
Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

# ZAPI - FERRAMENTA PARA ANÁLISE DE VULNERABILIDADES EM APIS

ANTÓNIO PEDRO MACHADO PINTO

Leiria, Setembro de 2023


Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

# ZAPI - FERRAMENTA PARA ANÁLISE DE VULNERABILIDADES EM APIS

ANTÓNIO PEDRO MACHADO PINTO
Número: 2212981

Projeto realizado sob orientação do Professor Doutor Marco António de Oliveira Monteiro (marco.monteiro@ipleiria.pt).

Leiria, Setembro de 2023

# AGRADECIMENTOS

Agradeço ao meu orientador, Marco Monteiro, pela orientação e auxílio proporcionado ao longo de todo o desenvolvimento. Quero também dedicar um agradecimento especial aos meus pais, família e amigos, cujo apoio constante impulsionou a concretização deste projeto. Por fim, um agradecimento a mim mesmo, como um lembrete para o futuro, pela persistência e foco mesmo durante momentos difíceis.

# RESUMO

As Application Programming Interfaces (APIs) desempenham um papel fundamental ao simplificar e agilizar a comunicação entre sistemas heterogéneos. No entanto, a despeito da sua utilidade, uma série de estudos recentes chamam à atenção, por estas surgirem como o principal vetor de ataque. Atacantes perspicazes, ao conseguirem acesso aos dados de uma API, podem comprometer informações sensíveis e, consequentemente, afetar a credibilidade de uma organização. É neste contexto que surge o projeto ZAPI, uma solução projetada para identificar vulnerabilidades em APIs RESTful tendo em conta as falhas identificadas no Open Web Application Security Project (OWASP) API Top 10 de 2023. Num cenário onde as opções gratuitas são limitadas, o ZAPI surge como uma alternativa, destinada a preencher essa lacuna.

Em suma, o ZAPI não só adota uma abordagem focada e atualizada para identificar vulnerabilidades de APIs, mas também desempenha um papel crucial ao preencher a carência de soluções gratuitas. Como resultado, promove o avanço contínuo da segurança nas APIs e contribui para um ecossistema digital mais resiliente e seguro.

*Palavras-chave: APIs, Vulnerabilidades, OWASP, Ataques, Cibersegurança*

# ABSTRACT

APIs play a fundamental role in simplifying and streamlining communication between heterogeneous systems. However, despite their usefulness, a series of recent studies draw attention to the fact that they have emerged as the primary vector for attacks. Skilled attackers, upon gaining access to an API data, can compromise sensitive information and consequently impact an organization's credibility. It is within this context that the ZAPI project arises, a solution designed to identify vulnerabilities in RESTful APIs, taking into account the flaws identified in the 2023 OWASP API Top 10. In a scenario where free options are limited, ZAPI emerges as an alternative aimed at filling this gap.

In short, ZAPI not only adopts a focused and updated approach to identify vulnerabilities in APIs but also plays a crucial role in bridging the lack of free solutions. As a result, it promotes the continuous advancement of API security and contributes to a more resilient and secure digital ecosystem.

*Keywords: APIs, Vulnerabilities, OWASP, Attacks, Cybersecurity*

# ÍNDICE

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS

| | |
|---|---|
| API | Application Programming Interface. |
| ASTRA | Automated Security Testing For REST API's. |
| CI/CD | Continuous Integration/Continuous Delivery. |
| CORS | Cross-origin Resource Sharing. |
| crAPI | Completely Ridiculous API. |
| CRUD | Create, Read, Update and Delete. |
| CSRF | Cross Site Request Forgery. |
| DDoS | Distributed Denial Of Service. |
| DOS | Denial Of Service. |
| gRPC | gRPC Remote Procedure Calls. |
| GUI | Graphical User Interface. |
| HMAC | Hash-based Message Authentication Code. |
| HTML | HyperText Markup Language. |
| HTTP | Hypertext Transfer Protocol. |
| IDE | Integrated Development Environment. |
| IPMA | Instituto Português do Mar e da Atmosfera. |
| JSON | JavaScript Object Notation. |
| JWT | JSON Web Token. |
| MITM | Man in the middle. |

| | |
|---|---|
| NLP | Natural Language Processing. |
| OAS | OpenAPI Specification. |
| OAuth | Open Authorization. |
| OWASP | Open Web Application Security Project. |
| Protobuf | Protocol Buffers. |
| QUIC | Quick UDP Internet Connections. |
| RAML | RESTful API Modeling Language. |
| REST | Representational State Transfer. |
| RPC | Remote Procedure Call. |
| SOAP | Simple Object Access Protocol. |
| SSRF | Server Side Request Forgery. |
| TCP | Transmission Control Protocol. |
| UDP | User Datagram Protocol. |
| UI | User Interface. |
| URL | Uniform Resource Locator. |
| UUID | Universal Unique Identifier. |
| UX | User Experience. |
| WAAP | Web Application and API Protection. |
| WADL | Web Application Description Language. |
| WAF | Web Application Firewall. |

XML Extensible Markup Language.

YAML YAML Ain't Markup Language.

# 1

# INTRODUÇÃO

A rápida evolução tecnológica, as tecnologias distintas existentes no mercado e a imergente quantidade de novos dispositivos conectados com a *internet*, veem reforçar a necessidade de boas práticas de segurança. Vivemos num mundo cada vez mais conectado e dependente das novas tecnologias. Muitos destes sistemas são construídos para serem de fácil utilização e as boas práticas de segurança são deixadas para segundo plano. A cibersegurança tem como princípio proteger todos os ativos de uma organização, contra ataques maliciosos, de modo a minimizar os riscos existentes.

Algo bastante adotado para desenvolver aplicações modernas é a utilização de Application Programming Interfaces (APIs), pois permitem a integração de diferentes sistemas e permitem a criação de novas aplicações e serviços, aproveitando conteúdo já desenvolvido. Por exemplo, se um programador pretendesse desenvolver uma aplicação para expor as condições atmosféricas de Portugal, ele poderia utilizar a API do Instituto Português do Mar e da Atmosfera (IPMA). Assim, a sua aplicação iria consumir os dados da API desenvolvida por uma entidade externa e existiria um menor esforço de desenvolvimento. Por norma, as APIs são frequentemente utilizadas para aceder a dados sensíveis, tais como informações financeiras ou pessoais, ou para realizar ações críticas, como executar transações financeiras. Isto representa um problema, caso um atacante consiga obter acesso não autorizado a estes dados. Uma exposição de dados através de uma falha na API pode resultar em consequências graves, tais como usurpação de identidade, prejuízos e perda de reputação de uma organização. Além disso, a inexistência de *logs* e monitorização pode dificultar a deteção e atrasar a resposta a incidentes de segurança. As APIs podem ser exploradas por meio de solicitações maliciosas, se a API não for programada com a lógica correta. Assim, o atacante não necessita de desenvolver um *exploit* para invadir o sistema, apenas necessita de compreender a lógica por trás da aplicação e tirar partido de uma solicitação não prevista pelo programador (Ball, 2022).

Existem alguns casos reais, que demonstram o problema. Por exemplo, o serviço de pagamentos Venmo sofreu um ataque, em 2019, onde foram expostas mais de 200 milhões de transações, o atacante acedeu à informação através da API

(Whittaker, 2019). Em 2021, dados de 700 milhões de utilizadores do LinkedIn foram colocados à venda na *dark web*. Para aceder ao conteúdo, o atacante explorou a API (A. T. Seals e T. Seals, 2021). No dia 11 de fevereiro de 2022, foi descoberta uma vulnerabilidade crítica na API do coinbase, a vulnerabilidade estava presente num *endpoint* que não era validado corretamente. A coinbase pagou $250.000 como recompensa (*Retrospective: Recent Coinbase Bug Bounty Award* 2022). Considerando este novo vetor de ataque, a Open Web Application Security Project (OWASP) lançou um projeto separado dedicado exclusivamente à segurança de APIs. A primeira versão, OWASP API Security Top 10, foi lançada em 31 de dezembro de 2019. No entanto, a versão mais atual é a OWASP API Security Top 10 2023, lançada em 3 de julho de 2023 (*OWASP API Security Top 10 2023 has been released* 2023).

Em 2021, a Gartner referiu o seguinte: "em 2022 os ataques às APIs tornar-se-ão o vetor de ataque mais comum". Também previu que "até 2024, os abusos e violações de dados quase duplicarão" (*API Security: Protect your APIs from Attacks and Data Breaches* 2021). Outro estudo conduzido pela Salt, no primeiro quadrante de 2023, afirma que "houve um aumento de 400% do tráfego malicioso em APIs ao longo dos últimos seis meses", "apenas 23% dos inquiridos acreditam que as suas abordagens de segurança, atuais, são muito eficazes na prevenção de ataques a APIs" e que "94% tiveram problemas de segurança nas APIs de produção" (*Salt security: State of API security report Q1 2023* 2023).

Com o crescimento da computação em nuvem e arquiteturas de microsserviços, as APIs tornam-se um meio de comunicação cada vez mais importante entre diferentes sistemas e componentes. No entanto, a segurança em APIs é um desafio agravado pela crescente complexidade e interconectividade dos sistemas modernos. Para enfrentar estes desafios, as organizações precisam de implementar medidas de cibersegurança adequadas para proteger as APIs e os sistemas com os quais interagem.

## 1.1 OBJETIVOS

Considerando o risco atual e as previsões realizadas pela Gartner, o objetivo deste projeto é contribuir com uma solução gratuita, uma vez que há uma grande escassez destas soluções na gestão segura de APIs. De uma forma mais concreta, pretende-se o seguinte:

- Desenvolver uma ferramenta gratuita e automatizada para identificar vulnerabilidades em APIs, uma vez que existem poucas ferramentas para este fim e o processo é predominantemente manual.

Assim, espera-se poder contribuir para melhorar o panorama geral da segurança nas APIs, ao facilitar o processo de identificação de vulnerabilidades.

## 1.2 ESTRUTURA DO RELATÓRIO

Este documento encontra-se dividido em seis capítulos com a seguinte estrutura:

- **Capítulo 1: Introdução**

  No primeiro capítulo, é efetuada uma introdução a abordar o tema em questão, são apresentados os problemas atuais com base em diversas estatísticas relevantes. Conclui-se essa secção expondo os objetivos delineados para o projeto;

- **Capítulo 2: Conceitos**

  No segundo capítulo, são apresentados os conceitos indispensáveis para a completa compreensão do tema e do projeto desenvolvido;

- **Capítulo 3: Estado da Arte**

  No terceiro capítulo, procede-se a uma análise de todas as soluções de caráter gratuito voltadas à identificação de vulnerabilidades em APIs;

- **Capítulo 4: Desenvolvimento**

  No quarto capítulo, procede-se à descrição do desenvolvimento do projeto proposto;

- **Capítulo 5: Resultados**

  No quinto capítulo, a aplicação desenvolvida é submetida a testes em várias APIs vulneráveis, juntamente com outras aplicações identificadas no estado da arte, de modo a analisar as vulnerabilidades identificadas por cada

uma delas. Finalizando com a extração dos resultados alcançados.

- **Capítulo 6: Conclusões**

  O sexto capítulo destaca-se por apresentar as conclusões extraídas do trabalho elaborado, resumindo de forma clara os principais resultados alcançados e as implicações encontradas;

# 2

# CONCEITOS

Neste capitulo serão apresentados os conceitos necessários para compreender o trabalho desenvolvido. Será realizada uma pequena introdução ao Hypertext Transfer Protocol (HTTP), às arquiteturas e protocolos utilizados no desenvolvimento de APIs, vulnerabilidades existentes e, por fim, os problemas e limitações atuais das ferramentas focadas em identificar vulnerabilidades.

É relevante notar que, embora tenham sido considerados diferentes tipos de APIs para contextualizar o tema, este projeto concentra-se exclusivamente nas APIs Representational State Transfer (REST).

## 2.1 HTTP

O protocolo HTTP é a base para a comunicação na *web*. Uma mensagem HTTP é tradicionalmente enviada pelo protocolo Transmission Control Protocol (TCP), exceto na versão mais atual do HTTP/3. Nesta ultima, a mensagem é envida por User Datagram Protocol (UDP) através de um protocolo recente, chamado Quick UDP Internet Connections (QUIC). Este protocolo garante maior velocidade e segurança, pois é utilizada uma nova abordagem para estabelecer conexões, chamada *0-RTT*, que significa tempo zero de ida e volta (Bishop, 2022). Assim, o cliente pode começar a enviar dados para o servidor imediatamente após a ligação ser estabelecida, sem esperar que o servidor confirme que está pronto para receber dados, ao contrário do *3-way handshake* do TCP. A segurança é assegurada pela encriptação de todos os dados. Porém, o foco deste trabalho será nas versões mais comuns do protocolo HTTP, nomeadamente HTTP/1 e HTTP/2.

A comunicação entre cliente e servidor é realizada com pedidos HTTP e respostas HTTP a um determinado recurso. Tanto os pedidos quanto as respostas têm *headers* que permitem passar informações adicionais. Além disso, embora possa existir um *body* com os dados a serem enviados ou recebidos, é importante observar que ele pode não ser utilizado e estar vazio. Um recurso é o alvo de um pedido HTTP, identificado pelo Uniform Resource Locator (URL) (Fielding et al., 2022). Por exemplo, uma

página *web*. Mas, para interagir com estes recursos é necessário utilizar um dos seguintes métodos (Fielding et al., 2022):

- **GET**
  Para obter um recurso;
- **POST**
  Permite criar ou adicionar um recurso;
- **PUT**
  Para substituir um recurso;
- **DELETE**
  Permite eliminar um recurso;
- **PATCH**
  Para atualizar parcialmente um recurso;
- **HEAD**
  Permite obter os *headers* de um recurso;
- **TRACE**
  Para realizar *debuging*;
- **OPTIONS**
  Quando se pretende obter os métodos disponíveis;
- **CONNECT**
  É usado para criar um túnel HTTP através de um servidor proxy.

Por fim, o servidor devolve uma resposta contendo um código de três dígitos. Existem cinco grupos, distintos, identificados pelo primeiro digito, são eles (Fielding et al., 2022):

- **1xx**
  Códigos informativos;
- **2xx**
  Associado a códigos de sucesso;
- **3xx**
  Quando ocorrem redirecionamentos, ou seja, um recurso mudou de localização;
- **4xx**
  Erros do cliente;
- **5xx**
  Erros do servidor.

Um dos aspetos importantes do HTTP é o facto de ele ser um protocolo *stateless* (Fielding et al., 2022). Assim, não é armazenada informação anterior. Sempre que é enviado um pedido é preciso fornecer toda a informação necessária. Deste modo, para manter o estado de um utilizador, hipotético, que fez vários pedidos HTTP, relacionados entre si, é vital utilizar *headers* adicionais em todos os pedidos, tais como *cookies* ou *tokens*.

## 2.2 APIS

As APIs definem um conjunto de regras que permitem a diferentes sistemas e aplicações comunicar entre si, utilizando protocolos padronizados (Adkuloo, 2023). Consistem num conjunto de *endpoints* (em APIs REST), ou seja, a terminação do URL, que representam um recurso ou função específica que pode ser acedida através da API.

O protocolo mais utilizado por APIs é o HTTP, o mesmo protocolo utilizado para transferir dados na *web* (Adkuloo, 2023). No entanto, também podem ser utilizados outros protocolos, tais como *WebSockets*, concebidos para a comunicação em tempo real. Ou seja, é criada uma conexão bidirecional e persistente para troca de informação. O servidor não é sobrecarregado com requisições e a informação percorre de forma mais fluída (Fette e Melnikov, 2011).

### 2.2.1 *APIs Públicas e Privadas*

As APIs públicas estão disponíveis para o exterior, assim, qualquer utilizador pode consumir os seus dados sem restrições de uso ou acesso (Adkuloo, 2023). No entanto, em alguns casos, é necessário realizar um registo para obter uma chave de utilização. Frequentemente, as empresas disponibilizam APIs públicas para permitir que outras aplicações possam aceder aos seus dados, recursos ou serviços de maneira segura e documentada (ex: Google Maps API).

Uma API privada é uma API usada internamente por uma empresa ou organização, com restrições de uso e acesso. São projetadas para atender às necessidades internas de uma empresa, equipa de desenvolvimento ou sistema (Adkuloo, 2023). Por exemplo, integrar os sistemas internos, como o controlo de *stock* ou a gestão de clientes.

### 2.2.2 *Formatos para troca de dados*

No sentido de possibilitar a comunicação entre diferentes sistemas é necessário realizar a conversão de dados num formato que passa ser transmitido através de uma rede ou armazenado num ficheiro. Os formatos mais utilizados são os seguintes:

- JavaScript Object Notation (JSON)
  Formato simples, rápido e de fácil utilização. Não é escrito em código de máquina, por isso, permite a fácil leitura do seu conteúdo (vegibit, 2023). Os dados são gravados por chave/valor, as chaves representam os nomes dos atributos e os valores, são o conteúdo do objeto (ver listagem 1). Pode ser utilizado em diferentes linguagens de programação e atualmente é um dos formatos mais utilizados nas APIs REST.

```
{
  "curso": "MCIF",
  "nome": "António Pinto"
}
```

Listagem 1: Exemplo JSON

- Extensible Markup Language (XML)
  Semelhante ao HyperText Markup Language (HTML), mas foi concebido para armazenar dados em vez de os exibir. É frequentemente utilizado para enviar dados para a *web*. Na sua estrutura existem *tags*, todas elas têm que ser fechadas (ver listagem 2). Por isso, este formato exige maior capacidade de armazenamento e a leitura, da estrutura, não é tão simples (redundância de dados; vegibit, 2023).

```
<aluno>
    <curso>MCIF</curso>
    <nome>António Pinto</nome>
</aluno>
```

Listagem 2: Exemplo XML

- YAML Ain't Markup Language (YAML)
  Criado para facilitar a leitura de dados. É simples e flexível. Utiliza a estrutura chave/valor referida no JSON (ver listagem 3). Inclui funcionalidades que vêm do Perl, C, XML, HTML e outras linguagens de programação (vegibit, 2023). Deriva do JSON, assim, aceita ficheiros nesse formato.

```
curso: MCIF
nome: António Pinto
```

Listagem 3: Exemplo YAML

- Protocol Buffers (Protobuf)

  Desenvolvido pela Google para ser compacto e eficiente. Não é possível ler o seu conteúdo de forma simplificada, pois, utiliza um formato de codificação binária. No entanto, é utilizado para transmitir dados de forma rápida e compacta. A especificação da estrutura dos dados é realizada num ficheiro, com extensão, *.proto* (ver listagem 4; Souza, 2018).

```
message Aluno {
  string curso = 1;
  string nome = 2;
}
```

Listagem 4: Exemplo de documento .proto

  **Nota:** O exemplo anterior é uma representação textual (.proto) da estrutura dos dados. Esta definição é utilizada para criar classes em diferentes linguagens de programação que podem codificar e descodificar os dados em formato binário Protobuf.

### 2.2.3 *Arquiteturas e protocolos*

Existem diversas arquiteturas e protocolos que podem ser adotados para o desenvolvimento de APIs, contudo as mais utilizadas são as seguintes:

- O Simple Object Access Protocol (SOAP) é um protocolo de comunicação entre diferentes sistemas por meio de um documento XML. Só é compatível com este formato e utiliza bastantes recursos (de rede) no envio de uma mensagem. É um protocolo *stateful*, mas pode funcionar como *stateless*. Funciona por HTTP, SMTP, TCP e UDP. É bom em requisitos de segurança, mas é difícil de desenvolver, pois, é preciso ter uma boa compreensão de todas as regras envolvidas (Reselman, 2023).
- As APIs RESTful utilizam a arquitetura REST. Esta arquitetura permite a comunicação entre diferentes aplicações e sistemas. O protocolo HTTP é utilizado para realizar operações de Create, Read, Update and Delete (CRUD). Foi desenvolvido para resolver algumas das limitações do SOAP(Amazon, 2023). Atualmente, é a arquitetura mais utilizada para desenvolver APIs,

devido à sua simplicidade e flexibilidade (Hat, 2023). Podem ser utilizadas múltiplas linguagens de programação e suporta múltiplos formatos de dados, tais como, JSON, XML, HTML e texto.

Uma arquitetura RESTful (ou REST) deve seguir as seguintes diretrizes (Amazon, 2023; Hat, 2023):

- *Uniform interface*
  Meio uniforme para interagir com um servidor, independentemente do dispositivo ou aplicação;
- *Client/Server*
  Arquitetura cliente servidor;
- *Stateless*
  O servidor não guarda nenhuma informação sobre o utilizador. Todos os dados são guardados no cliente e devem ser informados sempre que seja realizado um novo pedido;
- *Cacheable*
  Isso significa que os dados que o servidor envia, informam se podem ou não ser armazenados em *cache*, assim, é possível melhorar o desempenho;
- *Layered system*
  O cliente deve conseguir solicitar dados de um *endpoint* sem saber sobre a arquitetura do servidor subjacente;
- *Code on demand* (opcional)
  Permite que seja enviado código para o cliente, posteriormente, esse código pode ser executado.

