Qualidade e Confiabilidade de Software associado à Injeção de Vulnerabilidades e Ataques

João Silva
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologias da
Universidade de Coimbra

Inês Marçal
Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologias da
Universidade de Coimbra

***Abstract*- Assegurar qualidade num software pode tornar-se desafiador. Além de ser necessário garantir que o mesmo corresponda aos requisitos estabelecidos pelo cliente (requisitos funcionais), o mesmo deve de atender a determinadas restrições (requisitos não funcionais). Por outro lado, sendo a qualidade composta por diversos atributos, se apenas dois destes integrarem o software poderá continuar a afirmar-se que este apresenta qualidade? Aliado a isto, acresce ainda o aumento da complexidade de um software, dificultando de igual modo assegurar confiabilidade. Um meio pelo qual se poderá atingir este aspeto é através das técnicas de tolerância a falhas ao qual se encontra associado o conceito de tolerância a intrusões. Introduzido este conceito, é imprescindível referenciar uma metodologia que permita testar o mesmo, injeção de vulnerabilidades e ataques.**

Keywords- qualidade, confiabilidade, tolerância a falhas, injeção de vulnerabilidades e ataques

I. INTRODUÇÃO

Nos dias de hoje, o *software* desempenha um papel fundamental na nossa vida, já que necessitamos do mesmo para a execução de grande parte das atividades realizadas no dia a dia. Por outro lado, associado a esta dependência emergem novas questões: "se se considerar dois *software*s com a mesma funcionalidade, será indiferente escolher qualquer um?" Certamente que a resposta é "não", visto que cada *software* apresenta as suas vantagens e desvantagens. Fatores como a qualidade e confiabilidade têm um enorme impacto no que se refere à escolha de um *software*, sendo estes que demovem as empresas a criarem tecnologias cada vez mais confiáveis e que correspondam às expectativas dos clientes. No caso de sistemas sensíveis, como os de tempo real, a confiabilidade e qualidade são considerados componentes ainda mais críticos. Em contrapartida, incorporar estas características no desenvolvimento de *software* têm-se tornado extremamente desafiador e difícil de atender à medida que a tecnologia evolui e, deste modo, aumenta a complexidade dos requisitos estabelecidos pelos utilizadores.

Ora, sendo a confiabilidade e qualidade dois conceitos difíceis de definir e até mesmo de alcançar, é fundamental explicitar os atributos que são tidos em conta quando nos referimos a estes, uniformizando a maneira de pensar relativamente aos mesmos. No âmbito da confiabilidade, serão exploradas técnicas de tolerância a falhas, uma vez que tais métodos possibilitam a aproximação da maximização deste atributo. De modo a testar a eficácia destas mesmas técnicas, um procedimento a seguir poderá ser a injeção de vulnerabilidades e, posteriormente, de ataques. Assim, conseguiremos obter uma melhor perspetiva de como o sistema se comporta na presença de falhas.

II. QUALIDADE DE *SOFTWARE*

Diversas definições têm sido propostas, algumas das quais standardizadas. No entanto, as mesmas refletem o quão vago e abstrato poderá ser este conceito, tornando-se de difícil identificação os critérios que demonstram a sua presença, mas evidentes os que determinam a sua ausência. Este pode mesmo variar caso estejamos na perspetiva de um utilizador ou de um desenvolver. Por exemplo, o standard ISO 25010:2011 define qualidade de *software* como "o grau de satisfação de um *software* para com as necessidades implícitas e estabelecidas mediante as condições especificadas de uso".[1] Enquanto que o standard IEEE 729-1983 apresenta um total de quatro definições: "o conjunto de recursos e características de um produto de *software* que determinam sua capacidade de satisfazer determinadas necessidades: por exemplo, atender às especificações"; "até que ponto o *software* possui a combinação desejada de atributos"; "até que ponto um cliente ou usuário percebe que o *software* atende às suas expectativas"; "o conjunto de características de *software* que determinam até que ponto o *software* em uso irá de encontro às expectativas do cliente." [2]

Por outro lado, um *software* pode ser categorizado sob duas perspetivas: a vista funcional e a vista não funcional. A primeira remete para o que o *software* é suposto fazer, sendo detalhado as especificações do mesmo nos requisitos funcionais. Ora, neste ponto de vista, a qualidade representa a conformidade entre as funcionalidades que o *software* proporciona e as expectativas dos utilizadores. A vista não-funcional descreve como é que o sistema se comporta ao nível de outros aspetos além dos seus requisitos funcionais, como o desempenho e a usabilidade. É, precisamente, a estes conceitos que é associada a designação de atributos de qualidade. [4] Apesar destas duas visões fornecerem diferentes perspetivas sobre o que é

considerado qualidade, ambas são importantes para avaliar a presença deste atributo num *software*. [3]

De modo a estabelecer uma estrutura fixa e consolidada dos atributos considerados e, consequentemente, dos diferentes pontos de vista no processo de avaliação de qualidade de *software* foram desenvolvidos modelos de qualidade. Os mais conhecidos são o modelo de *McCall*, o *Boehm* e o ISO/IEC 25010. Os dois primeiros demonstram ser muito semelhantes, no entanto o modelo de *McCall* foca-se principalmente em medir a capacidade de o *software* atender às necessidades do utilizador, enquanto que o modelo de *Boehm* providencia uma maior ênfase ao atributo manutenção. [5] Para o estudo dos atributos que caracterizam tradicionalmente o conceito qualidade de *software* será utilizado como referência o modelo ISO/IEC 25010, visto ser o mais recente dos mencionados e se basear nos mesmos.

A. *Atributos*

1) *Adequação funcional*

Este atributo refere-se ao quão bem o *software* é capaz de providenciar as funcionalidades que atendem às necessidades especificadas e implícitas nos requisitos de sistema. Sendo essencial garantir que o *software* desempenhe as suas funções de forma completa, correta e adequada, estes três conceitos poderão ser definidos como subatributos do atributo adequação funcional. A completude funcional abrange todas as funções e recursos reivindicados pelo utilizador ou especificados nos requisitos. A segunda sub-caraterística tenta compreender o nível de precisão com que os resultados corretos são providenciados. E por último, a *appropriateness* foca-se na facilidade com que o *software* proporciona as funções corretas com vista a realizar as tarefas e os objetivos especificados (facilidade de uso, intuitividade da interface de utilizador. [1][6]

