UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

PLATAFORMA PARA TESTES DE SEGURANÇA EM APLICAÇÕES WEB

Inês Pereira Afonso

MESTRADO EM SEGURANÇA INFORMÁTICA

Trabalho do projeto orientado por:
Prof. Doutor Mário João Barata Calha

2020

Agradecimentos

Diferentes pessoas têm diferentes impactos na vida de um jovem. Ao longo desta jornada, cheia de altos e baixos, tive a sorte de ser acompanhada por várias pessoas e todas elas, direta ou indiretamente, trouxeram um contributo para o desenvolvimento e conclusão deste projeto.

Gostava primeiro de agradecer a quem me acompanhou na primeira linha, o meu colega, amigo e chefe, António Orlando, sem o qual nada disto teria sido possível, e que para além de toda a paciência e dedicação que teve, trouxe, e traz um valor pedagógico ímpar para a minha educação e crescimento pessoal. Quero também agradecer ao meu orientador, o professor Doutor Mário João Barata Calha, que também na primeira linha, desempenhou um papel crucial nesta jornada, trazendo ordem e disciplina onde ela faltava.

Gostaria também de agradecer à instituição de acolhimento, a Escrita Digital e a todos os seus colaboradores, pela oportunidade que me foi dada, e por toda a disponibilidade, flexibilidade e apoio que me deram.

Aos meus amigos que desempenharam um papel crucial neste processo, dando-me a motivação e forças sempre que estas faltavam, quero deixar um agradecimento especial à Madalena Andrade, Martim Gubert, Manuel Teixeira e Frederico Portas.

Por fim, nada disto teria sido possível sem o apoio incondicional da minha família, o meu pai, a minha mãe e a minha irmã, que sempre acreditaram em mim e deram-me todas as oportunidades para crescer da melhor forma e escolher o que queria fazer. Em especial à minha irmã, que me relembrou nos momentos mais difíceis, da beleza do mundo.

Resumo

A Internet revolucionou o mundo no qual vivemos, interligando milhões de pessoas diariamente. É uma tecnologia em recorrente expansão, com novas aplicações a serem disponibilizadas dia após dia. Cada vez mais, grandes infraestruturas como bancos e setores comerciais apostam neste novo meio e como tal a sua segurança nunca foi mais importante. Infelizmente vivemos numa situação onde a insegurança é uma realidade, o que torna este meio por vezes perigoso. Muitas vezes, esta insegurança advém de erros no código fonte das aplicações, causados pelos programadores que por vezes não têm o conhecimento ou práticas necessárias para a produção de soluções confiáveis e seguras.

Esta tese tenta contribuir um pouco para este problema, apresentando um estudo e desenvolvimento de uma ferramenta desenhada para auxiliar o programador a encontrar estes erros a tempo. Das duas grandes técnicas utilizadas para testar aplicações web, esta insere-se na técnica de *black-box*, no qual é assumido que não existe acesso ao código fonte. Ao longo do documento transportamos o utilizador para o meio da segurança informática, apresentando um estudo sobre vulnerabilidades em aplicações web e as suas técnicas de deteção, ferramentas já existentes no mercado, e por fim apresentamos a arquitetura e implementação para uma nova solução de software, focada em testes *black-box*, com recurso à técnica de *fuzzing*, que, ao contrário das estudadas, permite ao utilizador uma maior configurabilidade de uso, entre outras.

Palavras-chave: *fuzzing*, vulnerabilidades, *black-box*, segurança informática

Abstract

The Internet revolutionized the world that we live in, connecting millions of people daily. It is a technology in expansion, having new web applications being added day after day. More and more businesses like banks or retail-shops are going online, and for the that the safety of these application is of the utter most importance. Unfortunately, we live in world where the insecurity of web application is a reality, and so it can be a rather dangerous world. This insecurity is usually caused by errors in the applications source code, which are normally caused by the programmers themselves, for not having enough knowledge to deploy safe and reliable applications.

This thesis tries to contribute to this gap, by presenting a new tool aimed to help the detection of these errors before the applications are deployed. From the two major techniques used to test web applications, the one used, is a black-box technique, which test the application without needing access to their source code. Throughout this document we will transport the reader to the world of cybersecurity, presenting a study on web vulnerabilities and the techniques to detect them, the already existing tools on the market, and last but not least we present an architecture and implementation of a tool that follows one of this techniques, that uses a fuzzing approach to carry on the tests to the applications.

Keywords: fuzzing, vulnerabilities, black-box, cybersecurity

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A invenção do primeiro browser juntamente com a comercialização da Internet mudou drástica e abruptamente as nossas comunicações e economia. Vivemos agora num mundo onde, segundo o relatório da Internet World Stats [21], $\simeq$ 4.5 milhares de milhões de utilizadores estão interligados, contabilizando mais de 3.5 triliões de dólares em transações online em 2019 nos Estados Unidos [50]. O impacto que este sector apresenta torna a sua segurança imprescindível. Quebrá-la pode levar a variadas consequências como indisponibilidade de serviços, roubo de informação, violação de privacidade, entre outras. Em alguns casos as consequências podem assumir uma dimensão ainda maior ameaçando a própria existência da organização e originando grandes prejuízos financeiros.

Devido ao volumoso número de ataques a aplicações web sofridos nos últimos anos a segurança é um dos grandes desafios atuais e, como tal, tem sido alvo de bastante investimento e investigação. É um componente que deveria estar presente em todas as fases do desenvolvimento das soluções, desde o design, à implementação, aos testes, mas, no entanto, continuam a existir erros no código fonte e na arquitetura das soluções, resultando num considerável leque de vulnerabilidades. Uma das formas de combater este problema é através do uso de ferramentas de segurança, que, diferenciando-se pelo conhecimento que o utilizador tem sobre o sistema, podem ser ferramentas de teste *black-box* onde o atacante tem pouca ou nenhuma informação sobre o sistema, ferramentas *white-box*, onde existe acesso ao código fonte ou *grey-box*, um híbrido dos anteriores [32]. Estas ferramentas são um forte aliado na conceção das aplicações, expondo vulnerabilidades durante a fase de produção, poupando vastas horas de correções e reduzindo a janela de exposição.

As ferramentas de testes *black-box*, simulam cenários de ataque reais, testando a aplicação na perspetiva de um atacante, onde, normalmente, não existe nenhuma informação à *priori* sobre os sistemas. No entanto, são de difícil automatização pois têm uma grande dependência do utilizador para obter boas taxas de sucesso e têm tendência a encontrar apenas erros simples. Não obstante, estas ferramentas completam o processo de

produção verificando se todos os componentes de segurança de um sistema estão a funcionar corretamente e identificando potenciais vulnerabilidades de segurança resultantes de erros de implementação por parte do programador. Por outro lado, as ferramentas *white-box* testam a aplicação na perspetiva do programador e são facilmente automatizadas, no entanto a exaustão dos testes cresce muito com o tamanho da aplicação.

De forma a melhorar o tempo de produção e teste, a instituição de acolhimento deste projeto descrita na Seção 1.4 optou por utilizar ferramentas *black-box open-source*, mais especificamente ferramentas que utilizam a técnica de *fuzzing*, para tentar automatizar o processo de teste. Para tal, foram formadas duas equipas, uma responsável pelas correções e outra pelos testes (onde eu estava incluída). À equipa de testes foi atribuída a ferramenta *Zed Attack Proxy* (ZAP) da *Open Web Application Security Project* (OWASP), que segundo [41] é uma das ferramentas de teste de segurança *open-source* mais populares. A equipa de testes tinha como objetivo verificar os ficheiros corrigidos previamente pela equipa de correções com a ferramenta e durante cerca de um mês e meio validou mais de três mil ficheiros de *Active Server Pages* (ASP) clássico.

A metodologia inicialmente adotada pela equipa de testes recorria aos testes de *fuzzing default* da ferramenta. Infelizmente, estes testes revelaram-se inúteis devido à sua exaustão e à alta taxa de falsos negativos apresentada. Numa segunda tentativa foi elaborado um *script* manualmente, onde foi possível reduzir o número de valores dos parâmetros a testar, reduzindo um pouco o tempo de execução dos testes. No entanto, ainda existia o problema de identificação de *inputs* que resultava numa alta taxa de falsos negativos e para corrigir isso a equipa teve de recorrer a mecanismos externos que permitissem alimentar o ZAP com informação que mapeasse corretamente toda a aplicação.

Mesmo depois de aplicadas estas correções, o processo continuou bastante moroso e manual, sendo ainda necessária muita intervenção do utilizador o que pôs um travão em todo o processo de produção. É o objetivo da instituição de acolhimento adquirir uma solução capaz de executar testes de segurança *black-box* configuráveis, que executem em tempo útil e que seja de fácil utilização.

1.2 Objetivos

Esta tese é desenvolvida no contexto da segurança informática e apresenta uma ferramenta *black-box* para testes de segurança a aplicações web mais capaz e com maior capacidade de configuração do que as apresentadas no Capítulo 2. Esta ferramenta é *black-box*, não partindo de nenhum conhecimento sobre a aplicação que testa e recorre à técnica de *fuzzing* para executar os testes. Em adição é também apresentado um estudo sobre a técnica utilizada e sobre as ferramentas *open-source* da mesma categoria já existentes. Como tal este projeto tem como objetivos os seguintes:

- Investigação e documentação da técnica de *fuzzing*. Classificação e tipologia da

mesma;

- Estudo de algumas das mais frequentes e perigosas vulnerabilidades web, detetáveis pela ferramenta apresentada. Esclarecimento de como são originadas, exploradas e como impedir o seu aparecimento;

- Estudo e investigação sobre as ferramentas de teste *black-box open-source* existentes que recorrem à técnica de *fuzzing*. Documentação das vulnerabilidades sobre as quais atuam, as suas garantias, o seu desempenho e taxa de sucesso e erro;

- Elaboração de uma arquitetura para uma ferramenta de teste *black-box* que recorre à técnica de *fuzzing*. Esta arquitetura permite uma customização dos testes a executar, podendo configurar gamas de valores manualmente ou automaticamente, para posterior injeção nos *inputs*;

- Implementação da arquitetura e sua respetiva avaliação. Avaliação com base em testes feitos a duas aplicações diferentes, uma feita à medida e outra uma das soluções comercializadas pela instituição de acolhimento.

1.3 Contribuições

Como tal, este projeto traz as seguintes contribuições:

- Elaboração de uma arquitetura para uma ferramenta de teste *black-box* que recorre à técnica de *fuzzing*. Esta arquitetura permite uma customização dos testes a executar, podendo configurar gamas de valores manualmente ou automaticamente, para posterior injeção nos *inputs*;

- Implementação da arquitetura e sua respetiva avaliação. Avaliação com base em testes feitos a duas aplicações diferentes, uma feita à medida e outra uma das soluções comercializadas pela instituição de acolhimento.

1.4 Instituição de acolhimento

Esta dissertação/projeto foi realizada no âmbito da empresa Escrita Digital, fundada em Outubro de 2001 [10]. Esta é uma empresa de software que trabalha com aplicações web, desenvolvendo soluções em ASP clássico e na *framework* ASP.NET. Foi em tempos uma empresa de software à medida, evoluindo para um mercado mais restrito onde passou a disponibilizar aplicações finais, nomeadamente um portal de gestão de conteúdos e o mais complexo XRP que envolve diferentes soluções, todas relacionadas com a gestão de processos, desde a área de recursos humanos à de gestão de frotas.

A Escrita Digital é pioneira em alguns destes mercados sendo a principal fornecedora de empresas multinacionais de renome. Infelizmente, como tantas outras empresas de software, tem experienciado ao longo dos anos dificuldades na área da segurança informática, que é, cada vez mais, um requisito dos clientes. Como tal, este projeto visa beneficiar e melhorar todo este processo imprescindível ao desenvolvimento de uma solução integra.

1.5 Estrutura do documento

Este documento está dividido por capítulos, sendo que, no segundo capítulo, será apresentado o estudo que foi feito à segurança das aplicações web. Neste, será apresentada uma análise às vulnerabilidades web e como estas podem ser detetadas a tempo. As técnicas utilizadas para esta deteção serão posteriormente apresentadas, tal como algumas ferramentas *open-source* disponíveis no mercado que recorrem a estas técnicas.

De seguida, no terceiro capítulo, serão apresentados todos os requisitos funcionais e não funcionais, casos de uso e arquitetura de uma ferramenta que implementa uma das técnicas apresentadas, e serão também mostrados outros detalhes funcionais, desenhados para auxiliar e simplificar todo o processo de implementação.

No quarto capítulo é relatada a experiência da implementação da arquitetura mostrada no capitulo anterior, realçando aspetos globais de implementação e alguns dos pontos onde os maiores desafios foram encontrados e como foram superados. De seguida são apresentados todos os testes feitos ao sistema e por fim as suas validações, que clarificam que requisitos e objetivos foram cumpridos e quais não foram.

Por fim, no últimos capitulo, é apresentada uma reflexão sobre todo o trabalho e estudo e algumas notas sobre possíveis futuras alterações.

Capítulo 2

Contexto e Trabalho relacionado

As boas práticas de programação partem do principio que o programador conhece todos os pontos de entrada num programa e nunca assume ou confia nos valores daí oriundos. Como tal, é feita uma validação de todos esses dados, impedindo a exploração desses mesmos por atacantes e consequente estado erróneo do sistema. A maioria dos ataques a aplicações web são causados pela ausência destas boas práticas de programação, especificamente a falta de validação dos inputs do utilizador.

Neste capítulo será explicado o conceito de vulnerabilidade e quais os tipos que os atacantes conseguem explorar devido a estas más práticas. Vão também ser identificados quais os diferentes métodos e técnicas utilizadas para lutar contra estas vulnerabilidades e quais delas serão o alvo da solução apresentada. Será apresentada com pormenor a técnica *fuzzing*, mostrando as diferentes metodologias de teste utilizadas. Por fim, serão apresentadas e comparadas entre si, as diferentes soluções *open-source* já existentes no mercado, tornando claro com este estudo, quais são os seus maiores pontos de falha, realçando assim a componente inovadora deste projeto.

2.1 Vulnerabilidades

Uma vulnerabilidade, no contexto da segurança informática, é um erro introduzido num programa durante a fase de desenvolvimento que deixa o sistema exposto a ataques. Existem vários tipos de vulnerabilidades e dentro da categoria das vulnerabilidades web, notou-se um grande crescimento nos últimos anos. Assim sendo, é comum as empresas classificarem estas vulnerabilidades de acordo com o risco que apresentam, e no decorrer desta secção vamos detalhar as 10 que a OWASP considera mais críticas em [38]. Vamos brevemente explicar no que consistem e como são originadas e, no caso de ser uma das vulnerabilidades detetáveis pela ferramenta a apresentar, vamos exemplificar como pode ser executado um ataque. É importante notar que o tipo de ferramenta apresentado não tem uma arquitetura nem atua diretamente na área onde algumas destas vulnerabilidades ocorrem e, portanto, apenas as relevantes para o projeto serão apresentadas na

Subseção 2.1.1, Subseção 2.1.7 serão alvo da ferramenta.

2.1.1 *Injection*

As vulnerabilidades de injeção dividem-se em subcategorias dependendo do tipo de sistema de armazenamento empregue na aplicação web. Este sistema, normalmente uma base de dados, pode ser relacional, mantendo uma estrutura definida, onde os dados são armazenados em tabelas que, podem ou não ser dependentes entre si, ou pode ser não relacional, com uma estrutura orientada a documentos que permite que várias categorias de dados sejam guardadas na mesma estrutura de dados. As bases de dados não relacionais, por exemplo MongoDB [30] ou a NoSQL da Oracle, são utilizadas em grande escala e, quando mal geridas, tornam-se vulneráveis a injeção. Por outro lado, as bases de dados relacionais podem originar vulnerabilidades SQL Injection, no entanto a sintaxe utilizada para levar a cabo o ataque pode variar um pouco dependendo da solução utilizada. Existem vários tipos de soluções de bases de dados relacionais, sendo que as mais utilizadas [8] são a base de dados da Oracle [34], a MS SQL [24] da Microsoft e a MYSQL [33] agora também da Oracle.

Esta é uma das vulnerabilidades que as ferramentas de *fuzzing* de aplicações web são capazes de detetar e como tal será alvo de estudo. Contudo, como no contexto da instituição de acolhimento é utilizada uma base de dados relacional, vamos apenas considerar as vulnerabilidades oriundas desta, nomeadamente SQL Injection. Para efeitos práticos, vamos apenas retratar ataques a bases de dados relacionais do tipo MS SQL, pois é a solução implementada na empresa, sendo o ambiente de teste disponível.

SQL *Injection*

A comunicação entre a aplicação e a base de dados é feita numa linguagem interpretada chamada *Structured Query Language* (SQL), que pode ser usada para ler, alterar, apagar e adicionar dados. Para tal, a aplicação tem de construir *queries* que normalmente incluem dados de *input* do utilizador. Ora, se estas *queries* forem mal construídas podem originar uma vulnerabilidade de SQL Injection, podendo resultar em consequências catastróficas, desde o roubo e alteração de dados, à eliminação por completo da base de dados.

```
<%
   name = request.queryString["name"]
   objdb.query("select * from users where name=" & name)
%>
```

Listing 2.1: Exemplo de segmento de código vulnerável a SQL Injection

A listagem 2.1 exemplifica um segmento de código VBscript vulnerável a SQL Injec-

tion. O que o torna vulnerável é o facto da variável `name`, oriunda de um dos parâmetros da *querystring*, ser utilizada na construção da *query* para a base de dados, sem ser feita qualquer validação de segurança. Um atacante pode facilmente manipular o URL e modificar o valor do parâmetro `name` para, por exemplo, `' ; DROP TABLE users; --`, conseguindo com isto, apagar a tabela `users` e todos os dados nela contidos.

2.1.2 *Broken Authentication*

As vulnerabilidades de autenticação e de gestão de sessão são críticas em aplicações web e permitem que um atacante evite os mecanismos de autenticação para entrar na aplicação. Envolvendo maioritariamente falhas na proteção de credenciais e na proteção de *tokens* de sessão durante o seu ciclo de vida, podem ser originadas de diferentes formas, como por exemplo, o armazenamento não protegido (não cifrado) de credenciais, credenciais previsíveis e identificadores de sessão expostos na *querystring* do URL. Estas vulnerabilidades podem originar graves violações de privacidade, mas não serão alvo de estudo pois não são detetadas pelo tipo de ferramenta estudado.

2.1.3 *Sensitive Data Exposure*

As vulnerabilidades de exposição de dados sensíveis, tal como o nome indica, são originadas quando o programa ou aplicação expõe dados sensíveis que deviam estar contrariamente protegidos. Estes dados, e a sua proteção, caem normalmente sobre alguma legislação de privacidade, como por exemplo a *General Data Protection Regulation* [7]) e como tal, a presença destas vulnerabilidades pode trazer graves consequências judiciais. Estas vulnerabilidades podem ser causadas de várias formas, desde a utilização de protocolos de transporte inseguros para o transporte dos dados sensíveis, ao impróprio armazenamento dos mesmos e uso de palavras-passe fracas. No entanto, as ferramentas de *fuzzing black-box* não têm capacidade de deteção destas e, portanto, não serão alvo de estudo.

