

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Security Testing of Web APIs

Gonçalo André Carneiro Teixeira



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Hugo Pacheco

Second Supervisor: Prof. Nuno Macedo

October 6, 2023

Security Testing of Web APIs

Gonçalo André Carneiro Teixeira

Mestrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Prof. José Manuel de Magalhães Cruz

Arguente: Prof. Nuno Antunes

Vogal: Prof. Hugo José Pereira Pacheco

Vogal: Prof. Nuno Macedo

October 6, 2023

Resumo

Nos dias atuais temos visto uma crescente oferta de APIs RESTful para serviços Web e, com este crescimento, surge também uma série de novos desafios. Um desses desafios é garantir que a interface é robusta, não só em termos de *correctness*, mas também em termos de segurança. Atualmente, muitas ferramentas conseguem fornecer validação de *correctness*, seja por testes unitários ou testes de integração; quanto à segurança, temos visto o aparecimento de novas ferramentas e *frameworks*, mas o desenvolvimento e investigação nesta área são ainda muito escassos. Para um serviço *production-ready*, é tão essencial ser resistente a erros quanto ser seguro, especialmente com novas regulações de dados a serem aplicadas a cada ano.

As APIs Web geralmente são documentadas por um *schema* OpenAPI, que especifica os *end-points*, parâmetros, resultados e valores de *input*. Ultimamente, as ferramentas de testes automáticas têm recorrido crucialmente à informação extraída dos *schemas* para derivar semântica entre produtores e consumidores (um consumidor de um recurso do tipo A usa como *input* o resultado do produtor para o recurso do tipo A) e usam essa informação para gerar sequências de pedidos com base nas relações produtor-consumidor. Essas ferramentas têm detetado *bugs*, geralmente denotados por códigos 5xx, utilizando técnicas de *fuzzing* para gerar *inputs* aleatórios com base em dicionários para o tipo de *input* (declarado no *schema*).

Embora os *schemas* OpenAPI sejam uma excelente maneira de definir a estrutura da API, não está claro como é possível que possam capturar propriedades de segurança relevantes; um dos objetivos desta dissertação é estudar como é que propriedades de segurança podem ser expressas sobre o padrão OpenAPI com extensões mínimas.

As ferramentas de testes automáticos estão a ficar melhores a encontrar problemas que causam uma falha no serviço (por exemplo, códigos 5xx), mas atualmente estão limitadas na capacidade de verificar propriedades de segurança. Colecionar propriedades de segurança para avaliar e exemplos de violações são dois outros objetivos desta dissertação; os exemplos ajudarão a orientar a especificação de propriedades de segurança relevantes, o desenvolvimento de extensões de APIs ou ferramentas e o teste de violações de propriedades de segurança, ao mesmo tempo que é feita uma investigação e são gerados testes eficientes automatizados para as propriedades de segurança propostas, e, finalmente, como é que esses testes podem ser adicionados a uma ferramenta de testes existente, contribuindo para tornar as APIs em produção mais seguras. Para a validação de resultados, uma série de testes em Web APIs em ambiente local e de produção será conduzida e detalhada neste documento, fornecendo uma base para trabalho futuro neste tópico.

Palavras-chave: Web API, testes de segurança, ferramentas de teste automaticas, fuzzing, property-based testing, OpenAPI

Abstract

In these modern days, we see a growing offer of RESTful APIs for web services, and with this growth, a series of new challenges also arise. One of these challenges is ensuring the interface is robust, not only in terms of correctness but also in terms of security. Currently, many tools can provide correctness validation, either by unit testing or integration testing; as for security, we are now seeing the emergence of new tools and frameworks, but research in this area is still very scarce. For a production-ready service, it is as essential to be resistant to errors as it is to be secure, with new data regulations being enforced every year.

Web APIs are usually documented by an OpenAPI schema, stating the endpoints, parameters, output results, and values. Lately, automated testing tools have been crucially resorting on the information extracted from schemas to derive semantic relationships between producers and consumers (a consumer of a resource of type A uses the output of the producer for the resource of type A), and using this information to generate sequences of requests based on producer-consumer relationships. These tools often detect bugs, usually denoted by 5xx status codes, using *fuzzing* techniques, generating random inputs based on dictionary files for the kinds of inputs declared on the schema.

Although OpenAPI schemas are an excellent way to define the API structure, it is not clear how they can capture relevant security properties; one of this work's objectives is to study how can security properties be expressed over the OpenAPI standard with minimal extensions.

Automated testing tools are becoming good at finding problems that cause a malfunction on the service (e.g., 5xx status codes), but they are currently limited in their capacity to verify security properties. Collecting security properties to evaluate, and examples of breaches are two other objectives of this dissertation; the examples will help guide the specification of relevant security properties, the development of API or tool extensions, and the testing for security property violations while researching and generating efficient automated tests for the proposed security properties and, finally, how they can be added to an existing testing automation tool, contributing to making production-ready APIs more secure. For result validation, a series of tests on local and production-ready APIs from the collected examples will be detailed in this document, providing a basis for further work on this subject.

Keywords: Web API, security testing, automated testing tools, fuzzing, property-based testing, OpenAPI

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Hugo Pacheco for his unwavering support and guidance throughout the course of this research. His expertise in the field has been invaluable, and his enthusiasm for the subject has been a constant source of inspiration. Your insightful feedback and constructive criticism have been instrumental in shaping this thesis into what it is today. I am also extremely grateful to my second supervisor, Professor Nuno Macedo, whose deep insights and rigorous approach have been truly enlightening. Your encouragement and intellectual challenges have added layers of depth to my work and have helped me to grow as a researcher.

Special recognition is also extended to my managers at work, who have been extraordinarily accommodating in allowing me the flexibility to pursue this research alongside my professional responsibilities. The opportunity to balance work, life, and school has been essential for the successful completion of this thesis, and for that, I am deeply thankful.

Special thanks are due to my friends who have stood by me during the course of this journey. Your support, and, at times, much-needed distractions have made this arduous process a bit more bearable. You have all contributed, in your own unique ways, to the completion of this thesis. In the topic of friends, I could not leave a special one unmentioned, someone who I consider a brother, who's been with me from day one. I am forever thankful for the wise words, sometimes harsh but much needed, thankful for the journey we've shared together in this area, looking forward to what comes next.

I would also like to extend my heartfelt appreciation to my family. Your unwavering belief in me has been a constant source of strength. Thank you for your love, encouragement, and for the sacrifices you've made to support me throughout my academic pursuits. A special thank you to my father who, for years, has been very far away to provide for me and my family so that we could continue pursuing our dreams, I will be forever grateful for your sacrifices and teachings.

Here, I would like to add something special for my girlfriend, my "partner in crime", who has been with me since the very beginning, when I first started thinking about my future, who has been an incredible source of love, support, and motivation. It has been a beautiful journey, with many more miles to go. I am looking forward to the challenges and adventures that life has prepared for us, always holding the memories we have made very close to my heart. An exceptional thank you.

Lastly, I am indebted to all those who have made contributions, big or small, to my academic journey but have not been mentioned here. Your help has not gone unnoticed and is highly appreciated.

Thank you.

Gonçalo Teixeira

*“Because sometimes even if you know how something’s going to end,
that doesn’t mean you can’t enjoy the ride.”*

Ted Mosby, in *How I Met Your Mother*

Contents

1	Introduction	1
1.1	Motivation for Security Testing on Web APIs	1
1.2	OWASP Top Weaknesses	2
1.3	Objectives & Methodology	4
2	RESTful Web APIs & Security	5
2.1	What is REST?	6
2.2	OpenAPI Standard	6
2.3	API Prototype Overview	7
2.3.1	Mutator Requests	9
2.4	Security Properties	10
2.4.1	Broken Object & Function Level Authorization	11
2.4.2	Immutability in Subsequent GET Requests	12
2.4.3	Non-Interference Between an Out-of-Context Mutator Request	12
2.4.4	Mutability Between a Mutator Request	13
2.5	OpenAPI & Security Validation	14
3	Testing of Web APIs	17
3.1	Fuzz Testing & Property-based Testing	17
3.2	RESTler	18
3.3	Schemathesis	22
3.4	Metamorphic Testing	24
3.5	RESTest	27
3.6	Online Testing of Web APIs	28
3.7	Synthesis and Future Endeavors in Web API Testing	29
4	A Closer Look Into Schemathesis	31
4.1	Schemathesis Under The Hood	31
4.2	Stateful Testing in Depth	32
4.3	Hypothesis	33
4.4	Shrinking: Honing in on Minimal Failures with Hypothesis	35
4.5	Extensibility and Flexibility of Schemathesis	36
4.6	Summary and Key Takeaways	40
5	XSecEngine	41
5.1	OpenAPI Extensions for Security Testing	42
5.2	Checking Security Properties	45
5.2.1	Broken Object & Function Level Authorization	45

5.2.2	Immutability in Subsequent GET Requests	47
5.2.3	Non-Interference Between an Out-of-Context Mutator Request	49
5.2.4	Mutability Between a Mutator Request	50
5.3	Implementation Details	52
6	Evaluation and Results	54
6.1	Web API Prototype	54
6.2	FusionAuth	57
6.3	Discussion of Results	59
6.4	Test Scenario Implementation Complexity	60
6.5	Limitations	62
7	Conclusions	63
7.1	Future Work	64
	References	66
A	Test Workflow Implementations	70
B	Links Graph Implementation	75

List of Figures

5.1	XSecEngine Architecture Diagram	41
6.1	Web API Prototype Links Graph	56
6.2	FusionAuth Links Graph	58

List of Tables

2.1	Users' initial information, stored in the database	9
3.1	Systems tested as part of the evaluation for Schemathesis	24
6.1	Prototype API Test Scenarios	55
6.2	XSecEngine: Prototype API Test Results (values in seconds)	56
6.3	RESTler: Prototype API Test Results (values in seconds)	57
6.4	FusionAuth API Test Scenarios	58
6.5	XSecEngine: FusionAuth API Test Results (values in seconds)	59

Listings

2.1	Prototype Security Scheme	14
2.2	GET User Security Scopes	14
2.3	OpenAPI Links Example	15
4.1	Additional Hypothesis Strategies Example	36
4.2	Hypothesis Rule Creation	38
4.3	Dynamic Hypothesis Rule Creation	38
5.1	Schema Example for Users' API	43
5.2	Vulnerability in GET /api/users/:user_email	45
5.3	Broken Object & Function Level Authorization Rule	46
5.4	Side-effect in GET /api/users/:user_email	47
5.5	Immutability in Subsequent GET Requests Rule	48
5.6	Side-effects in PATCH /api/users/:user_email	49
5.7	Non-Interference Between an Out-of-Context Mutator Request Rule	50
5.8	Update Bypass in PATCH /api/users/:user_email	50
5.9	Mutability Between a Mutator Request Rule	51
5.10	Boilerplate for Test Workflow	52
A.1	Prototype API Test Workflow Implementation	70
A.2	FusionAuth Test Workflow Implementation	72
B.1	Links Graph Implementation	75

Abbreviations

API	Application Programming Interface
CWE	Common Weakness Enumeration
HTTP	Hypertext Transfer Protocol
JSON	Javascript Object Notation
JWT	JSON Web Token
LOCs	Lines of Code
OAS	OpenAPI Specification
REST	Representational State Transfer
URI	Uniform Resource Identifier

Chapter 1

Introduction

In today’s digital ecosystem, Web APIs form the backbone of many services and applications. They serve as conduits, linking disparate systems and enabling a symphony of functionalities. Given this centrality, ensuring the robustness and security of these interfaces becomes paramount. To comprehend the facets of this challenge, it is pivotal to delineate two intertwined yet distinct notions: **API correctness** and **API security**. While this thesis is rooted in the domain of security, an understanding of both dimensions is imperative.

Correctness — Within the vast landscape of software applications, correctness acts as the yardstick of its operational fidelity. It gauges whether an application adheres to its blueprinted specifications and demonstrates resilience against aberrant or malformed data inputs. In essence, it examines if the application performs reliably, upholding its intended behavior amidst unforeseen contingencies.

Security — Transcending mere operational reliability, security delves into fortifying applications against many external threats. Drawing from the repository of Common Weakness Enumerations [7], security measures address potential vulnerabilities, forestalling breaches, and unauthorized accesses. In Web APIs, this translates to stringent access controls, resistance to various attack vectors like injections, and meticulous data protection to thwart inadvertent disclosures.

The spotlight on Web API security has intensified recently, echoing the community’s growing cognizance of its significance. However, the journey toward automating the security validation process remains nascent. Conventionally, developers have resorted to crafting bespoke test cases for each unique security scenario, a daunting endeavor given the vast expanse of potential vulnerabilities. Some of these security nuances are so intricate that conventional unit tests falter in encapsulating them, thereby necessitating disproportionate manual efforts.

1.1 Motivation for Security Testing on Web APIs

As gateways to the digital realm, Web APIs play a pivotal role in connecting, integrating, and enabling various software systems. However, their very prominence makes them lucrative targets for cyber threats. Vulnerabilities in an API not only jeopardize its integrity but can cascade security

lapses across all systems hinged to it. Thus, an insecure API can potentially unveil a trove of sensitive data or functionality to ill-intentioned actors.

Security testing transcends the paradigms of conventional functional testing. While the latter examines the operational correctness of an application, the former grapples with an expansive spectrum of potential attacks, making the task intricate and labor-intensive. Notwithstanding the complexities, the imperative for robust security testing is glaringly evident — especially with the aspiration of automation in sight. Moreover, while the momentum around security is gradually mounting within the community, there remains a notable paucity of dedicated resources and efforts in this domain.

The inertia in advancing security testing can be attributed to several factors. Crafting test scenarios for specific security properties is daunting; scaling these to a generalized testing utility amplifies the challenge manifold. This complexity stems from the myriad architectures underpinning Web APIs and the heterogeneity in their specifications. While standards do exist to guide the design of Web APIs, they often suffer from discretionary adherence, further muddying the waters.

Presently, the toolscape for automated Web API testing seems rather myopic, with a predominant focus on pinpointing operational glitches — typically flagged by 5xx status codes — arising from diverse request inputs. Yet, within this assortment, only a handful truly cater to the nuanced demands of security. And even among these, the breadth of security properties they encompass remains disappointingly narrow.

In essence, as Web APIs continue to undergird our digital infrastructure, the quest for comprehensive, automated security testing tools remains both a challenge and an imperative for the community.

1.2 OWASP Top Weaknesses

The Open Web Application Security Project (OWASP) [28] is a nonprofit foundation that works to improve the security of software. Through community-led open-source software projects, hundreds of local chapters worldwide, tens of thousands of members, and leading educational and training conferences, the OWASP Foundation is the source for developers and technologists to secure the web.

From the OWASP Top 10 vulnerabilities for 2021 [29], most of the categories enumerated by OWASP will not be the focus of this work since they relate to topics like code injection, cross-site scripting, and cryptographic errors, these types of vulnerabilities are usually related to the technology or framework behind the Web API, and not to the system’s semantics and functionality; moreover, these types of vulnerabilities can be tested by *black-box* approaches, and examples can be found in the article *Metamorphic Testing for Web System Security* [6]. This work will be more focused on the fourth position: “A04:2021 - Insecure Design”, with 40 CWEs¹ (Common

¹Common Weakness Enumeration [7] (CWE) is a community-developed list of common software and hardware weakness types with security ramifications. A “weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. The CWE List and associated classification taxonomy serve as a language that can identify and describe these weaknesses in terms of

Weakness Enumeration) mapped. Even though OWASP [29] describes this category as “broad”, the foundation emphasizes that insecure design is not the source of all the other Top 10 risk categories; they also provide a means of differentiation between design flaws and implementation defects,

They have different root causes and remediation. A secure design can still have implementation defects, leading to vulnerabilities that may be exploited. An insecure design cannot be fixed by a perfect implementation as by definition, needed security controls were never created to defend against specific attacks. One of the factors that contribute to insecure design is the lack of business risk profiling inherent in the software or system being developed, and thus the failure to determine what level of security design is required.

