram omitidas.

O OWASP mantém uma lista, anualmente atualizada, dos dez tipos de vulnerabilidade que apresentam o maior risco para as organizações. Desta lista [OWASP 2010] foram escolhidos, para o presente trabalho, três tipos de vulnerabilidades a serem investigadas: SQL Injection, XSS e CSRF. As vulnerabilidades escolhidas são respectivamente a primeira, a segunda e a quinta da lista. As duas primeiras foram escolhidas por estarem nos primeiros lugares da lista enquanto a quinta foi escolhida por ser uma vulnerabilidade que apareceu em muitos resultados dos *scanners* comerciais nos trabalhos anteriores do grupo e é um tipo de vulnerabilidade que já causou problemas em grandes sites como Gmail e Yahoo. A próxima Seção apresenta estas vulnerabilidades.

3.1. *SQL Injection*

Explorar vulnerabilidades do tipo *SQL Injection* consiste em enganar o interpretador de SQL para que este execute comandos manipulados. Isto é feito através da entrada de dados do usuário, que é enviada como uma parte da requisição para a aplicação *Web*. Um atacante pode obter acesso ao banco de dados da aplicação e, de forma arbitrária, criar, ler, alterar ou apagar quaisquer dados disponíveis para a aplicação. Aplicações *Web* se tornam vulneráveis para este tipo de ataque quando dados providos pelo usuário para o interpretador não são validados ou recodificados.

A Figura 1 mostra a árvore de ataque modelada para representar as etapas que permitem explorar vulnerabilidades do tipo *SQL Injection*. As etapas de 1 a 4 mostram as diferentes maneiras de injetar códigos maliciosos, ou seja, as diferentes entradas de dados [Halfond 2006]. A etapa representada pelo nó 6 consiste em encontrar mecanismos de envio de dados do *site* que transmitem informações para o banco de dados, encontrando meios adequados de ataque. Após identificar os possíveis dados de entrada para injeção, é necessário criar o trecho de código SQL que será utilizado no ataque (representado pelo nó 5). O código SQL é construído de acordo com a intenção do ataque e deve considerar algumas técnicas para evitar mecanismos de detecção de assinatura, caso necessário. Após criar o código que será injetado, este pode ser testado em diferentes mecanismos de entrada (nó 7). Validações podem ser feitas pela aplicação apenas no lado do cliente e não no servidor. Nestes casos o atacante deve utilizar métodos para passar pela validação (nó 8). Os nós 9 e 10 representam os passos para verificar a resposta do ataque e, consequentemente, se este foi bem sucedido.

3.2. *Cross Site Scripting*

O ataque do tipo *Cross Site Scripting* (XSS) consiste em executar *scripts* no navegador da vítima permitindo roubo de sessão, inserção de conteúdo hostil, roubo de

informações pessoais e outros danos. Este tipo de ataque pode ser executado quando aplicações recebem dados do usuário e enviam estes dados para o navegador sem validação ou recodificação apropriada.

Os ataques de XSS podem ser classificados em três tipos (OWASP, 2010): (i) XSS armazenado (*Stored XSS*): dados de script são injetados na aplicação e armazenados por ela e quando a aplicação é acessada mais tarde, o script é executado; (ii) XSS refletido (*Reflected XSS*): o script injetado é imediatamente mostrado, portanto executado, logo após ser enviado para a aplicação; (iii) XSS baseado em DOM (*DOM-based XSS*): ataques deste tipo utilizam scripts para manipular o *Document Object Model* (DOM) do navegador da vítima, sendo assim, executado dinamicamente mesmo quando a aplicação não sofreu modificações.

A Figura 2 mostra a árvore de ataques modelada para ataques de XSS. O primeiro passo para efetuar este tipo de ataque é encontrar os chamados *XSS holes* (nó 1), que são indicadores de que a aplicação pode ser vulnerável. Um *XSS hole* pode ser descrito como uma parte da aplicação onde é possível injetar informações para serem exibidas pela aplicação. O próximo passo é testar a injeção de *scripts* na aplicação. Para isto, é necessário verificar se a aplicação valida caracteres especiais nos dados inseridos. Se não houver validação (nó 2), o *script* malicioso pode ser injetado (nó 3).