2.1.4 XML *External Entities*

Este tipo de vulnerabilidade pode ser encontrado em aplicações web que comunicam em formato XML entre o servidor e o cliente. Ocorrem, quando o XML de *input* contém uma referência para uma entidade externa maliciosa e é processado por um *parser* de XML mal configurado. Se uma aplicação utiliza um destes *parsers*, um atacante pode facilmente manipular o XML de *input* e conseguir acesso a ficheiros do sistema de ficheiros de servidor e interagir com este e com qualquer entidade externa a que este tenha acesso.

As ferramentas de *fuzzing* são capazes de detetar estas vulnerabilidades quado configuradas para lidar com aplicações que aceitam XML diretamente. Contudo, no âmbito da instituição de acolhimento todas as aplicações comunicam em formato HTML, e por isso,

o estudo desta vulnerabilidade é refutado, pois não é relevante para os objetivos futuros da instituição de acolhimento.

2.1.5 *Broken Access Control*

O controlo de acesso é a forma das aplicações web regularem quais a funcionalidades que os utilizadores têm acesso. Desta forma as aplicações conseguem manter uma hierarquia de perfis ou grupos de forma a especificar os acessos que cada utilizadores. As vulnerabilidades de controlo de acesso permitem que um utilizador aceda a funcionalidades para as quais não tem permissão podendo originar variadas consequências. Esta pode parecer uma vulnerabilidade simples à partida, mas uma boa implementação de controlo de acesso é muito difícil de produzir, aumentando a complexidade com o número de perfis e acessos. No entanto, esta vulnerabilidade não será alvo de estudo pois, o controlo de acesso pertence à parte lógica da aplicação, variando de aplicação para aplicação e dependendo de quem a implementa, tornando impossível para as ferramentas de *fuzzing* a sua deteção.

2.1.6 *Security Misconfiguration*

Esta vulnerabilidade pode ocorrer um aplicações que tenham alguma propriedade de segurança, a qualquer nível da pilha protocolar ou caso utilizem serviços na *cloud*. Existem inúmeras maneiras de originar esta vulnerabilidade, sendo que as ferramentas de *fuzzing* conseguem detetar algumas delas, por exemplo se as *flags Set-Cookie* ou *Secure Flag* dos pacotes HTTP estão ativas ou se o atributo *autocomplete* das palavras-passe está ativo.

2.1.7 *Cross-site scripting*

As vulnerabilidades de XSS [37] continuam a ser uma das mais abundantes em aplicações web conseguindo manter um lugar constante no top 10 da OWASP [38] ao longo dos anos. Elas estão presentes nas mais diversas aplicações e, na maioria dos casos, o seu impacto é mínimo, não conseguindo conceder ao atacante controlo sobre o sistema [42]. No entanto, dependendo do contexto no qual estão inseridas, as suas consequências podem agravar. Uma vulnerabilidade de XSS numa aplicação bancária poderá resultar em consequências muito mais catastróficas do que num mero site informativo. Em cenários mais calamitosos, um atacante consegue obter dados privados (por exemplo *cookies* ou qualquer outra informação sobre a sessão), que lhe permitem redirecionar a vítima para conteúdo por ele controlado ou roubando-lhe a sessão. Pode também conseguir instalar programas, modificar a apresentação do conteúdo, entre outros.

Estes ataques são conseguidos através da injeção de *scripts* em aplicações web benignas de forma a que este seja executado no *browser* da vitima. São vários os erros que

permitem a injeção destes *scripts* mas, todos eles localizam-se em pontos de entrada da aplicação que recebem dados do utilizador ou de fontes não fiáveis.

Podemos dividir estas vulnerabilidades em dois tipos: *reflected* XSS (não persistente) e *stored* XSS (persistente) [36]. Estas diferem apenas no tipo de dados envolvidos, isto é, se são dados não persistentes ou persistentes. Na categoria de *reflected* XSS os dados não são persistentes, isto é, são fornecidos pelo input do utilizador. Imaginemos uma página onde existe uma caixa de pesquisa que reimprime o texto que recebe. Se este input não for validado, o *browser* executará o *script* que for submetido. Esta categoria é menos problemática pois só afeta o utilizador que introduz o *input*, mas não deve ser negligenciada, dado ser um ataque ainda bastante utilizado. Basta por exemplo, que uma vítima carregue num link aparentemente inofensivo, oriundo de um atacante construído com valores maliciosos nos parâmetros, para o seu *browser* executar os *scripts* maliciosos. Na Figura 2.1 esquematiza-se um exemplo prático desta situação.

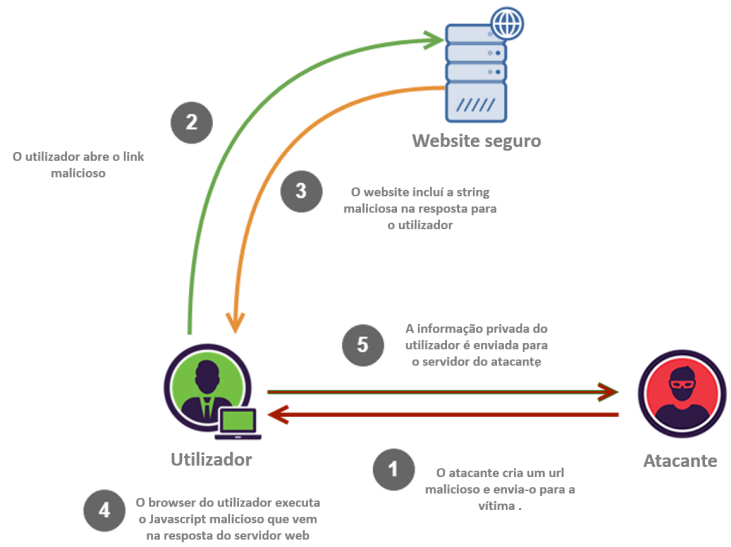


Figura 2.1: Exemplo de um ataque de *reflected* XSS [16]

Pelo contrário, na categoria de *stored* XSS, esquematizada na Figura 2.2, os dados são persistentes, ou seja, estão normalmente guardados numa base de dados e como tal, têm o mesmo valor para todos os utilizadores. Um bom exemplo desta categoria são os *blogs*. Considerando que as publicações dos utilizadores não são validadas antes de serem guardados na base de dados, um atacante pode facilmente submeter uma publicação contendo um *script* malicioso para a base de dados. Este *script* por sua vez, vai ser executado por todos os *browser* que abrirem a página da aplicação, onde as publicações são exibidas, vitimando inúmeros utilizadores.

A ferramenta a apresentar terá a capacidade de distinguir entre estas duas categorias, conseguindo adicionalmente, na vulnerabilidade de *reflected* XSS, indicar qual o ponto de

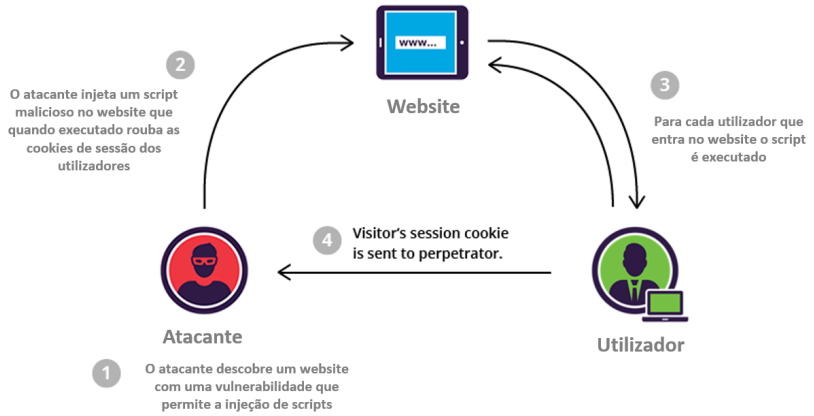


Figura 2.2: Exemplo de um ataque de *stored* XSS [17]

entrada ou parâmetro responsável pelo ataque. No Capítulo 3 será abordado com maior detalhe a capacidade de deteção de vulnerabilidades da ferramenta.

Existe ainda um terceiro tipo não tão estudado, o *Document Object Model* (DOM) XSS [35]. Neste tipo de XSS, o *script* malicioso é executado como resultado de uma modificação do ambiente DOM no *browser* da vítima [2]. Isto é, a resposta HTTP do servidor em si não contém o *script* malicioso, mas quando algo no *browser* da vítima despoleta a execução de algum *script* que faz uso do parâmetro malicioso, o *script* injetado no parâmetro em questão é também executado no *browser*, diferindo assim das duas outras categorias a cima referidas, onde o *script* malicioso já está contido na resposta do servidor.

2.1.8 *Insecure Deserialization*

Desserialização é o processo de converter dados que foram previamente serializados (isto é, convertidos num formato que pode ser guardado em disco ou numa *datastore*, normalmente em texto) no objeto original. As aplicações web utilizam muito estes processos na comunicação e normalmente já incluem ferramentas capazes de serializar e desserializar objetos de forma segura.

Esta vulnerabilidade ocorre quando os dados para desserializar são oriundos do input do utilizador. Nestas situações um atacante consegue tirar proveito da capacidade de algumas linguagens de programação serem personalizáveis no processo de desserialização. Este tipo de vulnerabilidade pode permitir que um atacante execute ataques de *Denial of Service* (DOS), que permitam contornar mecanismos de autenticação ou mesmo executar código remoto. Este tipo de vulnerabilidades não é detetável pela ferramenta a desenvolver.

2.1.9 *Using Components with Known Vulnerabilities*

Esta vulnerabilidade ocorre quando são utilizados componentes na aplicação com vulnerabilidades conhecidas. A probabilidade de ocorrência aumenta quando são utilizadas versões de software desatualizadas ou que caíram em desuso. As ferramentas de *fuzzing* são capazes de identificar algumas destas vulnerabilidades, dependendo da forma como estão implementadas. Por exemplo, no caso concreto deste projeto, que é desenvolvido especificamente para aplicações web, onde a comunicação entre cliente e servidor é feita no formato HTML, é possível extrair informação sobre o servidor e base de dados, se estes não tiverem configurados para esconder estas informações. Do *parse* da resposta HTML é possível também extrair informação sobre as versões das *frameworks* (se) utilizadas. No entanto, para conseguir detetar automaticamente estas vulnerabilidades é preciso uma ligação externa a um qualquer sistema de armazenamento de vulnerabilidades que indique, dada uma versão, se existe alguma vulnerabilidade conhecida associada, e como tal, não será implementado neste projeto.

2.1.10 *Insufficient Logging and Monitoring*

Esta vulnerabilidade difere um pouco das anteriores, pois não leva a uma direta intrusão, simplesmente aumenta o risco de não detetar uma intrusão a tempo. De pouco serve manter algum mecanismo de *logs* se estes não forem monitorizados, portanto apenas a presença dos dois pode reduzir o risco ao qual uma empresa está sujeita, quer financeira quer legalmente. Este também não é o tipo de problemas que as ferramentas de *fuzzing* detetam e, portanto, não será aprofundada.

2.2 Deteção de vulnerabilidades em aplicações web

A secção anterior sumariza e detalha as vulnerabilidades de aplicações web mais comuns nos dias de hoje, no entanto apenas algumas serão exploradas neste projeto, nomeadamente na vulnerabilidades apresentadas na Subseção 2.1.1 e Subseção 2.1.7. Nesta secção será explicado quais as duas técnicas mais utilizadas para encontrar e corrigir estas duas vulnerabilidades.

A primeira técnica, a análise estática, é uma técnica de *white-box*, que descobre vulnerabilidades ao analisar o código fonte das aplicações web. A segunda técnica, o *fuzzing*, que caí no reino do teste *black-box* (ou em alguns casos *grey-box*), encontra vulnerabilidades sem ter acesso ao código fonte e enquanto esta está a correr, podendo, entre outros, enviar pacotes de dados à aplicação ou tentar executar processos na aplicação alvo.

2.2.1 Análise estática

As ferramentas de análise estática encontram vulnerabilidades ao analisar o código, sem o executar [11]. Estas ferramentas automatizam grande parte do processo de auditoria de código, binários ou intermédios, sendo uma ferramenta extremamente útil ao longo do processo de desenvolvimento das aplicações.

Existem algumas limitações para as ferramentas que implementam esta técnica. Desde o tipo de vulnerabilidades que identificam, ao número de falsos positivos que podem levantar, o sucesso destas ferramentas está diretamente ligado a quem as desenvolve, pois são bastante complexas e difíceis de programar de modo a obter um alto nível de eficácia e precisão.

Para detetar vulnerabilidades, é feita uma procura no código por padrões e regras pré-definidas (dependendo do tipo de análise que implementam), impossibilitando assim a descoberta de novas vulnerabilidades e consequente elevado volume de falsos-positivos.

Não deixam de ser um forte aliado na prevenção e correção de erros nas aplicações e o seu uso durante o processo de desenvolvimento aumenta largamente a confiabilidade da segurança da aplicação. Estas ferramentas contribuem também para a prevenção de futuros erros pois, perante as vulnerabilidades descobertas, podem fazer sugestões de correções, garantindo que o problema é devidamente corrigido e educando também o programador.

Existem diferentes técnicas para obter as regras utilizadas na análise, sendo que estas enquadram-se em duas categorias: a análise lexical [43] e a análise semântica [44]. Na primeira o código é analisado em procura de funções ou bibliotecas que são consideradas inseguras, os chamados *sensitive sinks*. Para tal, o código é primeiro dividido em *tokens* e estes *tokens* são depois comparados com os *sensitive sinks* guardados numa base de dados. Este método pode gerar falsos positivos pois podem existir nomes de variáveis iguais a algum *sensitive sink*, sendo portanto a forma mais básica de análise estática, mas bastante utilizada.

Na segunda categoria, a análise semântica, a análise feita já é um pouco mais complexa. Ao ter em conta alguns aspetos semânticos, como a declaração de variáveis e os seus limites, variáveis de controlo de ciclos e fluxo de dados, é possível fazer um teste mais preciso e consequentemente com menor volume de falsos-positivos. Esta análise engloba três técnicas principais: *type checking*, no qual os limites das variáveis, dependendo do seu tipo, são analisados, *control flow analysis*, que ao executar todos os fios de execução possíveis no código deteta anomalias e, por fim, *data flow analysis* que verifica a forma como os dados, normalmente os inseridos pelo utilizador, fluem ao longo do código.

2.2.2 *Fuzzing*

O termo *fuzzing* foi introduzido em 1989 pelo professor Barton Miller na Universidade do Wisconsin [18] e desde então que diferentes significados têm sido atribuídos à palavra. Uma boa definição no contexto da segurança informática, foi proposta por Michael Sutton *et. al* em [3], que diz que *fuzzing* é o método para a descoberta de falhas em software através da introdução de *inputs* inesperados e procura de exceções nos respetivos *outputs*. É um processo tipicamente automatizado, ou semi-automatizado, que envolve fornecer e manipular repetidamente dados a alguma solução de software.

Existem vários tipos de *fuzzers* baseados no tipo de alvo que testam. Dentro destes tipos podemos encontrar *fuzzers* para programas locais, para programas de linha de comandos, para aplicações que têm como *input* e *output* algum tipo de ficheiro, *fuzzers* remotos, *fuzzers* de protocolos de rede, *fuzzers* de *browser* e *fuzzers* de aplicações web, entre outros. As metodologias de cada uns destes tipos podem variar, e como tal não existe uma metodologia padrão que possa ser aplicada a todos. É da responsabilidade do utilizador determinar qual a que mais se adequa ao alvo. No entanto, é possível discriminar as mesmas fases base em todos os tipos. Estas fases começam sempre pela identificação do alvo, isto é, qual o programa que será testado. Neste projeto todos os testes feitos têm como alvo os produtos desenvolvidos pela instituição de acolhimento e, portanto, esta fase torna-se pouco relevante. De seguida é realizada a identificação de todos os pontos onde são aceites dados dos utilizadores, pois é aqui que as vulnerabilidades são exploradas. Depois de mapeados todos estes pontos, a ferramenta tem de gerar os dados com os quais vai injetar a aplicação. É nesta fase que se encontra a componente inovadora do projeto. Depois de executados os testes é feita a monitorização do *output* e reconhecimento de exceções, por fim é determinada a *exploitability* de cada uma destas exceções, e este conhecimento final é transmitido ao cliente na forma de relatório.

O termo *fuzzing* no contexto informático tem evoluído muito com o tempo, no entanto a informação disponível é limitada e não existe um consenso global sobre os seus constituintes. O primeiro livro sobre *fuzzing* [3] afirma que a abordagem de uso de um *fuzzer* pode variar largamente, dependendo do contexto para o qual é utilizado e de inúmeros fatores externos. No entanto determina que, por mais diferentes que as metodologias sejam, podemos sempre encontrar as mesmas 6 fases no processo de *fuzzing*, esquematizadas na Figura 2.3. Até ao final desta secção cada uma destas fases vai ser examinada com maior pormenor, dando ao leitor uma maior noção sobre todo o processo de execução de um *fuzzer*.

1. **Identificação dos alvos -** Para determinar qual a metodologia mais adequada para o *fuzzer* é necessário primeiro identificar o alvo de análise. Não existe uma única metodologia e esta pode variar bastante de acordo com o alvo escolhido. Esta primeira fase de identificação do alvo é de grande importância pois molda total e intrinsecamente o restante processo.

Figura 2.3: Fases da metodologia de *fuzzing* [3]

No caso de auditorias de segurança onde o alvo é previamente selecionado, esta fase torna-se irrelevante. Contudo, em situações de investigações de segurança, com o objetivo de descobrir vulnerabilidades em aplicações gerais, esta fase é fundamental. Selecionar os alvos pode ser uma tarefa complicada e por isso, é comum os investigadores recorrerem ao historial dos vendedores de software, para verificarem se encontram vulnerabilidades noutras aplicações. Por vezes recorrem a sites como o SecurityFocus [13] ou o Secunia [12], que relatam os históricos de problemas de segurança de um vendedor. Desta forma o investigador ganhará uma inclinação para analisar aplicações web de vendedores com um maior historial nestes sites, pois existe uma maior probabilidade das suas aplicações serem criadas com más práticas de desenvolvimento.

2. **Identificação dos Inputs -** Após a identificação do alvo é necessário mapear a aplicação. Todas as vulnerabilidades das aplicações web são causadas pela aceitação de input do utilizador sem ser feita qualquer validação. A segunda fase da metodologia de *fuzzing* trata então de mapear todos estes pontos de entrada de inputs do utilizador.

