Análise de segurança em aplicações Web construídas a partir dos frameworks Django, Express e Flask

Rodrigo de Brito[1], Carlos André Batista de Carvalho[2]

Departamento de Computação – Universidade Federal do Piauí (UFPI)
Teresina – PI – Brasil

rb@ufpi.edu.br, candrebc@ufpi.edu.br

Resumo. *A escolha das tecnologias a serem usadas no desenvolvimento de uma aplicação é de grande importância, pois cada tecnologia possui pontos positivos e negativos, dependendo da situação. Atualmente o mercado dispõe de diversos frameworks contendo diferentes características nos mais variados aspectos, e um desses aspectos é como cada framework trata questões de segurança. Este trabalho apresenta características relacionadas a proteção oferecida pelos frameworks Django, Flask e Express em aplicações desenvolvidas com o intuito de realizar testes relacionados aos métodos para exploração de vulnerabilidades Injection, Broken Authentication e XSS. Utilizando ferramentas de análise estática sobre o código das aplicações e realizando testes com ferramentas de análise dinâmica e uma análise manual, será feita uma análise dos frameworks no que diz respeito aos métodos de segurança empregados por cada um, a eficácia das implementações de segurança e o esforço para tornar as aplicações seguras.*

Abstract. *The choice of technologies to be used in the development of an application is of great importance, because each technology has positive and negative points, depending on the situation. Currently the market has several frameworks containing different characteristics in various aspects, and one of these aspects is how each framework deals with security issues. This paper presents characteristics related to the protection offered by Django, Flask and Express frameworks in applications developed in order to perform tests related to the exploitation methods for Injection, Broken Authentication and XSS vulnerabilities. Using static analysis tools on the application code and performing tests with dynamic analysis tools and a manual analysis, an analysis of the frameworks will be made regarding the security methods employed by each one, the effectiveness of the security implementations and the effort to make the applications secure.*

1. Introdução

No decorrer do processo de desenvolvimento de uma aplicação *Web* uma das decisões a serem tomadas é a escolha das tecnologias que serão utilizadas ao longo do processo. Dentre as escolhas tecnológicas estão os *frameworks Web*. Um *framework*[1] é um conjunto de conceitos usados para resolver um problema de domínio específico. Cada *framework* possui bibliotecas com diferentes funções, seja para gerenciar o acesso a bancos de dados, sessões de usuários, envio de e-mails e como as aplicações tratam questões de segurança.

Com o aumento no uso de aplicações *Web* em organizações e a competição imposta pelo mercado, essas aplicações têm que ser desenvolvidas no menor tempo possível para encarar seus competidores [1]. Portanto, desenvolvedores podem deixar

[1] https://gaea.com.br/entenda-o-que-e-framework/

falhas de segurança ou fazerem uso de módulos, bibliotecas ou componentes de terceiros que podem estar vulneráveis. Tais fatores prejudicam seriamente a segurança das aplicações, um requisito essencial no ciclo de desenvolvimento de software. Aplicações *Web* conectadas através da Internet e de intranets implicam que elas são usadas para desenvolver algum tipo de negócio, logo, se tornando um alvo para usuários maliciosos que queiram obter algum tipo de ganho financeiro ou informação privilegiada sobre aquele negócio [2].

Oliveira et al. [1] e Likaj, Soheil e Giancarlo [3] realizaram comparações de segurança entre diversos *frameworks*, mas limitados a apenas uma vulnerabilidade. Ambos trabalhos demonstram métodos úteis na avaliação de *frameworks Web* e podem ser utilizados para tratar de outras vulnerabilidades.

Portanto, tendo em vista a quantidade de *frameworks* disponíveis no mercado e os muitos tipos de vulnerabilidades existentes, esse trabalho realiza uma análise em como Django, Flask e Express, alguns dos *frameworks* mais famosos para desenvolvimento de aplicações web, lidam com ataques do tipo *Injection*, *Broken Authentication* e *Cross-site scripting (XSS)*, visando encontrar eventuais falhas e sugerir formas para realizar suas correções. O intuito do trabalho é demonstrar como os *frameworks* selecionados desempenham em cada etapa de testes, mostrando o esforço necessário empregado por cada um para mitigar vulnerabilidades. Com os possíveis aspectos que cada *framework* pode encontrar para se tornar seguro, uma escolha entre qual deles é o mais adequado para um projeto é facilitada.

As vulnerabilidades testadas foram escolhidas com base na frequência que aparecem nos relatórios divulgados pela *OWASP Foundation*[2]. Na Figura 1 é apresentado o histórico do relatório da *OWASP Foundation* até 2017.

OWAPS TOP 10 - 2007	OWAPS TOP 10 - 2010	OWAPS TOP 10 - 2013	OWAPS TOP 10 - 2017
A1 - Cross Site Scripting (XSS)	A1 – Injection	A1 – Injection	A1 – Injection
A2 - Injection Flaws	A2 – Cross Site Scripting (XSS)	A2 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A3 - Malicious File Execution	A3 – Broken Authentication and Session Management	A3 – Cross-Site Scripting (XSS)	A3 – Sensitive Data Exposure
A4 - Insecure Direct Object Reference	A4 – Insecure Direct Object References	A4 – Insecure Direct Object References	A4 – XML External Entity (XXE)
A5 - Cross Site Request Forgery (CSRF)	A5 – Cross Site Request Forgery (CSRF)	A5 – Security Misconfiguration	A5 – Broken Access Control
A6 - Information Leakage and Improper Error Handling	A6 – Security Misconfiguration (NEW)	A6 – Sensitive Data Exposure	A6 – Security Misconfiguration
A7 - Broken Authentication and Session Management	A7 – Insecure Cryptographic Storage	A7 – Missing Function Level Access Control	A7 – Cross-Site Scripting (XSS)
A8 - Insecure Cryptographic Storage	A8 – Failure to Restrict URL Access	A8 – Cross-Site Request Forgery (CSRF)	A8 – Insecure Deserialization
A9 - Insecure Communications	A9 – Insufficient Transport Layer Protection	A9 – Using Components with Known Vulnerabilities	A9 – Using Components with Known Vulnerabilities
A10 - Failure to Restrict URL Access	A10 – Invalidated Redirects and Forwards (NEW)	A10 – Invalidated Redirects and Forwards	A10 – Using Components with Known Vulnerabilities