Uma forma simples de testar se a aplicação é vulnerável seguindo os passos 1, 2 e 3, é injetar, em todos os campos de entrada de dados, um *script* simples, que apenas exibe uma mensagem, como uma das variações abaixo [Snake 2010]:

```
<script>alert('XSS')</script>
&{alert('XSS')};
<img src=javascript:alert('XSS')>
```

Caso a aplicação valide os dados inseridos, ainda pode ser possível executar o ataque com um *script* específico para evasão de filtros e acessar a aplicação alvo utilizando um navegador compatível com a sintaxe do *script* (nó 4).

Os últimos passos para fazer o ataque XSS consistem em acessar a parte da aplicação onde o *script* é executado (nó 5), pois esta parte pode não ser diretamente a próxima a ser acessada caso o ataque seja de XSS armazenado. O nó 6 consiste em acessar a aplicação com um navegador adequado, pois diferentes navegadores processam *scripts* de formas diferentes e alguns possuem mecanismos de segurança contra ataques de XSS (Browser Scope Project, 2010). Isto é importante para identificar em quais navegadores o ataque será bem sucedido.

3.3. *Cross Site Request Forgery*

Um ataque de *Cross Site Request Forgery* (CSRF) força o navegador da vítima, que está autenticado na aplicação alvo, a enviar uma requisição à aplicação *Web* vulnerável. Como a vítima tem uma sessão autenticada no navegador, a aplicação *Web* executa a ação desejada em nome da vítima, de acordo com a requisição recebida. Este tipo de ataque é possível, pois aplicações *Web* autorizam requisições baseadas em credenciais que são enviadas automaticamente, como *cookies* de sessão, endereços IP, certificados SSL entre outros [Auger 2010]. A Figura 3 mostra a árvore modelada para esta

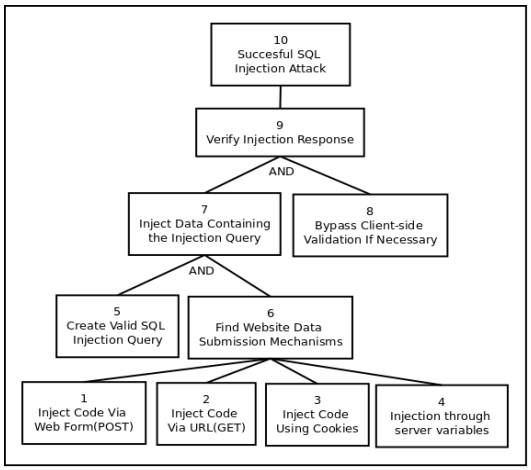

Figura 1. Árvore de Ataque - SQL Injection

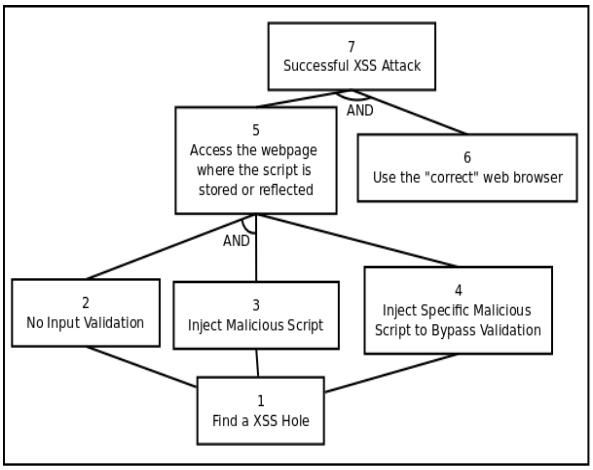

Figura 2.Árvore de Ataque–Cross Site Scripting

vulnerabilidade, cujo percurso segue a mesma lógica apresentada para vulnerabilidades precedentes.

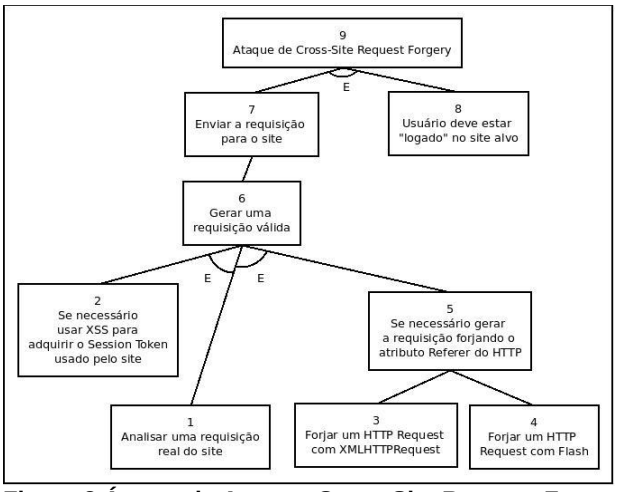

Figura 3.Árvore de Ataque–Cross Site Request Forgery

4. *J-Attack*

Esta seção tem como objetivo apresentar as soluções que foram utilizadas para o desenvolvimento da ferramenta proposta, descrevendo a aplicação, ferramentas, requisitos, soluções de implementação e procedimentos de teste.