- O GraphQL também é uma arquitetura, a principal diferença entre REST e GraphQL é que este ultimo não lida com recursos dedicados. Em vez disso, tudo é considerado um grafo, portanto, está tudo interligado. Isto significa que se pode alterar o pedido conforme a necessidade exata. O GraphQL tem um bom desempenho, remove automaticamente versões antigas da API e fornece a documentação da mesma. No entanto, apenas suporta JSON, a implementação da *cache* é complexa e a curva de aprendizagem é maior (Reselman, 2023).
- O gRPC Remote Procedure Calls (gRPC) é uma *framework, open source,* desenvolvida pela Google. Como o nome indica, é uma variante da arquitetura Remote Procedure Call (RPC). Permite a comunicação entre diferentes dispositivos utilizando o protocolo HTTP/2. As APIs desenvolvidas em gRPC são concebidas para serem eficientes e leves, e são frequentemente utilizadas em

arquiteturas de microsserviços onde o desempenho é uma preocupação fundamental. A comunicação é feita em binário por questões de desempenho. Para isso é utilizado o formato Protobuf que promete velocidades de transferência muito superiores ao JSON (Reselman, 2023).

### 2.2.4 *Especificações de desenvolvimento*

Tendo em conta a existência de diversas arquiteturas para o desenvolvimento de APIs, este projeto irá focar-se em apenas uma. Neste caso, a arquitetura escolhida será a REST, visto que é amplamente utilizada. No entanto, dadas as suas características, deve existir uma especificação de desenvolvimento para assegurar que a API seja bem documentada, facilmente compreensível e facilmente consumível por outros sistemas e programadores. As especificações mais comuns são as seguintes:

- OpenAPI Specification (OAS) (também conhecido como swagger)
  Formato para descrever uma API REST. Permite especificar os *endpoints* e as operações possíveis para cada um deles. Além disso, informa o processo de autenticação necessário. É a especificação mais utilizada atualmente (OpenAPI, 2021).
- RESTful API Modeling Language (RAML)
  Especificação para descrever APIs RESTful de forma legível, tanto por pessoas como por computadores, ou seja, ela tem uma sintaxe clara e simples que facilita a leitura e a escrita da API. Além disso, a sua estrutura lógica e consistente permite que os computadores processem e validem a API. Isso possibilita verificar se a API está conforme a especificação RAML, se funciona corretamente, se apresenta erros ou problemas de desempenho, além de gerar código, documentação e testes automáticos para a API. Descreve métodos, parâmetros, respostas, recursos e outras condições utilizadas em APIs modernas. Além disso, pode também descrer APIs que não obedecem a todas as restrições REST (*About raml* 2023).
- API Blueprint
  É uma especificação, *open source*, para descrever APIs *web* num formato simples, de fácil leitura e escrita. Baseia-se na sintaxe *Markdown* e é principalmente utilizada na criação de documentação (Testfully, 2021).
- Web Application Description Language (WADL)
  Baseia-se na linguagem XML e utiliza um conjunto de elementos e atributos

para descrever as funcionalidades de uma API. Destina-se a simplificar a reutilização de serviços *web*, baseados na arquitetura HTTP. Descreve parâmetros, *endpoints* e respostas (Hadley, 2009).

## 2.3 SEGURANÇA DAS APIS

Nas secções seguintes serão abordados alguns temas relacionados com a segurança das APIs. Nomeadamente, aspetos como autenticação, autorização e vulnerabilidades mais comuns.

### 2.3.1 *Autorização vs Autenticação*

Um dos aspetos importantes no desenvolvimento de aplicações é a validação da identidade. Assim, a autenticação e a autorização são mecanismos essenciais para garantir a segurança das aplicações.

A autenticação verifica se um utilizador é quem diz ser, enquanto a autorização determina que ações o utilizador tem permissão para realizar na aplicação (Cilwerner, 2023). Em conjunto, estes mecanismos ajudam a impedir o acesso não autorizado a dados sensíveis e a proteger contra atividades maliciosas, tais como, uma violação de dados. Ou seja, sem autenticação e autorização adequadas, as aplicações estão vulneráveis a ataques informáticos.

Assim, para garantir a autenticação de um utilizador, as APIs podem utilizar os seguintes métodos:

- **Nenhum**
  Método inseguro, não exige nenhum processo de autenticação para consumir os dados da API. Esta abordagem é mais utilizada em APIs internas, mas não é uma prática recomendada (Ball, 2022);
- **Basic Authentication**
  Neste processo é fornecido um nome de utilizador e uma senha. Esta informação é adicionada ao *header* ou *body* de cada pedido HTTP (Ball, 2022). Os dados podem ser fornecidos em *plaintext* ou utilizando codificação, tal como, base64. Por exemplo:

  ```
  Authorization: Basic YW50b25pbzppcGwyMDIz=
  ```

No entanto, é um método inseguro, visto que, se alguém capturar o pedido, pode aceder à informação (fácil de ser descodificado). Neste caso, o nome de utilizador e respetiva senha é **antonio:ipl2023**;

- **API Keys**
  Uma chave única é criada para o cliente. Ao solicitar uma requisição, essa chave é introduzida nos pedidos, permitindo, dessa forma, a utilização da API. A chave é uma sequência aleatória de números e letras, útil contra-ataques de *brute force*. Por exemplo:

  ```
  x-Api-key: be1238df05f841d09d27d6caafe081b3
  ```

  Contudo, se forem utilizados protocolos inseguros, um atacante pode capturá-la e aceder à API. Além disso, as chaves podem estar expostas *online* (ex: GitHub; Ball, 2022);

- **JSON Web Token (JWT)**
  *Token* utilizado para autenticação. São enviados os dados de acesso, nome de utilizador e senha, caso estejam corretos é devolvido o *token*. Nos pedidos subsequentes é introduzida esta informação no *header* de *authorization*. Por exemplo:

  ```
  Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
  eyJzdWIiOiIxIiwibmFtZSI6IkFudG9uaW8gUGludG8iLCJpYXQiOjE1MTYyMzkwMjJ9.
  VJmhFXJBckCmtJ5Smns2sk5KRxNemkLv98xolWLTWTU
  ```

  Mesmo sendo um método seguro, pode ser configurado incorretamente. O *token* de exemplo, além de não utilizar encriptação, ou seja, não é mais do que uma *string base64*, que pode ser descodificada para revelar o seu conteúdo original, também utiliza uma chave de assinatura insegura. Isto significa, que com um simples ataque de *brute force* era possível descobrir a senha, isto permitiria criar vários *tokens* válidos, assim, um atacante conseguiria aceder a qualquer conta existente (Ball, 2022);

- **Hash-based Message Authentication Code (HMAC)**
  O HMAC é um código de autenticação de mensagem baseado em *hash* que usa uma chave secreta partilhada, para criar uma assinatura para cada pedido HTTP. Essa assinatura é usada para confirmar se o pedido é proveniente de uma fonte esperada e se não houve alterações durante o pedido. Ou seja, garante a autenticidade e integridade das mensagens, pois, apenas o cliente e

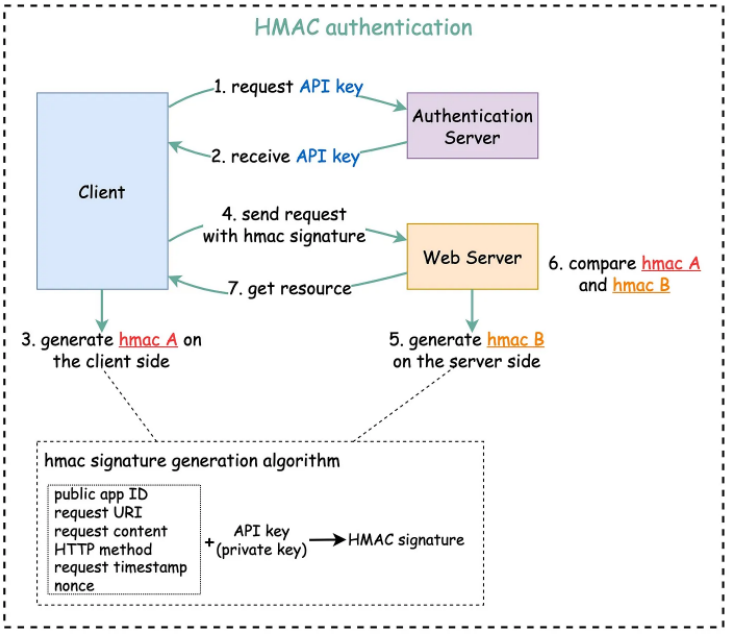


Figura 1: Autenticação HMAC

Fonte: Xu, 2022

o servidor, que possuem a chave partilhada, podem criar e verificar o código HMAC (Rodrigues, 2022). Isso aumenta a segurança da comunicação entre ambas as partes e reduz o risco de ataques maliciosos (ex:Man in the middle (MITM)).

Como é possível observar na figura 1, funciona da seguinte forma (Xu, 2022):

1-2. O cliente e o servidor acordam numa chave partilhada e secreta que será utilizada para gerar o HMAC. Esta chave deve ser mantida em segredo entre as partes envolvidas.

3. O cliente cria uma mensagem que será enviada ao servidor. Posteriormente, cria um código HMAC aplicando um algoritmo de função *hash* à mensagem e usando a chave partilhada.

4. O código HMAC é incluído na mensagem enviada ao servidor.

5. Ao receber a mensagem, o servidor recalcula o código HMAC usando a mesma chave e a mensagem recebida.

6. O servidor compara o código HMAC criado localmente com o código HMAC recebido do cliente.

7. Se ambos os códigos HMAC forem iguais, a mensagem é autêntica e não foi alterada durante a transmissão. O servidor processa a mensagem. Caso contrário, a mensagem é considerada inválida e rejeitada.

- **Open Authorization (OAuth)**

  O OAuth é um protocolo de autorização utilizado em aplicações *web* e APIs para permitir que os utilizadores concedam acesso a informações protegidas para outras aplicações, limitadamente, sem a necessidade de partilhar diretamente os seus dados de acesso (OAuth, 2023).

  Permite que um utilizador autorize uma aplicação (cliente) a aceder a recursos protegidos no seu nome, através da obtenção de um *token* de acesso. Esse *token* é usado pela aplicação cliente para fazer solicitações à API em nome do utilizador autenticado.

  Para utilizar este protocolo é necessário escolher um tipo de *grant*, ou seja, é o mecanismo que define como uma aplicação cliente obtém um *token* de acesso para aceder aos recursos protegidos. Cada tipo de *grant* especifica o fluxo e os passos necessários para autenticação e autorização da aplicação cliente (OAuth, 2023).

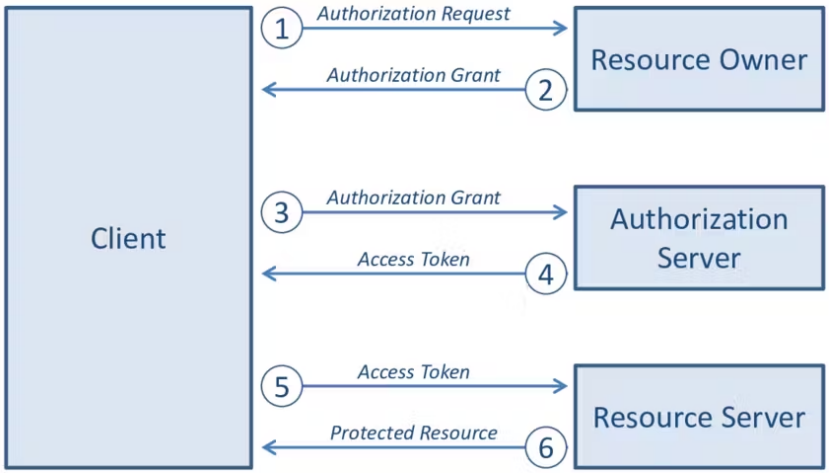


Figura 2: Autorização OAuth

Fonte: MODY, 2022

A figura 2, demonstra o fluxo típico de acesso (MODY, 2022).

1. O cliente envia um pedido de autorização ao proprietário do recurso (utilizador) para aceder a certos dados específicos.
2. O utilizador fornece acesso ao cliente.
3. O cliente envia o código de autorização para o servidor de autenticação.

4. O servidor de autenticação verifica a validade do código, se estiver correto, emite um *token* de acesso para o cliente.
5. O cliente utiliza o *token* para realizar pedidos aos recursos protegidos da API.
6. A API verifica se o *token* de acesso é valido, se for, fornece os dados solicitados ao cliente.

O OAuth é um protocolo seguro e amplamente utilizado, no entanto, se for implementado incorretamente, podem surgir várias falhas de segurança que podem comprometer a integridade e a privacidade dos utilizadores, tais como, falhas na gestão dos *tokens*, ataques de *phishing*, Cross Site Request Forgery (CSRF), problemas de redirecionamento, entre outros (Ball, 2022).

### 2.3.2 *Vulnerabilidades*

Há diversos tipos de ameaças que uma API está sujeita. Um estudo, realizado em 2021, intitulado por «Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study» (ver referências) identificou que existem sessenta e seis ameaças. Destas, dezassete delas estão relacionadas com a comunicação (HTTP) utilizada pela API para enviar e receber dados. Também foi identificado que vinte e uma ameaças dizem respeito a ataques de negação de serviço (DOS). As restantes vinte e oito ameaças são consideravelmente mais específicas, razão pela qual não foram sujeitas a categorização.

Este projeto (ZAPI) irá basear-se no OWASP API Security Project, no qual são identificadas as dez vulnerabilidades mais comuns em APIs. É uma organização sem fins lucrativos, o seu objetivo recai sobre a divulgação de material gratuito para todos conseguirem melhorar a segurança. O documento final é o resultado de um consenso entre especialistas de segurança de várias países. É utilizado um sistema de pontuação que tem em conta quatro aspetos, sendo estes a sua explorabilidade, prevalência, detetabilidade e impacto técnico, cada um dos quais pode ter uma classificação de um a três, sendo três o mais crítico e um, o menos crítico. Para cada uma das vulnerabilidades, fornecem uma descrição dos possíveis vetores de ataque, as falhas de segurança, os impactos que terá se for explorado, as razões pela qual uma API é vulnerável, exemplos de cenários de ataque e conselhos práticos de prevenção.

É composto pelas seguintes vulnerabilidades (*Owasp top 10 API security risks* 2023; P. Silva e R. Silva, 2019):

1. ***Broken Object Level Authorization***

   Os utilizadores conseguem aceder à informação de outros utilizadores sem a devida autorização. Não é realizada a devida validação, se quem deve aceder a determinada informação realmente tem permissão para o fazer.

   **Ataques conhecidos:** Pouya Darabi descobriu uma vulnerabilidade (2019) na API do Facebook que lhe permitia criar publicações nas páginas de outros utilizadores (Pouya, 2020).


Figura 3: Facebook BOLA

Fonte: Pouya, 2020

   A vulnerabilidade ocorria ao manipular o parâmetro *page_id* para o *id* da página de destino pretendida (figura 3).

2. ***Broken Authentication***

   Falhas na implementação da autenticação. O atacante pode assumir a identidade de outros utilizadores e aceder a dados sensíveis.

   **Ataques conhecidos:** A Peloton, empresa de equipamentos de ginástica, não tinha autenticação em vários *endpoints*, incluindo os com detalhes de treino e detalhes do utilizador (Isbitski, 2021).

API Request:

```
POST /stats/workouts/details HTTP/1.1
Host: api.onepeloton.co.uk
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0)
Gecko/20100101 Firefox/84.0
Accept: application/json, text/plain, */*
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 1794
Origin: https://members.onepeloton.co.uk
DNT: 1
Connection: close
Referer: https://members.onepeloton.co.uk/player/live/xxxREDACTEDxxx
Cookie: __cfduid=x; driftt_aid=x; DFTT_END_USER_PREV_BOOTSTRAPPED=true;
peloton_session_id=x

{"ids":["x","x","x","x","x","x",]}
```

Figura 4: Peloton Broken Authentication

Fonte: Isbitski, 2021

Qualquer indivíduo com acesso à *internet* poderia consultar diretamente as APIs da Peloton e obter informações pessoais dos utilizadores (figuras 4,5).

API Response


```
"ae[redacted]":{
  "age":41,
  "age_group":"40s",
  "avatar":"https://s3.amazonaws.com/peloton-profile-images/[redacted]
  "gender":"female",
  "short_gender":"F",
  "is_birthday":false,
  "is_digital":false,
  "is_profile_private":true,
  "location":"Maryland",
  "nth_workout":2,
  "start_time":1612526841,
  "user_id":"6[redacted]",
  "username":"Stj[redacted]",
  "workout_id":"ael[redacted]",
  "is_live":true,
  "bike_number":null,
  "is_studio":false,
  "peloton_id":"a6[redacted]",
  "peloton_start_time":1612525884,
  "tags_info":{
    "primary_name":"[redacted]",
    "total_joined":3
  },
  "authed_user_follows":null
```


Figura 5: Peloton Broken Authentication - Continuação

Fonte: Isbitski, 2021

3. ***Broken Object Property Level Authorization***

Falta de validação de autorização adequada ou incorreta ao nível das propriedades do objeto. Isso resulta na exposição ou manipulação de informações por partes não autorizadas (Excessive Data Exposure e Mass Assignment).

**Ataques conhecidos:** Em 2022, Laurence Tennant descobriu uma vulnerabilidade de Mass Assignment (para cada chave do pedido, neste contexto, certificados, atribuiu o valor *true*, conforme indicado na figura 6), no projeto freeCodeCamp, onde conseguiu adquirir todas as certificações de programação, que supostamente representam mais de 6000 horas de estudo, num único pedido (Tennant, 2022).

```
PUT /update-user-flag HTTP/2
Host: api.freecodecamp.dev
Cookie: _csrf=lsCzfu4[...]
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://www.freecodecamp.dev/
Csrf-Token: Tu0VHrwW-GJvZ4ly1sVEXjHxSzgPLLj99OLQ
Content-Type: application/json
Origin: https://www.freecodecamp.dev
Content-Length: 518
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-site
Te: trailers

{
  "name": "Mass Assignment",
  "isCheater": false,
  "isHonest": true,
  "isInfosecCertV7":true,
  "isApisMicroservicesCert":true,
  "isBackEndCert":true,
  "is2018DataVisCert":true,
  "isDataVisCert":true,
  "isFrontEndCert":true,
  "isFullStackCert":true,
  "isFrontEndLibsCert":true,
  "isInfosecQaCert":true,
  "isQaCertV7":true,
  "isInfosecCertV7":true,
  "isJsAlgoDataStructCert":true,
  "isRelationalDatabaseCertV8":true,
  "isRespWebDesignCert":true,
  "isSciCompPyCertV7":true,
  "isDataAnalysisPyCertV7":true,
  "isMachineLearningPyCertV7":true
}
```

Figura 6: freeCodeCamp BOPLA

Fonte: Tennant, 2022

4. ***Unrestricted Resource Consumption***

Para atender aos pedidos da API, é necessário ter recursos como rede, CPU, memória e armazenamento. Outros recursos como mensagens eletrónicas, SMS ou chamadas telefónicas são disponibilizados pelos fornecedores de serviços por integrações com a API, normalmente, são pagos por pedido.

Ataques bem sucedidos podem levar a um Denial Of Service (DOS) ou a um aumento dos custos operacionais, ou seja, não existe limite para consumir os dados.

**Ataques conhecidos:** Em 2020, a equipa de investigação da Checkmarx descobriu que o SoundCloud não aplicava *rate limiting* ao *endpoint /tracks* da API api-v2.soundcloud.com. Dado que não eram realizadas validações para o número de *IDs* informados, um atacante poderia sobrecarregar o servidor num único pedido (figura 7).

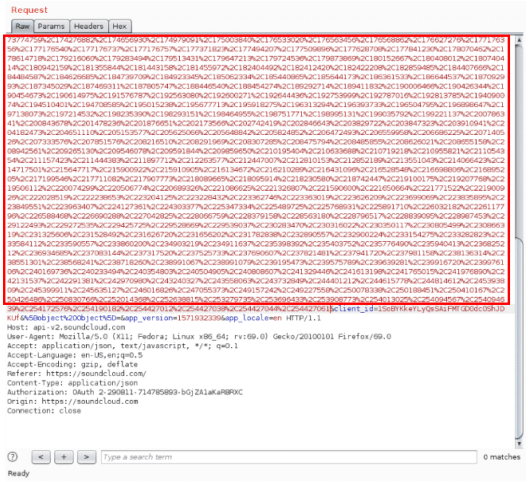


Figura 7: SoundCloud Unrestricted Resource Consumption

Fonte: Checkmarx, 2021

O investigador conseguiu obter 689 músicas num único pedido, aumentando o tempo de resposta do serviço em quase 9 vezes (figura 8). A vulnerabilidade poderia ser explorada por um ataque de Distributed Denial Of Service (DDoS), tornando os serviços indisponíveis (Checkmarx, 2021).

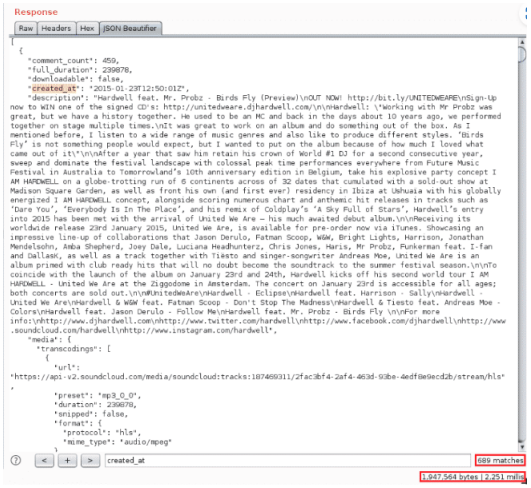


Figura 8: SoundCloud Unrestricted Resource - Continuação

Fonte: Checkmarx, 2021

5. ***Broken Function Level Authorization***

Os atacantes podem ganhar acesso a recursos de outros utilizadores e realizar ações sem a devida autorização. A figura 9 é um exemplo desta vulnerabilidade. Neste caso, um utilizador, sem permissões, consegue aceder a vários *endpoints* de administração. Assim, consegue realizar várias ações sem a devida permissão.

