Mestrado em Engenharia Informática
Dissertação/Estágio
Relatório Final

Assessing Web Services Robustness and Security Using Malicious Data Injection

Pedro Ricardo Saraiva Bento
prbento@student.dei.uc.pt

Orientador:
Prof. Doutor Nuno Laranjeiro
Data: 02 de Setembro de 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Assessing Web Services Robustness and Security Using Malicious Data Injection

Relatório Final

Autor:
Pedro Ricardo Saraiva Bento

Orientador:
Prof. Doutor Nuno Laranjeiro

Júri:
Prof. Doutor David Fonseca Palma
Prof. Doutor Paulo José Osório Rupino da Cunha

Setembro 02, 2015

Sumário

A tecnologia *Web Services* permite ligar aplicações criadas em diferentes plataformas, tendo atingido grande popularidade. Nos últimos anos, o uso desta tecnologia tem aumentado consideravelmente, não só como suporte a ambientes críticos de negócio, mas também em ambientes onde a robustez e segurança dos serviços é vital. Nestes ambientes, a presença de um problema de robustez ou uma vulnerabilidade de segurança pode traduzir-se em perdas a nível financeiro e/ou na reputação do fornecedor do serviço. A falta de metodologias e ferramentas adequadas para a deteção destes problemas é um dos fatores que contribui para a situação atual, onde os serviços falham na presença de entradas inválidas ou maliciosas.

Nesta dissertação é discutido o estado da arte em robustez e segurança em *Web Services* sendo proposta uma abordagem para deteção de falhas desta área. Esta baseia-se na introdução, em tempo de execução, de um conjunto de *input*s inválidos e maliciosos num serviço sob teste. Contrariamente à abordagem clássica para deteção destes problemas, as interfaces das aplicações sob teste na abordagem apresentada, são as de contacto com serviços externos, em particular com a base de dados. Deste trabalho resulta também a criação de uma ferramenta de testes, possibilitando a classificação do nível de segurança e robustez de um serviço.

Palavras-Chave: *Web Services*, Segurança, *SQL Injection*, *Second-Order SQL Injection*, Testes de Robustez.

Abstract

The Web Services technology allows us to connect applications built on different platforms, reaching great popularity. In the last years, the use of this technology has increased considerably, not only to support the critical business environments, but also in environments where robustness and safety of services is vital. The presence of a robustness problem or a security vulnerability can be translated into substantial losses in financial terms and/or reputation of the service provider. The lack of methodologies and tools for the detection of these problems is one of the factors contributing to the current situation where services fail in the presence of invalid or malicious inputs.

This dissertation discusses the state of the art of robustness and security in Web Services and proposes an approach for detection of this type of situations. This is based on the introduction, at runtime, of a set of invalid and malicious inputs to a service under test. Unlike the classical approach for detecting these problems, the interfaces of the applications under test in the developed approach, are those that contact with external services, in particular its interface with the database. This work also involves the creation of a testing tool, allowing the classification of the level of robustness and security of a service.

Keywords: Web Services, Security, SQL Injection, Second-Order SQL Injection, Robustness Testing.

Agradecimentos

Ao meu orientador, Professor Doutor Nuno Laranjeiro, por todo o apoio, disponibilidade, paciência e conhecimento académico que me proporcionou ao longo deste percurso. Muito obrigado por tudo.

Aos prezados júris, Professores Doutores David Fonseca Palma e Paulo José Osório Rupino da Cunha que pelas suas correções e sugestões dadas na defesa intermédia contribuíram muito para o melhoramento desta dissertação. Muito obrigado pelos vossos sábios conselhos.

A todos os professores que fizeram parte da minha vida académica e que com os seus conhecimentos fizeram de mim a pessoa que sou hoje.

À Faculdade de Ciências e Tecnologia da Universidade de Coimbra em especial ao Departamento de Engenharia Informática que me acolheu e me proporcionou a minha formação.

À cidade de Coimbra, ao seu espírito académico único e a todos os meus amigos que fizeram parte integrante das minhas vivências. Em especial ao Diogo, Rafael e aos meus colegas de casa por me terem acompanhado durante todo este meu percurso, desde o primeiro dia. Obrigado pela presença constante. Jamais vos esquecerei.

A toda a minha família, avós, tios, padrinhos e primos em especial ao meu primo Nuno pelo apoio e presença. Foram o suporte desta caminhada.

À Vanessa que com muito amor, apoio e companheirismo foi um fator facilitador para o meu bem-estar e força para continuar. Muito obrigado.

Aos meus pais amados, aos quais devo tudo o que sou hoje, um imenso e eterno agradecimento por todo o apoio incondicional dado durante toda a minha vida, por me terem dado a possibilidade de concretizar os meus sonhos e por nunca me ter faltado nada. Tenho um imenso orgulho em ser vosso filho e uma dívida eterna pela formação humana e académica que me proporcionaram. Um especial obrigado.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

CORBA	*Common Object Request Broker Architecture*
DAST	*Dynamic Application Security Testing*
HTTP	*Hyper-Text Transfer Protocol*
JDBC	*Java Database Connectivity*
MAFALDA	*Microkernel Assessment by Fault Injection Analysis and Design Aid*
POA	Programação Orientada a Aspetos
SGBD	Sistemas de Gestão de Bases de Dados
SOAP	*Simple Object Access Protocol*
SQL	*Structured Query Language*
WSDL	*Web Service Definition Language*
XML	*eXtensible Markup Language*

Capítulo 1 - Introdução

A dissertação aqui apresentada surge da necessidade crescente da procura de métodos e abordagens que permitam mitigar os riscos de ataques por *Structured Query Language* (SQL) *Injection* a *Web Services*, podendo estes ter causas devastadoras aquando do sucesso do referido ataque. A mitigação destes riscos pode ser alcançada através de um maior investimento em *scanners* e abordagens de segurança e robustez, que permitam expor vulnerabilidades existentes nas aplicações *Web*, podendo ser consequentemente corrigidas para impedir que se tornem alvos de *hackers* em busca de informação valiosa. De modo a evitar este tipo de situações, discute-se neste relatório o estado da arte referente à robustez e segurança nos *Web Services* e uma abordagem para deteção deste tipo de falhas. As vulnerabilidades de segurança a que esta abordagem dá resposta são as sensíveis aos ataques por SQL *Injection*, não detetando por isso outro tipo de falhas.

Esta dissertação foi produzida no âmbito da disciplina dissertação/estágio em Comunicações, Serviços e Infraestruturas do Mestrado em Engenharia Informática, pelo aluno Pedro Ricardo Saraiva Bento e orientada pelo Prof. Doutor Carlos Nuno Bizarro e Silva Laranjeiro, nas instalações do Departamento de Engenharia Informática, da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

1.1- Âmbito e Abordagem

Em ambientes baseados em *Web Services*, a presença de um problema de robustez ou vulnerabilidade de segurança pode causar danos irrecuperáveis, tais como a eliminação de toda a informação existente numa base de dados, fuga de informação confidencial e negação de serviço. Atualmente, as abordagens e ferramentas para deteção de problemas de robustez focam-se nas interfaces públicas dos serviços, esquecendo que a natureza dos *Web Services* faz com que eles estejam muitas vezes ligados a outros serviços, incluindo Sistemas de Gestão de Bases de Dados (SGBD). Esta dissertação foca-se, precisamente, neste problema, ou seja, na interação de um serviço com um SGBD, e tenta definir uma abordagem que permita compreender o comportamento de um serviço na presença de dados inválidos ou maliciosos. Apesar do âmbito desta dissertação se centrar principalmente em *Web Services* que recorrem a sistemas que comunicam com bases de dados, a técnica usada pode ainda ser aplicada em qualquer outro ambiente, diferindo no comportamento observável. Este foco está relacionado com o facto de que os ataques por SQL *Injection* serem os mais frequentes neste tipo de ambientes [1].

A abordagem proposta neste documento atua entre o sistema sob teste e os sistemas externos que usa, em particular o SGBD e tem como objetivo revelar problemas de robustez e segurança existentes na aplicação. A particularidade do método é que não requer alterações no sistema (base de dados, cliente ou aplicação), sendo apenas necessário substituir o componente que permite interagir como SGBD, por exemplo, o *driver Java Database Connectivity* (JDBC). Um dos objetivos é que a ferramenta criada e que implementa a abordagem seja o menos intrusiva possível.

Os objetivos desta dissertação são os seguintes:

- **Definição de uma abordagem para deteção de problemas de robustez e segurança:** A abordagem envolve a injeção de dados inválidos e maliciosos nos

resultados provenientes dos acessos à base de dados. Esta deve ser prática e o menos intrusiva possível;

- **Criação de uma ferramenta que implemente a abordagem definida:** este objetivo baseou-se na criação de uma ferramenta de testes que permitiu testar o nível de robustez e segurança do serviço. Assim conseguiu-se detetar as falhas existentes no sistema alvo, de modo a possibilitar a sua correção evitando deste modo, que ataques maliciosos à aplicação sejam bem sucedidos;
- **Aplicação da abordagem a um ambiente realista:** para além da construção da ferramenta, um dos principais objetivos consistiu em usar a ferramenta em serviços reais para executar todos os testes de robustez e segurança que foram definidos durante este ano letivo. Deste modo foi possível verificar se estes serviços continham vulnerabilidades. Seguiu-se a análise e discussão dos resultados obtidos.

1.2 – Organização

O estado da arte é apresentado no Capítulo 2, incluindo: uma breve introdução aos *Web Services* e à sua interação com bases de dados; uma breve explicação sobre os métodos de testes de robustez existentes; e por fim, para além de uma explicação mais aprofundada sobre dois tipos de vulnerabilidades (SQL *Injection* e *Second-Order* SQL *Injection*), são descritas também algumas abordagens cuja finalidade é a deteção destas falhas.

No Capítulo 3 apresentam-se algumas técnicas para injeção de valores inválidos ou maliciosos, incluindo uma explicação da abordagem definida e dos testes que foram executados e o método utilizado para a avaliação dos testes de robustez e de segurança.

O plano do trabalho produzido ao longo do ano letivo é demonstrado no Capítulo 4.

Finalmente, o Capítulo 5 conclui esta dissertação.

Capítulo 2 - Estado da Arte

No presente capítulo são discutidos os conceitos básicos sobre *Web Services* e o seu modo de interação com bases de dados (Secção 2.1). São ainda descritos, na Secção 2.2, os *black box tests*, *white box tests* e *gray box tests*, necessários para a compreensão do conteúdo existente neste relatório. Segue-se, na Secção 2.3, uma introdução aos testes de robustez, que estiveram presentes na aplicação elaborada durante o presente ano letivo, de modo a comprovar a robustez de aplicações sujeitas a esta ferramenta. Por fim, na Secção 2.4 são descritos dois tipos de testes de segurança (SQL *Injection* e *Second-Order* SQL *Injection*), assim como métodos usados neste tipo de testes, sendo o último o principal foco desta dissertação.

2.1 - Web Services

A tecnologia de *Web Services* é muitas vezes uma solução utilizada em Integração de Sistemas que permite a comunicação entre diferentes aplicações, quer sejam aplicações novas ou antigas [2]. Com esta técnica (*Web Services*) existe a possibilidade das aplicações enviarem e receberem dados entre elas, apesar de conterem linguagens diferentes. Este processo é feito através da transformação dos dados em formato *eXtensible Markup Language* (XML), que permite estabelecer ligações entre aplicações completamente diferentes. Depois de sofrerem esta transformação, os dados são encapsulados pelo protocolo *Simple Object Acess Protocol* (SOAP) e tipicamente enviados através do protocolo *Hyper Text Transfer Protocol* (HTTP) [3].Este modo de funcionamento permite a comunicação entre aplicações heterogéneas [4]. *Web Services* têm neste momento uma grande adesão, tendo já atingido uma grande variedade de setores, desde lojas *online* a corporações de media.

Tal como foi referido, o protocolo SOAP [5] é usado para realizar a troca de mensagens XML [6] entre as duas extremidades, consumidor (*Service Consumer*) e fornecedor (*Service Provider*), tipicamente usando o protocolo HTTP ou *Hyper Text Transfer Protocol Secure* (HTTPS). A cada ligação o consumidor (cliente) envia um pedido SOAP para o fornecedor (servidor). Depois de processar toda a informação enviada na mensagem de pedido, o servidor envia uma resposta mais uma vez no formato SOAP com o resultado obtido. Todo este funcionamento pode ser observado na Figura 2.1, onde se representa um ambiente típico baseado em *Web Services*.

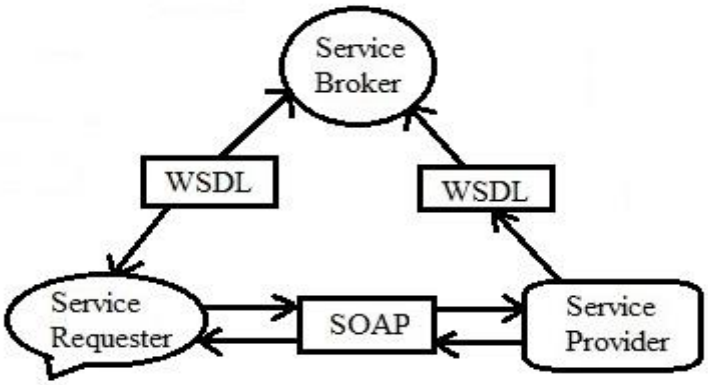

Figura 2.1 – Funcionamento típico de um *Web Service* (adaptado de [7]).

Os *Web Services* fazem uso de um documento XML, utilizado para descrever o serviço, cujo nome é *Web Service Definition Language* (WSDL) [3]. Este documento para além de oferecer uma descrição do *Web Service*, possibilita também a sua localização e fornece informação sobre os métodos disponíveis pelo serviço. Como é possível observar na Figura 2.1 existe também um *broker* que fornece as informações sobre os *Web Services* disponíveis ao *Service Consumer*, ou seja, o broker é responsável por enviar ao *Service Consumer* todas as informações sobre outros *Web Services* existentes, como por exemplo o tipo de operações que contêm.

A maioria dos *Web Services* recorrem a bases de dados de modo a poderem guardar a informação necessária para o funcionamento do serviço. Estes serviços têm o nome de *Database Web Services* [8], criados devido à necessidade crescente de aceder a dados através de interfaces de *Web Services* em ambientes heterogéneos. Na Figura 2.2 é apresentado um exemplo de um destes casos.

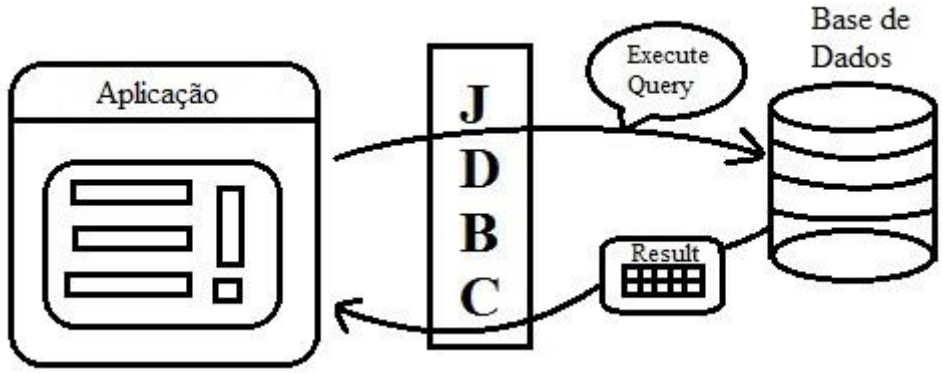

Figura 2.2 – Funcionamento entre um *Web Service* e uma Base de Dados.

A ligação entre a base de dados e o *Web Service*, como mostra a Figura 2.2, é feita através de um componente *middleware*, por exemplo, na linguagem Java, *driver* JDBC [9] que contém a definição de métodos que permitem estabelecer esta ligação. Este componente permite que a aplicação envie instruções SQL para a base de dados e retorne os seus respetivos resultados. Estas instruções SQL recebidas pelo componente, são processadas e reencaminhadas para a base de dados. Esta ao receber o pedido vai

processá-lo retornando um resultado correspondente à pesquisa realizada. Este *ResultSet* vai ser guardado e traduzido pelo componente *middleware* atrás referido, que será posteriormente retornado para a aplicação, com todo o conteúdo encontrado com a pesquisa efetuada.

2.2 – Métodos de Testes

Como acima referido existem três grupos de testes de *software*, sendo estes, *Black Box*, *White Box* e *Gray Box Tests* [10].

1) Black Box Tests

Na Figura 2.3 são ilustrados os testes *Black box*.

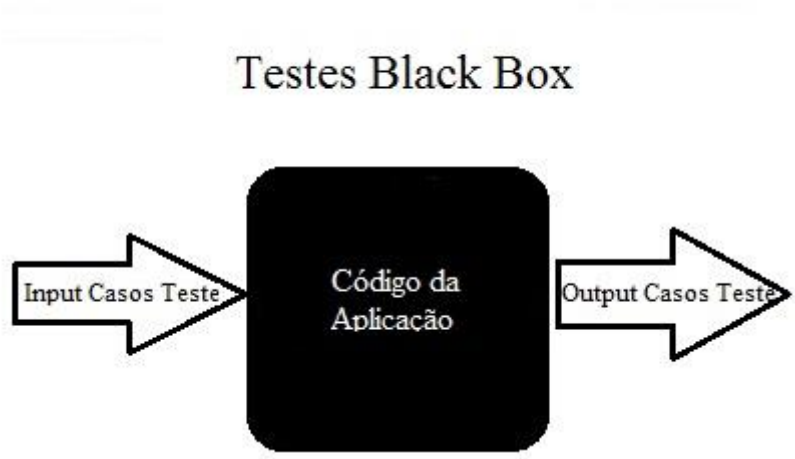

Figura 2.3 – Abordagem *Black Box Tests*. Adaptado de [11].

Neste tipo de testes, o analista não tem acesso ao código fonte e desta forma desconhece a estrutura interna da aplicação, ou seja, os testes a serem realizados estão relacionados com os requisitos da aplicação e com as ações que esta elaborar. Este tipo de testes envolve a injeção de parâmetros válidos e inválidos, e objetiva a observação do comportamento da aplicação quando submetida a diferentes *inputs*. Estes testes são projetados com o intuito de descobrir erros na aplicação e demonstrar que as funções existentes no *software* em questão, estão operacionais, isto é que as entradas são devidamente aceites e as suas saídas corretamente produzidas, e que a integridade das informações externas é mantida. Um dos objetivos destes testes é contrariar as ações requeridas pelo serviço, por exemplo, quando a aplicação pede para inserirmos o número de identificação fiscal e nós introduzimos um número negativo ou quando a aplicação pede a introdução de uma data de nascimento, à qual respondemos com uma data futura, entre outros [12]. Este tipo de testes possui vantagens e desvantagens.
Algumas das vantagens são:

- Eficiente quando usado em sistemas de grandes dimensões [10];
- Equilíbrio e imparcialidade dos testes, caso o analista e o programador sejam diferentes [12];
- Não é exigido ao analista o conhecimento da implementação da aplicação, nem da língua de programação usada [13].

No entanto, este tipo de testes possui algumas limitações/desvantagens, tais como:

- Falta de cobertura que os testes podem ter na aplicação, sendo esta um fator limitante, visto não se possuir informação sobre a aplicação [10];
- Falta de informação sobre a estrutura do sistema não permite a execução de testes em funções específicas [13];
- Possibilidade de repetição de testes, já efetuados anteriormente pelo programador, caso o analista e o programador sejam diferentes.