Even though the aforementioned category is considerably broad, the notes and examples provided by the foundation can help guide the study and research for this work.

OWASP also launched an API security project, and the most recent Top 10 risks are from 2019 [26], with “Broken Object Level Authorization” at the head of the list; this risk category is a specific type of Insecure Direct Object Reference (IDOR) attack. The foundation states [26]:

Attackers can exploit API endpoints vulnerable to broken object-level authorization by manipulating the ID of an object sent within the request. This may lead to unauthorized access to sensitive data. This issue is extremely common in API-based applications because the server component usually does not fully track the client’s state and relies more on parameters like object IDs sent from the client to decide which objects to access.

A practical example of this type of vulnerability can be expressed as follows: an e-commerce platform has an API in its core to generate receipts after purchase; the receipt can be viewed by clicking on a button that opens a new tab on the client’s browser; internally, the website makes a GET request to `/api/receipts/{receipt_id}` to fetch the PDF from the server. An attacker can inspect the website’s traffic and realize the request is formed with a `receipt_id` and then proceed to try to make a request with a different ID. A compromised system will accept the request, even though the attacker should not have permission to access the resource, thus incurring in a broken object-level authorization vulnerability.

OWASP [26] suggests some actions for the developers to take in order to prevent these types of attacks: implement a proper authorization mechanism that relies on the user policies and hierarchy; use an authorization mechanism to check if the logged-in user has access to perform the requested action on the record in every function that uses an input from the client to access a record in the database; prefer to use random and unpredictable values as GUIDs for records’ IDs; write tests to evaluate the authorization mechanism and do not deploy vulnerable changes that break the tests.

Part of the objectives for this work is to pave the way for detecting these types of vulnerabilities automatically, possibly by extending the existing state-of-the-art tools.

OWASP API Security Project [26] Top 10 Risks for APIs, like OWASP general Top 10 risks [29], also enumerate risks in injection, cross-site scripting and security configurations, which, once more, will not be the focus of this work; instead, this work will focus on the semantic and logic parts of the API related to security.

1.3 Objectives & Methodology

Anchoring this research, we embark on a quest to dissect specific security properties, probe the potential of augmenting an existing tool to assess these properties, and subsequently gauge the efficacy of this augmentation. Our expedition is underpinned by three research questions:

- RQ1** — Does the OpenAPI Specification offer avenues for extending its purview to encapsulate security properties, transcending its conventional security schema?
- RQ2** — What’s the feasibility and complexity of enhancing an existing testing tool to encompass a broader set of security properties?
- RQ3** — How proficiently do the expanded OpenAPI and testing tool configurations discern security breaches?

To navigate these questions, we commence by immersing ourselves in contemporary Web API testing paradigms and charting potential pathways for their enhancement. Furthermore, we contemplate embedding security properties directly into the OpenAPI schema or at the very least, create documentation methods to guide automated testing mechanisms.

The ensuing chapters unfold a holistic narrative. Initially, in Chapter 2, we elucidate core tenets surrounding Web APIs, encompassing RESTful architecture, the OpenAPI Standard, and an exposition of the targeted security properties. This foundational understanding is bolstered by a Web API prototype crafted specifically for this research. Subsequent chapters unravel the contemporary state-of-the-art (Chapter 3), underscoring existing inadequacies. Delving deeper, in Chapter 4, we explore Schemathesis (a stateful, property-based testing tool for Web APIs), elucidating its pivotal role and implications for our endeavor. This exploration continues with an empirical study (Chapter 5), shedding light on our enhancements to both Schemathesis and OpenAPI and culminates in an evaluative discourse (Chapter 6), reflecting on our findings and their broader implications. In the final notes, we chart potential trajectories for future exploration in this domain.

Chapter 2

RESTful Web APIs & Security

In an era where data exchange and system integrations are paramount, Web APIs have emerged as the backbone for many digital applications. They bridge the gap between platforms, allowing systems to communicate and share information seamlessly. However, as the popularity and reliance on Web APIs grow, so does the need to ensure their security. This chapter delves into the intricacies of RESTful Web APIs and the essential aspect of security testing, spotlighting the areas where vulnerabilities may arise and how we might mitigate them.

Firstly, we embark on a journey to understand REST, the architectural style introduced by Fielding. Recognizing the foundational principles behind REST is crucial as it forms the bedrock upon which many modern Web APIs are built. A clear understanding of this architecture not only provides insights into its design choices but also underscores the potential security implications.

We then shift our focus to the OpenAPI Specification (OAS). As a prevailing standard for Web API design, understanding OAS is imperative for anyone looking to grasp the nuances of modern Web API construction and documentation. However, while OAS offers comprehensive specifications for developing APIs, does it sufficiently address their security requirements?

This chapter also presents an overview of an API prototype built using the FastAPI framework. By walking through its implementation aspects, readers will gain firsthand insights into the practical challenges and solutions associated with creating and testing secure Web APIs.

A pivotal part of this chapter, and indeed this thesis, revolves around the identification and documentation of specific security properties. These properties set the stage for the subsequent chapters, highlighting the facets of security that this research seeks to address.

Lastly, we delve deeper into the intersection of OAS and security validation. Here, we probe the pressing question: Is OAS adequate for validating the security properties we have identified? Through this exploration, readers will be equipped with an enriched perspective on the existing gaps in Web API security and the road ahead.

In essence, this chapter establishes the foundational knowledge and context for the discussions that follow. It underscores the importance of security in Web APIs and the multifaceted challenges researchers and developers face in ensuring a robust and secure digital ecosystem.

2.1 What is REST?

According to Fielding's definition in his Ph.D. thesis [12], in 2000 — *Architectural Styles and the Design of Network-based Software Architectures* — REST (Representational State Transfer) is an architectural style for designing network-based software architectures. Fielding defines REST as a set of architectural constraints that aim to create a scalable, modular, and flexible distributed architecture for the World Wide Web.

Fielding [12] characterizes REST by a set of architectural constraints that define how the components of a system should interact and the properties they should have. These constraints include the use of a uniform interface, which describes how components communicate with each other using a standard set of operations and data representations; the use of a layered system, which allows for the separation of concerns among different components; and the use of statelessness, which means that each request made to a component should contain all the information needed to understand it, without relying on the component's previous state or the context of previous requests. Regarding this last sentence, it is certainly desirable for an API to be stateless, but that is not always achievable, leaving some methods stateful.

Additionally, Fielding [12] states REST is designed to be scalable and flexible, allowing for the creation of complex distributed systems that can evolve and adapt to changing requirements. It is also intended to be easy to understand and implement, using familiar concepts and technologies from the World Wide Web, such as URLs, HTTP, and XML.

Given the definition of the REST architecture, Richardson and Ruby, in their book, *RESTful Web Services* [33], define RESTful Web APIs as *Web services that adhere to the REST architectural style*, they explain that RESTful APIs are designed to be scalable, modular, and flexible, and they use HTTP methods and URLs to expose a set of resources that can be manipulated using a set of standard operations.

2.2 OpenAPI Standard

In recent years, the proliferation of Web Application Programming Interfaces (APIs) has been nothing short of remarkable. APIs, functioning as conduits for system interactions, come clothed in diverse programming languages with potentially varied architectural bases. For an end-user or client making requests to such an API, its underpinning architecture and design might be a mystery. Hence arose the imperative need for a uniform documentation mechanism.

Enter JSON Schema and the OpenAPI Specification (OAS) — two of the front-runners in API documentation. The latter, formerly known as Swagger, has etched its place as the de facto industry standard. Significantly, the latest iteration of OAS, version 3.1.0, incorporates JSON Schema, amplifying its capacity to validate the data exchanged between the client and the API.

JSON Schema [19] stands out as a declarative language tailored for annotating and validating JSON documents. It offers dual benefits: a concise documentation medium accessible to both developers and automated systems and a robust mechanism to uphold the integrity of client input

data. The efficacy of JSON Schema extends to facilitating automated tests, ensuring consistent data quality.

The OpenAPI Specification, on the other hand, is a beacon of standardization for RESTful APIs [25]. While it embraces the foundational tenets of JSON Schema, it goes a step further, embellishing the syntax with features intrinsic to API definition. Think of OAS as the Rosetta Stone of APIs—it elucidates operations, parameters, and responses, serving as a guide for both man and machine.

Contrasting with JSON Schema’s specialized focus on JSON document validation, OAS is a broader canvas, painting a vivid picture of an API’s capabilities. It sheds light on nuances such as the API’s base URL, permissible HTTP methods, valid parameter types, and the nature of response data. Furthermore, OAS takes the developer experience up a notch by exemplifying requests and responses, offering an intuitive grasp of the API’s *modus operandi*.

In sum, while JSON Schema fortifies data integrity and clarity, OAS stands as a lighthouse for developers navigating the vast seas of APIs, illuminating their path and enhancing their understanding.

2.3 API Prototype Overview

In order to facilitate development and to also make the concepts’ explanations clearer and tangible in a real-world perspective, a simple Web API prototype was developed using the FastAPI [10] framework. FastAPI is a modern, fast web framework designed for building Web APIs using Python. It is known for its simplicity, efficiency, and scalability. FastAPI capitalizes on Python type annotations to facilitate the automatic validation of requests and responses, as well as the generation of API documentation (OpenAPI). Some interesting features of FastAPI include the following:

1. **Swiftness** — FastAPI [10] is built on the foundation of Starlette [39], an asynchronous web framework, and harnesses the exceptional performance capabilities of underlying libraries such as Pydantic [30] and `asyncio`. This empowers FastAPI [10] to handle heavy workloads efficiently and process requests expeditiously.
2. **User-friendliness** — FastAPI [10] provides a user-friendly and intuitive API for defining routes, handling requests, and working with data models. It supports standard HTTP methods (e.g., GET, POST, PUT, DELETE) and enables the definition of API endpoints using Python functions.
3. **Type annotations and validation** — FastAPI [10] leverages Python’s type hints to automatically validate request and response data against predefined data models. This approach reduces errors and enhances documentation quality. Additionally, FastAPI generates interactive API documentation conforming to the OpenAPI standard.

4. **Asynchronous support** — FastAPI [10] has been designed to accommodate asynchronous programming using Python's `asyncio` library. It enables the creation of asynchronous code for request handling, database operations, and other Input/Output (I/O) bound tasks. Consequently, FastAPI is particularly well-suited for constructing high-performance web applications.
5. **Extensibility** — FastAPI is easily extendable, allowing for the incorporation of additional functionalities through middleware, dependency injection, and custom components. It seamlessly integrates with other Python libraries and frameworks, facilitating the combination of FastAPI with existing tools and services.

FastAPI [10] has gathered significant popularity within the Python community due to its performance, user-friendly interface, and robust features. It is commonly used for constructing Web APIs, microservices, and backend systems that need high performance and scalability. For all the simplicity that FastAPI [10] brings, as well as the fact that it follows the standards, namely OpenAPI, while dynamically generating a fully documented schema, FastAPI [10] was the framework of choice for developing this prototype.

In terms of vertical architecture, this prototype application establishes a connection to a PostgreSQL containerized database in order to perform operations on models. Finally, most endpoints available require authentication and authorization using OAuth 2.0 [18].

The API can be seen as a user management system with authorization policies. Users cannot view or change other users' information unless they are system administrators, e.g., User A cannot request information about User B. Some of these policies were purposely left unchecked in some testing scenarios to guide the development towards security testing, while other bugs were introduced for the same goal, e.g., bypassing authorization checks, introducing random values in the response to simulate non-deterministic responses, etc.; with this in mind, this application is in no way safe to use in production environments.

The endpoints use a Bearer token (JSON Web Token, JWT) for authentication and authorization; this token contains information about the entity performing the request: the **payload** contains information relevant to the application, for this API, the `email` claim was introduced in the payload, in addition to the `scopes` claim OAuth 2.0 introduces; apart from the payload, the JWT also contains an **header** (information about the algorithm used and token type) and a **signature** (to validate the token's authenticity). The JWT alone is not enough to authenticate an entity and provide authentic context to the API; for this to take effect, the system needs to be able to verify and parse the JWT contents. The system can check if the JWT is legitimate by checking the signature, and for that, the secret key used to sign the token is needed, as well as the algorithm type used to encrypt the token. This API prototype also provides an endpoint to generate a valid token with associated scopes. However, in a production setup, this management is usually done via an authorization and authentication platform.

In order to better explain the following concepts, the prototype will be used. The user's initial information is available in Table 2.1. **Users** will be referred to by their name, which is not a unique

attribute, but for this experiment, there are no users with the same name. **Context tokens** will be referred to also by the user’s name, e.g., Alice’s token is the token issued to Alice, containing their information, such as email and scopes available. One possible scenario would be: *A GET request is made to /api/users with Bob’s token as context*: the server will receive a GET request to /api/users (with a bearer token containing Bob’s email, username, and other claims, as well as scopes available), process the request and send a response.

Name	Email	Username	Scopes
Alice	alice@test.com	alice	users:create, users:read, users:delete, users:update
Bob	bob@test.com	bob	users:create, users:read, users:delete, users:update
Charles	charles@test.com	charles	users:create, users:read, users:delete, users:update
Admin	admin@test.com	admin	admin

Table 2.1: Users’ initial information, stored in the database

2.3.1 Mutator Requests

In web communication, the concept of “mutators” plays an integral role. To understand mutators, one must consider web requests as interactions with digital resources. Each resource, like a user profile or a product description, can be accessed, altered, or deleted through specific requests. Here, mutators stand out as unique requests designed to modify the response of another request. In simpler terms, they change the underlying data or resource linked to a response.

Consider the scenario where a PATCH request is sent to /api/users/:email. This request can be seen as a mutator for a GET request aimed at the same path. Why? Because the PATCH request might modify user details based on the provided email parameter, hence affecting the subsequent responses of GET requests.

The HTTP protocol, as prescribed by RFC7231 [11], encourages utilizing the same path for various methods—like GET, PUT, DELETE, and PATCH. This structure is logical: each method instructs the server on the kind of operation to perform on the designated resource. Yet, the flexibility of the web means that developers are not always strictly bound to these guidelines. Consequently, they might implement custom endpoints or unconventional methods. Such deviations can introduce complexities for users, especially when these methods and operations are not well-documented.

We have coined the term *mutator* with the specific aim to help differentiate and identify operations that alter the underlying resource of another endpoint. It is important to clarify that this concept isn’t just confined to GET requests. For instance, while a PUT request might retrieve a resource, this very resource can later be altered or “mutated” by a subsequent DELETE request on the same path. However, for the scope of our study, our primary focus lies on identifying mutators that affect the responses of GET requests.

2.4 Security Properties

When reflecting upon the vast terrain of API security, it is essential first to recognize the foundational layers that set the groundwork for a more nuanced exploration. Their ubiquity in modern software development mandates stringent security measures [38]. The two broad security property domains that emerge at the forefront of this discussion are:

1. Access Control Policies
2. Information Flow Security Properties

Access control policies provide a system's blueprint for determining who gets to access what and in which capacity [35]. They are often the first line of defense in preventing unauthorized actions or data breaches. These policies are often sufficient for single requests where the focus is predominantly on granting or denying access based on credentials, roles, or predefined permissions. On the other hand, information flow security properties, grounded in the work of Denning [9], provide a more comprehensive vantage, often involving the analysis of sequences of multiple requests to ascertain the security robustness of a system. They ensure that data flow within a system adheres to the established security constraints and that sensitive information is not leaked, intentionally or inadvertently, through multiple interactions with the API.