   A maioria das ferramentas de *fuzzing* recorre ao uso de *crawlers* para mapear estes pontos de entrada, que se traduzem nos parâmetros que uma página web aceita (tanto pela *querystring* como pelo *form*). Estes parâmetros são os pontos de entrada na aplicação e onde as vulnerabilidades são exploradas. Esta é talvez a fase mais importante da metodologia, pois se não for executada com precisão e sem o mapeamento de todos os pontos de entrada, o utilizador arrisca-se a não detetar algumas vulnerabilidades, comprometendo assim todo o teste.

   O estudo feito às ferramentas apresentadas na Subseção 2.3.1 tornará claro que os *crawlers*, por elas disponibilizados, não têm a capacidade de detetar links gerados dinamicamente por Javascript. Como os produtos da instituição de acolhimento recorrem a estas técnicas para a geração da maioria dos links, estes *crawlers*

revelaram-se inúteis, não conseguindo mapear mais do que dois por cento dos pontos de entrada das aplicações. Consequentemente a maioria dos pontos de entrada não qualificou para os testes permanecendo num possível estado vulnerável.

Todavia, no contexto da instituição já existe uma ferramenta capaz de produzir em formato HTML uma listagem dos links de todas as páginas com os seus respetivos parâmetros e, como tal, foi decidido reutilizá-la. Esta ferramenta mapeia com sucesso a totalidade dos links da aplicação e respetivos parâmetros pois acede acesso ao diretório onde uma cópia da aplicação está instalada (percorrendo o código de todos os ficheiros à procura dos parâmetros). Assim, dadas as decisões tomadas e material já disponível na instituição, a fase inicial de descoberta da superfície não será alvo de estudo neste projeto.

3. **Geração dos dados *fuzzed* -** Depois de identificados os pontos de entrada, é necessário gerar os dados que serão utilizados para o *fuzzing*. Esta fase deverá ser o mais automatizada possível e dependendo do contexto de teste, estes dados podem ser gerados de diversas formas e como tal, conseguimos dividir os *fuzzers* em duas categorias de geração de dados: *mutation based* e *generation based*.

   Na primeira categoria, *mutation based*, os dados são gerados a partir de mutações sobre outros dados pré-existentes e na segunda, *generation based*, são gerados sempre de forma diferente e configurável a partir do modelo feito à aplicação alvo. Dentro destas duas categorias conseguimos ainda discriminar cinco diferentes métodos:

   (a) ***Pregenerated test cases* -** Neste método é feito um estudo inicial da especificação do alvo em particular, para perceber o tipo de estrutura de dados utilizada e qual o intervalo de valores suportados. De seguida são gerados pacotes (ou ficheiros, dependendo do tipo de *fuzzer*) que testam ou violam os limites deste intervalo. Estes testes requerem bastante trabalho manual e como não existe uma componente aleatória, a análise termina quando todos os testes previamente gerados forem executados.

   (b) ***Random* -** Este é o tipo de método mais ineficiente. Os dados utilizados na execução dos testes são gerados de uma forma pseudoaleatória, o que leva a que a taxa de eficiência desta abordagem seja completamente imprevisível, sendo normalmente um método com uma baixa taxa de sucesso e eficácia.

   (c) ***Manual protocol mutation testing* -** Este método é ainda menos sofisticado que o anterior pois nenhuma parte da solução é automatizada. Todos os valores testados são inseridos manualmente pelo utilizador, o que pode trazer uma alta taxa de sucesso, mas que torna o processo inviável devido à baixa taxa de eficácia.

(d) ***Mutation or brute force testing*** **-** Este protocolo começa com uma amostra do protocolo ou formato de dados a testar e altera individualmente todos os bytes contidos no pacote de dados ou ficheiros. Esta abordagem é pouco eficiente e exaustiva, devido à falta de conhecimento à priori sobre os protocolos, não é adequada para o *fuzzing* das aplicações web.

(e) ***Automatic protocol generation testing*** **-** Esta é uma versão mais avançada do método anterior e requer um conhecimento anterior do protocolo e tipo de ficheiro a testar. Desta forma, em vez de variar todo o conteúdo da amostra, é criada uma gramática que define a especificação do protocolo e do ficheiro. Assim, o *fuzzer* identifica porções do pacote ou ficheiro que serão previamente *fuzzed* e porções que se mantiveram estáticas ao longo dos testes.

Este método é o mais adequado para o *fuzzing* de aplicações web, pois existe um conhecimento à priori do formato de protocolo utilizado, neste caso o *HyperText Transfer Protocol* (HTTP), e desta forma é possível reduzir em muito o número de testes a executar, especificando apenas testes que apresentem alguma probabilidade de ter sucesso. No Capítulo 3 será abordado com detalhe como com este tipo de método é possível utilizar uma API que represente um pedido HTTP de forma apenas a mutar os dados de *input*. Assim, em vez de alterar byte a byte o conteúdo do pacote e testando cada uma dessas mutações, apenas são alteradas as porções de código onde são inseridos os inputs do utilizador, que neste caso específico são os dados passados pela *querystring* ou os dados passados no corpo da mensagem com o método POST. Desta forma, reduzem-se o número de testes a executar, otimizando todo o processo.

4. **Execução dos dados *fuzzed*** **-** Esta fase funciona em sintonia com a anterior e é aqui que o teste em si é executado, teste este que pode envolver enviar pacotes de dados (construídos com os dados gerados na fase anterior) para o alvo, abrir um ficheiro ou lançar um processo no alvo. Toda esta fase deverá ser automatizada, caso contrário não está a ser eficazmente implemententada a técnica de *fuzzing* que tem como premissa a automatização do processo.

5. **Monitorização de exceções -** Esta é uma fase essencial no processo pois garante que os testes estão a ser corretamente executados. Ao controlar as exceções o programa é capaz de detetar se os servidores onde as aplicações alvo estão *hosted* continuam a funcionar corretamente e a responder normalmente aos pacotes enviados. Se, por exemplo for feito um teste no qual são enviados dezenas de milhares de pacotes para um servidor, pode acontecer este parar de funcionar corretamente. Se tal não for detetado, continuar-se-ão a enviar pacotes que não serão executados pelo servidor. Assim o teste não será realizado, assumindo-se que não foi encontrada nenhuma vulnerabilidade, contribuindo para um elevado número de falsos-positivos,

e consequentemente deixando a aplicação insegura.

Ao implementar mecanismos capazes de identificar estas exceções, o programa tem a capacidade de identificar que o servidor está inoperacional e alterar o seu comportamento, garantindo que os restantes pacotes apenas serão enviados quando este estiver de novo em funcionamento, ou comunicando o sucedido ao utilizador. Independentemente da abordagem tomada, o que importa é que o utilizador será informado do que foi corretamente, ou não, testado.

6. **Determinação da *exploitability* -** Nesta última fase, que ocorre depois de uma vulnerabilidade ter sido identificada, e dependendo do objetivo da auditoria, pode ser necessário verificar até que ponto é que este erro consegue ser explorado. Este é um processo manual, que só pode ser realizado por alguém qualificado e com um grande um conhecimento de segurança informática.

2.3 Ferramentas existentes

Tal como foi referido no Capítulo 1 já existem várias soluções *open-source* de testes *black-box* de aplicações web que utilizam a técnica de *fuzzing*, e ao longo desta secção vão ser apresentadas umas das mais conhecidas, sendo que a instituição de acolhimento já recorreu a uma delas para testar os seus produtos. São ferramentas bastante completas oferecendo normalmente várias funcionalidades sem ser o *fuzzer*, no entanto, apenas este vai ser analisado, tentando classificar a sua eficiência e sucesso.

Um *fuzzer* pode ser avaliado de acordo com a sua taxa de sucesso, que cresce com um grande volume de verdadeiros positivos e diminui com o volume de falsos negativos, e de acordo com a sua taxa de eficiência, medida pela quantidade de falsos-positivos. Estas taxas são normalmente obtidas através do uso de ferramentas chamadas *benchmarks*. Os *benchmarks*, têm como objetivo avaliar *fuzzers* e outros tipos de ferramentas, atribuindo classificações às mesmas. Para tal, fornecem um alvo propositadamente vulnerável para a ferramenta testar e extraindo de seguida métricas de acordo com a qualidade dos resultados da ferramenta. A ferramenta obtém melhores classificações quanto mais se aproximar dos resultados reais, quer de número de vulnerabilidades descobertas, quer de baixo volume de falsos-negativos.

Na secção seguinte serão apresentados os dois *benchmarks* que foram utilizados para categorizar as ferramentas alvo de estudo, o *benchmark* da OWASP [39] e o *benchmark Web Application Vulnerability Scanner Evaluation Project* [51]. De seguida serão apresentados mais duas seções sobre os *fuzzers* estudados e ainda uma sobre os resultados dos seus testes contra os dois *benchmarks*.

2.3.1 *Benchmarks*

Benchmarking é o ato de correr um programa com o objetivo de classificar o seu desempenho, utilizando normalmente um conjunto de testes específicos para o tipo de ferramenta a testar. Desta forma, conseguimos obter um método de comparação entre programas do mesmo tipo. Existem vários tipos de *benchmarks*, operando ao nível do hardware ou do software, no entanto no contexto do projeto vamos apenas considerar os que operam ao nível do programa.

Os *benchmarks* estudados neste projeto foram especificamente desenvolvidos para classificar ferramentas de deteção de vulnerabilidades, classificando a sua rapidez, cobertura e eficiência. O primeiro, o *benchmark* da OWASP, desenvolvido e mantido pela mesma, disponibiliza milhares de testes com vulnerabilidades intencionalmente inseridas, onde cada um é mapeado no número CWE da vulnerabilidade correspondente. Esta ferramenta pode ser utilizada com ferramentas de segurança de *Static Application Security Testing* (SAST), ferramentas de *Dinamic Application Security Testing* (DAST) e ferramentas de *Interactive Application Security Testing* (IAST), classificando-as de quatro modos: se identificam corretamente vulnerabilidades reais (verdadeiro positivo), se falham a identificar uma vulnerabilidade real (falso negativo), se identificam vulnerabilidades falsas (falsos-positivos) e se ignoram vulnerabilidades falsas (verdadeiro negativo).

A ferramenta WAVSEP é indicada apenas para testar ferramentas DAST e oferece também uma variedade de páginas vulneráveis para as ferramentas analisarem, avaliando as ferramentas nos mesmos quatro modos que o *benchmark* da OWASP utiliza.

2.3.2 *Zed Attack Proxy*

O *Zed Attack Proxy* (ZAP) [41] foi criado e é mantido ativamente pela OWASP [40]. É uma ferramenta de teste *black-box* para aplicações web que segundo [38] é uma das mais populares. É uma *cross-plataform*, de fácil e intuitiva utilização de forma a que os seus utilizadores, profissionais ou não, consigam tirar o maior proveito das suas funcionalidades.

Podendo funcionar como um *proxy* ou não, o ZAP pode ser utilizado de duas formas: através de uma intuitiva *User Interface* (UI) ou através de uma API REST. Esta API é utilizada através da execução de um ficheiro *.bat*, permitindo que este execute como um processo de fundo. Ambos os mecanismos disponibilizam uma variedade de funcionalidades para testar as aplicações e descobrir vulnerabilidades. É disponibilizado um *crawler*, utilizado para mapear as aplicações alvo. O mapa resultante é depois testado recorrendo aos dois *fuzzers* disponíveis, o *active scan* que executa uma variedade de ataques de acordo com as politicas configuradas (pode testar apenas para algumas vulnerabilidades especificadas), ou o *passive scan* que simplesmente analisa os pedidos e as respostas em vez de fazer pedidos adicionais (este tipo de *scan* é utilizado para vulnerabilidades

mais simples, por exemplo, cabeçalhos em falta ou *flags* desativas). O ZAP disponibiliza também um scanner de portos e produz relatórios em formato HTML ou XML com informação sobre as vulnerabilidades descobertas durante a análise.

2.3.3 *Arachni*

O *Arachni* [5] é uma plataforma para testes *black-box* de aplicações web escrita em Ruby, que funciona nas várias plataformas, Linux, MacOs e Windows. É *open-source* e disponibiliza uma API REST, sendo adequado tanto para profissionais de segurança como administradores de sistemas. Disponibilizando uma grande variedade de funcionalidades e testes com diferentes alvos, o *Arachni* suporta aplicações web bastantes complexas que recorrem frequentemente a tecnologias como o JavaScript, HTML5, manipulação do DOM e AJAX. O próprio afirma que consegue detetar vulnerabilidades de injeção, XSS, *File inclusion*, entre outras, sem paralelo na sua resiliência e precisão segundo [4]. Tal como o ZAP, incluí o motor de um *browser* que o permite detetar vulnerabilidades baseadas em DOM, relembrar o DOM XSS e mantém estado dos fluxos de execução das páginas.

Em termos de desempenho e eficiência o *Arachni* também promete, mesmo quando lida com milhares de pedidos HTTP, consegue minimizar qualquer atraso pois as suas ligações são assíncronas, permitindo a criação de várias instâncias dos *fuzzers* e execução paralela dos motores dos *browsers*. Como resultado da análise, disponibiliza relatórios em diferentes formatos bastantes detalhados que incluem *snapshots* das páginas afetadas.

2.3.4 Resultados

Em [22] são apresentados os resultados que ambas as ferramentas apresentadas anteriormente tiveram contra os dois *benchmarks* apresentados em 2.3.1. Neste estudo é feita uma avaliação ao primeiro *benchmark*, o da OWASP e é concluído que ambas as ferramentas têm um desempenho semelhante, existindo certas categorias de vulnerabilidades nas quais o *Arachni* tem mais sucesso e outras em que o ZAP é mais eficaz.

As ferramentas são também analisadas de acordo com o tempo que demoram a testar a aplicação que o *benchmark* disponibiliza. O ZAP demorou cerca de seis horas e meia a terminar enquanto que o *Arachni* demorou cerca de dez horas e meia.

De seguida são apresentadas medidas quantitativas, onde é apresentado o calculo da taxa de eficácia na identificação das seguintes categorias de vulnerabilidades: *Command Injection, LDAP Injection, XSS, SQL Injection* e *Path Traversal Attacks*.

Por fim os autores comparam os resultados do *benchmark* da OWASP com os do WAVSEP, que foram realizados por outro grupo de investigadores. Estes resultados são visíveis na Figura 2.4 e na Figura 2.5.

Em ambos os estudos foram utilizadas as mesmas medidas quantitativas, analisando as

	WAVSEP											
	SQLI			XSS			CMDI			Path Traversal		
	TPR	FPR	Score	TPR	FPR	Score	TPR	FPR	Score	TPR	FPR	Score
ARACHNI	100%	0%	100%	91%	0%	91%	100%	0%	100%	100%	13%	100%
ZAP	96%	0%	96%	100%	0%	100%	93%	0%	93%	100%	13%	100%

Figura 2.4: Resultados do teste do *benchmark* da OWASP [22]

	OWASP BENCHMARK											
	SQLI			XSS			CMDI			Path Traversal		
	TPR	FPR	Score	TPR	FPR	Score	TPR	FPR	Score	TPR	FPR	Score
ARACHNI	50%	2%	50%	64%	0%	64%	31%	0%	31%	0%	0%	0%
ZAP	58%	4%	58%	76%	0%	76%	33%	0%	33%	0%	0%	0%

Figura 2.5: Resultados do teste do *benchmark* da WAVSEP [22]

mesmas categorias de vulnerabilidades. É concluído que os *benchmarks* comportam-se de forma semelhante, especialmente nas categorias de *XSS* e *Path Traversal*, mas que existem grandes variações nas medidas de *SQL Injection*. Estas variações são por fim explicadas pois a versão utilizada do ZAP, não foi a mesma nos diferentes estudos, podendo ser esta uma das causas desta grande variação de resultados.

2.4 Conclusões

Neste capitulo foi dado a conhecer alguns dos tipos de risco a que uma aplicação web está exposta diariamente. Estes são normalmente causados por más práticas de programação que originam vulnerabilidades no código-fonte. Na secção 2.1 foi explicado em pormenor o conceito de vulnerabilidade e quais as mais comuns em aplicações web, detalhando as dez que a OWASP considera mais críticas.

Para combater a presença de vulnerabilidades das aplicações web é comum e recomendado recorrer a ferramentas de teste no processo de produção de software, evitando assim possíveis erros no código-fonte. Na secção 2.2 são apresentadas as duas técnicas principais para a elaboração destas ferramentas: a análise estática e a análise dinâmica, mais regularmente conhecida como *fuzzing*.

Ao detalhar cada uma destas técnicas é concluído que a primeira é uma técnica de *white-box*, na qual é necessário acesso livre ao código-fonte da aplicação, sendo esta uma das suas limitações. Enquanto que a segunda é uma técnica de *black-box*, na qual o mesmo não é necessário, sendo os testes realizados sobre o código da aplicação em execução.

O projeto em causa trata da implementação de um *fuzzer*, portanto na secção 2.2.2 esta técnica é abordada com maior detalhe. Desde as fases que a constituem, à forma como os vetores de teste são gerados esta secção reúne a pouca, mas consistente informação que

existe sobre a matéria.

Na última secção são apresentados dois dos *fuzzers open-source* mais conhecidos, o *Arachni* e o ZAP. É também apresentado um outro tipo de ferramenta, os *benchmarks*, que servem para avaliar as ferramentas de teste. Por fim são mostrados os resultados de um estudo que tem como premissa a avaliação do ZAP e do *Arachni* contra os dois *benchmarks* apresentados, o WAVSEP e o *benchmark* da OWASP.

Capítulo 3

Conceção do sistema

Uma das principais causas da insegurança das aplicações web é a falta de conhecimento que alguns programadores têm sobre a matéria, o que inevitavelmente resulta na existência de vulnerabilidades nas soluções. Podemos dividir os mecanismos de segurança das aplicações em dois, de acordo com o envolvimento do programador. O primeiro não precisa da sua intervenção e integra nele técnicas como *firewalls* de aplicações web. O segundo pelo contrário precisa da intervenção do programador e engloba técnicas como a análise estática e os testes de *black-box*. Estas ferramentas são capazes de descobrir vulnerabilidades, mas precisam da intervenção do programador para a correção das mesmas. Podem também contribuir indiretamente para a formação do programador, oferecendo soluções de correção para os erros encontrados.

Neste capítulo vai ser dado a conhecer em detalhe a arquitetura de uma ferramenta que implementa umas destas últimas técnicas, os testes de *black-box*. De forma a que o processo de desenvolvimento seja mais claro e organizado, serão primeiro definidos os casos de uso entre o ator do sistema e o sistema em si. De seguida será feito o levantamento e análise de requisitos funcionais e não funcionais, onde todas as características e comportamentos básicos do sistema serão clarificados e como estas características devem ser cumpridas.