Este foi o tipo de testes usados no trabalho desenvolvido, visto que a técnica implementada na ferramenta criada permite que os testes sejam aplicados sem se conhecer a aplicação alvo e o analista da aplicação não ser o mesmo que o programador.

2) White Box Tests

Relativamente aos testes *White Box*, a diferença em relação aos testes *Black Box* referidos anteriormente, é que o analista conhece o código fonte, como podemos observar na Figura 2.4, tem acesso a este desde o início, logo conhece a estrutura interna da aplicação e desse modo pode escolher os alvos dos testes. Consequentemente, existe um maior controlo e uma maior facilidade em entender o comportamento da aplicação e das suas funções [14], o que permite uma maior facilidade na deteção de erros lógicos no programa. Estes testes incluem a análise de fluxos de dados, controlo e informação, práticas de código, tratamento de exceções e erros no sistema, cujo objetivo é observar o comportamento da aplicação ao estar sujeita a ações que afetam o normal funcionamento da mesma.

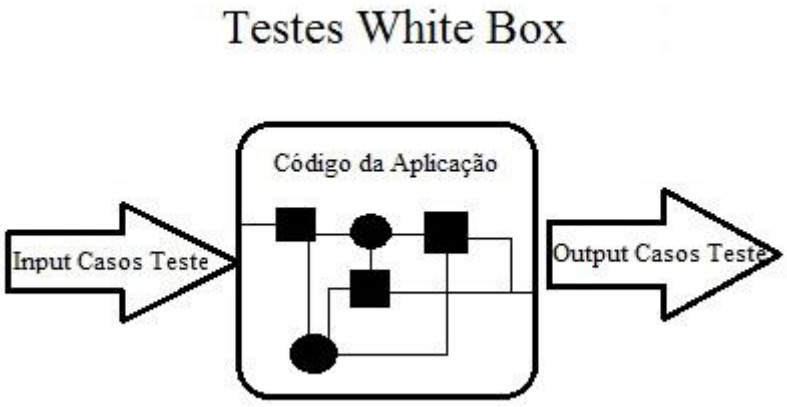

Figura 2.4 – Abordagem *White Box Tests.* Adaptado de [11].

Assim como os testes *Black Box*, os testes *White Box* possuem vantagens e desvantagens. Algumas das vantagens são:

- Possibilidade de analisar o código linha a linha, de modo a excluir toda e qualquer hipótese de existência de erros e/ou vulnerabilidades [13];
- Possibilidade de eliminar código desnecessário (*e.g.*, caminhos desnecessários entre operações, código que não é utilizado) [10];
- Possibilidade de testar a aplicação enquanto esta ainda se encontra em desenvolvimento.

Algumas das desvantagens são:

- Dispendioso tendo em conta que requer a presença de um analista com elevada experiência e conhecimento a nível de programação para a criação dos testes [12];
- A análise exaustiva do código, linha a linha, depende do tempo e orçamento do projeto, o que pode fazer com que existam caminhos da aplicação que não são testados, resultando na possibilidade de existência de erros que não são detetados [13];
- O tempo necessário que este tipo de testes requer é muito elevado [10].

O tipo de testes mencionado também foi usado algumas vezes durante o desenvolvimento da ferramenta, de modo a serem descobertos erros existentes que comprometam o bom funcionamento da mesma.

Um dos métodos usados em *White Box Testing*, recorre à análise de código estático especialmente em revisões de código. Pesquisas feitos ao longo dos anos comprovaram que a análise do código é uma das melhores maneiras para detetar e eliminar *bugs* existentes [15]. Tendo-se observado que esta deteção nem sempre é possível através de profissionais experientes, optou-se pela implementação de analisadores estáticos, ou seja: programas que percorrem o código existente numa aplicação, à procura de exceções particulares e erros que dificilmente são detetados, enquanto o código fonte se encontra estático (sem estar a correr), através de técnicas como *Data Flow Analysis* e *Taint Analysis* [16]. Estas ferramentas servem de suporte aos programadores para ajudar a combater a existência de *bugs* nas aplicações em desenvolvimento.

3) Gray Box Tests

Além dos testes apresentados acima, existem ainda os *Gray Box Tests* (ver Figura 2.5).

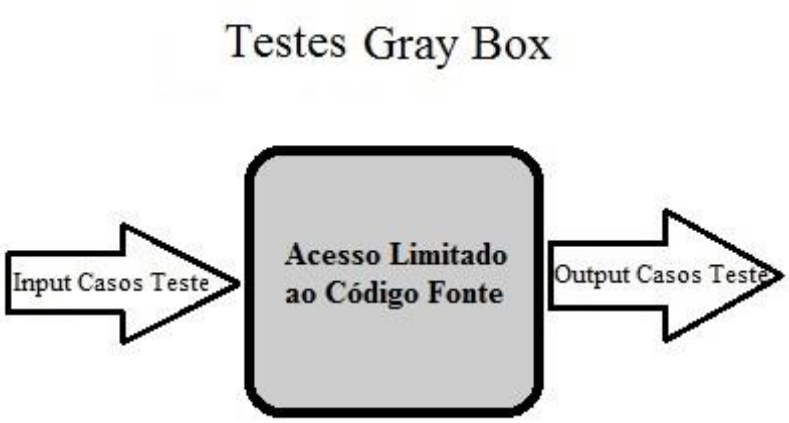

Figura 2.5 – Abordagem *Gray Box Tests*.

Estes testes são uma combinação de *White Box Tests* e *Black Box Tests*, ou seja, não são considerados testes *Black Box* porque o analista tem acesso a algumas operações internas do sistema, mas também não são considerados testes *White Box* já que estes têm acesso limitado ao código fonte da aplicação. Os *Gray Box Tests* baseiam-se num conhecimento limitado sobre a aplicação. Estes são usados para obter os benefícios de

Black Box Testing e *White Box Testing*, para tratar situações de testes mais complexos de forma mais eficiente [17].

Um método *Gray Box Testing*, para fazer testes de desempenho em *runtime*, de fácil implementação, através da utilização de *software* e hardware *Commercial Off the Shelf*, é apresentado por Coulter [18]. O autor usa um método modificado de *Gray Box Testing*, devido ao facto de o método original não suportar testes em tempo de execução, ou testes ao nível do sistema. O foco deste artigo é incluir a verificação e validação do desempenho em tempo real na metodologia deste tipo de testes, sem alterar as intenções do método original dos testes *Gray Box*.

A ferramenta desenvolvida recorreu apenas a testes *Black Box*, para realizar os testes, visto que não necessita de qualquer conhecimento inicial sobre o código da aplicação alvo, para poder aplicar os testes definidos.

2.3 – Testes de Robustez

O objetivo destes testes é observar o comportamento de uma aplicação na presença de *input*s inválidos/inesperados. Espera-se com estes, expor a aplicação alvo a problemas de robustez, através da ocorrência de exceções que exponham erros de programação ou de *design*. As aplicações são avaliadas consoante o número e tipo de erros detetados após a fase de testes estar concluída. Existem vários estudos que recorrem a este tipo de testes para avaliar a robustez de sistemas operativos, sendo [19] e [20] os que apresentam as ferramentas de teste de robustez mais conhecidas para os testes em questão, denominadas Ballista e *Microkernel Assessment by Fault Injection Analysis and Design Aid* (MAFALDA) respetivamente.

Ballista é uma ferramenta que é especializada em realizar testes de robustez (*Black Box*) em sistemas operativos [20], usando uma combinação de parâmetros válidos e inválidos usados nas chamadas de sistema, para bloquear o sistema operativo em questão. Estes parâmetros são retirados aleatoriamente de uma base de dados que contém todos os testes predefinidos a serem efetuados pela ferramenta. A robustez da aplicação é avaliada segundo a escala CRASH, que apresenta as seguintes classificações: *Catastrophic* (o sistema operativo fica corrompido; a máquina desliga ou reinicia e o problema persiste), *Restart* (a aplicação bloqueia e precisa de ser terminada à força), *Abort* (aplicação termina de forma anormal (abortou por si)), *Silent* (nenhum erro/exceção foi assinalado, quando deveria ter sido) e *Hindering* (o código de erro retornado não é o correto).

As interfaces *Portable Operating System Interface for Unix* foram os primeiros alvos desta ferramenta de testes de robustez, tendo sido mais tarde adaptada para sistemas operativos Windows [21]. Neste estudo são apresentados os resultados da execução de testes, que envolvem o tratamento de exceções sobre várias funções e chamadas ao sistema, resultantes da execução da ferramenta de testes de robustez Ballista. Os sistemas operativos sujeitos a estes testes foram o Windows 95, 98, CE, NT, 2000 e Linux. Apesar destes testes resultarem na deteção de vários problemas de robustez em todos os sistemas operativos, os mais graves foram detetados no Windows 95, 98 e CE, cujos sistemas operativos deixaram de funcionar por completo. Esta ferramenta também foi adaptada para várias implementações de *Common Object Request Broker Architecture* (CORBA) [22], assim como a sua classificação de falhas, de maneira a caraterizar melhor o contexto CORBA.

MAFALDA é uma ferramenta que permite a caraterização do comportamento de *microkernels* na presença de falhas [19]. Tal como a ferramenta Ballista, a MAFALDA é uma ferramenta desenvolvida com o intuito de testar sistemas operativos. Esta suporta injeção de falhas tanto nos parâmetros às chamadas de sistema como nos segmentos de memória. Esta corrupção é feita através da inversão de um valor de um bit dentro de uma sequência de bytes, *bit-flip*. Apesar desta mesma possuir estes dois métodos de injeção de falhas, apenas um é considerado nos testes de robustez, injeção de falhas nos parâmetros às chamadas de sistema. A robustez da aplicação é avaliada segundo a seguinte classificação: Erro detetado (o sistema operativo deteta o erro e reporta-o para a camada de aplicação), Morte do *kernel* (o *microkernel* congela, ou seja, o *microkernel* entra em *loop* infinito, *dead lock* ou espera um evento que não existe), Erro propagado (o erro não é detetado pelos mecanismos de deteção de erros do *microkernel* e por isso o erro propaga-se para a camada de aplicação).

Assim como a ferramenta Ballista, MAFALDA também sofreu uma evolução para *Microkernel Assessment by Fault Injection Analysis and Design Aid for Real Time* (MAFALDA-RT) *systems* [23], sendo melhorada com o objetivo de ampliar a análise de comportamentos de uma aplicação sujeita a falhas, de modo a incluir medidas de tempo de resposta. Esta ferramenta foi também usada num estudo [24], que tinha como objetivo, melhorá-la, de modo a permitir a caraterização de *middleware* baseado em CORBA. Este género de testes teve tanto sucesso que começaram a surgir estudos focados em componentes específicos de um sistema operativo [25]. Neste estudo, foi investigada a robustez das propriedades dos *driver*s dos dispositivos do Windows. *Kernels* mais recentes de sistemas operativos têm tendência a ser cada vez mais finos, devido à existência dos *driver*s dos dispositivos. A estes são atribuídas as respetivas tarefas existentes, sendo que grande parte dos *crashes* do sistema devem-se a *bugs* existentes nestes *driver*s dos dispositivos. O mesmo estudo conclui, baseado na robustez das propriedades do Windows (XP, Server 2003 e Vista), que na maioria dos casos, as versões dos sistemas operativos testadas, tendem a ser vulneráveis, visto que sofrem *crashes* de sistema quando são sujeitos à injeção de dados maliciosos específicos, o que nos permite realçar a importância dos testes de robustez.

Um dos primeiros exemplos de testes de robustez aplicados a *Web Services* pode ser encontrado no trabalho publicado por Siblini e Mansour [26]. Os autores propõem o uso da análise da mutação de parâmetros como técnica de testes de robustez para aplicar em *Web Services*, onde o ficheiro que descreve um *Web Service* (definido através do uso de WSDL), é inicialmente dividido. Aqui são introduzidos operadores de mutação, resultando na existência de diversos documentos modificados, que serão usados para testar o serviço. O conjunto de mutações que podem ser aplicadas à interface do documento, está relacionada com as chamadas de operações, mais precisamente, mensagens de entrada, mensagens de saída e respetivos tipos de dados. Os autores do estudo em questão, tentaram focar-se nos erros dos programadores responsáveis pelo desenvolvimento do *Web Service*, erros comuns estes que ocorrem ao definir, implementar e usar estas interfaces. Apesar de ser uma abordagem diferente, não é a mais indicada, visto que os parâmetros de operadores de mutação, são muito limitados e consistem basicamente em adicionar, apagar, trocar elementos ou criar tipos complexos a nulo.

Em [27] é usada uma abordagem que recorre à alteração do esquema XML através da utilização de operadores de perturbação. É apresentada uma forma de criar um esquema XML virtual, que é utilizada quando não existem esquemas XML e definido um modelo

formal de um esquema XML. São criados operadores de perturbação baseados no modelo, que são usados para modificar os esquemas reais ou virtuais, com o objetivo de cobrir todas as áreas de teste que foram definidas.

Uma metodologia para analisar a qualidade dos serviços compostos que usam técnicas de injeção de falhas, é apresentada em [28]. Esta baseia-se em dois aspetos principais: a reação dos serviços compostos quando submetidos a dados maliciosos e o efeito de atrasos neste tipo de serviços. Estes dados maliciosos incluem a substituição de dígitos num número, inversão de carateres em *strings* e datas e representação de datas em formatos diferentes. Neste estudo podia ter sido considerado também o uso de condições de limite, ou seja, exploração dos limites dos dados, por exemplo, aceder a uma posição de uma *string* que não exista, ou que exceda o seu tamanho.

É apresentada uma *framework* para testar a robustez de serviços construída com *Web Services - Business Process Execution Language* em [29], tendo em conta que a composição de vários serviços, pode resultar num serviço composto, com maiores vulnerabilidades a nível de robustez em relação a um mais simples, devido à complexidade existente no caso composto. Estes são também mais complexos no que toca à elaboração de testes em relação aos testes em aplicações normais, devido ao número de cenários de teste e ao custo que estes acarretam, que aumenta com o aumento de serviços de componente. A *framework* de testes apresentada, introduz um serviço virtual, que corresponde a um serviço de componente real. Esta simula situações dentro do serviço real, que fogem à rotina do sistema, permitindo deste modo avaliar a robustez do serviço, quando submetido a situações excecionais.

Rabhi em [30], apresenta um método de testes de robustez para *Web Services* compostos, que tem como foco testar automaticamente a robustez das operações dos serviços neles existentes. Um *Web Service* composto, é um conjunto de *Web Services* que forma um novo serviço mais complexo a nível funcional. O autor começa por apresentar outros métodos existentes para testar a robustez de serviços compostos seguindo-se a descrição da sua abordagem. Nesta são usadas especificações simbólicas que modelam o comportamento de um *Web Service* composto. A partir destas é traduzida automaticamente uma árvore de execução simbólica que caracteriza os caminhos de execução seguidos durante a execução simbólica. A partir desta são geradas outras subárvores por cada operação que permitem que o analista distinga os comportamentos das várias operações e configure um conjunto de casos de teste que vai aplicar a cada operação. De seguida são geradas subespecificações que são completadas usando regras de robustez apresentadas no artigo, sendo injetados valores específicos nos casos de teste que vão testar o comportamento dos serviços.

Cong *et al*. [31], realçam a importância dos testes de robustez no ciclo de desenvolvimento dos *drivers*. A geração e injeção de cenários de falhas eficientes são um fator importante para testar a robustez dos *drivers* de modo a poupar tempo e recursos. Neste artigo, é apresentada uma abordagem que permite a geração e injeção de cenários de falhas automaticamente para testar esta robustez. Nesta, começa por se identificar funções alvo que possam falhar quando são invocadas por *drivers*. Segue-se a configuração das falhas e geração dos respetivos cenários que vão permitir testar a robustez dos *drivers*. Para melhor compreensão dos resultados por parte do analisador é usado o *stacktrace*, com o objetivo de analisar todo o caminho percorrido e quais os métodos envolvidos. Todos os resultados obtidos são guardados numa base de dados, tendo sido considerados três resultados possíveis para a caracterização do *driver*: *pass*

(o *driver* trata o cenário de falhas com sucesso), *fail* (o sistema ou o *driver* deixam de funcionar resultando na ocorrência de um *crash*) e *null* (resultado inicial antes dos testes serem aplicados). Esta abordagem foi aplicada em doze *drivers* do Linux onde se observou a existência de vinte e oito *bugs*. Para confirmação dos mesmos, os autores procederam à sua validação manualmente. Conclui-se com este artigo, que através dos resultados obtidos, a abordagem é útil e eficiente no que toca à geração de injeção de cenários de falhas para testar a robustez de *drivers*.

Apesar de existirem diversos *scanners* para tratar este tipo de problemas, as abordagens usadas contêm algumas limitações. A grande maioria das abordagens existentes, recorrem à aplicação de testes a partir do cliente, onde a robustez é maior, em vez de aplicarem os testes a partir da base de dados, onde a robustez é menor, visto que o tratamento de dados provenientes da base de dados é de baixa qualidade, considerando-se, de forma errada, que os dados que estão inseridos na base de dados são seguros e já foram tratados anteriormente, antes da sua inserção [32]. No entanto, existem alguns métodos que envolvem a alteração dos dados na base de dados, possuindo ainda assim a limitação de ser necessário alterar o conteúdo da base de dados propositadamente, para ser possível a aplicação dos testes, o que acaba por se tornar pouco prático e de qualidade questionável. Numa situação ideal, a aplicação deveria ser feita a partir da base de dados, visto que esta é a parte mais suscetível a falhas, mas sem haver interferências no seu conteúdo.

2.4 – Testes de Segurança

À medida que o uso de *Web Services* aumenta, aumentam também os riscos associados à sua segurança. Logo, torna-se indispensável pensar na proteção e segurança destes serviços, com o objetivo de evitar possíveis ataques que possam comprometer dados sigilosos, arruinar bases de dados, entre outros já referidos anteriormente. Muitas das falhas de segurança resultam da fraca ou inexistente verificação ou validação dos dados [33].

Na Subsecção 2.4.1 são apresentados conceitos básicos de SQL *Injection*, apesar de este método não ser o foco nesta dissertação. Seguidamente é ainda feita uma explicação detalhada, na Subsecção 2.4.2, sobre *Second-Order* SQL *Injection*. A opção pela análise e tratamento destes dois casos de segurança nesta dissertação deveu-se ao facto de serem os ataques mais frequentes em *Web Services* [34].

2.4.1 – SQL Injection

Ataques por SQL *Injection* têm marcado presença nos serviços *Web* existentes há mais de uma década, prevalecendo sobre toda a segurança que tem acompanhado o seu desenvolvimento, devendo-se em grande parte à atratividade da informação que se encontra dentro das bases de dados. Segundo o relatório de Veracode's 2015 State of Security Software [35], o valor destes dados faz com que o número de ocorrências de ataques existentes, atinjam mais de 30% dos mais de 208.000 *Web Services* testados desde Outubro de 2013.

Uma das maiores preocupações das aplicações *Web Services* é a fuga de informação confidencial, a qual pode ser alcançada, por exemplo, através de ataques de SQL *Injection*. Este género de ataques aproveita-se de falhas em sistemas que recorrem ao uso de bases de dados via SQL. Isto ocorre quando o atacante, através de injeções de comandos, introduz instruções SQL maliciosas dentro de uma consulta (*query*), alterando a estrutura da instrução SQL original, o que permite a leitura, alteração e remoção de informações e recursos importantes, acabando por corromper bases de dados inteiras [36]. Através da manipulação das entradas de dados de uma aplicação, ou seja, basta que o utilizador possua alguns conhecimentos de SQL, e alguma dedicação para descobrir nomes de tabelas e campos, para poder recorrer a este método e causar problemas a uma aplicação que seja vulnerável. Um modo de testar a vulnerabilidade da aplicação, e um método bastante usado para fazer SQL *Injection,* é o uso de apóstrofos pelo utilizador no *input* (ver Figura 2.6). Caso o atacante receba um erro de sintaxe por parte da aplicação, existe uma grande possibilidade de que esta seja vulnerável a este tipo de ataques. Tal acontece devido à falta de tratamento adequado de exceções que consigam prevenir este tipo de situações, nomeadamente quando se trata de *software* desenvolvido sem conhecimentos técnicos adequados, que por norma, não passa pelos mesmos testes de segurança que o *software* desenvolvido por pessoal devidamente qualificado.