2) *Eficiência de desempenho*

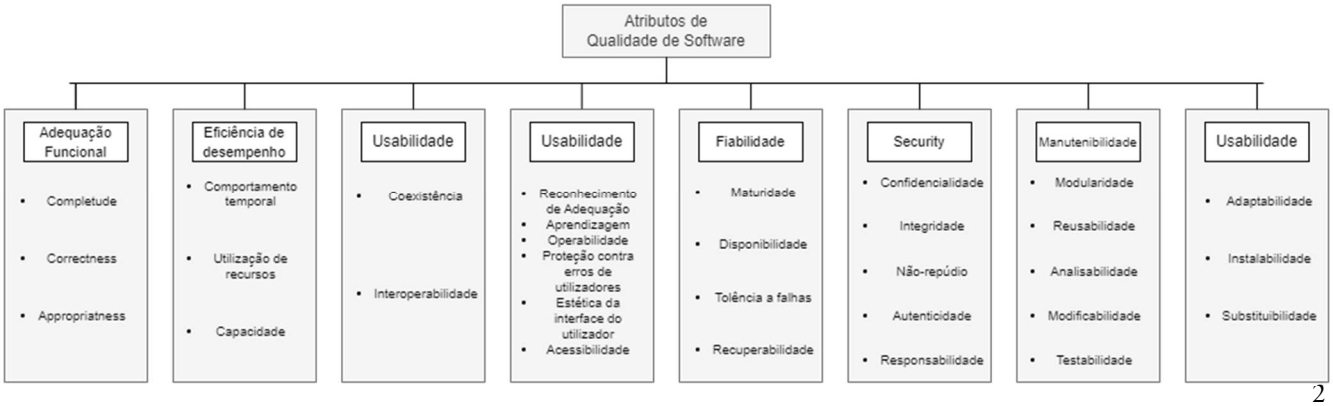
Remete para o desempenho do *software* consoante a quantidade de recursos utilizados, visto que diferentes condições implicam diferentes "cargas de trabalho". Ora, este atributo reflete, precisamente, o comportamento temporal, os recursos e a capacidade necessárias para que um *software* execute as suas funções. Estes três conceitos são essenciais para que seja possível definir este atributo, pelo que sub-caracterizam o mesmo. O comportamento temporal reflete os tempos de resposta, processamento e transferência das solicitações de um utilizador, enquanto que o segundo sub-atributo remete para a quantidade de recursos utilizados durante a execução das suas funções. Já através do sub-atributo capacidade pretende-se avaliar a capacidade de o *software* manter o seu desempenho perante uma quantidade crescente de utilizadores ou dados. [1][7]

3) *Compatibilidade*

Refere-se à capacidade de o *software* operar enquanto compartilha o mesmo ambiente com outro *software* ou *hardware*. Isto equivale a que diferentes serviços coexistam, mesmo que não se conheçam, sem efeitos colaterais ou que ao interagirem respeitem as restrições um do outro. [8] A este atributo encontram-se, assim, associados os sub-atributos coexistência e interoperabilidade. O primeiro remete para o quão bem um *software* consegue executar as suas funções eficientemente e sem interferir com os outros componentes de *software* e *hardware* presentes no mesmo ambiente. Já o segundo reflete a habilidade de trocar informações com outros componentes e de utilizar as mesmas. Ora, se um *software* apresentar problemas de compatibilidade, a sua qualidade será severamente afetada, limitando não só o seu alcance no mercado, visto ser dirigido apenas a um subconjunto de utilizadores, como também pondo em causa a sua estabilidade. Isto conduz a que o sistema não corresponda às expectativas do utilizador e, consequentemente, à insatisfação do mesmo. [1][7]

4) *Usabilidade*

Este atributo diz respeito à facilidade com que um *software* poderá ser utilizado, compreendido, aprendido, configurado e executado com vista a alcançar os objetivos especificados de forma eficaz, eficiente e satisfatória.[6] Ora, este atributo reflete, claramente, o quão bem os utilizadores conseguem interagir com o *software* e a qualidade da sua experiência, relacionando-se inteiramente com a satisfação do mesmo.[8] A este, por sua vez, encontram-se associados os seguintes sub-atributos: reconhecimento de adequação (reconhecer se um *software* é adequado para as necessidades requeridas), aprendizagem (simplicidade do processo que visa a aprendizagem do *software* em causa), operabilidade



(remete para a facilidade de manusear e controlar o *software*), proteção contra erros de utilizadores (tal como o nome indica, reflete a capacidade de proteger os utilizadores contra erros), estética de interface de utilizador (o quão agradável é a interface do utilizador) e acessibilidade (visa incluir o maior número de pessoas possível a utilizar um determinado *software*). [1][7]

5) Fiabilidade

Reflete a capacidade de um *software* executar as funções pretendidas sob condições especificadas e durante um período de tempo também estabelecido, sem erros ou falhas.[6] O atributo fiabilidade permite proporcionar aos utilizadores uma experiência consistente e previsível, o que contribui seriamente para o aumento da sua satisfação e confiabilidade para com o *software* em causa. Em simultâneo, poderá proceder-se à sua caracterização através dos seguintes sub-atributos: maturidade, que diz respeito à habilidade de um *software* atender às necessidade de confiabilidade requeridas numa operação normal; disponibilidade, onde se remete para a operacionalidade e acessibilidade de um *software* quando é necessário a sua utilização; tolerância à falhas, avaliando-se a capacidade de um *software* operar de forma expectável na presença de falhas ou erros; e, recuperabilidade que, por sua vez, representa a habilidade de um *software* conseguir regressar ao seu estado anterior ou retomar as operações normais em caso de interrupção ou falha. [1][7]

6) Segurança

Remete para a capacidade de um *software* proteger os seus dados, *assets* e recursos contra eventuais ataques, evitando o acesso, uso, divulgação, modificação, destruição ou interrupção não autorizados e mitigando as vulnerabilidades contidas no mesmo. A este atributo encontram-se associados os seguintes sub-atributos: não-repúdio, de modo a garantir que não seja possível negar a realização de uma determinada ação; responsabilidade, assegurando que as ações efetuadas por uma entidade apenas possam ser associadas a essa mesma entidade; autenticidade, através do qual se pretende comprovar a identidade de uma determinada entidade ou sistema; e a integridade e confidencialidade que serão detalhadas, precisamente, na categoria confiabilidade. [1][7]

7) Manutenibilidade

Refere-se à facilidade e eficácia com que um *software* pode ser modificado de modo a melhorá-lo, corrigi-lo ou adaptá-lo a novos ambientes e requisitos. [8] Um *software* difícil de manter irá implicar maiores custos associados e tempo requerido ao longo do seu ciclo de vida útil na correção de *bugs* e atualizações, uma maior dificuldade em testar e garantir os requisitos de qualidade e poderá mesmo vir a constituir um obstáculo se se pretender proceder ao seu dimensionamento de forma a acomodar o crescimento. Todos estes aspetos são considerados nos seus sub-atributos: modularidade, reusabilidade, analisabilidade, modificabilidade e testabilidade. A modularidade remete para a habilidade de efetuar alterações em componentes específicos de um *software* sem que afete as outras partes, requerendo, assim, a independência entre os mesmos. Por sua vez, o segundo sub-atributo refere-se à capacidade de os componentes de *software* poderem vir a ser reutilizados em outras partes do *software*, enquanto que através da analisabilidade pretende-se avaliar o impacto das modificações efetuadas. Já a modificabilidade remete para a capacidade de um *software* ser modificado sem degradar a sua qualidade e o sub-atributo testabilidade para a eficácia dos critérios de teste aplicados. [1][7]