Having established this general context, it is now apt to proceed into the more granular aspects of API security. The “low-level” properties are extensions and manifestations of the broader principles outlined above. We will uncover how the intricacies of access control policies manifest in the nuances of scopes and delve into the realms of information flow through the study of mutator behaviors, determinism, and the principle of non-interference.

As we venture deeper into the complex realm of web security, certain properties emerge as focal points, critical for ensuring the robustness and integrity of Web APIs. In this section, we have picked specific security properties to cast a spotlight upon based on their significance and impact on the web ecosystem.

1. **Broken Object & Function Level Authorization** — This twin challenge pertains to vulnerabilities in the way applications authorize objects and functions. Any lapse here could grant malevolent entities access to resources or functions beyond their intended boundaries. In a world where permissions demarcate responsibilities, ensuring stringent authorization mechanisms is paramount.
2. **Immutability in Subsequent GET Requests** — At its heart, this principle emphasizes the consistency of data fetched. When successive GET requests are made, the expectation is that the resource retrieved remains consistent, barring any external interventions. Understanding the circumstances and implications of this immutability is essential for ensuring data reliability.

3. **Non-Interference Between an Out-of-Context Mutator Request** — When making requests to an API, it is anticipated that each request functions within its own confines without unintended spill-overs. This property is about ensuring that a mutator request, even if it is out of context, does not meddle with or modify unrelated resources. It is a tenet of ensuring that actions within a system are compartmentalized and controlled.
4. **Mutability Between a Mutator Request** — In contrast to our exploration of immutability, this property throws light on the deliberate and necessary changes that a mutator request brings about. By studying this property, we seek to understand the range, boundaries, and implications of changes induced by mutator requests.

Through the lens of these properties, this section aspires to weave a comprehensive narrative on the security dynamics of Web APIs. We aim to unpack the nuances, highlight potential vulnerabilities, and ultimately guide toward a more secure and predictable web interface landscape.

2.4.1 Broken Object & Function Level Authorization

Broken Object Level Authorization and Broken Function Level Authorization are properties that, though distinct in their subjects, pivot around a central concern: unauthorized access to entities or functions through specific identifiers. A deeper dive into these vulnerabilities can be found in Section 1.2. Their significance is underscored by their consistent appearance atop the charts, as evinced by both the 2019 and 2023 editions of the OWASP API Security Project [26] [27].

To elucidate this with a hands-on instance, consider our prototype. Envision a scenario wherein Alice, using her unique token, sends a GET request targeting `/api/users/bob@test.com`. Assuming Alice does not possess the `admin` privilege, the ideal server response should be a `403 Forbidden` status code. This would clearly signal that Alice is overstepping her boundaries trying to access Bob's data, which indeed exists but remains outside her purview. A `200 OK` status code, coupled with a display of Bob's data, would starkly represent a breach since, within this application's architecture, non-admin users should not have the latitude to access the data of other users. Transposing this principle to functions, an illustrative example might be the "add to cart" function: Bob's cart should not be tampered with using Alice's token. Picture a scenario in an e-commerce setting where someone else, using their own credentials, sneaks an item into your cart—such an act would undeniably be flagged as a violation.

However, it is paramount to underscore the importance of context in these discussions. While the aforementioned properties resonate strongly within domains like user management platforms or e-commerce sites, their implications can be more fluid in other spheres. Take, for instance, a collaborative canvas application: Here, the canvas, albeit public, demands authentication to ensure traceable edits. Within this framework, if Alice crafts a canvas, Bob making edits to it is not a breach but rather a feature aligning perfectly with the system's stipulated specifications.

2.4.2 Immutability in Subsequent GET Requests

HTTP semantics, as dictated by RFC 7231 [11], delineate that two consecutive GET requests targeting the same resource ideally should yield identical results. Drawing from our previous discussion on REST in Section 2.1, it is implicit that a RESTful API should epitomize statelessness. However, the real-world intricacies sometimes prevent the strict adherence to this principle. Although RFC 7231 provides guidelines to manage such deviations, in certain situations, these aren't just unattainable but could also be deemed inapplicable. Some illustrative exceptions include:

1. GET requests incorporating a counter reflecting the frequency of that specific request.
2. GET requests that incorporate a timestamp indicating the time taken to process the request.
3. GET requests harnessing rate-limit mechanisms, designed to thwart *DoS* or *DDoS* attacks. Here, after a specified threshold, the server might return a 429 Too Many Requests response to prevent resource exhaustion¹.

In the ambit of this research, our primary focus is the ideal or “golden” standard: deterministic responses. For instance, consecutive GET requests to `/api/users/alice@test.com` should ideally yield uniform results, provided no other intervening requests have transpired. The crux of our concern is the potential security breach when an unintended side effect arises during a GET request's processing. To accentuate, consider an extreme hypothetical: a system inadvertently altering credit card details while processing a GET request to view said information. Such an anomaly would undeniably constitute a grave security breach.

2.4.3 Non-Interference Between an Out-of-Context Mutator Request

One of the nuanced security properties warranting attention is the principle of non-interference (information flow security domain), especially in the context of mutator requests that operate in distinct spheres of influence. Goguen and Meseguer first introduced non-interference as a formal definition in 1982, describing it as a mechanism to prevent high-level (more sensitive) security processes from influencing lower-level (less sensitive) processes, and thereby blocking unwanted information flows [15]. By this, we mean that actions performed in one context should not inadvertently affect data or behaviors in another unrelated context.

Consider this illustrative scenario: Let us assume we make two GET requests targeting `/api/users/alice@test.com` using Alice's authentication token. Now, if between these two requests, a PATCH request directed at `/api/users/bob@test.com` is executed using Bob's token, the outcome of our initial GET request should remain unchanged. Why? Simply because

¹A Denial of Service (DoS) attack seeks to overwhelm a target by flooding it with superfluous requests, thus denying legitimate users access. A Distributed Denial of Service (DDoS) attack amplifies this by leveraging multiple compromised systems as sources of the attack traffic [20].

Alice's and Bob's data domains are separate, and a mutator request in Bob's domain should have no bearing on Alice's data.

However, if we observe discrepancies in the responses of the two `GET` requests, it raises alarming concerns. Such an inconsistency suggests that the system is experiencing unintended side effects. Even though the `PATCH` request by Bob is a mutator in its domain, it should remain confined to that domain without spilling its influence over Alice's data. Any deviation from this expected behavior indicates potential system vulnerabilities, compromised integrity, or flawed design. This is akin to the concerns we highlighted in the preceding section but is triggered by different underlying issues. Ensuring non-interference is vital to maintain data integrity, isolate user actions, and fortify the system against unforeseen security breaches. Finally, it is important to clarify that this property assumes the determinism of `GET` requests, i.e., two consecutive `GET` requests for the same parameters should respond with the same value.

2.4.4 Mutability Between a Mutator Request

At its core, the principle of observing changes in a resource after a mutator request is rooted in system correctness. When a resource undergoes a mutation, a subsequent `GET` request ideally should reflect this change. Ensuring this behavior guarantees that a system accurately represents the modifications made to its resources.

In the prototype presented, it is posited that mutator requests invariably alter the resource unless the input values match the existing ones, in which case the resource remains unchanged. Behind this perspective is the conviction that changes to a resource should be decisively expressed. When a user communicates an intention to modify a resource they can access, the system should either depict the requested changes or return a client error if the new values do not conform to requirements.

Yet, this perspective becomes more intricate, especially within environments where resources are disseminated among multiple users in distributed systems. There, a change enacted by one user might not immediately manifest for another, particularly when permissions vary. This variance introduces an intriguing dimension of context-based visibility. It is not solely about the occurrence of a change but also the audience for that change and its timing.

For instance, when dealing with sensitive data like credit card details, certain attributes might be deliberately excluded from `GET` responses for security purposes. If a `PATCH` request amends such an attribute, subsequent `GET` requests could remain unchanged, given the consistent omission of that particular attribute.

Furthermore, the visibility and interaction with these resources are deeply connected to the session or context in which they're accessed. The relationship between context-based visibility and web session security can furnish profound insights into the intricate dynamics of resource manipulation and portrayal, especially in systems characterized by diverse user roles and permissions.

In conclusion, while anticipating specific outcomes after a mutator request stands as a hallmark of system correctness, the subtleties of context, permissions, and system architecture can influence this behavior in intricate ways, offering a compelling domain for further exploration.

2.5 OpenAPI & Security Validation

The OAS [25] provides a robust framework for documenting the security specifics of APIs. Via its **Security Scheme** property, OAS can encapsulate diverse authentication and authorization schemes, be it Bearer tokens, API tokens, or OAuth 2.0. The snippet in Listing 2.1 illustrates this by showcasing the security scheme of the prototype API: a security scheme is presented for OAuth 2.0 (`type` property), as well as the `scopes` available and where to get the token from (`tokenUrl` property); in this case, the flow is `password` indicating the end-users should provide a password in order to obtain a token.

```
1 ...
2 components:
3   ...
4   securitySchemes:
5     OAuth2PasswordBearer:
6       type: oauth2
7       flows:
8         password:
9           scopes:
10            admin: Admin privileges
11            read:users: Read users data
12            create:users: Create users data
13            update:users: Update users data
14            delete:users: Delete users data
15            tokenUrl: /api/v1/auth/token
```

Listing 2.1: Prototype Security Scheme

Endowed with the capability to articulate the required level of authorization for individual endpoints, the OpenAPI schema proves indispensable for API development. For instance, the `read:users` scope can be made requisite for accessing the `GET` method at `/api/users/{user_email}`, as shown in Listing 2.2.

```
1 paths:
2   /api/v1/users/{user_email}:
3     get:
4       parameters: ...
5       responses: ...
6       security:
```

```

7      - OAuth2PasswordBearer:
8        - read:users

```

Listing 2.2: GET User Security Scopes

This structured documentation not only aids developers but also simplifies the testing process. However, while the OAS excels at articulating authentication methods and scopes, its capacity to capture complex application contexts remains limited. Simple scenarios where a user’s scope defines their access (like `read:products`), are straightforward. Yet, situations demanding intricate context, such as a user accessing solely their data on a user management platform (elaborated upon in Subsection 2.4.1), present challenges. To capture these nuanced access controls without resorting to a cumbersome scope explosion is a significant shortcoming of the OAS.

Turning our attention to the security properties in Section 2.4, the OpenAPI Standard’s absence of a “mutator” concept is noticeable. Although OAS offers the `link` attribute to suggest potential data-flow between responses and requests, its semantics are not clear-cut. Listing 2.3 displays a `link` example, where the output of one endpoint serves as input for another; in lines 4 and 6, it is indicated which method this snippet describes. The responses array contains a success value (201 status code) with a link to another operation, `user_get_by_email`, accepting the `email` from the POST response as a path parameter (`user_email`). Yet, this relationship does not inherently translate to a semantic bond. Producer-consumer relations between endpoints, as first introduced by RESTler [4], do not equate to one acting as a mutator for another.

```

1  ...
2  paths:
3    ...
4    /api/users:
5      ...
6      post:
7        ...
8        summary: Users Create
9        operationId: users_create_api_v1_users_post
10       responses:
11         ...
12         '201':
13           description: User created
14           links:
15             UserGetByEmail:
16               operationId: user_get_by_email
17               parameters:
18                 user_email: '$response.body#/email'
19  ...

```

Listing 2.3: OpenAPI Links Example

The elusive semantics between API endpoints accentuate the challenges faced during security testing, especially in a black-box setting. Such settings are deprived of insights on how distinct endpoints interplay, which often encapsulates the security crux. Studies [3] have demonstrated that capturing interactions and sequences between endpoints can unearth bugs overlooked by unit tests. While unit testing tends to zero in on isolated code fragments, considering the broader interaction dynamics can expose vulnerabilities, especially when trying to account for all potential interaction scenarios.

In sum, although OAS offers a structured documentation approach for API security, its ability to capture intricate context and semantic endpoint relationships remains wanting. These deficiencies underscore the significance of comprehensive testing strategies that transcend unit testing, emphasizing interactions and context nuances.

Chapter 3

Testing of Web APIs

Writing unit tests for Web APIs (or any other type of system) has become increasingly challenging as the complexity of applications grows. Developers, amidst this complexity, may inadvertently overlook or omit important test cases, introducing a reliance on human diligence for ensuring software’s correctness and, by extension, its security [5]. Recognizing the human-error factor, researchers have delved into devising ways to automatically generate tests for applications, seeking to transfer the onus of correctness from human dependability to the efficacy of the automated tool used [2].

Automated testing tools for Web APIs have significantly evolved over recent years, but there remains ample scope, especially in the realm of security. These tools typically encompass three pivotal components: a schema learner, a generative core, and an oracle. The schema learner usually deduces sequences of requests, permissible input and output data, and potential responses [14]. Armed with this data, the generative core employs techniques, often grounded in fuzzing, to craft input data for the system [40]. The oracle, in essence, evaluates the outcome, setting the stage for the divergence in approaches adopted by contemporary tools – some aim to merely confirm schema conformance and correctness, while others aspire to verify semantic correctness [45, 32].

While the ambition for a comprehensive automated testing tool, particularly with a security-centric lens, might still be on the horizon, certain pioneers in the field have already made commendable strides. This chapter will delve into the current state of the art for automated testing tools, shedding light on pivotal concepts such as *fuzzing* and *online testing*.

3.1 Fuzz Testing & Property-based Testing

The world of automated software testing is vast, offering various methodologies to ensure the robustness and security of applications. Among these, fuzz testing and property-based testing stand out as pivotal approaches, each with its nuances, strengths, and applications.

Fuzz testing, or simply fuzzing, is eloquently captured by MacIver [21]:

Fuzzing is feeding a piece of code (function, program, etc.) data from a large corpus, possibly dynamically generated, possibly dependent on the results of execution on previous data, in order to see whether it fails.

In essence, fuzz testing challenges software components' resilience by presenting them with a myriad of unpredictable data inputs. This unpredictability is its hallmark. By inundating software with diverse data, often outside expected boundaries, fuzzing endeavors to detect vulnerabilities, bugs, or anomalies in software behavior. The scope of fuzzing, however, isn't rigidly set. As per MacIver [21], while some fuzzers focus on binary data, others delve into more structured datasets. Similarly, criteria for identifying a failure varies—ranging from system crashes to functions merely returning false values.

Venturing beyond the boundaries of fuzzing, we find property-based testing, a method with a distinct edge. MacIver [21] describes it as:

Property-based testing is the construction of tests such that, when these tests are fuzzed, failures in the test reveal problems with the system under test that could not have been revealed by direct fuzzing of that system.

This characterization underscores the essence of property-based testing: the meticulous crafting of tests to elicit specific failures that might elude traditional fuzzing.

A point of contention, however, emerges from MacIver's phrase "[...] that could not have been revealed by direct fuzzing [...]". This statement has sparked debates, with some arguing that fuzzing might very well be a subset of property-based testing. MacIver [21] himself acknowledges this overlap. Yet, a discernible difference can be observed: property-based testing operates with a guiding beacon—a target property. Employing techniques akin to fuzzing, it distinguishes itself by its intentional focus. The tests, along with the data generation, are steered purposefully towards a defined property, ensuring its comprehensive examination and validation.

In conclusion, both fuzzing and property-based testing have their unique contributions in the realm of software testing. While fuzzing scouts for vulnerabilities through an avalanche of random data inputs, property-based testing adds a layer of precision, tailoring tests to specific properties. A nuanced understanding of both methodologies empowers developers and testers to deploy a multifaceted testing strategy, bolstering software resilience against a spectrum of potential threats.