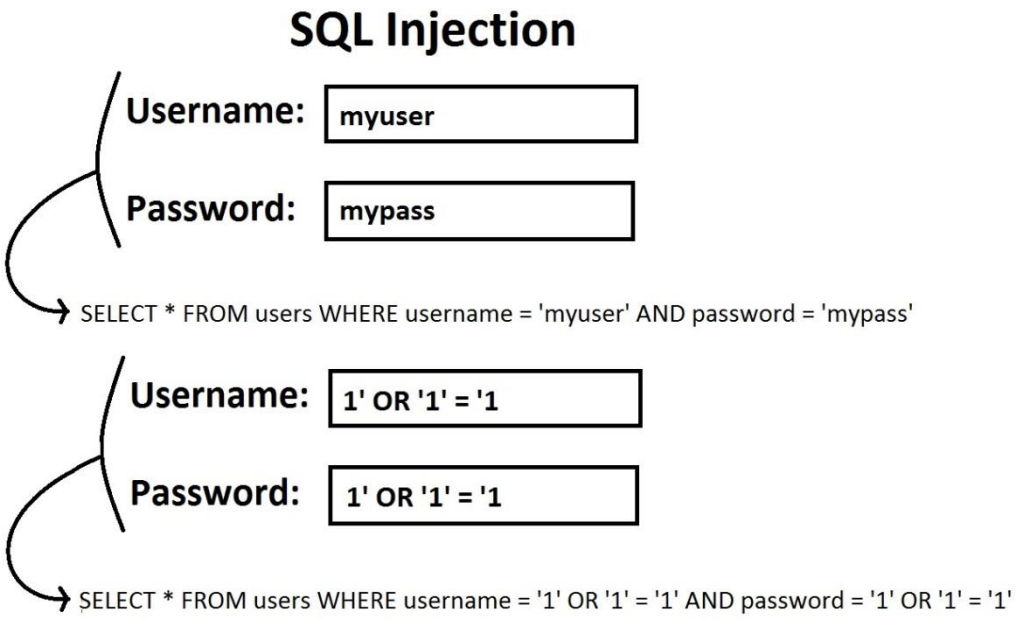

Figura 2.6 – Exemplo de *SQL Injection* (adaptado de [37]).

Como se pode verificar, uma coisa tão simples como um formulário de login, pode ser a causa de graves problemas (ver Figura 2.6).

Considerando o seguinte exemplo:

> **SELECT * FROM users WHERE *username* = 'myuser' AND *password* = '*password*';**

A instrução SQL referida anteriormente, mostra todos os registos da tabela utilizadores para um determinado *username* e *password* fornecidos pelo utilizador. Visto que a

concatenação da *query* com o *input* direto do utilizador torna esta aplicação vulnerável, basta que o utilizador insira ’ or ‘1’ = ‘1’, para que resulte na seguinte *query*;

> **SELECT * FROM users WHERE *username* = '1' OR '1' = '1' AND *password* = '1' OR '1' = '1';**

Com este comando, o utilizador ao usar 1’ OR ‘1’ = ‘1’ vai criar uma condição que é sempre verdadeira devido à cláusula “OR ‘1’ = ‘1’ ”, pois 1 vai ser sempre igual a 1, logo se esta condição é sempre verdadeira, então significa que o processo de autenticação vai ser sempre aceite.

Em [38] são explicados vários métodos para realizar ataques de SQL *Injection* e também algumas técnicas que ajudam na prevenção dos mesmos. Nesta publicação são referenciados ataques baseados em tautologias, *Union Query* e *Blind Injection*. *Tautology-based attacks,* são aplicados através da injeção de código em instruções SQL, que usa condições que sejam sempre verdadeiras. Este foi o tipo de ataque usado como exemplo anteriormente. *Union Query attacks*, referem-se a ataques que são executados através do uso da cláusula *Union*. O atacante adiciona uma instrução SQL com a cláusula *Union* à instrução original, com o objetivo de retirar informações adicionais da base de dados. Um outro tipo de ataque referido neste artigo é o *Blind Injection*, onde o atacante tenta adivinhar o que existe na base de dados, através do método tentativa e erro, analisando as respostas enviadas pelo servidor para conseguir realizar um ataque com sucesso.

Tendo já sido discutida a forma como usar SQL *Injection* para passar por um simples processo de autenticação, importa agora verificar até que ponto este tipo de ataques pode ser grave. A principal consequência de um ataque de SQL *Injection* bem sucedido está relacionada com o comprometimento da confidencialidade dos dados, da integridade da aplicação/base de dados, da autenticação nas aplicações e da informação de autorizações (permissões). Por exemplo, se um atacante conseguir efetuar um ataque que lhe permita obter direitos de administrador na base de dados, isto implica que se torna possível apagar toda a informação existente na base de dados. Para além desta consequência existem outras: obtenção de informação confidencial e alterações em bases de dados.

Leonard e Sims [39], apresentaram a análise de uma ferramenta que executa este tipo de ataques, para verificar se existem vulnerabilidades em *Web Services*, cujo nome é HP *WebInspect*. Esta ferramenta foi criada pela SPI Dynamics, a qual foi comprada pela HP em 2007. *WebInspect* é uma aplicação dinâmica de testes de segurança, *Dynamic Application Security Testing* (DAST), que testa a segurança de aplicações *web* automaticamente, através da execução de ataques às mesmas. Esta usa agentes de avaliação que atuam em diferentes partes da aplicação, os quais enviam os resultados obtidos da sua análise, para um componente de segurança que avalia estes dados. No fim da análise do sistema, é gerado um relatório que contém todas as falhas de segurança encontradas no *Web Service*. Através deste relatório, o cliente pode corrigir as falhas encontradas e depois correr a ferramenta novamente, de modo a confirmar a correção destas vulnerabilidades. Esta aplicação utiliza o método de execução de testes *Black Box* referido anteriormente, para fazer a análise de segurança da aplicação em questão. O autor fecha este artigo com uma conclusão que envolve sugestões referentes ao modo como podem ser combatidos estes ataques e realça a importância de implementar uma ferramenta DAST desde o início, durante o desenvolvimento de uma aplicação.

Uma nova abordagem é apresentada em [40], onde os autores referem a importância de ter um bom *scanner* que permita expor vulnerabilidades de SQL *Injection* numa aplicação. O artigo conclui que muitos dos scanners existentes são pouco eficientes e os resultados obtidos a partir destes são a prova disso, evidenciando o quão importante é a segurança nos *Web Services* e o quão medíocres são os meios existentes para a assegurar. Com o objetivo de contrariar esta situação, é apresentada uma ferramenta que consegue obter resultados mais eficazes do que os *scanners* comerciais existentes. Esta inclui várias fases no seu procedimento dos testes, desde a sua preparação à sua análise. Durante a fase de execução são mencionados alguns casos de SQL *Injection* com os quais os autores trabalharam, que permitiram compreender e observar quais as falhas existentes nos sistemas, falhas estas que eventualmente possibilitam que atacantes se infiltrem na aplicação para usufruir de informação confidencial existente. Esta fase de execução está divida em duas partes. A primeira envolve gerar *workload* sem considerar as injeções de dados maliciosos, com o intuito de perceber quais são as respostas típicas do *Web Service*. A segunda já implica a alteração de informação. Nesta fase são executados diferentes passos e em cada um deles é focado um serviço diferente. Cada passo contém também *slots*, responsáveis por focar um parâmetro específico, sendo posteriormente usados para executar períodos de ataques. Em cada um destes períodos, são efetuados ataques a um dado parâmetro da operação, onde é substituída a informação válida, por inválida, de modo a verificar se esta alteração vai causar uma reação não esperada na aplicação. Neste artigo conclui-se que é possível melhorar o estado da arte no que toca à deteção de vulnerabilidades e que é possível o desenvolvimento de *scanners* mais eficientes do que os que estão disponíveis comercialmente, realçando a importância desta temática e o impacto que esta tem na segurança dos *Web Services* existentes.

A principal limitação existente nos *scanners* deste tipo de testes, acaba por ser a mesma que nos testes de robustez, ou seja, a aplicação dos testes através do cliente, que acaba por ser a vertente mais segura do sistema. Na eventualidade de ser uma abordagem usada a partir da base de dados, possui a limitação de ter de se comprometer o conteúdo da base de dados para os testes poderem ser efetuados.

2.4.2 – Second-Order SQL Injection

Este ataque é bastante parecido com o ataque acima referido, com a diferença de que, neste ataque, os comandos SQL maliciosos são armazenados na base de dados, para serem executados mais tarde, em vez de serem executados naquele instante, como é apresentado na Figura 2.7.

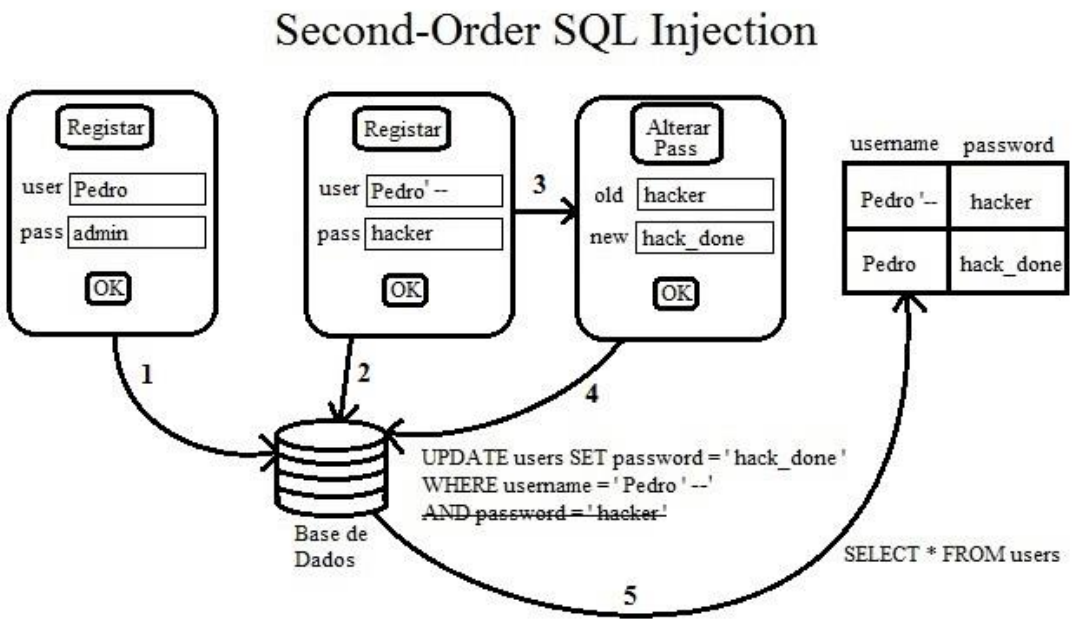

Figura 2.7 – Exemplo de *Second-Order SQL Injection*.

Neste exemplo, o ataque *Second-Order* SQL *Injection* ocorre da seguinte forma: supondo que existe um utilizador registado com o *username* = Pedro e *password* = admin. Considerando agora que ocorre um registo de um utilizador com o *username* = Pedro'-- e *password* = hacker. Seguidamente, o utilizador com o *username* = Pedro'-- quer alterar a sua *password*. Neste momento, ele vai desencadear o ataque *Second-Order* SQL *Injection*.

A *query* resultante de um pedido de alteração de *password* é a seguinte:

> **UPDATE users SET *password* = '*password*_nova' WHERE *username* = 'nome' AND *password* = '*password*_antiga';**

No entanto, neste caso, ao alterar a sua *password*, como se pode observar na Figura 2.7, o utilizador vai desencadear o ataque. O pedido de alteração de *password* vai resultar na seguinte *query*;

> **UPDATE users SET *password* = 'hack_done' WHERE *username* = 'Pedro' - -' ~~AND *password* = '*hacker*'~~;**

Devido à utilização de '--, o *username* a ser alterado vai ser o Pedro e visto –' ser interpretado como comentário pelo SQL, tudo o que se encontra a seguir a estes caracteres, é ignorado. Consequentemente, a comparação da *password*, já não vai ser feita, logo vai alterar a *password* do utilizador Pedro apesar de não ser este o utilizador que está a efetuar as alterações. A implementação deste tipo de ataques envolve dois processos: armazenamento e ativação. O primeiro implica guardar todo o *input* do utilizador na base de dados para que seja utilizado posteriormente. Já o processo de ativação, inclui o recolher da informação e sua concatenação com uma instrução SQL, a qual vai desencadear o ataque por *Second-Order* SQL *Injection*.

Muitas das aplicações *Web Services*, usam o tratamento correto dos dados ao inseri-los na base de dados, mas assim que estes se encontram armazenados, não são mais tratados

ou analisados para verificar se os dados são uma ameaça para a aplicação [32], logo, assume-se que o que está dentro da base de dados não é uma ameaça. Desta forma permite-se que estes tipos de ataques sejam possíveis quando a base de dados usa informação nela existente, como um *input* de uma *query* de uma pesquisa posterior. Esta forma de ataque é uma prova de que deve existir tratamento de dados antes e depois da inserção de informação em base de dados, ou seja, fazer o tratamento de dados quer sejam provenientes do utilizador ou da base de dados. Depois destes comandos SQL maliciosos se encontrarem armazenados, é uma questão de tempo até serem usados pela própria aplicação, provocando um ataque por SQL *Injection*. Este ataque requer mais conhecimentos em relação ao referido na Subsecção 2.4.1, pois neste caso, o atacante tem de saber que valores usar, de modo a que se tornem mais tarde num ataque bem sucedido. Este ataque não é facilmente detetado por *scanners* de segurança de aplicações *web* porque estes servem para verificar falhas antes de os dados estarem na base de dados e não quando já estão inseridos. Existem várias maneiras de diminuir a possibilidade de um ataque por *Second-Order* SQL *Injection*, uma delas é a utilização de consultas parametrizadas (*prepared statements*), onde os valores são passados usando parâmetros SQL em vez de se inserir os dados de entrada do utilizador diretamente na consulta.

Em [41], é descrita em detalhe a técnica de SQL *Injection*, apresentando vários métodos utilizados para fazer este tipo de ataque, incluindo, *Second-Order* SQL *Injection*, *String without quotes*, *Lenght Limits* e *Audit Evasion*, assim como problemas relacionados com a validação dos dados que são submetidos pelo utilizador. O autor mostra como podem ser feitos ataques deste tipo, apesar de serem tomadas algumas precauções para tentar contornar este problema, como por exemplo, ignorar as aspas isoladas. De qualquer forma, estas precauções não são suficientes para travar estes ataques, sendo que a melhor solução apresentada consiste em rejeitar todo o *input* malicioso em vez de o alterar.

Wei *et al.* [42], propuseram uma nova técnica que permite evitar ataques que tenham como alvo *Stored Procedures*. Para a avaliar foi desenvolvido um protótipo, que foi testado numa base de dados em SQL Server 2005. Esta técnica combina a análise estática do código com validação em tempo de execução com o objetivo de eliminar estes ataques. Foi usada, para experiência, uma base de dados protegida com este protótipo e outra sem estar protegida. Em ambos os casos foram feitos ataques de SQL *Injection*, incluindo ataques de *Second-Order* SQL *Injection*, os quais foram detetados pela técnica referida. Os autores chegaram à conclusão que esta técnica/ferramenta seria uma solução para defender uma aplicação de ataques deste tipo.

2.4.3 – Prevenção e Deteção de SQL Injection

A OWASP (*Open Web Application Security Project*) [43] detalha alguns métodos usados para combater ataques desta origem. Estes ataques ocorrem principalmente quando os programadores desenvolvem *queries* dinâmicas que incluem *input* do utilizador. Uma boa prática de programação é obrigatória para ajudar no combate a este tipo de falhas, de maneira a que, quando o *input* do utilizador contenha SQL malicioso, a estrutura da *query* não seja modificada. O uso de *prepared statements* e/ou *stored procedures* parametrizadas em todas as intruções SQL, em vez das típicas *statements*, é um dos métodos que permite tornar a aplicação mais segura e menos propícia a ataques

de SQL *Injection*. As *stored procedures*, visto que são compiladas antes do *input* do utilizador ser adicionado, fazem com que seja impossível a alteração da instrução SQL pelo *hacker*. Outra sugestão dada é a atribuição de privilégios que sejam o mais baixo possíveis (*least privilege*), isto é, se a aplicação necessita apenas de ler, então ela deve ter apenas permissão para ler e não qualquer outra, que permita a alteração de informação na base de dados. O mau tratamento de exceções é outro motivo pelo qual o atacante consegue infiltrar-se no sistema com uma frequência indesejável. Exceções que não são capturadas acabam por fornecer demasiada informação ao utilizador, dando a conhecer, inclusive, nomes de tabelas e nomes de procedimentos. O melhor método referido pelo autor é o uso de *stored* procedures e mesmo assim estas não são totalmente seguras. Seguem-se as *queries* parametrizadas e por fim as instruções SQL dinâmicas. Resumindo, as melhores opções referidas para um aumento da segurança são:

- Uso de *prepared statements* (*queries parametrizadas*);
- Uso de *stored procedures*;
- Fazer o *escape* do *input* do utilizador antes de ser introduzido na *query*;
- Dar apenas as permissões necessárias para o caso em questão (*least privilege*);
- Fazer validação do *input* antes deste ser processado pela aplicação.

Uma nova abordagem para detetar *Second-Order* SQL *Injection* através de uma solução estática e dinâmica, é proposta em [44]. O primeiro passo é a análise do código a fim de detetar pontos vulneráveis na aplicação, seguindo-se uma sequência de testes, com *inputs* maliciosos gerados, que vão averiguar se estas vulnerabilidades se confirmam. Neste trabalho explica-se igualmente o princípio de um ataque deste tipo e são estabelecidas definições formais sobre o método apresentado, de modo a descobrir a ligação entre o processo de armazenamento e ativação do ataque. Na análise ao código é feita uma procura pelas instruções SQL existentes incluindo os campos sobre a qual vai operar (nome da coluna), sendo depois extraídas, usando expressões regulares que sirvam o mesmo propósito. Posteriormente, são aplicadas as definições acima referidas, seguindo-se a geração da sequência de testes, que está relacionada com o mapeamento da relação entre as instruções SQL e os pedidos de HTTP, também explicado nas definições formais apresentadas pelos autores. O passo seguinte consiste na geração do *input* que será usado para testar a aplicação, o que implica a injeção de dados maliciosos que vai alterar a semântica da instrução SQL original. Por fim, a fase de testes onde se vai incorporar os *inputs* maliciosos com a sequência de testes, seguindo-se mais uma vez as definições formais já mencionadas. Estes testes são executados através de pedidos HTTP, na ordem da sequência de testes. Caso o processo de armazenamento falhe no envio do primeiro pedido, o segundo já não é enviado e segue-se para a próxima sequência de testes. Se o *input* malicioso for guardado com sucesso na base de dados, o segundo pedido é enviado, o qual vai revelar a existência de uma vulnerabilidade, caso esta exista. Finamente, Lu Yan *et al.* concluem que o método apresentado através da combinação das vantagens da análise estática e de testes dinâmicos permite a deteção de vulnerabilidades no sistema e reduz os falsos positivos.