4.1. O desenvolvimento do Projeto

Foi utilizado o modelo *Test Driven Development* (TDD) para o desenvolvimento do projeto. O modelo foi proposto por Kent Beck, mesmo autor da proposta da metodologia ágil *Extreme Programming* (XP) e a premissa básica do TDD pode ser resumida na frase de Ron Jeffries "*Clean code that works*" [Beck 2002].

Apesar de os modelos ágeis muitas vezes não possuírem uma fase de modelagem muito bem definida e de não gerarem artefatos baseados na *Unified Modeling Language* (UML) para documentação do sistema, neste projeto foram utilizados dois diagramas UML: diagrama de casos de uso e diagrama de classes. Estes dois diagramas foram selecionados pois permitem um planejamento inicial do sistema, que é importante para garantir uma boa arquitetura.

Outra característica utilizada no desenvolvimento foi a utilização de padrões de projeto (*design patterns*). Padrões de projeto são soluções padronizadas para problemas que ocorrem com frequência em projetos de software. Estas soluções definem apenas o conceito principal da solução e deve ser implementada de acordo com cada caso. Neste projeto foi utilizado o padrão "Strategy" [Gamma *et. al*., 1995]. O padrão Strategy define uma família de algoritmos, encapsula cada algoritmo e torna possível a utilização de qualquer um deles em determinado ponto do código. A utilização do Strategy no projeto permite que diferentes algoritmos sejam selecionados de acordo com a necessidade do teste em execução, i.e., de acordo com o ataque que se pretende injetar. Com isso a ferramenta ganha em flexibilidade (pode executar os testes com base no que está cadastrado no banco de dados) e escalabilidade (facilita a expansão da ferramenta, bastando para isso criar um novo algoritmo na família do Strategy).

As ferramentas que foram utilizadas durante o desenvolvimento do projeto foram: (i) Ubuntu Linux, como sistema operacional; (ii) Java, como linguagem de programação; (iii) JUnit, para os testes unitários; (iv) Netbeans IDE; (v) Apache Subversion, para o controle de versões; (vi) MySQL, como sistema gerenciador de banco de dados; (vii) MySQL Workbench, para modelagem; (viii) Dia, para modelagem; (ix) Eclipse, como ambiente de desenvolvimento integrado (ADI).

4.2. Os requisitos do Projeto

Os requisitos do injetor, entendidos pelos integrantes do Grupo de Engenharia de Sistemas e Informação da Faculdade de Tecnologia são descritos a seguir:

- Enviar requisições HTTP e verificar a resposta do *site* para as requisições;
- Mapear um *site* através dos *links* da página inicial;
- Procurar padrões nas respostas de requisições (todo o site ou em páginas específicas);
- Comparar duas ou mais respostas HTTP do servidor;
- Encontrar mecanismos de entrada de dados do *site* (*inputs*);
- Executar testes baseados em um banco de dados que contenha variações de ataques;
- Executar testes em uma determinada ordem;
- Possuir uma estrutura que possibilite a ampliação do sistema futuramente;
- Utilizar autenticação para verificar *sites* que necessitem desse recurso.

A partir do levantamento de requisitos mencionados, foi gerado o diagrama de casos de uso que é apresentado na Figura 4. Nele, definimos a interação do usuário com o sistema e as ações básicas que a aplicação deve ser capaz de executar.

O banco de dados tem a finalidade de armazenar casos de teste extraídos das árvores de ataque que serviram para guiar a ferramenta na execução dos testes. Foram identificadas as entidades principais: Vulnerabilidade (armazena tipos de vulnerabilidade), Teste (armazena testes para um determinado tipo de Vulnerabilidade) e Ação (armazena parâmetros que deve ser executada como passo de um teste).