De seguida será apresentada com pormenor, a arquitetura do sistema com todos os seus componentes e qual a sua relação com os requisitos previamente apresentados. Para cada componente será explicado qual o seu objetivo, quais os requisitos que cumpre e que ligações, de entrada ou saída, têm com o ambiente ou outros componentes. A forma como cada componente opera sequencialmente também será explicada, demonstrando os diferentes fios de execução através de intuitivos fluxogramas.

3.1 Conceitos relevantes

Para perceber como são feitos os testes *black-box* a uma aplicação web é necessário primeiro compreender como esta funciona. Uma aplicação web é uma solução de software

que corre num servidor e gera páginas HTML. Quando um utilizador quer aceder a uma aplicação tem de utilizar um cliente, normalmente um *browser*, para comunicar com o servidor. A comunicação entre o servidor e o cliente é feita utilizando o protocolo da camada aplicacional, o HTTP [23]. Este é responsável por transmitir os documentos de HTML gerados pelo servidor e vice-versa. Na Figura 3.1 esquematiza-se um exemplo da comunicação entre um servidor HTTP e um cliente, onde o cliente neste caso será um qualquer *browser*.

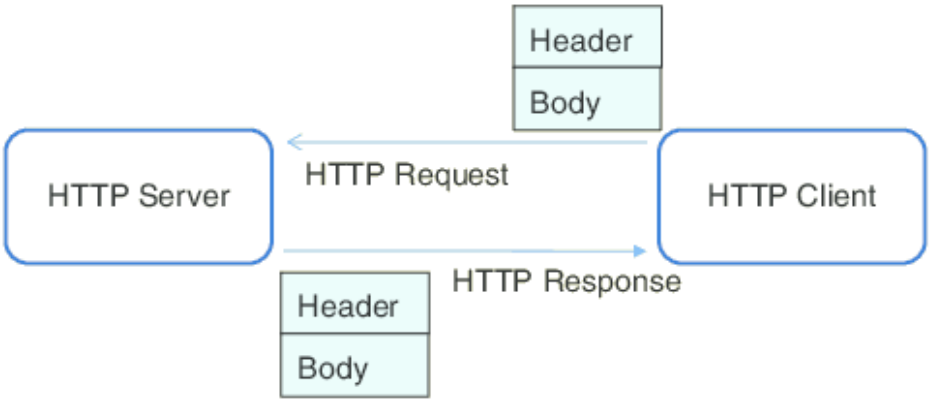


Figura 3.1: Comunicação HTTP [46]

Através da manipulação destes pacotes HTTP um atacante consegue explorar vulnerabilidades no código do servidor web. A estrutura destes pacotes, como podemos ver na Figura 3.2, é composta maioritariamente pelos cabeçalhos e pelo corpo. Nos cabeçalhos está o URL da página e a sua *querystring*, onde se podem encontrar parâmetros GET. E no corpo encontramos o documento HTML e os parâmetros POST. Estes parâmetros são pontos de entrada no servidor, e se este não estiver a fazer as validações suficientes, poderão ser pontos de vulnerabilidade. Se um atacante enviar um pacote e atribuir um valor de ataque a um destes parâmetros, pode conseguir explorar com sucesso uma vulnerabilidade (se esta existir).

Os testes de *black-box* são feitos à volta deste conceito: descobrir quais os parâmetros de uma página e enviar pedidos à mesma, variando o valor destes parâmetros, com valores específicos de ataque. Todas as menções feitas à palavra teste ”teste”ao longo deste documento referem-se a este conceito: ao envio de um pedido HTTP a uma qualquer página web com uma certa coleção de valores para cada parâmetro.

3.2 Definição dos casos de uso

Os casos de uso, no âmbito da engenharia de software, são uma lista de ações ou passos que definem as interações entre um ator e um sistema para atingir um determinado

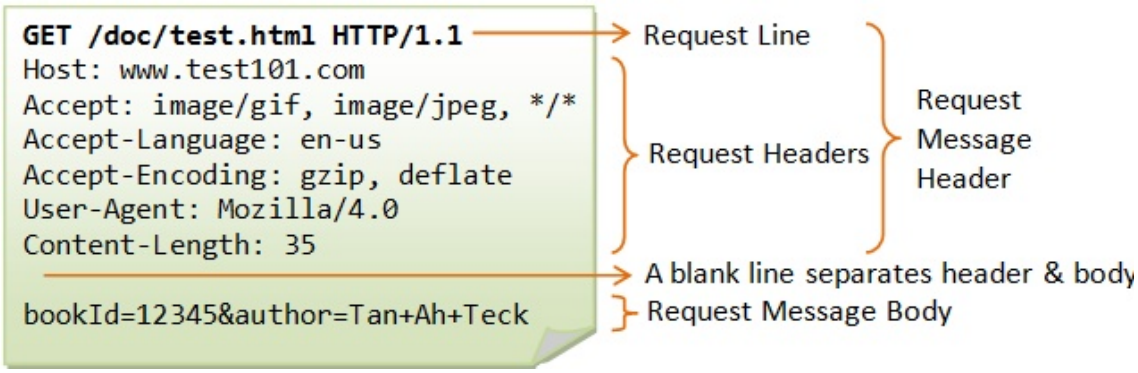


Figura 3.2: Pacote HTTP [20]

objetivo. Este ator pode ser um humano (como é o caso) ou outro qualquer sistema externo. Uma forma para expressar estas interações pode ser através de diagramas, que de forma simples e sucinta apresentam estas relações. Pode conter diferentes atores e incluir mais do que um diagrama mostrando as relações entre os vários sistemas. Na Figura 3.3 esquematiza-se o diagrama de casos de uso que expressa as funcionalidades desejadas para este projeto.

3.3 Levantamento e análise de requisitos

Uma sólida definição dos requisitos é a chave para o sucesso de qualquer projeto de software. Esta definição permite o avanço pelo processo de desenvolvimento com ideias sólidas e bem definidas, contribuindo para a sua boa estruturação. Soluções de software de *fuzzers* não são exceção e como tal, estes requisitos devem ser bem definidos e posteriormente bem implementados e cumpridos. Existem dois tipos de requisitos [45]: os requisitos funcionais e os não funcionais e ao longo desta secção vamos explicitar quais os desejáveis para a ferramenta.

Um requisito funcional define um comportamento básico de um sistema, é uma característica que garante que o sistema funciona da forma pretendida. Se os requisitos funcionais não forem cumpridos o sistema não irá funcionar. Eles definem o que o sistema deve fazer, e como tal, a ferramenta tem os seguintes:

1. O sistema recebe inputs de uma aplicação externa desenvolvida na instituição de acolhimento chamada ED Segurança;

2. Estes inputs cobrem a fase dois de *fuzzing* ao alimentarem a aplicação com os pontos de entrada do alvo (entendem-se estes inputs como os links e os parâmetros das páginas da aplicação web alvo de teste);

3. Os clientes que acedem ao sistema são previamente autenticados, tendo acesso exclusivo aos seus dados;

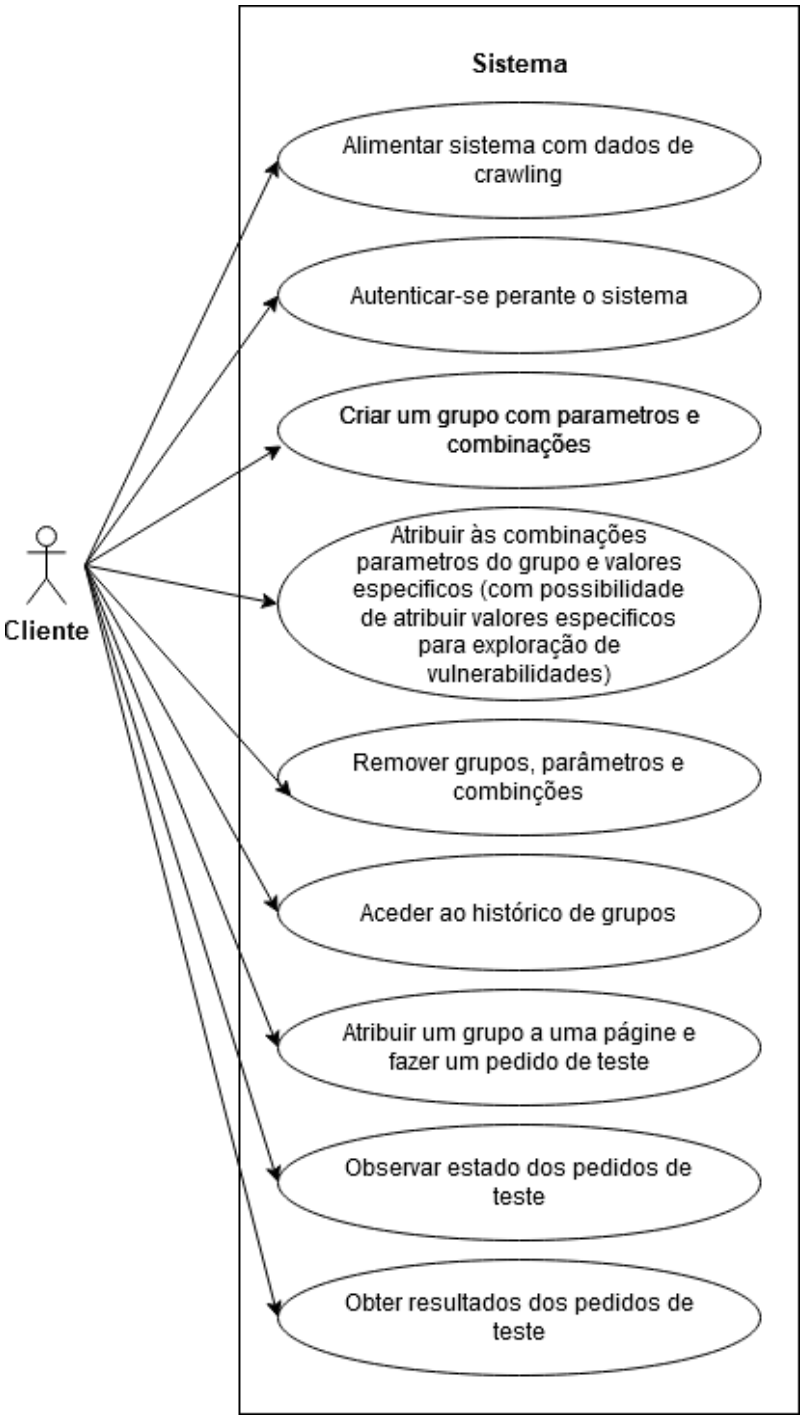


Figura 3.3: Diagrama de casos de uso

4. Os clientes podem criar grupos, que por sua vez podem ter parâmetros com diferentes valores e diversas combinações;

5. Os clientes podem criar combinações nos grupos e dentro destas, selecionar os diferentes parâmetros a variar e os respetivos valores. Dentro dos valores dos parâmetros das combinações, é também possível atribuir valores de teste específicos

para as vulnerabilidades de XSS e SQL *Injection* aos parâmetros;

6. Os clientes têm acesso ao histórico de grupos já criado e podem a qualquer momento escolher um para criar um novo pedido de teste a qualquer página;

7. Os clientes podem a qualquer momento remover estes grupos, combinações e parâmetros, desde que o grupo não esteja em uso por algum pedido em desenvolvimento.

8. Os pedidos dos clientes do sistema são executados por um motor de teste que corre individualmente e sem paralelismo num único servidor;

9. O motor de teste recebe os dados dos clientes do sistema e executa os testes às páginas através da técnica *fuzzing*;

10. O motor de teste faz uma cuidada gestão e monitorização das ligações aos servidores alvo, verificando sempre se estas continuam a funcionar corretamente e que os pedidos de teste estão a ser executados na integra;

11. O motor de teste analisa os resultados dos testes de *fuzzing* e cria um detalhado relatório dos resultados em HTML;

12. Os relatórios são disponibilizados para os clientes do sistema, sendo possível fazer o seu *download*, para que o cliente possa posteriormente analisar e efetuar as alterações necessárias à sua aplicação.

Os requisitos não funcionais [31] definem a forma como os requisitos funcionais devem ser cumpridos e implementados. Estes não afetam as funcionalidades básicas do sistema e, portanto, se não forem cumpridos o sistema continuará a funcionar. Ao definirem comportamentos de sistema, funcionalidades, etc., os requisitos não funcionais contribuem em muito para a experiência do utilizador. Para esta ferramenta os seguintes são desejáveis:

1. **Ambiente de desenvolvimento -** O ambiente de desenvolvimento, em termos informáticos, é o sistema computacional no qual um componente de software ou um programa é desenvolvido e executado. Dado que este projeto foi desenvolvido numa empresa, foi decidido que a ferramenta fosse desenvolvida utilizando o C # [25], que é uma linguagem desenvolvida pela Microsoft orientada a objetos e de alto nível.

2. **Desempenho -** O desempenho mede a quantidade de trabalho útil conseguido pelo sistema computacional. Este pode ser quantificado em termos de eficiência, velocidade de execução, tempo de resposta, entre outras. Um dos grandes motivos para a instituição de acolhimento investir neste projeto, e tal como foi mostrado

no capítulo 2, o fato de as ferramentas *open-source* existentes terem um mau desempenho. Este mau desempenho é causado por vários fatores, sendo o mais intrusivo a extensão do número de testes realizados a apenas uma página. Um dos requisitos para esta ferramenta é então um melhor tempo de execução, que pode ser conseguido pela afinação do volume de testes a cada página e por um rápido processamento dos resultados.

3. **Eficiência -** Um sistema eficiente faz um consumo equilibrado e não exaustivo dos recursos do sistema. Este é um requisito crítico para sistemas de alto processamento, como é a ferramenta apresentada. Dependendo do contexto, a ferramenta poderá ter de processar grandes volumes de dados e como tal, fazer uma utilização eficiente dos recursos é crucial, pois se os recursos forem levados à exaustão, graves consequências podem ser originadas comprometendo toda a credibilidade dos resultados da ferramenta (por exemplo: esgotar a capacidade de resposta a pedidos HTTP de um servidor alvo quebra a credibilidade dos resultados pois pode deixar bastantes testes por executar, ou mesmo a ferramenta deixar de ter capacidade de enviar pedidos HTTP).

4. **Flexibilidade -** Um sistema flexível deverá conseguir lidar com futuras mudanças nos requisitos. Este é um requisito desejável e que deverá estar presente nos seguintes aspetos: na entrada de inputs, o sistema deverá ser capaz de ser alimentado por qualquer programa ou fonte de dados. Na saída de inputs, o sistema deverá conseguir reproduzir relatórios em qualquer formato desejado. E em execuções paralelas, o sistema deverá ser capaz de ter entre uma a n réplicas a executar em paralelo, garantindo um ordenado e cordial acesso aos dados.

5. **Integridade dos dados -** A integridade dos dados refere-se a como estes são geridos, isto é, garante que os dados que o sistema manipula se mantém precisos ao longo do tempo. Este é um requisito crítico para qualquer sistema que utilize um sistema de armazenamento de dados, como é o caso, pois um dos seus componentes é uma base de dados, como veremos com maior detalhe nas secções seguintes. É, portanto, crucial a implementação de mecanismos que garantam uma manipulação, armazenamento e processamento correto destes dados.

6. **Segurança -** A segurança garante a proteção dos sistemas e das suas redes contra roubo ou danos de hardware, software, dados ou serviços. Este é um requisito não funcional que todos os produtos de software devem ter presentes. A segurança deverá existir em todas as camadas, desde a interface com o cliente, à camada de negócios e por fim à camada de acesso aos dados.

   Na camada da interface com o utilizador é necessário garantir acesso exclusivo aos seus próprios dados, e para tal deverá ser utilizado um mecanismo de autenticação.

Este mecanismo garante também que apenas os utilizadores do sistema desejado é que têm acesso ao mesmo. Toda a interface com o utilizador terá de ser propriamente validada, especialmente os pontos de entrada de inputs do utilizador, garantindo que estes estão a ser propriamente validados. É também necessário garantir que qualquer componente de software, que manipule os dados do cliente e que esteja num local acessível por alguma rede, seja fortemente testado e validado de forma a que qualquer ataque se torne inviável.

Na camada de acesso aos dados, a forma como estes são acedidos, também tem de ser cuidadosamente estudada e protegida de forma a que uma situação de roubo e manipulação externa seja impossível. Por fim, as ligações entre os componentes também têm de ser confiáveis de forma a que todos os componentes interajam de forma segura e ordenada.

7. **Implantação de software -** Tal como no ambiente de desenvolvimento, existem outros requisitos impostos pela instituição de acolhimento, a respeito do ambiente de software no qual o produto irá ser *deployed*. Um destes é o sistema de armazenamento de dados utilizado, uma base de dados relacional gerida pelo software da Microsoft o Microsoft SQL Server [24], que será o utilizado pela ferramenta.

   No contexto da instituição a ferramenta irá ser fundida com outra ferramenta de segurança, desenvolvida internamente, chamada ED Segurança e, como tal, iremos utilizar o sistema de autenticação já desenvolvido nesta, não sendo necessário implementar um próprio para a ferramenta.

8. **Usabilidade -** A usabilidade é descrita como a capacidade que um sistema computacional tem, para garantir condições aos seus utilizadores para realizarem as suas tarefas de forma segura e eficiente, enquanto desfrutando da experiência. Como tal, é um objetivo que a camada da interação do utilizador consiga proporcionar uma boa experiência ao utilizador, sendo que esta será mais simples e intuitiva do que as ferramentas analisadas no capítulo 2, proporcionando ao cliente um maior controlo e configuração sobre os testes a realizar. De forma a simplificar a experiência do cliente deverá ser mantido um histórico dos grupos de teste criados por este, reduzindo sempre que possível a repetição de operações ou configurações.

3.4 Arquitetura

Dados os requisitos apresentados anteriormente, nesta secção será apresentada a solução arquitetural que visa cumpri-los. Esta genérica e simples arquitetura servirá de base para a implementação de uma ferramenta de *fuzzing* de aplicações web.

A Figura 3.4 ilustra a solução proposta e todos os seus componentes. No centro da Figura encontram-se (dentro do maior quadrado) os componentes principais da arquitetura,

aos quais pertence o motor de testes, e todos os seus subcomponentes, a interface do utilizador e o sistema de armazenamento de dados. E fora desse quadrado os componentes de ligação que podem ser externos ao sistema principal, o sistema de autenticação, interface de ligação e de saída, os dados de entrada da interface de ligação e os dados gerados pela interface de saída, os relatórios.