Como se pode constatar, pelo que já foi dito antes, os *Web Services* estão longe de estarem seguros, tendo em conta o elevado número de ataques assim como o número reduzido de *scanners* eficientes. Tendo em conta estes fatores, a elevada quantidade de *Web Applications* que contêm vulnerabilidades tanto de robustez como de segurança, que comprometem a sua integridade, e a abordagem única apresentada considerou-se pertinente a contribuição proposta nesta dissertação, considerando que o resultado da ferramenta irá ser uma mais-valia para a segurança dos *Web Services* existentes. Posto isto, é seguidamente apresentada esta abordagem e toda a evolução e modo de funcionamento da ferramenta desenvolvida.

Capítulo 3 – Metodologia e Abordagem

Neste capítulo são discutidas algumas abordagens relacionadas com a mesma temática que servem de base a esta dissertação (Secção 3.1). Posteriormente, na Secção 3.2, é feita a descrição dos testes criados, onde é ainda apresentado o modelo de falhas, o perfil dos testes e a classificação dos mesmos.

3.1 - Técnicas para Injeção de Valores Inválidos ou Maliciosos

Como definido inicialmente, o objetivo passou por testar a robustez e segurança de uma aplicação através da injeção de dados inválidos e maliciosos na mesma, de tal modo que vulnerabilidades existentes que possam ser usadas para ataques fiquem expostas para que as possamos corrigir. Para tais ataques poderem ser efetuados existem diversas abordagens que podem ser implementadas:

- Injeção de dados maliciosos através do *input* do cliente;
- Alteração da informação existente na base de dados;
- Alteração da informação diretamente na aplicação (servidor).

A abordagem que recorre à alteração do cliente consiste na **injeção de dados maliciosos através do *input* do cliente** [44]. Nesta abordagem, a origem dos ataques é o cliente, que injeta dados maliciosos através dos *input*s requeridos pela aplicação, de modo a tentar encontrar falhas no sistema para que as consiga explorar. Este tipo de ataques recorre ao uso de dados que consigam comprometer a segurança do sistema, contornando-a através da injeção de informação que o sistema não está à espera, ou seja, injeção de dados que a aplicação não consegue tratar, não tem capacidade de filtrar. SQL *Injection* é um dos métodos usados neste tipo de ataques, onde o cliente injeta instruções SQL, em áreas que não são direcionadas para tal, por exemplo, o uso de instruções SQL em formulários de *login* é um exemplo, como se pode verificar na Figura 2.6.

Outra possibilidade é a **alteração da informação existente na base de dados** [45]. Este tipo de abordagem está centrada na base de dados e nas informações nela existentes. Normalmente, este método recorre à introdução de dados maliciosos diretamente na base de dados, ou seja, as tabelas da base de dados de uma aplicação são alteradas diretamente, havendo a inserção de dados maliciosos, que permitam testar a robustez e segurança de uma aplicação. Para isto, parte-se do pressuposto que os campos modificados na base de dados vão ser usados futuramente.

Através da **alteração da implementação da aplicação (servidor)** [46], recorrendo a modificações feitas na mesma, como por exemplo, nos valores retornados pelas funções, é também possível comprometer a robustez/segurança da aplicação. Este método envolve a criação de uma ferramenta interna implementada diretamente na aplicação que irá servir de apoio ao programador para testar a aplicação à medida que esta seja desenvolvida. Exemplificando de forma sintética, imagine-se que foi feita uma pesquisa à base de dados e que a informação retornada chega à aplicação, então é neste momento que vão ser feitas as alterações. A aplicação em vez de ir buscar os valores que foram retornados pela base de dados, vai buscar valores maliciosos, como a uma tabela de testes, utilizando-os para realizar então os testes de robustez e segurança. Resumindo, seria, por exemplo, pegar na abordagem elaborada e em vez de a usar numa versão alterada do JDBC, usá-la diretamente na aplicação, alterando o código desta.

Ao contrário de todas as abordagens atrás referidas, a metodologia aqui apresentada não interfere nem altera o normal funcionamento do sistema, não existindo alterações em clientes, em aplicações, ou em bases de dados, como se pode observar na Figura 3.1.

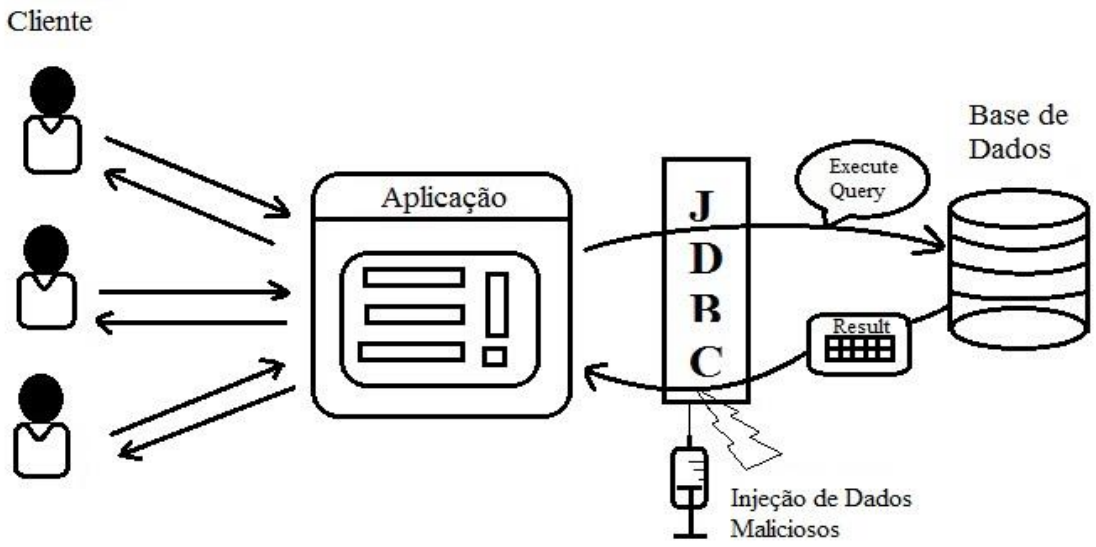

Figura 3.1 – Funcionamento da técnica, em Java, para injeção de valores inválidos e maliciosos.

Esta abordagem surgiu da ideia de que qualquer método usado para testar a segurança e robustez de uma aplicação não deve implicar alterações diretas no funcionamento ou estrutura do sistema em questão, ou seja, não deve ser necessário fazer alterações ao *Web Service* para podermos executar os testes. Desta forma, a ferramenta está presente no sistema apesar de não se notar a sua presença, ou seja, a única coisa que é feita para a aplicação da ferramenta é a substituição do *driver* JDBC entre a aplicação e a base de dados. Basicamente, todas as alterações efetuadas ocorrem no meio da comunicação entre a base de dados e a aplicação, através de um componente *middleware* (JDBC). Em outras linguagens existem também componentes semelhantes que permitem realizar as mesmas funções [47].

A implementação desta ferramenta acaba por ser alheia ao código dos *Web Services*, não precisando de ser adaptada a cada sistema alvo para ser usada. Esta invisibilidade torna-se possível graças ao conceito de Programação Orientada a Aspetos (POA) [46], mais especificamente ao AspectJ [47], que foi a primeira linguagem a ser desenvolvida, sendo a mais popular atualmente. Este conceito, POA, foca os problemas que, em Programação Orientada a Objetos não têm solução apropriada, os chamados problemas de *crosscutting* que dizem respeito a código que implementa funcionalidades secundárias e que é usado em diversos pontos da aplicação, ou seja, é código que, em Programação Orientada a Objetos, não diz respeito ao comportamento do objeto em si, não fazendo parte da sua estrutura. Nesta dissertação foi usado o conceito de *join point* de POA, que tem como objetivo identificar os pontos da aplicação (*e.g.*, métodos) que devem ser abordados para fazer, ou não, alterações e *advice* que corresponde ao código

que vai ser executado quando estes *join points* são encontrados. Código este que tem como finalidade adicionar métodos, variáveis ou neste caso a alteração do resultado esperado pela aplicação. Tendo em conta que um dos objetivos desta dissertação passou por desenvolver uma ferramenta que fosse o menos evasiva possível, o recurso a esta linguagem, que é especializada em POA, foi obrigatório. Esta permitiu também que a ferramenta proposta fosse automatizada de modo a que não fosse preciso adaptá-la para cada base de dados.

O funcionamento da abordagem referida é bastante simples. Primeiro e antes de tudo, é introduzida a ferramenta, entre a aplicação e a base de dados, substituindo o *driver* JDBC existente, como já foi referido. Posteriormente, o cliente faz as transações e pedidos normais com a aplicação, a qual vai aceder à base de dados para realizar os pedidos efetuados. Neste acesso, os pedidos vão passar pela ferramenta, que vai aceder à base de dados normalmente, fazendo as pesquisas necessárias e extraindo os resultados. No final do processamento do pedido feito pelo cliente, entra em ação a abordagem implementada. Com a ferramenta criada, no momento em que os resultados estão prestes a ser transmitidos de novo para a aplicação, estes vão ser alterados e substituídos, conforme o teste a ser aplicado. Após a sua substituição, os dados são enviados para a aplicação, como se fossem os resultados das pesquisas requeridas por esta. Esta alteração de resultados vai possibilitar a verificação da existência de problemas de robustez ou segurança e em que circunstâncias eles surgem, de tal modo, que se possa proceder à sua correção futuramente.

3.2 – Abordagem para Avaliação de Robustez e Segurança

Nesta secção são abordados os Testes de Robustez e Segurança desenvolvidos, usados pela ferramenta implementada. Nos testes recorre-se ao uso de dados predefinidos que exploram as falhas existentes nos *Web Services* em cada acesso à base de dados. Nestes podem ser consideradas quatro fases, como se pode observar na Figura 3.2:

Figura 3.2 – Fase de testes da ferramenta.

1) **Preparação dos testes:** configuração do sistema onde vão ser feitos os testes, ou seja, configuração da aplicação, do cliente, da base de dados e substituição do *driver* JDBC pela ferramenta criada.
2) **Recolha de Informação:** recolha da informação que corresponde ao normal funcionamento da aplicação, cujo objetivo é observar o comportamento normal

do *Web Service* de modo a obter a estrutura da aplicação para os testes poderem ser aplicados. Esta estrutura corresponde aos pontos de acesso à base de dados existentes no código, ou seja, vai ter o papel de um mapa, para a aplicação saber onde vai ter de efetuar os testes, de modo a verificar se existem vulnerabilidades naquela localização em específico. É importante referir que esta recolha é efetuada em ambiente e *workload* controlados. Isto significa que todos os testes vão ser executados num ambiente configurado pelo analista, tendo a possibilidade de aumentar ou diminuir o volume de transações no sistema. Esta fase decorre durante um tempo predefinido, não havendo quaisquer alterações ou modificações no normal funcionamento do serviço;

3) **Execução dos Testes de Robustez/Segurança:** aplicação dos testes, através da injeção de parâmetros inválidos, de modo a descobrir problemas existentes no sistema;
4) **Caraterização do Serviço:** classificação do serviço através dos resultados obtidos a partir da introdução de testes de robustez no normal funcionamento da aplicação (*e.g.*, o serviço lança uma exceção na existência de um *null pointer*, logo foi necessário avaliar quais os danos causados e quais as consequências para a aplicação). Mais uma vez é importante referir que este normal funcionamento da aplicação corresponde a uma simulação do *workload* da mesma no seu dia-a-dia, podendo haver alterações para aumentar ou diminuir o congestionamento do sistema para simular todos os cenários possíveis.

As fases apresentadas anteriormente devem ser seguidas de uma forma sequencial. Estes passos podem também ser usados por qualquer ferramenta de testes de robustez desde que seja usada a mesma abordagem aqui descrita.

1) Preparação dos Testes

Inicialmente, para a preparação dos Testes, é necessário configurar todos os componentes presentes no sistema que vai ser testado, de modo a que tudo esteja devidamente preparado para não existirem falhas ao nível da estruturação e configuração do sistema. Este é um passo importante para que os resultados dos testes executados sejam precisos. Existem alguns requisitos que devem ser cumpridos para que a ferramenta funcione tais como, a linguagem da aplicação sob teste tem ser Java e conter a versão 6 do Java EE com a versão Servlet 3.0 tendo em conta que a ferramenta utiliza filtros HTTP que necessitam de deteção automática da parte do servidor, disponível apenas na versão do Servlet acima referida. Depois dos requisitos se verificarem, é necessária a presença de um ou mais clientes que executam os serviços da aplicação. É preciso configurar também a aplicação que contem os serviços referidos e a base de dados que contem toda a informação existente. É ainda necessário considerar a existência de um servidor capaz de alojar todo este sistema. Por fim, deve ser feito o *deployment* da ferramenta, sendo esta um ficheiro do tipo jar. A mesma é incluída entre a aplicação e a base de dados, substituindo o jar do *driver* JDBC existente pela ferramenta com a qual os testes vão ser realizados.

Após a preparação e configuração de todos os componentes procede-se à fase detalhada em seguida.

2) Recolha de Informação

Antes de se proceder à execução dos testes existem alguns detalhes que requerem atenção especial. Esta fase corresponde ao normal funcionamento da aplicação, onde apenas se tem o papel de observador, assistindo simplesmente ao seu desempenho sem esta estar sujeita a quaisquer intervenções externas. O cliente possui um papel crucial nesta fase, tendo em conta que toda a informação necessária para a construção do mapa provém do *workload* gerado pelos pedidos do mesmo, sendo também importante para a duração que a mesma vai ter, já que esta está relacionada com o número de pedidos que o cliente faz ao servidor, ou seja, o tempo que esta fase vai durar corresponde ao número introduzido numa variável existente na ferramenta. Este número vai corresponder ao número máximo de pedidos recebidos do cliente até ocorrer a transição para a fase seguinte. Exemplificando, se definirmos o valor vinte como máximo, a fase de aprendizagem vai durar até o cliente efetuar vinte pedidos à aplicação. Para controlar o momento em que este valor é atingido existe uma outra variável que é incrementada sempre que a ferramenta deteta um pedido novo do cliente. Esta deteção é feita através de filtros HTTP, que sempre que intercetam uma chamada para um dos serviços, verificam se o número máximo de pedidos do cliente corresponde ao número máximo configurado até mudar de fase. Esta etapa é indispensável para os testes elaborados, pois é recolhida informação crucial sobre a aplicação e a sua estrutura. Com esta é possível criar um mapa da aplicação para permitir a execução dos testes. Este é composto por todos os pontos de acesso à base de dados existentes na aplicação. A construção deste mapa é feita da seguinte forma: em tempo de execução, a ferramenta vai guardar as posições em que existem chamadas à base de dados. Este armazenamento de informação vai ser feito através do AspectJ e StackTrace. O AspectJ vai intercetar todas as chamadas dos métodos que fazem pedidos à base de dados e neste momento, e enquanto se encontrar na fase de aprendizagem, a ferramenta vai usar o StackTrace para guardar a linha e nome do ficheiro onde esta chamada foi feita. Este processo é efetuado para todos os pontos de acesso pelos quais a aplicação passa durante esta fase, criando desta forma, o referido mapa.

3) Execução dos Testes

Nesta subsecção é feita uma explicação de tudo o que está relacionado com a aplicação dos testes, tanto de robustez como de segurança, no *Web Service* em avaliação. Na Figura 3.3 está representado o funcionamento do sistema, quando sujeito à ferramenta de testes em questão.

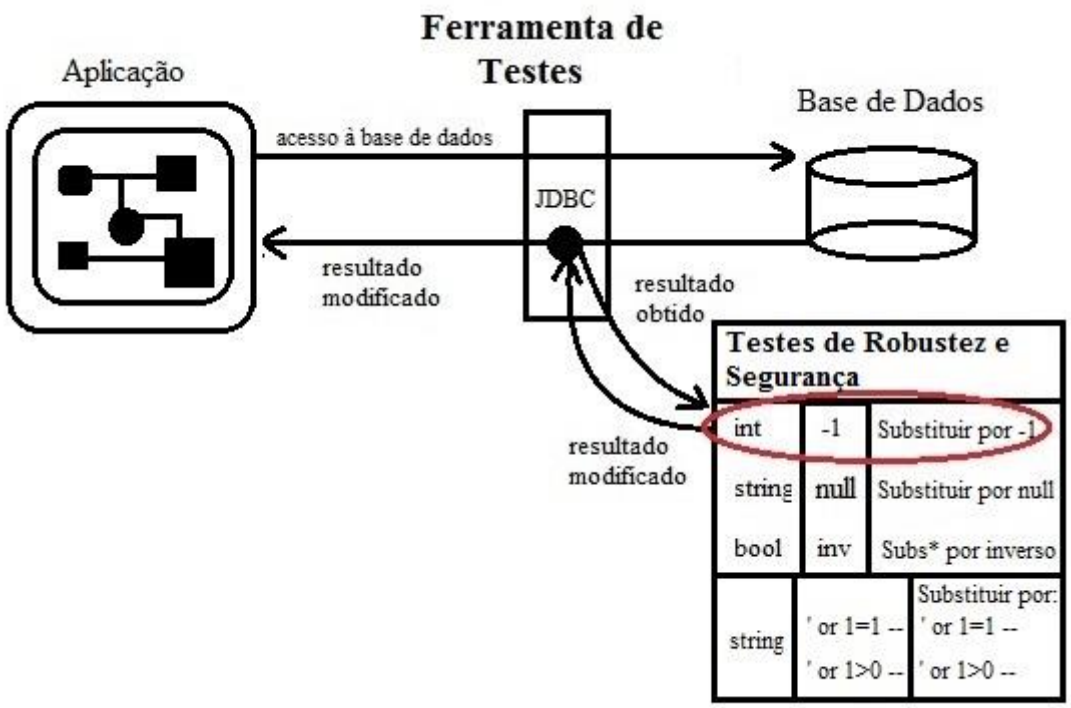

Figura 3.3 – Funcionamento de um sistema sujeito à ferramenta de testes.

O elemento que requer mais atenção é a ferramenta de testes, que vai atuar como um *driver* JDBC modificado e fazer a ligação entre a aplicação e a base de dados do *Web Service*. Esta ferramenta vai receber as pesquisas SQL enviadas pela aplicação e os resultados destas pesquisas da base de dados. Estes resultados vão ser alterados pela ferramenta com o objetivo de procurar vulnerabilidades na aplicação. Estes vão ser substituídos por dados inválidos existentes na Tabela 3.1. Depois de modificados, vão ser transmitidos de novo para a aplicação. Este é o funcionamento base da aplicação para todo o tipo de testes excetuando alguns pormenores que serão referidos posteriormente aquando da explicação do funcionamento individual de cada teste. É importante realçar que com este tipo de abordagem, o código fonte desta não precisa de ser do conhecimento do analista para se poderem executar os testes. Tanto do lado do servidor como do cliente, não é necessário o seu conhecimento.

i. Testes de Robustez I e II

Estes testes envolvem a manipulação de parâmetros, com o objetivo de injetar dados inválidos contra os quais a aplicação pode não estar protegida. A diferença dos Testes de Robustez I para os Testes de Robustez II baseia-se no facto de que nos últimos, só é feita uma injeção por cada pedido do cliente, enquanto nos primeiros esta injeção ocorre em todas as chamadas à base de dados. Esta manipulação de dados referida anteriormente, obrigou à criação de algumas regras que devem ser seguidas nesta abordagem. É importante realçar que estas regras visam atingir as condições de limite dos tipos de dados, que são por norma a origem dos problemas no que toca à robustez das aplicações, visto elas não fazerem a validação deste tipo de situações. Seguem-se exemplos de situações contra as quais as aplicações não costumam estar protegidas:

- Valores máximos e mínimos válidos correspondentes a um determinado tipo de dado (valor máximo possível para um parâmetro e valor mínimo possível para um parâmetro);
- Valores que excedem o máximo e mínimo possível para aquele tipo de dados (valor máximo possível para um parâmetro mais um, valor mínimo possível para um parâmetro menos um);
- Valores a *null* e vazios (*strings* nulas ou vazias);
- Valores com caraterísticas especiais (carateres *non-printable* em *strings*)
- Valores que causam problemas de *data type overflow* (adicionar carateres a uma *string* para que ultrapasse o seu tamanho máximo)
- Valores válidos com dados inválidos (usar datas inválidas)
- Uso de valores maliciosos com o objetivo de explorar potenciais falhas de segurança no sistema [50][51], como por exemplo SQL *Injection*, um exemplo de ataque frequentemente utilizado contra *Web Services* [52].