A Ação pode ser de um dos seguintes tipos: Requisição, Procurar Padrão, Encontrar Entradas, Comparar Respostas. A Figura 5 apresenta o modelo relacional do Projeto. No modelo relacional há sete tabelas representando as principais entidades identificadas no Diagrama Entidade Relacionamento, sendo que quatro tabelas são

complementares à tabela “acao” e contém os dados específicos para cada tipo de Ação identificada.

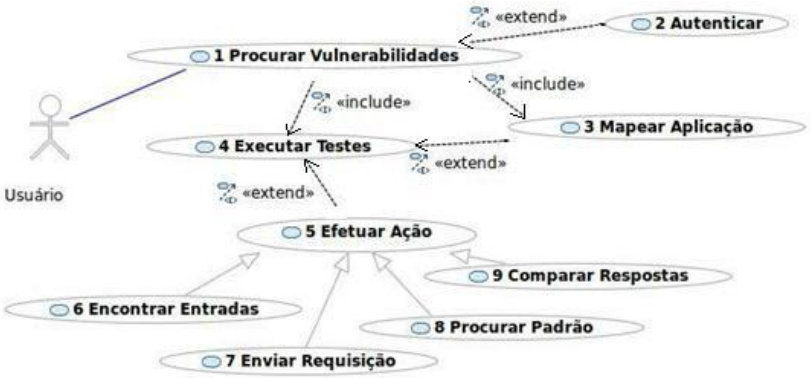

Figura 4. Diagrama de Casos de Uso da *J-Attack*

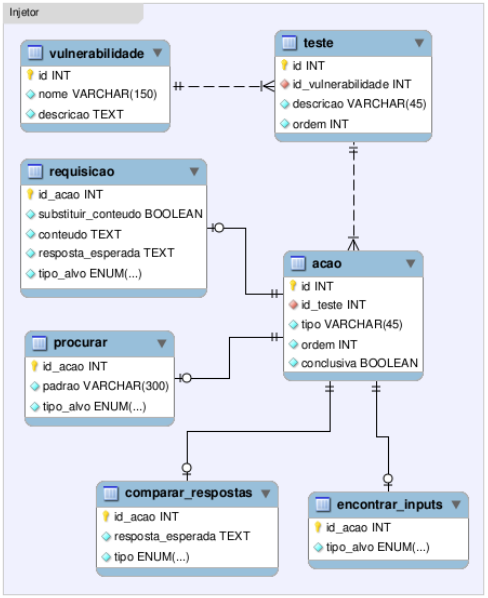

Figura 5. Modelagem Relacional do Banco de Dados da *J-Attack*

A Figura 6 apresenta o diagrama de classes do sistema. Os métodos de acesso aos atributos (“get” e “set”) das classes foram omitidos para simplificar a visualização. No diagrama é possível identificar a utilização do padrão de projeto Strategy que tem como tarefa a seleção de algoritmos em tempo de execução, pois a ferramenta precisa variar a execução de acordo com a definição do banco de dados.

4.3. Implementação e Testes

Inicialmente, foram criadas as classes com os atributos e as assinaturas dos métodos. Testes unitários foram criados, a seguir, para cada classe projetada, contendo testes para todos os métodos da respectiva classe, exceto “gets” e “sets” que não executam nenhuma validação ou processamento adicional além do acesso aos atributos.

Os testes unitários criados foram baseados na especificação e depois da criação dos testes unitários cada método correspondente foi implementado e validado até que se obtivesse sucesso na validação.