Figura 1. Histórico do relatório da *OWASP*

Como mostra a Figura 1, *Injection*, *Cross-site scripting* e *Broken Authentication* estiveram entre as 3 vulnerabilidades mais comuns em quase todas as edições, sendo que *Broken Authentication* não esteve entre as 3 primeiras apenas na edição de 2007 e

[2] https://owasp.org/www-project-top-ten

Cross-site scripting não esteve na edição de 2017. Em 2021 foi lançada uma nova edição do relatório (Figura 2), na qual a vulnerabilidade *Cross-site scripting* foi encaixada na categoria *A03:2021-Injection.*

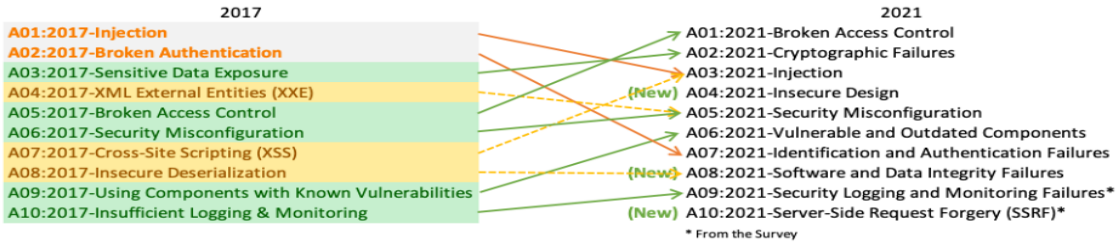

Figura 2. *OWASP Top Ten* 2021

Injection caiu para a terceira posição e *Broken Authentication* agora chamado de *"Identification and Authentication Failures"* ocupa a sétima posição. O relatório de 2021 descreve que uma maior disponibilidade de *frameworks* padronizados parece estar ajudando na queda no ranking, sendo um fator que também influenciou a escolha dessas vulnerabilidades para análise das proteções oferecidas pelos *frameworks*.

Na Seção 2 está o referencial teórico. Na Seção 3 estão os trabalhos relacionados. Na Seção 4, a metodologia. Na Seção 5, uma discussão sobre os resultados utilizando as ferramentas de análise, seguida dos resultados apresentados pelos *frameworks* Django, Flask e Express a respeito das vulnerabilidades testadas. Na Seção 6 é feita a conclusão do trabalho e trabalhos futuros partindo desse estudo.

2. Referencial teórico

Django, Flask e Express são ferramentas utilizadas para construção de aplicações *Web*. Apesar de possuírem propostas similares, a filosofia de cada uma diverge em alguns pontos. Django é um *framework* que possui mais configurações adicionadas na sua inicialização, com uma proposta diferente de *Express e Flask*, *microframeworks* onde funcionalidades são adicionadas à medida que são necessárias para a aplicação, dando essa liberdade para o desenvolvedor. O ciclo de vida do desenvolvimento seguro (Figura 3) pode guiar desenvolvedores em busca de um maior nível de segurança em suas aplicações.

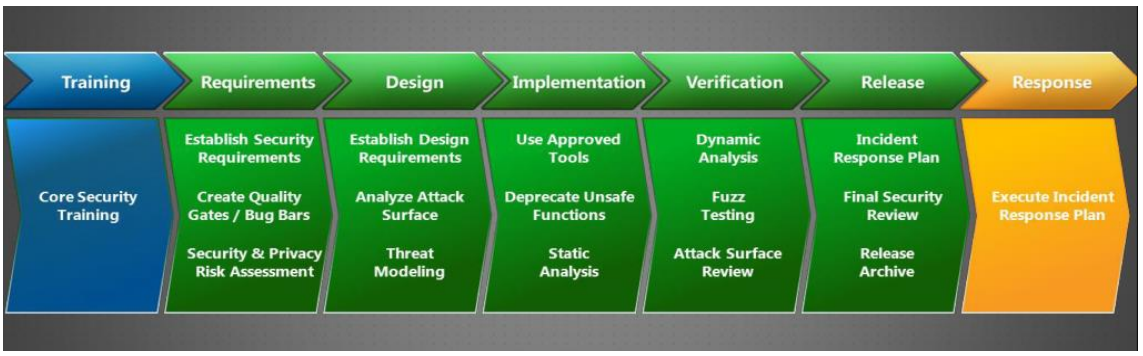

Figura 3. Ciclo de vida do desenvolvimento seguro

De acordo com o ciclo de vida do desenvolvimento seguro[3] mostrado na Figura 3, na fase da implementação de uma aplicação, uma das etapas presentes é a análise estática. A análise do código-fonte antes da compilação (chamado de *bundle* no contexto das aplicações *Web*) fornece um método altamente escalável de revisão de código de segurança e ajuda a garantir que as políticas de codificação segura estejam sendo seguidas.