3.2 RESTler

RESTler [4] is the first stateful REST API fuzzer. According to the authors, RESTler analyses the API specification of a cloud service and generates sequences of requests that automatically test the service through its API.

RESTler generates test sequences by inferring producer-consumer dependencies¹ among request types declared in the specification and by analyzing dynamic feedback from responses observed during prior executions. Inferring sequences of requests based on consumer-producer relationships can be seen as something trivial and static since those sequences can be generated from a well-structured OpenAPI schema, while analyzing dynamic feedback can be substantially more challenging; one example that can be achieved is, for instance, learning that a request C, after a request sequence $A \rightarrow B$ is refused by the service, and therefore avoiding this combination in the future [4].

RESTler [4] detects bugs found in request sequences by analyzing service malfunctions (i.e., 5xx status codes) when processing the requests with different inputs generated by fuzzing techniques. The authors used RESTler [4] to test GitLab, an open-source Git service, as well as several undisclosed Azure and Office365 cloud services; RESTler detected 28 bugs in GitLab and several bugs in each of the Azure and Office365 cloud services tested. At the time of the writing of the paper, the bugs found were confirmed and fixed by the services' owners.

RESTler [4] processes the API specifications and constructs a test-generation grammar, encoded in executable Python code; this consists of generating a request and code to process the expected response for said request; inputs for the requests can either be static (user-provided) or *fuzzable*, with an associated type, the *fuzzable* inputs are replaced by one value taken from a (small) dictionary of values for said type. RESTler [4] then automatically infers relationships between requests from the responses it analyses, thus creating more test scenarios and broadening the search space.

To validate RESTler [4] the authors formulated three research questions:

- RQ1** — Are both inferring dependencies among request types and analyzing dynamic feedback necessary for effective automated REST API fuzzing?
- RQ2** — Are tests generated by RESTler exercising deeper service-side logic as sequence length increases?
- RQ3** — How do the three search strategies implemented in RESTler compare across various APIs?

The authors [4] answered **Q1** with a blog post experimental setup with three different test-generation algorithms: ignoring dependencies between requests, ignoring server-side dynamic feedback, and combining the found dependencies with the analysis of server-side dynamic feedback. They concluded that combining the two techniques is essential to finding relevant test sequences. The authors [4] proceeded to answer **Q2** by testing GitLab's services; they concluded that longer sequence lengths consistently lead to increased server-side coverage, and with this, they concluded that, in fact, as the sequence length increases the tests exercise deeper server-side

¹A producer, A, in a producer-consumer relationship between A and B, is a service that outputs a resource that can be used as input by B, e.g., in a bookstore API service, two services could be specified: a DELETE service to delete books from the database, and a GET service to retrieve a book from the database, in this case, a relationship is established between the GET service as a producer and the DELETE service as a consumer for the same resource type.

logic. Finally, the authors [4] answered **Q3** by testing the three different search algorithms: *BFS*, *BFS-Fast*, and *RandomWalk*; they concluded that *RandomWalk* found more bugs than the other two algorithms, in a 5-hour timeout, but the code coverage was lower.

More recently, a paper [3], by the same authors as the paper introducing RESTler [4], described how can a stateful REST API fuzzer be extended with active property checkers to detect and capture violations to security rules automatically. The authors introduced four security rules² [3] to test RESTful APIs against:

1. **Use-after-free** rule [3] — A resource that has been deleted must no longer be accessible. As an illustration, if a `DELETE` request is sent to the URI `/users/user-id1` with the intention of removing the account associated with `user-id1`, subsequent attempts to utilize `user-id1` must be unsuccessful and will result in a `404 Not Found` HTTP status code in their response [3]. This concept, known as the use-after-free rule, is commonly associated with memory management problems in programs written in C and C++. The principle remains the same in that the program is attempting to access a resource that has already been released or deleted.
2. **Resource-leak** rule [3] — A resource that was not created successfully must not be accessible and must not “leak” any side-effect in the backend service state. To put it succinctly, if a `PUT` or `POST` request to create a new resource is not executed successfully (for any cause), all subsequent attempts to interact with that resource must also result in a `4xx` response [3]; additionally, there must be no discernible changes to the backend service state that are related to the creation of that resource; e.g., the failed-to-be-created resource must not be included in the user’s resource count for service quotas, and its name must be available for reuse by the user [3]. For example, if an incorrect `PUT` request is sent to create the URI `/users/user-id1`, a `4xx` response must be received. All subsequent requests to access this URI, whether for reading, updating, or deleting, must also fail [3].
3. **Resource-hierarchy** rule [3] — A child resource of a parent resource must not be accessible from another parent resource. In other words, if a child resource is created from a parent resource and is identified as such in service resource paths like `<parentType>/parent/<childType>/child/`, the child resource must not be accessible when the parent resource is substituted with another parent resource [3]. For instance, after sending `POST` requests to the following URIs:

`/users/user-id1`, `/users/user-id2`, and
`/users/user-id1/reports/report-id1`

to create users `user-id1`, `user-id2`, and to add report `report-id1` to user `user-id1`, subsequent requests to the URI `/users/user-id2/reports/report-id1` must fail as

²These four rules the authors [3] proposed are aligned with some of the HTTP semantics specifications [11].

per the resource-hierarchy rule, meaning `report report-id1` belongs to user `user-id1` but not to user `user-id2` [3].

4. **User-namespace rule** [3] — A resource created in a user namespace must not be accessible from another user namespace. For instance, after sending a POST request to create the URI `/users/user-id1` using the token `token-of-user-id1`, resource `user-id1` must not be accessible using another token `token-of-user-id2` belonging to a different user [3]. This type of violation, referred to as a user namespace violation, occurs when a resource created within the namespace of one user becomes accessible within the namespace of another user; if such a violation were to occur, an attacker could exploit it to execute REST API requests using an unauthorized authentication token and carry out unauthorized operations on resources belonging to another (victim) user [3].

Both the properties defined in this thesis (see Section 2.4) and those posited by RESTler offer distinct perspectives in ensuring the robustness of APIs against security threats.

Our properties underscore the consistent, predictable, and secure behavior of APIs across diverse operational settings. In contrast, RESTler’s properties are primarily an interpretation of the HTTP semantics pertaining to resources—like ensuring existence post-creation and ensuring non-existence post-deletion. These are much in line with your mutators’ concepts. Particularly noteworthy are RESTler’s final two properties concerning resource hierarchy and the user namespace. The critical difference lies in their approach to access control. RESTler’s notion of access control is more ad-hoc and deeply embedded within their properties. To illustrate, there isn’t anything in the HTTP specification that suggests an object hierarchy in the manner they have presumed. Furthermore, the association between userIDs and user tokens in RESTler is manually defined rather than being driven by an overarching standard or guideline.

Our extension of OAS, incorporating scopes and related features, aims to express these relations in a more generalized manner, not confined by the restrictions of RESTler’s assumptions. Moreover, while RESTler predominantly focuses on access control, we also aim to capture properties related to information flow. Together, both perspectives emphasize the necessity to deter unauthorized access while ensuring that even authorized interactions maintain their security and predictability. By synergizing these viewpoints, our goal is to craft a thorough and robust API security testing approach.

For each security rule, the authors [3] implemented an active checker; an active checker monitors the state space exploration performed by the main driver of stateful REST API fuzzing and suggests new tests to assert that specific rules are not violated. Thus, an active checker augments the search space by executing new tests that violate specific rules. In contrast, a passive checker monitors the search performed by the main driver without executing new tests. Active checkers [3] were designed following two principles:

1. Checkers are independent from the main driver of stateful REST API fuzzing and do not affect its state space exploration [3]. Enforced by running all the checkers whenever the main driver has finished executing a new test case.

2. Checkers are independent from each other and generate tests by analyzing the requests executed by the main driver, excluding those executed by other checkers [3]. Enforced by prioritizing the order of applying checkers based on their semantics so that they operate on different test cases and do not interfere with each other.

Finally, the active checkers [3] extend the main driver of baseline stateful REST API *fuzzing* in two ways: they extend the state space by executing additional tests, and they check for responses other than 5xx and can flag unexpected 2xx responses as rule-violation bugs. Thus, active checkers [3] clearly increase the bug-finding capabilities of the main driver: they can find bugs that the main driver alone would not find.

The authors [3] concluded, through experimentation, that their fuzzing tool was able to find about a handful of new bugs in each of the services provided by Azure and Office-365 cloud services, using the base fuzzer extended with active checkers. About one-third of the bugs found were rule violations detected by their security checkers; the bugs were then reported to the service owners, and all had been fixed. In the paper's [3] conclusion, the authors asked themselves, "How general are these results?" and to know that, they answered that not only testing on more services was necessary, but also checking more properties to detect different kinds of bugs and security vulnerabilities. Finally, the authors [3] stated:

Given the recent explosion of REST APIs for cloud and web services, there is **surprisingly little guidance about REST API usage from a security point of view**.

Our paper makes a step in that direction by contributing four rules whose violations are security-relevant and which are non-trivial to check and satisfy.

This goes directly to the motivation of this thesis and corroborates the fact that not only there's little guidance about creating and using a REST API from a security standpoint, but there are also even fewer tools and methods to test APIs for security vulnerabilities.

3.3 Schemathesis

Schemathesis [17] derives structure- and semantics-aware fuzzers from Web API schemas in the OpenAPI or GraphQL formats, using a property-based testing tool, Hypothesis [22] (more about Hypothesis on Section 4.3). From the specification [17], the derived fuzzers can be incorporated into unit-test suites or run directly, with or without end-user customization of data generation and semantic checks.

The authors [17] stated they constructed the most comprehensive evaluation of Web API fuzzers to date, running eight fuzzers against sixteen real-world open-source web systems. They also compared Schemathesis [17] performance versus other tools. From the evaluation, the authors [17] concluded that Schemathesis was the only tool to handle more than two-thirds of their target services without a fatal internal error.

The authors [17] also state the following about schemas and API errors:

In practice, OpenAPI or GraphQL schemas are often unsound, permitting inputs or actions not handled by the service. This may be due to errors in the schema or the service implementation because working with sound schemas is uneconomical or because of application-level constraints (database constraints, order of event timestamps, relations between endpoints, etc.), which cannot be expressed in the schema.

Schemathesis [17] also tries to enforce some of the universal and simple constraints defined in RFC 7231 [11]:

1. 200 OK responses must have a non-empty body
2. 204 No Content and 205 Reset Content responses must have an empty body
3. 302 Found responses to a POST request must allow the subsequent response to use either POST or GET methods
4. 405 Method Not Allowed responses must have an Allow header listing supported methods
5. 500 Internal Server Error responses are always errors
6. GET fails after successful DELETE (equivalent to the use-after-free rule described in Section 3.2)
7. GET fails after unsuccessful POST (equivalent to the resource-leak rule described in Section 3.2)

Moreover, internally, Schemathesis [17] distinguishes single-request tests from those that make a sequence of requests to multiple endpoints using the data from past responses. One of Schemathesis' strengths is the ease of customization via its command-line interface (CLI) or from Python. The four main ways to customize tests for a specific API are hooks, checks³, serializers, and format strategies.

For the evaluation, the authors [17] experimented with their tool and previous Web API fuzzers, measuring defect detection, run-time, and consistency of reporting. The experiments were restricted to containerized open-source services, ensuring they were representative, reproducible, and would not attack live systems (see Table 3.1 for reference on the system tested by Schemathesis).

³Checks are custom test oracles, which allow verification of user-defined properties of responses received from the application under test. Because checks are decoupled from data generation, they can be run for both known-valid and known-invalid test cases [17].

Service	Language	Framework	Endpoints	Schema type	Schema source
aalises/age-of-empires-II-api	Python	Flask 1.1.2	8	OpenAPI 3.0.0	Static
creativecommons/cccatalog-api	Python	Django 2.2.13	8	Swagger 2.0	Dynamic, dgf-yasg 1.17.1
ryo-ma/covid19-japan-web-api	Python	Flask 1.1.2	4	Swagger 2.0	Dynamic, flasgger 0.9.4
disease-sh/api	JavaScript	Express 4.17.1	34	Swagger 2.0	Static
postmanlabs/httpbin	Python	Flask 1.0.2	73	Swagger 2.0	Dynamic, flasgger 0.9.0
jupyter-server/jupyter_server	Python	Tornado 6.1.0	29	Swagger 2.0	Static
jupyterhub/jupyterhub	Python	Tornado 6.1.0	35	Swagger 2.0	Static
mailhog/MailHog	Go	Net/HTTP	2	Swagger 2.0	Static
fecgov/openFEC	Python	Flask 1.1.1	85	Swagger 2.0	Dynamic, flask-apispec 0.7.0
ajnisbet/opentopodata	Python	Flask 1.1.2	2	OpenAPI 3.0.2	Static
rtyley/otto	Rust	Tide 0.14.0	2	OpenAPI 3.0.3	Static
fossasia/pslab-webapp	Python	Flask 1.1.2	3	Swagger 2.0	Dynamic, flasgger 0.9.5
pulp/pulpcore	Python	Django 2.2.17	67	OpenAPI 3.0.3	Dynamic, dgf-spectacular 0.11.0
darklynx/request-baskets	Go	Net/HTTP	20	Swagger 2.0	Static
microsoft/restler-fuzzer	Python	Flask 1.1.2	6	Swagger 2.0	Static
IBM/worklog	Python	Flask 1.0.2	9	Swagger 2.0	Dynamic, flasgger 0.9.1

Table 3.1: Systems tested as part of the evaluation for Schemathesis

For ease of analysis, the authors [17] parsed the 250GB of raw logs into a JSON summary and further reduced this dataset to report the duration, number of events, and per-run reports of each unique defect. Manual defect triage and deduplication are impractical for such a large and extensible evaluation; instead, where possible, they monitored the fuzzing process using Sentry⁴ for error tracking and performance monitoring. This gave them a cross-language notion of unique defects, i.e., internal server errors deduplicated by code location—regardless of triggering endpoint or what the fuzzer was attempting to check at the time. The authors added semantic errors to their defect count by counting each kind of bug report parsed from saved logs only once per endpoint, regardless of variations or how many times it was observed.

In summary, the authors [17] stated that Schemathesis is the only fuzzer in their comprehensive evaluation to handle every real-world schema and web service and consistently report more defects than the previous state-of-the-art.

3.4 Metamorphic Testing

Metamorphic testing [6] is a software testing technique that aims to alleviate the oracle problem (determining the expected behavior of a software system under different input conditions; more details about this concept below) by using metamorphic relations (MRs) to automatically derive test inputs and evaluate the outcomes of test cases. In metamorphic testing, MRs are used to transform an initial set of test inputs into follow-up test inputs. If the system outputs for the initial and follow-up test inputs violate the corresponding MR, it is concluded that the system is faulty. Metamorphic testing has been applied to various application domains, including computer

⁴Sentry is an error tracking and performance monitoring platform

graphics, web services, and embedded systems. It has been shown to be effective in detecting faults and vulnerabilities in software systems.

In the paper [6], the authors propose a metamorphic testing approach called Metamorphic Security Testing for Web-interactions (MST-wi) specifically designed for security testing of web systems. MST-wi integrates test input generation strategies inspired by mutational fuzzing and addresses the oracle problem in security testing. To implement metamorphic testing on web systems, MST-wi [6] enables engineers to specify metamorphic relations (MRs) that capture various security properties of web systems using a domain-specific language called SMRL (Security Metamorphic Relation Language). The MRs are then transformed into executable Java code used to perform security testing on web systems automatically. MST-wi also includes a catalog of 76 system-agnostic MRs that can be used to automate security testing in web systems and detect vulnerabilities. To facilitate the specification of MRs, MST-wi [6] provides an Eclipse editor that helps test engineers write and edit MRs in SMRL. MST-wi also automatically collects input data from web systems and uses the collected data and MRs to test the systems and detect vulnerabilities automatically. The paper's authors also provide guidelines to help test engineers improve the testability of the web systems under test with respect to MST-wi.