As regras propostas na Tabela 3.1 estão definidas para cada tipo de dados e foram baseadas em problemas de validação apresentados acima e em artigos e trabalhos sobre testes de robustez em serviços apresentados anteriormente [19][20][53].

Tipo de Dados	**Nome do Teste**	**Mutação de Parâmetro**
Number *int* *float* *double* *short* *long* *BigDecimal*	Número Menos Um Número Um Número Zero Adicionar Um Número Subtrair Um Número Número Máximo Positivo Número Máximo Negativo Número Máximo Positivo Menos Um Número Máximo Negativo Mais Um	Substituir por -1 Substituir por 1 Substituir por 0 Adicionar 1 Subtrair 1 Substituir pelo número máximo positivo Substituir pelo número máximo negativo Substituir pelo número máximo positivo menos um Substituir pelo número máximo negativo mais um
String	*String* Nula *String* Vazia *String* Predefinida *String* Não Imprimível Adicionar *String* Não Imprimível *String* AlfaNúmerica *String* Maliciosa	Substituir por valor nulo Substituir por *string* vazia Substituir por *string* predefinida Substituir por *String* com caracteres não imprimíveis Adicionar caracteres não imprimíveis à *string* Substituir por *string* alfanumérica *String* maliciosa predefinida
Date	Data Nula Data Máxima Mais Um Data Máxima Menos Um Adicionar 100 à Data Subtrair 100 à Data Data – dia-mês-ano: 29-02-1984 31-04-1998 01-13-1997 00-12-1994 31-06-1998 32-08-1993 31-12-1999 01-01-2000	Substituir por valor nulo Substituir por data máxima válida no parâmetro mais um dia Substituir por data máxima válida no parâmetro menos um dia Adicionar 100 anos à data Subtrair 100 anos à data Substituir por datas com o seguinte formato: dd-MM-yyyy Substituir pela seguinte data inválida: 29-02-1984 Substituir pela seguinte data inválida: 31-04-1998 Substituir pela seguinte data inválida: 01-13-1997 Substituir pela seguinte data inválida: 00-12-1994 Substituir pela seguinte data inválida: 31-06-1998 Substituir pela seguinte data inválida: 32-08-1993 Substituir pela seguinte data inválida: 31-12-1999 Substituir pela seguinte data inválida: 01-01-2000

Boolean	Boolean predefinido Resultado Inverso Boolean	Substituir por valor predefinido Substituir por resultado inverso
Time	Hora nula Hora Máxima Mais Um Hora Mínima Menos Um Adicionar 24 Horas Subtrair 24 Horas Hora – hh-mm-ss: 17-02-61 16-04-00 01-00-10 03-61-12 25-08-30 00-08-50 25-61-61 00-00-00	Substituir por valor nulo Substituir por hora máxima válida no parâmetro mais uma hora Substituir por hora mínima válida no parâmetro menos uma hora Adicionar 24 horas ao tempo Subtrair 24 horas ao tempo Substituir por hora com o seguinte formato: hh-mm-ss Substituir pela seguinte hora inválida: 17-02-61 Substituir pela seguinte hora inválida: 16-04-00 Substituir pela seguinte hora inválida: 01-00-10 Substituir pela seguinte hora inválida: 03-61-12 Substituir pela seguinte hora inválida: 25-08-30 Substituir pela seguinte hora inválida: 00-08-50 Substituir pela seguinte hora inválida: 25-61-61 Substituir pela seguinte hora inválida: 00-00-00
TimeStamp	Data/Hora nulos Hora Máxima Mais Um Hora Mínima Menos Um Adicionar 24 Horas Subtrair 24 Horas Tempo – dia-mês-ano hh-mm-ss: 29-02-1984 17-02-61 31-04-1998 16-04-00 01-13-1997 01-00-10 00-12-1994 03-61-12 31-08-1998 25-08-30 32-08-1993 00-08-50 31-12-1999 25-61-61 01-01-2000 00-00-00	Substituir por valor nulo Substituir por hora máxima válida no parâmetro mais uma hora Substituir por hora mínima válida no parâmetro menos uma hora Adicionar 24 horas ao tempo Subtrair 24 horas ao tempo Substituir tempo com seguinte formato: dd-MM-yyyy hh-mm-ss Substituir pela seguinte data/hora inválida: 29-02-1984 17-02-61 Substituir pela seguinte data/hora inválida: 31-04-1998 16-04-00 Substituir pela seguinte data/hora inválida: 01-13-1997 01-00-10 Substituir pela seguinte data/hora inválida: 00-12-1994 03-61-12 Substituir pela seguinte data/hora inválida: 31-06-1998 25-08-30 Substituir pela seguinte data/hora inválida: 32-08-1993 00-08-50 Substituir pela seguinte data/hora inválida: 31-12-1999 25-61-61 Substituir pela seguinte data/hora inválida:01-01-2000 00-00-00

Tabela 3.1 – Tabela de Testes de Robustez I e II.

As regras referentes a *TimeStamp*, por exemplo, foram criadas de raiz assim como *Boolean* e *Time*. Como foi já mencionado, algumas regras já existiam e foram adaptadas para esta ferramenta em questão, como é o caso do grupo *Date, Numbers e Strings*. De modo a garantir a total cobertura da aplicação em relação às suas falhas, foram definidas algumas regras em relação ao procedimento dos testes:

- Todas as operações provenientes do *Web Service* devem ser testadas;
- Para cada parâmetro, todos os testes representados na Tabela 3.1 devem ser aplicados, ou seja, por exemplo, considerando que o tipo de parâmetro corresponde a um número, todos os testes existentes na tabela relacionados com números devem ser aplicados;
- Cada teste pode ser repetido diversas vezes para cada caso em específico, para verificar que o comportamento resultante é sempre o mesmo. Este número de repetições é configurável, de tal modo que cabe ao analista escolher o número de vezes que quer executar cada teste;
- A fase de testes apenas termina quando todos os pontos de acesso à base de dados forem sujeitos a esta injeção maliciosa ou a aplicação termine de forma abrupta (*crash*);

- A transição dos Testes de Robustez I para Testes de Robustez II acontece somente quando forem testados todos os pontos de acesso à base de dados.

O número de repetições dos testes depende do conhecimento que o analista em questão tem acerca do serviço e da cobertura que estes conseguiram oferecer na primeira execução.

Durante a execução dos testes de robustez, existem três cenários diferentes que podem ser criados:

- Testes de Robustez I: a ferramenta executa apenas os testes de Robustez I;
- Testes de Robustez II: a ferramenta executa apenas os testes de Robustez II;
- Testes de Robustez I e II: a ferramenta executa os testes de Robustez I e II.

A ferramenta interceta todas as operações feitas pela aplicação à base de dados, como está representado na Figura 3.3. Quando a ferramenta recebe os resultados da base de dados, verifica a estrutura adquirida na fase de Recolha de Informação de modo a averiguar se aquele caso já sofreu uma alteração anteriormente. Não se tendo verificado esta alteração, procede-se à marcação da *flag* existente, caso contrário continua a execução. Esta sinalização vai permitir à ferramenta saber quando já testou todas as entradas para a base de dados existentes na aplicação, de modo a terminar a fase de testes, se o analista quiser executar apenas os Testes de Robustez I, ou poder avançar para os Testes de Robustez II, o que acaba por ser um método de controlo na transição para o próximo tipo de testes, assim como garantir total cobertura dos testes sob o sistema. Depois de fazer esta verificação no mapa adquirido, acede aos casos de teste representados na Tabela 3.1, e dependendo do tipo de dados em questão substitui a informação correta por dados inválidos retirados da tabela mencionada. Depois desta substituição ser feita, os resultados são reencaminhados para a aplicação. É importante referir que nesta fase, todas as operações feitas que requerem o acesso à base de dados vão ver os seus resultados alterados, ou seja, a ferramenta vai alterar todos os resultados das chamadas à base de dados. Apesar da posição já estar marcada como tendo sido sujeita a testes, no caso de Robustez I, os valores são sempre alterados. O aumento do número de testes pode resultar numa maior probabilidade de serem detetados problemas no sistema.

O momento em que é feito o restauro do sistema também é relevante para o resultado final dos testes, tendo em conta que os resultados obtidos, fazendo o restauro depois dos Testes de Robustez I ou depois dos Testes de Robustez II, vai ser diferente. Isto deve-se ao facto de a acumulação de injeções causar a ocorrência de mais situações de falhas, logo, executar os testes com a base de dados limpa ou comprometida, vai afetar o resultado final dos testes. Para cobertura do maior número de cenários possível é aconselhável a aplicação de ambas as situações.

Como já foi dito anteriormente, a diferença dos Testes de Robustez I para os Testes de Robustez II baseia-se no facto de que nos últimos, só é feita uma alteração por cada invocação à operação do serviço, isto é, por cada vez que o cliente executa uma operação no servidor, a ferramenta, durante esta fase, vai apenas alterar um valor, marcando a *flag* no mapa, tal como nos testes anteriores, para que neste caso, não volte a alterar aquele valor quando a operação for executada. Assim que aquela posição no mapa é marcada, a ferramenta não volta a alterar o valor daquela chamada à base de

dados, mesmo que esta seja acedida novamente. Visto que, o programa só cessa os testes, aquando da marcação de todas as *flags* existentes no mapa ou de uma paragem inesperada do sistema (*crash)*, garante-se, deste modo, total cobertura dos mesmos sob a aplicação, assegurando assim, que todos os possíveis pontos de vulnerabilidade do sistema, são testados, isto considerando que o *workload* aplicado permite que a ferramenta detete a localização de todos os pontos de acesso à base de dados. Na eventualidade do *workload* configurado não ser o suficiente para construir o mapa de toda a aplicação, esta cobertura não pode ser assegurada. Consequentemente, e realçando novamente que todos os pontos de acesso à base de dados são detetados, a segurança do sistema é garantida, para os testes aplicados, sempre que o resultado dos testes confirme que não existe nenhuma falha, ou que o sistema é vulnerável, caso o resultado revele que existem violações, na presença de certos elementos maliciosos.

ii. Testes de Segurança

Os Testes de Segurança, assim como os de Robustez, envolvem a manipulação de parâmetros, com o mesmo objetivo que foi referido nos testes anteriores, *i.e.*, injeção de dados inválidos em pontos da aplicação onde esta possa ser vulnerável. Deste modo, foi novamente necessária a criação de regras, que garantissem o objetivo proposto. Algumas destas regras encontram-se definidas na Tabela 3.2 (ver Anexo A para tabela completa). Estas regras estão categorizadas segundo o seu tipo de ataque. Esta categorização foi baseada na literatura existente [54][55], sendo possível a distinção entre quatro tipos de ataques diferentes:

- *Tautology-based attacks*: injeção de código em uma ou mais instruções condicionais de modo a que estas se verifiquem sempre verdadeiras (e.g, ‘ or 1=1 --). Esta técnica é muito usada para ultrapassar páginas de autenticação e extrair informação de base de dados;
- *Union Queries*: injeção da instrução *Union* em pesquisas SQL. Esta aplicação permite, que o atacante controle por completo a segunda instrução a ser introduzida, podendo assim ser retirada informação de uma tabela em específico existente na base de dados. O resultado deste ataque é a devolução da união dos resultados da pesquisa original com a pesquisa injetada;
- *Piggy-Backed Queries*: o atacante tenta injetar instruções adicionais dentro da instrução original. O objetivo deste método não é alterar a instrução original, mas sim incluir novas e distintas instruções que vão usar a original (método *piggy-back*) para poder fazer ataques à base de dados. Como resultado, esta vai receber múltiplas instruções SQL. A original vai ser a primeira a ser executada, mas com esta, são também injetadas as instruções maliciosas que foram introduzidas previamente. Este ataque permite a introdução de qualquer tipo de comando SQL, incluindo *Stored Procedures* nas instruções adicionais, sendo executadas graças à instrução original;
- *Alternate Encodings*: o texto injetado nesta categoria de ataques é modificado de modo a evitar a sua deteção por parte das técnicas de prevenção existentes. Este ataque é usado em conjunto com outros ataques, ou seja, sozinho não oferece nenhum método para atacar aplicações, representa simplesmente uma técnica que permite que os ataques feitos não sejam detetados por ferramentas de segurança. Trata-se de um método de evasão para que o atacante possa explorar vulnerabilidades de uma aplicação sem ser descoberto. Esta técnica permite, por exemplo, ultrapassar técnicas de prevenção que procurem caracteres maus, ou seja, caracteres que possam ser interpretados como possíveis ataques de SQL

Injection (*e.g.*, aspas, plicas, operadores de comentário). Para isto, os atacantes recorrem à codificação das suas *strings* de ataque (*e.g.*, transformar informação em hexadecimal, ASCII, codificação de caracteres Unicode). Tendo em conta que, as técnicas de deteção comuns não tentam avaliar todas as *strings* codificadas, o ataque passa por elas sem ser detetado.

Ao contrário dos Testes de Robustez I e II, estas vão ser aplicadas apenas a um tipo de dados, como se pode observar na tabela mencionada e foram baseadas em artigos e trabalhos sobre problemas de segurança referidos anteriormente [40][41][42].

Regras	**Categoria do Ataque**	**Mutação de Parâmetro**
Produzidas	*Tautology-Based Attack*	Substituir por: ‘ OR 1 = 1 ‘ OR 1 = 1 -- ‘ OR 1 > 0 ‘ OR 0 < 1 ‘) OR 1 = 1 -- ‘) OR (1 = 1 -- ‘ OR ‘1’ = ‘1’ -- ‘ OR ‘x’ = ‘x ‘ OR 0 = 0 -- ‘ OR 1 = 0 -- OR 0 = 0 -- OR 1 = 1— ‘ OR ‘b’ > ‘a -- ‘ OR ‘a’ < ’b ‘ OR a <= b ‘ OR c >= c -- ‘ OR a LIKE a -- ‘ OR a NOT LIKE b ‘ OR c NOT IN (d, e, f) ‘ OR b IN (a, b, c) -- ‘ OR d BETWEEN c AND e -- ‘ OR b NOT BETWEEN c AND d
	Piggy-Back Queries	Substituir por: ‘; DROP table users; - - ‘; DROP table utilizadores; - - ‘; DROP table clients; - - ’; SHUTDOWN; -- ‘; SELECT top 1 FROM users; --
	Union *Queries*	Substituir por: ‘ UNION select * FROM users - - ‘ UNION all select * FROM users - - ‘ UNION select * FROM members where id = 1 --

Adaptadas	*Tautology-Based Attack*	Substituir por: “ OR “a” = “a “ OR 0 = 0 -- “ OR 0 = 0 # “ OR 1 = 0 -- “ OR 1 = 1 -- ‘ OR 0 = 0 # OR 0 = 0 # ‘) OR (‘a’ = ‘a “) OR (“a” = “a 2989 OR 1 = 1 -- hi’ OR ‘a’ = ‘a hi’ OR 1 = 1 -- hi’) OR (‘a’ = ‘a hi” OR “a” = “a hi” OR 1 = 1 -- hi”) OR (“a” = “a ‘hi’ OR ‘x’ = ‘x’; 0 OR 1 = 1
	Alternate Encondings	Substituir por: ‘; exec(0x73689574646f776e) --
	Union *Queries*	Substituir por: ‘ UNION ALL SELECT ‘ UNION SELECT

Tabela 3.2 – Tabela de Testes de Segurança.

Toda a informação que se encontra apresentada nesta mesma tabela foi sustentada na literatura existente [40][41][42]. As regras definidas na tabela que correspondem à linha “Produzidas”, foram as criadas de raiz durante o presente ano letivo. Existe também a linha “Adaptadas” que diz respeito às regras que foram adaptadas para a abordagem provenientes de outros artigos e trabalhos, como já foi referido. Com o objetivo de cobrir todos os pontos suscetíveis a ataques existentes no sistema, foram seguidas as mesmas regras definidas para os Testes de Robustez I e II, com exceção do último ponto referente à transição entre estes, o qual não se aplica aos Testes de Segurança.

Assim como nos Testes de Robustez I e II, o número de repetições dos testes depende do conhecimento que o analista em questão tem acerca do serviço e da cobertura que estes conseguiram oferecer na primeira execução.

4) Caraterização do Serviço

Nesta subsecção descreve-se uma possibilidade para classificar o nível de robustez de um serviço. Este nível de robustez depende da gravidade das falhas encontradas na aplicação. Uma possível abordagem para esta classificação é o uso da escala CRASH [20], uma escala bem conhecida neste tipo de situações. Esta escala foi usada como base para a caraterização do serviço, estando organizada da seguinte forma:

- *Catastrophic*: o sistema operativo fica corrompido; a máquina bloqueia ou reinicia, persistindo o problema;
- *Restart*: a aplicação bloqueia e precisa de ser terminada à força;
- *Abort*: aplicação termina de forma anormal (*e.g.*, uma exceção inesperada é lançada);
- *Silent*: nenhum erro/exceção foi assinalado, quando deveria ter sido;
- *Hindering*: o código de erro retornado não é o correto.

Capítulo 4 – Avaliação Experimental

A validação da ferramenta desenvolvida refletiu-se como um passo importante na elaboração deste trabalho. No presente capítulo é apresentada e discutida a avaliação experimental feita, onde são analisados os resultados obtidos a partir da introdução da ferramenta num sistema real, com o objetivo de averiguar se a ferramenta está a corresponder às espectativas, que impacto tem nas aplicações sob teste e se a abordagem apresentada funciona devidamente.

De modo a validar a ferramenta desenvolvida, seguiu-se um processo de testes que permitiu revelar *bugs* existentes:

- **Aplicação Teste para validar a ferramenta:** inicialmente foi criada uma aplicação que iria servir para testar o funcionamento da ferramenta e dos testes definidos. Depois de implementada cada fase de testes (Robustez I, Robustez II e Segurança), foi usada esta aplicação para verificar se os testes estavam a produzir resultados interessantes e corretos. Os testes que fazem parte das tabelas de injeções foram executados um a um e de forma isolada com o objetivo de verificar se produziam os resultados esperados. Cada teste era repetido pelo menos três vezes para observar se os resultados eram consistentes. Esta aplicação foi também usada para fazer *debugging* ao código da ferramenta, com o objetivo de detetar erros de implementação. Caso não fossem revelados *bugs* na aplicação e tudo estivesse a correr conforme planeado, passávamos à fase seguinte;
- **Introdução da ferramenta em aplicações reais:** aqui foi introduzida a ferramenta numa aplicação real com o intuito de verificar mais uma vez se não existiam *bugs* na implementação. Esta fase revelou-se bastante importante, tendo em conta que proporcionou a deteção de diversos erros que não tinham sido detetados até então e dificilmente seriam se fosse através da análise de código ou em aplicações mais simples. Isto deveu-se à complexidade da aplicação, já que o tempo de processamento desta era bastante maior do que a aplicação testes desenvolvida, o que revelou a origem de *bugs* que só iriam aparecer consoante o tempo de execução. Os testes eram repetidos pelo menos três vezes, tal como na fase anterior, para verificar se a consistência se mantinha.