Na etapa de verificação, é feita a análise dinâmica. Realizar a verificação em tempo de execução do software, que está totalmente compilado ou empacotado, ajuda verificar comportamentos da aplicação que só são aparentes quando todos os componentes estão integrados e em execução. Isso normalmente é alcançado usando uma ferramenta ou conjunto de ataques pré-construídos ou ferramentas que monitoram especificamente o comportamento da aplicação quanto à corrupção de memória, problemas de privilégios do usuário e outros problemas críticos de segurança.

Por fim, são realizados testes manuais, que são os testes direcionados às vulnerabilidades *Injection, Broken Authentication e XSS*. Essa última etapa de testes busca descobrir vulnerabilidades potenciais que estão no escopo desta pesquisa. Após cada etapa de testes, são observadas as vulnerabilidades encontradas e então é feita uma análise da proteção oferecida pelo *framework* sendo testado: quais estratégias são utilizadas para solução da vulnerabilidade.

2.1. Realização de testes e mitigação de vulnerabilidades

Para realização de testes de segurança nas aplicações desenvolvidas a partir dos *frameworks*, é utilizada como estratégia a criação de um ambiente de testes como em [2, 4], contendo implementações de aplicações feitas a partir dos *frameworks* Django, Express e Flask.

Após a realização dos testes, em ocasiões em que sejam encontradas falhas, é importante identificar de onde partem as vulnerabilidades encontradas para que seja possível entendê-las e encontrar uma forma para mitigá-las. Conforme identificado o ponto da aplicação que a tornou vulnerável, é feita uma busca por métodos adequados que possam solucionar a vulnerabilidade específica, através da documentação do *framework* ou em sites especializados em soluções de segurança, como o portal da *OWASP*. Depois que a possível solução para a vulnerabilidade é encontrada e aplicada, os testes são realizados novamente para que seja possível concluir se a mitigação foi feita ou não. A mitigação de uma falha em uma aplicação acontece quando é implementada uma solução que seja capaz de protegê-la nos cenários em que ela antes se mostrava vulnerável, sem afetar outros componentes que já se mostravam seguros anteriormente ou gerar novas falhas [3].

2.2. Sobre as vulnerabilidades abordadas

A primeira vulnerabilidade abordada neste trabalho é *SQL Injection,* da categoria *Injection*. Uma falha explorada principalmente em contextos onde é necessária a comunicação de uma aplicação com um banco de dados, como por exemplo, quando um usuário deseja realizar *login* em um sistema. São inseridos usuário e senha. Com esses

[3] https://docs.microsoft.com/pt-br/windows/security/threat-protection/msft-security-dev-lifecycle

dados, a aplicação gera uma *query SQL*: um comando capaz de executar ações no banco de dados. Usuários maliciosos podem tentar modificar a *query SQL* executada no banco de dados, gerando comportamentos inesperados.

Broken Authentication é dividido em duas áreas: gerenciamento de credenciais e gerenciamento de sessão. Uma aplicação está vulnerável em um desses pontos em cenários como: quando permite ataques automatizados de força bruta, permite que sejam criadas senhas fracas ou que já foram divulgadas em vazamentos de dados, não faz uso adequado de *tokens* de sessão: responsáveis por identificar um usuário enquanto logado, assim como o descarte desses *tokens* quando o usuário realiza *logout,* entre outros cenários. A utilização de *CAPTCHA* ou *reCAPTCHA* após um número grande de requisições por um mesmo *host* é um recurso adicional importante na proteção contra força bruta, impedindo um número incomum de tentativas de *login*, por exemplo.

XSS ou *Cross-site scripting* são ataques nos quais é injetado um *script* malicioso em uma página ou servidor *Web*. Ataques *XSS* ocorrem quando um atacante utiliza aplicações *Web* para enviar código malicioso, geralmente no formato de *script* a ser executado em navegadores para os demais usuários que visitarem a página onde o *script* foi injetado. Permitir a inserção de algumas *tags* da linguagem HTML em campos públicos de uma página *Web* ou que outros usuários tenham acesso, como uma seção de comentários e não tratar como essas *tags* são interpretadas, pode gerar comportamentos indesejados. Esse trabalho contém uma análise voltada ao *Stored XSS*, no qual o código malicioso é armazenado no banco de dados da aplicação, onde um cliente se conecta através do servidor das aplicações construídas a partir dos *frameworks* dessa análise.

3. Trabalhos relacionados

Na literatura há muitas referências à como alguns frameworks tratam questões de segurança, voltados a uma única vulnerabilidade. Hassan et al. [4] por exemplo, demonstram como funciona o gerenciamento de sessão e alguns tipos de ataque que podem explorar quebra de autenticação. Likaj, Soheil e Giancarlo [3] falam sobre *CSRF*, os tipos de ataque existentes, como explorá-los e estratégias utilizadas para realizar a mitigação dessa vulnerabilidade. Para ferramentas de testes, Micheelsen e Thalmann [5] apresentam uma ferramenta de análise estática chamada PyT, e mostram resultados de como aplicações feitas em *Django e Flask* performaram após passar pela análise do PyT. Ablahd e Dawood [6] apresentam o SQLID, uma ferramenta feita utilizando recursos do Flask com Django para análise estática de vulnerabilidades *Injection*. Oliveira et al. [1] reuniram 10 *web service frameworks* para realizar um teste inicial utilizando ferramentas de análise de segurança, dentre elas o *OWASP Zed Attack Proxy (ZAP)*, onde os *frameworks* que apresentaram vulnerabilidades no âmbito geral são eliminados nesta etapa, os *frameworks* que passaram dessa etapa são submetidos a testes específicos à vulnerabilidade *DoS*.