8) Portabilidade

Este atributo remete para a capacidade de um *software* ser transferido de um ambiente para outro e, consequentemente, de poder ser executado em diferentes *hardwares* e sistemas operacionais sem modificações. Um *software* portátil proporciona, assim, aos utilizadores uma maior flexibilidade e acessibilidade, economizando-se, simultaneamente, nos custos associados ao desenvolvimento de novas versões de *software* adequadas a cada ambiente e na sua respetiva manutenção.[8] Deste modo, poderá caracterizar-se o atributo portabilidade através de três sub-atributos: a adaptabilidade, que remete para a capacidade de uma *software* se adaptar a diferentes ambientes; instabilidade, que se refere à facilidade de instalação de um determinado *software*; e a substituibilidade, reflete a habilidade de um *software* substituir outro com propósitos semelhantes no mesmo ambiente. [1][7]

III. CONFIABILIDADE DE *SOFTWARE*

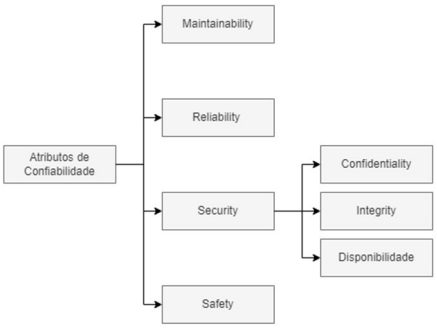
Definida como "a capacidade de fornecer um serviço tal que o mesmo seja confiável"[9], a confiabilidade aponta para ser uma das características mais importantes e requisitadas durante o desenvolvimento de *software*. Contudo, a definição que a confiabilidade apresenta acaba por ser bastante vaga e por vezes, dependendo do contexto, de difícil compreensão e implementação. As dúvidas quanto à confiabilidade surgem precisamente em tentar perceber o que realmente é um *software* confiável. *Software* confiável atende principalmente a duas características:

- evitar falhas inaceitavelmente frequentes/graves
- cumpre o fim para o qual está destinado [4][9]

Ainda assim, estas duas características voltam a ser de difícil definição, já que cada sistema para o qual o *software* é desenvolvido apresenta diferentes características, diferentes limitações, diferentes

atributos. Uma forma simplista de olhar para o *software*, usando como ponto de partida as duas características explicitadas anteriormente, é tentar perceber se o mesmo atinge o objetivo esperado e/ou não apresenta quaisquer erros durante a sua execução. Infelizmente, a concordância para com estas duas características não é assim tão simples, é muito mais que isso, é necessário que esta concordância tenha como base certos atributos aos quais a confiabilidade recorre.[10] Além disso, é necessário compreender quais os tipos de ameaças que podem pôr em causa a confiabilidade do *software* e, por conseguinte, tentar reduzir/mitigar da melhor forma estas mesmas ameaças. A confiabilidade assume, assim, um papel importante durante o processo de desenvolvimento, visto reforçar a confiança na escolha do *software* a utilizar num determinado sistema.

A. Atributos



1) Maintainability

É normalmente definido como a probabilidade de um *software* poder a vir ser "reparado" e/ou alterado com via a resolver situações adversas enquanto o mesmo é executado, alterações essas que incluem: resolver bugs, efetuar atualizações ou adicionar novas *features*. A *maintainability* é um atributo sem dúvida importante a ter em conta, não só pela importância que é conseguir manter o *software* operacional, como também pela sua relação parcial com a qualidade de *software*, já que em parte este mesmo atributo se encontra interligado com os custos aplicados durante o ciclo de desenvolvimento de *software*. [11] Os custos referidos acabam por estar relacionados com este atributo de diversas maneiras:

- Possíveis falhas não corrigidas enquanto o *software* é executado podem levar a grandes custos no futuro para a sua correção.
- O facto de o *software* não poder ser reparado durante a sua execução implica que o mesmo tenha de ser abortado para sua reparação/atualização, levando a que o custo englobe não só a falha a ser corrigida, como também as perdas de rendimento que o *software* pode gerar por se encontrar parado.

2) Reliability

A *reliability* é definida como a probabilidade de um *software* estar livre de falhas por um certo período de tempo num ambiente específico, ou seja, a capacidade que um *software* tem de continuar a cumprir o seu propósito corretamente.[4][12] É provavelmente o atributo mais importante, já que é neste que a definição de confiabilidade se apoia.

Como se pode imaginar é um atributo complicado de atingir a 100%, já que por vezes para um *software* ser confiável não depende só do próprio. Este *atributo* depende não só do ambiente que rodeia o *software* como das possíveis ameaças e danos que as mesmas possam vir a causar. [13]

3) Security

Normalmente, os atributos de *confidentiality*, *integrity* e *availability* são vistos de forma separada, no entanto os mesmos encontram-se interligados e são considerados atributos/características da *security*, formando a *CIA triad*.[16] Embora a *security* possa ser caracterizada de diversas maneiras, os seus atributos refletem de certa forma uma definição abrangente deste conceito:

- *Confidentiality*: ausência de divulgação não autorizada de um serviço/informação. É visto, portanto, como a capacidade de limitar o acesso apenas a quem tem a devida autorização.
- *Integrity*: proteção de um serviço ou informação contra modificações ilícitas (acidentais ou não) e/ou não detetadas, as quais podem levar em parte à corrupção do sistema no qual o *software* se integra.
- *Availability*: proteção contra possíveis *denials of service*, ou seja, garantia de o sistema fornecer um serviço sempre que necessário.

Claramente, podemos concluir que a *security* desempenha um papel fundamental na preservação dos seguintes modos básicos de acesso: "ver (*confidentiality*), modificar(*integrity*) e usar(*availability*)" [15], permitindo aumentar a confiabilidade perante um *software*.

4) Safety

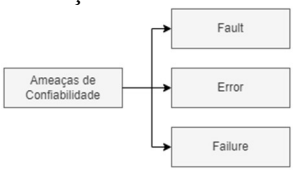
Este atributo é definido pela ausência de consequências catastróficas que possam provocar danos tanto aos utilizadores do *software* como ao ambiente em que o mesmo se encontra implementado.[12] Um *software* confiável deverá, assim, oferecer as funcionalidades para o qual foi especificado, minimizando, simultaneamente, o risco de poder vir a causar danos ao meio envolvente.

B. Ameaças

Segundo a visão da confiabilidade, é possível ter um conjunto de ameaças que podem influenciar o comportamento do *software* e, por sua vez, reduzir o nível de confiança para com o mesmo. É, assim, de todo recomendável que estas ameaças sejam removidas ou em último caso mitigadas de modo a ser

possível maximizar os atributos anteriormente enunciados.

À semelhança dos atributos, as ameaças poderão ser de difícil definição e variar de *software* em *software*, podendo mesmo depender umas das outras e originar uma cadeia de ameaças entre si.



1) Fault

Reflete, normalmente, o motivo pelo qual o sistema não consegue realizar a ação para o qual foi concebido. As *faults* podem ser, assim, entendidas como o conjunto de condições que devem ser reunidas para que o sistema sofra um desvio do comportamento considerado normal. Estas podem ser categorizadas através das seguintes questões: "Causadas por quem?", "Porquê?", "Quando?", "Onde?", "Por quanto tempo?". A resposta a estas questões providencia um melhor entendimento acerca de uma determinada *fault* e como prevenir que novas possam ocorrer. [4][12]

2) Error

Um *error* é definido como a diferença entre o estado atual do sistema e o estado ideal do mesmo, ou seja, é uma medida que permite avaliar o quão longe o *output* se encontra do pretendido.[4] É também considerado como um ponto de ligação entre *faults* e *failures*, já que no final de contas um erro é uma mudança errada de estado do sistema.