**Ataques conhecidos:** Em 2018, Jon Bottarini descobriu que um utilizador (sem privilégios) podia efetuar alterações em alertas nos monitores, New Relic Synthetics. A exploração envolvia o envio de um pedido legítimo para um *endpoint* da API que, por norma, só seria visível para a administração (Bottarini, 2018).

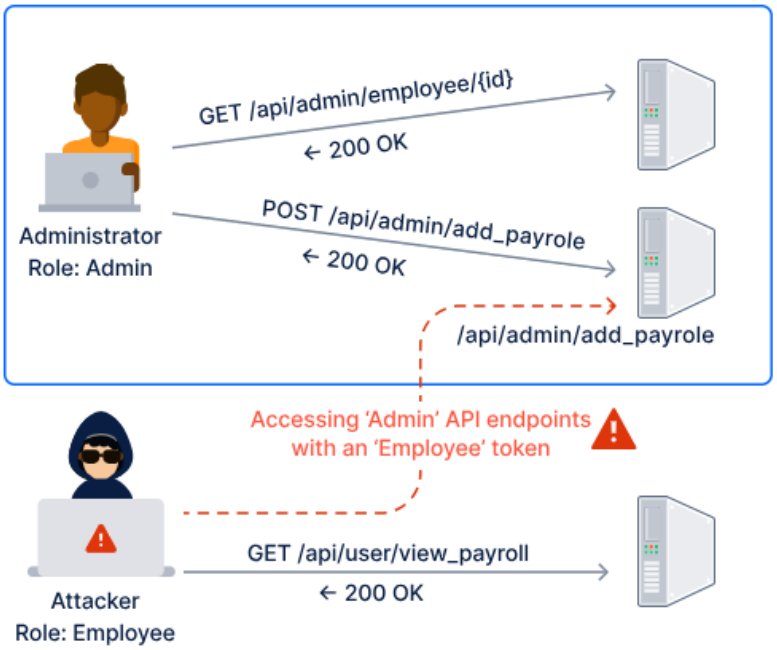


Figura 9: BFLA

Fonte: Levo, 2023

6. ***Unrestricted Access to Sensitive Business Flows***

Falha na proteção dos fluxos de negócio sensíveis, tais como pagamentos, transferências, aprovações, entre outros. Dessa forma, é possível que os atacantes possam interferir ou abusar destes fluxos.

**Ataques conhecidos:** Pessoas que tentaram adquirir bilhetes para a última digressão de Taylor Swift foram "obrigadas" a comprar ingressos no mercado de revenda a preços inflacionados. Isto porque, a Ticketmaster declarou que sofreu um ataque de *bots*. Em outras palavras, um ataque automatizado para a aquisição de inúmeros bilhetes visando a revenda (Faguy, 2023).

7. ***Server Side Request Forgery***

Os atacantes podem induzir a API a fazer pedidos maliciosos a outros sistemas internos ou externos por meio de uma vulnerabilidade de Server Side Request Forgery (SSRF).

**Ataques conhecidos:** Em 2023, a API do Microsoft Azure continha duas

vulnerabilidades de SSRF. Ao manipular alguns *headers* (figura 10) os atacantes conseguiam aceder a serviços internos, realizar DOS e contornar *firewalls* (Matan, 2023).


Figura 10: Microsoft Azure SSRF

Fonte: Matan, 2023

8. ***Security Misconfiguration***

Implementações erradas em termos de segurança, tais como, mensagens de erro com informação sensível, Cross-origin Resource Sharing (CORS) mal implementado, métodos HTTP não utilizados, entre outros.

**Ataques conhecidos:** Uma falha na API da Uber, em 2019, permitia acessos indevidos às contas dos utilizadores (Prakash, 2019). O Universal Unique Identifier (UUID) de qualquer utilizador, da Uber, era obtido ao fornecer o número de telefone ou endereço eletrónico noutro *endpoint* da API, este era devolvido na mensagem de erro.

**Request**

```
POST /p3/fleet-manager/\_rpc?rpc=addDriverV2 HTTP/1.1
Host: partners.uber.com
{“nationalPhoneNumber”:”99999xxxxx”,”countryCode”:”1"}
```

**Response**

```
{
    “status”:”failure”,
    ”data”: {
        “code”:1009,
        ”message”:”Driver ‘47d063f8-0xx5e-xxxxx-b01a-xxxx’ not found”
        }
}
```

Figura 11: Uber Security Misconfiguration

Fonte: Prakash, 2019

Ao fornecer o UUID do utilizador no pedido à API e ao utilizar o *token* divulgado na resposta era possível controlar qualquer conta (figuras 11,12).


```
{
   “status”:”success”,
   ”data”:{
      “data”:{
         “language”:”en”,
         ”userUuid”:”xxxxxx1e”
      },
      ”getUser”:{
         “uuid”:”cxxxxxc5f7371e”,
         ”firstname”:”Maxxxx”,
         ”lastname”:”XXXX”,
         ”role”:”PARTNER”,
         ”languageId”:1,
         ”countryId”:77,
         ”mobile”:null,
         ”mobileToken”:1234,
         ”mobileCountryId”:77,
         ”mobileCountryCode”:”+91",
         ”hasAmbiguousMobileCountry”:false,
         ”lastConfirmedMobileCountryId”:77,
         ”email”:”xxxx@gmail.com”,
         ”emailToken”:”xxxxxxxx”,
         ”hasConfirmedMobile”:”no”,
         ”hasOptedInSmsMarketing”:false,
         ”hasConfirmedEmail”:true,
         ”gratuity”:0.3,
         ”nickname”:”abc@gmail.com”,
         ”location”:”00000”,
         ”banned”:false,
         ”cardio”:false,
         ”token”:”b8038ec4143bb4xxxxxx72d”,
         ”fraudScore”:0,
         ”inviterUuid”:null,
         ”pictureUrl”:”xxxxx.jpeg”,
         ”recentFareSplitterUuids”:[
            “xxx”
         ],
```


Figura 12: Uber Security Misconfiguration - Continuação

Fonte: Prakash, 2019

9. ***Improper Inventory Management***

Quando existe uma má gestão no ciclo de vida de desenvolvimento da API (versões). Os atacantes podem conseguir acesso à informação sensível ou até obter o controlo do servidor ao explorar uma versão anterior com menos

controlos de segurança implementados.

**Ataques conhecidos:** O Justdial, serviço de pesquisa da Índia, sofreu uma violação de dados, em 2019, que expôs os dados de mais de 100 milhões de utilizadores, incluindo nome, endereço eletrónico, número de telemóvel, género, data de nascimento, fotografia, empresa, entre outros. Quando a empresa redesenhou as suas aplicações, a antiga API foi deixada em funcionamento, sem proteção, e com acesso à base de dados dos utilizadores (Kumar, 2019).

10. ***Unsafe Consumption of APIs***

Os programadores tendem a confiar mais nos dados recebidos de APIs de terceiros do que nas entradas dos utilizadores, o que os leva a adotar padrões de segurança menos robustos. Para comprometer as APIs, os atacantes concentram-se nos serviços de terceiros que estão integrados, em vez de tentarem comprometer diretamente a API alvo.

**Ataques conhecidos:** A vulnerabilidade Log4Shell permitiu que os atacantes executassem código arbitrário em quase todos os serviços *web* que usam a biblioteca Apache Log4j. Devido ao consumo inseguro de APIs no Log4j, mais especificamente pela capacidade de reconstruir a estrutura dos dados fornecidos pelo utilizador sem a devida validação (Best, 2023). Os atacantes poderiam explorar essa vulnerabilidade enviando pedidos contendo código malicioso que, posteriormente, poderia ser executado no servidor (figura 13).

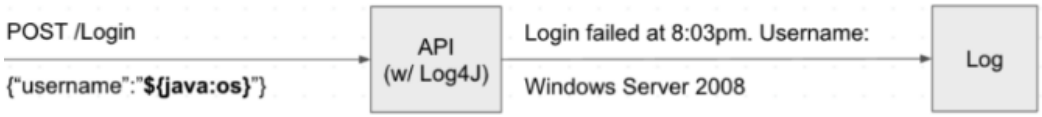


Figura 13: Log4J Unsafe Consumption of APIs

Fonte: Shkedy, 2021

### 2.3.3 *Diferenças entre segurança em APIs e web*

Nas aplicações *web* tradicionais a informação é renderizada pelo servidor e posteriormente enviada para o navegador do utilizador (cliente). Por esse motivo, existem poucos parâmetros que podem ser abusados e a proteção pode ser reali-

zada com uma Web Application Firewall (WAF). As vulnerabilidades do OWASP Top 10 para *Web* surgiram neste contexto (Eduardo, 2022; GoCache, 2022). No entanto, as APIs mudaram esse cenário. Elas enviam e recebem dados do servidor *backend* e agora é o cliente que renderiza a informação obtida. Assim, a superfície de ataque aumenta significativamente (Apisecurity.io, 2019). Apesar de existirem vulnerabilidades em comum com aplicações *web*, como *injection*, as APIs são um caso muito específico. Considerando que as principais ferramentas de análise de vulnerabilidades e segurança têm tradicionalmente como alvo as aplicações *web* convencionais, é notório que a sua eficácia é limitada. Este enfoque, centrado em cenários de aplicações *web* convencionais, não abrange adequadamente as especificidades e desafios associados às APIs (Eduardo, 2022; GoCache, 2022). Como tal, surge a necessidade de desenvolver soluções mais abrangentes e adaptáveis, capazes de abordar as crescentes ameaças em APIs. Um exemplo disso são as WAFs.

Os WAFs detetam o tráfego malicioso mediante regras sobre o tráfego HTTP, ou seja, guiam-se no conteúdo das *strings* capturadas. Contudo, é preciso verificar o objetivo do pedido (contexto). Logo, como os WAFs não compreendem o contexto, não detetam falhas em APIs, tais como, autorização, *mass assignment*, *excessive data exposure*, entre outros (Hardcastle, 2021). Por exemplo, um utilizador não deve conseguir aceder aos dados de outro. Uma WAF não iria bloquear a requisição, caso um pedido fosse realizado por alguém não autorizado. Assim, uma simples WAF já não é uma solução eficaz porque não consegue distinguir o tráfego fidedigno do malicioso.

Dadas as limitações existentes surgiu a Web Application and API Protection (WAAP) (evolução de um WAF), projetada para proteger aplicações *web* e APIs utilizando inteligência artificial e *machine learning* (Paloalto, 2023).

### 2.3.4 *Metodologias de testes de segurança*

A análise de vulnerabilidades em APIs pode ser realizada recorrendo a várias metodologias, são elas:

- **Black Box**

  É um método de teste onde o *pentester* não tem acesso ao código da implementação da API, e em vez disso concentra-se nos pedidos e respostas da mesma (Malik, 2023). Este tipo de teste é útil para assegurar se a API funciona como esperado e cumpre os requisitos estabelecidos na documentação.

No entanto, nem sempre existe documentação pública e não há nenhum acesso (autenticação) cedido ao *pentester*.

- **White Box**

  Neste caso, existe acesso à implementação da API, assim como a métodos de autenticação válidos. É um teste focado na lógica interna da API e é útil para identificar *bugs* e erros lógicos. Por isso, é mais fácil ser realizado por programadores, uma vez que estes têm conhecimento do código e da arquitetura do sistema (Malik, 2023). Ou seja, é útil para testar todo o ciclo de desenvolvimento da API.

- **Grey Box**

  Método que une características tanto *black box* como *white box*. O *pentester* ou programador tem acesso a alguns dados, mas não a todos, da implementação da API. Este tipo de teste é útil para identificar problemas com a implementação, enquanto avalia também a API numa perspetiva externa (Malik, 2023). Logo, é um teste bastante robusto e completo.

### 2.3.5 *Problemas e limitações atuais*

Uma vez que a API lida com dados sensíveis, é necessário realizar testes de segurança como em qualquer outro *software*. No entanto, a API apresenta a particularidade de não exibir Graphical User Interface (GUI), ou seja, não existem botões e *inputs* de texto. Por isso, todos os *endpoints* existentes devem ser percorridos para testar todos os *headers*, parâmetros e os respetivos corpos da mensagem HTTP (*body*). Assim sendo, os testes realizados são muito mais técnicos e com nível de *know how* elevado (Agarwal, 2020; Sharieh e Ferworn, 2021).

Atualmente o processo de identificação de vulnerabilidades é realizado de forma manual, segundo o artigo elaborado por Ehsan et al. (2022) deve-se pelos seguintes fatores:

- **Falta da adoção de um padrão para descrever APIs REST**

  Existe flexibilidade na metodologia usada para desenvolver, implantar e consu-

mir APIs RESTful. Para abranger todos os casos é preciso existir um padrão do documento da API RESTful, por exemplo, OAS. Assim, uma aplicação automatizada poderia ser compatível com qualquer metodologia usada;

- **Várias dependências de parâmetros**

  As APIs apresentam dependências que tem que ser satisfeitas, caso contrário uma mensagem de erro é apresentada, por exemplo, *bad request* (Mirabella et al., 2021). Ou seja, existem combinações de parâmetros que formam pedidos válidos ou inválidos. Existe uma hierarquia que é necessária ser seguida. Um pedido à API pode necessitar de outro pedido GET anterior e assim por diante. A título de exemplo, a API do PayPal requer um objeto JSON composto por mais de 200 parâmetros de modo a criar uma fatura (*PayPal Invoices API* 2023). É importante conseguir ultrapassar esta limitação, pois, sem acesso ao código torna-se inviável descobrir todas as dependências, e assim a análise será limitada e duvidosa (Martin-Lopez et al., 2021). Basta um *endpoint* estar vulnerável para comprometer uma organização;

- **Autenticação**

  Numa perspetiva *black box*, ou seja, sem existir acesso ao código-fonte da API e se o processo de autenticação estiver bem implementado, a autenticação torna-se um problema. Este problema deve-se ao facto de existirem diversos métodos de autenticação e sem esta é impossível de testar o *endpoint*. Ao ignorar os testes onde a autenticação está bem implementada, resultaria em resultados não confiáveis e com pouca cobertura aplicacional (Ehsan et al., 2022).

A aplicação a ser desenvolvida, além de pretender contribuir com uma solução gratuita, tem como principal finalidade a deteção das 10 vulnerabilidades mais comuns em APIs utilizando automatização. Considerando os desafios e limitações mencionados anteriormente, a aplicação empregará estratégias para garantir que essas limitações não comprometam os resultados. Estas estratégias incluem a adoção do padrão OAS bem documentado, para evitar dependências de parâmetros, bem como a opção de introdução de chaves de autenticação por parte do utilizador, caso sejam necessárias para analisar o *endpoint*.

# 3

# ESTADO DA ARTE

Atualmente estão disponíveis algumas ferramentas, gratuitas, para testar APIs RESTful. Estas ferramentas auxiliam o programador ou *pentester* a identificar falhas mais facilmente. O foco desta comparação será apenas entre ferramentas gratuitas. As ferramentas analisadas são as seguintes:

- Fuzzapi [1]
  É uma ferramenta de análise de vulnerabilidades em APIs. No entanto, não é automatizada. É necessário informar todos os *endpoints* a ser testados, os métodos a utilizar, *headers* e respetivos parâmetros. Neste momento é um projeto obsoleto, ao deixar de ser atualizado desde 2020;
- Automated Security Testing For REST API's (ASTRA) [2]
  Ferramenta desenvolvida para identificar falhas de segurança durante o processo de desenvolvimento de APIs. Permite identificar as seguintes vulnerabilidades:
  - *SQL injection (OWASP API10);*
  - *Cross site scripting (OWASP API10);*
  - *Information Leakage (OWASP API3);*
  - *Broken Authentication and session management (OWASP API2);*
  - *CSRF (incluindo Blind CSRF, OWASP API8);*
  - *Rate limit (OWASP API4);*
  - *CORS (incluindo tecnicas de bypass, OWASP API8);*
  - *JWT attack (OWASP API2);*
  - *CRLF (OWASP API10);*
  - *Blind XXE injection (OWASP API10)*
  - *Server-side Requrest Forgery (OWASP API7)*
  - *Template Injection (OWASP API10)*

1 https://github.com/Fuzzapi/fuzzapi
2 https://github.com/flipkart-incubator/Astra

Esta ferramenta segue a mesma abordagem que a Fuzzapi, ou seja, é necessário informar o URL com o respetivo *endpoint*, método a utilizar (ex: POST), *headers* e o *body*.

Em 2023, foi atualizada para permitir a utilização de uma coleção postman, ou seja, já não era necessário informar todos os *endpoints* manualmente. No entanto, apesar de ser uma ferramenta com várias funcionalidades, não demonstrou resultados satisfatórios durante os testes realizados. Mais concretamente, uma API vulnerável foi utilizada com o intuito de visualizar os resultados da aplicação, contudo, estes revelaram-se praticamente inexistentes, em adição à instabilidade da ferramenta;

- RESTler [3]
  Ferramenta para testar APIs REST na *cloud*. Não é uma ferramenta para encontrar vulnerabilidades, ela foca-se em fornecer dados aleatórios, inválidos ou inesperados como parâmetros para avaliar o comportamento da API, ou seja, é uma ferramenta de *fuzzing*. A ferramenta recebe um documento OAS e a partir desta especificação gera os testes;

- RESTest [4]
  *Framework* para realizar testes em APIs RESTful no contexto *black box*. Segue uma abordagem baseada em modelos, onde os casos de teste são gerados automaticamente pela OAS da API em teste. O seu principal foco é encontrar *bugs* e não vulnerabilidades;

- RESTTESTGEN [5]
  É uma ferramenta para realizar testes *black box* em APIs REST. À semelhança de algumas ferramentas descritas anteriormente, também utiliza a OAS como ficheiro de *input*. Este documento é necessário para obter informações sobre os *endpoints* disponíveis, as operações permitidas e o formato dos dados introduzidos. No entanto, tal como a ferramenta anterior, não foi desenvolvida com o intuito de encontrar vulnerabilidades;

- Kiterunner [6]
  Uma técnica muito utilizada para descobrir conteúdo *web* é a realização de ataques de *brute force*, com dicionários, contendo os nomes mais típicos de documentos ou pastas. O Kiterunner foi desenvolvido como uma ferramenta para descoberta de conteúdo, mas focado em APIs, visto que as ferramen-

3 https://github.com/microsoft/restler-fuzzer
4 https://github.com/isa-group/RESTest.git
5 https://github.com/SeUniVr/RestTestGen.git
6 https://github.com/assetnote/kiterunner.git

tas tradicionais não eram muito eficientes. Assim, esta ferramenta consegue descobrir vários *endpoints*;

- Arjun [7]
  Ferramenta, *open source*, para descobrir parâmetros de aplicações. Útil para encontrar parâmetros em endpoints das APIs, mesmo os escondidos propositadamente;
- Burp Suite
  É um *software* para a realização de testes de segurança em aplicações *web*. Bastante completo e robusto, permite assim encontrar vulnerabilidade de forma mais fácil. Útil para utilizar com outras ferramentas de modo a testar as APIs corretamente;
- OWASP ZAP [8]
  É uma ferramenta semelhante ao Burp Suite, mas totalmente gratuita e *open source*. Permite a identificação de vulnerabilidades em aplicações *web*. Está disponível para Windows, Linux e macOS;
- Postman
  *Software* para os programadores projetar, construir e testar APIs. Suporta a criação de variáveis globais, coleções e testes específicos para cada pedido. Assim, permite visualizar se a API funciona como é suposto e facilita a correção de erros de implementação;
- Automatic API Attack Tool [9]
  Ferramenta que permite gerar e executar ataques com base numa especificação OAS. A ferramenta analisa a especificação da API, cria cenários de ataque com base nela e envia pedidos para os *endpoints*, reportando resultados detalhados de sucesso ou falha. Pode também ser estendida para incluir vetores de ataque como XSS, SQLi e RFI. Além disso, permite a integração com fluxos de trabalho Continuous Integration/Continuous Delivery (CI/CD). No entanto, é um projeto obsoleto, pois já não é atualizado desde 2019;
- openapi3-fuzzer [10]
  É uma ferramenta de *fuzzing* compatível com a OAS 3. Envia vários tipos de *payloads* para todos os *endpoints* definidos num ficheiro de especificação OAS. Posteriormente, verifica se as respostas obtidas correspondem às definições especificadas no ficheiro, e encerra a execução caso não haja correspondência.

7 https://github.com/s0md3v/Arjun.git
8 https://github.com/zaproxy/zaproxy
9 https://github.com/imperva/automatic-api-attack-tool
10 https://github.com/vwt-digital/openapi3-fuzzer

Assim como se observou em exemplos prévios, esta ferramenta pertence à categoria de *fuzzing* e não se concentra na deteção de vulnerabilidades conforme as diretrizes da OWASP;

- APIClarity [11]
  É uma ferramenta de segurança em APIs, *open source*, com foco na especificação de desenvolvimento OAS. Funciona ao capturar e analisar o tráfego das APIs num ambiente Kubernetes, identificando potenciais riscos como APIs sombra (não estão oficialmente documentadas), zumbi (são obsoletas, mas continuam em execução), discrepâncias de especificações, exposição de dados sensíveis, entre outros. Algumas das características chave da ferramenta APIClarity incluem:
  - Reconstrução automática da especificação OAS com base no tráfego da API observado;
  - Teste ativo dos *endpoints* da API para deteção de problemas de segurança na implementação;
  - Compara o tráfego capturado da API com a especificação OAS fornecida pelo utilizador. Isto permite identificar APIs sombra, zumbi e diferenças entre o tráfego capturado versos o especificado;
  - Fornecimento de um painel para monitorizar e alertar sobre questões de segurança.
- Akto [12]
  É uma plataforma de segurança, *open source*, para APIs. É utilizada por equipas de segurança para manter um inventário de APIs, testar vulnerabilidades e detetar problemas em tempo real. O Akto abrange categorias como o OWASP Top 10 e o HackerOne Top 10. Promete efetuar testes avançados que envolvem análises da lógica do negócio, reduzindo falsos positivos através da análise de padrões de tráfego. O Akto pode ser integrado com várias fontes de tráfego, como burpsuite, AWS, postman, GCP e gateways. Em suma, a plataforma apresenta estas características:
  - Permite realizar um inventário das APIs;
  - Executa testes de lógica em CI/CD;
  - Identifica vulnerabilidades em tempo real.