Mateo et al. [2] utilizaram combinações de ferramentas de análise estática, dinâmica e interativa, onde, as ferramentas de análise estática tiveram a função de analisar o código fonte da aplicação, sem que a aplicação estivesse ativa naquele momento. Ferramentas de análise dinâmica realizaram uma varredura por vulnerabilidades enquanto a aplicação estava em execução. Ferramentas de análise

interativa já são um pouco mais complexas, combinando características dos dois tipos mencionados anteriormente.

Visando também a utilização de um ambiente para realizar testes relacionados a uma vulnerabilidade específica, Likaj, Soheil e Giancarlo [3] criaram aplicações contendo formulários, para testes relacionados à *CSRF*.

4. Metodologia

Para realizar a análise dos *frameworks*, foi criada uma aplicação em cada um deles onde foram realizados testes manuais e automatizados. Os testes ainda foram aplicados conforme o ciclo de vida do desenvolvimento seguro (Figura 1).

Para esse trabalho, foram utilizados *Embold*[4] e *SonarQube*[5] como ferramentas de análise estática. Essas ferramentas foram escolhidas pois oferecem suporte às linguagens de programação utilizadas nos *frameworks* presentes neste trabalho, *Python* e *JavaScript*.

Para avaliação, *Embold* utiliza uma métrica chamada de *KPI Summary:* uma medida quantificável de desempenho ao longo do tempo para um objetivo específico que atribui uma nota de 0 a 100 em vários critérios, dentre eles a segurança da aplicação. *SonarQube* possui algumas métricas um pouco diferentes do *Embold*, apontando principalmente as vulnerabilidades e os chamados *security hotspots (*pontos de acesso de segurança)*:* os quais não são necessariamente problemas abertos a ataques. Os pontos de acesso de segurança destacam trechos de código sensíveis que precisam ser revisados manualmente, podendo haver ou não alguma vulnerabilidade dado o contexto específico.

Para a análise dinâmica, contida na etapa de verificação, foi utilizado o *OWASP ZAP*, uma ferramenta capaz de analisar aplicações em tempo de execução, apontar falhas na aplicação (quando existirem e estiverem dentre as falhas testadas pelo *OWASP ZAP*) e como resolvê-las. Para sua escolha, foram levados em consideração sua presença em trabalhos citados na seção anterior, contendo testes relevantes para o trabalho proposto. O *Burp Suite* também foi utilizado nessa etapa, automatizando ataques que buscam explorar falhas como *SQL Injection*, *XSS* e quebra de autenticação, fazendo o uso de dicionários com *payloads* maliciosos que buscam explorar essas vulnerabilidades.

Os testes finais são realizados de forma manual através da troca de parâmetros de requisição, alterando *payloads*, cabeçalhos, etc. Esses testes buscam encontrar falhas de configuração não cobertas por ferramentas de análise estática e dinâmica. A avaliação foi feita de maneira cíclica, ou seja, é selecionada uma vulnerabilidade para que possam ser feitos testes nas aplicações visando encontrar possíveis falhas e sugerir formas para resolvê-las, trabalhando desse modo com cada vulnerabilidade. A ordem foi *SQL Injection*, *Broken Authentication e XSS*.

4.1. Desenvolvimento do ambiente de testes

---

[4] https://embold.io/

[5] https://www.sonarqube.org/

Para realização de testes relacionados a *SQL Injection e Broken Authentication,* o ambiente de testes contém dois formulários: um para cadastro de um usuário e outro formulário de *login,* possibilitando a autenticação dos usuários. Após autenticado, deverá ser estabelecida uma sessão para aquele usuário. Para *XSS*, o usuário pode adicionar dados na página *Web*, como por exemplo, comentários. Para isso, a aplicação construída tem um modelo similar ao de um *Blog*, onde são feitas postagens e em cada postagem é possível que usuários adicionem comentários, que são visíveis aos próximos usuários que visitarem a postagem.

A medida que foram detectadas possíveis vulnerabilidades, foram realizadas buscas na documentação do framework ou em fontes recomendadas por colaboradores, buscando encontrar sugestões de bibliotecas ou implementações que possam mitigar as vulnerabilidades, como Likaj, Soheil e Giancarlo [3] fizeram.

4.2. Testes e análise de resultados

Para realização dos testes, foram construídas, utilizando ferramentas padrões disponibilizadas pelos *frameworks,* aplicações com ambientes voltados para os principais cenários em que as vulnerabilidades contidas nesse estudo são exploradas, como cadastro de usuários, *login* e solicitação de dados de páginas *Web*. Após os testes utilizando ferramentas de análise estática e dinâmica, foram realizados testes sobre *SQL Injection*, *Broken Authentication* e *XSS*, respeitando a abordagem cíclica.

Para interceptação e análise das requisições feitas, foi utilizado o *proxy* disponível na ferramenta *Burp Suite*, possibilitando a troca de parâmetros legítimos pelo uso de listas com *payloads* maliciosos. Foram analisados comportamentos como tempo de resposta das requisições, *status* de resposta retornado e erros gerados nas aplicações durante os testes.