3) Failure

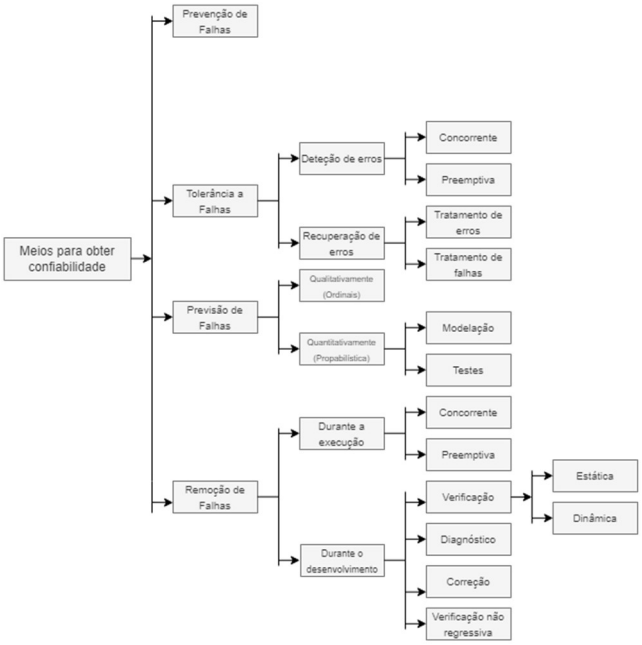
Por fim, as *failures* surgem através das ameaças anteriormente mencionadas, consistindo essencialmente na concretização das *faults* face aos *errors* gerados durante o desenvolvimento de um *software*.[4] Este tipo de ameaça é, portanto, definida como a incapacidade de um sistema (ou partes do mesmo) de cumprir os objetivos (segundo as suas especificações) inicialmente propostos do mesmo. [12]

4) Relacionando os conceitos Fault, Error e Failure

De modo a ilustrar melhor como estas 3 ameaças se relacionam, poderá considerar-se o seguinte caso:
Suponha-se que temos um dispositivo que faz monitorização da frequência cardíaca. O mesmo é composto por sensores que detetam sinais elétricos produzidos pelo coração, um processo que analisa estes mesmos sinais e calcula a frequência cardíaca e ainda um monitor que permite visualizar qual é a frequência cardíaca para a equipa médica. Considera-se que um dos sensores sofre uma avaria, levando o mesmo a efetuar leituras incorretas dos sinais (*fault*). Por sua vez, o processo recebe as leituras incorretas e calcula incorretamente a frequência cardíaca do paciente (*error*). Finalmente, a equipa médica, apoiada no cálculo da frequência cardíaca, toma uma decisão quanto ao tratamento que irá providenciar àquele paciente, o que, neste caso, pode resultar em sérios danos para o mesmo. Está-se, precisamente, perante uma *failure*, já que o dispositivo não será capaz de efetuar a função para o qual foi definido inicialmente e, assim, fornecer corretamente a frequência cardíaca à equipa médica.

C. Meios

Como já referido, obter confiabilidade num *software* não é fácil, como tal existem diversos meios para tentar atingir os atributos da mesma. Estes meios servirão, no fundo, para mitigar as ameaças anteriormente referidas e em contrapartida tornar o *software* mais confiável. É de esperar que estes meios sejam aplicados durante o processo de desenvolvimento, de modo a no futuro evitar problemas/danos que possam ser causados aos utilizadores e desenvolvedores do *software*.



1) Prevenção de Falhas

É um dos pontos mais importantes durante o desenvolvimento de *software*, pois mesmo que não tenha impacto imediato, as falhas podem no futuro ser a causa para inúmeros problemas, levando à perda de recursos como tempo e dinheiro. É imprescindível realizar uma análise minuciosa do *software* desenvolvido para identificar possíveis pontos de falha e, assim, implementar medidas de correção preventivas antes que possa ser tarde demais. [17]

A prevenção de falhas pode-se tornar complicada em alguns casos, como por exemplo, se a mesma for efetuada por alguém que não tenha muito conhecimento no projeto em questão ou até mesmo se os requisitos não estiverem corretamente definidos/priorizados. [18]

2) Tolerância a Falhas

Através da tolerância a falhas consegue-se obter confiabilidade de uma forma interessante, já que na ocorrência de falhas é garantido que o *software* continua a agir em conformidade com o seu propósito. Este meio consegue dividir-se em 2 implementações, deteção e recuperação de erros.

Dentro da deteção de erros temos 2 tipos de classes de técnicas: concorrente (efetua a deteção de erros durante a execução do *software*) e preemptiva (efetua a deteção de erros enquanto um *software* esteja suspenso, onde, por sua vez, pode detetar falhas "adormecidas" ou erros que não sejam tão visíveis).[4][12]

A recuperação de erros consiste na transformação de um sistema que esteja num estado errôneo, isto é, com erros e/ou falhas, para um estado em que estes mesmo defeitos deixem de existir. Assim como a deteção, a recuperação pode ser novamente dividida em 2 tipos: tratamento de erros ou falhas. O tratamento de erros pode consistir em diversos processos (*rollback*, *rollforward* ou compensação), mas estes apresentam sempre o mesmo propósito, eliminar erros. Já o tratamento de falhas remete para a ideia de impedir que as falhas encontradas voltem a ativar-se, sendo também possível dividir este conceito em diferentes processos (*diagnosis*, isolamento, reconfiguração, reinicialização). [12]

3) Previsão de Falhas

Esta técnica permite fazer um estudo do comportamento do *software* de modo a prever falhas não só no presente, como no futuro, bem como as possíveis consequências que as mesmas possam trazer. A avaliação segundo esta técnica pode seguir 2 aspetos: qualitativos (ordinais) ou quantitativos (probabilística).[4][17]

Os primeiros procuram identificar e classificar os diferentes modos de "falhas", ou combinações de eventos que podem levar o sistema a falhar.

Já os aspetos quantitativos visam efetuar uma avaliação em termos probabilísticos até que ponto alguns atributos pertencentes à confiabilidade são cumpridos. Estes podem ainda ser divididos em 2 categorias: modelação e testes, que serão complementares uma à outra, já que a modelação precisa de dados que podem ser obtidos pelos testes.[12]

4) Remoção de Falhas

A remoção de falhas consiste num conjunto de técnicas focadas na redução de falhas, podendo ser implementada durante a fase de desenvolvimento ou durante o ciclo de execução do *software*. Se olharmos para a fase de desenvolvimento este meio pode ser dividido em 4 fases: verificação (estática ou dinâmica), diagnóstico, correção e verificação de "não regressão".[17] A junção destas fases consistirá no seguinte: verificação inicial se os requisitos para aquele *software* são seguidos ou não, caso não sejam deverá ser efetuado um relatório (diagnóstico) indicando as falhas que evitam que esses mesmos requisitos não sejam seguidos. Por último, será feita a correção (ou remoção) destas mesmas falhas e a re-avaliação se perante esta forma o *software* já se apresenta em conformidade com o que foi estipulado.

