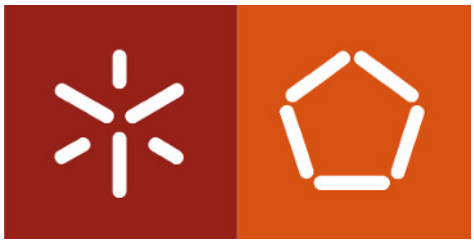

Mestrado em Engenharia Informática
Universidade do Minho

Engenharia de Segurança

Projeto 1 - OWASP Proactive Controls

João Miranda - PG41845
Sandro Cruz - PG41906

23 de Março de 2020

Conteúdo

Capítulo 1

Introdução

Neste primeiro projeto da unidade curricular de Engenharia de Segurança iremos desenvolver o tópico dos controlos pro-activos fornecidos pela OWASP (Open Web Application Security Project).
Com este trabalho pretendemos que seja um resumo informativos dos diferentes controlos pro-activos aconselhados pela OWASP, de forma a que possam servir como um guia em como desenvolver software que seja seguro para o utilizador e para o software. O software desenvolvido dera ser também de fácil manutenção e monitorização para os gestor do software, o que este projecto também ira abordar.

Capítulo 2

O que é a OWASP?

A **OWASP (Open Web Application Security Project)** ou **Projeto Aberto de Segurança em Aplicações Web**, é uma comunidade online que cria e disponibiliza de forma gratuita artigos, metodologias, documentação, ferramentas e tecnologias no campo da segurança de aplicações web.

Todas as ferramentas, documentos, fóruns e capítulos do OWASP são grátis e abertos a todos os interessados em aperfeiçoar a segurança em aplicações. Promovemos a abordagem da segurança em aplicações como um problema de pessoas, processos e tecnologia, porque as abordagens mais eficazes em segurança de aplicações requerem melhorias nestas áreas. A OWASP é um novo tipo de organização. O fato de ser livre de pressões comerciais permite fornecer informação de segurança de aplicações imparcial, prática e de custo eficiente.

A OWASP não é filiada a nenhuma empresa de tecnologia, apesar de apoiar o uso de tecnologia de segurança comercial. Da mesma forma que muitos projetos de software de código aberto, a OWASP produz vários tipos de materiais de maneira colaborativa e aberta.[2]

Capítulo 3

Controlos Pro-Activos OWASP

Os controlos pro-activos da OWASP descrevem os diferentes tipos de controlos que todos os programas desenvolvidos devem seguir.
Os programadores são base de qualquer *software*. De forma a que um programador consiga desenvolver software seguro ele deve ser apoiado pela organização que trabalham, dada a complexidade que existem em desenvolver software seguro. Um programador necessita sempre de desenvolver software em que a segurança é sempre a prioridade, mas muitas vezes os programadores estão destinados a falhar, pois muitos programadores não aprendem a programar código seguro ou/e não aprendem criptografia, muitas das linguagens e *frameworks* utilizadas no desenvolvimento de software não tem controlos de segurança ou são inseguras por defeito. Também é raro que as organizações disponibilizarem os recursos necessários para os programadores terem a perspectiva de como desenvolver software seguro, e mesmo quando o fazem podem haver falhas no design e nos requisitos que podem levar ao software não ser seguro.[1]
É por estes factores que a OWASP decidiu criar estes controlos pro-activos, de forma a que um programador possa ter acesso a estes recursos, para que consiga desenvolver software seguro.
Controlos pro-activos definidos pela OWASP

- Definir os Requisitos de Segurança;
- Utilização de *Frameworks* e Bibliotecas de Segurança;
- Acesso seguro a Bases de Dados;
- Codificar e Formatar Dados;
- Validar todos os inputs;
- Implementar Identidade Digital;

- Implementar Controlos de Acesso;
- Proteger Todos os Dados;
- Monitorizar e Implementar *Logs* de Segurança;
- Manipulação de Erros e Excepções.

3.1 Definir os Requisitos de Segurança

3.1.1 Descrição

Um requisito de segurança é uma função necessária que o software tem que satisfazer uma ou varias propriedades para que um software seja considerado seguro. Os requisitos de segurança derivam de *standards* da industria, leis e de vulnerabilidades antigas. Os requisitos de segurança definem novas *features* ou modificações de *features* existentes de forma a resolver um problema de segurança ou eliminar uma potencial vulnerabilidade.
Os requisitos de segurança ajudam a criar uma base composta por diferentes funcionalidades de segurança para uma aplicação. Em vez de se ter de criar uma abordagem diferentes para a segurança de cada *software* desenvolvido, os requisitos de segurança padrão permitem que um programador reutilize os melhores protocolos e melhores práticas.

3.1.2 OWASP Application Security Verification Standard (ASVS)

O OWASP ASVS (Padrão de Verificação de Segurança de Aplicativos OWASP) é um catálogo de requisitos de segurança e de critérios de verificação disponíveis, podendo ser uma fonte de requisitos de segurança detalhados para equipas de desenvolvimento.
Os requisitos de segurança são catalogados em diferentes *buckets* com base numa função de segurança compartilhada de ordem superior. O ASVS contém categorias como autenticação, contolo de acesso, tratamento (registo de erros) e serviços de *web*. Cada categoria contém uma coleção de requisitos que representam as melhores práticas para essa categoria, elaboradas como declarações verificáveis.

3.1.3 Aumento dos requisitos com histórias de utilizadores e casos de uso indevidos

Os requisitos ASVS são declarações verificáveis básicas que podem ser expandidas com histórias de utilizadores e casos de uso indevidos. A vantagem de uma história de utilizador ou um caso de uso indevido é que vincula a aplicação exatamente ao que o utilizador ou atacante faz ao sistema.