Este processo de validação da ferramenta permitiu: corrigir inúmeros *bugs* de implementação, detetar código não usado, corrigir testes que não estavam bem definidos e consequentemente a produzir resultados errados, automatizar a ferramenta e eliminar código redundante.

O estudo experimental aqui apresentado baseou-se em seis etapas:

1) **Preparação do ambiente de testes:** recolha de *Web Services* onde iriam ser aplicados os testes definidos;
2) **Configuração e Execução do *Workload*:** configurar e executar o *workload* do sistema para simular um cenário de normal funcionamento da aplicação;
3) **Recolha de Informação:** obtenção da informação sobre a estrutura da aplicação (pontos de acesso à base de dados), de modo a elaborar o mapa pelo qual a ferramenta se vai guiar para aplicar os testes de robustez e segurança;

4) **Execução dos testes de Robustez e Segurança:** uso da ferramenta desenvolvida para detetar vulnerabilidades nos serviços, através da aplicação dos testes referidos;
5) **Verificação dos resultados:** verificação manual dos resultados obtidos na fase anterior para confirmar as vulnerabilidades apresentadas;
6) **Análise dos resultados:** análise e discussão dos resultados obtidos.

Num primeiro cenário foram usados os Testes de Robustez I e II para testar a implementação de sete serviços existentes no TPC-App benchmark [56].

4.1 – Resultados do TPC-App Web Service

A aplicação encontrada para a execução de testes, foi uma aplicação de *benchmark*, cujo nome é *TPC Benchmark App* (TPC-App) [56]. Esta aplicação simula um ambiente de transações *business-to-business*, onde são efetuados pedidos por parte de utilizadores virtuais. Estes utilizadores efetuam transações como se se tratasse de um ambiente real, ou seja, escolhem produtos, fazem compras, entre outros. Esta aplicação usa uma base de dados PostgreSQL onde guarda toda a informação necessária para a simulação e foi desenvolvida por um programador com mais de dois anos de experiência em *Web Services*, base de dados e programação em Java. Apesar de a ferramenta ter sido testada num ambiente que usa uma base de dados PostgreSQL, esta pode ser aplicada noutros sistemas que possuam outras bases de dados. Para isto, basta fazer o *download* do *driver* correspondente à base de dados em questão e das bibliotecas que este usa em *compile-time*. Tendo em conta que a API é a mesma, não precisam de ser feitas quaisquer alterações na ferramenta para que esta funcione noutras bases de dados.

Neste estudo foi usada a ferramenta desenvolvida para testar a robustez e segurança de sete serviços do TPC-App:

- **ProductDetailService:** a informação relacionada com os detalhes dos produtos encontra-se implementada neste serviço;
- **CreateOrderService:** aqui está implementado todo o código necessário para a criação de pedidos desde o método de pagamento, ao cálculo do custo total da compra, à verificação da disponibilidade dos produtos em stock;
- **ChangePaymentMethodService:** alteração do método de pagamento para efetuar as compras;
- **NewProductsService:** todos os novos produtos a serem introduzidos na base de dados passam por este serviço;
- **OrderStatusService:** está implementado todo o processo necessário para verificação do estado da compra do cliente;
- **NewCustomerService:**: este serviço corresponde a todo o processo relacionado com a criação de um novo cliente, resumindo trata toda a informação para adicionar clientes à base de dados;
- **ChangeItemService:** serviço que vai ser usado sempre que é necessário fazer alterações aos produtos existentes (*e.g.*, custo do produto, nome, disponibilidade em stock).

É importante referir que foram feitos diferentes testes, repetidos exaustivamente de modo a comprovar a consistência de resultados.

1) Testes de Robustez I e II

Durante esta fase de testes, foram detetados e testados vinte pontos de acesso à base de dados, dos quarenta e oito existentes. Esta falta de cobertura deveu-se ao facto de na fase de aprendizagem os pedidos não terem passado por aqueles pontos de acesso à base de dados e consequentemente não foram detetados nem testados. Dos vinte testados todos continham problemas de robustez. Os resultados obtidos para os Testes de Robustez I e II, no fim desta fase de avaliação experimental encontram-se apresentados na Figura 4.1.

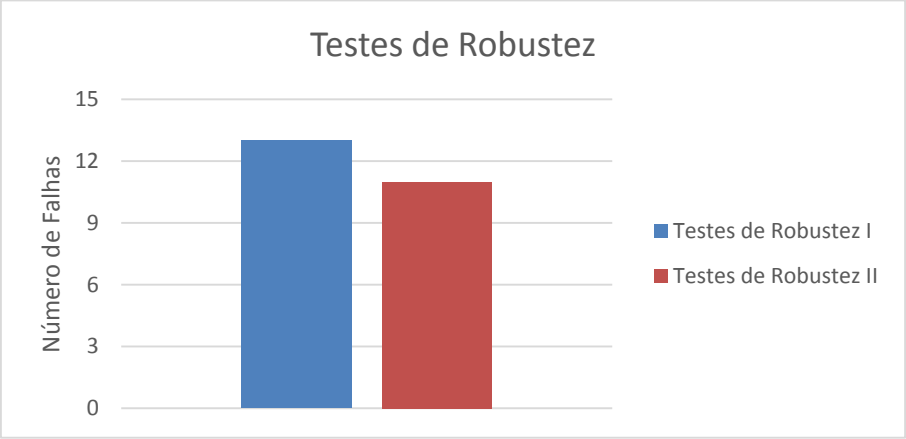

Figura 4.1 – Problemas de Robustez no TPC-App.

Como se pode observar pela Figura 4.1, foi possível detetar treze problemas de robustez diferentes quando a aplicação foi sujeita a fase de Testes de Robustez I. Em relação aos Testes de Robustez II, o número de falhas foi muito aproximado ao do primeiro caso, tendo-se registado onze falhas, menos duas do que o anterior. Esta diferença de duas falhas dos Testes de Robustez I para os Testes de Robustez II, proveio do acumular de injeções consecutivas na aplicação, ou seja, tendo em conta que a aplicação usava resultados retornados pela base de dados para fazer outro tipo de operações (somar, subtrair, multiplicar, dividir), o facto de haver a alteração dos dados em todas as injeções fez com que esta obtivesse, consequentemente, mais exceções. Tendo em conta que os Testes de Robustez II são aplicados segundo um método diferente (uma injeção por pedido do cliente), isto fez com que a aplicação conseguisse contornar certas injeções onde apenas um valor era alterado. Resumindo, a causa para este aumento de duas falhas deveu-se ao acumular de um elevado número de injeções de dados maliciosos. Os resultados apresentados mostram que existe uma falta de verificação dos resultados retornados pela base de dados, levando a ocorrências de falhas quando a aplicação é sujeita a valores não esperados. É pertinente referir também que as restrições em vigor nesta aplicação visam a possibilidade de inserção de valores maliciosos na base de dados, ou seja, segundo código analisado, da aplicação em questão, os valores injetados podem eventualmente ser introduzidos na base de dados caso estes não causem uma exceção. Usando um exemplo mais específico, quando o

cliente acaba de fazer o pedido das suas compras, este vai ser introduzido na base de dados, consequentemente guardando todos os dados maliciosos injetados que não originaram nenhuma exceção, mas que futuramente a podem causar, na eventualidade da aplicação usar aqueles valores para efetuar outras operações (semelhante ao que acontece nos testes de Robustez I com o acumular de informação maliciosa). Esta falta de validação dos dados deve-se ao princípio errado de que tudo o que está dentro do sistema é seguro e não necessita de controlo. Como já referido durante esta dissertação, é fulcral que todas as transações de dados sejam controladas para impedir a presença de vulnerabilidades no sistema. Durante a fase de testes de Robustez I foi ainda documentado um *crash* do sistema. Apesar deste ter sido documentado, não foi possível apurar a/as causa/as que originaram este problema. Algumas das ocorrências encontradas durante a fase de testes de Robustez I e II foram:

- *Numeric Field Overflow*: acontece quando o valor absoluto recebido é maior do que o esperado, originando uma exceção (*e.g.,* valor máximo de *long* + 1);
- *Number Format Exception*: esta exceção ocorre quando se tenta converter um tipo de dados para um número e não é possível devido ao seu formato (*e.g.*, *null*40 para *float*);
- *Null Pointer Exception*: ocorre quando se tenta aceder a um objeto que na realidade é um valor *null* (*e.g.*, multiplicação de *null* por *infinity*)*;*
- *PSQL Exception*: quando ocorre uma exceção relacionada com a base de dados, no estudo feito foram registadas várias ocorrências deste tipo de exceção por diferentes causas (*e.g.,* inserir dados na posição -1 na base de dados).

Caracterizando esta aplicação segundo a escala CRASH apresentada anteriormente, foi possível observar a ocorrência de um *Restart* e treze *Aborts* nos testes de Robustez I e onze *Aborts* nos testes de Robustez II.

Os resultados obtidos para os Testes de Robustez I e II encontram-se detalhados nas Tabelas 4.1 e 4.2 respetivamente, abaixo apresentadas.

Teste	Descrição	Resultado	Detalhes	Justificação	Possível Solução	Código

Valor máximo de *long*	Substituir por valor máximo de *long*	PSQL *Exception: numeric field overflow*	*A field with precision* 9, *scale* 0 *must round to an absolute value less than* 10^9.	O valor absoluto recebido é maior do que o esperado, originando uma exceção "*numeric field overflow*". Como faz addr_id++ vai passar do limite.	Verificar os limites dos tipos de variáveis. O valor absoluto tem de ser < 1000000000. Verificar se é maior que o valor esperado, caso isso se verifique, substituir pelo valor máximo permitido	addr_id = rs.getLong(1);
						addr_id++;
						ps138.setLong(1, addr_id);
						int update = ps1.executeUpdate();

Valor *null*	Substituir por *null*	*Number Format Exception: for input string* "*null*40" *NullPointerException*	N/A	Tenta passar null40 para float o que origina uma exceção "*Number Format Exception*" ao fazer o *parse* Tenta multiplicar *null* por *infinity*. *Float* não pode ter valor *null* ou *Infinity*	Verificar o valor retornado pela base de dados. Este valor não pode ser "*null*" ou qualquer outro valor que não seja números, ".", "," ou "x" Verificar o valor retornado pela base de dados. Só podem ser usados valores com os quais seja possível fazer operações como adicionar, multiplicar, dividir ou subtrair. Estes valores têm de ser do tipo *float*	String token = st2.nextToken(); partialVolume *= Float.parseFloat(token.replace(",", ".")); Float tax = taxRate * subTotal;

Valor mínimo de *long*	Substituir por valor mínimo de *long*	PSQL *Exception: numeric field overflow*	*A field with precision* 9, *scale* 0 *must round to an absolute value less than* 10^9.	O valor absoluto recebido é maior do que o esperado, originando uma exceção "*numeric field overflow*"	Verificar os limites dos tipos de variáveis. O valor absoluto tem de ser < 1000000000. Verificar se é maior que o valor esperado, caso isso se verifique, substituir pelo valor máximo permitido	addr_id = rs.getLong(1);
						addr_id++;
						ps1.setLong(1, addr_id);
						int update = ps1.executeUpdate();

String alfanumérica	Substituir por *string* alfanumérica (abc12)	*Number Format Exception: for input string* "abc1240"		Tenta passar abc1240 para *float* o que origina uma exceção "*Number Format Exception*" ao fazer o *parse*	Verificar o valor retornado pela base de dados. Este valor não pode ser uma *string* que contenha caracteres diferentes de números, ".", "," ou "x"	String token = st2.nextToken(); partialVolume *= Float.parseFloat(token.replace(",", "."));

Valor -1	Substituir por valor -1	PSQL *Exception: ERROR: insert or update on table* "address" *violates foreign key constraint* "addr_co_id"	*Key* (addr_co_id)=(-1) *is not present in table* "country"	O valor recebido viola uma chave estrangeira. O valor -1 não está presente na tabela "country"	Verificar se a posição onde queremos guardar os dados existe. Não é permitida a inserção de valores em que a posição é <1	addrId = rs.getLong(1);
						addrId = new Long(addrId.longValue() + 1);
						ps1.setLong(1,addrId);
						int update = ps1.executeUpdate();

Valor -1 sub	Subtrair 1 ao valor a ser devolvido	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"	*Detail: Key* (addr_id)=(34771) *already exists*	Tenta inserir um valor duplicado que viola a chave primária de "pk_address".	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addrId = rs.getLong(1); addrId = new Long(addrId.longValue() + 1);

Valor 0	Substituir por valor 0	PSQL *Exception: insert or update on table* "address" *violates foreign key constraint* "addr_co_id"
		
		SQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"
			Key (addr_co_id)=(0) *is not present in table* "country".
			
			Key (addr_id)=(1) *already exists.*
				O valor recebido viola uma chave estrangeira. O valor 0 não está presente na tabela "country"
				
				O valor já se encontra na tabela e é uma chave primária de "pk_address". A variável addr_id é incrementada como podemos ver na coluna "Código", daí o valor "1"
					Verificar se a posição onde queremos guardar os dados existe. Não é permitida a inserção de valores em que a posição é <1
					
					Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias
						addr_id = rs.getLong(1);
						addr_id++;
						ps1.setLong(1, addr_id);
						int update = ps1.executeUpdate();

Valor 1	Substituir por valor 1	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"	*Key* (addr_id)=(2) *already exists.*	Tenta inserir um valor duplicado que viola a chave primária de "pk_address". A variável addr_id é incrementada como podemos ver na coluna "Código", daí o valor "2"	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addr_id = rs.getLong(1);
						addr_id++;
						ps138.setLong(1, addr_id);

Valor +1 *plus*	Adicionar 1 ao valor a ser devolvido	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_customer" PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"	*Key* (c_id)=(24576) *already exists.* *Key* (addr_id)=(34408) *already exists.*	Tenta inserir um valor duplicado que viola a chave primária de "pk_customer" ou "pk_address"	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addrId = rs.getLong(1); addrId = new Long(addrId.longValue() + 1);
Non printable char	Substituir por *non printable character*	*NullPointerException*	N/A	Tenta multiplicar um Non printable character por um inteiro. Neste caso sumAllItems vai conter o non *printable character*. SumAllItems é do tipo *float* logo não pode ter este tipo de valor	Verificar o valor retornado pela base de dados. Este valor não pode ser uma *string* que contenha non *printable characters*	subTotal = sumAllItems * (1-(cDiscount/100));

Tabela 4.1 – Tabela de problemas de Robustez I do TPC-App.

Teste	**Descrição**	**Resultado**	**Detalhes**	**Justificação**	**Possível Solução**	**Código**

Valor máximo de *long*	Substituir por valor máximo de *long*	PSQL *Exception: numeric field overflow*	*A field with precision* 4, *scale* 0 *must round to an absolute value less than* 10^4.	O valor absoluto recebido é maior do que o esperado, originando uma exceção "*numeric field overflow*". Como faz addr_id++ vai passar do limite.	Verificar os limites dos tipos de variáveis. O valor absoluto tem de ser < 10000. Verificar se é maior que o valor esperado, caso isso se verifique, substituir pelo valor máximo permitido	addr_id++;
						ps138.setLong(1, addr_id);
						int update = ps1.executeUpdate();

Valor mínimo de *long*	Substituir por valor mínimo de *long*	PSQL *Exception: numeric field overflow*	*A field with precision* 9, *scale* 0 *must round to an absolute value less than* 10^9.	O valor absoluto recebido é maior do que o esperado, originando uma exceção "*numeric field overflow*"	Verificar os limites dos tipos de variáveis. O valor absoluto tem de ser < 1000000000. Verificar se é maior que o valor esperado, caso isso se verifique, substituir pelo valor máximo permitido	addr_id = rs.getLong(1);
						addr_id++;
						ps138.setLong(1, addr_id);
						int rc = ps138.executeUpdate();

Valor -1 sub	Subtrair 1 ao valor a ser devolvido	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"	*Key* (addr_id)=(35062) *already exists*	Tenta inserir um valor duplicado que viola a chave primária de "pk_address"	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addr_id = rs.getLong(1);
						addr_id++;
						ps1.setLong(1, addr_id);
						int update = ps1.executeUpdate();

Valor -1	Substituir por valor -1	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"	*Key* (addr_id)=(34419) *already exists*	Tenta inserir um valor duplicado que viola a chave primária de "pk_address"	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addrId = rs.getLong(1);
						addrId = new Long(addrId.longValue() + 1);
						ps1.setLong(1,addrId);
						int update = ps1.executeUpdate();

Valor 0	Substituir por valor 0	*PSQL Exception: ERROR: insert or update on table* "address" *violates foreign key constraint* "addr_co_id"
		
		PSQL Exception: ERROR: duplicate key value violates unique constraint "pk_address"
			Key (addr_co_id)=(0) *is not present in table "country"*
			
			Key (addr_id)=(1) *already exists*
				O valor recebido viola uma chave estrangeira. O valor 0 não está presente na tabela "country"
				
				Tenta inserir um valor duplicado que viola a chave primária de "pk_address"
					Verificar se a posição onde queremos guardar os dados existe. Não é permitida a inserção de valores em que a posição é <1
					
					Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias
						res = rs.getLong(1);
						ps138.setLong(7, addr_co_id);
						int rc = ps138.executeUpdate();
						
						addr_id = rs.getLong(1);
						addr_id++;
						ps1.setLong(1, addr_id);
						int update = ps1.executeUpdate();

Valor +1 *plus*	Substituir por valor +1	PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_customer"
		
		PSQL *Exception: ERROR: duplicate key value violates unique constraint* "pk_address"
			Key(addr_id)=(34461) *already exists.*
			
			Key(addr_id)=(34419) *already exists*
				Tenta inserir um valor duplicado que viola a chave primária de "pk_costumer" ou "pk_address"	Verificar se o elemento que está a ser criado já existe na base de dados. Não é permitida a duplicação de chaves primárias	addrId = rs.getLong(1);
						addrId = new Long(addrId.longValue() + 1);
						ps1.setLong(1,addrId);
						int update = ps1.executeUpdate();

Null	Substituir por valor *Null*	*Exception: For input string*: "*null*8" *NullPointerException*	N/A	Tenta passar *null*8 para *float* o que origina uma exceção "*Number Format Exception*" ao fazer o *parse* Tenta multiplicar *null* por *Float*. *Float* não pode ter valor *null* ou *Infinity* Tenta multiplicar *null* por *infinity*. *Float* não pode ter valor *null* ou *Infinity*	Verificar o valor retornado pela base de dados. Este valor não pode ser "*null*" ou qualquer outro valor que não seja números, ".", "," ou "x" Verificar o valor retornado pela base de dados. Só podem ser usados valores com os quais seja possível fazer operações como adicionar, multiplicar, dividir ou subtrair. Estes valores têm de ser do tipo *float*	String token = st2.nextToken(); partialVolume *= Float.parseFloat(token.replace(",", ".")); Float tax = taxRate * subTotal;
Non printable char	Substituir por *non printable character*	*NullPointerException*	N/A	Tenta multiplicar um *Non printable character* por um inteiro. Neste caso sumAllItems vai conter o *non printable character*. SumAllItems é do tipo *float* logo não pode ter este tipo de valor	Verificar o valor retornado pela base de dados. Este valor não pode ser uma *string* que contenha non *printable characters*	subTotal = sumAllItems * (1-(cDiscount/100));

Tabela 4.2 – Tabela de problemas de Robustez II do TPC-App.