Durante o ciclo de execução do *software*, será igualmente importante corrigir falhas e, consequentemente, removê-las, já que se pretende que o *software* continue a produzir o *output* correto no maior número possível de casos. São, assim, apresentados 2 tipos de manutenção: corretiva (onde se procura corrigir falhas que já tenham levado a erros/falhas) ou preemptiva (remover falhas encontradas após a execução do *software* e que eventualmente possam causar erros).[12]

IV. Resiliência E Robustez

A robustez pode ser definida, segundo *Laprie*, como: "a capacidade de fornecer um serviço de forma correta segundo condições fora do seu domínio normal de operação". Ora, torna-se evidente que tanto a robustez quanto a tolerância a falhas compartilham o objetivo de garantir que um sistema possa continuar a fornecer um serviço mesmo em condições adversas. Por outro lado, poderá definir-se robustez através do seguinte conjunto de atributos: *adaptability* (capacidade de um serviço adaptar dinamicamente o seu estado e comportamento consoante o contexto); *reparability* (habilidade de um sistema e dos seus mecanismos de reparação lidarem com situações inesperadas); *self-healability* (capacidade de um sistema entender autonomamente que não está a operar corretamente e estabelecer os ajustes necessários); *recoverability* (habilidade de restaurar os serviços essenciais durante um ataque e recuperar totalmente após o mesmo); *predictability* (comportamento expectável de um sistema não-determinístico); *survivability* (capacidade de resistir e recuperar a eventos adversos).[8]

A resiliência, por sua vez, é definida por *Laprie* como a junção dos conceitos confiabilidade e robustez: "a persistência de fornecer um serviço tal que o mesmo seja confiável na presença de mudanças" [19]

V. Técnicas de Tolerância a Falhas

As técnicas de tolerância a falhas providenciam um meio através do qual se poderá detetar e recuperar de falhas em *software*, garantindo que um sistema continue a operar correta e confiavelmente mesmo na presença destas. Estas podem ser categorizadas em *Single-version* ou *Multi-version*.

A. Single-version

Nesta técnica é aplicada redundância apenas a uma única versão de um módulo/componente de *software*

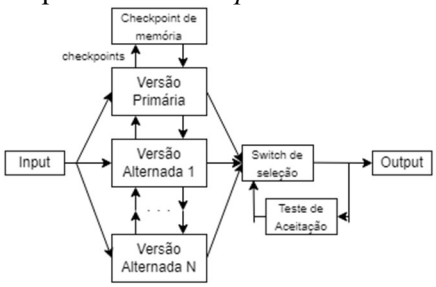
com vista a detetar e a recuperar de falhas. Para o efeito poderão ser consideradas as seguintes abordagens: deteção de erros, tratamento de exceções, checkpoint e *restart*, *process pairs* e diversidade de dados. [20][21]

B. *Multiple-version*

Esta técnica consiste no desenvolvimento de duas ou mais versões de *software,* o mais variadas possível, integrando diferentes algoritmos, designers e linguagens, que, por sua vez, poderão vir a ser executadas simultaneamente ou em sequência. Ora, esta estratégia rege-se, precisamente, na conjetura que em diferentes implementações dificilmente se comete erros semelhantes, pelo que os pontos de falha também poderão divergir. Assim, perante a ocorrência de uma falha numa destas versões em um determinado *input*, pretende-se que pelo menos uma das outras versões assegure o fornecimento de um *output* preciso e correto. [20]

1) *Recovery Blocks*

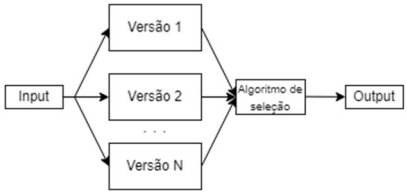
A técnica *recovery blocks* é semelhante à abordagem *stand-by sparing*[1] e combina os conceitos básicos dos métodos *checkpoint* e *restart*[2] aplicados na presença de múltiplas versões dos componentes de um *software*. Esta abordagem é constituída, assim, por uma versão primária (bloco principal), várias versões alternativas (blocos seguintes), sendo estas executadas sequencialmente, e ainda por um *Acceptance Test*, responsável por validar o *output* das diversas versões.



Quando se entra num bloco de recuperação, o estado atual do sistema é imediatamente guardado num *checkpoint* com vista a que seja possível retornar a um estado operacional válido perante uma falha e, consequentemente, a próxima versão retomar a execução. Após a primeira versão terminar de executar, o *output* da mesma é validado pelo *Acceptance Test* no sentido de verificar se cumpriu com sucesso os objetivos para os quais foi destinada. Se não tiver sido o caso, é efetuado *backward recovery* para o estado original, previamente marcado através do *checkpoint*, e a primeira versão é substituída pela segunda. Quando esta terminar, o *Acceptance Test* é executado novamente. Esta estratégia continua até que um suplemente cumpra corretamente as suas tarefas ou nenhuma das versões tenha sido executada com sucesso. É lançada uma exceção sempre que o *Acceptance Test* falhar, refletindo-se também numa falha para a estratégia *recovery blocks.* [20][21][22][23]

2) *N-Version Programming*

Nesta estratégia, as múltiplas versões (N) de *software* independentes, todavia equivalentes em termos de funcionalidade, são desenvolvidas por equipas de desenvolvimento e designers diferentes a partir da mesma especificação inicial. Esta assume tal abordagem, visto estar assente no pressuposto que *software*s independentes, executados concorrentemente, refletem falhas aleatórias (independência estatística), garantindo que não ocorre uma exaustão imediata das N cópias redundantes como na estratégia anterior. Todo este sistema constitui uma unidade dependente de um algoritmo de decisão genérico (habitualmente designado por eleitor) que, por sua vez, é responsável por determinar um consenso (N) ou pelo menos um resultado maioritário (N - 1) do conjunto de versões em causa. Caso nenhuma destas últimas situações referidas se verifique, é lançada uma exceção com o intuito de informar o resto do sistema que ocorreu uma falha na execução da tarefa. [20][21][22]



3) *Recovery Blocks vs N-Version Programming*

As diferenças entre estas duas abordagens não são muitas, mas é importante realçar as existentes [20]:

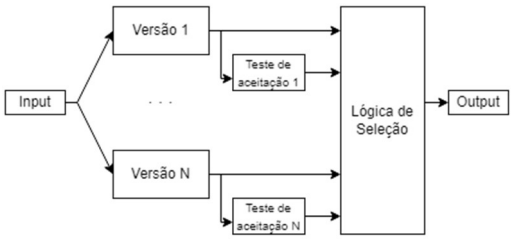
- A técnica *Recovery Blocks* tem como base a utilização *Acceptance Tests* (juiz), enquanto que a estratégia *N-Version Programming* recorre a um algoritmo de decisão (eleitor).
- No caso convencional da técnica *Recovery Blocks*, as diversas versões são executadas sequencialmente, embora já existam abordagens que incluem a execução em paralelo das diversas alternativas, enquanto que a estratégia N-*Version Programming* foi projetada desde início para executar as versões concorrentemente. Consequentemente, a primeira incorre num maior custo temporal ao invés da segunda.
- Na técnica *Recovery Blocks* são aceites diferentes saídas corretas provenientes das diversas alternativas, enquanto que o algoritmo *N-Version Programming* possui apenas uma única solução correta.