Na etapa de *SQL Injection* foram utilizadas estratégias encontradas nos laboratórios da *PortSwigger*[6], podendo dividir os ataques em *Classic SQL Injection* (*Error-based* e *Union-based*) e *Blind SQL Injection (Boolean-based e Time-based)*. Para execução dos testes, foi utilizado o *Burp Suite* para automatização das requisições, mudando apenas o campo de usuário na tela de *login* das aplicações utilizando uma lista de *payloads*[7]. Os testes relacionados à *Stored XSS*, também tiveram como base laboratórios da *PortSwigger* e uma lista de *payloads*[8].

5. Resultados

Os resultados dos testes utilizando as ferramentas de análise estática, dinâmica, manuais e sobre as vulnerabilidades *SQL Injection*, *Broken Authentication* e *Stored XSS* foram documentados no decorrer desta seção.

5.1. Ferramentas de análise estática e dinâmica

Para Django, o *scan* utilizando *Embold* encontrou uma falha relacionada à exposição de credenciais de segurança dentro do arquivo *settings.py.* Há várias soluções para esse problema, como armazenar as credenciais em um ambiente separado da

---

[6] https://portswigger.net/
[7] https://github.com/payloadbox/sql-injection-payload-list
[8] https://github.com/payloadbox/xss-payload-list

aplicação, dentro do servidor que ela está em execução, mostrado no CWE-798[9]. Foi realizado um novo scan na aplicação, dessa vez não salvando as credenciais diretamente no arquivo *settings.py*, mas armazenando credenciais utilizando *Github Actions*[10], e nesse novo *scan* não foram reportadas vulnerabilidades no *Embold*. O *scan* realizado pelo *SonarQube* não encontrou vulnerabilidades diretas relacionadas à aplicação construída em Django. No entanto, foram destacados 2 *security hotspots*: o primeiro relacionado à permitir qualquer método *HTTP* em suas rotas. A solução para esse *security hotspot* é especificar o método aceito por cada rota, utilizando um recurso de Django chamado *@require_http_methods*[11]. O segundo ponto foi relacionado à variável de configuração *DEBUG* ter o valor *true*.

Uma varredura foi feita utilizando *OWASP ZAP,* que não encontrou nenhuma vulnerabilidade, apenas alertas informativos quanto ao uso do cabeçalho *Access-Control-Allow-Origin*[12] por estar definido como "*" permitindo assim que qualquer aplicação possa enviar requisições para o servidor. Para remover esse alerta, foi realizada uma configuração do *CORS*[13] (*Cross-Origin Resource Sharing*) da aplicação no arquivo *settings.py*, indicando o *host* que pode se comunicar com o servidor.

Durante a análise da aplicação construída a partir do *framework* Flask, o único ponto apontado pela ferramenta *Embold* foi a utilização de *queries SQL* puras em alguns pontos da aplicação. Uma discussão sobre esses pontos é apresentada com mais detalhes na subseção 5.2, específica aos testes relacionados à *SQL Injection*. *SonarQube* não encontrou vulnerabilidades diretas na aplicação construída em Flask, no entanto foi chamado atenção para 2 *security hotspots* relacionados às rotas *"/login"* e *"/signup",* onde eram permitidos mais de um método por rota.

OWASP ZAP trouxe alguns pontos como "*Cookie* sem o atributo *SameSite*", se referindo a assinatura dos cookies responsáveis por gerenciar sessões de usuário. Essa mudança é feita no pacote *session*, nativo do *framework* Flask, discutido na seção sobre *Broken Authentication*. Os outros alertas são para má configuração de cabeçalhos, como *Content-Security-Policy,* a ausência de tokens *Anti-CSRF* (relacionado a vulnerabilidade *CSRF*, que está fora do escopo) e ausência do cabeçalho *Anti-clickjacking*. Uma biblioteca chamada *Talisman*[14] foi testada, mas não apresentou uma compatibilidade ideal com versões do *framework* Flask mais recentes. Sendo assim, para resolver os alertas apontados pelo *OWASP ZAP*, os cabeçalhos de resposta foram configurados manualmente em todos os *endpoint*s.

Embold não encontrou falhas na aplicação construída utilizando Express e ainda deu nota máxima (100) no quesito segurança para a aplicação. *SonarQube* encontrou 2 *security hotspots* de risco baixo, sendo a configuração do *CORS* de forma incorreta e a divulgação da impressão digital da tecnologia utilizada como base do servidor. Usuários maliciosos podem utilizar essa informação para focar seus ataques na tecnologia específica, uma vez descoberta. A recomendação é tornar a obtenção dessa informação o

---

[9] https://cwe.mitre.org/data/definitions/798.html
[10] https://damienaicheh.github.io/github/actions/2021/04/15/environment-variables-secrets-github-actions-en.html
[11] https://docs.djangoproject.com/en/4.0/topics/http/decorators/
[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Origin
[13] https://developer.mozilla.org/pt-BR/docs/Web/HTTP/CORS/Errors
[14] https://github.com/GoogleCloudPlatform/flask-talisman

mais difícil possível. Como essa informação é divulgada pelo cabeçalho *x-powered-by* em cada requisição, há dois meios para removê-lo: a partir de simplesmente a*pp.disable("x-powered-by")* ou adicionando uma biblioteca chamada *Helmet*[15] para Express. A Tabela 1 apresenta um resumo com os resultados encontrados por cada ferramenta de análise.