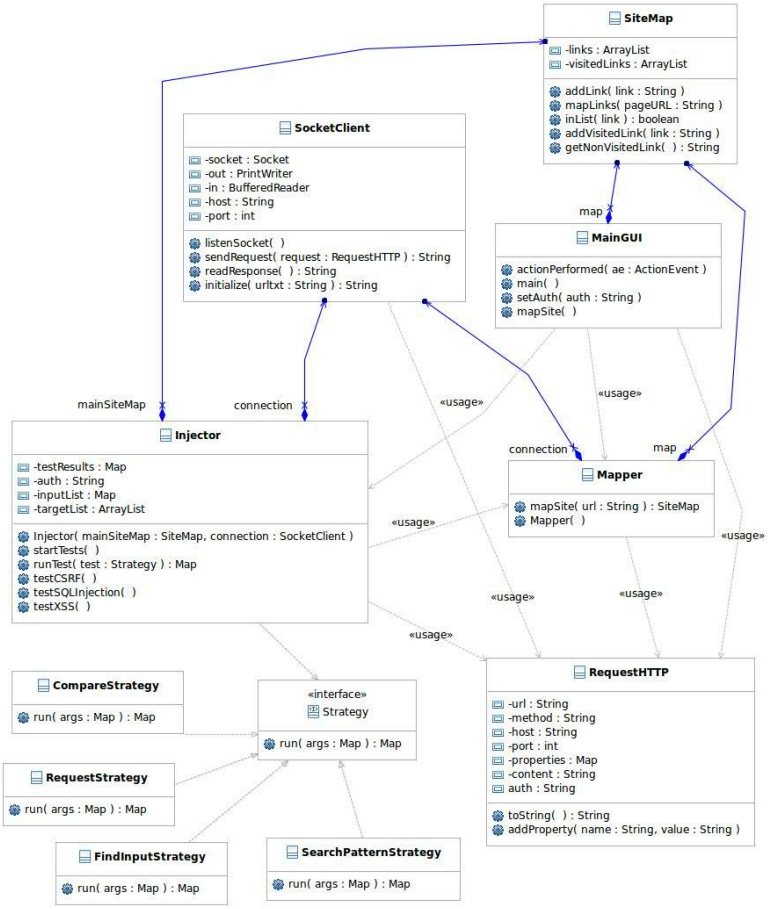

Figura 6. Diagrama de Classes da *J-Attack*

O uso de alguns *stubs* (classe que substitui a classe chamada) foi necessário para possibilitar os testes de integração da ferramenta a cada parte implementada. Os testes unitários foram utilizados, nesta fase, para os testes de regressão, executados a cada nova funcionalidade desenvolvida.

A ferramenta, depois de totalmente integrada, foi validada com um sistema simples. O banco de dados foi populado com casos de teste básicos (para maiores detalhes ver Fernandes *et al* (2010)) para a validação inicial.

5. Estudo de Caso

Como estudo de caso, três aplicações *Web* foram testadas para detectar vulnerabilidades de segurança. Os testes foram projetados com base nas árvores de ataque e os resultados obtidos foram comparados com os resultados de um *scanner* de vulnerabilidade comercial. A ferramenta foi utilizada para realizar, de forma automática, os testes realizados manualmente em Fernandes *et al.* (2010) e se mostrou eficiente para a automatização dos testes.

As três aplicações *Web* selecionadas foram desenvolvidas em Java e têm códigos fonte disponíveis na Internet. A primeira aplicação é um Gerenciador de Relacionamento com o Cliente (CRM) (Eberom 2010), usa MySQL Server 5.0 e servidor de aplicação Tomcat 5.5.28. Além disso, usa tecnologia Hibernate, *framework* Struts e JasperReports para a geração de relatórios. A segunda aplicação é um Gerenciador de Ensino a Distância (Amadeus 2010), usa PostgreSQL 8.4 e servidor de aplicação Tomcat 5.5.28. Também usa Hibernate e AJAX (*Asynchronour Javascript and XML*). A terceira aplicação é um gerenciador de *Call Center* de uma clínica de saúde, usa *framework* Java *Server Faces*, servidor de aplicação JBoss 4.2, MySQL Server 5.0, Hibernate e AJAX. As três aplicações foram selecionadas porque os serviços providos são aplicáveis para áreas comerciais e acadêmicas e usam tecnologias recentes que podem impactar a presença de vulnerabilidades de segurança. Os nomes das ferramentas são preservados, pois não permitem a publicação de resultados relacionados à marca e foram nominadas App1, App2 e App3 sem nenhuma ordem.

Os experimentos foram executados usando um *scanner* de vulnerabilidades comercial, que também por questões de sigilo tem seu nome preservado. Primeiramente, o *scanner* comercial foi utilizado para inspecionar as aplicações. Depois, os testes que usaram os casos de teste projetados com base na árvore de ataque foram executados com a *J-Attack* em cada uma das vulnerabilidades detectadas. Os resultados obtidos com os *scanners* comerciais foram comparados, minuciosamente validados manualmente e classificados como corretos, falsos positivos ou falha de cobertura. O objetivo dos testes era validar a eficácia da *J-Attack* e a validação manual foi necessária para saber se a ferramenta foi incapaz de explorar a vulnerabilidade ou se a vulnerabilidade não existia.