4) *N Self-Checking Programming*

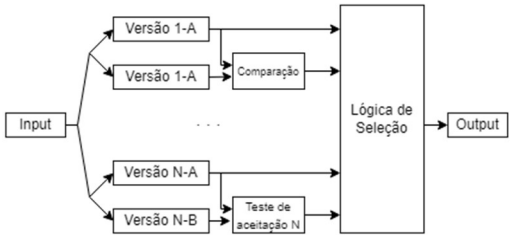
[1] Enquanto um módulo/componente de software se encontra operacional, os outros permanecem à espera. Perante uma falha, este é removido e substituído pelo próximo da lista de reserva.

[2] Abordagens alusivas às técnicas de tolerância a falhas de versão única que têm como ideia base guardar o estado válido de um sistema e perante uma determinada falha retomar ao mesmo, efetuando, assim, *backward recovery*. Para isso são posicionados diversos *checkpoints* no código, onde o sistema se encontra em estado consistente, e os mesmos são acionados por determinados eventos no sentido de guardar o seu estado.

Esta técnica apresenta duas abordagens e combina ambas as estratégias detalhadas anteriormente, *Recovery Blocks* e *N-Version Programming*, com pequenas variações estruturais. A primeira abordagem utiliza *Acceptance Tests*, procedendo-se a um desenvolvimento independente destes relativamente às versões de *software*, apesar de os requisitos serem comuns. Esta utilização de testes separados para cada versão, constitui, precisamente, a diferença mais notória entre este modelo de *N Self-Checking* e os *Recovery Blocks*. Aliado a isto, a cada versão é associado ainda um *ranking* que simboliza o nível de confiabilidade expectável para as mesmas. Contudo, tal como acontece no *Recovery Blocks*, esta estratégia também permite que as versões e os respetivos *Acceptance Tests* sejam executados quer sequencialmente (necessário empregar *checkpoints*) ou concorrentemente (requer algoritmos de consistência de estado), sendo selecionada pelo *Acceptance Test* o *output* da versão com maior classificação.[20][22]



A segunda abordagem procede à comparação de cada par de versões com vista a detetar erros. Para o efeito é utilizado um algoritmo de decisão que seleciona o *output* correto. Dado que este algoritmo é independente da aplicação, o seu processo de desenvolvimento torna-se relativamente mais simples que o dos *Acceptance Tests*. Apesar de isto se refletir numa vantagem para esta estratégia, a mesma apresenta uma falha teórica quando os pares provenientes das diferentes comparações produzem *outputs* diferentes, o que pode mesmo gerar inconsistências no resultado correto. [20]



VI. Injeção de Vulnerabilidades e Ataques

No contexto de confiabilidade dois conceitos que devem receber especial destaque são a tolerância a falhas e a segurança, já que estes são de enorme importância para obter um comportamento confiável por parte do *software*. A injeção de vulnerabilidades e ataques surge, assim, baseando-se na mesma ideia apresentada pela injeção de falhas, ou seja, introdução deliberada de perturbações num sistema de modo a avaliar o seu comportamento. Neste caso, as falhas e erros ganham outra nomenclatura, dado que o tema aqui abordado é a segurança, pelo que estes conceitos passam a ser vistos como vulnerabilidades e intrusões (ataques). Contudo, os mecanismos testados por esta técnica são um pouco diferentes, já que tem como objetivo avaliar a tolerância a intrusões.

A injeção de vulnerabilidades e ataques torna-se, assim, uma técnica relevante para a confiabilidade não só por misturar 2 conceitos importantes para a mesma, como também pelo papel fundamental que apresenta na sua obtenção. [24]

A. Conceitos

1) Vulnerabilidade

Reflete uma fraqueza existente num sistema que pode vir a ser explorada por um atacante com vista a executar um ataque com sucesso e, consequentemente, causar dano.[4]

2) Exploit

Consiste num pedaço de código ou *software* que tira partido de uma vulnerabilidade presente num sistema com vista a desencadear um comportamento não intencional ou imprevisível por parte do mesmo. Normalmente, um *exploit* encontra-se associado a uma vulnerabilidade em particular.[16]

3) Mecanismos de Segurança

Como referido a injeção de vulnerabilidades e ataques permite testar mecanismos de tolerância a intrusões, também conhecidos como mecanismos de segurança.

a) Sistemas de Deteção de Intrusões (IDS)

Com o aumento de ataques nas redes dos sistemas, os *IDSs* têm-se tornado cada vez mais fundamentais na área da segurança. *IDSs* são, portanto, *software* (ou *hardware*) especializado em efetuar a monitorização de possíveis problemas que possam ocorrer a nível da segurança de um sistema, nomeadamente intrusões, emitindo alertas na presença de potenciais comportamentos maliciosos. [25][26]

b) Sistemas de Prevenção de Intrusões (IPS)

Um *IPS* tem um comportamento, em parte, parecido com um *IDS*, com a diferença de que na presença de possíveis intrusões este age de modo a reduzir os impactos destas no sistema, detendo e detetando o ataque antes de atingir o alvo.[26] Como possíveis ações um *IPS* pode:

- dar *drop* dos pacotes de comunicações detetadas como maliciosas;
- rejeitar tráfego;
- reiniciar a comunicação.

c) Firewall

Até ao momento, os mecanismos referidos apenas agem perante tráfego externo. As *Firewalls*, além desta funcionalidade, apresentam também o cuidado de observar que tipo de tráfego é transmitido para o

exterior. Estas funcionam à base de regras que podem ser definidas tanto para portas específicas como para protocolos ou até mesmos endereços *IP*.[26]

d) Defense-in-Depth

Este mecanismo tem como ideia base a utilização de múltiplos controlos de segurança (camadas) de modo a criar redundância e proteger um sistema contra potenciais ameaças. Assim, através desta estratégia de segurança pretende-se que caso alguma das camadas de segurança falhe ou consiga ser ultrapassada, existam outras camadas que detenham ou reduzam o impacto do ataque. Esta estratégia permite aumentar os níveis de segurança e a proteção contra diferentes vetores de ataque. [16]

B. Procedimento

Esta metodologia permite testar a segurança de um *software* através da injeção de vulnerabilidades reais e posterior exploração das mesmas, avaliando-se, deste modo, a sua capacidade de detetar e responder a tais ameaças.

A injeção de vulnerabilidades pode ser efetuada de maneira automatizada, por meio de uma ferramenta, ou manualmente. Contudo, em ambos os casos, procede-se à análise do código fonte em busca de pontos estratégicos onde se possa vir a introduzir uma vulnerabilidade (alteração do código). Posteriormente, é utilizado um conjunto de *inputs* de ataque com o intuito de explorar as vulnerabilidades inseridas. No caso de uma *web application*, através deste procedimento pretende-se alterar a *query SQL* enviada para a base de dados. No final do ataque, a ferramenta de injeção de ataques avalia se o ataque foi bem-sucedido, o que equivale, se compararmos com as técnicas de injeção de falhas, a um estado de erro. [24]

Ora, uma potencial aplicação desta abordagem é, precisamente, na avaliação da eficácia dos mecanismos de deteção de intrusões (*IDS*, *IPS*, *firewalls*…).