11 https://github.com/openclarity/apiclarity
12 https://github.com/akto-api-security/akto

- pynt [13]
  Pynt (pynt.io) é uma solução gratuita de testes de segurança em APIs que visa trazer segurança desde a fase de desenvolvimento. O Pynt executa ataques nas APIs para identificar problemas críticos. Integra-se nas ferramentas existentes de desenvolvimento e nos fluxos de trabalho CI/CD, proporcionando flexibilidade na sua utilização. Pode ser utilizado através da aplicação Postman, do invólucro Newman CLI ou diretamente via terminal. A solução afirma que abrange as 10 principais vulnerabilidades em APIs, definidas pela OWASP;
- metlo [14]
  O metlo é uma ferramenta de segurança em APIs, *open source*, que permite criar um inventário dos *endpoints*, identifica comportamentos maliciosos por parte dos utilizadores e bloquear, em tempo real, o tráfego malicioso.

  Em suma, tem as seguintes características:

  - Identifica atividades maliciosas em tempo real;
  - Bloqueia automaticamente utilizadores maliciosos para prevenir danos;
  - Cria um registo completo dos *endpoints*;
  - Realiza testes preventivos antes da API entrar em produção;
  - Avalia *endpoints* com dados pessoais e atribui riscos;
  - Identifica vulnerabilidades como *endpoints* não autenticados e com dados sensíveis expostos;
  - Integra-se com fluxos de trabalho de CI/CD;
  - Utiliza algoritmos para identificar desvios no comportamento normal;
  - Fornece contexto sobre ataques para corrigir rapidamente as vulnerabilidades.

A tabela 1 fornece um resumo das opções viáveis disponíveis. Uma alternativa viável é aquela que apresenta automação, que consiga identificar vulnerabilidades, possibilitando testar diversos *endpoints* de forma simultânea, e que seja possível executar a ferramenta num simples computador, sem a necessidade de configurar uma infraestrutura para esse fim. Foi elaborada para facilitar a escolha das melhores opções disponíveis tendo em conta as seguintes características:

13 https://www.pynt.io/
14 https://github.com/metlo-labs/metlo

- **Compatibilidade:** Esta coluna destaca qual é o ficheiro de entrada necessário para a aplicação funcionar adequadamente;
- **Automatização:** Identifica se a ferramenta em questão é automatizada;
- **Identificação de vulnerabilidades:** Avalia se a aplicação consegue identificar vulnerabilidades;
- **Operacionalidade:** A última coluna analisa se a aplicação está total ou parcialmente operacional.

| Ferramenta | Compatibilidade | Automatizada? | Identifica vulnerabilidades? | Está operacional? |
|---|---|---|---|---|
| pynt | Coleção Postman | ✓ | ✓ | ✓ |
| Akto | Coleção Postman | ✓ | ✓ | ✓ |
| ASTRA | Coleção Postman | ✓ | ✓ | ✗ Testes com coleções Postman não operacionais |
| APIClarity | OAS | ✓ | ✓ | ✓, mas está projeteda para funcionar apenas num cluster Kubernetes |
| metlo | OAS | ✓ | ✓ | ✗ Versão beta com funcionalidades limitadas |
| OWASP ZAP | OAS | ✓ | ✓, mas focada em *web* | ✓ |

Tabela 1: Resumo das ferramentas

Em resumo, este capítulo permite observar que há uma escassez de soluções gratuitas para a realização de testes (análise de vulnerabilidades) em APIs (Tesauro, 2023). Entre as mencionadas, destacam-se o Astra, pynt, Akto e metlo. No entanto, é importante notar que o Astra, apesar das atualizações, não está completamente funcional devido à incapacidade de utilizar coleções do Postman (dblk4, 2023). Adicionalmente, as restantes ferramentas são relativamente recentes, com menos de oito meses de existência. Neste contexto, o metlo, embora promissor, ainda não está totalmente operacional, já que a funcionalidade de análise de vulnerabilidades encontra-se em fase beta e não está disponível para uso. Consequentemente, restam apenas as ferramentas Akto e pynt como opções viáveis.

# 4

# DESENVOLVIMENTO

Os autores do artigo «Automated API Testing» afirmam que os testes manuais são um processo muito demorado e exaustivo. Existe um elevado risco de ocorrerem erros e torna-se inviável testar APIs com abundância de *endpoints*. Ou seja, não são adequados para grandes organizações e projetos com poucas horas laborais. Por isso, o teste automatizado não só é um processo mais rápido, como também reduz as hipóteses de erros.

Assim sendo, e concordando com o descrito anteriormente, foi implementada uma solução que utiliza uma abordagem hibrida, isto é, automatizada e manual. Tal como foi referido na introdução, esta solução gratuita utiliza automatização para detetar as dez vulnerabilidades mais comuns em APIs. Além disso, está projetada para uma metodologia *greybox*, devido aos problemas e limitações atuais, referidos na secção 2.3.5 do capítulo 2.

## 4.1 ARQUITETURA

A ferramenta desenvolvida integra diversos projetos *open source*, com o propósito de evitar a reimplementação de aplicações já desenvolvidas. Nesse sentido, é adotado o OWASP ZAP como núcleo central da aplicação, sendo daí originada a designação "ZAPI". O OWASP ZAP ajuda na identificação de falhas comuns na *web*, presentes também em APIs.

Diversas ferramentas, tais como o ffuf, JWT_Tool e Arjun, são utilizadas com o intuito de auxiliar na deteção de vulnerabilidades. Mais concretamente, o ffuf é empregue como uma ferramenta de fuzzing, destinada a testar diversos parâmetros. O JWT_Tool é utilizado com o intuito de identificar vulnerabilidades em *tokens* JWT, enquanto a ferramenta Arjun é aplicada para detetar parâmetros ocultos. A linguagem de programação utilizada é o Python, devido à sua simplicidade e ao seu grande número de bibliotecas disponíveis.

Com o intuito de integrar todas as ferramentas, foram adotadas duas abordagens distintas. Nos casos das ferramentas desenvolvidas em Python, o processo revelou-se

mais simplificado, uma vez que existe a possibilidade de importação direta da ferramenta para a aplicação em desenvolvimento. Contudo, nas demais situações, recorreu-se à chamada de função os.system(). Esta função permite a execução de comandos no sistema operativo e disponibiliza o respetivo *output* para a aplicação desenvolvida. Adicionalmente, de modo a integrar as funcionalidades do OWASP ZAP com a ferramenta em desenvolvimento, foi utilizada a API (em python) disponibilizada pelo OWASP ZAP (ZAP, 2022). Esta API possibilita a modificação do comportamento do OWASP ZAP conforme os critérios desejados, permitindo, assim, a definição de critérios específicos para as APIs conforme as necessidades do projeto.

De modo a facilitar a interação, testes e visualização de resultados, o Flask é utilizado como *framework web*. Uma das principais vantagens do Flask é a sua capacidade de fornecer uma interface gráfica amigável e acessível aos utilizadores. Esta característica é importante, uma vez que os testes mediante uma interface gráfica tendem a ser consideravelmente mais didáticos do que os testes via *script* (linha de comandos). Assim, proporciona uma experiência mais intuitiva e direta aos utilizadores, tornando o processo de teste mais acessível a um público mais amplo.

Com o intuito de otimizar os testes em APIs, foram utilizados *scripts* públicos disponibilizados na página oficial do OWASP ZAP. Estes *scripts* foram concebidos para otimizar a análise, excluindo do âmbito de testes todas as tecnologias que não apresentem relevância no contexto de APIs. Por exemplo, são eliminadas da análise todas as vulnerabilidades relacionadas com o lado do cliente (*browser*), tais como, *Cross Site Scripting* (Bennetts, 2017).

Por fim, foram desenvolvidos *scripts* adicionais com o propósito de efetuar testes de segurança nas APIs, expandindo, assim, as funcionalidades da ferramenta. Ou seja, estes *scripts* auxiliam na deteção de potenciais vulnerabilidades de segurança.

A representação gráfica presente na figura 14, apresenta uma visão geral da arquitetura da ferramenta ZAPI, que sintetiza todas as informações mencionadas anteriormente.

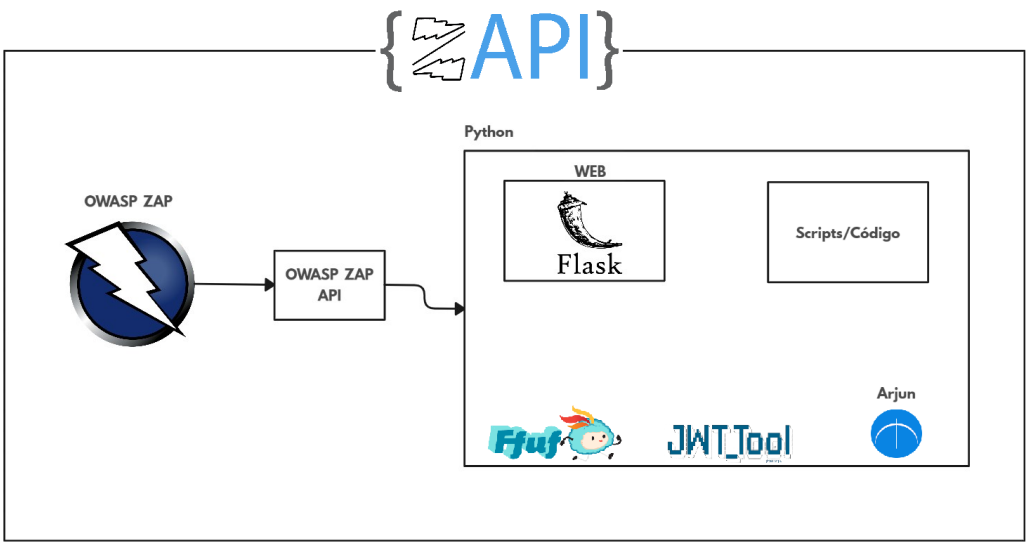


Figura 14: Arquitetura

Simplificando, a aplicação ZAPI tem os seguintes requisitos:

- Compatível apenas com APIs RESTful;
- Necessita de uma especificação de desenvolvimento, OAS, bem documentada;
- Permite testes manuais e automatizados;
- Metodologia de teste *greybox*, por ser necessário dispor dos *tokens* de autenticação para prosseguir com a análise nestes *endpoints*.

## 4.2 INFRAESTRUTURA TECNOLÓGICA

Para o sistema operativo, optou-se pelo uso de distribuições Linux reconhecidas. Ubuntu, nas APIs vulneráveis e Kali Linux no desenvolvimento da aplicação. Este enfoque na escolha de um sistema operativo sólido visa garantir uma base estável para o desenvolvimento da solução proposta. Adicionalmente, a escolha do Kali Linux deve-se ao facto de incorporar a maioria das ferramentas necessárias.

Quanto ao ambiente de desenvolvimento, a plataforma selecionada foi o PyCharm, uma Integrated Development Environment (IDE) conhecida pelo seu conjunto abrangente de ferramentas e funcionalidades específicas para a linguagem de programação Python. Esta escolha proporcionou um ambiente propício à codificação eficaz e *debugging*, contribuindo, assim, para o progresso eficiente do projeto.

## 4.3 COBERTURA APLICACIONAL

A tabela 2, expõe as vulnerabilidades que o ZAPI consegue detetar mediante um teste automatizado. As vulnerabilidades que escapam à deteção, são resultado das dificuldades mencionadas no final do capítulo. Isto acontece devido à limitação da aplicação em compreender se a manipulação dos dados poderá conter, ou não, vulnerabilidades, uma vez que lhe falta inteligência artificial.

| Vulnerabilidades | Consegue detetar? |
|---|---|
| BOLA | ✓ |
| Broken Authentication | ✓ |
| BOPLA | ✓ |
| Unrestricted Resource Consumption | ✓ |
| BFLA | ✗ |
| Unrestricted Access to Sensitive Business Flows | ✗ |
| SSRF | ✓ |
| Security Misconfiguration | ✓ |
| Improper Inventory Management | ✗ |
| Unsafe Consumption of APIs | ✗ |

Tabela 2: Cobertura aplicacional

## 4.4 FUNCIONAMENTO

Na concretização deste projeto, foi uma prioridade a consideração dos requisitos essenciais de User Interface (UI) e User Experience (UX). Compreendendo a importância desses aspetos na relação entre o utilizador e a aplicação, foi realizado um esforço para garantir que a experiência proporcionada fosse simultaneamente intuitiva e simples.

Para a criação de uma *interface* visualmente apelativa e de fácil compreensão foi realizada uma análise de possíveis *layouts*. A seleção de cores e a disposição dos elementos foram escolhidos visando criar uma coerência visual que não apenas captasse a atenção, mas também comunicasse claramente a hierarquia informativa (ver apêndice A).

Além disso, cada etapa do processo de desenvolvimento foi pensada com o intuito de proporcionar uma experiência desprovida de complexidades desnecessárias,

assegurando que as informações essenciais fossem transmitidas de forma direta e acessível.

Nas secções seguintes, serão apresentadas explicações detalhadas acerca do funcionamento da aplicação, bem como das suas características. Inicialmente, será apresentado um fluxograma com o intuito de explicar de forma prática o funcionamento da aplicação. É importante destacar que, para ilustrar o funcionamento da aplicação, é necessário dispor de uma API para testes. Neste contexto, optou-se por utilizar a API Completely Ridiculous API (crAPI) como exemplo, para demonstrar o funcionamento da solução ZAPI. Esta API vulnerável será abordada com mais detalhe no capítulo seguinte.

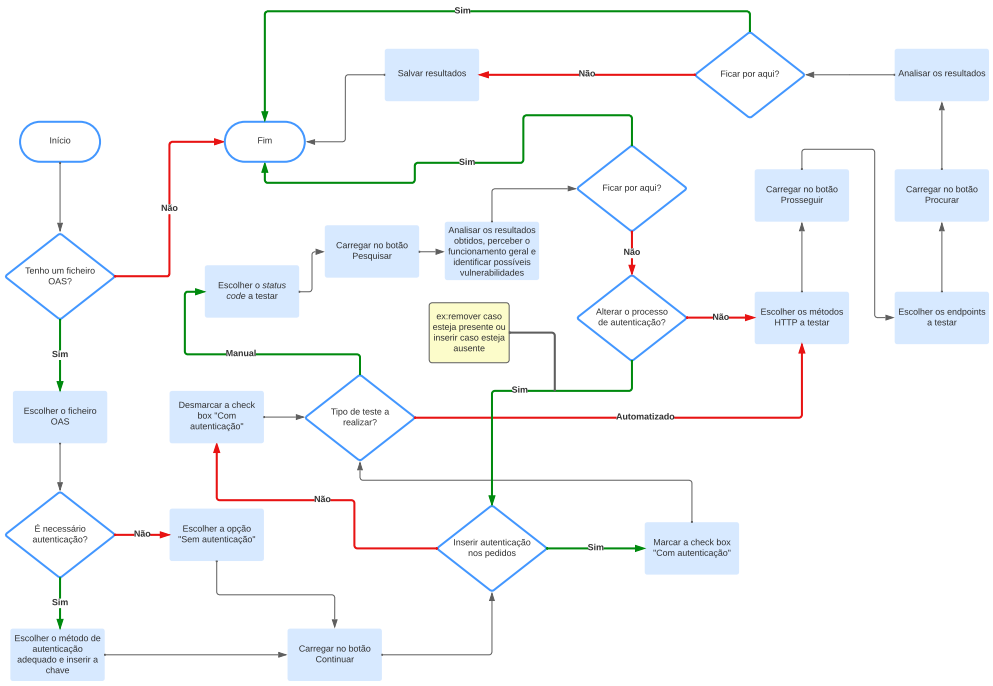


Figura 15: Fluxograma do ZAPI

O fluxograma apresentado na figura 15 representa o funcionamento da solução desenvolvida na perspetiva do utilizador. Como se pode observar, é necessário dispor de um ficheiro OAS para usar a ferramenta. Adicionalmente, é necessário fornecer a chave de autenticação, caso exista. Em seguida, o utilizador tem a possibilidade de decidir se deseja ou não incluir a chave de autenticação nos pedidos HTTP.

Continuando, existem duas opções disponíveis: uma para efetuar testes manuais e outra para testes automatizados. A decisão para permitir testes manuais foi tomada, uma vez que, apesar de existirem ferramentas destinadas a este propósito, como o Postman, o processo aí envolvido é extenso. A título de exemplo, para criar regras

no Postman e analisar os resultados, são necessários sete passos (ir até aos testes, inserir o *script* com o *status code* a ser testado, prosseguir, selecionar os *endpoints* a serem testados, escolher a opção de guardar as respostas HTTP, realizar a análise e, por fim, selecionar manualmente cada um dos *endpoints* cujas respostas HTTP se pretende visualizar). O ZAPI simplifica este processo, reduzindo o número de passos necessários para apenas três. Portanto, caso o utilizador opte pela realização de testes manuais, apenas terá de especificar o *status code* a ser testado, clicar em "Pesquisar" e, por fim, analisar os resultados obtidos. Nos testes manuais, o utilizador possui a flexibilidade de alterar o método de autenticação, ou seja, efetuar pedidos com ou sem a chave de autenticação, consoante o seu objetivo.

Se desejar, poderá realizar testes automatizados, nos quais terá de selecionar os métodos HTTP a serem testados (GET, POST, PUT...), os *endpoints* a serem testados e, por fim, analisar ou guardar os resultados. Naturalmente, o utilizador pode avançar diretamente para os testes automatizados, sem a necessidade de realizar previamente uma abordagem manual.

Prosseguindo para a explicação do uso da ferramenta, explicarei a seguir o uso da ferramenta demonstrando o funcionamento da mesma com base em exemplos práticos, utilizando a API crAPI como base para os testes.

### 4.4.1 *Página inicial*

Ao interagir com a aplicação o utilizador tem que inicialmente escolher o ficheiro OAS, no formato JSON. Este ficheiro deve ser o mais completo possível, visando assegurar a realização de testes abrangentes e confiáveis. A seguir, será apresentada a opção para escolher o método de autenticação da API, caso escolha JWT, *Basic Authentication* ou API *Key*, não é necessário informar a parte inicial do *header*, só o valor, isto é, a chave de autenticação. Se porventura a API utilizar outro método de autenticação, o utilizador pode selecionar a opção "Outro" e adicionar manualmente a chave, por exemplo, um *cookie* (figura 16). A autenticação é necessária, porque como já foi visto anteriormente, na secção 2.3.1 do capítulo 2, o protocolo HTTP é *stateless*, por isso, caso a API tenha autenticação é necessário enviar uma chave para todos os *endpoints*.

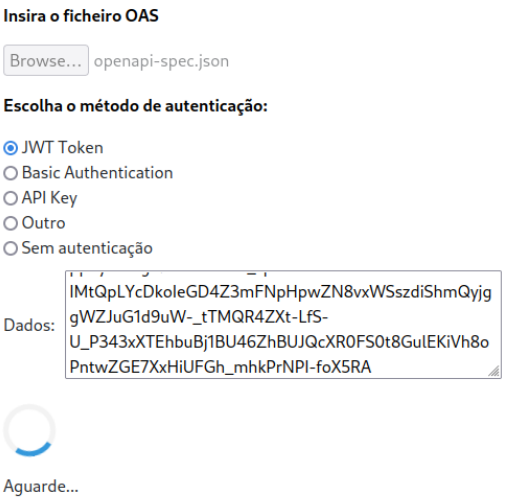


Figura 16: Página Inicial

### 4.4.2 *Testes sem autenticação*

Por omissão, todos os testes são efetuados sem recurso ao mecanismo de autenticação. Tal abordagem permite verificar o correto funcionamento da autenticação e analisar os dados a que um utilizador não autenticado tem acesso através da API.

#### 4.4.2.1 *Página de testes*

Ao prosseguir, o utilizador terá a possibilidade de realizar uma análise manual, localizada no canto superior esquerdo, ou uma análise automatizada, no canto superior direito da página *web* (figura 17). Ou seja, na abordagem manual consegue perceber o funcionamento da API. Por exemplo, é pedido um código de resposta

(*status code*), para verificar quais *endpoints* responderam com esse código, os dados serão devolvidos na secção "Resultados" da página *web*.