Requisitos com histórias de utilizadores

Uma história de utilizador é um requisito expresso da perspetiva de uma meta do utilizador final. O formato da história transformou-se na maneira mais popular de expressar requisitos devido aos seguintes motivos:

- Foca no ponto de vista de uma função que irá usar ou ser impactada pela solução
- Define o requisito na linguagem que tem significado para essa função
- Ajuda a esclarecer o verdadeiro motivo do requisito
- Ajuda a definir requisitos de alto nível sem ser necessário entrar em detalhes

Casos de uso indevidos

Um caso de uso indevido é considerado um caso de uso inside-out, sendo assim um recurso que não deve ser implementável num sistema. Oferece outro ponto de vista do sistema para o gerenciamento de requisitos de segurança. Acima de tudo, esses requisitos são muito significativos de acordo com o modelo Kano de satisfação das partes interessadas. As vantagens dos casos de uso indevidos são considerados:

- Aumento da qualidade pela observação de requisitos não funcionais
- Os desenvolvedores e os clientes podem perceber melhor o sistema
- O modelo é de fácil compreensão e utilização
- As medidas são visíveis pois os riscos e as contramedidas são visualizados
- A análise de risco é possível desdo a iniciação
- Os riscos podem ser reconhecidos pelo cliente
- A rastreabilidade é garantida pois é necessária uma revisão para o aumento da segurança dos recursos

Exemplo - A3:2017-Exposição de dados confidenciais

História de utilizador: Em vez de atacar diretamente a criptografia, os atacantes roubam chaves, executam ataques *man-in-the-middle* ou roubam dados do texto limpo do servidor durante o transporte ou do cliente (navegador *internet*). Geralmente, é necessário um ataque de forma manual. As bases de dados de senhas podem ser recuperadas na utilização da força bruta pelas GPUs (Graphics Processing Units).

Caso de uso indevido: Como atacante, roubo as chaves que foram expostas na aplicação para obter acesso não autorizado à aplicação do sistema.

3.1.4 Implementação

A correta utilização dos requisitos de segurança envolve quatro passos: descoberta/seleção, investigação/documentação, implementação e confirmação da implementação (teste) correta de novos recursos e funcionalidades de segurança numa aplicação. Os passos vão ser explicados posteriormente.

Descoberta/Seleção

O processo é iniciado com a descoberta e seleção de requisitos de segurança. Nesta fase, o desenvolvedor está a entender os requisitos de segurança de uma fonte padrão (p.e. ASVS) e, escolhendo os requisitos a incluir numa determinada liberação de uma aplicação. O ponto de descoberta e seleção é a escolha de um número gerenciável de requisitos de segurança para esta versão e a continuação da iteração para cada versão sendo cada mais adicionadas determinadas funcionalidades de segurança.

Investigação/Documentação

Durante a investigação e a documentação, o desenvolvedor analise a aplicação existente em relação ao conjunto novo de requisitos de segurança para determinar se a aplicação cumpre o requisito ou se é necessário algum desenvolvimento. Esta investigação culmina na documentação dos resultados da revisão.

Implementação/Confirmação

Após a necessidade de desenvolvimento ser determinada, o desenvolvedor deve modificar a aplicação de forma a adicionar uma nova funcionalidade ou remover uma opção insegura. Nesta fase, o desenvolvedor determina primeiro o design necessário para cumprir o requisito e, de seguida, conclui as alterações no código para cumprir o requisito. Os casos de teste (confirmação) devem ser criados de forma a confirmar a existência da nova funcionalidade ou a refutação da existência de uma opção anteriormente insegura.

3.1.5 Prevenção de vulnerabilidades

Os requisitos de segurança definem a funcionalidade de segurança de uma aplicação. Uma melhor segurança incorporada desde o início de um ciclo de vida de aplicações resulta na prevenção de muitos tipos de vulnerabilidades.

3.2 Utilização de *Frameworks* e Bibliotecas de Segurança

3.2.1 Descrição

As bibliotecas de codificação seguras e as estruturas de *software* com segurança incorporada ajudam os desenvolvedores de *software* na proteção contra falhas de

projeto e implementação relacionadas à segurança. Um desenvolvedor que desenvolve uma aplicação do zero pode não ter conhecimento, tempo ou orçamento suficientes para uma implementação ou manutenção adequadas dos recursos de segurança. A utilização de estruturas de segurança ajuda o alcance de objetivos de segurança com maior eficiência e precisão.

3.2.2 Práticas recomendadas de implementação

Na incorporação de bibliotecas ou estruturas de terceiros no *software* do desenvolvedor, é importante considerar as práticas recomendadas seguintes:

- Utilizar bibliotecas e estruturas de fontes confiáveis, ativamente mantidas e amplamente utilizadas por muitas aplicações
- Criar e manter um catálogo de inventário de todas as bibliotecas de terceiros
- Manter proativamente as bibliotecas e as componentes atualizadas
- Utilizar uma ferramenta (p.e. OWASP Dependency Check e Retire.js) para identificar dependências do projeto e verificar se existem vulnerabilidades conhecidas e divulgadas publicamente para todos os códigos de terceiros
- Reduzir a superfície de ataque encapsulando a biblioteca
- Dispor apenas o comportamento necessário ao *software*

3.2.3 Prevenção de vulnerabilidades

As estruturas e bibliotecas seguras podem ajudar a impedir uma ampla variedade de vulnerabilidades de aplicações *web*. É essencial a manutenção/atualização dessas estruturas e bibliotecas utilizando componentes com vulnerabilidades conhecidas tais como: injeção, quebra de autenticação, exposição de dados confidenciais, entidades externas de XML (XXE), controlo de acesso quebrado, configuração incorreta de segurança, cross-site scripting XSS, deserialização insegura, monitorização e *logging* insuficientes.

3.3 Acesso seguro a Bases de Dados

3.3.1 Descrição

Consiste na indicação e recomendação de áreas para o acesso seguro a todos os repositórios de dados, incluindo bases de dados relacionais e NoSQL. Estas áreas vão ser descritas de seguida.

3.3.2 Áreas

Segurança das queries

A injecção de SQL ocorre quando a entrada não confiável do utilizador é adicionada dinamicamente numa query em SQL de forma insegura, geralmente através da concatenação básica de *strings* tornando-se assim num dos riscos de segurança de aplicações mais perigosos. A sua exploração é facilitada e pode levar ao roubo, limpeza ou modificação de toda a base de dados. A aplicação pode até ser usada para executar comandos perigosos no sistema operativo que hospeda a base de dados, fornecendo assim ao atacante uma vantagem na sua rede.
Para a mitigação da injeção de SQL, a entrada não confiável deve ser impedida de ser interpretada como parte de um comando SQL. A melhor resolução possível é utilizando a técnica de programação conhecida como *Parameterização da Query*. Essa defesa deve ser aplicada ao SQL bem como à construção dos procedimentos armazenados.