Oracle problem [6] — refers to the challenge of determining the expected behavior of a software system under different input conditions. This is a common challenge in software testing since it is often difficult to predict how a system will behave under different inputs, particularly when the system is complex or has not been thoroughly tested. The authors [6] discuss how metamorphic testing can be used to address the oracle problem. Specifically, they argue that metamorphic testing can be particularly effective for identifying unexpected or anomalous behavior that may indicate the presence of security vulnerabilities. To solve the oracle problem in the context of metamorphic testing, the authors [6] recommend the following approach:

1. **Identify and define metamorphic relations** [6] — In order to use metamorphic testing effectively, it is important to identify and define appropriate metamorphic relations for the system under test. These relationships should be chosen based on understanding the system's requirements and constraints and the types of transformations likely to reveal vulnerabilities.
2. **Generate test cases** [6] — Once the metamorphic relations have been identified, they can be used to generate test cases that evaluate the system's behavior under different input conditions. This may involve applying transformations to the input data, the execution environment, or the system's internal structure.
3. **Execute the test cases** [6] — After the test cases have been generated, they can be executed to evaluate the behavior of the system under the different input conditions. Any deviations from expected behavior may indicate the presence of security vulnerabilities.

4. **Analyze the results** [6] — Once the test cases have been executed, it is important to analyze the results to determine whether the system behaved consistently and as expected under the different input conditions. Any deviations from expected behavior should be carefully examined to identify the cause and determine whether they indicate the presence of security vulnerabilities.

Overall, the authors [6] argue that metamorphic testing can be an effective approach for solving the oracle problem by allowing for the exploration of a wide range of input conditions and the identification of unexpected or anomalous behavior that may indicate the presence of security vulnerabilities.

A research question [6] was formulated to determine which types of vulnerabilities could be discovered using Metamorphic Security Testing for Web-interactions (MST-wi). To answer this question, the authors [6] systematically analyzed weaknesses in the Common Weakness Enumeration (CWE) database. They implemented metamorphic relations (MRs) to address each weakness, using SMRL or reusing existing MRs from the MST-wi catalog if possible. The resulting MRs were used to automatically test the weaknesses to determine whether they could be discovered through MST-wi.

The authors [6] reported that 45% of all 223 weaknesses and 48% of 204 generic weaknesses in the CWE database could be addressed through MST-wi. They also provided a breakdown of the percentage of weaknesses that could be addressed in each category of the CWE views for common security architectural tactics, CWE Top 25, and OWASP Top 10. They also analyzed the distribution of the reasons why MST-wi could not address certain weaknesses. The authors [6] provided examples of weaknesses in their analysis through a table that includes details such as the name of the weakness, the security design principle affected, and whether the weakness belongs to the CWE Top 25 or OWASP Top 10 lists. Overall, the results of this analysis suggest that MST-wi is a valuable approach for discovering a significant number of vulnerabilities in web systems.

In the examination of metamorphic testing, the study has touched upon several key CWEs pertinent to system vulnerabilities. Some of these weaknesses show significant alignment with the properties we have defined for our research context. For instance, our emphasis on *Broken Object & Function Level Authorization* resonates with *CWE-285: Improper Authorization* and *CWE-862: Missing Authorization*, pinpointing the necessity for robust authorization protocols. However, not all CWEs from the metamorphic testing discussion might align seamlessly with our defined properties, underscoring the importance of context specificity in vulnerability assessments. From our analysis, metamorphic testing excels at detecting vulnerabilities related to the adjacent technology, such as cryptographic issues, injection vulnerabilities, and Server-Side Request Forgery. However, it is not so efficient, or in the best case scenario, as efficient as the state-of-the-art techniques they explored, leaving some domains left to cover, such as human-induced problems related to the system's semantics and overall behavior, access control policies, and the intricates related to information flow security: non-interference between non-overlapping contexts. Through this lens, while metamorphic testing offers a broad framework to unearth system weaknesses, its efficacy in

our particular context demands a nuanced appraisal, especially when juxtaposed with the specific CWEs we are evaluating.

3.5 RESTest

RESTest [24] is a robust open-source framework designed for black-box testing of RESTful Web APIs. Leveraging the OpenAPI Specification as its foundational input, the framework boasts a rich array of test case generation methodologies. This encompasses contemporary techniques such as fuzzing, adaptive random testing, and constraint-based testing.

Delving deeper into its mechanics, RESTest employs bespoke test data generators, which ensure the creation of pertinent and realistic test data. This adaptability allows RESTest to seamlessly meld with various frameworks and libraries, thereby furnishing compatibility for both offline and online testing scenarios. The integration with the Allure framework [1] further amplifies its utility, furnishing users with insightful graphical test reports.

The open-source nature of RESTest [24] stands as a testament to its extensibility. Developers and testers can effortlessly introduce novel test generation strategies and data generators, making it a holistic solution for RESTful API testing.

A succinct workflow, as delineated by the authors [24], captures the essence of the framework:

1. **Test Model Creation** — At its core, RESTest employs a model-based testing paradigm. This involves two primary models: the system model, which pertains to the API specification, and the test model, articulated as a configuration YAML⁵ file. This test model, aside from capturing all test-related configurations, also encodes the data for each parameter. This can range from established data dictionaries to custom test data generators like airport codes.
2. **Abstract Test Case Formulation** — By harnessing the power of both system and test models, abstract test cases are formulated. These cases, being platform-agnostic, offer the flexibility to be later metamorphosed into executable cases suitable for diverse testing frameworks and languages.
3. **Executable Test Case Generation** — The abstract test cases undergo a transformation, rendering them into executable test cases compatible with specific testing frameworks, an exemplar being REST Assured [16].

Even in its nascent stage, RESTest [24] has showcased its prowess, having been instrumental in uncovering tangible bugs in commercial APIs. As the authors anticipate augmenting its capabilities with additional data generators and test case strategies, it is pertinent to note a significant limitation: RESTest does not innately focus on security aspects. While its testing techniques can inadvertently unearth some security inconsistencies, it is not tailored explicitly toward security

⁵YAML, an intuitive and cross-language data serialization language, finds its utility in a myriad of programming scenarios. From the simplification of configuration files to data visualization, its design is rooted in catering to the varied data types inherent to dynamic programming languages [46].

testing or vulnerability detection. Nevertheless, the overarching vision for RESTest is clear: to evolve it into an all-encompassing framework adept at online testing and vigilant monitoring of RESTful APIs.

3.6 Online Testing of Web APIs

In 2022, researchers from the *Universidad de Sevilla* published a study [23] on online testing of Web APIs; the study reports on the results of an empirical study on the use of automated test case generation for online testing of RESTful APIs. The study used the RESTest [24] framework to generate and execute over 1 million test cases for 15 days non-stop in 13 industrial APIs. A multi-bot architecture was used to scale the testing process, and the results showed the presence of 390 thousand failures, which were triaged into 254 bugs. Some of the key challenges to adopting online testing of RESTful APIs were identified, including bugs and improvements to the documentation. The study showed the potential of online testing as a must-have feature in the industry but highlighted some challenges to overcome for its full adoption in practice. The authors [23] highlighted the size of popular API repositories, which index over 24 to 30 thousand APIs, emphasizing the significance and pervasiveness of Web APIs; the authors [23] also state that checking the correct functioning of Web APIs is crucial, and the industry is shifting toward online testing, where APIs are continuously tested while in production, however, the generation of test cases still requires manual work and research on automated generation of test cases for RESTful APIs has been limited to controlled environments and lab settings: there is a gap between industrial solutions focused on test case execution and research approaches focused on automated test case generation.

Online testing can be either a passive monitoring activity or an active testing process [23], where the latter consists of generating and executing test cases in the APIs in production. The authors adopted the latter definition of online testing in their research [23]. The authors [23] then described the increasing popularity of online testing platforms for Web APIs in the industry, highlighting different popular platforms such as Datadog [8], RapidAPI Testing [31], and Sauce Labs [36], they also mentioned that some prior research had been done on online testing of RESTful APIs, but with limited test cases and using a single testing technique; finally, the authors presented their study as the first empirical study on the use of automated test case generation techniques for online testing of RESTful APIs in industrial-like settings, which resembles the testing as a service offered by industrial platforms. The study was conducted over 15 days to examine the failure and fault detection capability of different black-box test case generation techniques and to identify key challenges for their adoption in practice.

The authors [23] outlined five challenges that pose obstacles in adopting test case generation techniques for online testing of RESTful APIs. These challenges are:

1. **Automated fault identification** — accurately determining the root cause of the many failures found during testing is challenging and requires more sophisticated approaches.

2. **Effective human interaction** — incorporating human input can improve the accuracy of the classification and avoid duplicated clusters; this could be addressed through active learning algorithms.
3. **Optimal selection of testing strategies** — automatically determining the most appropriate testing techniques based on various factors is an open problem.
4. **Optimal test execution scheduling** — scheduling the test execution to optimize the available resources (quota, time, and economic budget) and testing strategies is currently done manually.
5. **Optimization of computational resources** — minimizing the use of computational resources while testing is both an engineering and research challenge.

The authors [23] also discussed the impact of using RESTest as the main tool in the testing ecosystem on manual work and computational resources. The manual work required to deploy all 75 test bots included writing 26 thousand lines of code (LOCs) for test configuration files and using 50 data dictionaries with 52,084 values. However, 95% of the LOCs were duplicated, highlighting the need for a less verbose data format in RESTest. The computational resources consumed increased over 15 days of online testing, with RAM and disk usage being the main factors. The computational cost per bot showed differences between different APIs; the research suggested that restarting bots regularly could reduce RAM usage and highlighted the overhead of handling large responses.

Now, pivoting to the intersection of online testing and security: If security testing in an offline setup is inherently complex, transitioning to an online environment adds another layer of intricacy. Online testing brings forth unique challenges — how can we ensure that security tests do not inadvertently expose sensitive data in a live setting? What are the risks of unintentionally triggering malicious activities? Furthermore, the dynamic nature of online environments demands rapid responses, something offline testing scenarios do not necessarily mandate. Thus, the challenges in online security testing are not merely amplified versions of their offline counterparts; they introduce entirely new concerns. The intersection of these two domains — online testing and security — demands in-depth exploration, especially in the rapidly evolving landscape of Web APIs.

3.7 Synthesis and Future Endeavors in Web API Testing

This chapter's intricate examination of the current panorama in Web API testing provides a compelling backdrop against which the subsequent stages of this thesis will unfold. As Web APIs burgeon in prominence, laying the infrastructural groundwork for countless digital applications, the imperative for rigorous, holistic testing regimes becomes incontrovertible.

Several salient observations crystallize from our exploration:

1. **Pinnacle of Testing Techniques** — Both RESTler and Schemathesis epitomize the zenith of contemporary testing methodologies. They are emblematic of collective endeavors to

guarantee that APIs, whether stateful or stateless, operate within expected parameters and remain resilient against latent vulnerabilities. Meanwhile, the black-box approach championed by RESTest underscores the nuances of testing from an external perspective, revealing vulnerabilities that could be readily exploited by third-party actors.

2. **The Ascendancy of Online Testing** — Research emanating from the Universidad de Sevilla [23], highlighting the inexorable transition to online testing, accentuates the industry's commitment to ensuring that APIs undergo rigorous pre-deployment evaluations and endure continuous scrutiny during their operational lifespan. While the challenges concomitant with online testing are undeniably formidable, they delineate the contours of the next epoch in Web API testing.
3. **Security: An Unfulfilled Imperative** — Albeit tools like RESTest are commendable in their diagnostic acumen, discerning operational anomalies and defects, there exists a conspicuous lacuna with respect to comprehensive security assessments. The prevailing narrative for most tools tilts towards operational fluency, occasionally sidelining the exigencies of targeted security vigilance.

Against this canvas, the pressing need for an integrative tool that melds the finesse of stateful testing with meticulous security evaluations becomes self-evident. The security properties delineated in Section 2.4 bring to the fore some of the most pressing challenges that pervade the Web API domain. From vulnerabilities pertaining to authorization to the intricate dance of safeguarding data immutability while permitting controlled mutability, the multifaceted challenges intrinsic to API design and deployment are laid bare.

Enter Schemathesis. Its track record in property-based stateful testing establishes it as a prime candidate for the ambitious task at hand. Beyond its existing prowess, Schemathesis boasts an inherently flexible and potent stateful engine capable of discerning complex interrelations and dependencies. This positions it uniquely to address the nuanced and multi-layered challenges associated with the defined security properties.

Opting to extend Schemathesis is not merely a matter of convenience but of strategic alignment. Leveraging its stateful engine not only promises enhanced capability but also intends to bridge the discernible chasms in the current testing milieu. The envisioned augmentation seeks to elevate Schemathesis from a tool of operational diagnostics to a beacon of security assurance in Web APIs.

In the forthcoming chapter, a more granular exploration of Schemathesis and its foundational counterpart, Hypothesis, will be undertaken. This will furnish a deeper understanding of their underpinnings, paving the way for the proposed extensions and culminating in a comprehensive, erudite solution that epitomizes both operational and security excellence in the realm of Web API testing.

Chapter 4

A Closer Look Into Schemathesis

We have already had a general overlook at Schemathesis [17] in Section 3.3, and since the work detailed in the following chapters will be based upon the implementation of this tool, there is a pressing need of clarifying details and inner mechanics regarding Schemathesis [17] and its core component, Hypothesis [21, 22].

The Schemathesis [17] approach is to go over the schema endpoint methods, verifying the schema conformance and the response behaviors, but this tool also has a module for stateful testing, which will be covered in Section 4.2, leveraging data flow from one operation to the following, taking advantage of *links* defined in the OpenAPI schema, as introduced in Section 2.5.

4.1 Schemathesis Under The Hood

In the vast landscape of API testing, Schemathesis [17] stands as a testament to the potency of synergizing multiple components into one cohesive framework. At first glance, the tool may appear as a straightforward testing utility. However, as we delve deeper, its intricacies reveal a tightly woven tapestry of logic, automation, and adaptability.

Schemathesis operates based on a schema-driven approach. By extracting detailed specifications from OpenAPI [25] or GraphQL schemas, it procures a roadmap to guide its testing processes. These schemas offer valuable metadata about endpoints, request formats, response structures, and potential exceptions. Such a detailed and methodical approach ensures that testing is not just surface-level; instead, it is a comprehensive audit of the entire API landscape.

The underlying architecture of Schemathesis revolves around a modular design. Each module focuses on a specific aspect of testing, such as input validation, response assessment, or stateful verification. By segregating these responsibilities, Schemathesis can streamline the testing process, allowing for parallel execution, easier troubleshooting, and modular enhancement.

Hypothesis [21], a robust property-based testing tool, is not merely an adjunct to Schemathesis; it is its beating heart. Where Schemathesis defines the what (i.e., what needs to be tested based on

schemas), Hypothesis defines the how, generating a myriad of test cases that span both typical and edge-case scenarios.

Property-based testing, as championed by Hypothesis, shifts the paradigm from handcrafted test cases to automatically generated tests. Rather than specifying exact inputs to test, developers define properties that outputs should always fulfill. Hypothesis then assumes the onus of generating many test inputs, ensuring these properties hold true across the board.

In the context of Schemathesis, this means that for every endpoint defined in a schema, Hypothesis explores a vast spectrum of potential inputs, both valid and invalid. This exhaustive exploration is invaluable in identifying unanticipated vulnerabilities or erratic behaviors in the API.