1) Injecção de vulnerabilidades

Este constituiu um ponto fulcral durante a aplicação de "injeção de vulnerabilidades e ataques", já que todo o processo depende maioritariamente da qualidade das vulnerabilidades injetadas e se as mesmas se adequam ao contexto em que são inseridas (mecanismo de segurança a ser testado). De seguida, são demonstrados alguns exemplos de vulnerabilidades que podem ser injetadas ao utilizar a técnica de "injeção de vulnerabilidades e ataques".

a) SQL Injection

Considere-se o seguinte excerto de código:

```
$id=intval($_GET['id'])
```

Caso a variável $id estiver a ser utilizada numa *query*, este poderá constituir, precisamente, um ponto fulcral onde se poderá proceder à injeção de uma vulnerabilidade. Sendo que a função *intval* em *PHP* tem como papel converter um *input* em um inteiro (se não for possível retorna -1), a injeção da vulnerabilidade passaria por simplesmente remover esta função do código acima. A alteração resulta no seguinte:

```
$id=$_GET['id']
```

Através desta alteração, o código acima passará a ser vulnerável a *SQL Injection*, permitindo manipular de forma maliciosa *queries SQL*. Um exemplo de *exploit* seria a utilização de um *input* como "[qualquer coisa] or 1=1", já que junto, por exemplo, com uma condição "where =" o mesmo iria sempre ser avaliado com valor *true*.

b) Buffer Overflow

A linguagem C é pouco utilizada para o desenvolvimento de *software*, mas quando usada é necessário ter especial cuidado no que toca à gestão de memória. Atente-se ao seguinte exemplo:

```
char buffer[10];
strncpy(buffer, "biiiiiiiig string", sizeof(buffer)-1);
```

No código acima é utilizada a função *strncpy* que visa definir o tamanho do conteúdo a ser copiado para a variável "buffer", impedindo, assim, que seja possível copiar um *input* maior do que o suposto e, consequentemente, de escrever por cima de memória. Para injetar uma vulnerabilidade neste código poderíamos recorrer à função *strcpy*, visto que a mesma possui a mesma funcionalidade que *strncpy*, mas sem ser possível limitar a quantidade caracteres.

```
char buffer[10];
strcpy(buffer, "biiiiiiiig string");
```

Desta forma a memória adjacente poderá ser reescrita caso o *input* a ser copiado seja maior que o tamanho da variável "buffer".

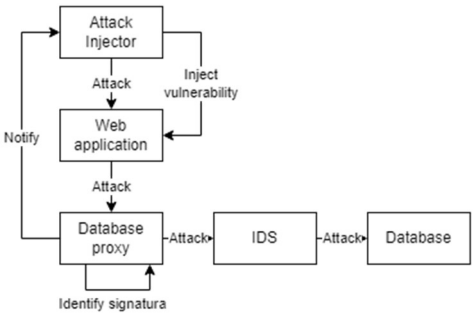
2) Mecanismos de segurança

a) IDS/IPS

Os *IDSs* e *IPSs* constituem componentes fundamentais no estabelecimento de uma estratégia de segurança, no entanto, apenas são eficazes se forem configurados corretamente e se detetarem ou responderem, no caso dos *IPSs*, com precisão a potenciais ataques. Ora, a técnica de "injeção de vulnerabilidades e ataques" torna-se essencial neste aspeto, visto testar o comportamento destes mecanismos simulando ataques reais.

Fonseca *et al.* desenvolveram um cenário onde aplicam, precisamente, esta estratégia no âmbito de um *IDS* baseado em rede que, por sua vez, monitora o tráfego entre uma aplicação *web* e uma base de dados na tentativa de detetar ataques *SQL*. Para o estudo foi ainda implementado um injetor de vulnerabilidades, cuja função é procurar por possíveis pontos do código

fonte da aplicação *web* onde possam ser injetadas vulnerabilidades, e um injetor de ataques que explora, posteriormente, as mesmas. Após a injeção de uma vulnerabilidade, o injetor de ataques executa um ataque com uma determinada assinatura, tendo como alvo a base de dados. Procede-se ainda à utilização de uma base de dados *proxy* entre a base de dados alvo e a aplicação com vista a monitorar o tráfego e, consequentemente, a identificar a assinatura de um determinado ataque. A *proxy*, ao detetar uma assinatura, notifica o injetor de ataques que o ataque alcançou a base de dados alvo. Por fim, é comparado o *output* do *IDS* com a informação que o injetor do ataque possui acerca do mesmo, no sentido de quantificar a precisão com que este foi detetado. [28]



b) Defense-in-Depth

Devido à enorme complexidade deste mecanismo, torna-se necessário compreender as camadas pelo qual o mesmo é constituído, já que cada uma destas representa um controlo de segurança diferente. Este conhecimento irá facilitar as próximas etapas de teste. Antes demais, é fundamental avaliar qual a melhor estratégia a aplicar para testar a técnica *defense-in-depth*. Utilizando o conhecimento adquirido na etapa anterior, é avaliado em que pontos poderão ser injetadas vulnerabilidades e, consequentemente, os ataques a serem realizados.[29] Este procedimento, como já referido, pode tanto ser efetuado manualmente como utilizando ferramentas que automatizam esta etapa. Além do tipo de vulnerabilidades/ataques é preciso ainda definir as camadas a testar, já que o objetivo da aplicação de "injeção de vulnerabilidades e ataques", neste caso, é testar cada um dos controlos de segurança implementados.

Após esta primeira fase onde foram obtidos dados importantes sobre o mecanismo de *defense-in-depth* e de ter sido efetuado um planeamento em torno do mesmo, deve-se proceder à fase de testes das diversas camadas. Para tal, são injetadas vulnerabilidades em certas camadas de forma a comprometer as mesmas e, posteriormente, são executados ataques com vista a avaliar o comportamento dos restantes controlos de segurança perante os mesmos. A junção de 2 ou vários cenários de ataque pode ser realizada de modo a ser possível testar mais camadas.[16]

Por fim, deve-se proceder a uma análise minuciosa dos resultados com o objetivo de documentar possíveis falhas por parte dos controlos de segurança. Isto inclui a identificação de vulnerabilidades que tenham sido *exploited* com sucesso ou até mesmo falhas na deteção/mitigação dos ataques por parte da *defense-in-depth*. Esta análise deve focar-se não só em aspetos técnicos, como também no impacto que os ataques injetados causaram no sistema. Com base na mesma, são ainda efetuadas recomendações priorizadas consoante o respetivo nível de risco, de modo a melhorar o mecanismo de *defense-in-depth*. Estas orientações podem aconselhar a aumentar o número de camadas, melhorar as existentes ou até mesmo proceder à sua substituição

VII. Conclusão

Ao longo deste artigo foi possível compreender a importância de conceitos como a qualidade e confiabilidade no contexto de desenvolvimento de *software*. Ambos demonstraram ser vagos tanto nas suas definições como implementações, levando a que fosse necessária uma melhor explicitação dos mesmos. De modo a introduzir os diversos atributos de qualidade de *software* foi estabelecida a noção de modelo de qualidade. A utilização de um *standard* como referência, neste caso o ISO 25010, permite instituir algum consenso dos atributos que se encontram a ser considerados. Ao nível da confiabilidade destacam-se não só as ameaças que possam pôr em causa o funcionamento de um *software* em conformidade com o esperado, como também os diferentes meios para a obtenção do mesmo.