2) Testes de Segurança

Para aplicar estes testes houve a necessidade de fazer algumas alterações no código do TPC-App, que tinham como objetivo não alterar a lógica dos serviços e ao mesmo tempo demonstrar a aplicação da ferramenta. Estas alterações envolveram apenas a criação de mais algumas instruções SQL para poderem ser efetuados os ataques por *Second-Order* SQL *Injection*. Dos oito acessos à base de dados existentes foi possível detetar quatro, tendo-se verificado vulnerabilidades de segurança em todos eles. Mais uma vez, assim como nos testes de robustez, obteve-se metade da cobertura visto que os pedidos do cliente não passaram pelos restantes pontos de acesso. No final desta fase de testes, foi possível observar a existência de dezasseis falhas diferentes na aplicação TPC-App. Uma vez mais, a falta de validação dos dados provenientes da base de dados, resulta na existência de vulnerabilidades no sistema, as quais podem ser usadas para retirar informação confidencial da aplicação. É de realçar novamente que, uma boa prática de programação é fundamental para reduzir o número de vulnerabilidades nas aplicações *Web*. Durante este procedimento experimental a injeção de valores maliciosos permitiu a obtenção de valores confidenciais tais como:

- Palavras-passe: foi possível retirar da base de dados as palavras-passe de cada utilizador existente na base de dados;
- Nome de contas: os nomes dos utilizadores guardados na base de dados também foram conseguidos através da introdução de instruções SQL maliciosas;
- Detalhes do utilizador: a base de dados retornou também uma quantidade considerável de informação de cada utilizador (*e.g.*, número de telefone, cartões de crédito, métodos de pagamento, moradas);
- Nomes de empresas: os locais de trabalho dos utilizadores foram também adquiridos assim como os respetivos contactos de telefone e respetivas moradas;
- Detalhes dos produtos: informações sobre os produtos existentes na base de dados também foi obtida através destes ataques, no entanto grande parte desta informação não é relevante visto que é acessível aos clientes dos serviços (*e.g.*, nomes de produtos, dimensões, preços, descrição).

Tendo em conta a dimensão da base de dados usada por esta aplicação, limitou-se a dez o número de resultados retornados pela aplicação. Visto que o objetivo desta avaliação experimental era revelar vulnerabilidades no sistema, o número de resultados retornados não é relevante.

Na Tabela 4.3, encontram-se com mais detalhe os resultados obtidos para os testes de segurança.

Teste	Categoria do ataque	Resultado obtido no Serviço	Justificação	Possível Solução
' or 1=1 --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre cartões de crédito de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	Implementação de *Prepared Statements* ou *Stored Procedures*. A verificação de toda a informação retornada pela base de dados também pode evitar este tipo de situações, rejeitando todos os resultados que sejam indicadores de ataques por SQL.
' or "='	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre métodos de pagamento usados pelos utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como vazio = vazio é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	
' or '1'>'0	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre palavras-passe de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como 1 > 0 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	
' or '0'<'1	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre contactos de telefone de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como 0 < 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	

" or "a" = "a	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre nomes de empresas a que os utilizadores existentes na base de dados pertenciam.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como a = a é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
" or 0 = 0 --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre cartões de crédito de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 0 = 0 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
" or 1 = 1 --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre cartões de crédito de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
" or 1 = 1 or "" = "	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre palavras-passe de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.

' or 'x' = 'x	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre palavras-passe de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como x = x é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
x' or 1 = 1 or 'x' = 'y	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre taxas a aplicar sobre os produtos, de países existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
hi' or 'a' = 'a	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre disponibilidade em stock, ids e dimensões de produtos existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira. Como a = a é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.
hi' or 1 = 1 --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre moradas de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.

' or 0 = 0 --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre moradas de utilizadores existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 0 = 0 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	
' or 1 = 1 or '' = '	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre cidades e códigos postais de moradas existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	
' or '1' = '1' --	*Tautology-based attacks*	A instrução injetada fez com que a aplicação retornasse um ResultSet com vários resultados em vez de apenas um, como era suposto. A partir da pesquisa efetuada foi possível obter informação sobre disponibilidade em stock, ids e dimensões de produtos existentes na base de dados.	Visto ser uma tautologia, a instrução SQL depois de sofrer esta injeção vai ser sempre verdadeira, ignorando tudo o que se encontra à frente de " -- ". Como 1 = 1 é sempre verdade, vamos obter todos os dados da base de dados que dizem respeito à tabela e ao campo pesquisado.	

Tabela 4.3 – Tabela de falhas de Segurança do TPC-App.

Capítulo 5 - Planeamento

Neste capítulo é fornecida toda a informação referente ao trabalho desenvolvido durante o ano letivo, tudo o que foi desenvolvido é explicado nas próximas subsecções. Na Secção 5.1 é apresentada toda a evolução que decorreu durante o primeiro semestre. Na Secção 5.2 descreve-se todo o percurso do segundo semestre, com todas as tarefas realizadas. O planeamento para o desenvolvimento da ferramenta durante o ano letivo, que permitiu executar os testes de robustez e segurança já referidos de modo a revelar falhas existentes numa aplicação, dividiu-se nas seguintes fases de construção (ver Figura 5.1).

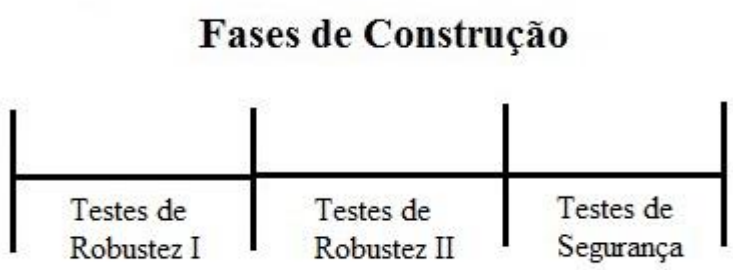

Figura 5.1 – Fases de construção da ferramenta.

Segue-se a descrição de cada uma das fases indicadas acima na Figura 5.1:

- **Testes de Robustez I:** neste cenário surgiram os primeiros testes de robustez. Nesta fase a ferramenta desenvolvida começou a intercetar as ligações e operações com a base de dados, alterando os resultados que esta deveria retornar. Estes resultados foram modificados através da substituição dos dados pela aplicação de um conjunto de regras (para mais detalhes consultar Secção 3.2);
- **Testes de Robustez II:** esta etapa diz respeito aos testes de robustez mais refinados. A diferença entre os Testes de Robustez I e os Testes de Robustez II é que os últimos foram executados de maneira diferente, no que toca ao número de injeções por serviço (ver Secção 3.2);
- **Testes de Segurança:** este último cenário refere-se aos ataques por *Second-Order* SQL *Injection*, onde todos os casos de *Second-Order* SQL *Injection* foram testados, sendo este o principal foco deste trabalho.

5.1 – Planeamento Primeiro Semestre

A deteção de problemas de robustez e presença de vulnerabilidades de tipo *Second-Order* SQL *Injection* é feita através da identificação de pontos de vulnerabilidade existentes na aplicação, pontos estes que poderão ser possíveis alvos, na existência de um ataque. A identificação destas falhas, permite que o programador ou fornecedor do serviço possa proceder à resolução do problema. A Figura 5.2 ilustra o plano efetivo de trabalho que foi desenvolvido no primeiro semestre.

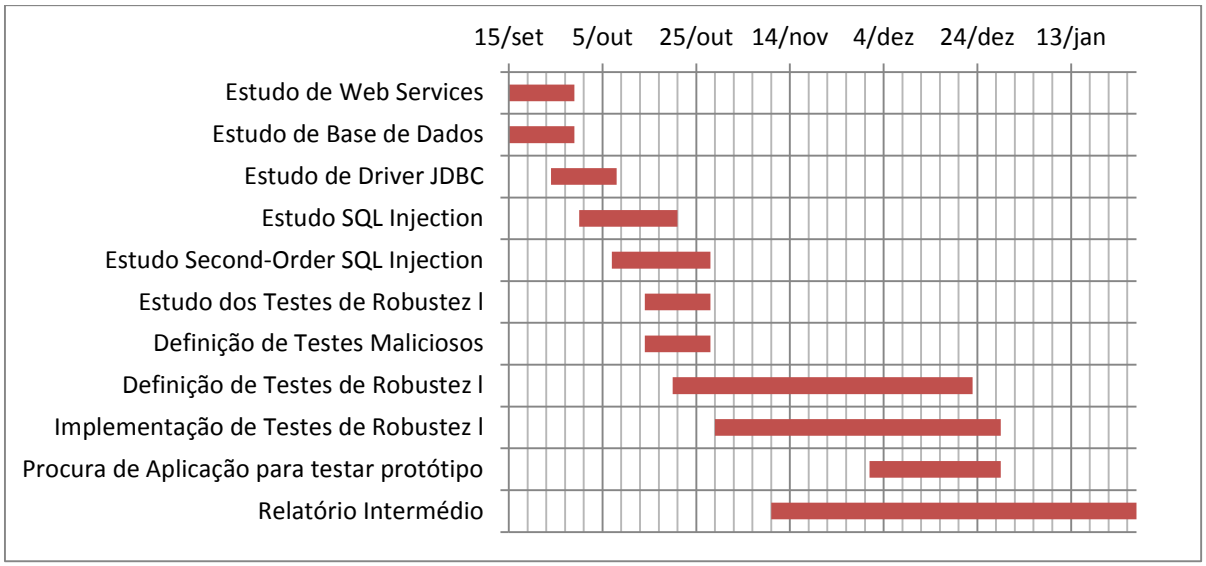

Figura 5.2 – Plano de trabalho efetivo do primeiro semestre.

Analisando a Figura 5.2, o primeiro passo consistiu no estudo da implementação de *Web Services* e Bases de Dados, com o objetivo de relembrar conceitos e aperfeiçoar conhecimentos previamente obtidos. Após esta etapa fez-se o estudo do *driver* JDBC para PostgreSQL. A ferramenta criada, baseou-se em modificações a este *driver*, pelo que o conhecimento do seu funcionamento foi imprescindível. Seguidamente, iniciou-se a tarefa que envolveu o estudo e a recolha de informação sobre SQL *Injection*, recolha esta que serviu para dar início à criação de algumas bases em relação a este assunto. O estudo sobre *Second-Order* SQL *Injection*, sendo o foco principal desta dissertação, consistiu no próximo passo. De seguida, procedeu-se à pesquisa de informação dos Testes de Robustez I e à definição de *Testes Maliciosos* que foram usados pela ferramenta desenvolvida, para a execução dos testes. Da etapa seguinte fez parte a criação de uma tabela com a *Definição de Testes de Robustez I*, a qual contém todos os testes que foram efetuados nas aplicações submetidas a esta ferramenta. Posteriormente, foi criado o protótipo correspondente à tarefa *Implementação de Testes de Robustez I*, que serviu para efetuar os primeiros testes de robustez. Protótipo este que no final do segundo semestre deu origem à ferramenta de testes. Antes da finalização do protótipo, deu-se início à procura de uma aplicação para testar o protótipo. Finalmente, foi elaborado o relatório intermédio.

5.2 – Planeamento Segundo Semestre

No segundo semestre, deu-se continuidade ao trabalho desenvolvido até então. A abordagem apresentada anteriormente foi aprofundada e expandida de modo a ser usada em todos os testes mencionados. O método de execução dos Testes de Robustez I, desenvolvidos no primeiro semestre, foi otimizado. A Figura 5.3 resume todas as tarefas concretizadas ao longo do segundo semestre.

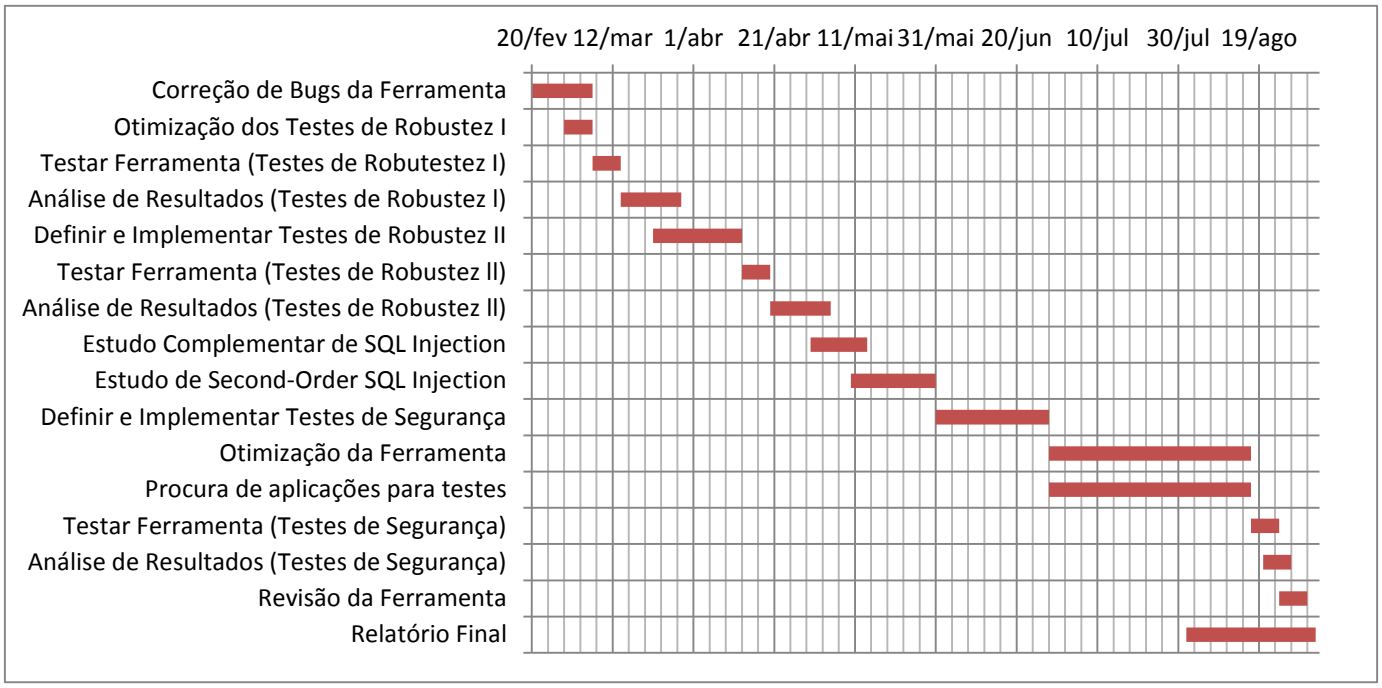

Figura 5.3 – Plano de trabalho efetivo do segundo semestre.

No segundo semestre realizou-se uma correção de *bugs* da ferramenta que foram detetados quando se fizeram alguns testes experimentais. Seguiu-se a *Otimização dos Testes de Robustez I*, que serviu para modificar o método como estes eram executados e o modo como é feita a transição para os Testes de Robustez II. Concretizou-se ainda a tarefa *Testar Ferramenta (Testes de Robustez I)* onde foram executados os testes de Robustez I numa aplicação real (TPC-App). Posteriormente, foi feita a análise dos resultados obtidos nos testes, com o objetivo de verificar quais os danos causados pelos mesmos e qual a classificação obtida pela aplicação em questão. Seguidamente, foram definidos e implementados os Testes de Robustez II, os quais usam a mesma tabela de testes que está definida nos Testes de Robustez I. Finalizada esta implementação, passou-se então à fase de testes da ferramenta, que serviu também para verificar se a mesma estava a funcionar corretamente com a respetiva análise dos resultados obtidos. Após esta etapa, fez-se um estudo complementar de SQL *Injection* onde foram relembrados e aprofundados conhecimentos sobre este tema tendo-se seguido o *Estudo de Second-Order* SQL *Injection*, que teve como meta aprofundar conhecimentos sobre este assunto assim como adquirir informação sobre os testes que iriam ser usados nos futuramente. Depois disto, os Testes de Segurança foram definidos e implementados. A *Otimização da Ferramenta* foi a tarefa seguinte, onde foram feitas todas as alterações no *driver* de modo a que a ferramenta ficasse automatizada o mais possível, para que não fosse necessário fazer quaisquer alterações no sistema a testar. Simultaneamente, foram também executadas as tarefas *Relatório Final* e *Procura de Aplicações para Testes*, que visaram encontrar uma aplicação que satisfizesse os requisitos propostos no início desta dissertação. A fase de Testes de Segurança e a respetiva análise dos

resultados obtidos foi o passo seguinte. Por fim, e previamente à elaboração do Relatório Final, procedeu-se à *Revisão da Ferramenta*, com o objetivo principal de fazer uma última inspeção à ferramenta e a todo o seu conteúdo.

Capítulo 6 - Conclusão

O uso de *Web Services* tem vindo a aumentar consideravelmente, de tal forma que ao aumentar o número de utilizadores que usufrui deste serviço, aumenta também o número de ataques e falhas de segurança existentes. Posto isto, torna-se indispensável a criação de métodos e ferramentas que tenham a capacidade de combater este tipo de problemas, de modo a mitigar, ao máximo, o número de ameaças. O crescimento na área da segurança deve acompanhar a evolução dos *Web Services*, de maneira a estar sempre um passo à frente dos responsáveis pelos ataques que comprometem a segurança destes serviços e dos seus clientes. Uma vez que a nossa era cada vez mais se converte aos sistemas informáticos, é fulcral que os nossos dados estejam seguros e salvaguardados, de maneira a que nenhuma informação confidencial existente nestes sistemas seja exposta.

Apesar da existência de inúmeras ferramentas e abordagens que combatem este problema, este número não é suficiente para garantir a segurança das aplicações existentes. A fraca qualidade no que toca à eficiência das ferramentas comerciais é também um obstáculo para quem tem interesse em aumentar a segurança dos seus serviços. É importante continuar a apostar nesta área e a desenvolver novos e melhorados métodos que ajudem a controlar e reduzir estes ataques.

Nesta dissertação apresenta-se uma abordagem e uma ferramenta desenvolvida e usada para prevenção do aumento indesejado do número de ataques com êxito. A ferramenta é introduzida entre a base de dados e o *Web Service* e tem como objetivo detetar vulnerabilidades de robustez e segurança que possam ser alvos de ataques. Esta deteção é feita através de injeção de valores maliciosos, armazenados em tabelas dentro da ferramenta, durante uma fase de testes de robustez ou segurança. Já a injeção é feita no momento em que a base de dados retorna os resultados de uma pesquisa SQL feita pela aplicação, e consequentemente, a mesma vai receber um resultado malicioso que substitui o resultado original correspondente à pesquisa feita pela aplicação. Caso esta seja vulnerável ao teste em questão, vai ocorrer algum tipo de erro caso seja um teste de robustez, ou retornar valores confidenciais da base de dados, na eventualidade de ser um teste de segurança.

Esta é uma abordagem única, que permite realçar os problemas do lado mais fraco em termos de robustez e segurança da aplicação e mais suscetível a ataques, ou seja, o lado em que a ligação com a base de dados é feita. Os *scanners* existentes, na sua maioria não atacam esta vertente e os que o fazem, são maioritariamente obrigados a alterar a informação da base de dados, alterando o sistema em si, o que não acontece com a ferramenta criada, que é completamente autónoma e não implica a alteração de nenhum componente do sistema.