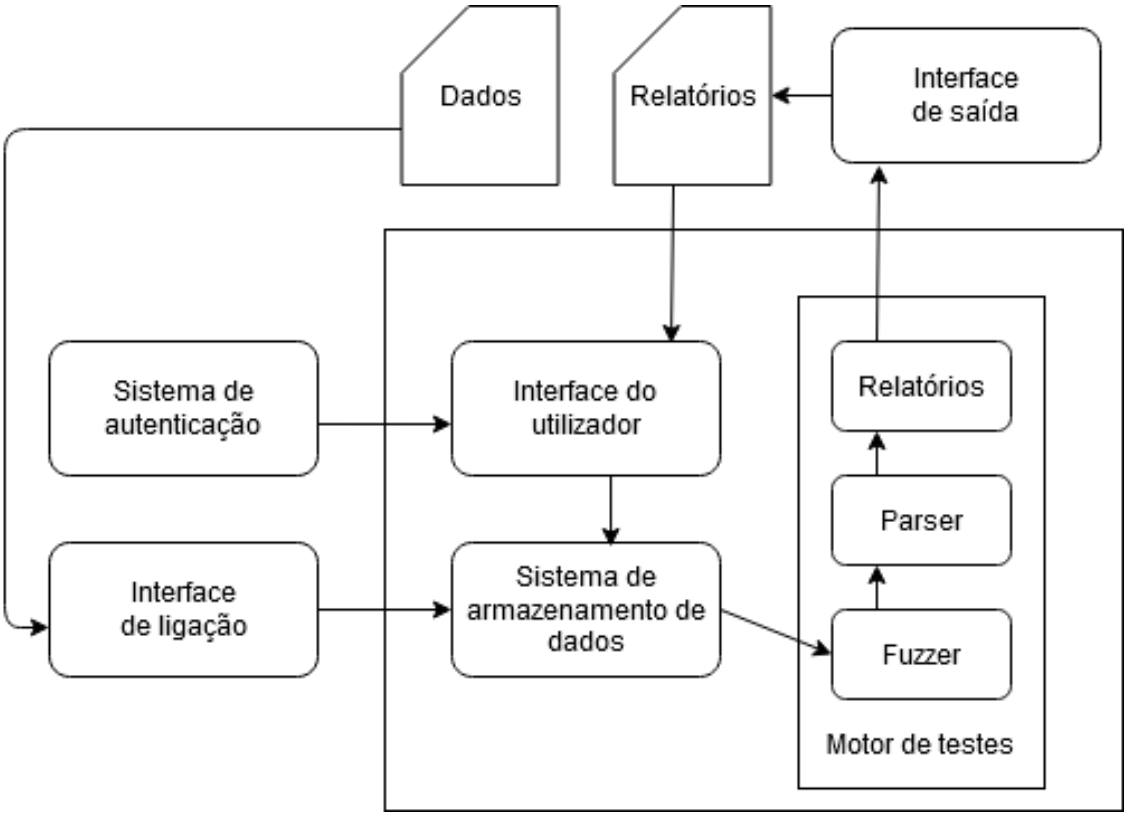


Figura 3.4: Arquitetura proposta

Ao longo desta secção cada um destes componentes irá ser apresentado, explicitando para cada, qual o seu objetivo, quais as suas ligações de entrada e saída, que requisitos cumpre e como.

3.4.1 Sistema de autenticação

A inclusão de um componente responsável pela autenticação garante parte da segurança da aplicação. Ao introduzir um mecanismo de autenticação assegura-se que apenas os utilizadores desejados acedem à aplicação e apenas têm acesso aos seus próprios dados. O principal objetivo deste mecanismo, para além da segurança, é funcionar como porta de entrada para a interface do utilizador, garantindo que este tem sempre de passar por este componente antes de aceder à interface do utilizador.

Na arquitetura proposta este encontra-se isolado no seu próprio modulo para cumprir mais dois requisitos: primeiro, a implantação de software, isto é o seu *deployment* e

segundo a flexibilidade.

Como a ferramenta irá ser implementada num contexto específico, esta irá funcionar integralmente com outra ferramenta já existente, sendo que foi decidido que o mecanismo de autenticação desta outra ferramenta deveria ser reciclado, logo que não era desejada a implementação deste componente, pois já existe outra solução disponível. Adicionalmente, ao isolarmos este componente é garantido também que outras soluções para a implementação deste componente são possíveis, tornando a arquitetura geral e flexível.

Componente	Entrada	Saída
Sistema de autenticação	Dados do utilizador	Interface do utilizador

Tabela 3.1: Ligações ao sistema de autenticação

Na Tabela 3.1 encontramos as ligações feitas ao sistema de autenticação. Este receberá como entrada, dados do utilizador, mais concretamente as suas credenciais, e após a sua validação com sucesso, reencaminhará o utilizador para a interface do utilizador, enviando informação sobre o utilizador para esta.

3.4.2 Interface de ligação

A interface de ligação tem como objetivo servir de porta para aplicações, programas, *scripts* ou serviços externos alimentarem a aplicação com os dados de *crawling*. Esta poderá ser uma biblioteca com funções documentadas e simples ou uma intuitiva interface, e tem como objetivo cumprir o requisito de flexibilidade ao tornar esta aplicação compatível com inúmeras outras.

Terá de ser acordado um formato de dados, que os produtos externos terão de cumprir e terão de processar, de forma a conseguir alimentar a aplicação. Com este componente é garantida a segurança da aplicação pois limita-se a forma como os inputs entram nesta, sendo que este componente funciona como um filtro que valida e garante que apenas dados seguros e validados entram no sistema, mais especificamente, no sistema de armazenamento de dados.

Componente	Entrada	Saída
Interface de ligação	Dados de *crawling*	Sistema de armazenamento de dados

Tabela 3.2: Ligações à interface de ligação de dados

Na Tabela 3.2 encontramos as ligações de entrada e saída estabelecidas por este componente. É definido que a interface de ligação recebe dados de *crawling* e envia dados para o sistema de armazenamento de dados. A fonte destes dados de *crawling* não é discriminada para, uma vez mais, cumprir o requisito de flexibilidade, permitindo que qualquer programa ou solução de software, ou mesmo manualmente, sejam esta fonte. Por fim este componente guarda estes dados no sistema de armazenamento de dados.

3.4.3 Interface do utilizador

A interface do utilizador poderia ser implementada de diversas formas, no entanto, para respeitar o primeiro requisito não funcional, o ambiente de desenvolvimento, será implementada como uma aplicação web, desenvolvida utilizando a linguagem da Microsoft o C #. Adicionalmente, para cumprir o requisito sete, a implantação de software, estará também preparada para trabalhar com uma base de dados relacional da Microsoft.

Um outro requisito não funcional muito importante é a usabilidade, e este pode ser garantido de diversas formas, no entanto temos de garantir que este componente, responsável pela interação com o utilizador, é simples e agradável, proporcionando uma experiência detalhada e fluída ao utilizador. Outro ponto que também deverá garantir é a possibilidade de armazenamento de históricos de operações e configurações, impedindo assim a possibilidade do utilizador se repetir.

Dado este ser um componente que interage diretamente com o sistema de armazenamento de dados, alimentando-o com novos dados oriundos do utilizador, é imperativo, para cumprir o requisito de segurança e da integridade de dados, que todos os pontos nos quais os dados do utilizador entram sejam validados de forma a não comprometer os dados e toda a sua estrutura de armazenamento.

Componente	Entrada	Saída
Interface do utilizador	Dados do utilizador	Sistema de armazenamento de dados

Tabela 3.3: Ligações à interface do utilizador

Na Tabela 3.3 encontramos as ligações feitas à interface do utilizador, intuitivamente, esta recebe dados do utilizador, que depois de validados armazena no sistema de armazenamento de dados.

3.4.4 Motor de testes

O motor de testes é responsável por todo o processo de teste desde que o pedido do cliente é feito, até obter o relatório com os resultados. Este poderá futuramente ser modular, existindo várias instâncias a correr em paralelo, cada uma processando os seus pedidos. Com isto, a ferramenta conseguirá ter sucesso mesmo em ambientes onde o fluxo de dados é consideravelmente maior, conseguindo cumprir com eficiência e com bom desempenho os desafios apresentados.

Componente	Entrada	Saída
Motor de testes	Sistema de armazenamento de dados	Sistema de armazenamento de dados

Tabela 3.4: Ligações ao motor de testes

Na Tabela 3.4 são expressas as ligações de entrada e saída deste componente. Os dados que recebe de entrada são os originados na interface do utilizador, que são posteriormente guardados no sistema de armazenamento, local onde o motor de testes os recebe. Todos os dados que este componente gera são também guardados neste mesmo sistema de armazenamento de dados, para serem posteriormente acedidos pela interface do utilizador.

De forma a simplificar a complexidade deste componente, este foi dividido em três subcomponentes: o *fuzzer*, que é responsável por obter os pedidos do cliente, preparar os vetores de ataque e executar os testes à aplicação alvo; o *parser* que funciona em paralelo com o *fuzzer*, analisando o corpo das respostas que o *fuzzer* recebe e por fim o módulo de geração dos relatórios, que perante os erros encontrados pelo *parser* compila toda a informação num relatório em determinado formato. Até ao final desta secção, cada um destes componentes vai ser detalhado, explicando o que cada um é responsável por fazer e como.

3.4.4.1 *Fuzzer*

Este componente inicia o processo de teste e é o componente mais crítico em termos de desempenho e eficiência, pois é o que envolve um maior processamento. Sumariamente é responsável por obter os dados dos pedidos dos clientes, armazenados no sistema de armazenamento de dados, criar os vetores de ataque, enviar o pedido HTTP para a página alvo de teste, receber a sua resposta, passar a resposta para o componente adjacente (responsável pelo processamento da resposta, o *parser*) e repetir tudo até não ter mais testes para executar.

Tal como todos os componentes que tem acesso direto ao sistema de armazenamento de dados, estas interações têm de ser seguras, de forma a manter a integridade dos dados. Depois de obter os dados deste sistema o componente tem de percorrer estes, de forma a colocá-los num certo formato iterativo, para conseguir executar todos os testes da forma mais eficaz possível. Para cada combinação de teste, o *fuzzer* é responsável por enviar um pedido HTTP e esperar pela sua resposta, sendo neste momento essencial uma cuidada monitorização do servidor alvo, isto é, garantir que este continua a operar normalmente e que as ligações continuam funcionais, garantindo a eficiência do sistema. Quando o *fuzzer* recebe a resposta, apenas tem de entregá-la ao seguinte componente, o *parser* e continuar a sua execução.

3.4.4.2 *Parser*

Este componente funciona em paralelo com o *fuzzer* e é alimentado por este. Após a recepção de uma resposta HTTP, o *parser* tem como função percorrer o corpo deste e, ao comparar com padrões constantes de ataque, determinar se a resposta contém ou não uma vulnerabilidade explorada.

As vulnerabilidades alvo desta ferramenta são facilmente detetáveis por padrões absolutos, resultantes dos valores fornecidos aos parâmetros do pedido HTTP. O *parser* tem então de percorrer o corpo da resposta HTTP e tentar detetar alguns destes padrões. De seguida, tem de guardar informação relativa a estas ocorrências, num determinado formato, para que no final, o componente de geração de relatórios, execute o relatório a partir destes dados.

3.4.4.3 Geração de relatórios

O último componente do motor de testes funciona individualmente e é chamado quando o *fuzzer* e o *parser* acabam a sua execução para um dado pedido. Nesse momento, este componente é responsável por percorrer os dados gerados pelo *parser*, que contém os resultados dos testes e reencaminhá-los para um outro componente, a interface de saída. É neste componente que o relatório em si é criado e posteriormente disponibilizado na interface do utilizador.

3.4.5 Interface de saída e Relatórios

Este componente é incluído nesta arquitetura para cumprir um importante requisito não funcional, a flexibilidade. Ao disponibilizar uma simples e intuitiva interface, com funções que recebem dados num certo formato e produzem um relatório em qualquer formato, assegurasse, que de forma fácil e acessível, qualquer programador possa implementar uma nova interface que gerará relatórios num determinado formato. Este componente trabalha diretamente com o último subcomponente do motor de testes, o gerador de relatórios, que chama as suas funções e alimenta-as com os dados que recebe do *parser* e que por fim gera o relatório. Este relatório é finalmente guardado no servidor, onde o motor de testes é *hosted*, podendo ser posteriormente disponibilizado para download na interface com o cliente.

Componente	Entrada	Saída
Interface de saída	Motor de testes	Sistema de armazenamento de dados

Tabela 3.5: Ligações à interface de saída

As ligações de entrada feitas a este componente são feitas pelo motor de testes e as ligações de saída, os dados que gera, são direcionados para o sistema de armazenamento de dados, como indicado na Tabela 3.5.

3.4.6 Sistema de armazenamento de dados

O sistema de armazenamento de dados é um componente essencial e central nesta arquitetura, pois é responsável por toda a gestão e armazenamento dos dados da aplicação.

Como foi mostrado na secção anterior, o sétimo requisito não funcional descreve o ambiente para o qual a ferramenta irá ser implementada e *deployed* e, como tal, o sistema de armazenamento será uma base de dados relacional, mais especificamente a base de dados MS SQL [24].

De forma a cumprir outro requisito, que é considerado crítico para qualquer arquitetura que inclua um sistema de armazenamento de dados, a integridade de dados, temos de garantir que todos os acessos feitos a este componente são realizados de forma segura. Assim é essencial garantir que todos os dados provenientes de inputs de utilizador, antes de serem armazenados, têm de passar por um processo de validação de forma a garantir a segurança do sistema.

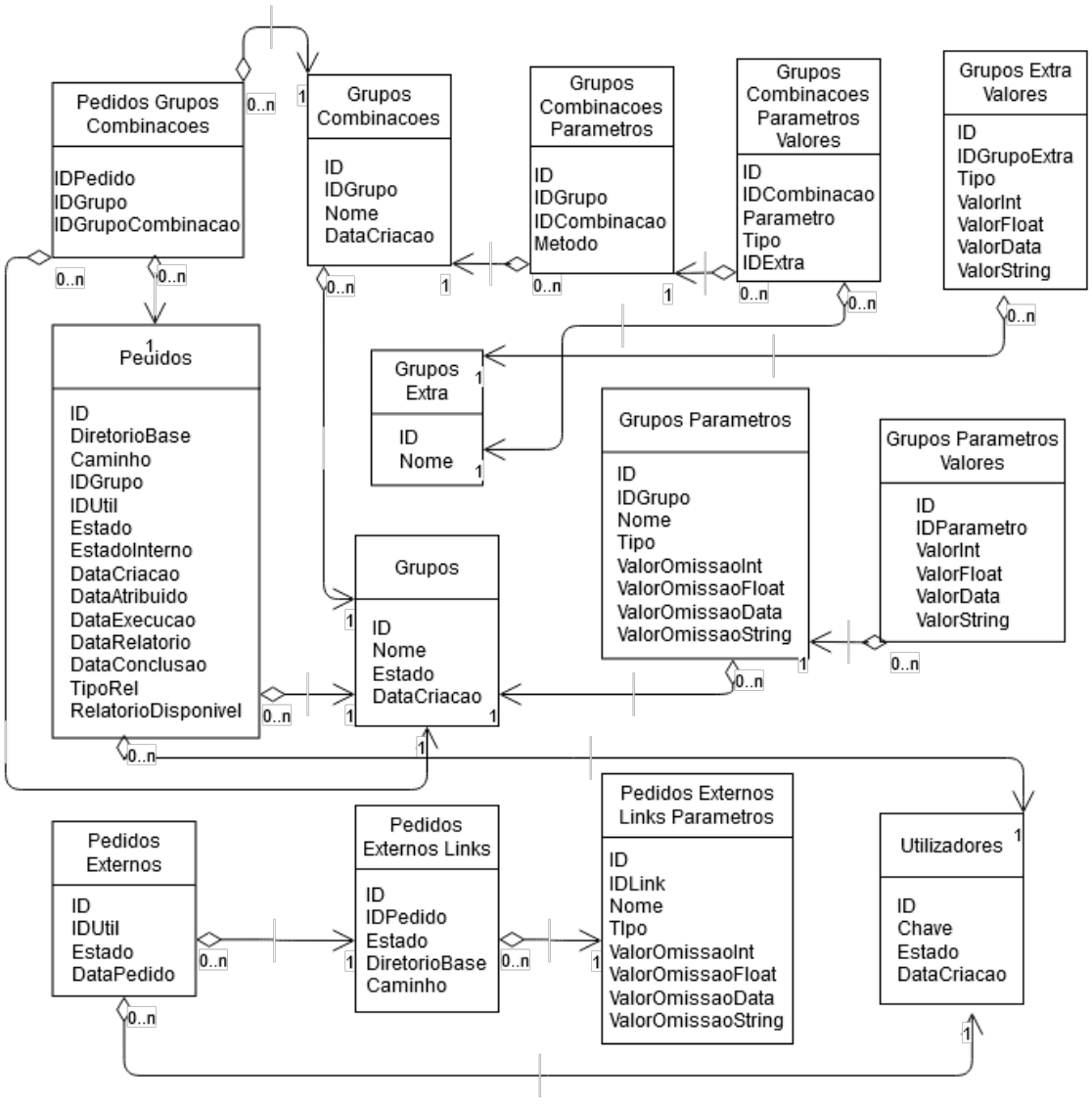


Figura 3.5: Diagrama de classes

Com base nos conceitos apresentados na secção 3.4.3 da interface do utilizador, determinou-se que o modelo de dados que a base de dados deverá possuir é o apresentado na Figura

3.5. De forma a garantir a segurança e autenticidade de dados, foi criado o conceito de utilizador, assegurando assim que cada utilizador apenas acede aos seus dados. Os restantes conceitos: grupos, combinações, parâmetros e valores, desdobram-se em várias tabelas, que expressam as suas propriedades e as relações entre si. Por fim, foram também incluídas tabelas, com o prefixo "Pedidos Externos", especificamente desenhadas para armazenar os dados oriundos da interface de ligação (dados de *crawling*),

Componente	Entrada	Saída
Sistema de armazenamento de dados	Sistema de autenticação	Interface do utilizador
Sistema de armazenamento de dados	Interface de ligação	Interface do utilizador
Sistema de armazenamento de dados	Interface do utilizador	Motor de testes
Sistema de armazenamento de dados	Motor de testes	Interface do utilizador
Sistema de armazenamento de dados	Interface de saída	Interface do utilizador

Tabela 3.6: Ligações ao sistema de armazenamento de dados

Na Tabela 3.6 encontram-se todas as ligações que o sistema de armazenamento de dados terá. Este é sem dúvida o componente com mais ligações, sendo que quase todos os outros componentes interagem com este. Desde a interface do utilizador, que armazena os dados que recebe do utilizador neste componente, ao motor de testes que também armazena os dados que gera neste, este componente é crítico e bastante central nesta arquitetura, sendo de extrema importância que seja implementado de forma a garantir que todos estes acessos sejam feitos de forma ordenada e segura.