De entre os meios observados, a tolerância a falhas e as suas técnicas surgiram como fator principal para grande parte dos *softwares* nos dias de hoje continuarem a apresentar um comportamento aceitável na presença de falhas/erros. Baseado nesta ideia, surge o conceito de tolerância a intrusões, onde se procura obter um funcionamento adequado na presença de comportamentos maliciosos, o que levou à introdução de técnicas de "injeção de vulnerabilidades e ataques". Finalmente, é de frisar que estas técnicas assumem alguns desafios aos quais é importante atender: como identificar os modelos de injeção de vulnerabilidades?; (o quê, onde, quando, como, …); como gerar ataques eficazes e realistas?; como automatizar este processo de modo a que o mesmo seja escalável?

Referências

[1] "ISO 25000 Portal," ISO/IEC 25010. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[2] Fitzpatrick, Ronan, "Software quality: definitions and strategic issues" (1996). Reports. Paper 1. Available: http://arrow.dit.ie/scschcomrep/1

[3] D. Vavilkin, “What is software quality? let's define and measure it!”, UTOR, 23-Sep-2021. [Online]. Available: https://u-tor.com/topic/software-quality-defined-and-measure.
[4] Madeira, Henrique, Slides da disciplina Qualidade e Confiabilidade de Software.
[5] Berander, Patrik; Damm, Lars-Ola E., *et al*.; Software quality attributes and trade-offs; Jun – 2005. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=57d7ef7a35d480e2ebd41f66ece451c4d7a7a40a
[6] Kaur, S.; Software Quality; July-2012; Available at: https://www.researchgate.net/publication/303805322 _Software_Quality
[7] Codacy; ISO/IEC 25010 Software Quality Model; 17-Mar-2021; Available at: https://blog.codacy.com/iso-25010-software-quality-model/.
[8] Mohammadi, N.G.; Paulus, S., *et al*.; An Analysis of Software Quality Attributes and their Contribution to Trustworthiness; May-2013; Available at: https://www.researchgate.net/publication/256635247_An_Analysis_of_Software_Quality_Attributes_and_their_Contribution_to_Trustworthiness.
[9] Avizienis, A.; Laprie, J.-claude *et al.* Fundamental Concepts of Dependability; Sept-2001; Available at: https://www.researchgate.net/publication/2408079_Fundamental_Concepts_of_Dependability.
[10] Dependable System; ScienceDirect. Available at: https://www.sciencedirect.com/topics/ computer-science/dependable-system.
[11] Maintainability; ScienceDirect. Available at: https://www.sciencedirect.com/topics/engineering/maintainability.
[12] Aviz˘ienis, A.; Laprie, J.-C. *et al*; Fundamental Concepts of Dependability. Available at: https://people.cs.rutgers.edu/~rmartin/teaching/spring03/cs553/readings/avizienis00.pdf.
[13] Heddaya, A.; Helal, A.; Reliability, Availability, Dependability and Performability: A User-centered View 4-Dez-1996; https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1b1127bc1da736efb33cb4579f5e08904c37a2fa
[14] Pfleeger, C.P.; Pfleeger, S.L. *et al;* Security in Computing. Prentice Hall; Jan-2015; Available at: https://www.eopcw.com/assets/stores/Computer%20Security/lecturenote_1704978481security-in-computing-5-e.pdf.
[16] Nuno Antunes, Slides de Avaliação e Gestão de Cibersegurança.
[17] Avizienis, A. ; Laprie, J.-C et al; Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Computer Society; 2004; Available at: https://www.nasa.gov/pdf/636745main_ day_3 - algirdas_avizienis.pdf.
[18] B.Dhanalaxmi; Dr.G.Apparao Naidu *et al.;* A Review on Software Fault Detection and Prevention Mechanism in Software Development Activities. IOSR Journal of Computer Engineering (IOSR-JCE).; Dez-2015; Available at: https://mail.iosrjournals.org/iosr-jce/papers/Vol17-issue3/Version-2/K017326168.pdf.
[19] Laprie, Jean-Claude; From Dependability to Resilience; Available: https://users.ece.cmu.edu/~koopman/dsn08/fastabs/dsn08fastabs_laprie.pdf
[20] Hameed, O.A.A., Resen, I.A. and Hussain, S.A.A.A. *et al;* Software Fault Tolerance: A Theoretical Overview; Jun-2019; Available at: https://www.researchgate.net/publication/333720230_Software_Fault_Tolerance_A_Theoretical_Overview.
[21] Alam, M.J.; Analysis of Different Software Fault Tolerance Techniques; 2009; Available at: https://www.researchgate.net/publication/328346045_Analysis_of_Different_Software_Fault_Tolerance_Techniques.
[22] Torres-Pomales, W. and Center, L.R; Software Fault Tolerance: A Tutorial. National Aeronautics and Space Administration; 2000; Available at: https://ntrs.nasa.gov/api/citations/20000120144/downloads/20000120144.pdf.
[23] RANDELL, B.R.I.A.N. and XU, J.I.E.; The Evolution of the Recovery Block Concept.; 4-Jun-1997; Available at: http://history.cs.ncl.ac.uk/anniversaries/40th/webbook/dependability/recblocks/rec_blocks.html.
[24] Fonseca, J., Vieira, M. and Madeira, H.; Vulnerability & Attack Injection for Web Applications. IEEE.; 2009; Available at: http://bdigital.ipg.pt/dspace/bitstream/10314/3529/1/Fonseca_DSN09_CR_Final%20IEEE.pdf.
[25] Bace1 R. and Mell P.; Intrusion Detection Systems; NIST Special Publication on Intrusion Detection System; http://cs.uccs.edu/~cchow/pub/ids/NISTsp800-31.pdf
[26] CCNA; Firewalls, IDS, and IPS Explanation and Comparison; https://study-ccna.com/firewalls-ids-ips-explanation-comparison/
[27] Fonseca J.; Vieira M. and Madeira H.; Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection; IEEE Computer Society; 2014; https://core.ac.uk/download/pdf/148389372.pdf
[28] Milenkosk A. and Vieira M.; Evaluating Computer Intrusion Detection Systems: A Survey of Common Practices; Sept-2015; https://www.researchgate.net/publication/282527251_Evaluating_Computer_Intrusion_Detection_Systems_A_Survey_of_Common_Practices
[29] Vieira, M.; Securing Web Applications: Techniques and Challenges; ENIGMA — Brazilian Journal of Information Security and Cryptography, Vol. 1, No. 1; Sep. 2014; https://enigma.unb.br/index.php/enigma/article/view/21/14