Precaução na parameterização da query

Certas localizações numa query de uma base de dados não são parametrizáveis, sendo diferentes para cada fornecedor da base de dados. Deve ser realizada uma validação de correspondência exata com muita precaução ou uma fuga manual ao confrontar parâmetros de uma query de uma base de dados que não podem ser vinculados a uma consulta parametrizada. Além disso, embora o uso de queries parameterizadas tenha um impacto positivo no desempenho, algumas destas em implementações específicas de bases de dados afetarão negativamente o desempenho. Deve-se assim, testar o desempenho das queries, feito com recursos abrangentes de cláusula ou pesquisa de texto.

Configuração segura

Os sistemas de gerenciamento de base de dados (SGBDs) nem sempre são enviados numa configuração segura por padrão. Deve ser tomada precaução para garantir que os controlos de segurança disponíveis no SGBD e na plataforma de hospedagem estejam ativados e configurados corretamente. Existem padrões, guias e referências disponíveis para os SGBDs mais comuns.

Autenticação segura

Todo o acesso à base de dados deve ser devidamente autenticado. A autenticação no SGBD deve ser realizada de maneira mais segura possível, ocorrendo apenas num canal seguro e onde as credenciais devam estar estar devidamente protegidas e disponíveis para utilização.

Comunicação segura

A maioria dos SGBDs suporta uma variedade de métodos de comunicação (p.e. serviços, APIs) seguros que são autenticados e cifrados e inseguros que são o dual do anterior. Uma boa prática seria utilizar apenas as opções de comunicação segura.

3.3.3 Prevenção de vulnerabilidades

A adoção destas áreas permitiu a prevenção de injeção de SQL bem como uma barreira contra os riscos de aplicações móveis (Controlos fracos por parte do servidor).

3.4 Codificar e Formatar Dados

3.4.1 Descrição

Codificar e formatar os dados são técnicas utilizadas de forma a proteger o *software* contra ataques de injecção.
Codificar envolve a translação de um carácter especial em algo diferente ou equivalente que já não é perigoso para o intérprete, como por exemplo transformar o símbolo em &lt. Formatação de dados envolve adicionar caracteres especiais antes de caracteres/*strings* para evitar que seja mal interpretado, por exemplo adicionar antes de "para que seja interpretado como texto e não como fecho de uma *string*.
A codificação de saída é melhor aplicada antes do conteúdo ser transmitido para o interpretador de destino. Caso essa defesa seja realizada antecipadamente, no processamento de uma solicitação, a codificação ou formatação poderá interferir no uso do conteúdo noutras partes do programa. Um dos exemplos aplicados é na utilização do HTML pois, caso o conteúdo seja formatado antes de ser armazenado nas bases de dados e se o UI formatar automaticamente esse conteúdo, numa segunda vez o conteúdo não será disposto corretamente devido a esta formatação dupla.

3.4.2 Codificação de saída contextual

A codificação de saída contextual é uma técnica muito importante de programação de segurança necessária para interromper o XSS. Essa defesa é realizada na saída, quando é criada uma interface com o utilizador, no último momento antes que os dados não confiáveis sejam adicionados ao HTML. O tipo de codificação depende da localização no documento em que os dados estão a ser transmitidos ou armazenados. Sendo assim, os tipos de codificação essenciais para criar interfaces de utilizador seguras incluem as codificações de entidade HTML, atributo HTML, *Javascript* e do URL.

3.4.3 Exemplos de codificação

OWASP Java Encoder

O OWASP Java Encoder é uma classe de codificador de alto desempenho *drop-in* de simples utilização e sem dependências. Este projeto ajuda os desenvolvedores da *Java Web* na defesa do *Cross Site Scripting* (XSS). Os ataques XSS são um tipo de injeção, na qual *scripts* malaciosos, normalmente feitos em *Javascript*, são injetados em *sites* confiáveis. Uma das principais defesas para interromper este *script* entre *sites* é precisamente a utilização da técnica de codificação de saída contextual.

.NET

A partir do .NET 4.5, a biblioteca de *scripts* contra o XSS faz parte da *framework*, apesar de não ser ativada por padrão. Pode-se especificar o uso do *AntiXSSEncoder* nesta biblioteca como o codificador padrão para todas as aplicações utilizarem as configurações do web.conf. Quando aplicado, é importante codificar contextualmente a sua saída, significando a utilização da função correta da biblioteca *AntiXSSEncoder* no local apropriado dos dados no documento.

PHP

O *framework Zend* fornece a componente *zend-escaper* para gerir a complexidade do problema, expondo as funcionalidades de formatação HTML, atributos de HTML, *Javascript*, CSS e URL's de forma a garantir que sejam seguras para o *browser*.

3.4.4 Outros tipos de de codificação e defesa da injeção

A codificação pode ser utilizada para neutralizar conteúdo contra outras formas de injeção. É possível a "Formatação de comando do Sistema Operativo"que consiste na neutralização de certos meta-caracteres especiais ao adicionar uma entrada num comando do sistema operativo. Esta defesa pode ser utilizada para interromper as vulnerabilidades da injeção de comandos.
Existem outras formas de formatação que podem ser usadas na interrupção da injeção como a formatação do atributo XML que trava as várias formas de injeção XML e de injeção de uma diretoria XML.

3.4.5 Codificação e canonização de caracteres

A codificação *Unicode* é um método para armazenar caracteres com vários bytes. Sempre que os dados de entrada são permitidos, podem ser inseridos usando o *Unicode* para mascarar códigos maliciosos e permitir uma variedade de ataques. Por outro lado, a canonização é um método no qual os sistemas convertem dados num formato padrão. As aplicações *web* geralmente utilização a canonização de

caracteres para garantir que todo o conteúdo seja do mesmo tipo quando armazenado ou exibido.
A segurança contra ataques relacionados à canonização implica que uma aplicação deva ser segura quando *Unicode* e outras representações de caracteres mal formados forem inseridas.

3.4.6 Prevenção de vulnerabildades

Com a introdução da codificação e formatação de dados foi possível prevenir a injeção de forma geral (incluindo a injeção por parte do cliente) e o XSS.

3.5 Validar todos os *inputs*

3.5.1 Descrição

A validação de *inputs* é uma técnica de programação que garante que apenas os dados formatados corretamente possam entrar numa componente do sistema do *software*.

3.5.2 Sintaxe e validade semântica

Uma aplicação deve verificar se os dados são válidos de forma sintática e semântica antes da sua utilização.
A validade da sintaxe significa que os dados estão no formato esperado. Por exemplo, uma aplicação pode permitir que um utilizador selecione um ID de conta de quatro dígitos para executar algum tipo de operação. A aplicação deve assumir que o utilizador está a inserir uma carga útil de injeção SQL e verificar se os dados inseridos têm exatamente quatro dígitos e se são compostos apenas por números.
A validade semântica inclui aceitar apenas *inputs* que estejam dentro de um intervalo aceitável para a funcionalidade e o contexto da aplicação. Por exemplo, uma data de início deve ser anterior a uma data de término na escolha de períodos.