3.5 Sequência de operações

Para terminar a conceção do sistema e partir para a etapa da implementação, todos os aspetos funcionais e não funcionais devem estar bem definidos e estruturados, e como tal, é necessário estabelecer uma sequência das ações da ferramenta. Mais concretamente, é necessário mostrar qual é a ordem de operação dentro de cada componente e como estes se ordenam entre si. Ao longo desta secção será explicado, através de intuitivos fluxogramas, quais os fios de execução que a ferramenta deverá ter. Começando com uma visão global da execução da ferramenta, desde a entrada de dados de *crawling* até à saída dos relatórios, seguindo para uma abordagem mais pormenorizada dos dois componentes mais relevantes, a interface do utilizador e o motor de testes.

3.5.1 Global

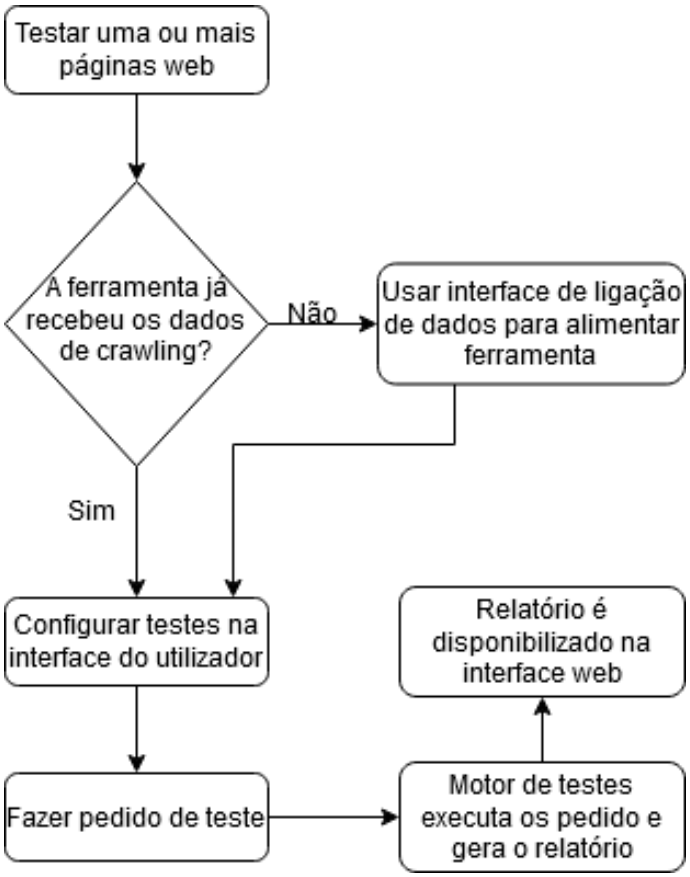


Figura 3.6: Fluxograma da ferramenta

Antes de entrar em detalhe sobre o funcionamento sequencial dos componentes mais importantes do sistema, a interface do utilizador e o motor de testes, é necessário perceber como a ferramenta e todos os seus componentes funcionam como um todo. O fluxograma da Figura 3.6 descreve essas relações. O processo para realizar um pedido de teste começa com a alimentação da ferramenta, que é feita recorrendo ao componente apresentado anteriormente, a interface de ligação de dados. Este funciona como uma ponte entre qualquer programa ou software externo, para fornecer os dados de *crawling* à ferramenta.

Depois da ferramenta receber esta informação, o utilizador poderá acede-la através da interface do utilizador, onde poderá começar a configurar, para cada uma destas páginas, os valores de teste. Posteriormente a esta configuração, o utilizador finalizará o pedido, que irá disparar o início da execução do motor de testes. Este, ao iniciar, verifica se tem pedidos para executar, e se sim, inicia o seu processamento. O motor de testes executa os testes e gera os relatórios com os resultados da sua análise, que são depois disponibilizados na interface do utilizador.

3.5.2 Interface do utilizador

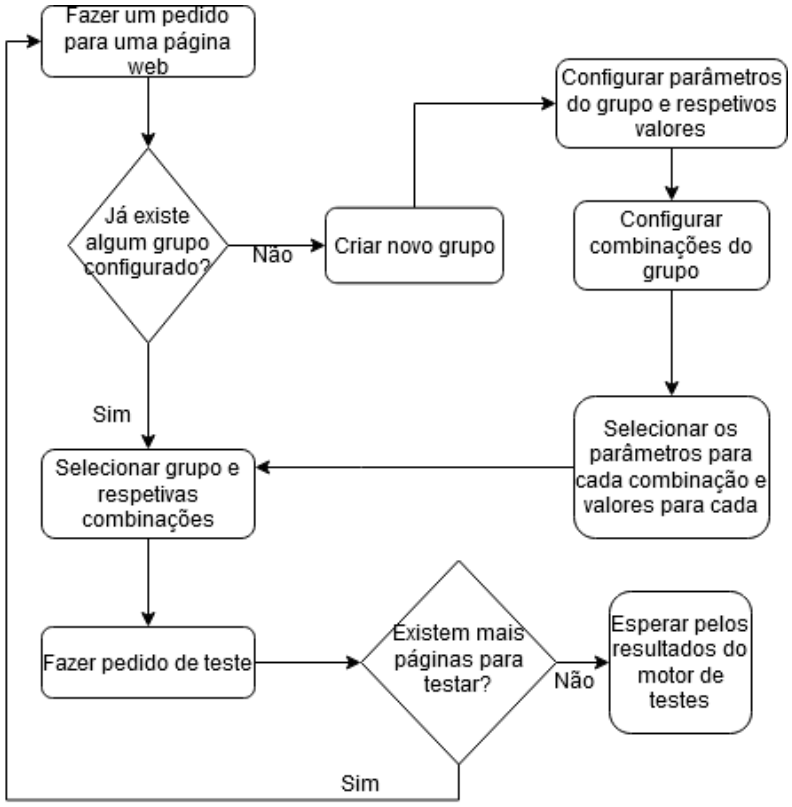


Figura 3.7: Fluxograma da interface do utilizador

A interface do utilizador, para permitir grande configurabilidade, funcionará à volta de quatro conceitos principais: grupos, combinações, parâmetros e valores de parâmetros. Um grupo nada mais é do que um conjunto de uma a n combinações e uma coleção de um a n parâmetros com um a n valores. Uma combinação, por sua vez, é uma coleção de um a n parâmetros do grupo, cada um com um a n dos seus valores e com possibilidade de atribuir a estes parâmetros grupos especiais para a exploração de vulnerabilidades.

O fluxograma da Figura 3.7 descreve as ações que um utilizador deverá tomar para realizar um pedido de teste a uma página. Para fazer um pedido de teste é preciso atribuir um grupo, e dentro desse grupo escolher entre uma a n das suas combinações para testar. Se já existir algum grupo criado com as configurações desejadas, basta escolher esse e quais combinações e criar o pedido. Caso contrário, é preciso configurar um novo grupo, onde será necessário especificar quais os seus parâmetros e valores e detalhar também uma coleção de combinações. Estas combinações, por sua vez, irão ter uma coleção de parâmetros oriundos dos parâmetros do grupo e valores específicos. Depois de criado o grupo basta seguir o procedimento descrito atrás e criar o pedido.

3.5.3 Motor de testes

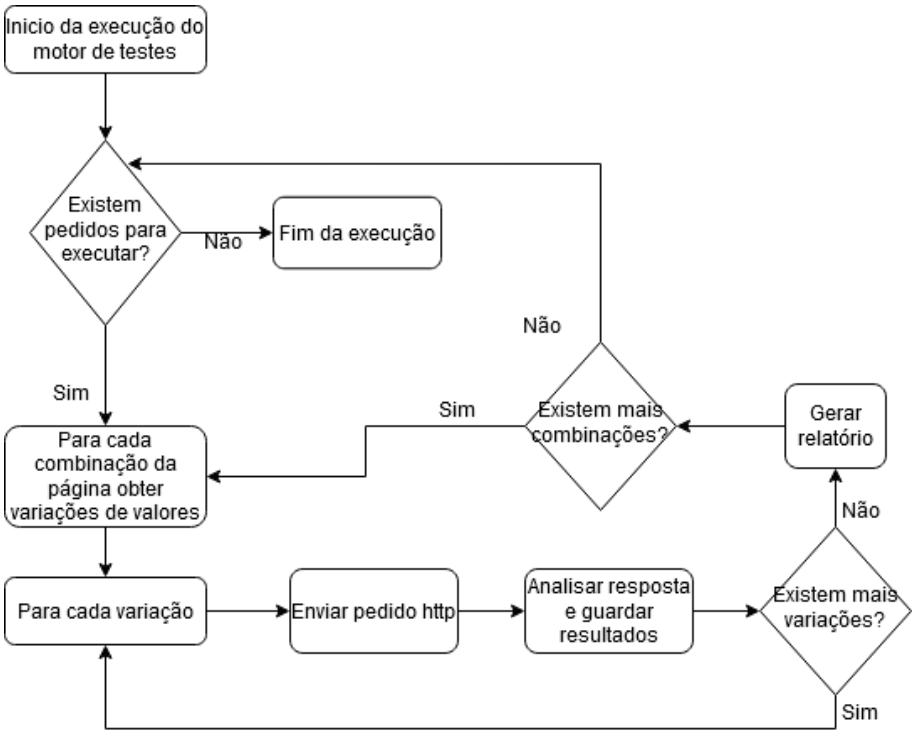


Figura 3.8: Fluxograma do motor de testes

O fluxograma da Figura 3.8 descreve o fluxo de ações de uma execução normal do motor de testes. Tal como ilustra, quando o motor de testes inicia a sua execução, a primeira coisa que verifica é se tem algum pedido para executar. Se sim, vai obter todos os pedidos pendentes e percorrer cada um deles. Em cada pedido tem de ir buscar os grupos e as combinações especificadas para a respetiva página. De seguida, tem de percorrer cada uma dessas combinações e para cada uma delas obter as diferentes coleções (sendo que cada uma corresponde a uma variação dos parâmetros da página a testar).

Para cada coleção o motor envia um pedido HTTP com os parâmetros configurados com os valores da coleção. A seguir a fazer o pedido, espera pela resposta, e aquando da sua chegada envia essa resposta para o *parser*, que analisa o corpo e verifica se foi encontrada alguma vulnerabilidade. Quando o motor acaba de percorrer todas as coleções, repete o mesmo procedimento para todas as combinações de uma página e quando completa o ciclo e todas as combinações e respetivas coleções já foram percorridas, gera o relatório. A seguir a concluir o relatório verifica se ainda existe mais algum pedido e, se sim, repete todo este processo, se não, concluí a sua execução.

3.6 Conclusões

Ao longo deste capítulo é apresentada uma possível arquitetura para uma ferramenta de *fuzzing* de aplicações web. São primeiro elucidados alguns conceitos chave que clarificam como uma aplicação web funciona e como pode ser testada. Para tal, é feita uma pequena introdução ao protocolo HTTP e à estrutura dos seus pacotes, mostrando como estes podem ser manipulados, de forma a explorar vulnerabilidades nas aplicações.

De seguida são apresentados os casos de uso desejados para a ferramenta, mostrando quais as funcionalidades que a ferramenta deverá ter. Depois é feita uma introdução aos conceitos de requisito funcional e não funcional, seguido de uma listagem, respetivamente, dos desejados para a arquitetura. Posteriormente, é apresentada a arquitetura que nasce destes requisitos e todos os seus componentes, detalhando para cada um qual a sua função, quais as suas ligações de entrada e saída, que requisitos garante e como o consegue. Por fim, encontramos uma secção na qual são descritas as sequências das ações, através de fluxogramas, que a ferramenta deverá ter. É mostrado, de forma mais pormenorizada, quais as sequências de ação da ferramenta na sua totalidade, e de dois dos seus componentes mais importantes, o motor de testes e a interface do utilizador.

Com este capítulo é concluído que uma arquitetura no seu eu mais abstrato, tenta definir boas bases e normas, cumprindo com os requisitos encontrados, de forma a agilizar todo o processo de implementação e tornar a solução final num produto mais forte e bem definido. No próximo capítulo, será abordado o processo de implementação da arquitetura aqui proposta.

Capítulo 4

Implementação do sistema, testes e validação

Depois do estudo feito aos requisitos do sistema, sequência de eventos e resultante arquitetura, é implementado o sistema. Nesta secção será mostrado o resultado do processo de implementação dessa arquitetura, tal como todas as escolhas que foram determinantes para conseguir cumprir com os requisitos e objetivos definidos ao longo deste documento. Primeiro serão apresentados alguns aspetos gerais de implementação da ferramenta, tais como as tecnologias utilizadas e as ligações entre os componentes arquiteturais. De seguida serão apresentados em detalhe os processos de implementação de alguns dos componentes mais críticos, nos quais foram encontrados os maiores desafios. Por fim, são apresentados os resultados dos testes feitos ao sistema e as suas validações, isto é, que objetivo e requisitos é que a solução implementada conseguiu cumprir, e quais não.

4.1 Implementação

Nesta secção vai ser discutido o processo de implementação dos componentes que foram mais desafiantes, começando por apresentar o sistema de armazenamento de dados, passando pela interface do utilizador, pela interface de ligação de dados, terminando no motor de testes. Para cada um deles, serão mostrados alguns detalhes de implementação, tais como as tecnologias utilizadas, os requisitos cumprem e como, e alguns aspetos mais desafiantes e críticos na sua implementação.

4.1.1 Interface de ligação de dados

A implementação deste componente teve de ter em conta o cumprimento do requisito de flexibilidade, de forma a puder ser utilizado por outros programas ou soluções, mais concretamente por outros projetos de C#, no seu meio de *deployment*. Como tal, este foi implementado como uma *Dynamic Link Library* (DLL) [29]. Uma DLL nada mais é do que uma biblioteca de código que pode ser utilizada por vários programas em simultâneo.

O uso das DLL evita a repetição de código ao modular segmentos de código, fazendo com que possam ser utilizados por tantos programas quantos forem precisos. Outra mais valia das DLL é o facto de serem de fácil uso e de não precisarem de ser *re-linked*, aquando um *update*.

```
public int AdicionarPedido(Guid chave){ ... }

public int AdicionarPedidoLink(Guid chave, int idpedido, int
    idmodulo, string diretoriobase, string caminho){ ... }

public int AdicionarPedidoLinkParametros(Guid chave, int
    idlink, string nome, Tipo tipo, int? valoromissaoint,
    double? valoromissaofloat, DateTime? valoromissaodata,
    string valoromissaostring){ ... }

public int ConcluirPedido(Guid chave, int idpedido){ ... }

public int DestruirPedido(Guid chave, int idpedido){ ... }
```

Listing 4.1: Funções da biblioteca de ligação

Ao recorrer a uma DLL, assegura-se que programas externos possam aceder, indiretamente e com segurança, ao sistema de armazenamento de dados, pois funcionando como um *layer* de filtros, a DLL implementada oferece uma coleção de funções que permite a qualquer programa fornecer dados, num certo formato, para serem posteriormente armazenados no sistema de armazenamento de dados. A Listagem 4.1.1 representa as funções disponibilizadas pela DLL e o tipo de dados que aceitam. Cada uma destas funções tem o seu objetivo, sendo que o objetivo final é fornecer à ferramenta informação sobre as páginas (os seus parâmetros e *links*) que o utilizador quer testar.

Componente	**Entrada**	**Saída**
Interface de ligação	*Strings*, *Doubles*, Inteiros, Guids, *Datetime*	Sistema de armazenamento de dados

Tabela 4.1: Ligações à interface de ligação de dados

Na Tabela 4.1 encontramos as ligações, a um nível de implementação, feitas a este componente. Como vimos na Listagem 4.1.1 a DLL recebe dados através das funções, num formato específico, que varia entre *strings*, inteiros, *Globally Unique Identifier* (GUID) [27] (que são inteiros muito grandes utilizados como identificadores únicos), *doubles*, entre outros. Como saída, a DLL armazena os dados que recebe, depois de validados e considerados seguros, no sistema de armazenamento de dados.

4.1.2 Interface do utilizador

A interface do utilizador podia ter sido implementada de diversas formas, desde que cumprisse com dois requisitos não funcionais: o ambiente de desenvolvimento, na qual é dito que a linguagem a ser utilizada é o C#, e o requisito da implantação do software, no qual é definido que este componente terá de comunicar diretamente com uma base de dados relacional, mais especificamente gerida pelo software da Microsoft o Microsoft SQL Server.

Outro requisito não funcional, talvez o mais importante, é a usabilidade e como tal, ao longo do capitulo anterior foram descritos cuidadosamente os conceitos base e as sequências de operações que o utilizador terá de fazer para configurar um pedido. Como tal, todas estas especificações vão ter de ser cumpridas na implementação da interface do cliente.

Como a interface precisa de uma ligação ativa à internet, para comunicar com o base de dados, e de forma a simplificar a sua utilização, foi decidido que a interface seria uma aplicação web. Para esta decisão também contribuiu o facto de ser o tipo de solução de software mais utilizado na instituição, sendo que toda a equipa, eu inclusive, já adquirimos bastante treino e experiência na área. Outro fator determinante foi o da simplicidade de utilização pois, ao ser uma aplicação web, os utilizadores não têm de instalar nada na sua máquina, basta apenas ter acesso a um *browser* e não existe necessidade de reinstalar a ferramenta, em caso de novas versões.

A interface do utilizador foi então implementada na *framework* da Microsoft, o .NET, sendo um projeto de *web forms*, escrito em C#. Ao todo foram desenvolvidos 7 formulários diferentes, recorrendo a diversas outras bibliotecas como o Bootstrap [15], que é uma biblioteca de estilos CSS e JQuery [19] que é uma biblioteca com funções JavaScript.

Na Tabela 4.2 estão expressas as ligações feitas a este componente. Recebe como dados de entrada os dados que o utilizador insere nos sete diferentes formulários. De seguida, e de forma a cumprir o requisito de segurança e de integridade dos dados, valida cuidadosamente estes dados e guarda-os no sistema de armazenamento de dados.