1. Localize um mecanismo de submissão de dados (por exemplo, um campo de busca).
2. Insira um caractere especial (por exemplo, “’”) e submeta a consulta.
3. Observe a resposta normal. Se a aplicação retornar uma mensagem de erro de sintaxe pode ser que esteja vulnerável.
4. Injete uma consulta SQL com uma condição verdadeira adicional para ser concatenada com a consulta original. Isso fará que a consulta retorne um resultado manipulado (por exemplo, ‘ union select load_file (‘/etc/passwd’),1 #)
5. Observe a resposta e verifique se a consulta foi executada e retornou resultados esperados

Figura 7. Caso de Teste para *SQLInjection*

A Figura 7 apresenta um caso de teste para a vulnerabilidade *SQL Injection*. O caso de teste apresentado foi derivado dos passos 5 AND 6, 7 AND 8, 9 e 10 da árvore apresentada na Figura 1.

1. Localize uma entrada de dados de usuário que tenha sido apresentada em uma das páginas da aplicação (i.e., tabela de dados, relatórios de tela, etc).
2. Insira um código *javascript* (por exemplo, <script>alert(‘vulnerável a XSS’</script>) e submeta a requisição.
3. Observe se o *javascript* foi executado (se aparecem mensagens *popups* na tela).

Figura 8. Caso de Teste para *XSS*

A Figura 8 apresenta um caso de teste para a vulnerabilidade *XSS*. O caso de teste apresentado foi derivado dos passos 1, 2 AND 3, 5 AND 6 e 7 da árvore apresentada na Figura 2. Por questões de restrição de espaço, o caso de teste para a vulnerabilidade CSRF foi omitido e pode ser encontrados em Fernandes *et al.* (2010).

6. Resultados e Discussão

Na Figura 9 é apresentada a interface da *J-Attack* quando executava um caso de teste de XSS. As interfaces para os demais tipos de vulnerabilidades foram omitidas devido a restrição de espaço. Em todos os casos, a ferramenta efetua o mapeamento da aplicação Web alvo e depois executa os testes cadastrados no banco de dados.

A aplicação encontrou os mecanismos de entrada de dados do site alvo, depois enviou requisição contendo um *script* para todas as páginas encontradas no passo anterior e verificou se o *script* estava presente na resposta. Neste passo, o ataque se mostrou bem sucedido. Para finalizar, a ferramenta efetuou uma busca pelo *script* injetado anteriormente em todo o site para verificar a existência de XSS armazenado. Na resposta o conjunto requisição-resposta aponta onde a vulnerabilidade foi encontrada.

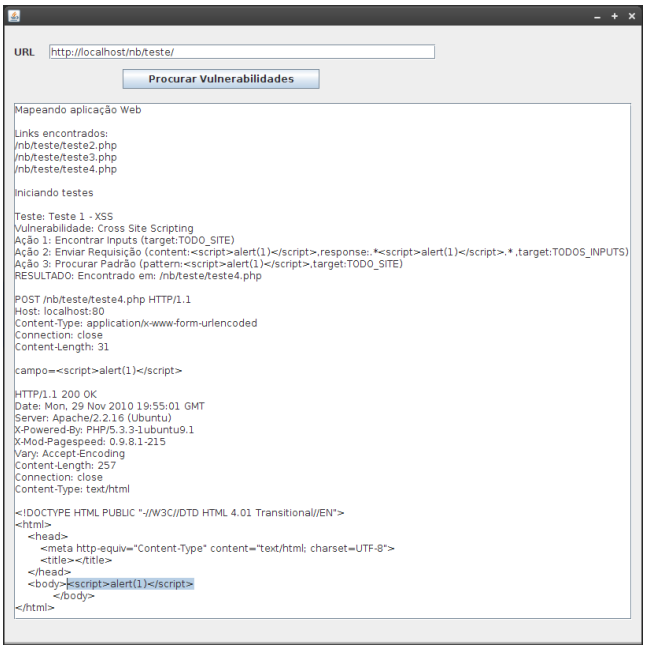

Figura 9. Interface da *J-Attack – Execução de ataque XSS*

A Figura 10 apresenta a execução da ferramenta com um teste de SQL Injection carregado. Primeiramente, a aplicação encontrou os mecanismos de entrada de dados do site alvo, em seguida enviou requisição com uma apóstrofe para as páginas encontradas no passo anterior. Depois a ferramenta enviou uma nova requisição contendo o valor "1" para as mesmas páginas. Finalmente, foram comparadas as respostas das duas requisições anteriores e verificado que houve diferença, pois uma das respostas retornou uma mensagem de erro do banco de dados e o ataque não pode ser feito por este cenário.