3.5.3 *Whitelisting e Blacklisting*

A *Blacklisting* tenta verificar se os dados fornecidos não possuem conteúdo denominado de "mau". Um exemplo seria de uma aplicação *web* poder bloquear o *input* que contém um texto (p.e, <SCRIPT> para ajudar a impedir o XSS). No entanto, essa defesa pode ser evitada com uma *tag* de *script* em letras somente minúsculas ou maiúsculas e minúsculas.
Por outro lado, a *Whitelisting* tenta verificar se um determinado dado corresponde a um conjunto de regras denominado de "bens conhecidos". Como exemplo, poderia ser utilizada uma regra de validação da lista de permissões para um estado dos EUA sendo esta um código de duas letras que é apenas um dos estados válidos dos EUA.

Na criação de *software* seguro, a lista de permissões é a abordagem mínima recomendada. A *Blacklisting* é suscetível a erros e pode ser contornada com várias técnicas de evasão sendo perigosa quando depende de si mesma. Embora a *Blacklisting* possa ser evitada, também pode ser útil para ajudar a detetar ataques óbvios. Portanto, enquanto que a *Whitelisting* ajuda a limitar a superfície de ataque, garantindo que os dados tenham a validade sintática e semântica correta, a *Blacklisting* deteta e impede ataques potenciais óbvios.

3.5.4 Validação do Cliente/Servidor

A validação de *input* deve ser sempre feita no Servidor para segurança. Embora a validação no Cliente possa ser útil para fins de segurança e funcionais, muitas vezes pode ser facilmente ignorada. Por exemplo, a validação do *Javascript* pode alertar o utilizador de que um campo específico deve consistir em números, mas o Servidor da aplicação deve validar que os dados enviados consistem apenas em números no intervalo numérico apropiado para esse recurso.

3.5.5 Expressões regulares

As expressões regulares fornecem uma forma de verificação da correspondência dos dados a um padrão específico. No entanto, existem cuidados a ter em conta:

- **Potencialização do *Denial-Of-Service*:** deve-se ter cuidado ao criar expressões regulares pois, caso seja criadas expressões não apropriadas podem resultar em possíveis condições de DoS. No entanto, existem ferramentas que podem testar a vulnerabilidade ao DoS da expressão regular.
- **Complexidade:** as expressões regulares podem ser complicadas na manutenção ou compreensão dos desenvolvedores. No entanto, existem alternativas de validação que envolvem escrever métodos de validação que podem ser mais fáceis na manutenção.

3.5.6 Limitação da validação de *input*

A validação de *input* nem sempre torna os dados seguros, pois certas formas de *input* complexo podem ser válidas mas ainda perigosas. Um exemplo é a utilização de um endereço eletrónico válido que pode conter um ataque de injeção SQL ou um URL válido que pode conter um ataque de *script* entre *sites*. Devem ser sempre aplicadas defesas adicionais para além da validação de *input* de dados como a parametrização da *query* ou a formatação.

3.5.7 Desafios da validação de dados serializados

Algumas formas de *input* são tão complexas que a validação pode proteger minimamente a aplicação. É perigosa a desserialização de dados não confiáveis ou dados que podem ser manipulados por um atacante. O único padrão arquitetural seguro é a não aceitação de objetos serializados de fontes não confiáveis

ou a desserialização em capacidade limitada apenas para tipos de dados simples. Deve ser evitado a todo o custo o processamento de dados serializados e a utilização e formas mais fáceis de defesa (p.e., JSON). Caso não seja possível, deve-se considerar a lista seguinte de defesas de validação ao processar dados:

- Implementação de verificações de integridade ou utilização da criptografia para os objetos serializados de forma a impedir a criação de objetos perigosos ou o *tampering* de dados.
- Imposição de restrições restritas de tipo durante a desserialização antes da criação do objeto.
- Desvio da técnica do código que está em espera de um conjunto de definição de classes.
- Isolamento do código que desserializa, para que seja executado em ambientes com privilégios muito baixos, como receptáculos temporários.
- Desserialização de exceções e falhas na segurança de *logs*.
- Restrição/Monitorização da conectivadade de receptáculos ou servidores que façam a desserialização, alertanto se um utilizador desserializa constantemente.

3.5.8 *Input* inesperado do utilizador (Atribuição em massa)

Algumas *frameworks* suportam a ligação automática de parâmetros de solicitações HTTP a objetos do lado do Servidor utilizados pela aplicação. Esse recurso pode permitir que um atacante atualize objetos do lado do Servidor que não foram modificados, modifique o seu nível de controlo de acesso e contorne a lógica comercial da aplicação. Esse ataque tem várias denominações tais como *Atribuição em massa*, *Vinculação automática* ou *Injeção de objetos*. Por exemplo, se o objeto do utilizador possuir um privilégio de campo que especifique o nível de privilégio do utilizador na aplicação, o atacante poderá procurar páginas nas quais os dados do utilizador são modificados (privilégio de *admin*). Se a ligação automática estiver ativada de forma insegura, o objeto do servidor que representa o utilizador será modificado de forma comprometedora. No entanto, o problema pode ser resolvido com duas soluções: evitar vincular o *input* diretamente e utilizar DTO's (*Data Transfer Objects*); ativar a ligação automática com uma configuração das regras da lista de permissões para cada página ou recurso para definir que campos têm permissão para serem ligados automaticamente.

3.5.9 Validação e higienização de HTML

Considerando uma aplicação que necessite de aceitar HTML de utilizadores, a validação ou formatação não ajudará na higienização: ss expressões regulares não são expressivas o suficiente para entender a complexidade do HTML; a

codificação ou formatação de HTML fará com que o HTML não seja renderizado corretamente. Para o contorno deste problema, foi desenvolvidas bibliotecas que fazem a análise e limpeza do texto formatado em HTML: a *HtmlSanitizer* e a *OWASP Java HTML Sanitizer*.