Componente	Entrada	Saída
Interface do utilizador	Dados do formulário inseridos pelo utilizador	Sistema de armazenamento de dados

Tabela 4.2: Ligações à interface do utilizador

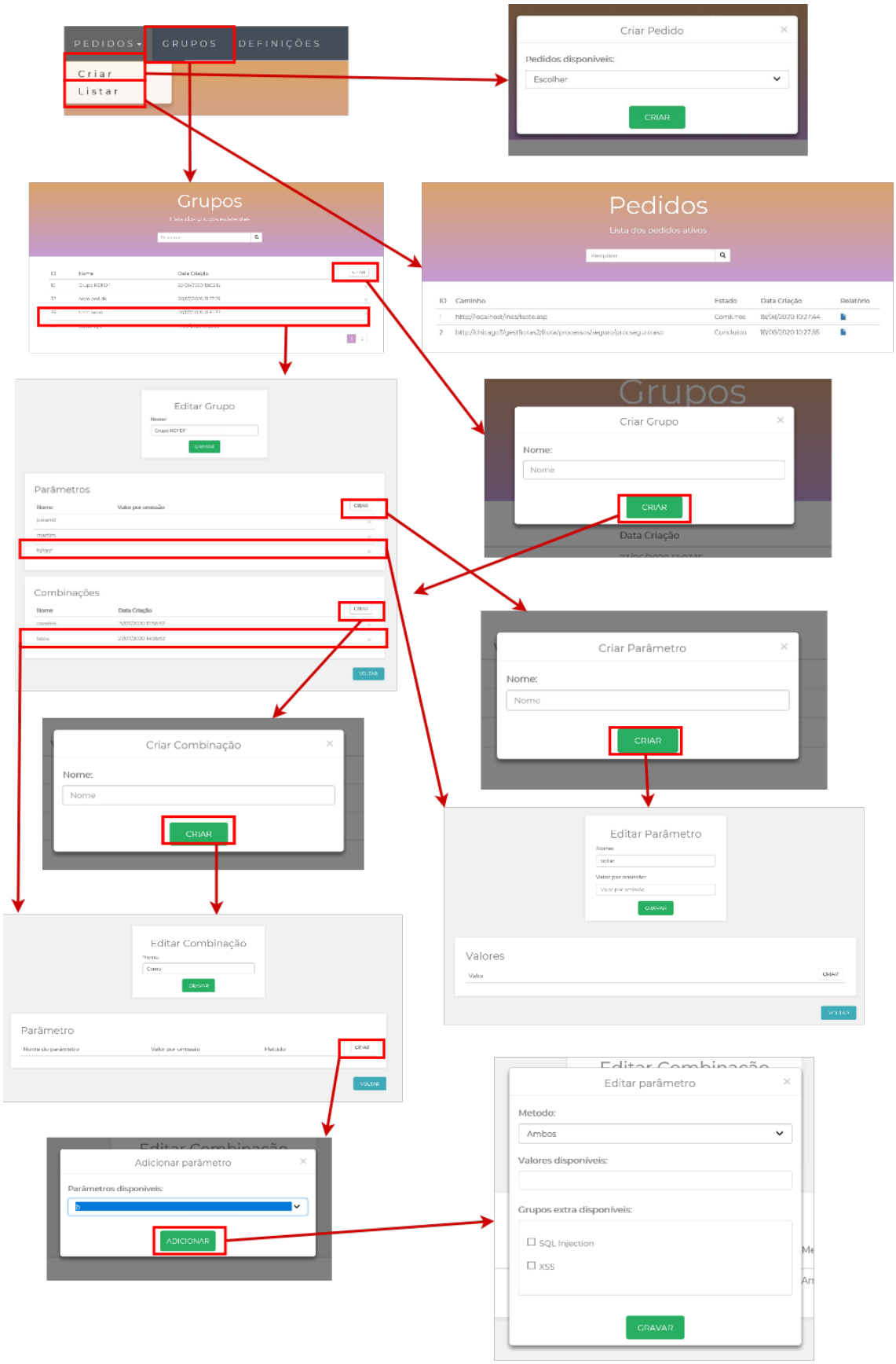


Figura 4.1: Sequência de formulários da interface do utilizador

Na Figura 4.1 estão expressos os sete formulário desenvolvidos e as sequências de ações que o utilizador tem de executar, de forma a fazer um pedido de teste. Como é visível, os conceitos (grupo, pedido, combinações, parâmetros) definidos no capítulo anterior, tal como a sequências de ações, foi seguido à risca. De tal forma que para criar um pedido o utilizador, ou já tem um grupo configurado, ou tem de criar um novo. No segundo caso terá de, para esse novo grupo, configurar os parâmetros e combinações desejadas, tal como esquematizado na Figura 4.1 e tal como foi descrito no capítulo anterior.

4.1.3 Motor de testes

Tal como a interface do utilizador, o motor de testes pode ser implementado de diversas formas, desde que cumprisse alguns requisitos não funcionais: o ambiente de desenvolvimento, que especifica que a linguagem no qual este componente tem de ser desenvolvido (C#) e a implantação de software, que dita que o componente comunicará com uma base de dados relacional, gerida com o software Microsoft SQL Server, descrito na próxima secção. O componente terá também de ter um bom desempenho, eficiência, flexibilidade e dado comunicar com o sistema de armazenamento de dados, terá de contribuir para a integridade dos mesmos.

Como tal, este componente foi implementado na *framework* da Microsoft, o .NET, sendo um projeto do tipo *console app*, que funciona como um serviço de execução periódica, sendo que no contexto de *deployment*, foi configurado para executar a cada hora.

O motor de testes foi escrito de forma a executar de acordo com as sequências de ação definidas no capítulo anterior. Quando este inicia a sua execução (e relembrando a divisão do motor de testes em três subcomponentes, esquematizado na Figura 3.4) o *fuzzer* começa por aceder ao sistema de armazenamento de dados e verifica se existem pedidos para executar. De forma a cumprir dois dos requisitos, a segurança e a integridade de dados, todos os acessos feitos ao sistema de armazenamento de dados terão de ser feitos de forma segura e, para tal, neste componente terá se ser utilizada uma biblioteca, parte integrante do sistema do armazenamento de dados, para permitir estes acessos. Na próxima secção será explicado com maior detalhe como todos estes acessos são feitos e mantidos seguros.

Depois de obter os dados, o *fuzzer* tem de preparar os vetores de ataque para a página em causa e, para tal, foi criada uma classe iteradora que recebe estes dados e devolve em cada iteração os parâmetros da página preenchidos. Este iterador contribui para o desempenho da ferramenta, pois limita bastante o número de testes por página, revelando aqui uma das suas componentes inovadoras, comparativamente com as ferramentas apresentadas no capitulo 2.

Na listagem 4.2 é apresentado um exemplo do funcionamento do iterador criado para o núcleo do motor de testes. Aqui, é esquematizada uma situação de teste, no qual uma página com três parâmetros (param1, param2 e param3) é testada. Cada um desses

parâmetros tem atribuído respetivamente dois, três e dois valores (visíveis nas três primeiras linhas da listagem). O iterador, a partir desses valores devolve iteração após iteração, as combinações visíveis nas restantes linhas da listagem. No total, são devolvidas doze combinações, para serem posteriormente utilizadas nos pedidos HTTP. O iterador devolve sempre o número de valores de cada parâmetro multiplicados entre si, logo neste caso temos dois * três * dois = doze combinações.

```
param1 = {a,b}
param2 = {c,d,e}
param3 = {f,g}

1 - param1=a & param2=c & param3=f
2 - param1=a & param2=c & param3=g
3 - param1=a & param2=d & param3=f
4 - param1=a & param2=d & param3=g
5 - param1=a & param2=e & param3=f
6 - param1=a & param2=e & param3=g
7 - param1=b & param2=c & param3=f
8 - param1=b & param2=c & param3=g
.....
12 - param1=b & param2=e & param3=g
```

Listing 4.2: Exemplo das resultantes iterações de uma coleção de parâmetros e respetivos valores

Depois de enviado cada pedido *http* (com as combinações mostradas no exemplo) o *fuzzer* fica a espera da resposta, e após a sua receção envia-a para o *parser*. O *parser*, por sua vez, percorre o corpo do pacote à procura de padrões específicos que indiquem a presença de uma vulnerabilidade no código. Se encontrar alguns desses padrões no corpo da resposta, guarda essa informação para que no final do processamento de um pedido, o gerador de relatórios os utilize para gerar o relatório.

Nesta fase, entra em cena um outro requisito, o da flexibilidade. Era desejado que a ferramenta pudesse produzir relatórios em diferentes formatos e para tal, foi criada uma interface, com simples funções que recebem os dados, relativos aos erros encontrados, que geram relatórios num qualquer formato. Ao fazer isto qualquer programador pode facilmente criar uma classe que implemente esta interface para produzir relatórios no formato desejado. No contexto de *deployment*, foi criada uma classe que implementa esta interface e que produz relatórios em formato HTML.

Componente	Entrada	Saída
Motor de testes	Sistema de armazenamento de dados	Sistema de armazenamento de dados

Tabela 4.3: Ligações ao motor de testes

Na Tabela 4.3 encontramos todas as ligações feitas a este componente. Ambas as entradas e saídas deste, são direcionadas ao sistema de armazenamento de dados, que como será explicado na próxima secção é feito com o auxilio de uma biblioteca que acede a uma gama de *stored procedures*, criados para manipular as tabelas da base de dados de forma segura.

4.1.4 Sistema de armazenamento de dados

A escolha do tipo de sistema de armazenamento de dados foi feita à priori, sendo um dos requisitos não funcionais. Existiam inúmeras soluções possíveis, desde base de dados não relacionais e os seus respetivos softwares para gestão, a soluções menos eficientes como guardar dados em ficheiros, entro outros. Para cumprir com o requisito que foi imposto pela instituição de acolhimento, neste projeto será utilizada uma base de dados relacional, desenvolvida pela Microsoft e gerida pelo software o Microsoft SQL Server [24], com a estrutura de dados apresentada na secção 3.4.6.

Todos os dados com que o sistema lida, desde os dados resultantes da interface de ligação de entrada, os dados colhidos e manipuladas na interface do utilizador, os dados gerados pelo motor de testes e pela interface de saída, serão guardados na mesma base de dados e como tal, todos os clientes através da interface do utilizador acedem à mesma fonte de dados. Assim, é importante garantir que todos os acessos a esta base de dados são seguros, de forma a assegurar a sua integridade.

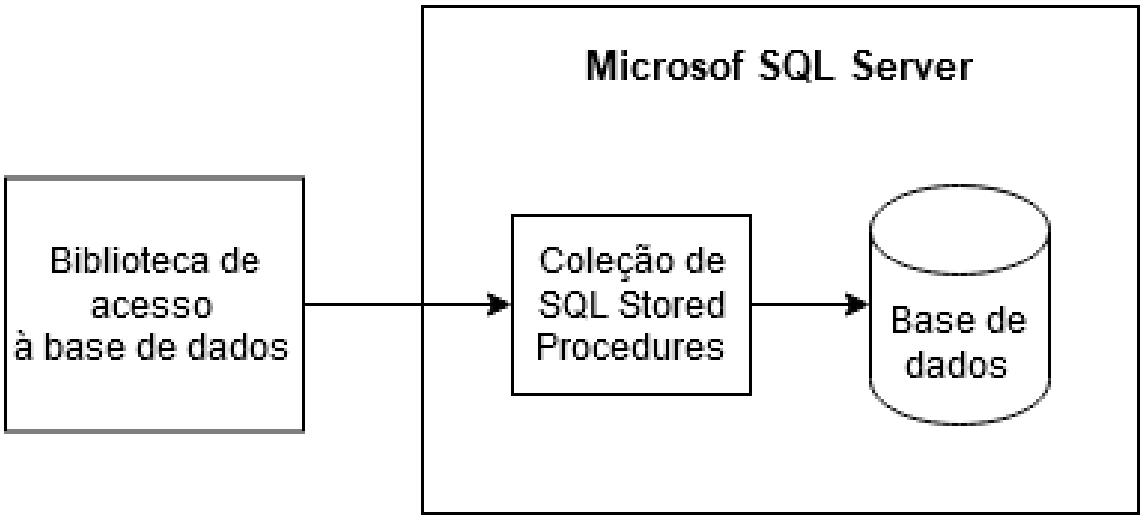


Figura 4.2: Implementação do sistema de armazenamento de dados

Para cumprir os requisitos de segurança, o sistema de armazenamento de dados foi implementado tal como ilustra a Figura 4.2. A figura está dividida em duas secções (o quadrado grande com os dois componentes dentro e o componente externo). Na maior secção (quadrado grande com os dois componentes) encontram-se a base de dados relacional, com todas as suas tabelas, e uma coleção de procedimentos. Uma das vantagens do uso do software de gestão da Microsoft (Microsoft SQL Server) é o facto de este nos permitir criar *stored procedures* [26] para manipular as tabelas da base de dados.

Um *stored procedure* nada mais é do que uma função guardada na base de dados. Ao utilizar estas funções para todos os acessos à base de dados é introduzida mais uma camada de proteção adicional, permitindo que sejam feitas validações quer lógicas quer de segurança, e garantindo com que o acesso às tabelas nunca seja feito diretamente pelos componentes.

Componente	**Entrada**	**Saída**
Sistema de armazenamento de dados	Dados (string, guid) do sistema de autenticação	Interface do utilizador
Sistema de armazenamento de dados	Dados dos formulários da interface do utilizador, previamente validados	Motor de testes
Sistema de armazenamento de dados	Dados recebidos e validados pela dll de ligação	Interface do utilizador
Sistema de armazenamento de dados	Dados gerados pelo motor de testes	Interface do utilizador
Sistema de armazenamento de dados	Links dos relatórios gerados pela interface de saída	Motor de testes

Tabela 4.4: Ligações ao sistema de armazenamento de dados

Ao todo foram desenvolvidos quarenta e sete procedimentos para a manipulação das tabelas da base de dados. Estes procedimentos fazem a verificação dos parâmetros que recebem, quer logicamente quer em termos de segurança, e devolvem coleções de dados, alteram dados existentes, criam novos dados ou apagam dados existentes.

Existe ainda uma terceira camada, também desenhada para conferir proteção ao sistema, que é um componente externo à base da dados, apresentado na Figura 4.2 com o *label* "Biblioteca de acesso à base de dados". Esta é uma biblioteca de funções, escrita em C#, que todos os componentes que acedem à base de dados deverão utilizar. A biblioteca foi escrita e desenhada pelos programadores seniores da escrita digital e tem como objetivo invocar os *stored procedures* existentes na base da dados, de forma segura, podendo

ou não passar parâmetros. É então um componente externo que deverá ser utilizado por todos os projetos de C# que acedem à base de dados.

Na Tabela 4.4 estão esquematizadas todas as ligações feitas ao sistema de armazenamento de dados. Todos estes acessos, tal como explicado anteriormente, são feitos através da biblioteca de C# (o componente externo). Estas ligações correspondem aos acessos dos componentes como a interface do utilizador, motor de testes, DLL de ligação de dados, que foram descritas nas secções anteriores. Basta apenas acrescentar que estes acessos são mediados pela biblioteca de C#, que por sua vez chama um dos quarente e sete procedimentos possíveis, que manipulam as tabelas, criando, apagando ou alterando linhas das mesmas.

4.2 Testes

Os testes de software são utilizados para quantificar a qualidade de um produto ou solução e para verificar se cumpre com todos os requisitos funcionais e não funcionais [1]. Os testes de software podem ser divididos em três categorias, cada uma delas constituída por diferentes tipos de testes. Na categoria dos testes funcionais, que tal como o nome indica verificam se os requisitos funcionais foram cumpridos, encontram-se os testes de usabilidade, os testes de unidade, entre outros. Na categoria dos testes não funcionais, que verificam se estes requisitos não funcionais foram cumpridos, encontram-se os testes de desempenho, escalabilidade, entre outros. E por fim a categoria dos testes de manutenção.

Existe uma grande variedade de testes, sendo que cada um pertence a uma destas três categorias, e cabe ao programador escolher quais os testes que se adequam à solução, sendo que esta escolha depende totalmente do contexto e do tipo de solução. Para este projeto foram realizados testes funcionais, mais especificamente testes de usabilidade e testes não funcionais, realizados por mim e pela equipa de programadores da instituição de acolhimento.

Na Tabela 4.5 são apresentados sumariamente os diferentes testes feitos aos componentes da arquitetura da ferramenta. Para testar a correta funcionalidade do sistema de armazenamento de dados foram feitas chamadas a todos os procedimentos que este disponibiliza, sendo que foram feitas chamadas com valores alvo, por exemplo passando valores a *null* ou identificadores de elementos que não existem, para verificar se os procedimentos conseguem executar corretamente em todos os casos.

Para a interface do utilizador, todos os formulários e respetivos campos (*text-box*, *check-box*, botões, entre outros) foram testados. No motor de testes diferentes pedidos foram executados também com valores de exceção, por exemplo, pedidos para páginas não existentes entre outros. Na interface de ligação e de saída, as funções que ambas disponibilizam foram testadas passando diferentes gamas de valores. Com estes testes foram detetados alguns erros que foram corrigidos atentamente antes de passar a ferramenta

Componente	**Teste**	**Resultado**
Sistema de armazenamento de dados	Chamadas aos procedimentos	Procedimento manipulam corretamente a base de dados
Interface do utilizador	Preenchimento dos formulários	Formulários submetidos com sucesso
Motor de testes	Dados recebidos e validados pela ddl de ligação	Interface do utilizador
Interface de saída	Chamadas às funções disponibilizadas	Funções executam corretamente e geram o relatório no formato desejado
Interface de ligação	Chamadas às funções disponibilizadas	Funções recebem os valores e guardam-nos com sucesso no sistema de armazenamento de dados

Tabela 4.5: Testes feitos aos diferentes componentes

para produção.

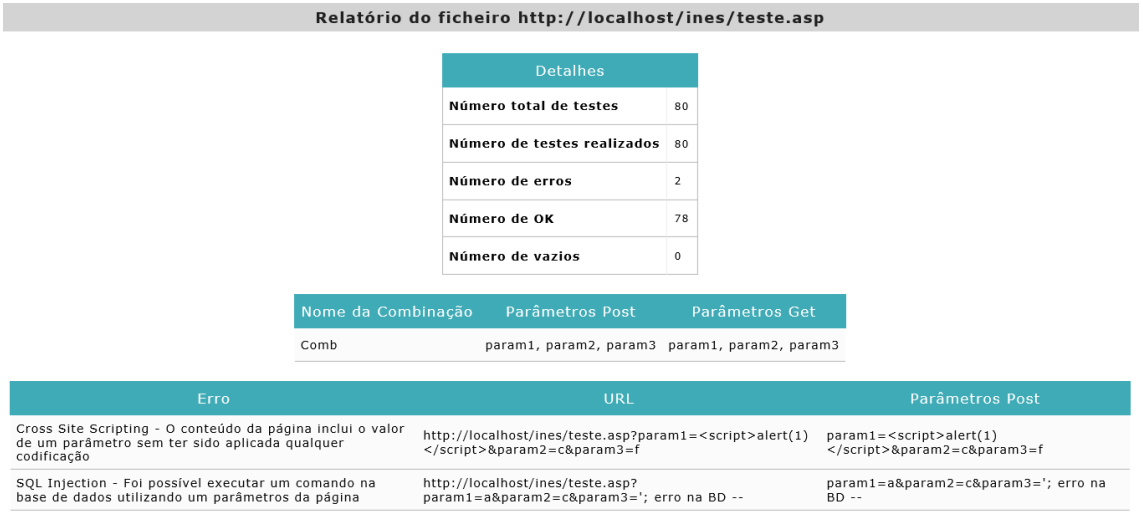
Os testes de usabilidade foram realizados em ambiente de produção. Depois da ferramenta ser cuidadosamente testada por mim, foi posta em ambiente de produção, onde foi utilizada por quatro programadores, que recorreram à ferramenta para testar as páginas que desenvolviam. O resultado deste teste foi bastante positivo, sendo que todos consideraram a experiência bastante agradável e simples, e melhor ainda, conseguiram cumprir com facilidade todos os casos de uso apresentados na secção 3.2. Destes testes não surgiu nenhum pedido de alteração quer da interface do utilizador, quer do formato dos relatórios.