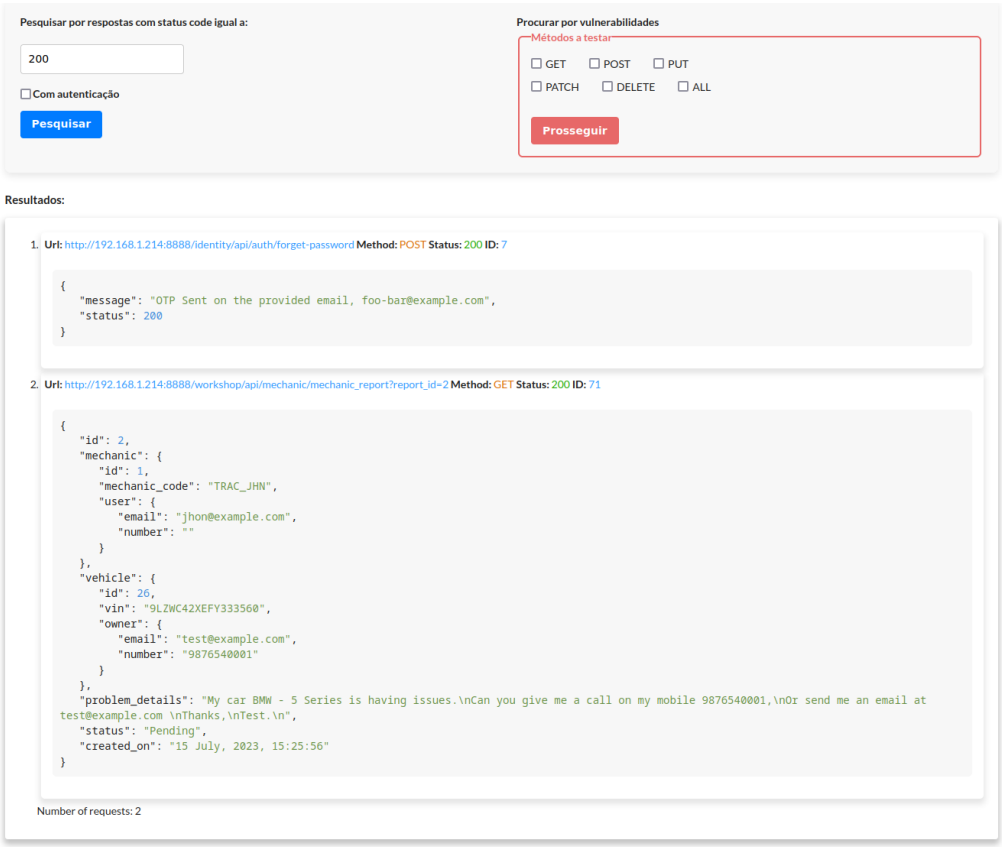


Figura 17: Análise Manual

Supondo que os programadores utilizam o código padrão HTTP para respostas de sucesso, 200, então é possível ver quais *endpoints*, responderam com esse estado (figura 17). Semelhante à ferramenta Postman, mas um processo significativamente mais ágil. Esta análise permite verificar se todos os *endpoints* implementam a autenticação corretamente. No entanto, caso seja utilizado outro código, também é possível realizar a mesma análise. Nesse caso, a aplicação devolveria os resultados na secção "Restantes respostas" e, nessa altura, seria possível ver como a API respondeu (figura 18).

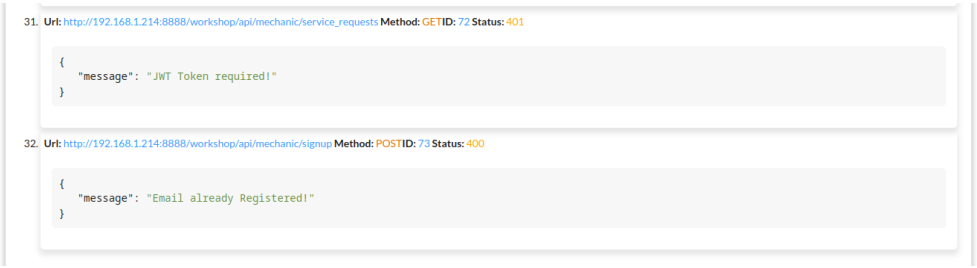


Figura 18: Análise Manual - Restantes respostas

Adicionalmente, há uma secção denominada "Mudanças entre pedidos", que visa simplificar a deteção de alterações nas respostas de um determinado *endpoint*. A figura 19 retrata esta situação. Inicialmente o pedido com *id* 75, apresentava a mensagem "Email não registado", contudo, no subsequente pedido, identificado como 76, a mensagem foi alterada para "Senha incorreta", isto permite identificar, mais facilmente, se certos comportamentos podem apresentar vulnerabilidades ou não.


Figura 19: Mudanças entre os pedidos da API

Considerando o exemplo de utilização ilustrado na figura 17, verifica-se que dois *endpoints*, **forget-password** e **mechanic_report?report_id=2** estão acessíveis sem autenticação. Além disso, na figura 18, está indicado o código de resposta, 401 e 400, das mensagens que não satisfazem o filtro (*status code* igual a 200). Analisando as respostas, é possível retirar algumas conclusões. Neste exemplo, o *endpoint* **signup** está vulnerável, por devolver informação demasiado específica.

O ZAPI, tem ainda a funcionalidade de análise automatizada, ou seja, continuando o mesmo exemplo e supondo ser um teste minucioso, existe a possibilidade de escolher os métodos a testar, para evitar danos na API. De modo geral, o método GET é o mais seguro de ser testado, porque se for implementado corretamente devolve apenas recursos e não realiza ações como, por exemplo, de remoção. Tendo em conta que apenas dois *endpoints* estão disponíveis sem autenticação (neste exemplo),

pode-se selecionar apenas estes (figura 20), já que os restantes não fazem sentido (neste contexto) uma vez que estão relacionados com processos de autenticação que serão explorados em detalhe posteriormente.

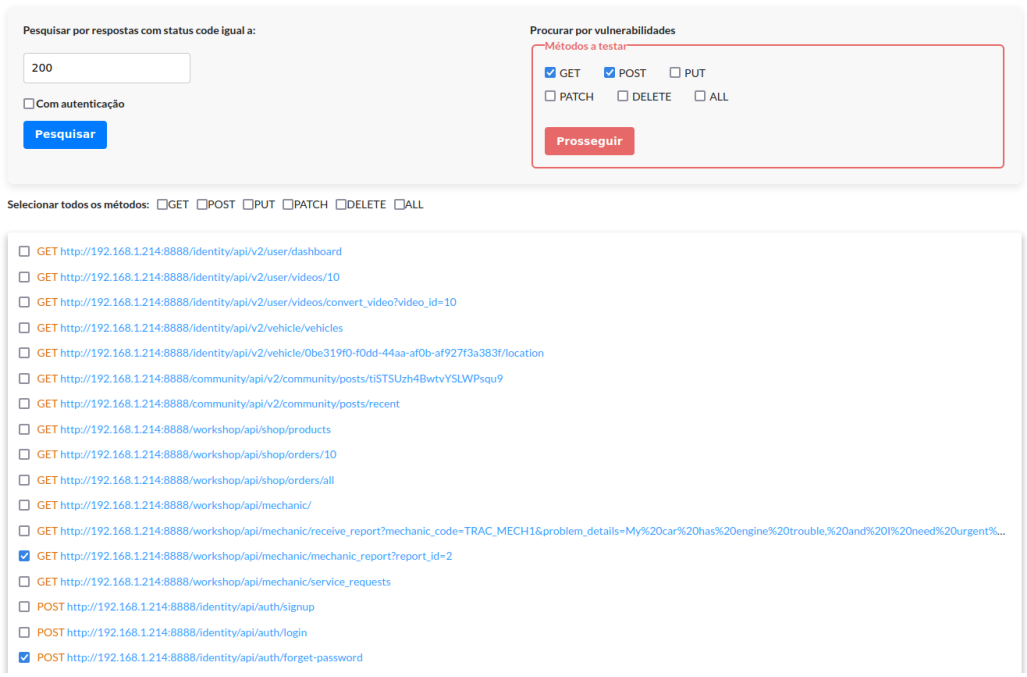


Figura 20: Análise automatizada

Por fim, ao clicar no botão “Procurar” a análise é iniciada. Após a conclusão deste processo, os dados são exibidos em forma de lista ordenada, consoante o tipo de vulnerabilidade, do OWASP API Top 10 de 2023. Além disso, é possível guardar os resultados para consulta futura ou partilha, conforme mostra a figura 21.

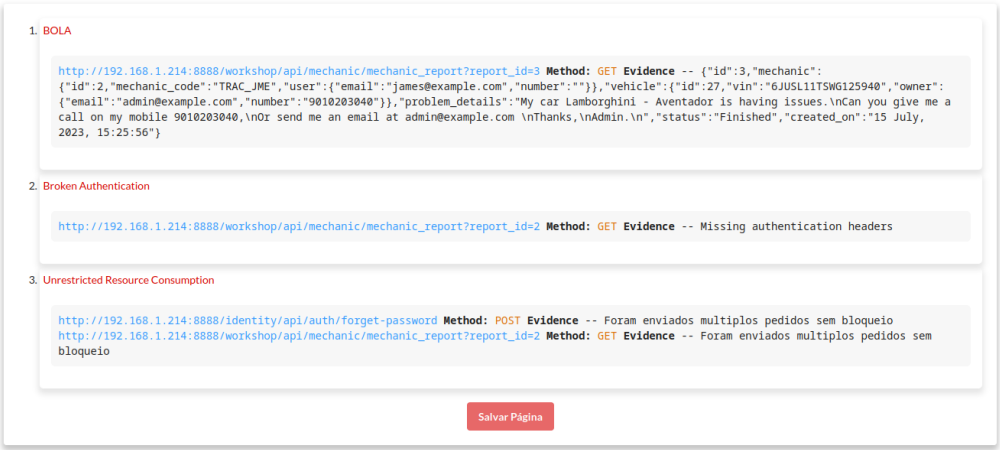


Figura 21: Análise automatizada - Resultados

#### 4.4.2.2 *Página Resultados*

Dando continuidade aos testes na API de exemplo, após a análise automatizada, é possível inferir que a aplicação revelou três tipos distintos de vulnerabilidades e resultou em quatro *findings* (figura 21). Esta constatação ocorreu em apenas dois dos *endpoints* da API.

Uma das vulnerabilidades identificadas, classificada como BOLA, permitiu a alteração do número de **report_id**, resultando no acesso a dados de outros utilizadores.

Relativamente à vulnerabilidade associada à autenticação, importa salientar que esta já havia sido detetada durante a fase de testes manuais.

Por último, a terceira vulnerabilidade evidenciou a possibilidade de enviar múltiplos pedidos para obter determinado recurso. Esta situação pode induzir à instabilidade do sistema e à ocorrência de um potencial ataque de DOS.

### 4.4.3 *Testes com autenticação*

A segunda modalidade de utilização da aplicação é mediante a realização de testes com autenticação. Para utilizar esta abordagem, é imperativo selecionar a opção "Com autenticação" situada no canto superior esquerdo da aplicação *web* (figura 23). Esta funcionalidade desencadeará a incorporação do mecanismo de autenticação em cada *endpoint*.

O processo de autenticação opera da seguinte forma: A *framework web* envia o método de autenticação, com a respetiva chave de autenticação fornecida pelo utilizador, para a API do OWASP ZAP. Nesse ponto, é utilizado o *endpoint* "replacer". A função *replacer* permite que sejam definidas regras que serão aplicadas aos pedidos e respostas HTTP antes de serem enviados. Logo, é possível configurar um *header* de autenticação para todos os *endpoints* pretendidos.

Em seguida, a aplicação procede com uma atualização, o que implica a remoção e a subsequente importação do ficheiro OAS, assegurando, assim, a conformidade com as especificações desejadas. Por fim, os dados obtidos são enviados para a *framework web* (Flask), de modo a serem apresentados ao utilizador (ver figura 22).

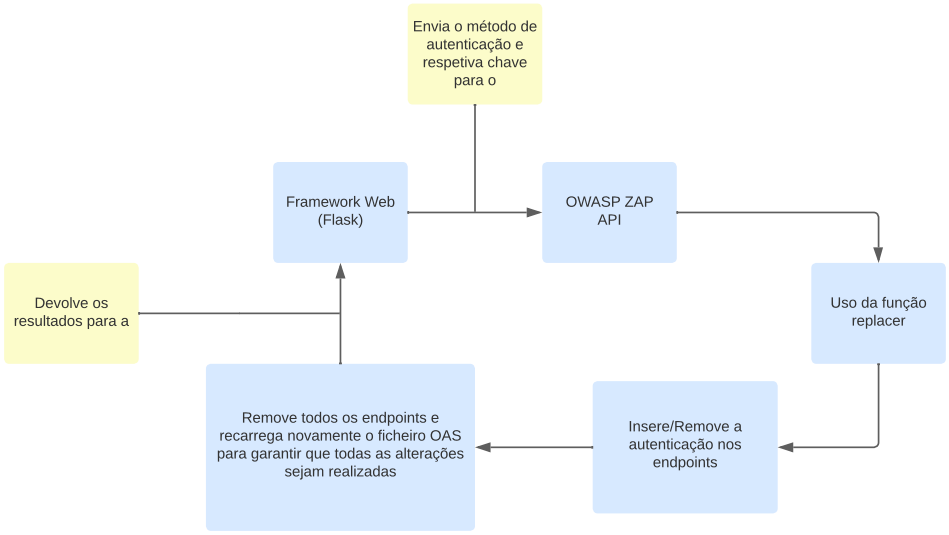


Figura 22: Fluxograma do processo de autenticação

#### 4.4.3.1 *Página de testes*

Ao aplicar as mesmas escolhas de filtragem utilizadas anteriormente, é observável que o leque de resultados apresentados aumenta consideravelmente. A existência de mais resultados na abordagem manual relativamente ao caso anterior, deve-se ao facto de que os pedidos estão a ser executados sob autenticação. Assim, agora observamos um aumento no número de resultados que atendem ao critério de *status code* igual a 200. Adicionalmente, a aplicação oferece a capacidade de ocultar respostas. Este recurso revela-se especialmente útil ao obter uma lista exaustiva de resultados, permitindo a apresentação somente daquilo que é pertinente (figura 23).

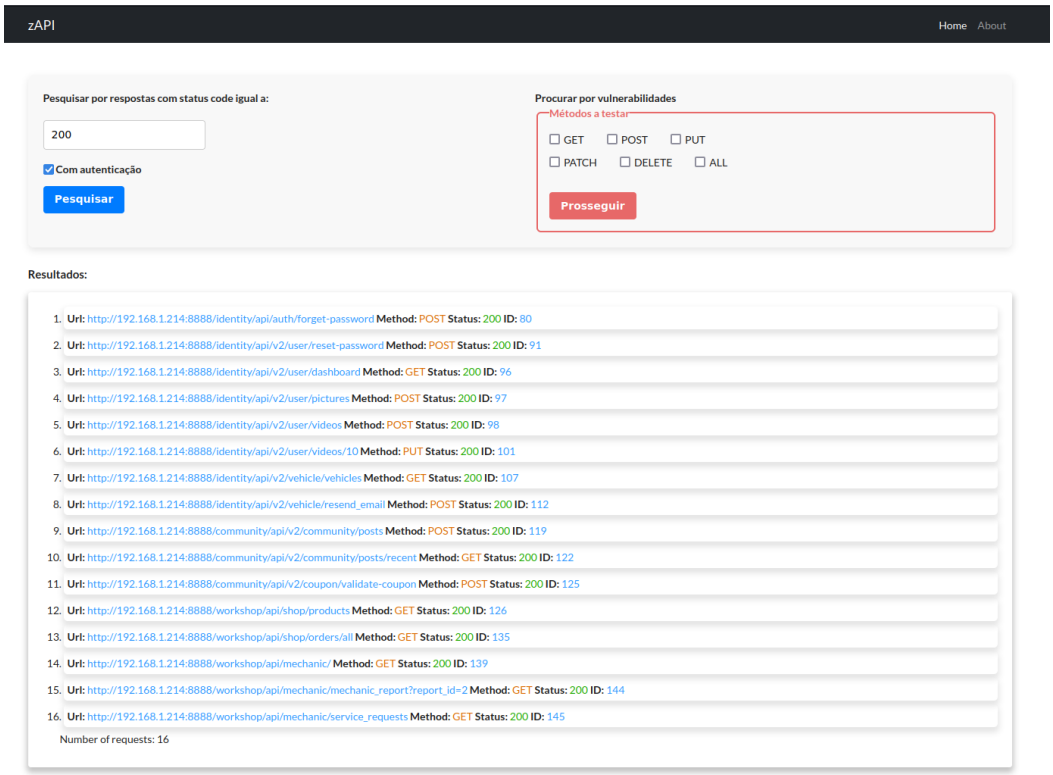


Figura 23: Análise manual com autenticação

#### 4.4.3.2 *Análise manual*

Ao adotar uma abordagem de investigação manual, focada individualmente em cada *endpoint*, surge uma observação. O *endpoint* **posts/recent**, devolve na resposta excesso de informações, ou seja, uma excessiva exposição de informações sensíveis. Esta exposição é uma das vulnerabilidades identificadas e abordadas no OWASP API Top 10 de 2023 (BOPLA).

De modo mais técnico, a aplicação utiliza um *token* associado ao **utilizador b**, para realizar pedidos com autenticação. No entanto, um exame mais minucioso revela que, dados inerentes ao **utilizador a** são igualmente incorporados na resposta, abarcando elementos como o endereço eletrónico, *url* da fotografia e a identificação do veículo, **vehicleid** (ver figura 24).

Assim, a aplicação simplifica a análise dos resultados, mesmo ao recorrer a uma abordagem manual.


Figura 24: Análise manual - Excessive Data Exposure

Analisando a secção de "Restantes Repostas", torna-se visível a existência de um *endpoint* destinado à consulta da localização do veículo (figura 25). Como resultado da correlação entre a informação previamente exposta, de forma vulnerável, no *endpoint* precedente, emerge uma situação em que o **utilizador b**, mediante a submissão da UUID neste novo endpoint, consegue obter acesso à localização do veículo pertencente ao **utilizador a**. Esta situação demonstra a junção de duas vulnerabilidades. Primeiramente, dados sensíveis são obtidos (BOPLA) e, consequentemente, estes dados são utilizados numa vulnerabilidade de BOLA, para obter acesso ilícito à localização do veículo de outro utilizador (listagem 5). É relevante destacar que os testes manuais realizados, necessitam que o analista consiga interpretar os resultados, de modo a detetar potenciais vulnerabilidades. No entanto, tal como já foi referido, este processo está muito mais ágil no ZAPI, em comparação com outras ferramentas, como o Postman.

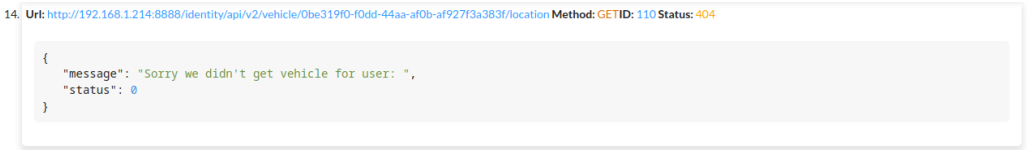


Figura 25: Endpoint para obter a localização do veículo

```
{
    "carId": "e449ba19-0ef8-477a-9cea-50e6362fb1a3",
    "vehicleLocation":
    {
        "id": 2,
        "latitude": "39.0247621",
        "longitude": "-77.1402267"
    },
    "fullName": "UserA"
}
```


Listagem 5: Resposta JSON da localização do veículo

#### 4.4.3.3 *Análise automatizada*

Prosseguindo para uma análise automatizada e com autenticação, são selecionados os *endpoints* a serem submetidos a testes, nos quais a aplicação realizará uma sequência de iterações com o intuito de identificar um leque abrangente de vulnerabilidades.

O processo de automatização está delineado na figura 26. Inicialmente, os *endpoints* a serem testados e escolhidos pelo utilizador são incorporados num contexto específico. Um contexto no OWASP ZAP representa um ambiente controlado onde se definem as configurações de teste e URLs alvo, com o propósito de analisar partes específicas de uma aplicação durante as análises de segurança. Este método assegura, por exemplo, que apenas os *endpoints* selecionados são submetidos a análise.

Uma vez que o contexto é criado, os *scripts* auxiliares, destinados a identificar vulnerabilidades, são executados. Dentro destes *scripts*, várias técnicas são aplicadas. Por exemplo, para detetar vulnerabilidades do tipo BOLA, são utilizadas diversas expressões regulares para identificar potenciais parâmetros a serem manipulados. Caso um candidato seja identificado, a aplicação gera dados semelhantes, tais como nomes, endereços de correio eletrónico ou números. No caso dos números, apenas o último dígito é aleatoriamente gerado (por exemplo, no número 123, apenas o 3 seria alterado).

Posteriormente, o *script* cria um novo pedido HTTP contendo os dados recém-gerados. Para avaliar a validade dos resultados obtidos, é utilizada a medida de similaridade do cosseno, uma métrica estatística que quantifica a semelhança entre dois vetores. A presença de semelhança indica a alta probabilidade de os novos parâmetros gerados serem vulneráveis. Além disso, diversos *scripts* recorrem a bibliotecas de Natural Language Processing (NLP), para otimizar os resultados das expressões regulares. Por exemplo, são realizadas validações para verificar se

uma determinada *string* existe nos dicionários de várias línguas, como o inglês e o português.

Concluídos os testes, os *scripts* invocam o *endpoint* "alert" da API do OWASP ZAP para inserir as vulnerabilidades identificadas. Completada esta etapa, o OWASP ZAP entra em ação, dando início a um *active scan*. Trata-se de uma verificação automatizada que simula ataques à aplicação alvo, visando a identificação de potenciais vulnerabilidades de segurança. Ou seja, o OWASP ZAP procura por vulnerabilidades *web* comuns às APIs. Por fim, os dados são extraídos do OWASP ZAP e enviados para a *framework web*.