A Ferramenta apresentada tem, no entanto, algumas limitações. Esta foi desenvolvida para aplicações Java, de tal modo que, só pode ser aplicada em serviços desenvolvidos segundo a mesma linguagem de programação. Além disso, tendo em conta que a ferramenta usa filtros HTTP é necessário a versão 6 do Java EE com a versão Servlet 3.0 para haver uma deteção automática dos filtros. Apesar disso, considera-se que estas limitações não são relevantes, tendo em conta que noutras linguagens existentes, surgem estruturas muito semelhantes o que permite a transação desta técnica

para outros tipos de linguagens, tratando-se apenas de uma questão de implementação para que a técnica possa ser aplicada em outros ambientes com linguagens diferentes.

Relativamente ao trabalho efetuado, este contém também algumas limitações, tendo em conta que estava programada a elaboração de um artigo científico que infelizmente não foi possível terminar, por falta de tempo e de conteúdo no que toca a avaliação experimental. Esta última foi outra limitação do trabalho, visto que estava planeada a introdução da ferramenta em ambientes reais o que acabou por não ser possível, à exceção do caso do TPC-App.

6.1 – Obstáculos Encontrados

Durante o decorrer deste ano letivo, foram encontrados diversos problemas que influenciaram o resultado obtido. Inicialmente a principal dificuldade foi a aprendizagem de tudo o que envolve *Web Services*, tendo em conta que as bases adquiridas até ao início deste ano letivo eram muito limitadas em relação a este tema. Para ultrapassar este problema, foi necessário um estudo considerável sobre esta temática de modo a perceber o funcionamento dos mesmos e de tudo o que envolve a sua criação. Foi importante relembrar também alguns conhecimentos de bases de dados para poder desenvolver os testes de segurança e para que estes produzissem o resultado esperado.

Tudo o que envolve SQL *Injection* e *Second-Order* SQL *Injection*, foi também uma novidade no meu percurso académico, mas ao mesmo tempo uma mais-valia relativamente ao conhecimento adquirido. Foi necessária muita dedicação e pesquisa para entender esta temática e o que ela envolve: desde os métodos como são feitos estes ataques, às regras pelas quais estes se executam, especialmente *Second-Order* SQL *Injection*, que acaba por ter uma complexidade de maiores dimensões do que os típicos ataques por *First-Order* SQL *Injection*.

A implementação relacionada com filtros HTTP, Soap Handlers e AspectJ trouxe também alguma dificuldade ao desenvolvimento da ferramenta, devido à falta de experiencia relacionada com estes componentes em Java. Posto isto, foi necessário um período de familiarização, para obter conhecimentos sobre estes, de modo a que fosse possível a sua aplicação na ferramenta desenvolvida. O maior tempo despendido foi com os filtros HTTP, devido a *bugs* existentes no código que perduraram durante alguns dias até finalmente terem sido detetados.

O principal obstáculo encontrado neste ano letivo foi, sem dúvida alguma, a recolha e configuração de aplicações para efetuar os testes. Este teve como origem a falta de *Web Services* disponíveis *online* que se adaptassem a este trabalho. Os problemas foram vários e entre eles destacam-se:

- **Dificuldade em encontrar *Web Services*:** encontrar *Web Services* que se aplicassem às nossas especificações ou que fossem adequados para uma dissertação de Mestrado em Engenharia Informática, foi o principal problema;

- **Difícil Configuração de *Web Services*:** a difícil configuração dos poucos *Web Services* encontrados foi outro problema, tendo como consequência o dispêndio de uma quantidade de horas considerável. Houve inclusive alguns casos onde se ocupou bastante tempo na configuração de aplicações para Jboss, já inoperacionais devido à descontinuidade do projeto.

A presença de cadeiras de mestrado durante este ano foi também um fator inibidor importante que condicionou o tempo disponível, principalmente devido à complexidade destas e ao facto de duas delas fazerem parte do plano de estudos do segundo semestre. Foi sentida alguma dificuldade na gestão do tempo para as cadeiras e a dissertação.

6.2 – Trabalho Futuro

A abordagem e ferramenta aqui apresentadas foram desenvolvidas com o intuito de que estas pudessem ser usadas futuramente por outros tanto a nível experimental, como de otimização ao nível de implementação, aumentando assim o leque de testes disponíveis que possam oferecer um maior nível de cobertura às aplicações sob teste.

Outro objetivo a curto e médio prazo seria o de ultrapassar as limitações existentes nesta ferramenta, como por exemplo, desenvolver outra ferramenta baseada na mesma abordagem mas numa outra linguagem de programação para que o número de *Web Services* que possam ser alcançados seja maior.

A implementação de outros testes de segurança para outros tipos de ataques nesta ferramenta também seria bastante interessante de modo a aumentar o impacto que esta pode ter nas aplicações existentes, revelando deste modo um maior número de falhas para diferentes tipos de ataques.

Referências

[1] D. Stuttard, *The web application hacker's handbook: discovering and exploiting security flaws*. Indianapolis, IN: Wiley Pub, 2008.

[2] Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer, *Applied SOA: Service-Oriented Architecture and Design Strategies*, 1st ed. Indianapolis, IN: WIley Pubishing, Inc, 2008.

[3] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Computer Society*, vol. 6, no. 2, pp. 86–93, 2002.

[4] Thomas Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*, 1st ed. Upper Saddle River: Prentice Hall, 2005.

[5] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." Web Services Activity: XML Protocol Working Group, 2007.

[6] Doug Tidwell, "Introduction to XML." IBM, 2002.

[7] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture," *IBM systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[8] Oracle, "Database Web Services," 2002. [Online]. Available: http://www.oracle.com/technetwork/testcontent/database-web-services-133122.pdf. [Accessed: 05-Aug-2015].

[9] Oracle, "Oracle JDBC Memory Management," 2009. [Online]. Available: http://www.oracle.com/technetwork/topics/memory.pdf. [Accessed: 05-Aug-2015].

[10] M. E. Khan, F. Khan, and others, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 3, no. 6, pp. 12–15, 2012.

[11] Testing Whiz, "Software Testing Services: Understanding White box Testing and Black box Testing Approaches," 2011. [Online]. Available: http://blog.testing-whiz.com/2011/11/understanding-white-box-testing-and.html. [Accessed: 05-Aug-2015].

[12] S. Nidhra and J. Dondeti, "Black Box and White Box Testing Techniques - A Literature Review," *International Journal of Embedded Systems and Applications*, vol. 2, no. 2, pp. 29–50, Jun. 2012.

[13] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*, 2nd ed. Hoboken, N.J: John Wiley & Sons, 2004.

[14] M. E. Khan and others, "Different approaches to white box testing technique for finding errors," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 3, pp. 1–14, 2011.

[15] A. G. Bardas, "Static Code Analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99–107, 2010.

[16] "Static Code Analysis - OWASP," 2015. [Online]. Available: https://www.owasp.org/index.php/Static_Code_Analysis. [Accessed: 22-Aug-2015].

[17] S. Singh, E. S. Singh, and M. Rakshit, "A Review of Various Software Testing Techniques," *IJREAT International Journal of Research in Engineering & Advanced Technology*, vol. 1, no. 4, pp. 1–4, 2013.

[18] A. C. Coulter, “Graybox Software Testing in Real-Time in the Real World.” Cleanscape, 2010.
[19] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, “MAFALDA: Microkernel Assessment by fault injection and design aid,” presented at the 3rd European Dependable Computing Conference, EDCC-3, Prague, Czech Republic, 1999, pp. 143–160.
[20] P. Koopman and J. DeVale, “Comparing the Robustness of Posix Operating Systems,” presented at the 29th Annual International Symposium on Fault-Tolerant Computing, Madison - Wisconsin, 1999, pp. 1–8.
[21] C. P. Shelton, P. Koopman, and K. DeVale, “Robustness testing of the Microsoft Win32 API,” in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 261–270.
[22] J. Pan, P. Koopman, Y. Huang, R. Gruber, and M. L. Jiang, “Robustness testing and hardening of CORBA ORB implementations,” in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, 2001, pp. 141–150.
[23] M. Rodriguez, A. Albinet, and J. Arlat, “MAFALDA-RT: a tool for dependability assessment of real-time systems,” presented at the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2002, 2002, pp. 267–272.
[24] E. Marsden and J.-C. Fabre, “Failure mode analysis of CORBA Service Implementations,” in *Middleware 2001*, 2001, pp. 216–231.
[25] M. Mendonca and N. Neves, “Robustness testing of the Windows DDK,” in *Dependable Systems and Networks, 2007. DSN’07. 37th Annual IEEE/IFIP International Conference on*, 2007, pp. 554–564.
[26] R. Siblini and N. Mansour, “Testing web services,” in *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, 2005, p. 135.
[27] W. Xu, J. Offutt, and J. Luo, “Testing web services by xml perturbation,” in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, 2005, p. 10–pp.
[28] M. G. Fugini, B. Pernici, and F. Ramoni, “Quality analysis of composed services through fault injection,” in *Business Process Management Workshops*, 2008, pp. 245–256.
[29] S. H. Kuk and H. S. Kim, “Robustness testing framework for web services composition,” in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, 2009, pp. 319–324.
[30] I. Rabhi, “Robustness Testing of Web Services Composition,” 2012, pp. 631–638.
[31] K. Cong, L. Lei, Z. Yang, and F. Xie, “Automatic Fault Injection for Driver Robustness Testing,” 2015.
[32] S. de Vries, “A Corsaire White Paper: A Modular Approach to Data Validation in Web Applications,” 2006.
[33] C. Brandão, F. Braz, L. Militelli, M. Rodrigues, M. Hamada, and R. Montoro, “As 10 vulnerabilidades de segurança mais críticas em aplicações WEB,” 2007. [Online]. Available: https://www.owasp.org/images/4/42/OWASP_TOP_10_2007_PT-BR.pdf. [Accessed: 28-Feb-2015].
[34] A. Chopra and M. Kaufman, “Rise in Web Application Intrusions 2013-2014,” 2014.
[35] Chris Wysopal, “STATE OF SOFTWARE SECURITY,” Veracode, Burlington, Massachusetts, 6, 2015.

[36] K. Elshazly, Y. Fouad, M. Saleh, and A. Sewisy, "A Survey of SQL Injection Attack Detection and Prevention," *Journal of Computer and Communications*, vol. 02, no. 08, pp. 1–9, 2014.
[37] A. John, "SQL Injection Prevention by Adaptive Algorithm," *IOSR Journal of Computer Engineering*, vol. 17, pp. 19–24, 2015.
[38] Sampada Gadgil, Sanoop Pillai, and Sushant Poojary, "SQL INJECTION ATTACKS AND PREVENTION TECHNIQUES," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 1, no. 4, pp. 293–296, 2013.
[39] G. Leonard and S. Sims, "Securing Web Applications Made Simple and Scalable," 2013. [Online]. Available: http://www.sans.org/reading-room/whitepapers/analyst/securing-web-applications-simple-scalable-34970. [Accessed: 28-Feb-2015].
[40] N. Antunes and M. Vieira, "Detecting SQL Injection Vulnerabilities in Web Services," 2009, pp. 17–24.
[41] Chris Anley, "Advanced SQL Injection In SQL Server Applications," NGSSoftware Insight Security R e search (NISR), 2002.
[42] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures.pdf," presented at the Australian Software Engineering Conference (ASWEC'06), Dept. of Electrical and Computer Engineering Iowa State University, 2006, pp. 1–8.
[43] OWASP, "SQL Injection," 2014. [Online]. Available: https://www.owasp.org/index.php/SQL_Injection. [Accessed: 10-Aug-2015].
[44] L. Yan, X. Li, R. Feng, Z. Feng, and J. Hu, "Detection Method of the Second-Order SQL Injection in Web Applications," in *Structured Object-Oriented Formal Language and Method*, vol. 8332, S. Liu and Z. Duan, Eds. Cham: Springer International Publishing, 2014, pp. 154–165.
[45] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, 2009, pp. 47–53.
[46] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," 2008, pp. 77–86.
[47] F. Di Gregorio and D. Varrazzo, "Psycopg – PostgreSQL database adapter for Python — Psycopg 2.6 documentation," 2015. [Online]. Available: http://initd.org/psycopg/docs/. [Accessed: 28-Feb-2015].
[48] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.
[49] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001—Object-Oriented Programming*, Springer, 2001, pp. 327–354.
[50] D. L. Stuttard, *The web application hacker's handbook: finding and exploiting security flaws 2e*, 2nd ed. Indianapolis, IN: WIley Pubishing, Inc, 2011.
[51] Web Application Security Consortium, "WASC Threat Classification." 2010.
[52] M. Vieira, N. Antunes, and H. Madeira, "Using Web Security Scanners to Detect Vulnerabilities in Web Services," CISUC, Department of Informatics Engineering University of Coimbra, CISUC, Department of Informatics Engineering University of Coimbra – Portugal, 2009.
[53] N. Laranjeiro, M. Vieira, and H. Madeira, "A robustness testing approach for SOAP Web services," *Journal of Internet Services and Applications*, vol. 3, no. 2, pp. 1–19, 2012.

[54] Atefeh Tajpour, Maslin Massrum, and Mohammad Zaman Heydari, "Comparison of SQL injection detection and prevention techniques," 2010, pp. V5–174–V5–179.
[55] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006, vol. 1, pp. 13–15.
[56] Transaction Processing Performance Council, "TPC Benchmark App (Application Server) Standard Specification, Version 1.3," 2008. [Online]. Available: http://www.tpc.org/tpc_app/default.asp. [Accessed: 01-Mar-2015].

Anexo A

Neste anexo, na Tabela A.1, encontram-se as regras que foram definidas para os Testes de Segurança.

Regras	Categoria do Ataque	Mutação de Parâmetro
Produzidas	*Tautology-Based Attack*	Substituir por: ‘ OR 1 = 1 ‘ OR 1 = 1 -- ‘ OR 1 > 0 ‘ OR 0 < 1 ‘) OR 1 = 1 -- ‘) OR (1 = 1 – 1' OR '1' = '1')) /* ‘ OR ‘1’ = ‘1’ -- ‘ OR ‘x’ = ‘x ‘ OR 0 = 0 -- ‘ OR 1 = 0 -- OR 0 = 0 -- OR 1 = 1— ‘ OR ‘b’ > ‘a -- ‘ OR ‘a’ < ’b ‘ OR a <= b ‘ OR c >= c -- ‘ OR a LIKE a -- ‘ OR a NOT LIKE b ‘ OR c NOT IN (d, e, f) ‘ OR b IN (a, b, c) -- ‘ OR d BETWEEN c AND e -- ‘ OR b NOT BETWEEN c AND d “ OR 1 = 1 OR ““ = “ ‘ OR 1 = 1 OR ‘‘ = ‘
	Piggy-Back Queries	Substituir por: ‘; SELECT * FROM information_schema.tables - - ';SELECT table_name into hack FROM information_schema.tables where table_type = 'BASE TABLE' and table_name not like '%pg%' and table_name not like '%sql%' Limit 1 -- ‘; DROP table users; - - ‘; DROP table utilizadores; - - ‘; DROP table clients; - - ‘; SELECT * into hack FROM information_schema.tables; /* ’; SHUTDOWN; -- ‘; SELECT top 1 FROM users – ‘; DROP table members; - -
	Union *Queries*	Substituir por: ‘ UNION SELECT * FROM users - - ‘ UNION all SELECT * FROM users - - ‘ UNION SELECT * FROM members where id = 1 – ‘ UNION ALL SELECT NULL,concat(TABLE_NAME) FROM information_schema.TABLES WHERE table_type = 'BASE TABLE' and

		table_name not like '%pg%' and table_name not like '%sql%'
	Others	Substituir por: ’ OR ‘’ = ‘ “ OR “” = “ ‘ – ‘ ; ‘ AND 1=0 -- ‘ AND 1=1 – HAVING INSERT LIKE LIMIT ORDER BY SELECT UPDATE ' AND 1=(SELECT COUNT(*) FROM information_schema.tables); -- ' OR 1=(SELECT COUNT(*) FROM information_schema.tables); --
Adaptadas	*Tautology-Based Attack*	Substituir por: “ OR “a” = “a “ OR 0 = 0 -- “ OR 0 = 0 # “ OR 1 = 0 -- “ OR 1 = 1 -- ‘ OR 0 = 0 # OR 0 = 0 # ‘) OR (‘a’ = ‘a “) OR (“a” = “a 2989 OR 1 = 1 -- hi’ OR ‘a’ = ‘a hi’ OR 1 = 1 -- hi’) OR (‘a’ = ‘a hi” OR “a” = “a hi” OR 1 = 1 -- hi”) OR (“a” = “a ‘hi’ OR ‘x’ = ‘x’; 0 OR 1 = 1 ‘ OR a = a -- x’ OR 1 = 1 OR ‘x’ = ‘y
	Alternate Encondings	Substituir por: ‘; exec(0x73689574646f776e) --
	Union *Queries*	Substituir por: ‘ UNION ALL SELECT ‘ UNION SELECT ‘ UNION ALL SELECT NULL,concat(schema_name) FROM information_schema.schemata--

	Piggy-Back Queries	Substituir por: ‘; exec master..xp_cmdshell; ‘; exec xp_regread; ‘; SHUTDOWN — ’;exec master..xp_cmdshell ‘nslookup www.google.pt’-- ‘;exec sp ‘;exec xp \"; DROP table users; -- ‘; SELECT user FROM mysql.user; — ‘;SELECT 1; #comentário ‘;SELECT /*comentário*/1; ‘;SELECT user(); ‘;SELECT system_user(); ‘;SELECT host, user, password FROM mysql.user; -- ‘;SELECT grantee, privilege_type, is_grantable FROM information_schema.user_privileges; — ‘;SELECT table_schema, table_name, column_name, privilege_type FROM information_schema.column_privileges; — ‘;SELECT grantee, privilege_type, is_grantable FROM information_schema.user_privileges WHERE privilege_type = ‘SUPER’; -- ‘;SELECT host, user FROM mysql.user WHERE Super_priv = ‘Y’; -- ‘;SELECT database();-- ‘; SELECT schema_name FROM information_schema.schemata; -- ‘;SELECT distinct(db) FROM mysql.db — ‘;SELECT table_schema, table_name, column_name FROM information_schema.columns WHERE table_schema != ‘mysql’ AND table_schema != ‘information_schema’; -- ‘;SELECT table_schema,table_name FROM information_schema.tables WHERE table_schema != ‘mysql’ AND table_schema != ‘information_schema’ ‘;CREATE USER hack IDENTIFIED BY ‘inj′; -- ‘; DROP USER admin; --
	Others	Substituir por: ‘ OR (EXISTS) 1’ || ‘ 1’ AND ‘1’ = ‘0 1’ AND ‘1’ = ‘1 1’ + lower(‘‘) + upper(‘‘) +’ 1” AND “1” = “0 1” AND “1” = “1 OR - -- # ‘ ‘ (SELECT top 1 --’; ‘ OR SELECT * “ “ AND 1 = 0 -- “ AND 1 = 1 -- “ ‘ OR 1 --’ ” = -- = ‘ = ; 1 AND 1 = 0 1 AND 1 = 1 1+31337

		1+31337-31337 2989 AND 1 = 0 -- 2989 AND 1 = 1 -- 2989 OR 1 = 0 -- admin'-- AS ASC DELETE DESC DISTINCT ())[]]{}} ,@variable @variable ' OR uname like '% ' OR userid like '% ' OR username like '% ,`@^*$;# PRINT PRINT @@variable PROCEDURE REPLACE --sp_password to_timestamp_tz truncate tz_offset bfilename SELECT @@version

Tabela A.1 – Tabela de Testes de Segurança.