Considerando as três aplicações inspecionadas, o *scanner* comercial detectou 3 XSS, 4 SQL Injection e 15 CSRF. Os resultados foram validados manualmente e 2 vulnerabilidades XSS e 12 CSRF foram detectadas corretamente, as demais 8 vulnerabilidades eram falsos positivos. Além dessas vulnerabilidades, outras 9 novas

vulnerabilidades foram detectadas quando os casos de teste baseados na árvore de ataque foram aplicados. Essas novas vulnerabilidades se configuram como falha na cobertura da ferramenta comercial. A Tabela 1 apresenta estes resultados.

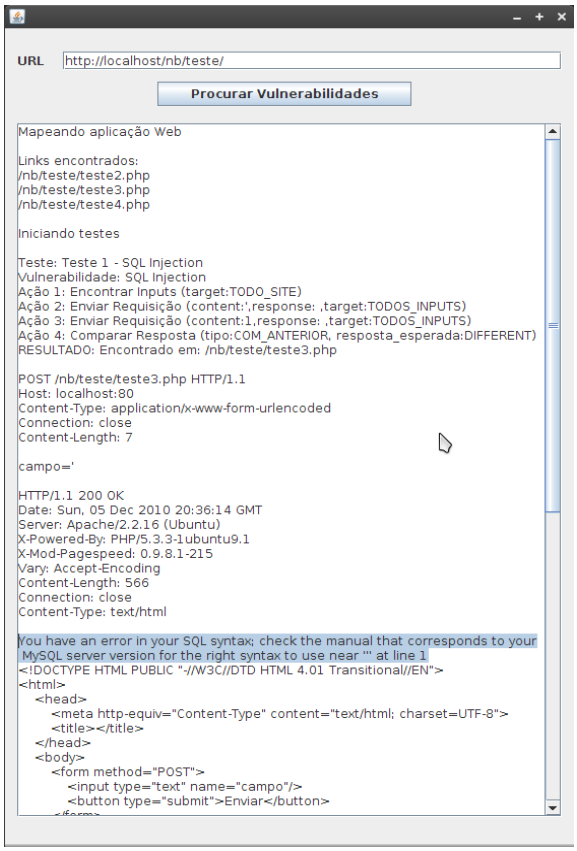

Figura 10. Interface da *J-Attack – Execução de ataque SQL Injection*

Os resultados por aplicação podem ser encontrados em Fernandes *et al*.(2010) e não foram apresentados aqui por restrição de espaço. A *J-Attack* em seus testes iniciais conseguiu efetuar os ataques planejados nos casos de teste e em casos de falsos positivos acusou a impossibilidade do ataque. Ao reexecutar os teste manuais utilizando a *J-Attack* algumas falhas de implementação foram encontradas e, neste momento, estão sendo corrigidas para gerar uma versão que se possa disponibilizar para uso de terceiros.

Tabela 1. Resultados dos Testes por tipo de vulnerabilidade

	XSS	SQLInj	CSRF	Total
Detectada pelo *scanner* comercial	3	4	15	22
Detectada corretamente	2	0	12	14
Falso positive	1	4	3	8
Novas vulnerabilidades (falha de cobertura)	0	0	9	9

7. Considerações e Trabalhos Futuros

A ferramenta desenvolvida possibilita a automatização de testes de vulnerabilidades de segurança em aplicações *Web*. A versão inicial da ferramenta trabalha apenas com algumas ações básicas, mas já é capaz de cobrir muitos tipos de testes. A ferramenta

apresenta uma grande flexibilidade devido à sua operação de acordo com o banco de dados. O banco de dados pode ser preenchido com vários tipos de ataques e preferencialmente com uma grande quantidade de variações, pois isto é importante para ataques como XSS e SQL Injection e outros tipos de ataques que usam a injeção como meio de ataque.

A ferramenta foi desenvolvida tendo como objetivo a escalabilidade. Sendo assim, é possível expandir a ferramenta para que ela passe a cobrir mais vulnerabilidades de segurança. Novos tipos de ações podem ser definidos e adicionados ao padrão Strategy do projeto. Novos testes e vulnerabilidades podem ser adicionados ao banco de dados.

A interface gráfica da ferramenta se mostrou bastante simples, facilitando o uso, porém pode ser melhorada e mecanismos de parametrização de execução podem ser adicionados, possibilitando a limitação de execução para testes específicos, por exemplo.