Furthermore, the synergistic relationship between Schemathesis and Hypothesis is accentuated when tests uncover failures. Hypothesis's ability to shrink problematic inputs to their simplest form aids in pinpointing the crux of issues, streamlining the debugging and rectification process.

In sum, the melding of Schemathesis's schema-driven strategy with Hypothesis's property-based testing prowess creates a formidable duo. Together, they ensure that API testing is not just rigorous but also efficient, bridging the gap between comprehensive validation and real-world applicability.

4.2 Stateful Testing in Depth

Stateful testing in Web APIs refers to testing the behavior and functionality of an API that relies on the maintenance of specific states or conditions throughout multiple interactions. In stateful testing, the API's behavior is evaluated based on its ability to handle and respond correctly to a sequence of requests that depend on the current state of the system.

Stateful testing is particularly relevant in scenarios where the API's responses or behavior change depending on previous requests or actions. These changes can include modifications to the internal state of the server, such as storing user sessions, tracking data changes, or maintaining authentication tokens.

The purpose of stateful testing is to ensure that the API functions correctly and consistently under various sequences of requests, considering the changing states. By testing different state transitions and their corresponding responses, potential issues like data corruption, incorrect state handling, or inconsistent behavior can be identified and addressed.

Taking a brief, resumed look at how Schemathesis [17] handles stateful testing, we can list the following points:

1. **Test Case Generation** — Schemathesis uses the API schema to generate a set of test cases automatically. It analyzes the available endpoints, request methods, query parameters, request bodies, and response schemas defined in the API specification. Based on this information, Schemathesis generates a variety of test cases, covering different combinations of inputs and edge cases.

2. **Stateful Property Detection** — Schemathesis identifies stateful properties within the API by analyzing the schema and the responses received during testing. Stateful properties can include authentication tokens, session management, or any other server-side state that affects the API's behavior. Schemathesis aims to maintain the necessary states to accurately simulate the behavior of the API.
3. **Stateful Test Execution** — Once the initial test cases are generated, Schemathesis executes them while maintaining the states required by the API. This means that the library tracks and retains any relevant state information, such as cookies or tokens, between subsequent requests.
4. **Stateful Transitions** — During testing, Schemathesis applies stateful transitions by manipulating the relevant states between requests. For example, if a particular endpoint requires authentication, Schemathesis ensures that the authentication token is obtained and used correctly in subsequent requests that require authentication.
5. **Property Validation** — After each request, Schemathesis verifies that the API responses comply with the expected schema defined in the API specification. It checks the response status codes, response formats (such as JSON), and the structure and types of the response data. This ensures that the API behaves correctly and consistently, considering both the state transitions and the defined schema.

By combining property-based testing (verifying adherence to the API schema) with stateful testing (maintaining and manipulating states between requests), Schemathesis [17] provides a comprehensive approach to API testing. It helps uncover issues related to state management, authentication, session handling, and other state-dependent behaviors, ensuring that the API behaves correctly across various scenarios and transitions.

4.3 Hypothesis

Hypothesis [22] is a Python library that provides a framework for property-based testing. Property-based testing, as mentioned in Section 3.1, is a technique for testing software by generating random inputs and checking that the outputs of a program satisfy specific properties.

Hypothesis [22] makes it easy to write property-based tests in Python by providing a decorator that one can use to mark a function as a property-based test. When the test is executed, Hypothesis will automatically generate random inputs for the test function and check that the outputs satisfy the specified properties.

To use Hypothesis [22], one needs to specify the properties to be tested and the types of inputs the test function should accept. Hypothesis [22] will generate random inputs of the specified types and pass them to the test function. We can also use Hypothesis's built-in strategies to specify more complex input generation rules.

Now, remembering the points presented in the previous section about how Schemathesis [17] handles stateful testing, we can establish connections between points and how Hypothesis [22] comes into play:

1. **Test Data Generation** — Hypothesis generates diverse and randomized test data based on the properties or assumptions defined by Schemathesis [17]. This test data can be used as inputs for API requests, simulating different scenarios and covering various edge cases. For example, we can use Hypothesis to generate different combinations of input parameters, payloads, or authentication tokens.
2. **State Transition Simulation** — In a stateful testing environment, the system's behavior often depends on specific states or conditions. Hypothesis can be used to simulate state transitions by manipulating relevant states between API requests. This may involve modifying session data, authentication tokens, or other server-side states. By controlling and modifying states, Hypothesis helps validate how the API behaves under different state changes.
3. **Property-Based Testing** — With Hypothesis, we can define properties or invariants the API should uphold throughout its state transitions. These properties represent expected behavior, constraints, or conditions the API should adhere to. Hypothesis generates test data and evaluates the API's responses, checking if the defined properties are consistently satisfied across different state transitions. If a counterexample is found, Hypothesis provides detailed information to help diagnose and address the issue. These properties are generated by Schemathesis [17] when analyzing the API's schema.
4. **Test Case Reduction** — Hypothesis includes a test case reduction feature (a shrinking mechanism, to be explored in the next section) that automatically simplifies failing test cases to their minimal representation. This reduction process helps identify the core conditions or inputs that lead to a failure. When applied to stateful testing, test case reduction can help isolate and reproduce specific state transitions or interactions that cause undesirable behavior in the API.

By combining the capabilities of Hypothesis with a stateful testing approach, we can generate diverse test data, simulate state transitions, and validate properties or invariants about the behavior of the Web API. This allows for more thorough testing, covering a more comprehensive range of scenarios and helping to uncover potential issues related to state management, API behavior under changing states, and property violations.

Finally, it is crucial to delve deeper into the concept of *bundles* within a stateful testing environment. As articulated by MacIver [21], bundles serve as intermediaries in the process of rules' interaction. In essence, when a rule produces a value (or multiple values) during its execution, these values are provided to a bundle. Later, the same rule or possibly other rules can request values, and the bundle will furnish these previously stored values in return. In this way, a bundle acts as a dynamic repository, evolving based on the information rules supplied to it. Schemathesis [17] creates a *bundle* for each API operation. The state machine keeps storing values regarding

rule execution for each of those *bundles*; the values stored are an instance of the `StepResult` class implemented by `Schemathesis` [17], containing the `Case` and the `Response` objects, and the elapsed time in milliseconds.

4.4 Shrinking: Honing in on Minimal Failures with Hypothesis

In property-based testing, “shrinking” is an essential feature that aids in debugging. Tools such as Hypothesis incorporate a refinement step post-failure [41]. When a test fails, instead of presenting developers with possibly intricate input data, Hypothesis simplifies or “shrinks” the failing test case. The aim is to identify the most straightforward input that yields the same failure.

Consider a scenario where a test fails due to a list of integers. The initial failure might involve a more extended list. Hypothesis will then attempt to ascertain whether a more concise version of this list can replicate the failure. This streamlined failing case offers developers a clearer perspective of the core issue, devoid of unnecessary complexities [42].

Such a refinement process proves indispensable, especially in intricate contexts like Web APIs. Imagine an API faltering due to an embedded SQL injection within a lengthy string. If Hypothesis pinpoints this vulnerability, the shrinking mechanism might deduce that only a minuscule fragment of this string triggers the flaw. This surgical precision not only expedites the debugging process but also illuminates the core vulnerability, facilitating a more robust resolution.

The benefits of shrinking in the domain of stateful testing are particularly pronounced. Stateful tests, by their nature, involve sequences of operations, each potentially influencing the next. A failure in such a scenario might be the culmination of a series of API interactions and state transitions. Here, Hypothesis’s task is to identify the shortest sequence that still culminates in the identified failure. Such distilled insights are invaluable for developers, offering a clear path through the maze of actions and repercussions [43].

Yet, as with any automated process, shrinking brings its own set of challenges. The computational cost of generating numerous reduced variants of the input to determine the most minimal failure can be considerable. Moreover, as test cases grow in complexity, especially in stateful contexts, identifying the shortest action sequence or input set is far from trivial.

To efficiently navigate this process, Hypothesis integrates several heuristics and strategies. One of its core tactics is ordered shrinking, where it operates under the principle that smaller values (like numbers or shorter lists) are simpler, hence giving precedence to these during the shrinking process [41]. The tool also delves into structural shrinking, examining not just value sizes but the structural intricacies of the data, ensuring the reproducibility of failures throughout the shrinking process. Furthermore, its use of lazy data structures allows for efficient exploration of vast spaces of possible values, while relevance-based shrinking and principles from delta debugging optimize the identification of minimal failing cases [42, 47]. Such a blend of heuristics and strategies ensures that the shrinking process remains both comprehensive and efficient.

In wrapping up our exploration of shrinking, it is evident that this feature, though perhaps understated, plays a pivotal role in the property-based testing landscape. By spotlighting the most

elemental failures, Hypothesis equips developers with a laser-focused lens to address and rectify vulnerabilities and bugs [41].

4.5 Extensibility and Flexibility of Schemathesis

As the landscape of Web API security evolves, it becomes indispensable for testing tools to mirror this evolution. Tools need to be malleable, adapting to not just the current but also foreseeable complexities. This brings us to the heart of Schemathesis: its core design revolving around extensibility. Through a comprehensive array of features, it empowers developers to fine-tune its offerings, tailoring them to address specific testing prerequisites.

Several avenues for customization exist within Schemathesis, encompassing nuances from tweaking the data generation strategy (courtesy of Hypothesis) to redefining how test properties themselves are fashioned. This section provides an exposition on the various extension modalities and their alignment with bespoke testing needs.

An exemplar of Schemathesis's extensibility is furnished by its official documentation [44]. Here, we are introduced to a scenario where an endpoint, `/api/auth/password/reset`, anticipates a `token` within the request payload. Testing endeavors often demand exploring the boundaries of plausible token values. To that end, Listing 4.1 elucidates how Schemathesis, aided by Hypothesis strategies, facilitates the dynamic modification of the payload — interchanging between a randomly generated token, a legitimate token from an arbitrary email or one from a known email — line 5 declares the target endpoint for this test, line 8 retrieves a random boolean, making the `if` statement true in around 50% of the cases, if the clause is true, then the body's `token` property will be updated with a token based on a random email or the target user's email, using the Hypothesis `draw` method, finally, it will make the call to the API and validate its response.

```
1 from hypothesis import strategies as st
2
3 schema = ... # Load the API schema here
4
5 @schema.parametrize(endpoint="/api/auth/password/reset/")
6 @schema.given(data=st.data())
7 def test_password_reset(data, case, user):
8     if data.draw(st.booleans()):
9         case.body["token"] = data.draw(
10             (st.emails() | st.just(user.email)).map(create_reset_password_token)
11         )
12     response = case.call_asgi(app=app)
13     case.validate_response(response)
```

Listing 4.1: Additional Hypothesis Strategies Example

A testament to Schemathesis's adaptability is its harmony with external modules. Such alliances bolster its scope, allowing for a rich and integrative testing milieu. For instance, Schemathesis's interoperability with third-party tools like `pytest-asyncio` equips developers to proficiently test asynchronous APIs, harnessing the full potential of Python's asynchronous paradigm.

Advancing further into its capabilities, the stateful testing mode of Schemathesis warrants a detailed discussion [37]. As touched upon earlier, this mode pivots on the `links` delineated in the OpenAPI schema, leveraging them to sculpt a state machine founded on the Hypothesis Rule-based State Machine [34]. The official guide [37] illuminates the methods for extending this stateful interface, prescribing the requisite implementable methods.

For optimizing test scenarios, Schemathesis facilitates certain strategic interventions: the `setup` and `teardown` methods, alongside the `initialize` decorator from Hypothesis. These tools, working in unison, ensure a consistent starting point for every test iteration. For instance, while `setup` can be maneuvered to define administrative tokens or initialize data structures, an associated method can spawn multiple user profiles, which can then be methodically eradicated during `teardown`. This orchestrated sequence fortifies the consistency of test preconditions, obliterating disparities between successive runs.

However, achieving finesse in test scenario commencement and conclusion is only part of the puzzle. The heart lies in refining the core test processes. We have talked about how we can enhance both the start and the end of a test scenario, but how can we extend/enhance the actual tests and their generation? Schemathesis documentation [37] is not very clear on this topic, at least, in a straightforward way; after digging into the actual implementation of the state machine, we discovered that Hypothesis `rules` were being created (as expected), but more importantly, how they were being created; this enabled us to better understand how to further enhance the test case generation to include extra steps/transitions in the state machine; e.g., instead of the regular state transitions, checking the Schemathesis defined conditions (schema conformance, data input and response validation, etc.), we could insert an extra step between one state and the other to verify a special condition (for instance, if an header is present in the response) and (optionally) make that test fail, or even include a more complex logic, for example transitioning into another completely different state; ultimately, the possibilities are endless, but the complexity is also elevated, since we are now dealing with lower level code, and interacting directly with Hypothesis mechanisms. Listing 4.2 illustrates how can we include additional rules into Schemathesis workflow for API testing: we have omitted irrelevant methods for this example (to be better explained in the next chapter), in line 3 we have added a method to initialize the testing scenario with multiple users, using the `POST /api/users` method, assuming there is a link from this request to `GET /api/users/:user_email`, we've also defined an extra `rule` in order to check if violations to access control policies are found (line 13); this `rule` has an optional `target bundle`, meaning after this rule's execution, it will write to that bundle signaling a state transition; the `rule` decorator accepts multiple extra parameters, we've decided to include the `StepResult` containing the last successful call found to `GET /api/users/:user_email`, in order to assess if that call should actually have been successful: lines 18-22 assert if the call had no violations, reporting the

error otherwise; if everything goes as expected the original `StepResult` is returned, creating a new state for that *bundle*.

```

1      ...
2      class APIWorkflow(BaseAPIWorkflow):
3          @initialize(
4              target=BaseAPIWorkflow.bundles["/api/users"]["POST"],
5          )
6          def init_users(self):
7              result = []
8              for user in USERS:
9                  case = schema["/api/users"]["POST"].make_case(body=user)
10                 result.append(self.step(case))
11             return multiple(*result)
12
13         @rule(
14             target=BaseAPIWorkflow.bundles["/api/users/{user_email}"]["GET"],
15             step_result=APIWorkflow.bundles["/api/users/{user_email}"]["GET"].
16                 filter(match_status_code("2XX"))
17         )
18         def check_access_controls(self, step_result: StepResult):
19             try:
20                 check_if_no_violation(step_result)
21             except ViolationException as e:
22                 report_violation(e)
23                 raise e
24
25             return step_result
26     ...

```

Listing 4.2: Hypothesis Rule Creation

Defining rules with decorators, as shown in Listing 4.2, can be a repetitive task, even more so if the Web API contains multiple endpoints that need validation; fortunately, Python provides means to dynamically create a class, generating methods to encapsulate in its blueprint, this allowed us to dynamically define multiple rules for multiple endpoints, without having to manually define each rule. An example of this can be found in Listing 4.3: we have defined an abstract workflow to manage the dynamic rule creation leading up to the generation of a new testing workflow, containing the extra rules defined; each rule is dynamically generated in `get_additional_rules`, and finally included in the creation of a new object type in `get_api_workflow`; this is equivalent as manually defining each `rule` decorated method in the previous example, and the advantage is clear: an automatic way of defining `rule` methods, decreasing the development time complexity.

```

1      ...

```

```

2  class Workflow:
3      class APIWorkflow(self.schema.as_state_machine()):
4          # generic workflow initialization
5
6      self.APIWorkflow = APIWorkflow
7      self.method_path_rules = get_method_path_rules()
8
9      def get_additional_rules(self) -> dict:
10         return {
11             f"check_access_control_{method}_{path}": rule(
12                 target=self.APIWorkflow.bundles[path][method],
13                 step_result=self.APIWorkflow.bundles[path][method].filter(
14                     match_status_code("2XX")
15                 ),
16             )(check_access_control)
17         for method, path in self.method_path_rules
18         }
19
20     def get_api_workflow(self):
21         # inject the rules and the evaluation service into the APIWorkflow
22         class
23         return type(
24             "APIWorkflow",
25             (self.APIWorkflow,),
26             {**get_additional_rules, **other_arguments},
27         )
28
29     # get workflow with new rules
30     workflow = Workflow().get_api_workflow()
31     ...