Tabela 1. Resumo dos resultados das ferramentas de análise

	Embold	SonarQube	OWASP ZAP
Django	Dados sensíveis no arquivo settings.py	Especificar o método *HTTP* utilizado em cada rota da aplicação e variável *DEBUG* definida como *true*	Alerta sobre o cabeçalho *Access-Control-Allow-Origin*
Flask	Uso de queries SQL puras	Algumas rotas da aplicação aceitavam mais de um método *HTTP*	Ausência do atributo *SameSite* em *cookies* de sessão, ausência de cabeçalhos de segurança
Express	-	Configuração do *CORS* e divulgação da tecnologia utilizada por meio de cabeçalhos	Ausência de cabeçalhos, mas resolvido facilmente

5.2. Testes relacionados à *SQL Injection*

A aplicação em Django foi a mais simples e rápida no quesito de iniciar o projeto e deixá-la em execução por utilizar menos bibliotecas externas e portanto requerer menos configurações após sua inicialização.

Não foram encontradas falhas relacionadas a *SQL Injection* na aplicação feita em Django, onde quase nenhuma configuração específica foi necessária, podendo guiar um usuário com pouca ou nenhuma experiência com o *framework* Django à não cometer erros e assim construir uma aplicação segura sem muitas complicações relacionadas a *SQL Injection*. Por padrão, Django utiliza *query parameterization e input escaping,* duas estratégias que constituem o *Django ORM*. No entanto, o *Django ORM* também disponibiliza o método *raw()*, que aceita um parâmetro do tipo *string* contendo uma *query* a ser executada no banco. Durante os testes foi possível observar que o método *raw()* não realiza validações, ou seja, desde que a *string* obedeça a sintaxe *SQL,* ela será executada. O que torna possível que um usuário malicioso realize *SQL Injection*.

A aplicação em Flask teve um desempenho consistente durante os testes relacionados a *SQL Injection*, no entanto, uma maior complexidade foi exigida para

[15] https://github.com/helmetjs/helmet

tornar a aplicação funcional. Foi necessário instalar e configurar a biblioteca *SQLAlchemy*. *SQLAlchemy* é um kit de ferramentas *SQL* e biblioteca de mapeamento objeto-relacional (*ORM*) responsável por orquestrar a comunicação entre a aplicação e o banco de dados, similar ao *Django ORM*. Como detectado na seção a respeito da análise estática por meio da ferramenta *Embold*, Flask demonstrou estar vulnerável a *SQL Injection* por possuir *queries SQL* sendo executadas através de *raw SQL*, onde não é feita a parametrização por meio de um *ORM*. A utilização de *raw SQL* permite realizar ações que comprometem a integridade, confidencialidade e o controle ao acesso da aplicação, havendo diferentes métodos para mitigação nas diferentes fases de desenvolvimento, segundo o *CWE-89*[16].

Nos testes realizados na aplicação construída a partir do *framework* Express, não foram encontradas falhas *SQL Injection*. A complexidade para construir a aplicação foi semelhante à da aplicação em Flask. No entanto, houve uma pequena diferença na definição do *ORM*. Para as aplicações utilizando Flask foi mais comum encontrar *SQLAlchemy* nas aplicações utilizadas como exemplo para os testes do que qualquer outro *Python ORM*, como *Pony*[17], por exemplo. Para Express, no entanto, verificou-se que duas ferramentas sempre apareciam com certa constância: *Sequelize*[18] e *TypeORM*[19]. Nenhum dos dois *ORMs* deram indícios de alguma vulnerabilidade.

Para *raw SQL, Sequelize* dispõe de uma técnica chamada *Replacements* explicada na seção *Raw Queries* de sua documentação. Após a realização de alguns testes onde não são encontradas execuções de comando indesejadas, percebe-se que é realizada uma parametrização na construção da *query SQL*, mostrando que essa estratégia é uma estratégia segura, no caso do *Sequelize ORM*. A Tabela 2 apresenta um resumo com os principais pontos observados nas aplicações.

Tabela 2. Resumo dos resultados sobre SQL Injection

	SQL Injection
Django	Possui um ORM nativo, o Django ORM. Não apresentou falhas. Não realiza validações em *raw SQL*.
Flask	Não possui um ORM nativo. *SQLAlchemy* foi a opção mais utilizada encontrada, não apresentando falhas. Também não valida *queries* SQL geradas através de *raw* SQL.
Express	Maior variedade de ORMs disponíveis. *Sequelize* e *TypeORM* foram testados e não apresentaram falhas durante os testes.

SQL Injection é uma vulnerabilidade existente há muitos anos e com um potencial destrutivo enorme, sendo assim, é justo que as formas para combater esse tipo de falha já estejam bem consolidadas. Vale ressaltar que o uso de *raw SQL* abre brechas

[16] https://cwe.mitre.org/data/definitions/89.html
[17] https://ponyorm.org/
[18] https://sequelize.org/
[19] https://typeorm.io/

para ações indesejadas, já que por geralmente não utilizarem parametrização, o banco de dados sozinho não é capaz de diferenciar quais comandos são maliciosos e quais são genuínos. Então, a partir do estudo feito, *Django ORM*, *Sequelize* e *SQLAlchemy* se saíram como opções sólidas para serem utilizados como *ORM*, prevenindo ataques do tipo *SQL Injection* dos mais variados tipos.