A adição de uma sessão de controle do banco de dados na ferramenta também seria interessante para evitar a necessidade de manipulação de dados diretamente no sistema gerenciador de banco de dados. A ferramenta também precisa ser testada e validada com outras aplicações *Web*.

Referências

Acunetix (2007), "70% of websites at immediate risk of being hacked!" Disponível em http://www.acunetix.com/news/security-audit-results.htm, acesso março/ 2011.

Acunetix (2011), "Acunetix Web Vulnerability Scanner" Disponível em http://www.acunetix.com/vulnerability-scanner/, acesso março/2011.

Amadeus (2010). "Portal do software público brasileiro". Disponível em http://www.softwarepublico.gov.br/ver-comunidade?community_id=9677539. Último acesso em dezembro/2010

Auger, R. (2010) "The Cross-Site Request Forgery (CSRF/XSRF) FAQ". Disponível em http://www.cgisecurity.com/csrf-faq.html, acesso novembro/2010.

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

Bau, J., Bursztein, E., Gupta, D, Mitchell, J. (2010) "State of the Art: Automated Black-Box Web Application Vulnerability Testing". IEEE Symposium on Security and Privacy, Oakland, USA. Páginas 332-345.

Beck, K. (2002) "Test Driven Development: By Example**"**. (5ª edição). Addison-Wesley Professional.

Browser Scope Project (2010) "Security Test Results**"**. Disponível em http://browserscope.org/?category=security, acesso novembro/2010.

Christey, S. and Martin, R.A. (2007) "Vulnerability type distributions in CVE", Technical Report 1.1, vol. 10, p.04, MITRE Corporation, May 2007.

Eberom (2010). “A CRM and Project Management Tool”. Disponível em http://sourceforge.net/projects/eberom/. Último acesso em dezembro/2010.

Fernandes, P. C. S., Basso, T., Moraes, R., Jino, M. (2010) “Attack Trees Modeling for Security Tests in Web Applications” 4th. Brazilian Workshop on Systematic and Automated Software Testing (SAST). Natal - RN, Brasil.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. (1999) “RFC 2616: Hypertext Transfer Protocol - HTTP/1.1”. Disponível em: http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html, acesso novembro de 2010.

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

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) “Design Patterns: Elements of Reusable Object-Oriented Software”. 1st. Ed. Estados Unidos da América: Addison-Wesley.

Halfond, W. G. J., Viegas, J., Orso, R. (2006) “A Classification of SQL-Injection Attacks and Countermeasures”. International Symposium on Secure Software Engineering – ISSSE 2006, Arlington, Virginia, 2006.

Havij. “ITSecTeam”. Disponível em http://www.itsecteam.com/en/projects/project1.htm, acesso abril/2011.

HP (2011), “HP WebInspect” Disponível em https://h10078.www1.hp.com/ cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-201-200%5E9570_4000_100__, acesso março/2011.

IBM (2011), “IBM Rational AppScan” Disponível em http://www-01.ibm.com/software/ awdtools/appscan/, último acesso março/2011.

NTA Monitor (2008), Annual Web Application Security Report 2008, Disponível em http://www.nta-monitor.com, último acesso fevereiro/2011.

OWASP (2010) “Top 10 Project”. Disponível em http://www.owasp.org/ index.php/Category:OWASP_Top_Ten_Project, acesso novembro de 2010.

Schneier, B (1999) “Attack Trees: Modeling Security Threats”, Dr. Dobb’s Journal.

Singhal, A., Winograd, T. and Scarfone, K. (2007) “Guide to Secure Web Services: Recommendations of the National Institute of Standards and Technology,” Report, National Inst. of Standards and Tech, US Dep.of Commerce, 2007, pp. 800–95.

Snake, R. “XSS (Cross Site Scripting) - Cheat Sheet Esp: for filter evasion”. Disponível em http://ha.ckers.org/xss.html, acesso novembro/2010.

SqlHelper “Stored Procedure and Documentation Tool”. Disponível em http://www.pikauba.com/sqlhelp/details.htm, acesso abril/2011.

SqlMap “Automatic Sql Injection and Database Takeover Tool”. Disponível em http://sqlmap.sourceforge.net, acesso abril/2011.

Vieira, M., Antunes, N., Madeira, H. (2009) "Using Web Security Scanners to Detect Vulnerabilities in Web Services". Pract.Exp.Report. DSN 2009, Lisboa, Portugal.