3.5.10 Validação da funcionalidade em bibliotecas e *frameworks*

Todas as linguagens e a maioria dos *frameworks* fornecem bibliotecas de validação ou funções que devem ser aproveitadas para validar dados. As bibliotecas de validação geralmente cobrem tipos de dados comuns, requisitos de comprimento, intervalo de números inteiros, entre outros. Muitas bibliotecas e *frameworks* permitem definir a sua própria expressão regular ou lógica para validação personalizada de uma maneira que permita ao programador aproveitar essa funcionalidade em toda a aplicação.

3.5.11 Prevenção de vulnerabilidades

Com a validação de *input* foi possível a prevenção de vulnerabilidades como a redução da superfície de ataque das aplicações dificultando os ataques contra uma aplicação, fornecimento de segurança a certas formas de dados, específica a certos ataques não podendo ser aplicada com segurança como regra geral. No entanto, a validação de *input* não deve ser utilizada como método principal para prevenção de XSS, injeção de SQL, entre outros.

3.6 Implementar identidade digital

3.6.1 Descrição

A identidade digital é a representação exclusiva de um utilizador à medida que este se envolve numa transação *online*. A autenticação é o processo de verificação da veracidade de uma entidade. O controlo de sessões é um processo no qual um Servidor mantém o estado de autenticação dos utilizadores, para que o utilizador possa continuar a utilizar o sistema sem se autenticar novamente.

3.6.2 Níveis de autenticação

Nível 1: *Passwords*

As *passwords* são bastante importantes pois são necessárias políticas, armazená-las com segurança e às vezes é necessária a redefinição destas.

- **Requisitos das *passwords***
  - Devem conter pelo menos oito caracteres se a autenticação é multifator e outros controlos também forem utilizados. Se o multifator não for possível, deve-se aumentar o tamanho dos caracteres para dez.

  - Todos os caracteres ASCII de impressão bem como caracteres de espaço devem ser aceitáveis em segredos memorizados.
  - Uso de senhas longas e frases secretas.
  - Remoção dos requisitos de complexidade, pois estes têm eficácia limitada. Recomenda-se a adoção do multifator ou de senhas com comprimentos mais longos.
  - Verificação da frequência do uso da mesma *password*, isto é, verificar se as *passwords* utilizadas não são usadas frequentemente. Pode-se optar por bloquear as *passwords* mais comuns que atendem aos requisitos de comprimento acima constatados.

- **Implementação de um mecanismo de recuperação de *password* segura**

  É comum que uma aplicação possua um mecanismo para que um utilizador obtenha acesso à sua conta no caso de se esquecer da *password*. Um bom fluxo de trabalho de *design* para um recurso de recuperação de *password* utiliza elementos de autenticação multifator. Por exemplo, pode utilizar uma pergunta de segurança, e gerar um *token* para um dispositivo.

- **Implementação de um armazenamento seguro de *password***

  De forma a fornecer controlos fortes de autenticação, uma aplicação deve armazenar com segurança as credenciais do utilizador. Além disso, devem estar em vigor controlos criptográficos para o caso de uma credencial ser comprometida, o atacante não ter acesso imediato a essas informações.

Nível 2: Autenticação multifator

A autenticação multifator garante que os utilizadores sejam quem alegam ser, exigindo que eles se identifiquem com um combinação de algo que se sabe (*password*), algo que se possui (contacto de telemóvel) e a biometria que caracteriza algo que o utilizador seja (impressão digital). A utilização de *passwords* como um fator único fornece uma segurança fraca. As soluções multifatoriais fornecem uma solução mais robusta, exigindo que o atacante adquira mais do que um elemento para se autenticar no serviço.

Nível 3: Autenticação baseada em criptografia

A autenticação baseada em criptografia é necessária quando o impacto dos sistemas comprometidos pode levar a danos pessoais como perdas financeira significativas, prejudicar o interesse público ou violações civis e criminais. Sendo assim, esta autenticação é baseada na prova de posse de uma chave por meio de um protocolo criptográfico.

- **Gerenciamento de sessões**

Depois da autenticação inicial do utilizador ser bem sucedida, pode ser utilizada uma aplicação para rastrear e manter esse estado de autenticação por um período limitado de tempo. Isso permite que o utilizador continue a utilizar a aplicação sem precisar de manter a autenticação numa nova solicitação. O rastreamento do estado do utilizador é denominado de **Gerenciamento de Sessão**.

- **Geração e expiração da sessão**

  O estado do utilizador é rastreado numa sessão que normalmente é armazenada no Servidor para o gerenciamento tradicional de sessões baseadas em *web*. Portanto, é fornecido um identificador de sessão ao utilizador para que este possa identificar qual sessão do Servidor contém os dados corretos. O Cliente só necessita de manter esse identificador de sessão, o que também mantém os dados confidenciais da sessão do Servidor fora do Cliente. Alguns controlos podem ser implementados para solucionar o gerenciamento de sessões tais como:

  - Verificar se o identificador da sessão é longo, exclusivo e aleatório.
  - A aplicação deve gerar uma nova sessão durante a autenticação e a nova autenticação.
  - A aplicação deve implementar um tempo limite inativo após um períodode inatividade e uma vida útil máxima absoluta para cada sessão, sendo que depois os utilizadores devem recorrer a uma nova autenticação. A duração do tempo limite deve ser inversamente proporcional ao valor dos dados protegidos.

- ***Cookies* do *browser***

  Os *cookies* do *browser* são um método comum para a aplicação *web* armazenar identificadores de sessão para aplicações *web* que implementam técnicas padrão de gerenciamento e sessões. Devem ser consideradas algumas defesas ao usar *cookies* do *browser* sendo estas:

  - Quando os *cookies* do *browser* são utilizados como mecanismo para rastrear a sessão de um utilizador, estes devem estar acessíveis a um conjunto mínimo de domínios e caminhos, sendo marcados para expirar no período de validade da sessão ou logo após.
  - O sinalizador *seguro* deve ser definido para a garantia da transferência feita através de um canal seguro (TLS).
  - O sinalizador *HttpOnly* deve ser definido para impedir que o *cookie* seja acessado via *Javascript*.
  - A adição de atributos *samesite* aos *cookies* impede que alguns *browsers* modernos enviem *cookies* com solicitações entre *sites* e fornece

proteção contra falsificação de solicitações entre *sites* e ataques de vazamento de informações.

- ***Tokens***

  As sessões do Servidor podem ser limitadas para algumas formas de autenticação. Os *serviços sem estado* permitem o gerenciamento de dados da sessão no Cliente para fins de desempenho, para que o Servidor tenha menos carga em armazenar e verificar a sessão do utilizador. Essas aplicações *sem estado* geram um *token* de acesso de curta duração que pode ser utilizado para autenticar uma solicitação do cliente sem enviar as credenciais do utilizador após a autenticação inicial.