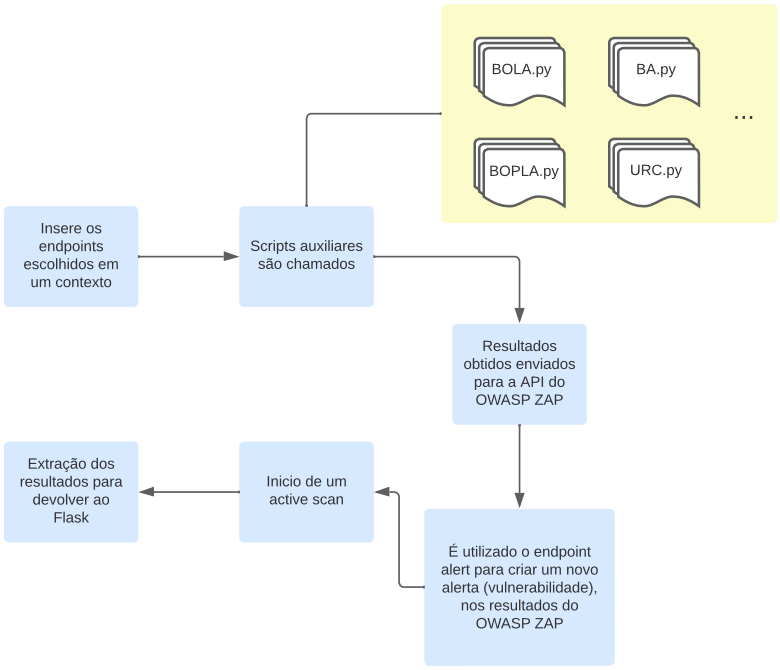


Figura 26: Fluxograma do processo de análise de vulnerabilidades

As figuras 27 e 28 ilustram claramente a apresentação de diversos resultados.

Tal como, é inerente a qualquer aplicação, a mesma está sujeita a possíveis equívocos, conhecidos como falsos positivos, instaurando a possibilidade de requerer análises manuais em determinadas circunstâncias.

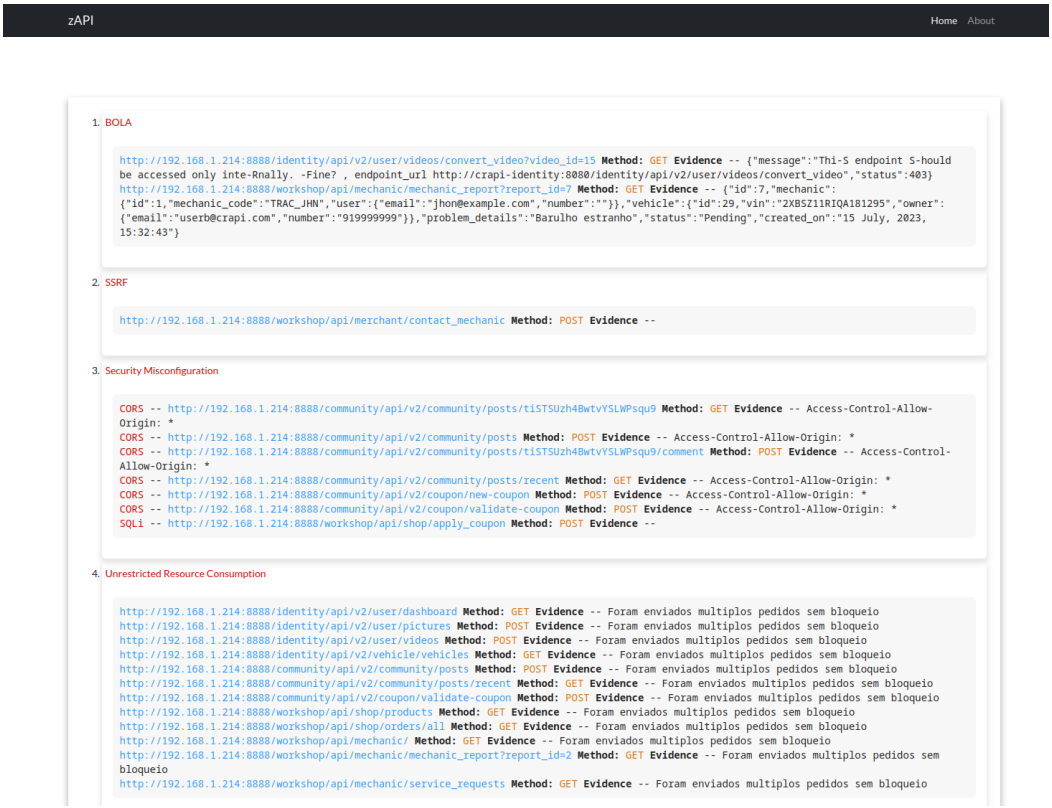


Figura 27: Resultados da análise automatizada com autenticação

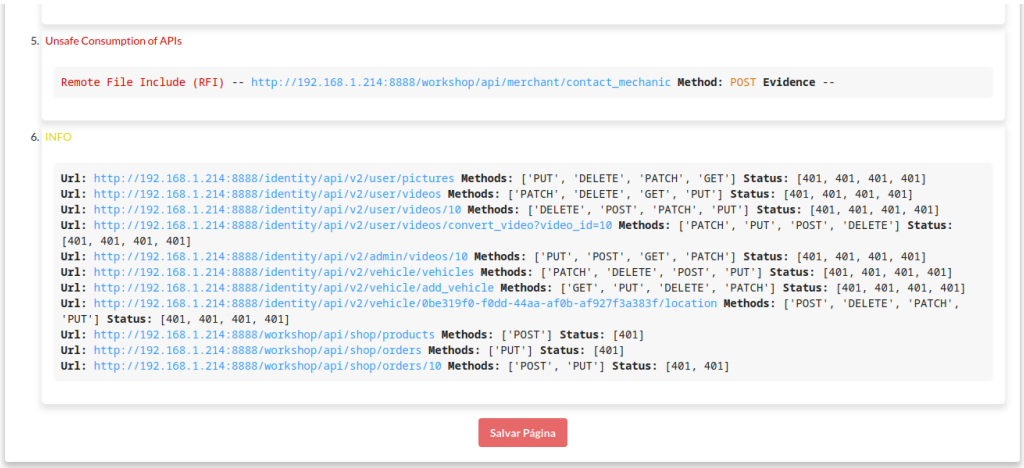


Figura 28: Resultados da análise automatizada com autenticação - Continuação

### 4.4.4 *Testes auxiliares*

Outra característica do ZAPI, reside no retorno de informações, relevantes ou não, nos resultados obtidos. A título ilustrativo, considerando uma situação em que o

ficheiro OAS especifica um dado *endpoint* que utiliza o método GET. Nesse cenário, a aplicação não se limitará a verificar apenas esse método, mas também avaliará a possibilidade de outros métodos, tais como POST, PUT, entre outros. Caso seja constatado, a aplicação apresentará tal detalhe nas informações dos resultados (INFO). Esta funcionalidade, pode ajudar a enriquecer a qualidade e abrangência da análise realizada.

Por último, a aplicação disponibiliza dados no terminal da IDE. Esta abordagem visa a redução do excesso de informações intercaladas entre os resultados, proporcionando uma clareza que evita confusões, considerando a potencial utilidade, ou não, da informação fornecida. Na figura 29, é possível identificar parte deste tipo de informação. Neste contexto específico, são comunicados detalhes referentes a possíveis parâmetros *id* que, em circunstâncias de vulnerabilidade, podem ser sujeitos a manipulação. Além disso, o conteúdo do *token* JWT também é exibido.

Com o intuito de exemplificar este cenário e prosseguindo com a utilização da API de demonstração, na figura 30, a aplicação emite um alerta para a potencial vulnerabilidade do endpoint **location** no contexto do UUID. Como já foi mencionado anteriormente, tal situação verifica-se, pois, era possível obter a localização do veículo de outros utilizadores.

```
Found potential ID parameter:
Path: http://192.168.1.214:8888/workshop/api/shop/orders/return_order?order_id=33
Method: POST
Parameter: order_id

Found potential ID parameter:
Path: http://192.168.1.214:8888/workshop/api/mechanic/mechanic_report?report_id=2
Method: GET
Parameter: report_id

{'sub': 'userb@crapi.com', 'role': 'user', 'iat': 1692097908, 'exp': 1692702708}
```

Figura 29: Informação do terminal da IDE


Figura 30: Informação do terminal da IDE - Continuação

## 4.5 DIFICULDADES

Ao longo do desenvolvimento da aplicação, surgiram várias dificuldades significativas. A arquitetura REST, pela sua natureza flexível, concede aos programadores uma liberdade de implementação da API, possibilitando uma variedade de abordagens distintas. No entanto, essa flexibilidade também acarreta desafios na criação de uma aplicação capaz de abranger todos os cenários possíveis. Para ilustrar, na funcionalidade de procurar *IDs* não foi suficiente utilizar, apenas, expressões regulares (regex). Em determinados casos, foi necessário adotar uma biblioteca de NLP para alcançar resultados mais precisos. Adicionalmente, é importante salientar que a documentação associada à API do OWASP ZAP apresenta limitações ao restringir-se a cenários específicos. Encontrar referências para a execução de certas ações foi uma tarefa árdua, tendo em conta que muitas das informações estavam obsoletas.

Além disso, nos *endpoints* destinados a registos ou inícios de sessão, é importante evitar erros de *Broken Authentication*, uma vez que os utilizadores precisam aceder a esses pontos para obter o respetivo *token*/chave. Embora a aplicação utilize uma lista de nomes comuns (ex: login, logout, signup, signin...), existe a possibilidade de um programador utilizar um nome invulgar, o que poderia resultar em falsos positivos. Existe ainda a necessidade de um ficheiro OAS bem documentado e abrangente, mas o maior desafio reside na interpretação contextual. A aplicação requer um nível de inteligência (que nesta fase ainda não possui) que lhe permita analisar os dados fornecidos e interpretá-los com precisão, constituindo a maior dificuldade encontrada.

# 5

# RESULTADOS

Este capítulo apresenta os resultados obtidos por meio da utilização da ferramenta ZAPI em diversas APIs vulneráveis. Adicionalmente, são analisados e discutidos os resultados, comparando com as ferramentas gratuitas selecionadas no capítulo estado da arte.

## 5.1 ARQUITETURA DE TESTES

No desenvolvimento do presente projeto, foram utilizados recursos provenientes de projetos públicos, nomeadamente, APIs vulneráveis. As escolhas das APIs vulneráveis foram conduzidas criteriosamente, visando uma compreensão detalhada das vulnerabilidades existentes.

Foram elas:

- crAPI [1]
  A crAPI tem como propósito auxiliar na compreensão dos dez riscos de segurança mais críticos em APIs (OWASP API:2019);
- VAmPI [2]
  A VAmPI é uma API vulnerável desenvolvida em Flask (framework web) com vulnerabilidades do OWASP Top 10, criada para avaliar a eficácia das ferramentas de segurança em APIs e para aprendizagem;
- vAPI [3]
  A vAPI é uma API vulnerável, que simula cenários do OWASP API Top 10 (OWASP API:2019), mediante exercícios.

Através do diagrama presente na figura 31, é possível visualizar de maneira elucidativa que três instâncias do sistema operativo Ubuntu, cada uma delas com uma API suscetível a vulnerabilidades, foram deliberadamente selecionadas como

1 https://github.com/OWASP/crAPI
2 https://github.com/erev0s/VAmPI
3 https://github.com/roottusk/vapi

alvo para as aplicações, pynt, Akto e ZAPI. Esta representação gráfica oferece uma visão simplificada da configuração adotada para conduzir a análise e teste da solução mencionada.

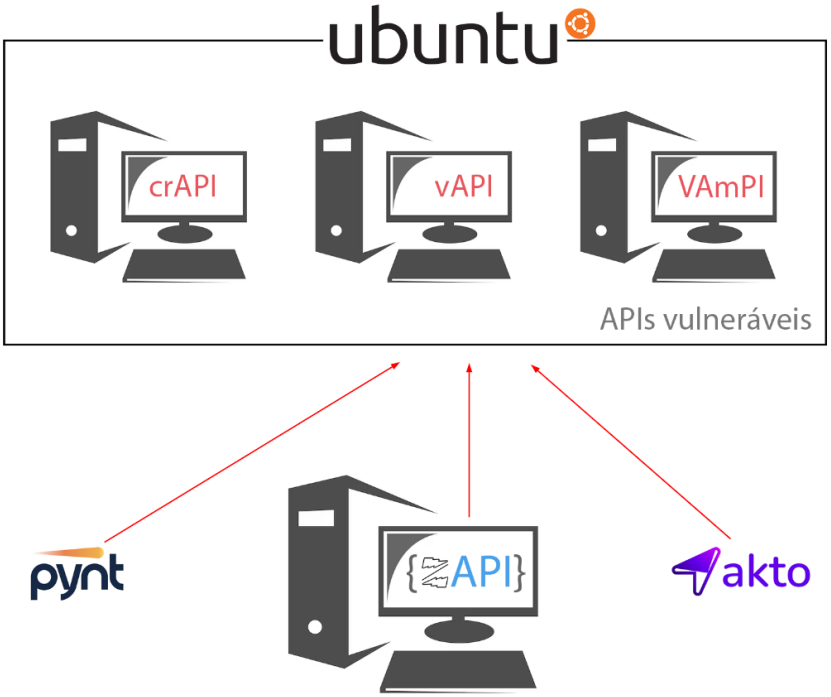


Figura 31: Diagrama de testes

## 5.2 TESTES

Para a realização dos testes, foram utilizadas as APIs mencionadas previamente como alvos da análise. Posteriormente, foram aplicadas as ferramentas OWASP ZAP, pynt, Akto e ZAPI, de modo a efetuar uma análise por possíveis vulnerabilidades.

Todas as ferramentas proporcionam a escolha entre duas opções iniciais: a utilização de um ficheiro OAS ou de uma coleção Postman, para dar início aos testes.

Contudo, optou-se por adotar uma estratégia diferente para a realização dos testes no OWASP ZAP. Essa estratégia envolveu a configuração desta aplicação como um *proxy* no ambiente do Postman. Este procedimento garantiu que todos os pedidos fossem enviados de forma já devidamente formatada (*headers* e parâmetros corretos).

No que concerne à aplicação pynt, procedeu-se à execução pelo newman (executa coleções do Postman pelo terminal), para possibilitar a realização de testes num ambiente local.

Já em relação à ferramenta Akto, optou-se por efetuar o respetivo *download* e subsequente execução numa instância docker, também em ambiente local.

É importante mencionar que os testes foram conduzidos num contexto de autenticação e todas as ferramentas utilizadas estavam nas versões mais recentes disponíveis, garantindo, assim, a utilização das capacidades e funcionalidades mais atualizadas.

Além disso, todas as variáveis e configurações foram corretamente preenchidas, com o propósito de identificar o maior número, possível, de vulnerabilidades. Os resultados obtidos foram organizados e registados na tabela 3. Nesta tabela, para cada API vulnerável, são exibidas as quantidades de vulnerabilidades identificadas por cada uma das ferramentas utilizadas (*max.* 10).

| Ferramenta | N.º de vulnerabilidades encontradas (OWASP API Top 10) | | |
|---|---|---|---|
| | crAPI | vAPI | VAmPI |
| pynt | 1 | 1 | 1 |
| Akto | 1 | 0 | 0 |
| OWASP ZAP | 2 | 2 | 2 |
| ZAPI | 4 | 5 | 4 |

Tabela 3: Análise comparativa

Com base nos resultados obtidos, é possível constatar a eficácia limitada das ferramentas atuais. Tanto o pynt como o Akto detetaram, apenas, um tipo de vulnerabilidade contida no OWASP API Top 10, nomeadamente *Security Misconfiguration* (ver apêndice B). Nesse sentido, torna-se evidente a urgência de promover uma maior investigação e desenvolvimento no campo da segurança de APIs, visando a criação de soluções robustas e eficazes em termos de teste e proteção. A este respeito, existe uma previsão da Gartner, que refere o seguinte: "até 2026, 40% das organizações optarão por um fornecedor de WAAP com base na excelência das suas funcionalidades de proteção avançada de APIs e segurança de aplicações *web*" (Liu, 2022).

Embora o OWASP ZAP tenha demonstrado um desempenho superior, dado o seu foco para a *web*, ainda não engloba a maioria das vulnerabilidades.

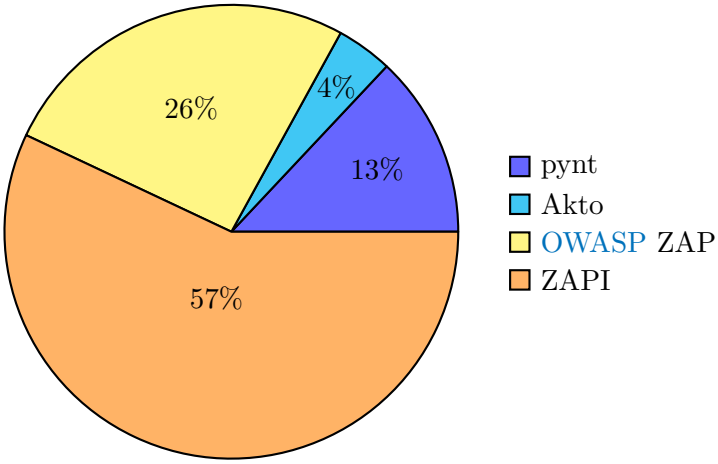


Figura 32: Percentagem de vulnerabilidades encontradas

Em contrapartida, o ZAPI identificou o maior número de vulnerabilidades. O ZAPI, construído com base no OWASP ZAP, possui a capacidade de identificar não apenas todas as vulnerabilidades inerentes ao próprio OWASP ZAP, mas também agrega os resultados provenientes da combinação de vários *scripts* e ferramentas criadas especificamente para a deteção de vulnerabilidades. Esta integração, contribuiu para a eficácia na identificação de vulnerabilidades.

No que diz respeito às outras ferramentas utilizadas, é notório que estas não conseguiram corresponder às expectativas. Ressalta-se que os testes em questão foram conduzidos em múltiplas ocasiões, todavia, os resultados obtidos não sofreram nenhumas modificações ao longo das repetições.

O gráfico ilustrado na figura 32 confirma estas conclusões, exibindo que 57% das vulnerabilidades foram identificadas pelo ZAPI, 26% pelo OWASP ZAP, 13% pelo pynt e, por último, 4% pelo Akto.

# 6

# CONCLUSÕES

Ao finalizar esta jornada de investigação e desenvolvimento, verificou-se que os objetivos iniciais estabelecidos para este projeto foram concluídos com sucesso.

Num contexto marcado pela escassez de soluções gratuitas e pela comparação com as alternativas disponíveis no mercado, o ZAPI surge como uma aplicação útil na tarefa de identificação de vulnerabilidades. Ao simplificar a deteção de falhas de segurança, esta ferramenta desempenha um papel crucial ao permitir a redução do tempo investido na procura por vulnerabilidades. Além disso, promove um mapeamento mais eficaz do funcionamento da API, com potenciais vetores de ataque em ambientes RESTful.

Este projeto não apenas atingiu os seus objetivos iniciais, mas também permitiu a aquisição e aprofundamento de conhecimento acerca das vulnerabilidades em APIs. O conhecimento obtido não só viabiliza uma melhoria nas práticas de desenvolvimento seguro, mas também abre portas para a realização de auditorias de cibersegurança mais robustas em APIs.

Durante o percurso de desenvolvimento foram encontrados diversos desafios. Pois, foi escolhido um tema que, embora recente, ganhou proeminência nos últimos anos, enfrentou-se a complexidade de lidar com múltiplos casos possíveis, problemas complexos e ameaças em constante evolução. No entanto, todas as adversidades foram superadas com sucesso, pois, dada a impossibilidade de uma cobertura aplicacional para todos os casos possíveis, foram considerados apenas os casos mais comuns. Cada obstáculo enfrentado proporcionou um aprendizado valioso, enriquecendo ainda mais o processo de desenvolvimento.

Em última análise, este projeto não só fortaleceu o conhecimento teórico e prático no campo da segurança de APIs, mas também atuou como um instrumento de crescimento profissional e pessoal. Ao alcançar resultados satisfatórios, ampliando a compreensão das vulnerabilidades e possíveis contramedidas, o ZAPI é um recurso valioso para profissionais de cibersegurança e programadores.

## 6.1 TRABALHO FUTURO

Durante o desenvolvimento do projeto, foram surgindo algumas ideias e melhorias que poderiam ser implementadas para melhorar o funcionamento da aplicação. Por exemplo, a implementação da compatibilidade com outras arquiteturas ou protocolos, tais como SOAP e GraphQL.

Adicionalmente, considerando a liberdade para diversas formas de implementação, a incorporação de técnicas de inteligência artificial é indispensável, de modo a englobar um espectro mais abrangente de todos os cenários possíveis, com uma margem de erro reduzida. Esta melhoria iria potenciar a eficácia da aplicação, permitindo uma análise mais abrangente e precisa das vulnerabilidades em APIs, considerando múltiplos contextos. Além disso, iria permitir uma análise 100% automatizada com pouca margem de erro.

Por último, seria prudente considerar a implementação de diversas opções de testes, abrangendo desde abordagens mais agressivas até as mais conservadoras. Isto é, em situações em que se trate de um serviço crítico, os testes aplicados à API seriam conduzidos com muito mais cautela. Além disso, pretende-se disponibilizar a ferramenta desenvolvida à comunidade como um projeto *open source*. Isto irá permitir que programadores e entusiastas melhorem e adaptem a ferramenta às suas necessidades e promove o seu desenvolvimento constante.