```

Listing 4.3: Dynamic Hypothesis Rule Creation

In essence, Schemathesis's foundational architecture, rooted in Python and enriched with type annotations, serves as a beacon for extensibility. While it delivers a potent suite of built-in functionalities, its true prowess lies in its ability to be molded and extended, seamlessly aligning with diverse testing landscapes.

4.6 Summary and Key Takeaways

In concluding this chapter, it is pertinent to encapsulate the salient features of Schemathesis and elucidate the rationale that underpins our selection of this tool as the bedrock for our research endeavors.

Schemathesis, when employed in conjunction with Hypothesis, stands out even in its rudimentary mode. When assessing on a per-endpoint basis, it proffers substantial capabilities. Foremost among these is its adeptness in both testing and the autogeneration of input data. Moreover, the inherent extensibility of Schemathesis cannot be understated. It offers the latitude to integrate custom strategies for data input, along with the provision to add specialized checks—augmenting those already inherent to Schemathesis.

A significant attribute that warrants special attention is the stateful testing module. This feature resonates acutely with our research imperatives. It facilitates an in-depth exploration into the nuances of how sequences of requests can modulate the behavior of an API and subsequently mold the responses of future requests. The ingenuity of the stateful approach is further accentuated by the accompanying shrinking mechanisms. Without such mechanisms, error analysis could become an arduous undertaking. Consider a scenario wherein a prolonged sequence of requests culminates in an error. The shrinking mechanism in Schemathesis enables the identification of a more concise sequence that triggers the same error, thus streamlining the error diagnosis process.

While the capabilities of RESTler were thoroughly assessed and acknowledged, it was discerned, after meticulous contemplation, that Schemathesis would offer a more lucid path for augmentation and intricate analysis. Several facets informed this decision. The fact that Schemathesis is constructed in Python presents an immediate environment of familiarity and ease of maneuverability. Furthermore, Schemathesis's capability to orchestrate a state machine is particularly advantageous when endeavoring to conduct stateful tests, in contrast to RESTler, which has no framework for crafting tests depending on sequences of requests; given our defined properties to test Web APIs against, Schemathesis stateful testing module was a significant upside, which made the choice between RESTler and Schemathesis as ground tool easier.

In the next chapter, we will be putting these extensibility features up to test, implementing mechanisms to check for the security properties mentioned in Section 2.4.

Chapter 5

XSecEngine

In this chapter, we will provide the details of the empirical work and implementation of the tool, *XSecEngine*, the main contribution, designed to check the security properties defined in Section 2.4, as well as the extensions proposed to OpenAPI Specification [25] in order to facilitate the checking of the aforementioned properties, and ultimately, to better document the API in terms of security.

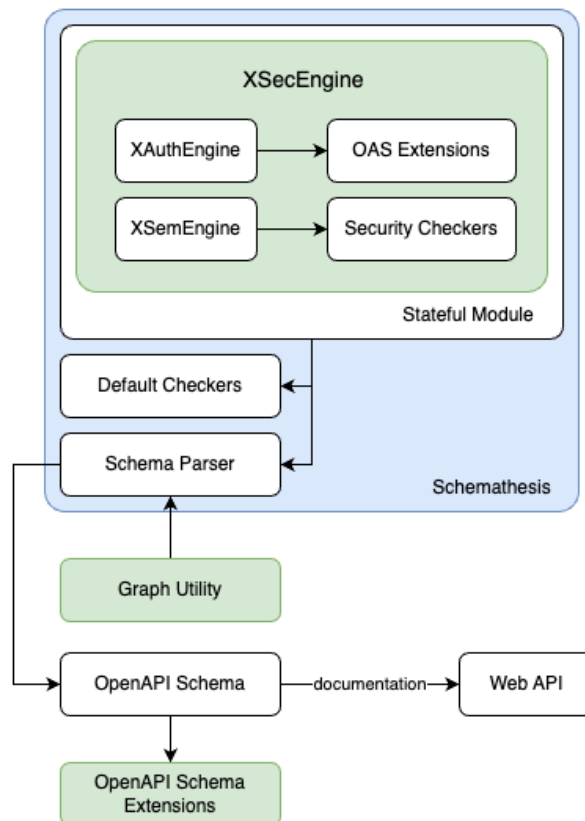


Figure 5.1: XSecEngine Architecture Diagram

XSecEngine is an extension to Schemathesis [17] that consists of two main components: *XSemEngine* and *XAuthEngine*; in short, *XSemEngine* is responsible for enforcing the semantic and security checks while *XAuthEngine* is responsible for parsing the schema extensions.

Figure 5.1 provides a high-level picture of our solution’s architecture (our contributions are marked as green): the figure provides a very simple diagram regarding Schemathesis concepts, with a spotlight on *XSecEngine* and its components. All of the concepts marked as green will be the focus of this chapter.

5.1 OpenAPI Extensions for Security Testing

As previously pointed out in Section 2.5, the OpenAPI Standard [25] lacks the means to define ownership, context, and mutators, the reason for which we have introduced an extension to the Standard, *x-auth*. This extension, henceforth named *XAuth*, consists of **3 components**: *mappers*, *overrides* and *mutators*.

1. **Mappers** — an object with key-value elements, essentially like a map data structure. This object contains information about how the claims should be connected to the request’s context, i.e., which values in the token are allowed for the request to be made. The key-value properties can be abstracted to claim-value, where the claim is the property in the token (email, iat, id, etc.), and the value can be either a static string or a dynamic value such as a query parameter if it starts with the character \$. This can be very useful when a path is `/api/users/:user_email` to define that only tokens with the claim `email` matching the query parameter `user_email` are valid; the mapper would be `email:$user_email`; these mappers can also be extended to support dynamic references to other values such as headers, using the same mechanism to reference query parameters, with a character \$, we could also reference header values by another character;
2. **Overrides** — this object contains the necessary properties to override regarding the last property (*mappers*), it contains an array `scopes` to be filled with scopes that are expected to override any violations found in the *mappers*; using the previous example, an administrator (scope: `admin`) has the wrong claim for the email, and therefore they would not be able to manage the user information, although it is a system requirement; therefore, we could add the scope `admin` to the *scope overrides* to let the user know that the request can also be made by administrators, even though they may not have the correct claim value present in the token;
3. **Mutators** — an array of objects containing information about which endpoints “mutate” this request’s resource, i.e., which endpoints change the resource that’s tied to the current endpoint; as an example, the endpoint for GET at `/api/users/:user_email` has many “mutators”, one of which could be a PATCH for the same path, because the PATCH changes

specific properties for a resource. Of course, some operations may not have direct mutators, such as the `GET /api/health-check` that asks the server the current status, and others may not directly relate to a resource but rather a collection of resources or even a function result. Nevertheless, this property helps to give more semantics to the schema, providing helpful information about how the endpoints work together, rather than just having the `links` defined, which would only give information about which data could be used from one endpoint's response to another (as covered in Section 2.5). Finally, this property is crucial to guide the tests related to mutators and the semantics they provide.

In order to better illustrate these concepts, a part of the schema for the prototype Web API is presented in Listing 5.1 (irrelevant parts omitted for readability, and extensions highlighted as red). Starting with the first operation, `POST /api/v1/users`, this request does not require authentication, and the response can be used in the `GET` and `PATCH` methods for the path `/api/v1/users/{user_email}`. Both the `PATCH` and `GET` methods for `/api/v1/users/{user_email}` contain the `x-auth` property, mapping the token claim email to the query parameter `user_email` and accepting the scope `admin`. The `GET` method also has a “mutator” defined pointing to the `PATCH` method. To simplify, both the `GET` and `PATCH` methods aforementioned require the user claim to match the query parameter in the path to ensure the user making the request is the owner of that resource unless the user making the request is an administrator (`admin` scope present in the token), in which case the mappers' rule should be overwritten; finally, the `GET` operation has information about the `PATCH` method stating that a request to that endpoint will change the current resource.

With this extension, two of the proposed security properties defined in Section 2.4, Broken Object & Function Level Authorization, can now be more easily tested using this extension. The other security properties will take advantage of the *mutators* defined in the schema extensions, to be covered in the next section.

One downside we can point out regarding this extension is that many lines of code need to be added to the schema where that information is needed for testing and/or documentation purposes, which can be a very time-consuming procedure. Still, as there are tools to dynamically generate an OpenAPI schema, the same principle could also be applied here; we will discuss this further in Section 6.4.

```
1  ...
2  /api/v1/users:
3    post:
4      ...
5      operationId: users_create_api_v1_users_post
6      responses:
7        '201':
8          links:
9            UserGetByEmail:
10              operationId: users_get_by_email_api_v1_users__user_email__get
```

```

11         parameters:
12             user_email: '$response.body#/email'
13         UserUpdateByEmail:
14             operationId:
15                 users_update_by_email_api_v1_users__user_email__patch
16             parameters:
17                 user_email: '$response.body#/email'
18 /api/v1/users/{user_email}:
19     get:
20         ...
21         x-auth:
22             mappers:
23                 email: $user_email
24             overrides:
25                 scopes:
26                     - admin
27             mutators:
28                 - endpoint: PATCH /api/v1/users/{user_email}
29         operationId: users_get_by_email_api_v1_users__user_email__get
30         responses:
31             '200':
32                 links:
33                     UserUpdateByEmail:
34                         operationId:
35                             users_update_by_email_api_v1_users__user_email__patch
36                 parameters:
37                     user_email: '$response.body#/email'
38         security:
39             - OAuth2PasswordBearer:
40                 - read:users
41     patch:
42         ...
43         x-auth:
44             mappers:
45                 email: $user_email
46             overrides:
47                 scopes:
48                     - admin
49         operationId: users_update_by_email_api_v1_users__user_email__patch
50         security:
51             - OAuth2PasswordBearer:
52                 - update:users

```

Listing 5.1: Schema Example for Users' API

5.2 Checking Security Properties

To test for the security properties defined in Section 2.4, after adding the links between operations and the extensions needed, as covered in the previous Section, we had to extend the base class for stateful testing implemented by Schemathesis [17] based upon Hypothesis [21].

The main extensions to the implementation were mainly the addition of new `rules` to the stateful engine implemented by Schemathesis [17], the `rule` interface (decorator) was implemented by Hypothesis [34]; rules are essentially a set of properties that are checked in a stateful environment, they have the previous context and may have a target *bundle* to write on, before state transitions.

According to MacIver [34], multiple rules can be chained together: a test using Hypothesis's state machine does not just run each rule in isolation; it creates an instance of the machine and then runs multiple rules in succession; this is needed to simulate a real environment system, where the user may issue multiple requests to different endpoints, and simulate how they work together and what vulnerabilities they may expose.

5.2.1 Broken Object & Function Level Authorization

To probe for vulnerabilities in Broken Object-Level Authorization and, by extension, Broken Function-Level Authorization (as delineated in Section 2.4), we deliberately introduced an exploitable flaw. Specifically, in the GET request path of `/api/users/:user_email`, we eliminated the critical user-matching step when processing the request. As portrayed in Listing 5.2, the vulnerability manifests when the verification condition on line 12 is removed. By doing so, the system inadvertently grants access to any authenticated user, irrespective of whether the request originates from the correct user or an administrator. This oversight, deceptively minimalistic in nature, can inadvertently expose an entire system. The vulnerability is a stark illustration of how minor code changes or overlooked lines of code can lead to significant security breaches.

```

1 @router.get(
2     path="/api/users/{user_email}",
3     dependencies=[Security(get_current_user, scopes=[user_scopes.read()])],
4 )
5 def user_get_by_email(
6     user_email: EmailStr,
7     db=Depends(get_db),
8     current_user=Depends(get_current_user)
9 ) -> UserOut:
10     # Commenting out the following lines renders this endpoint insecure,
11     # allowing any authenticated user unrestricted access
12     if current_user.email != user_email and "admin" not in current_user.scopes:
13         raise UserUnauthorizedException()
14 
```

```
15     return UserService.get_user_by_email(db, user_email)
```

Listing 5.2: Vulnerability in GET /api/users/:user_email

Referring back to the OpenAPI extension incorporated into this endpoint (as presented in Listing 5.1, lines 20-25), the endpoint's access is stipulated to be limited to the query parameter `alice@test.com`, specifically with either Alice's or an administrator's authentication token. A straightforward solution for this verification might be integrating a new check within the foundational Schemathesis checks; however, we wanted our solution to be decoupled from Schemathesis to a certain extent, and the next properties, namely the ones requiring a stateful execution, could not be implemented using Schemathesis native checker interface.

To address this limitation, we crafted a custom rule to be enforced for each endpoint; this rule is triggered upon receiving a successful response, and it sets in motion a transition contingent on the successful response status (2xx), with the vulnerable endpoint as the target for evaluation. Upon state transition, the system initially checks for `x-auth` schema information. Since Schemathesis lacks native support for this, we parse the information manually. The ensuing evaluation procedure (as shown in code in Listing 5.3) can be distilled into the following sequence:

1. Ascertain the presence of authorization headers. The absence of these headers indicates that the API is not mandating authorization. In such instances, the evaluation returns early, registering no errors.
2. Extract and interpret the token. For successful decryption of payload data, one necessitates both the encryption algorithm and the secret key — details typically accessible to developers or sourced from CI secret stores.
3. Dissect the `x-auth` OpenAPI schema property to retrieve both `mappers` and `override` attributes. It is noteworthy that an endpoint can house multiple mappers without constraints. For instance, a mapper might correlate the `email` claim with the `user_email` query parameter, and another might associate the `organization` claim with the string `orgXYZ`. Concurrently, numerous override scopes can also be defined.
4. Collate request details (comprising query or path parameters and the utilized token) with the data aggregated in the preceding step. Any discrepancies indicative of security breaches are flagged. Consequently, this step tags the endpoint as a failure, prompting Hypothesis to distill the request sequence into a minimal reproducible failure instance.

```
1 def check_x_auth(cls, state_machine, step_result: StepResult) -> StepResult:
2     if not step_result.case.headers:
3         return step_result
4
5     # change headers to simulate making a request as another user
6     # find the first token that's not equal to the case header
```

```

7     token = get_token_from_header(
8         step_result.case.headers.get("Authorization", None)
9     )
10    user, token = next(
11        filter(lambda context: context[1] != token, state_machine.contexts),
12        None,
13    )
14
15    case = step_result.case
16
17    case.headers["Authorization"] = get_token_header_value(token)
18    response = case.call_and_validate()
19
20    try:
21        # check if the claims and overrides are conformant
22        XAuthEngine.check_x_auth(response, case)
23    except AssertionError as e:
24        # do something with the error, e.g., logging, telemetry, etc.
25        raise e
26
27    return step_result

```

Listing 5.3: Broken Object & Function Level Authorization Rule

5.2.2 Immutability in Subsequent GET Requests

In order to check for this property, we have defined an extra `rule` for each GET endpoint available; this rule has the purpose of making two consecutive GET requests to the same endpoints, with the exact same query or path parameters, then it will compare the responses' status codes and bodies; if there are differences, it will throw an error signaling that the endpoint, upon request, is leaking a side effect that should not be permitted. Listing 5.4 presents an example of this issue; in line 13, we see that a call is made to `update_user_base` (this is an example), and this method changes a property into a random integer for every user in the database. This is clearly a problem, although harmless in our prototype, but can shed light on real-world scenarios where inappropriate actions are taken, leaking side-effects into resources, e.g., inadvertently changing a user's credit card information with a GET call.

```

1     @router.get (
2         path="/api/users/{user_email}",
3         dependencies=[Security(get_current_user, scopes=[user_scopes.read()])],
4     )
5     def user_get_by_email (
6         user_email: EmailStr,
7         db=Depends (get_db) ,
8         current_user=Depends (get_current_user)