Os testes de funcionalidade foram realizados maioritariamente por mim e por alguns membros da equipa. Todas as funcionalidades foram testadas a fundo e mostraram funcionar corretamente. A ferramenta foi largamente testada com várias páginas dos produtos desenvolvidos pela empresa e por algumas páginas criadas propositadamente para o efeito de demonstração. Estas últimas, foram desenvolvidas de forma a incluir no seu código fonte vulnerabilidades de XSS e de SQL Injection para a ferramenta detetar. Os resultados foram bastante animadores sendo que a ferramenta conseguiu detetar todas as vulnerabilidades deste género. Na próxima secção será abordado com maior detalhe para como estes testes foram feitos, mais especificamente às páginas de demonstração.

4.3 Validação do sistema

Ao longo desta secção serão apresentados os resultados do processo de validação, começando por demonstrar os objetivos que foram cumpridos e depois os requisitos. Para tal, foi feito um teste no qual todas as funcionalidades foram testadas, sendo que o objeto de alvo, como mencionado na secção anterior, foi uma página criada propositadamente com vulnerabilidades. Esta página, escrita em ASP e chamada ”teste.asp”, foi desenvolvida de forma a consumir valores provenientes de três parâmetros (chamados param1, param2 e param3), e de forma a ter duas vulnerabilidades no seu código, uma de XSS e uma de SQL Injection.

Para iniciar o teste, foi utilizada a interface de ligação para carregar a informação (link e parâmetros) da página. Tal foi feito com recurso à aplicação externa (ED Segurança), que posteriormente recebeu a DLL, permitindo carregar a informação através das funções disponibilizadas na biblioteca. De seguida tornou-se necessário configurar o teste em si, e tal foi feito na interface do utilizador, onde foi criado um novo grupo com os parâmetros descritos a cima (Relembrando: param1 = a,b, param2 = c,d,e, param3 = f,g). Depois foi configurada uma combinação (nomeada ”comb”), na qual os parâmetros previamente configurados foram selecionados. Cada um desses parâmetros da combinação foi ainda configurado para incluir os valores dos grupos extra de XSS e SQL Injection (sendo que cada um destes dois grupos extra contem um valor de ataque). Por fim, foi criado o pedido com o grupo configurado, e, depois de ser processado pelo motor de testes, foi obtido o seguinte relatório expresso na Figura 4.3.



Relatório do ficheiro http://localhost/ines/teste.asp

Detalhes	
Número total de testes	80
Número de testes realizados	80
Número de erros	2
Número de OK	78
Número de vazios	0

Nome da Combinação	Parâmetros Post	Parâmetros Get
Comb	param1, param2, param3	param1, param2, param3

Erro	URL	Parâmetros Post
Cross Site Scripting - O conteúdo da página inclui o valor de um parâmetro sem ter sido aplicada qualquer codificação	http://localhost/ines/teste.asp?param1=<script>alert(1)</script>&param2=c&param3=f	param1=<script>alert(1)</script>&param2=c&param3=f
SQL Injection - Foi possível executar um comando na base de dados utilizando um parâmetros da página	http://localhost/ines/teste.asp?param1=a&param2=c&param3='; erro na BD --	param1=a&param2=c&param3='; erro na BD --

Figura 4.3: Relatório gerado pelo motor de testes

Ao analisar o relatório é possível verificar que todos os testes foram executados. Tal como explicado na secção 4.1.3 o número de testes depende do número de parâmetros e respetivos valores, neste caso o teste continha três parâmetros com quatro, cinco e quatro

valores, respetivamente, o que origina um total de quatro*cinco*quatro = oitenta testes. Na primeira linha do relatório esse valor é confirmado. Na segunda linha é validado que todos esses testes foram executados, logo que não ocorreu nenhuma exceção que fizesse com que o pedido HTTP não fosse entregue ou que não tivesse resposta. Na terceira linha encontramos o número de erros, tal como mencionado anteriormente a página tinha duas vulnerabilidades no seu código fonte, ambas foram descobertas. Na quarta linha encontra-se setenta e oito ”OK”logo oitenta-dois = setenta e oito pedidos sem erros na resposta. E por fim na última linha verificamos que todas as respostas recebidas foram processadas pelo servidor, pois apresentavam conteúdo (respostas com volume superior a zero bytes).

Na última tabela encontramos mais detalhes sobre os erros encontrados, desde uma descrição do tipo de vulnerabilidade ao link que as originou. A primeira, de XSS, foi resultante da exploração do parâmetro ”param1”, e a segunda, de SQL Injection, do parâmetro ”param3”.

Com este teste de funcionalidade é concluído que a ferramenta cumpre com todos os requisitos e opera de acordo com o que foi definido na capitulo 3. A aqui arquitetura apresentada possibilita a implementação de uma ferramenta que cumpre com todos os requisitos e objetivos estabelecidos ao longo do documento.

4.4 Conclusões

Ao longo deste capítulo é relatado o processo de implementação da arquitetura apresentada no capítulo 3. Os componentes com maior relevância foram apresentados, sendo eles a interface de ligação, que foi implementada como uma DLL, a interface do utilizador, que foi criada como uma aplicação web, utilizando a *framework* da Microsoft o .NET escrita em C#, sendo um projeto do tipo *web forms*. O motor de testes, que foi desenvolvido com recurso à *framework* .NET, também escrito em C# desenvolvido como um projeto do tipo *console app* e por fim o sistema de armazenamento de dados, que incluí vários componentes desde a base de dados em si, relacional, o software para sua gestão, o Microsoft SQL Server, que permitiu a criação de um *layer* de *stored procedures* para a manipulação das tabelas e uma biblioteca escrita em C# para a invocação destes *stored procedures*.

Para todo estes componentes foram explicados alguns detalhes mais técnicos e alguns resultados da sua implementação, tal como todas as ligações que são feitas aos mesmos. De seguida foram apresentados os resultados dos testes feitos ao sistema implementado, testes funcionais e não funcionais, que demonstram a qualidade do sistema e que requisitos foram cumpridos.

Por fim, foram apresentadas as validações dos sistema, nas quais é apresentado o sucesso ou insucesso da ferramenta implementada em relação ao que foi definido no capitulo

3. Aqui foi mostrado que objetivos, casos de uso, requisitos funcionais e não funcionais foram cumpridos com sucesso e quais não foram.

Capítulo 5

Conclusão e Trabalho Futuro

5.1 Conclusão

Ao longo deste documento foi documentado e estudado o desenvolvimento e implementação de uma ferramenta de testes de segurança *black-box*. Esta ferramenta foi desenvolvida no contexto de uma instituição de acolhimento e recorre à técnica de fuzzing para detetar vulnerabilidades em aplicações web.

Antes de apresentada a arquitetura, são definidos todos os requisitos funcionais e não funcionais e casos de uso desejados para a ferramenta, nos quais é explicado que a ferramenta será implementada utilizando a *framework* .NET, mais especificamente a linguagem C#, e que utilizará um sistema de armazenamento de dados com uma base de dados relacional gerido pelo software da Microsoft, o Microsoft SQL Server. De seguida foi apresentada a arquitetura da ferramenta onde são dados a conhecer todos os seus componentes e qual o seu papel no cumprimento dos requisitos. São identificados os seguintes: uma interface de ligação, que mais tarde na implementação, é apresentada como uma DLL; um motor de testes, que executa os pedidos dos utilizadores e é implementado como uma aplicação de consola na *framework* do .NET; um sistema de armazenamento de dados, que oferece uma biblioteca que garante a segurança dos acessos às dezenas de *stored procedures* criados para a manipulação das tabelas da base de dados; uma interface de utilizador, implementada como uma aplicação web também na *framework* do .NET e, por fim, uma interface de saída, que se traduz numa interface para qualquer classe em C# implementar.

Ao longo das duas secções, a da arquitetura e a da implementação, é mostrado como esta ferramenta e todos os seus componentes foram criados e como comunicam entre si, de forma a cumprir os objetivos definidos. De seguida, são referidos os testes feitos à mesma, que garantem com que estes requisitos, casos de uso e objetivos, são cumpridos. São apresentados os testes funcionais e não funcionais feitos à mesma, onde é relatado o processo de teste a uma página criada propositadamente para o efeito, contendo vulnerabilidades no seu código fonte. Com isto, concluísse que este documento relata o processo

de sucesso de desenvolvimento e implementação de uma ferramenta de testes *black-box* que recorre à técnica de *fuzzing* para testar aplicações web.

5.2 Trabalho Futuro

Um dos requisitos deste projeto e de certa forma um objetivo, foi desenvolver uma solução que fosse simples e adequada às necessidades especificas da instituição de acolhimento, mas que em simultâneo fosse adaptável e flexível a futuras mudanças nos requisitos. Ao longo deste documento foi apresentada uma arquitetura e respetiva implementação de uma ferramenta de testes de segurança que cumpre exatamente esse requisito, desde a forma como o cliente se autentica, à forma como recebe os dados de *crawling*, a como produz os relatórios de resultados. Esta ferramenta oferece interfaces e camadas que proporcionam uma grande adaptabilidade a futuras mudanças.

Como tal, num futuro, será desejável que a ferramenta receba dados de outras fontes externas, que é possível devido à biblioteca de ligação. Será desejável, se o contexto assim o escolher, que a ferramenta produza relatórios noutro tipo de formatos (por exemplo .pdf, .csv, .txt) e que permita ao utilizador, talvez através de uma configuração na interface do utilizador, especificar o nível de detalhe destes relatórios, podendo ter um relatório com mais ou menos informação.

Outra área de interesse para explorar no futuro, é a capacidade de oferecer mais informação sobre as vulnerabilidades encontradas. Como vimos no capitulo 2, as vulnerabilidades de XSS dividem-se em três categorias, *reflected, stored e DOM*, e é possível através de técnicas no *parsing* das respostas HTTP (que envolvem guardar os valores utilizados para o teste, entre outros) distinguir entre estas categorias, oferecendo mais informação ao utilizador. Poderá também ser interessante, no momento de configuração dos parâmetros das combinações, oferecer mais grupos extra (sem ser os de exploração da vulnerabilidade de XSS e SQL Injection). Estes terão gamas de valores, não de ataque, mas de omissão, por exemplo um grupo com uma data, um inteiro, um *float*, uma string, para no caso das páginas validarem os tipos dos parâmetros, aumentar as probabilidades de sucesso de execução da página.

Por fim, se for desejável introduzir a ferramenta num ambiente onde terá de lidar com grandes volumes de dados, será interessante afinar o motor de testes de forma a poder ter n instâncias a correr em paralelo, desta forma conseguindo processar n vezes mais dados. Concluindo, a ferramenta está preparada para lidar com futuras mudanças nos requisitos, mesmo tendo sido criada à imagem da minha empresa poderá ser transformada numa ferramenta que adaptável a inúmeros contextos.

Glossário

black-box Sistema no qual apenas existe conhecimento sobre os seus inputs e output, sem nenhum conhecimento interno.

browser Software para aceder à World Wide Web.

cross-plataform Software implementado em múltiplas plataformas computacionais.

fuzzing Método para a descoberta de falhas em software através da introdução de input inesperado e procura de exceções nos respetivos outputs.

grey-box Combinação de *black-box* e *white-box*.

open-source Adjetivo que se dá a qualquer software cujo código fonte é público.

white-box Sistema no qual apenas existe conhecimento sobre os seus componentes internos.

Siglas

API *Application Programming Interface*.

ASP *Active Server Pages*.

CWE *Common Weakness Enumeration*.

DAST *Dinamic Application Security Testing*.

DLL *Dynamic Link Library*.

DOM *Document Object Model*.

DOS *Denial of Service*.

GUID *Globally Unique Identifier*.

HTML *Hypertext Markup Language*.

HTTP *HyperText Transfer Protocol*.

IAST *Interactive Application Security Testing*.

MS SQL *Microsof SQL Server*.

OWASP *Open Web Application Security Project*.

REST *Representational state transfer*.

SAST *Static Application Security Testing*.

SQL *Structured Query Language*.

UI *User Interface*.

URL *Uniform Resource Locator*.

WAVSEP *Web Application Vulnerability Scanner Evaluation Project*.

XML *Extensible Markup Language*.

XSS *Cross Site Scripting*.

ZAP *Zed Attack Proxy*.

Bibliografia

[1] Guru 99. What is software testing? definition, basics & types, 2020. https://www.guru99.com/software-testing-introduction-importance.html.

[2] Acunetix. Dom xss: An explanation of dom-based cross-site scripting, 2015. https://www.acunetix.com/blog/articles/dom-xss-explained/.

[3] Michael Sutton | Adam Greene | Pedram Amini. *Fuzzing - Brute Force Vulnerability Discovery*. Addison Wesley, 2007.

[4] Arachni. Vulnerability detection, 2017. https://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/#vulnerability-detection.

[5] Arachni. Web application security scanner framework, 2017. https://www.arachni-scanner.com/.

[6] Security Tools Benchmarking. Evaluation od web application vulnerability scanners in modern pentest/ssdlc usage scenarios, 2018. http://sectooladdict.blogspot.com/2017/11/wavsep-2017-evaluating-dast-against.html.

[7] Intersoft consulting. General data protection regulation - gdpr, 2019. https://gdpr-info.eu/.

[8] C# Corner. Most popular databases in the world, 2019. https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/.

[9] Ibéria Vitória de Sousa Medeiros. *Detection of vulnerabilities and automatic protection for web applications*. PhD thesis, Universidade de Lisboa, 2016.

[10] Escrita Digital. Quem somos, 2017. http://www.escritadigital.pt/edicoes/escritadigital/desenvolvimento.asp?categoria=411&artigo=904&rev=.

[11] Science Direct. Static analysis - an overview, 2020. https://www.sciencedirect.com/topics/engineering/static-analysis.

[12] Flexera. Secunia research, 2020. https://www.flexera.com/products/operations/software-vulnerability-research/secunia-research.html.

[13] Security Focus. Securityfocus, 2010. https://www.securityfocus.com/.

[14] Inc. Git Hub. Wavsep, 2014. https://github.com/sectooladdict/wavsep.

[15] GitHub. Bootstrap, 2020. https://getbootstrap.com/.

[16] Imperva. Cross site scripting (xss) attacks, 2020. https://www.imperva.com/learn/application-security/cross-site-scripting-xss-attacks/.

[17] Iocscan. Reflected cross site scripting (r-xss), 2020. https://medium.com/iocscan/reflected-cross-site-scripting-r-xss-b06c3e8d638a.

[18] Gadi Evron | David Maynor | Noam Rathaus | Charlie Miller | Robert Fly | Yoav Naveh | Aviram Jenik. *Open Source Fuzzing Tools*. Syngress Publishing, Inc, 30 Corporate Drive Burlington, MA 0180, 2019.

[19] jQuery. jquery, 2020. https://jquery.com/.

[20] LifeWire. How http works: Hypertext transfer protocol explained, 2020. https://www.lifewire.com/hypertext-transfer-protocol-817944.

[21] Miniwatts MarketingGroup. World internet usage and population, Outubro 2019. https://internetworldstats.com/stats.htm.

[22] Balume Mburano. Evaluation of web vulnerability scanners based on owasp benchmark. *2018 26th International Conference on Systems Engineering (ICSEng)*, Dezembro 2018.

[23] MDN. Hypertext transfer protocol (http), 2020. https://developer.mozilla.org/en-US/docs/Web/HTTP.

[24] Microsoft. Microsoft sql server 2019. https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[25] Microsoft. C# documentation, 2020. https://docs.microsoft.com/en-us/dotnet/csharp/.

[26] Microsoft. Create a stored procedure, 2020. https://docs.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-ver15.

[27] Microsoft. Guid struct, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.guid?view=netcore-3.1.

[28] Microsoft. Httpclient class, 2020. https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=netcore-3.1.

[29] Microsoft. What is a dll, 2020. https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library.

[30] MongoDB. The database for modern applications, 2019. https://www.mongodb.com/.

[31] Lianping Chen | Muhammad Ali Babar | Bashar Nuseibeh. Characterizing architecturally significant requirements. *IEEE*, 2012.

[32] US Department of Homeland Security CISA Cyber +Infrastructure. Black box, white box, and gray box testing, Dezembro 2005. https://www.us-cert.gov/bsi/articles/tools/black-box-testing/black-box-security-testing-tools.

[33] Oracle. My sql, 2019. https://www.mysql.com/.

[34] Oracle. Oracle database, 2019. https://www.oracle.com/database/.

[35] OWASP. Dom based xss, 2015. https://www.owasp.org/index.php/DOM_Based_XSS.

[36] OWASP. Types of cross-site scripting, 2017. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting.

[37] OWASP. Cross-site scripting (xss), Junho 2018. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[38] OWASP. Owasp top ten 2017 project, Fevereiro 2018. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project.

[39] OWASP. Benchmark, Novembro 2019. https://www.owasp.org/index.php/Benchmark.

[40] OWASP. Owasp foundation, Dezembro 2019. https://www.owasp.org/.

[41] OWASP. Owasp zed attack proxy project, Junho 2019. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[42] Dafydd Stuttard | Marcus Pinto. *The Web Application Hacker's Handbook*. Wiley Publishing, 10475 Crosspoint Boulevard, Indianapolis, Indiana, 2008.

[43] Tutorials Point. Compiler design - lexical analysis, 2020. https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm.

[44] Tutorials Point. Natural language processing - semantic analysis, 2020. https://www.tutorialspoint.com/natural_language_processing/natural_language_processing_semantic_analysis.htm.

[45] QRA. Functional vs non-functional requirements: The definitive guide, 2020. https://qracorp.com/functional-vs-non-functional-requirements/.

[46] ResearchGate. Http protocol communication schema, 2020. https://www.researchgate.net/figure/HTTP-protocol-communication-schema_fig1_224227608.

[47] Emmanuel Cecchet | Veena Udayabhanu | Timothy Wood | Prashant Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. *Proceedings of 2nd USENIX Conference on Web Application Development (WebApps)*, 2011.

[48] Emmanuel Cecchet | Veena Udayabhanu | Timothy Wood | Prashant Shenoy. Open source web vulnerability scanners: The cost effective choice? *Proceedings of the Conference for Information Systems Applied Research*, 2014.

[49] Miguel Pupo Correia | Paulo Jorge Sousa. *Segurança no Software*. FCA, Av. Praia da Vitória 14 A, 1000-247 Lisboa, 2017.

[50] Statista. Retail e-commerce sales worldwide from 2014 to 2023, Agosto 2019. https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/.

[51] WAVSEP. The web application vulnerability scanner evaluation project, 2014. https://code.google.com/archive/p/wavsep/.