- **JWT (JSON *Web Tokens*)**

  O JSON *Web Token* (JWT) é um padrão aberto que define uma maneira compacta e independente de transmitir com segurança informações entre partes como um objeto JSON. Essas informações pode ser verificadas e confiáveis porque são assinadas digitalmente. Um *token* JWT é criado durante a autenticação e é verificado pelo Servidor antes de qualquer processamento. No entanto, os JWT's são normalmente criados e entregues a um Cliente sem serem guardados pelo Servidor de forma alguma. A integridade do *token* é mantida através do uso de assinaturas digitais, para que um Servidor possa posteriormente verificar se o JWT ainda é válido e não foi modificado desde a sua criação.

3.6.3 Prevenção de vulnerabilidades

A identidade digital permitiu a prevenção de vulnerabilidades como a quebra de autenticação, o gerenciamento da sessão e a fraca autorização e autenticação.

3.7 Implementar Controlos de Acesso

3.7.1 Descrição

Controlo de acesso (ou garantir autorização) é o processo de garantir ou negar um pedido de acesso a um utilizador, processo ou programa, o controlo de acesso também envolve o acto de garantir e revogar esses privilégios de acesso. Deve se notar que o acto de autorizar (verificar o acesso a diferentes funcionalidades ou recursos) não é o equivalente de autorização (verificar identidade).
As diferentes aplicações do controlo de acesso tem ramificações em muitas diferentes áreas do software dependendo na complexidade do sistema de controlo de acessos. Por exemplo, a gestão do controlo de acessos a *metadata*. Existem três diferentes tipos designes para a gestão de controlo de acessos que devem ser consideradas:

- **Discretionary Access Control (DAC) / Controlo de Acesso Discricionária (CAD)** - o que significa a restrição de acesso a objectos (exe., ficheiros, etc) baseada na identidade e o tipo do sujeito (exe., utilizador, processo) e/ou os grupos a que o objecto a ser acedido pertencem.
- **Mandatory Access Control (MAC) / Controlo de Acesso Obrigatório (CAO)** - é a restrição de acesso baseada na sensibilidade (,que é representada por um identificador) da informação contida no sistema de recursos e a autorização formal dos utilizadores para aceder a informação sensível.
- **Role Based Access Control (RBAC) / Controlo de Acesso Baseado em Funções (CABF)** - é um modelo em que o controlo de acessos aos recursos, em que as acções permitidas aos recursos são identificadas com as funções em vez de um identificador individual.
- **Attribute Based Access Control (ABAC) / Controlo de Acesso Baseado em Atributos** - garante ou nega pedidos do utilizador baseado-se em atributos arbitrários do utilizador, atributos arbitrários do objecto e condições ambientais que podem ser reconhecidas globalmente e ainda mais relevante as policias aplicadas.

3.7.2 Princípios de Design de Controlo de Acesso

Os seguintes requisitos de design do sistema de controlo de acessos devem ser considerados na fase inicial do desenvolvimento de software.

Design Access Control Thoroughly Up Front / Idealizar o Controlo de Acesso em Primeiro

Depois de ser escolhido o tipo de controlo de acesso é muitas vezes difícil alterar para outro tipo de controlo de acesso, dado que exige alterar o controlo de acesso existente na aplicação com um novo tipo. O controlo de acesso é uma das áreas principais no que toca a aplicação de software seguro, e é necessário ser desenhado e idealizado com antecipação.
A idealização do controlo de acesso pode começar como algo simples, mas rapidamente pode evoluir para algo complexo e com medidas pesadas de segurança de controlo. Quando se faz a avaliação das capacidades do controlo de acessos em *frameworks*, é necessário certificar que os controlos de acesso alinham-se com os requisitos do controlo de acessos definido para o software.

3.7.3 Forçar Todos os Pedidos a Passarem por Verificações de Controlo de Acesso

Certificar que todos os pedidos passam por uma camada de verificação do controlo de acesso. Tecnologias como os filtros do Java ou outros mecanismos de processo de pedidos automáticos são o ideal, dado que vão certificar que todos os pedidos vão passar por uma espécie de verificação dos controlos de acesso

Negar o Acesso por Predefinição

Negar o Acesso por Predefinição é o principio de que se um pedido não for especificamente permitido, então é negado. Há muitas formas de introduzir esta regra no código da aplicação, alguns exemplos de aplicação:

- O código da aplicação dispara um erro ou excepção em que processa um pedido de controlo de acesso. Nestes casos o controlo de acesso deve ser sempre negado.
- Quando o administrador cria um novo utilizador ou o utilizador regista a sua conta, essa conta deve ter o menor ou nenhumas permissões de acesso por defeito até aos acessos serem configurados.
- Quando uma no nova função é adicionada a aplicação todos os utilizadores devem ter o acesso negado a essa nova função até estar propriamente configurados os acesso a ela.

Principio do Menos Privilegiado

Certificar que todos os utilizadores, programas e processos são tem o mínimo possível de acessos possíveis.

As Funções de um Utilizador não Devem ser *Hardcoded*

Muitas *frameworks* utilizam por predefinição um controlo de acesso que é baseado em funções do utilizador. É comum encontrar aplicações com verificações desta natureza:

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
    deleteAccount();
}
```

Mas é necessário ter precaução quando se implementa este tipo de programação, dado que pode ter as seguintes limitações ou riscos:

- Programar em função das funções dos utilizadores é frágil, dado que é fácil criar código incorrecto ou faltar uma verificação no código.
- Programar em função das funções dos utilizadores não permite *multi-tenancy*.
- Programar em função das funções dos utilizadores não permite a regras de controlo de acesso especificas a dados ou horizontais.
- Bases de dados com muitos controlos de acessos são difíceis de fazer auditoria ou verificar se a politica de controlo de acesso esta a ser aplicada a 100%.

Em vez de se utilizar esta metodologia deve-se utilizar a seguinte:

```
if (user.hasAccess("DELETE_ACCOUNT")) {
    deleteAccount();
}
```

Programação baseada nos atributos ou recursos de controlo de acesso são o ponto de partida para a criação de um sistema de controlo de acessos bem desenhado e funcional. Este tipo de programação também permite uma grande capacidade de customizáção no que toca aos controlos de acessos.

Registar Todos os Eventos Relativos ao Controlo de Acessos

Todos os controlos de acesso falhados devem ser registados, dado que servem como indicativo de um utilizador com intenções maliciosas estar a tentar explorar as vulnerabilidades do sistema.

3.7.4 Prevenção de vulnerabilidades

Os controlos de acesso permitem impor politicas nos utilizadores que lhes proíbem de aceder a funções fora das suas permissões, estas medidas permitem impedir:

- Que os utilizadores tentem "passar por cima"de controlos de acesso modificando o URL, modificar a pagina HTML ou o estado interno da aplicação, ou simplesmente utilizar uma API para atacar os controlos de acesso.
- Permitir que uma chave primaria seja modificada para a de um outro utilizador, permitindo assim ao atacante utilizar a conta desse utilizador.
- Elevação de privilégios, o que permite ao atacante utilizar a conta de um utilizador sem ele estar iniciado na aplicação, ou fazer-se passar por um administrador.
- Manipulação de metadata, como replicar ou alterarem uma *cookie* ou um *token* de forma a elevarem os seus privilégios como utilizador ou modificar o conteúdo de uma mensagem.
- Utilizadores tentem aceder a pedidos ao website aos quais eles não deveriam ter acesso.

3.8 Proteger Todos os Dados

3.8.1 Descrição

Dados sensíveis como passwords, números de cartões de credito, registos médicos, dados pessoas e segredos de negócios exigem uma protecção extra, particularmente se esses dados fazem partes das leis de privacidade da União Europeia

(GDPR), leis de protecção de dados financiais como o PCI *Data Security Standard* (PCI DSS) ou outras leis.
Atacantes podem roubar dados da web ou *webservices* de muitas maneiras diferentes, como por exemplo: se uma informação sensível for enviada pela internet sem tipo de segurança, um atacante pode interceptar essa informação e visualiza-la ou até mesmo modificar o seu conteúdo, um ataque possível é também utilizar *SQL injection* para roubarem dados sensíveis e assim exporem esses dados ao publico.

3.8.2 Classificação dos Dados

É importante classificar os dados de um sistema e determinar quais dados são mais sensíveis. Cada categoria de dados pode ser mapeada utilizando as regras de protecção necessárias para cada nível de sensibilidade. Por exemplo, informação que ira ser publica não é da mesma categoria de sensibilidade que os de os cartões de credito.

3.8.3 Encriptar Todos os Dados em Transição

Quando dados sensíveis são transmitidos utilizando qualquer tipo de rede, comunicação segura *end-to-end* ou encriptação em transito deveram ser consideradas. TLS é o protocolo criptográfico que é mais comum e com maior suporte para criar uma comunicação segura, o TLS é utilizado em quase todo o tipo de aplicações (web, *webservices*, *mobile*) para comunicar utilizando uma rede de forma segura. TLS tem que ser propriamente configurado numa grande variedade de forma de forma a defender de forma segura as comunicações.
O principal beneficio de uma camada de transporte segura é a protecção dos dados de aplicações web de modificações e visualizações desses dados de pessoas ou entidades não desejadas quando os dados são transmitidos entre o cliente e o servidor que aloja o web server, e entre a aplicação web e a *back end* ou outro qualquer componente.

3.8.4 Encriptar os Dados Quando Estão Armazenados

A primeira regra da gestão de dados sensíveis é evitar todos ao máximo armazenar este tipo de dados, e se os dados tem que ser armazenados então é estritamente necessário utilizar o criptografia para proteger os dados de qualquer modificação ou divulgação não autorizada.
A criptografia é um dos tópicos mais complexos no que toca a segurança da informação, e é um dos tópicos que mais necessita de experiência e de conhecimento. É difícil escolher o método de criptografia mais correcto para cada situação, dado que cada um tem os seus pontos positivos e os seus pontos negativos, em adição a isto é um estudo de métodos criptográficos envolve teoria matemática e de números, o que cria uma grande barreira de entrada.
Em vez de criar de raiz mecanismos de criptografia é fortemente recomendado utilizar soluções de código aberto como o Projecto Google Tink, Libsodium, e

os vários serviços disponíveis de cloud ou as frameworks já existentes em algum software.

Aplicações Mobile: Armazenamento Local Seguro

As aplicações mobile sofrem um risco de existir uma fuga de dados, pois estes dispositivos são regularmente perdidos ou roubados. Como regra geral apenas os dados estritamente necessários devem ser armazenados num dispositivo móvel, mas se existir a necessidade de se armazenar dados um dispositivo móvel esses dados devem ser armazenados nos directórios específicos de casa sistema operativo.

Ciclo de Vida das Chaves

As aplicações contem vários "segredos"que são necessários para efectuar operações relacionadas com a segurança, essas operações incluem, passwords para conectar ao SQL, credencias de serviços *third party*, passwords, chaves SSH, chave utilizadas para encriptação, etc. Uma revelação não autorizada ou a modificação destes segredos pode levar a segurança do sistema ficar completamente completamente comprometida. Na gestão de segredos de aplicação é necessário considerar o seguinte:

- Não se deve armazenar segredos no código fonte, ficheiros de configuração ou passar-los utilizando variáveis de ambiente. Deve se utilizar ferramentas como GitRob ou TruffleHog para dar *scan* dos repositórios para encontrar segredos que estejam públicos, de forma a resolver esta falha de segurança.
- Guardar as chaves e outros segredos da aplicação em cofres de segredos como KeyWhiz ou Hashicorp's Vault project ou Amazon KMS de forma a armazenar de forma segura estes segredos e conseguir aceder a eles quando a aplicação está em execução.

3.8.5 Prevenção de vulnerabilidades

Com a protecção dos dado evita-se várias vulnerabilidades como a modificação de dados, as fugas de dados, ataques as chaves de uma aplicação e ataques a comunicações da aplicação.

3.9 Monitorizar e Implementar *Logs* de Segurança

3.9.1 Descrição

Logging é conceito que muitos programadores utilizam para o *debugging* e para efeitos de diagnostico. Utilizar *logs* para efeitos de segurança é um conceito simples, eles são feitos de forma a registar informação referente aos sistemas de segurança da aplicação. Monitorizar é fazer uma avaliação continua da aplicação em tem real utilizando os *logs* e as varias formas automatização. Essas