# BIBLIOGRAFIA


*About raml* (2023). URL: https://raml.org/about-raml (acedido em 14/01/2023).

Adkuloo, Neelabja (ago. de 2023). *What is API: Types, examples, protocols, how does it work*. URL: https://www.mailmodo.com/guides/api/ (acedido em 01/09/2023).

Agarwal, Saurabh (fev. de 2020). *REST API testing : From Manual Approach to Automation Techniques*. URL: https://www.linkedin.com/pulse/rest-api-testing-from-manual-approach-automation-saurabh-agarwal/ (acedido em 28/12/2022).

Amazon (2023). *PT*. URL: https://aws.amazon.com/pt/compare/the-difference-between-soap-rest/ (acedido em 01/03/2023).

*API Security: Protect your APIs from Attacks and Data Breaches* (2021). URL: https://www.gartner.com/en/webinars/4002323/api-security-protect-your-apis-from-attacks-and-data-breaches (acedido em 08/11/2022).

Apisecurity.io (2019). *Owasp API Security top 10*. URL: https://apisecurity.io/owasp-api-security-top-10/ (acedido em 20/12/2022).

Ball, Corey (2022). *Hacking APIs: breaking web application programming interfaces*. San Francisco: No Starch Press. ISBN: 9781718502444.

Bennetts, Simon (jun. de 2017). *Scanning apis with zap*. URL: https://www.zaproxy.org/blog/2017-06-19-scanning-apis-with-zap/ (acedido em 22/12/2022).

Best, Stephanie (2023). URL: https://salt.security/blog/api10-2023-unsafe-consumption-of-apis (acedido em 08/08/2023).

Bishop, Mike (2022). *RFC 9114*. URL: https://www.rfc-editor.org/rfc/rfc9114.html (acedido em 22/12/2022).

Bottarini, Jon (2018). URL: https://hackerone.com/reports/334143 (acedido em 06/08/2023).

Checkmarx (jul. de 2021). URL: https://checkmarx.com/blog/checkmarx-research-soundcloud-api-security-advisory/ (acedido em 06/08/2023).

Cilwerner (2023). *Autenticação vs. Autorização - Microsoft entra*. URL: https://learn.microsoft.com/pt-pt/azure/active-directory/develop/authentication-vs-authorization (acedido em 10/09/2023).

dblk4 (2023). *Unable to run against Postman Collection - issue #144 - Flipkart-incubator/Astra*. URL: https://github.com/flipkart-incubator/Astra/issues/144 (acedido em 18/08/2023).

Díaz-Rojas, Josué Alejandro et al. (2021). «Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study». Em: *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pp. 207–218. DOI: 10.1109/CONISOFT52520.2021.00036.

Eduardo, Carlos (jul. de 2022). *Carlos Eduardo*. URL: https://www.gocache.com.br/seguranca/por-que-api-security-difere-de-web-application-security/ (acedido em 10/10/2022).

Ehsan, Adeel et al. (2022). «RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions». Em: *Applied Sciences* 12.9. ISSN: 2076-3417. DOI: 10.3390/app12094369. URL: https://www.mdpi.com/2076-3417/12/9/4369.

Faguy, Ana (fev. de 2023). *Ticketmaster urges Congress to crack down on ticket scalping as lawmakers take aim over Taylor Swift debacle*. URL: https://www.forbes.com/sites/anafaguy/2023/02/23/ticketmaster-urges-congress-to-crack-down-on-ticket-scalping-as-lawmakers-take-aim-over-taylor-swift-debacle/?sh=747b7f1cc4c5 (acedido em 07/08/2023).

Fette, I. e A. Melnikov (dez. de 2011). *The websocket protocol*. URL: https://www.rfc-editor.org/rfc/rfc6455 (acedido em 22/12/2022).

Fielding, Roy T., Mark Nottingham e Julian Reschke (2022). *RFC 9110*. URL: https://www.rfc-editor.org/rfc/rfc9110.html (acedido em 22/12/2022).

GoCache (jul. de 2022). *Por que Web Application Security não é suficiente para segurança em API's?* URL: https://www.gocache.com.br/noticias/por-que-web-application-security-nao-e-suficiente-para-seguranca-em-apis/ (acedido em 10/10/2022).

Hadley, Marc (2009). *Web Application Description Language*. URL: https://www.w3.org/submissions/wadl/ (acedido em 01/03/2023).

Hardcastle, Jessica (ago. de 2021). *Why WAFS don't work according to a hacker - sdxcentral*. URL: https://www.sdxcentral.com/articles/interview/why-wafs-dont-work-according-to-a-hacker/2021/08/ (acedido em 14/01/2023).

Hat, Red (2023). *O que É api rest?* URL: https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api (acedido em 01/03/2023).

Isbitski, Michael (mai. de 2021). URL: https://salt.security/blog/the-peloton-api-security-incident-what-happened-and-how-you-can-protect-yourself (acedido em 06/08/2023).

Isha, Abhinav Sharma e M. Revathi (2018). «Automated API Testing». Em: *2018 3rd International Conference on Inventive Computation Technologies (ICICT)*, pp. 788–791. DOI: 10.1109/ICICT43934.2018.9034254.

Kumar, Mohit (abr. de 2019). *Over 100 million Justdial users' personal data found exposed on the internet*. URL: https://thehackernews.com/2019/04/justdial-hacked-data-breach.html (acedido em 08/08/2023).

Levo (2023). URL: https://docs.levo.ai/vulnerabilities/v1/OWASP-API-10/A5-BFLA (acedido em 19/08/2023).

Liu, Nancy (2022). *Gartner names Akamai, Cloudflare, Imperva Cloud WAAP leaders*. URL: https://www.sdxcentral.com/articles/news/gartner-names-akamai-cloudflare-imperva-cloud-waap-leaders/2022/09/ (acedido em 10/10/2022).

Malik, Keshav (set. de 2023). *Types of penetration testing: A comprehensive guide*. URL: https://www.getastra.com/blog/security-audit/types-of-penetration-testing/ (acedido em 10/09/2023).

Martin-Lopez, Alberto, Sergio Segura e Antonio Ruiz-Cortés (2021). «RESTest: Automated Black-Box Testing of RESTful Web APIs». Em: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. Virtual, Denmark: Association for Computing Machinery, pp. 682–685. ISBN: 9781450384599. DOI: 10.1145/3460319.3469082. URL: https://doi.org/10.1145/3460319.3469082.

Matan, Liv (mai. de 2023). *Uncovering 3 azure API management vulnerabilities - when good apis go bad*. URL: https://ermetic.com/blog/azure/when-good-apis-go-bad-uncovering-3-azure-api-management-vulnerabilities/ (acedido em 07/08/2023).

Mirabella, A. Giuliano et al. (2021). «Deep Learning-Based Prediction of Test Input Validity for RESTful APIs». Em: *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*, pp. 9–16. DOI: 10.1109/DeepTest52559.2021.00008.

MODY, VIRAG (abr. de 2022). *How oauth 2.0 works*. URL: https://goteleport.com/blog/how-oauth-authentication-works/ (acedido em 19/08/2023).

OAuth (2023). *O que É o oauth 2.0 e o que ele Faz por você?* URL: https://auth0.com/pt/intro-to-iam/what-is-oauth-2 (acedido em 10/09/2023).

OpenAPI (2021). *OpenAPI Specification V3.1.0*. URL: https://spec.openapis.org/oas/v3.1.0 (acedido em 01/03/2023).

*OWASP API Security Top 10 2023 has been released* (2023). URL: https://owasp.org/blog/2023/07/03/owasp-api-top10-2023 (acedido em 05/08/2023).

*Owasp top 10 API security risks* (2023). URL: https://owasp.org/API-Security/editions/2023/en/0x11-t10/ (acedido em 06/08/2023).

Paloalto (2023). *What is web application and API protection?* URL: https://www.paloaltonetworks.com/cyberpedia/what-is-web-application-and-api-protection (acedido em 10/09/2023).

*PayPal Invoices API* (2023). URL: https://developer.paypal.com/docs/api/invoicing/v2/#invoices_create (acedido em 14/01/2023).

Pouya (2020). *Create post on any Facebook page*. URL: https://www.darabi.me/2020/12/create-invisible-post-on-any-facebook.html (acedido em 06/08/2023).

Prakash, Anand (set. de 2019). URL: https://web.archive.org/web/20191211192446/https://appsecure.security/blog/how-i-could-have-hacked-your-uber-account (acedido em 06/08/2023).

Reselman, Bob (jan. de 2023). *An architect's guide to apis: Soap, rest, GraphQL, and grpc*. URL: https://www.redhat.com/architect/apis-soap-rest-graphql-grpc (acedido em 01/03/2023).

*Retrospective: Recent Coinbase Bug Bounty Award* (fev. de 2022). URL: https://www.coinbase.com/blog/retrospective-recent-coinbase-bug-bounty-award (acedido em 25/10/2022).

Rodrigues, Bernardo (2022). *Autenticação de Mensagens via HMAC*. URL: https://pt.linkedin.com/pulse/autentica%C3%A7%C3%A3o-de-mensagens-via-hmac-bernardo-rodrigues (acedido em 10/09/2023).

*Salt security: State of API security report Q1 2023* (2023). URL: https://content.salt.security/state-api-report.html (acedido em 21/08/2023).

Seals, Author: Tara e Tara Seals (2021). *Data for 700m linkedin users posted for sale in Cyber-Underground*. URL: https://threatpost.com/data-700m-linkedin-users-cyber-underground/167362/ (acedido em 25/10/2022).

Sharieh, Salah e Alexander Ferworn (2021). «Securing apis and chaos engineering». Em: *2021 IEEE Conference on Communications and Network Security (CNS)*. DOI: 10.1109/cns53000.2021.9705049.

Shkedy, Inon (dez. de 2021). *Log4Shell – the API security challenge*. URL: https://securityboulevard.com/2021/12/log4shell-the-api-security-challenge/ (acedido em 19/08/2023).

Silva, Paulo e Rui Silva (2019). *OWASP API Security Top 10 2019*. URL: https://github.com/OWASP/API-Security/raw/master/2019/pt-pt/dist/owasp-api-security-top-10.pdf (acedido em 29/12/2022).

Souza, Evandro F. (jun. de 2018). *Protobuf - Uma Alternativa AO JSON E XML*. URL: https://medium.com/trainingcenter/protobuf-uma-alternativa-ao-json-e-xml-a35c66edab4d (acedido em 14/03/2023).

Tennant, Laurence (jul. de 2022). *Hunting for mass assignment vulnerabilities using github CodeSearch and grep.app*. URL: https://blog.includesecurity.com/2022/07/hunting-for-mass-assignment-vulnerabilities-using-github-codesearch-and-grep-app/ (acedido em 07/08/2023).

Tesauro, Matt (2023). *API security tools*. URL: https://owasp.org/www-community/api_security_tools (acedido em 18/08/2023).

Testfully (out. de 2021). *Introduction to API blueprint*. URL: https://testfully.io/blog/api-blueprint/ (acedido em 14/01/2023).

vegibit (2023). *An overview of data serialization formats – JSON, YAML, and XML*. URL: https://vegibit.com/an-overview-of-data-serialization-formats-json-yaml-and-xml/ (acedido em 14/03/2023).

Whittaker, Zack (jun. de 2019). *Millions of Venmo transactions scraped in warning over privacy settings*. URL: https://techcrunch.com/2019/06/16/millions-venmo-transactions-scraped/?guccounter=1 (acedido em 25/10/2022).

Xu, Alex (abr. de 2022). URL: https://blog.bytebytego.com/p/how-to-design-a-secture-web-api-access (acedido em 18/08/2023).

ZAP, OWASP (2022). URL: https://www.zaproxy.org/docs/api/ (acedido em 29/10/2022).