5.3. *Broken Authentication*

Django possui algumas configurações que são implementadas por padrão e que auxiliam na segurança da aplicação no contexto de *Broken Authentication,* através de pacotes como *UserAttributeSimilarityValidatior* que verifica se o usuário não está inserindo uma senha que contenha caracteres já encontrados em algum outro campo do formulário sendo enviado, como por exemplo, seu nome ou data de nascimento. *MinimumLengthValidator* realiza uma verificação no tamanho da senha. *CommonPasswordValidator* impede que sejam passadas senhas muito simples, como "abcd1234". Por fim, *NumericPasswordValidator* impede que sejam usados apenas números na senha, como "12345678". Django, no entanto, não impede que inúmeras requisições seguidas sejam realizadas pelo usuário, necessitando de uma biblioteca extra para esse ponto. A implementação do *CAPTCHA* foi usada como validação adicional, já que Django não protege completamente as rotas onde formulários são enviados, em caso de ataques de força bruta. Para a implementação do *CAPTCHA* foi utilizada a biblioteca *django-simple-captcha*[20]. O gerenciamento de sessões no *framework* Django é feito através de *cookies*, sendo salvo em uma tabela chamada *django_session*.

Flask não apresentou nenhuma resistência a ataques de força bruta, permitindo múltiplas requisições pelo formulário de *login* criado. Para gerenciamento de sessões, Flask dispõe de um módulo nativo chamado *session.* No entanto, são poucas as bibliotecas disponíveis para gerenciamento de *CAPTCHA* em Flask. Foi testada a biblioteca *flask-simple-captcha*[21] que apresenta problemas durante sua configuração, sendo incompatível com a versão atual do *framework* Flask. Em decorrência dos problemas citados, a solução encontrada para Flask foi a implementação do *reCAPTCHA*[22].

Express não mostrou nenhuma proteção nativa para ataques de força bruta, também não há validações para criação de senhas fortes, como sugere o relatório da *OWASP* em *Broken Authentication*[23]. Para validação dos dados enviados foi escolhida a biblioteca *celebrate*[24], um *middleware* que torna simples a validação dos parâmetros enviados em uma requisição e permite adicionar validações personalizadas para cada campo, por exemplo, validar se está sendo passado um endereço de e-mail válido. Para proteção contra ataques do tipo força bruta, a biblioteca *rate-limiter-flexibe*[25] conseguiu mitigar esses ataques, permitindo configurar permissões como 5 requisições por IP a cada 5 segundos. Essa biblioteca, porém, não disponibilizou a exibição de um *reCAPTCHA*, estratégia utilizada para validar se as requisições não estão sendo feitas de

---

[20] https://github.com/mbi/django-simple-captcha
[21] https://github.com/cc-d/flask-simple-captcha
[22] https://developers.google.com/recaptcha/docs/v3
[23] https://owasp.org/www-project-top-ten/2017/A2_2017-Broken_Authentication
[24] https://github.com/arb/celebrate
[25] https://dev.to/wandealves/tratamento-de-ataques-forca-bruta-e-ddos-em-api-nodejs-2gp

forma automatizada. Para essa validação, o *reCAPTCHA* foi implementado através da biblioteca *express-recaptcha*[26], um *middleware* entre o *reCAPTCHA* da *Google* e a aplicação Express. Em relação ao gerenciamento de IDs de sessões, Express possui a biblioteca *express-session*[27] que atua como um *middleware*, gerenciando sessões através de *cookies,* já que servidores *Web* não gerenciam sessões através do *localStorage*[28]. A Tabela 3 apresenta um resumo dos principais pontos observados durante os testes relacionados à *Broken Authentication.*

Tabela 3. Resumo dos resultados sobre *Broken Authentication*

	Broken Authentication
Django	Possui bibliotecas internas já configuradas para gerenciamento de sessão. Também dispõe de ferramentas para validar campos na requisição. Não impede que sejam realizadas muitas requisições por um único usuário (força bruta).
Flask	Não apresenta proteção para ataques de força bruta, mas dispõe de um módulo para gerenciamento de sessões utilizando *cookies*.
Express	Não apresentou proteção para ataques de força bruta ou gerenciamento de sessão.

Broken Authentication é uma vulnerabilidade de escopo muito amplo com bastante pontos que necessitam de cuidado. O estudo feito sobre Django mostra que ele está na frente dos outros frameworks em relação à essa vulnerabilidade, porém não se torna totalmente imune, uma vez que permite múltiplas requisições em um curto espaço de tempo deixando a aplicação vulnerável nesse quesito. Express e Flask requerem um cuidado maior em relação aos métodos de mitigação implementados, cabendo ao desenvolvedor cuidar de cada detalhe a respeito da autenticação de sua aplicação.

5.4. *Cross-site scripting (XSS)*

Segundo a arquitetura[29] proposta pelo *framework* Django, as páginas *Web* solicitadas são retornadas através de funções de visualização que recuperam o *template HTML* solicitado pelo usuário. Dos *payloads* testados, nenhum gerou comportamentos inesperados na página. Segundo a seção de segurança da documentação[30], *templates* Django escapam caracteres específicos que são particularmente perigosos para o *HTML* da página (termo chamado *escaping*). Esse é um comportamento padrão dos *templates* nativos em Django, mas que pode ser desabilitado manualmente pelo desenvolvedor.

Os *payloads* também foram testados na aplicação em Flask, que não demonstrou comportamentos inesperados. Flask faz uso de uma *template engine* chamado *Jinja2*[31],

[26] https://github.com/pdupavillon/express-recaptcha
[27] https://expressjs.com/en/resources/middleware/session.html
[28] https://medium.datadriveninvestor.com/cookies-vs-local-storage-2f3732c7d977
[29] https://docs.djangoproject.com/en/4.0/topics/http/views/
[30] https://docs.djangoproject.com/en/4.0/topics/security/
[31] https://jinja.palletsprojects.com/en/3.1.x/

mencionado na seção de segurança de sua documentação[32]. Vale ressaltar ainda que há exceções destacadas na documentação em que o *escaping* também pode ser desabilitado com o uso da *template tag "safe"* (como em Django). Um ponto importante sobre *Jinja2* é *XSS* por injeção de atributo. Por exemplo, a *tag*: *<input value={{ value }}>* deixa um possível vetor de ataque. Caso *value* seja carregado do banco de dados como um *script*, o navegador o interpreta como tal, não havendo *escaping*. Para mitigação, colocando *{{ value }}* entre aspas (simples ou duplas) resolve essa situação: *<input value="{{ value }}">*, como indicado na documentação.