ferramentas podem também ser utilizadas em operações, *debugging* e para a segurança do sistema.

3.9.2 Benefícios de Utilizar *Logs* no Contexto da Segurança de um Sistema

Os *logs* de segurança podem ser utilizados para:

- Fornecer dados a sistemas de detecção de intrusões;
- Analises forenses e investigações;
- Satisfação dos requisitos de conformidade regulamentar

3.9.3 Implementação de Sistemas de *Logging*

A lista seguinte indica as melhores praticas na implementação de um sistema de *logs* de segurança:

- Os *logs* seguirem uma formatação padrão e os outros sistemas dessa organização devem seguir essa mesma formatação. Um exemplo de *framework* comum é a Apache Logging Services que permite criar *logs* consistentes entre várias linguagens de programação.
- Não ter *logs* muito extensos nem muito curtos. Por exemplo, deve-se sempre ter uma *timestamps* e informação da identidade como o IP ou o ID do utilizador, mas não se deve armazenar informação privada ou confidencial.
- As *timestamps* devem ser consistentes ao longo dos diferentes logs do sistema.

Logging para Detecção e Resposta a Intrusões

Os *logs* podem ser utilizados para identificar actividade que poderá ser maliciosa de um utilizador, actividade potencialmente maliciosa inclui:

- Submeter dados que são fora do limites numéricos expectáveis.
- Submeter dados que envolve alterar o tipo de dados que é expectável obter.
- Pedidos que violam as regras de acesso.
- (Para uma lista mais compreensiva sobre os possíveis ataques esta disponível aqui `https://www.owasp.org/index.php/AppSensor_DetectionPoints`.

Quando é detectado este tipo de actividade a aplicação deve ser capaz de registar esta actividade e categorizar como de uma actividade risco elevado para o sistema. Idealmente a aplicação também deve saber responder a um ataque já identificado, como por exemplo invalidar a sessão do utilizador ou banir a conta do utilizador. Este tipo de mecanismos de resposta permite ao software reagir em tempo real a um ataque já identificado.

3.9.4 Design de um Sistema de *Logs* Seguros

As diferentes soluções de *logging* devem ser construídas e geridas de forma segura, para isso devem seguir as seguintes regras:

- Codificar e validar qualquer tipo de carácter perigoso antes de registar o *log* de forma a prevenir *log injection* ou o forjamento de *logs*.
- Não se deve armazenar *logs* que contem informação sensível ou confidencial. Como por exemplo: não se dever armazenar nos *logs* passwords, ID's de sessões, cartões de credito, etc...
- Proteger a integridade dos *logs*. Um atacante pode tentar modificar o conteudo dos logs, para evitar isto as permissões de acesso aos ficheiros que contem os *logs* e auditorias aos *logs* devem ser consideradas.
- Os *logs* devem ser armazenados também num sistema central, isto é feito para garantir que se um nodo dos *logs* for comprometido esses dados não são perdidos. Isto também permite uma monitorização central dos *logs*.

3.9.5 Prevenção de vulnerabilidades

Os *logs* ajudam a evitar problemas de repudio, dado que toda a actividade dos utilizadores esta registada. A utilização dos *logs* em conjunto com um software de monitorização permite expandir o leque ainda mais das vulnerabilidades prevenidas, dado que pode analisar e agir contra ataques de "força bruta", *sql injection*, etc...

3.10 Manipulação de Erros e Excepções

3.10.1 Descrição

A manipulação de excepções é um conceito de programação que permite que uma aplicação responda com diferentes erros. A manipulação de excepções e erros é critico para criar código que seja confiável e seguro.
Manipulação de excepções e erros ocorre em todas as áreas de uma aplicação incluindo áreas criticas para a *business logic*, código de *frameworks* e as funções de segurança de um sistema.
Manipulação de erros é também importante numa perspectiva da detecção de intrusões. Certos ataques contra uma aplicação podem fazer disparar um erro, que pode ajudar a detectar que tipo de ataque em progresso.

Erros na Manipulação de Erros

A universidade de Toronto descobriram que ate pequenos erros no manipulação de erros ou simplesmente falhas na manipulação desses mesmos erros pode levar a falhas catastróficas num sistema. Erros na manipulação pode levar a diferentes vulnerabilidades do sistema:

- **Fugas de Informação:** fugas de informação sensível em mensagem de erro pode não intencionalmente ajudar os atacantes, por exemplo: um erro que retorna uma *stack trace* ou detalhes de erros podem fornecer ao atacante informação sobre o sistema. Até pequenas diferenças na forma como se faz a manipulação dos erro (exemplo., retornar "Invalido User"ou "Password Invalida"em erros de autenticação) podem fornecer os dados importantes aos atacantes. Como descrito acima, é necessário armazenar os erros para efeitos de *debugging* ou forenses, mas esta informação não pode ser exposta, especialmente para um cliente externo.

- **TLS bypass:** Erros na manipulação de erros do TLS podem levar a comprometer as conexões TLS por completo, como aconteceu no Apple goto "fail bug" que comprometeu todas as conexões TLS dos sistemas da Apple.

- **DOS:** A incapacidade da manipulação de erros pode levar aos sistema "ir a baixo", esta é um vulnerabilidade muito fácil de explorar por atacantes. Outras falhas na manipulação de erros podem levar ao aumento do consumo de CPU ou de espaço de disco que podem degradar o sistema.

3.10.2 Concelhos

- Manipulação de excepções numa forma centralizada para evitar blocos "try/catch"duplicados no código. Garantir que todo comportamento não expectável é propriamente tratado dentro da aplicação.

- Garantir que todas as mensagens de erro não fornecem informação critica, mas elas devem na mesma dar informação suficiente para que utilizador consiga responder.

- Garantir que todas as excepções são registados nos *logs* de uma forma que fornece informação suficiente aos sistemas de suporte possam entender o problema.

- Verificar e testar cuidadosamente o código que faz a manipulação de erros.

Capítulo 4

Conclusão

Com este projecto esperamos ter conseguido o objectivo de resumir os controlos pro-activos aconselhados pela OWASP, e ter explicado correctamente a quem interesse em desenvolver software mais seguro ou ajudar quem esteja a desenvolver software e não queira cometer erros de segurança.
Graças a este trabalho ficamos a compreender melhor algumas formas e medidas que nos podem ajudar imenso em como desenvolver software mais seguro e quão simples a maior parte delas são. Também foi muito positivo para perceber alguns erros que nos cometemos no passado e como poderíamos corrigir e melhorar esses projectos.

Bibliografia

[1] Owasp proactive controls, 2018.

[2] Wikipedia. Owasp, Nov 2019.