# APÊNDICES

# APÊNCICE A - ZAPI

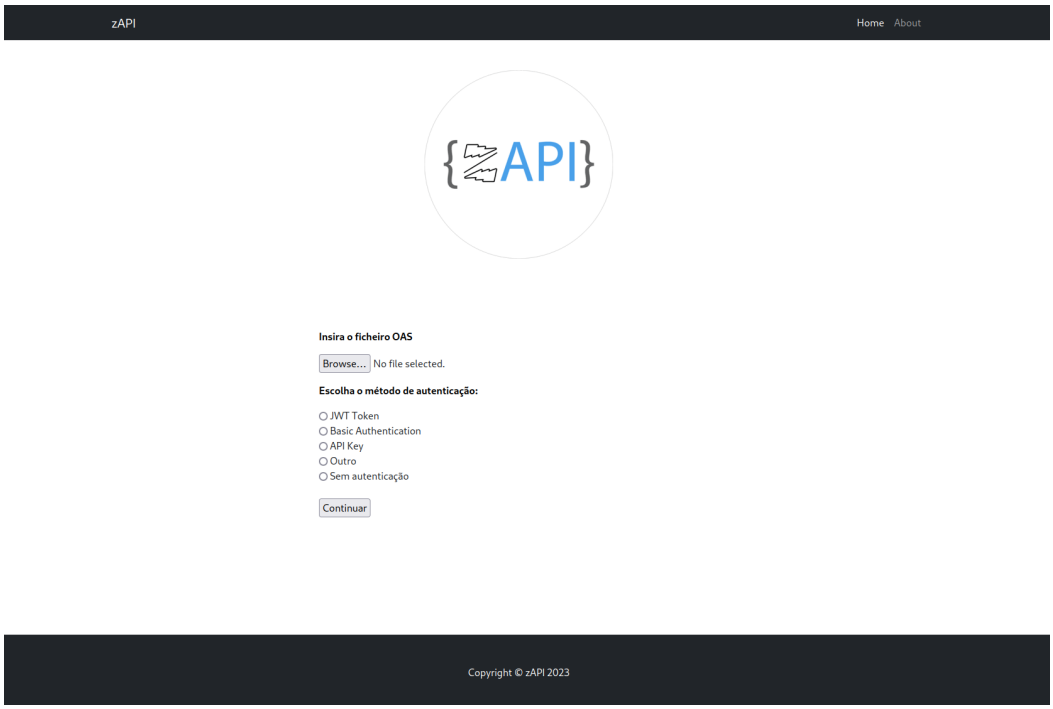


Figura 33: Página inicial

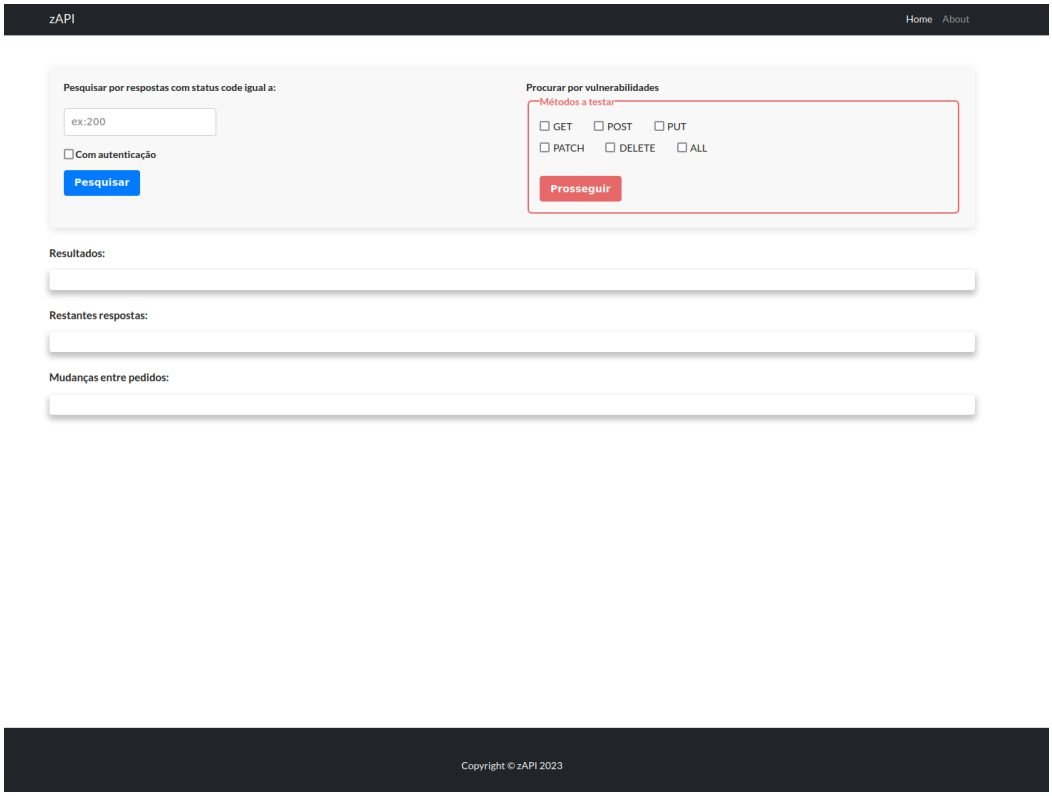


Figura 34: Página para análise manual e automatizada

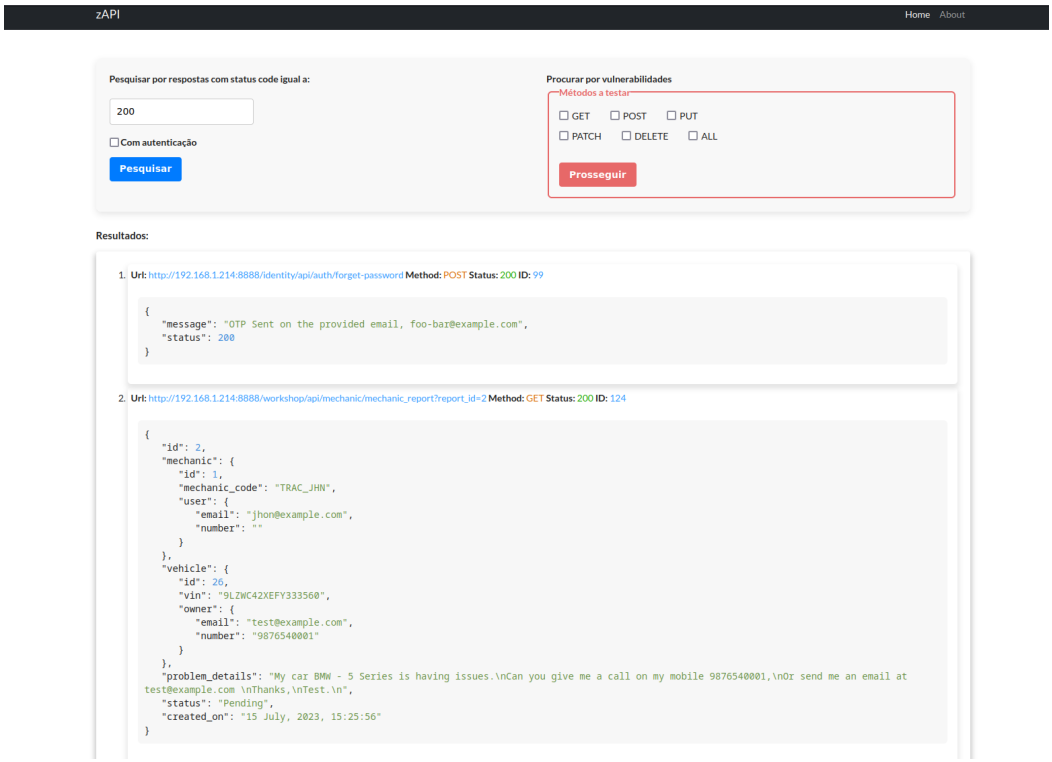


Figura 35: Página com dados preenchidos conforme os filtros

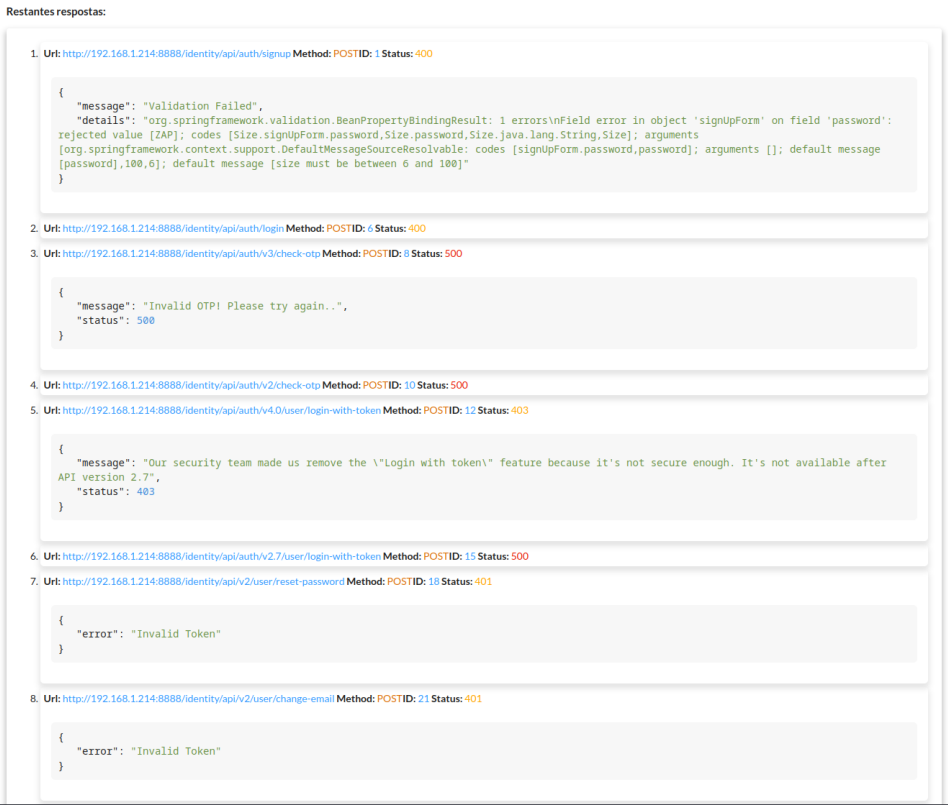


Figura 36: Página com dados preenchidos conforme os filtros - continuação

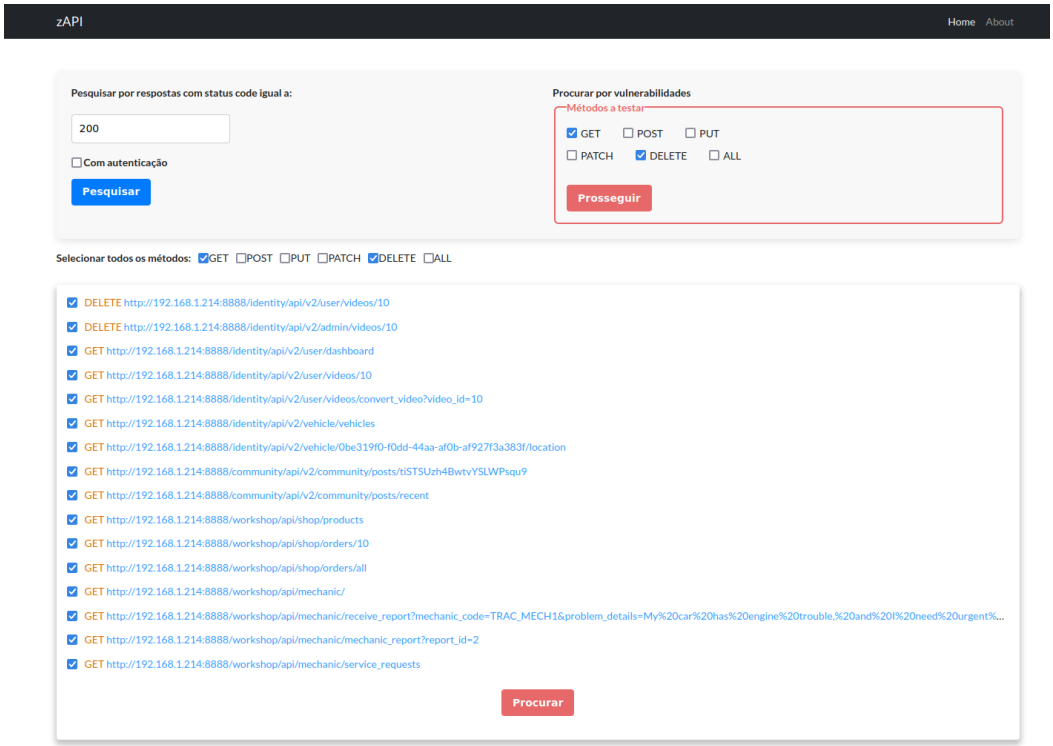


Figura 37: Escolha dos métodos a testar durante a análise automatizada

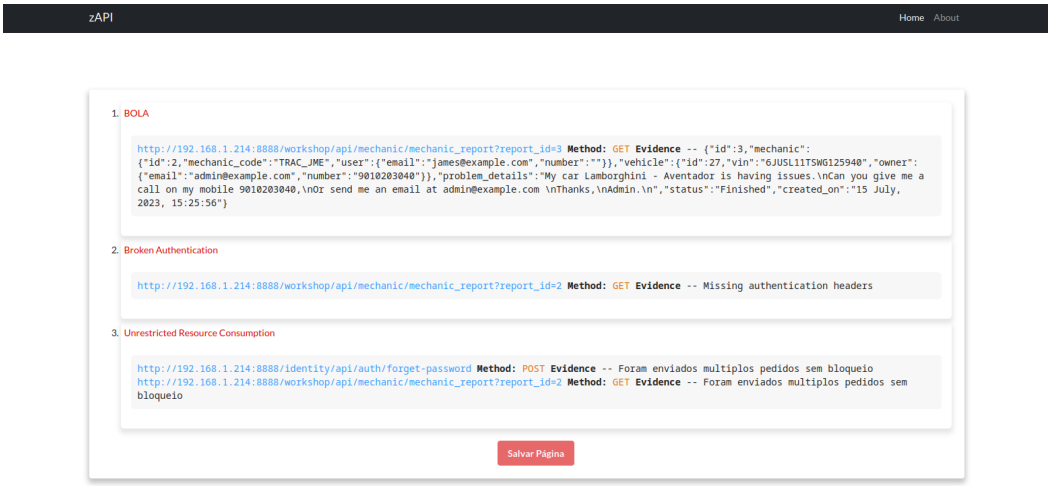


Figura 38: Página de resultados

# B

# APÊNCICE B - RESULTADOS DAS FERRAMENTAS ALTERNATIVAS

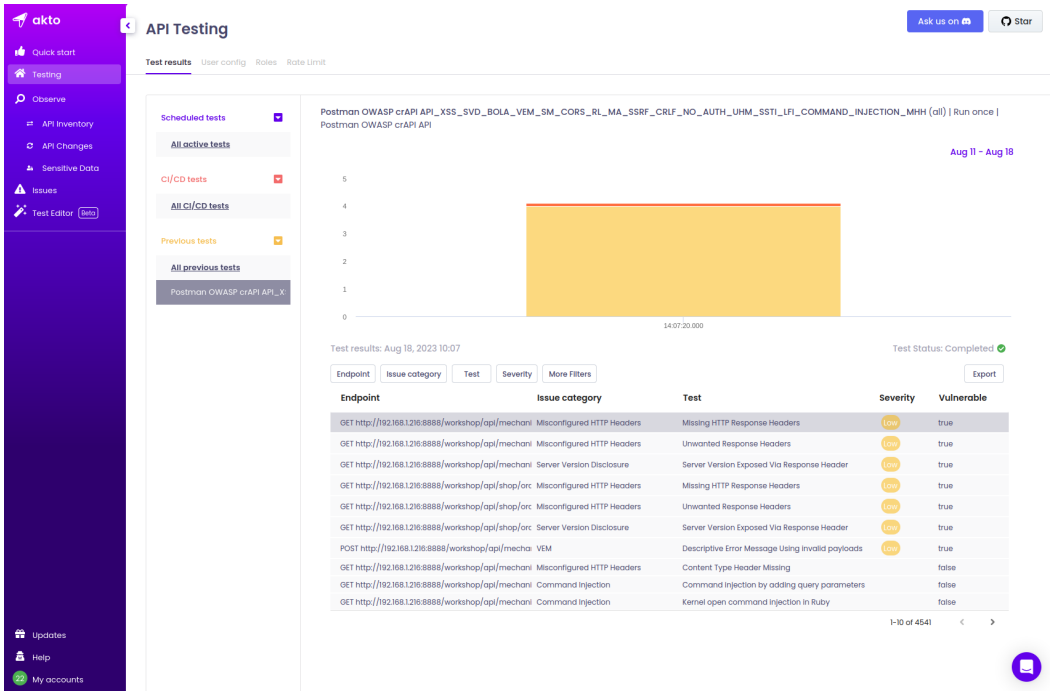


Figura 39: Akto - crAPI

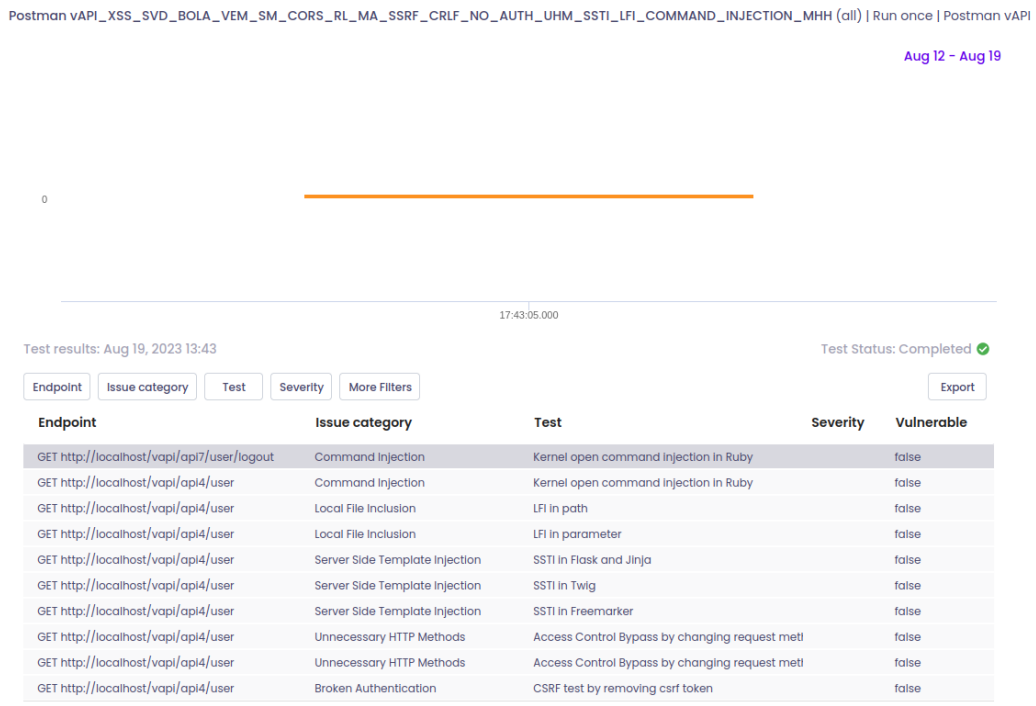


Figura 40: Akto - vAPI

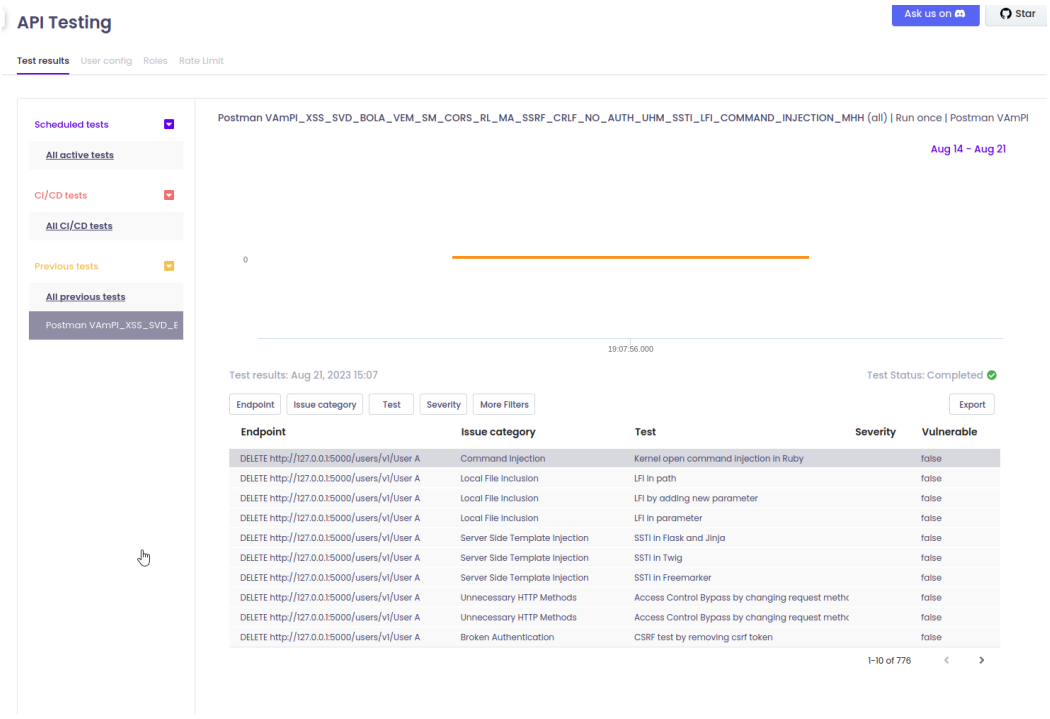


Figura 41: Akto - VAmPI

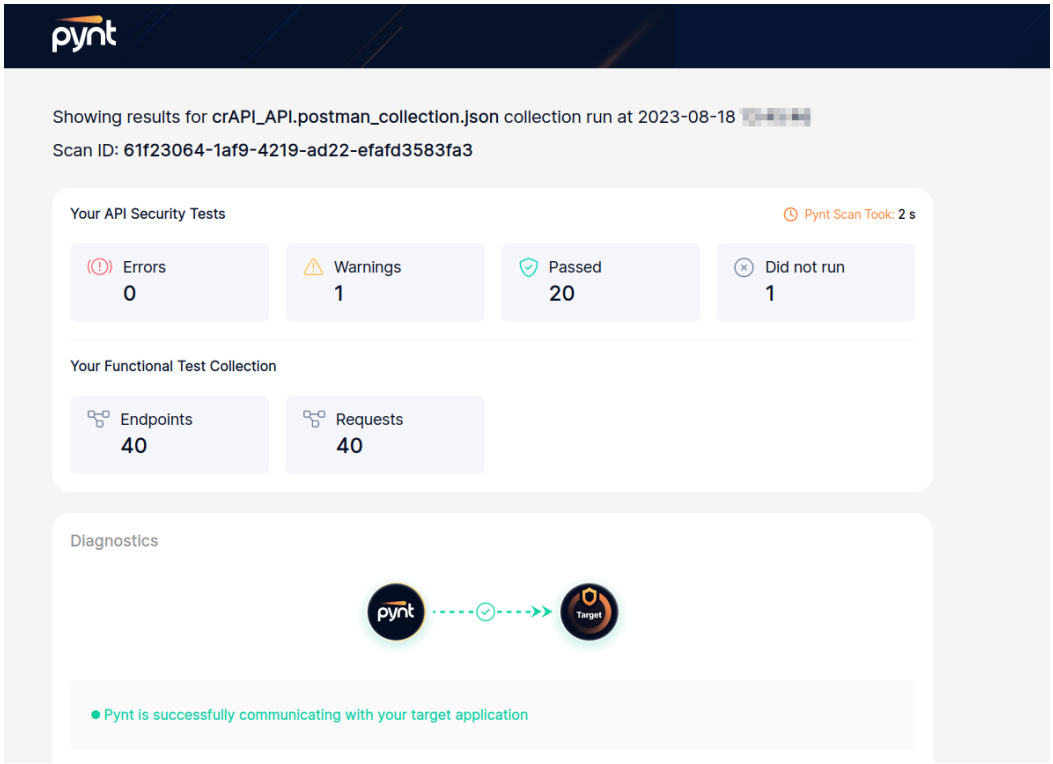


Figura 42: Pynt - crAPI

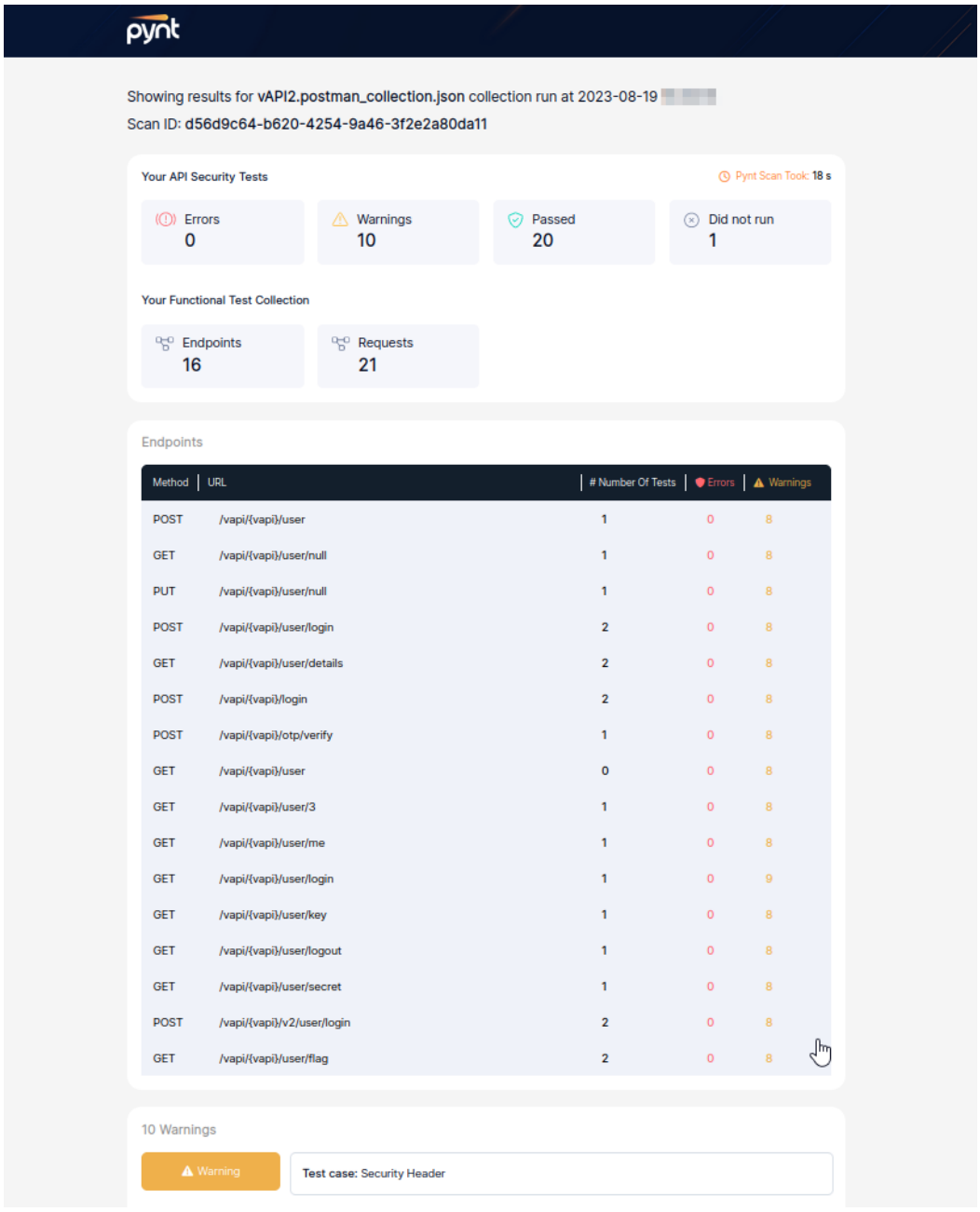


Figura 43: Pynt - vAPI

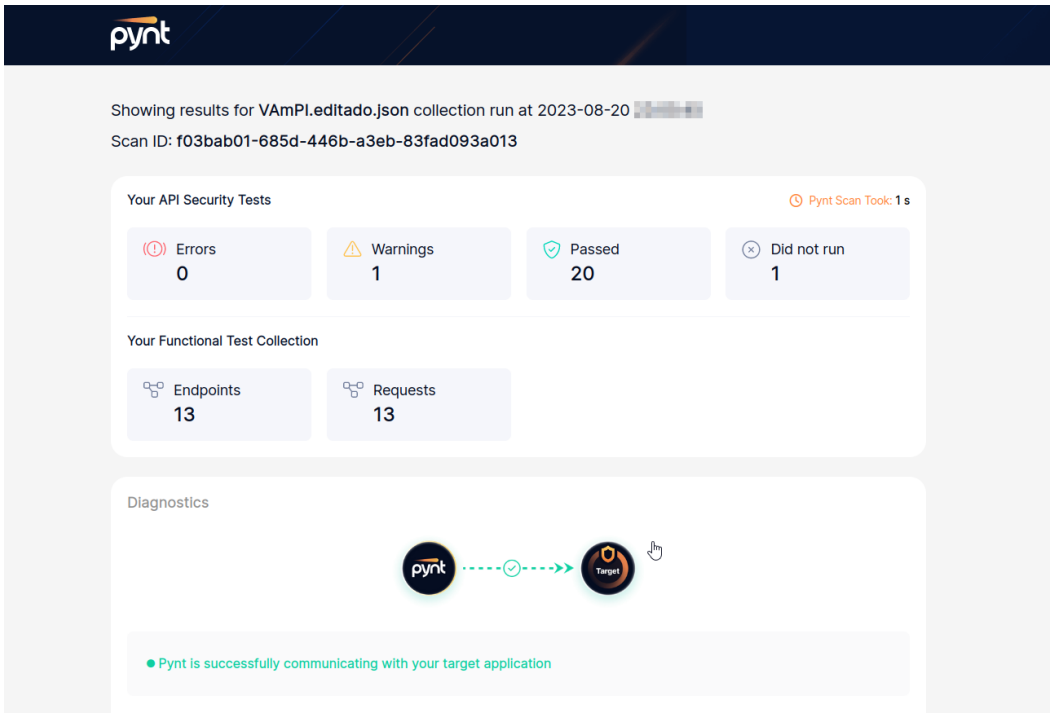


Figura 44: Pynt - VAmPI

# DECLARAÇÃO

Declaro, sob compromisso de honra, que o trabalho apresentado neste projeto, com o título *"ZAPI - Ferramenta para análise de vulnerabilidades em APIs"*, é original e foi realizado por António Pedro Machado Pinto (2212981) sob orientação de Professor Doutor Marco António de Oliveira Monteiro (marco.monteiro@ipleiria.pt).

*Leiria, Setembro de 2023*

Assinado por: **António Pedro Machado Pinto**
Num. de Identificação: 15633489
Data: 2023.09.29 18:52:59+01'00'


António Pedro Machado Pinto