O *framework* Express não recomenda o uso de uma *template engine* específica como os outros *frameworks*, mas possui uma lista em sua documentação com uma breve descrição sobre cada uma delas. Através de aplicações como *Insomnia*[33] e *Postman*[34], capazes de realizar requisições *Web* e mostrar suas respostas, foram realizados testes na aplicação construída a partir do *framework* Express. Ao armazenar um dado como *"<script>alert('XSS!')</script>"* há um problema, já que caso o cliente da aplicação não utilizar *escaping,* será exibido um alerta. Para resolver essa falha, foi utilizada a biblioteca *xss*[35] que utiliza uma estratégia de *whitelist*, ou seja, caso uma *tag* não for explicitamente declarada segura, a biblioteca realiza *escaping* na entrada especificada, evitando o possível retorno de *tags* maliciosas. A Tabela 4 apresenta um resumo dos principais pontos relacionados à *Stored XSS* encontrados.

Tabela 4. Resumo dos resultados sobre *Stored XSS*

	Stored XSS
Django	Realiza *escaping* através de uma *template engine* nativa.
Flask	Realiza *escaping* através da *template engine Jinja2*. Injeção por atributo (na tag <a> especificamente) é um risco, necessitando a adição do cabeçalho *Content-Security-Policy (CSP)* em suas requisições.
Express	Não recomenda o uso de uma *template engine* específica (mas cita algumas em sua documentação) e os dados enviados pelo cliente da aplicação não são validados.

Cross-site scripting (XSS) ou especificamente *Stored XSS*, tem uma forma de mitigação eficaz em *template engines*, sendo utilizada por padrão em aplicações Django e Flask. O *framework* Express apesar de não sugerir uma *template engine* específica, apresenta algumas dessas ferramentas em sua documentação. No entanto, a mitigação da vulnerabilidade *XSS* realizada em Express através de uma biblioteca adicional não demonstrou dificuldade em sua execução.

6. Conclusão e Trabalhos Futuros

[32] https://flask.palletsprojects.com/en/2.1.x/security/
[33] https://insomnia.rest/
[34] https://www.postman.com/
[35] https://www.npmjs.com/package/xss

O uso de variadas ferramentas de análise estática como *Embold* e *SonarQube* foi capaz de trazer valiosos *insights* a respeito da qualidade do código das aplicações, do ponto de vista de segurança. Cada ferramenta trouxe diferentes fontes de informação sobre as vulnerabilidades encontradas, assim como diferentes pontos que não são identificados como falha imediatamente, mas requerem checagem manual do desenvolvedor, para que não possam trazer problemas futuros. A ferramenta de análise dinâmica *OWASP ZAP* desempenha um papel importante na análise da aplicação em execução, indicando pontos onde a segurança pode ser melhorada.

Cada *framework* possui pontos específicos que podem trazer mais comodidade, dependendo do escopo do projeto. Ter muitas soluções prontas, como no *framework Django,* não significa que um certo cuidado não é necessário, tendo em vista que os testes realizados mostram pontos da aplicação que podem se tornar vulneráveis, caso não configurados seguindo boas práticas. Express e Flask, apesar de minimalistas, requerem bastante cuidado na implementação de cada funcionalidade.

Esse trabalho trata de como a segurança é tratada principalmente pelo lado do servidor, porém algumas falhas podem ser evitadas com o auxílio de mitigações feitas na aplicação do lado do cliente, diminuindo a quantidade de requisições que o servidor deve tratar, além de dificultar ações realizadas por um usuário malicioso, sendo um tópico interessante para um trabalho futuro. Trabalhos futuros podem também explorar mais vulnerabilidades do lado do servidor, como *HTTP Request Smuggling* e *Server-side request forgery.*

Referências

[1] Oliveira, R. A., RAGA, M. M., Laranjeiro, N., Vieira, M., An approach for benchmarking the security of web service frameworks. Future Generation Computer Systems, v. 110, p. 833-848, set. 2020.

[2] Mateo T. F., Bermejo H. J.-R., Bermejo H. J., Sicilia M. J.-A., Argyros, M.I. On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications. Appl. Sci., Spain, v. 10, n. 24, dez. 2020.

[3] Likaj, X., Soheil, K., Giancarlo, P. Where We Stand (or Fall): An Analysis of CSRF Defenses in Web Frameworks., ACM., p. 370-385, out. 2021.

[4] Hassan, M. M., Nipa, S. S., Akter, M., Haque, R., Deepa, F. N., Rahman, M., Siddiqui, M. A., Sharif, M. H. Broken Authentication and Session Management Vulnerability: A Case Study Of Web Application. International Journal of Simulation. abr. 2018.

[5] Micheelsen, S., Thalmann, Bruno. A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications. Aalborg University. mai. 2016.

[6] Ablahd AZ, Dawood SA. Using Flask for SQLIA Detection and Protection. Tikrit Journal of Engineering Sciences 2020; 27(2): 1-14.