```

```

9      ) -> UserOut:
10      if current_user.email != user_email and "admin" not in current_user.scopes:
11          raise UnauthorizedException()
12
13      update_user_base(db)
14
15      return UserService.get_user_by_email(db, user_email)

```

Listing 5.4: Side-effect in GET /api/users/:user_email

As we’ve discussed in Section 2.4, there are some exceptions to this property, e.g., sending a trace ID in the response body in order to facilitate debugging and audit: this would cause a false-positive to be thrown; such cases need to be identified and handled, the prototype we have implemented behaves as expected, i.e., without these type of exceptions. In Listing 5.5, we can see how this rule’s core was implemented. We see an importance in describing the `state_machine` methods called in this rule and the following ones: `before_call` and `after_call` are utility methods designed to help developers take actions before and after a request is made (log information, update fields, etc.), `call_and_validate` is the method responsible for actually making the request to the Web API as well as validate its response, using Schemathesis core checkers (conformance with the schema). In the aforementioned listing, we issue two consecutive calls to the GET endpoint defined in the `bundle.case` data structure, and then proceed to compare the two responses, throwing an error if they are not similar.

```

1 def check_two_gets(cls, state_machine, bundle: StepResult):
2     state_machine.before_call(bundle.case)
3     result_1 = bundle.case.call_and_validate()
4     state_machine.after_call(bundle.case)
5
6     state_machine.before_call(bundle.case)
7     result_2 = bundle.case.call_and_validate()
8     state_machine.after_call(bundle.case)
9
10    try:
11        cls._compare_responses(
12            response1=result_1.response, response2=result_2.response
13        )
14    except AssertionError as e:
15        # do something with the error, e.g., logging, telemetry, etc.
16        raise e

```

Listing 5.5: Immutability in Subsequent GET Requests Rule

5.2.3 Non-Interference Between an Out-of-Context Mutator Request

Revisiting this property's definition, in Section 2.4,

When making requests to an API, it is anticipated that each request functions within its own confines without unintended spill-overs. This property is about ensuring that a mutator request, even if it is out of context, does not meddle with or modify unrelated resources. It is a tenet of ensuring that actions within a system are compartmentalized and controlled.

an example of a violation of this property can be found in Listing 5.6: line 14 introduces a side-effect into another user's data, unrelated to the target user; this is a minimal example of a violation, but, again, real-world applications can suffer from this, e.g., by inadvertently changing another resource using a wrong SQL query. We have implemented a custom rule to be applied to each GET request available that contemplates at least one *mutator* in the `x-auth` extension property, declared in the OpenAPI schema; this rule's implementation details can be found in Listing 5.7, and, in sum, this rule verifies if a *mutator* request made to a different context alters the value on the original resource returned by the GET endpoint.

```

1  @router_v1.patch(
2      path="/{user_email}",
3      dependencies=[Security(get_current_user, scopes=[user_scopes.update()])],
4  )
5  def user_update_by_email(
6      user_email: str,
7      update: UserUpdate,
8      db=Depends(get_db),
9      current_user=Depends(get_current_user)
10 ) -> UserOut:
11     if current_user.email != user_email:
12         raise UnauthorizedException()
13
14     method_with_side_effect(other_user_email)
15
16     return UserService.update_user(db, user_email, update)

```

Listing 5.6: Side-effects in PATCH `/api/users/:user_email`

Moreover, it is very hard to generate valid input data for the *mutator* request without previous knowledge; therefore, we have set this rule to be applied once the mutator defined for the endpoint had already been reached and returned a success status code (2xx); this way we can ensure the *mutator* request will be successful between two GET requests for the same endpoint. One important detail about this rule is that since it is almost impossible to randomly generate a query or path parameter for the *mutator* request **different** than the one for the GET request, this rule will only

be valid for examples where these two contexts are different, this drastically reduces the probabilities of this rule being checked, but in a large search space it is theoretically possible to find two non-overlapping contexts.

```

1 def check_get_mutator_non_interference(
2     cls, state_machine, get: StepResult, mutator: StepResult, case: Case
3 ):
4     if mutator.case.path_parameters == get.case.path_parameters:
5         # we need two different contexts in order to check this rule
6         return StepResult(...)
7
8     state_machine.before_call(get.case)
9     response_1 = get.case.call_and_validate()
10    state_machine.after_call(get.case)
11
12    case.path_parameters = mutator.case.path_parameters
13    case.headers = mutator.case.headers
14    state_machine.before_call(case)
15    response_2 = case.call_and_validate()
16    state_machine.after_call(case)
17
18    state_machine.before_call(get.case)
19    response_3 = get.case.call_and_validate()
20    state_machine.after_call(get.case)
21
22    try:
23        cls._compare_responses(response_1, response_3)
24    except AssertionError as e:
25        # do something with the error, e.g., logging, telemetry, etc.
26        raise e
27
28    return StepResult(...)

```

Listing 5.7: Non-Interference Between an Out-of-Context Mutator Request Rule

5.2.4 Mutability Between a Mutator Request

This property’s objective is to ensure a resource is altered upon the processing of a *mutator* request. This property can be violated by “forgetting” to update a resource, e.g., using a wrong SQL query, or even by using a wrong abstract method, as presented in Listing 5.8.

```

1 @router_v1.patch(
2     path="/{user_email}",
3     dependencies=[Security(get_current_user, scopes=[user_scopes.update()])],
4 )
5 def user_update_by_email(

```

```

6         user_email: str,
7         update: UserUpdate,
8         db=Depends(get_db),
9         current_user=Depends(get_current_user)
10    ) -> UserOut:
11        if current_user.email != user_email:
12            raise UserUnauthorizedException()
13
14        return UserService.get_user(db, user_email)

```

Listing 5.8: Update Bypass in PATCH /api/users/:user_email

We have defined an additional rule to check for this property: this rule was applied to every GET endpoint that had at least one *mutator* defined in the schema under the x-auth extension property. The objective was to get the initial value for the response by issuing a request to the GET endpoint, then proceed to make a request to the mutator endpoint (with the same query and path parameters), and, finally, retry the first GET request. The first and the last GET requests should contain different values, provided the *mutator* was successful and contains a payload that will mutate the resource into a new value. The core implementation for this rule can be consulted in Listing 5.9.

We have identified some examples where Hypothesis would generate input data that would not change the resource, thus incurring a false-positive, since the engine verified that both GET requests contained the same value; these edge cases need to be confirmed.

By extension, similar to the last rule, this rule depends on the engine having found a successful *mutator* request in order to be applicable.

```

1 def check_get_mutator(cls, state_machine, get: StepResult, mutator: Case):
2     case.headers = get.case.headers
3     case.path_parameters = get.case.path_parameters
4
5     state_machine.before_call(get.case)
6     response_1 = get.case.call()
7     state_machine.after_call(response_1, get.case)
8
9     state_machine.before_call(mutator)
10    response_2 = mutator.call_and_validate()
11    state_machine.after_call(response_2, mutator)
12
13    state_machine.before_call(get.case)
14    response_3 = get.case.call()
15    state_machine.after_call(response_3, get.case)
16
17    if not response_2.ok:
18        # responses 1 and 3 should be equal
19        try:
20            cls._compare_responses(response_1, response_3)

```

```

21         except AssertionError as e:
22             # do something with the error, e.g., logging, telemetry, etc.
23             raise e
24
25         return StepResult(...)
26
27         # else responses 1 and 3 should be different; this can raise false
28         # positives if the mutator does not change the response,
29         # e.g., by making a request with the same data
30         try:
31             cls._compare_responses_should_be_different(response_1, response_2)
32         except AssertionError as e:
33             # do something with the error, e.g., logging, telemetry, etc.
34             raise e
35
36         return StepResult(...)

```

Listing 5.9: Mutability Between a Mutator Request Rule

5.3 Implementation Details

In order to provide readers with additional insight regarding the implementation details, this section will cover a few points we would like to clear out. This will help understand some rationales regarding the implementation as well as to facilitate further work on this solution.

In Listing 5.10, a basic, generic version of a testing workflow is presented: the main class, `APIWorkflow`, is an extension from our implementation (containing the core functionalities and rules, as defined in the previous section), this class provides three essential overridable methods: `setup`, `init` and `teardown`; these methods' role has already been described in Section 4.6. Additionally, we have also implemented an *Evaluation Service* in order to test this implementation performance, as defined in line 5. This code snippet provides a simpler interface for users to implement their own logic regarding the API they are trying to test with our implementation; moreover, in Appendix A, readers can analyze and take as examples the two implementations we have made for the two APIs we have tested (to be explored in the next Chapter).

```

1 schema = schemathesis.from_path(
2     config.OPENAPI_SCHEMA_PATH, base_url=config.BASE_URL
3 )
4
5 evaluation_service = EvaluationService(results_dir=config.RESULTS_DIR)
6
7 workflow = Workflow(evaluation_service=evaluation_service)
8 BaseAPIWorkflow = workflow.get_api_workflow()
9

```

```
10 class APIWorkflow(BaseAPIWorkflow):
11     def setup(self):
12         # method that runs in the beginning of every test scenario
13         pass
14
15     @initialize(target)
16     def init(self):
17         # method responsible for initializing any data or data structures
18         # needed for the test execution
19         pass
20
21     def teardown(self):
22         # method responsible for clearing everything, resetting the API state
23         # to the initial scenario, this is important to maintain consistency
24         # between test executions
25         pass
26
27 workflow.print_links()
28 APIWorkflow.TestCase.settings = settings(...)
29
30 APIWorkflow.run()
31
32 evaluation_service.generate_evidence()
```

Listing 5.10: Boilerplate for Test Workflow

In Listing 5.10, line 27, we can see that a call to `print_links` was made; we have found that, for large OpenAPI schemas, it was hard to visualize the connections (links) between endpoints, declared in the schema, and by extension, it was hard to grasp the search space exploration that would be taken into the tests. In order to alleviate this issue, we have implemented a custom graph data structure to show the sequences that Schemathesis will try to test, as well as save that graph in a `.dot` file in order to better assess the scope visually. The implementation details can be found in Appendix B.

Chapter 6

Evaluation and Results

In this chapter, we will be discussing the aspects of the extension to Schemathesis [17] we have implemented. We have tested our own Prototype API (see Section 2.3 for details), as well as a production-ready, real-world API, FusionAuth [13], an authentication, authorization, and user management system that serves as a comprehensive solution for securing web and mobile applications. It is designed to be a more flexible and scalable alternative to other identity solutions, catering to modern application needs. The evaluation conducted will help answer the research questions defined in Chapter 1, as well as provide insight on the limitations and the complexity associated with testing Web APIs using our extension to Schemathesis and to the OAS.

The evaluation consisted of testing multiple scenarios for both our Web API Prototype and FusionAuth, trying to find violations to the security properties we have defined in Section 2.4. In addition, in order to have a comparable benchmark, we have also tested the scenarios with RESTler [4], since it was the only tool that supported checking for security properties. We have also collected time metrics in order to evaluate the implementation's performance by aggregating results over a sample of three executions.

6.1 Web API Prototype

To rigorously assess our implementation's efficacy in identifying vulnerabilities related to the security properties elucidated in Section 2.4, we architected an assortment of test cases. The specifics of these tests are chronicled in Table 6.1. Additionally, for each test scenario, we juxtaposed our API's performance with RESTler's. This comparison was essential to discern if RESTler could detect the affiliated error, thus enabling us to gauge the precision and robustness of our tool. It is important to note that when we mention that *RESTler should not be able to detect a property*, we are referring to RESTler's original implementation; in theory, based on the discussion in Section 3.2, it is possible to implement active checkers to verify the properties we have defined, although we have not gauged the complexity and effort it would take. The comparison between

our tool and RESTler is mostly for performance (run time) measures while checking if RESTler could detect any of the vulnerabilities introduced.

Test ID	Bug Introduced	Schema Details	Affected Endpoints	Expected Errors
1	N/A	Enhanced	N/A	None
2	Authorization bypass in GET user	Default	GET /api/v1/users/:user_email	None
3	Authorization bypass in GET user	Enhanced	GET /api/v1/users/:user_email	X_AUTH_VIOLATION
4	Authorization bypass in PATCH user	Default	PATCH /api/v1/users/:user_email	None
5	Authorization bypass in PATCH user	Enhanced	PATCH /api/v1/users/:user_email	X_AUTH_VIOLATION
6	Inject random integer in GET user	Enhanced	GET /api/v1/users/:user_email	X_SEM_CHECK_2_GETS
7	Inject bug in PATCH user	Enhanced	PATCH /api/v1/users/:user_email	X_SEM_NON_INTERFERENCE
			PATCH /api/v1/users/:user_email	X_SEM_MUTATOR

Table 6.1: Prototype API Test Scenarios

In light of the scenarios presented in Table 6.1, the *Bugs Introduced* column contains information on the changes made to the prototype Web API:

1. **Authorization bypass in GET user** — Bug introduced to bypass the user check for GET /api/v1/users/:user_email, this bug allows users with a different email than the target user’s email to be able to fetch information about the user. This vulnerability was originally defined in Section 5.2, Listing 5.2. After analyzing RESTler [4], we have concluded that it should not be able to detect this vulnerability;
2. **Authorization bypass in PATCH user** — Bug introduced to bypass the user check for PATCH /api/v1/users/:user_email, it is very similar to the previous item. In sum, it allows unauthorized users to update information about a different user. Similarly, RESTler should not be able to detect this vulnerability;
3. **Inject random integer in GET user** — This bug injects a random integer in the response body when performing a GET request to /api/v1/users/:user_email, causing GET requests to have a non-deterministic behavior. RESTler should be able to detect the inconsistency between two consecutive GET requests;
4. **Inject bug in PATCH user** — This bug consists on *forgetting* to update the user, retrieving it un-updated. RESTler should not be able to detect this issue.

The *Schema Details* column in Table 6.1 represents how the OpenAPI schema was configured in each test scenario: (1) Default — the schema has no extensions proposed by our approach, i.e., no x-auth properties, in any endpoint; (2) Enhanced — by contrast, an enhanced OpenAPI schema has the information needed for our tool to be able to process responses and assert if the properties are not violated.

It is imperative to emphasize that there is a consistent, unconditional definition of links between endpoints for both our tool and RESTler. The interconnections among the endpoints pertinent to our prototype API are visually depicted in Figure 6.1, where each arrow denotes a potential information flow.

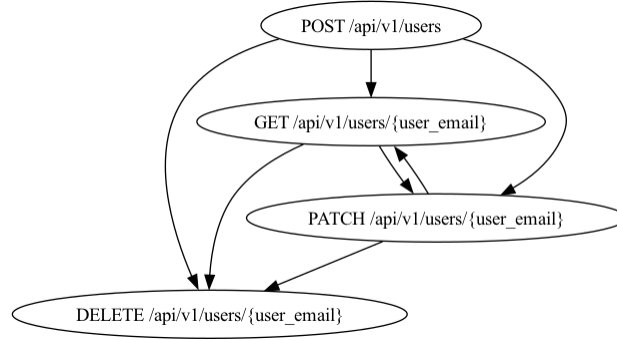


Figure 6.1: Web API Prototype Links Graph

Finally, the column *Expected Errors* in the Table 6.1 represent the types of errors we expect the tool to detect, each value directly corresponds to one property defined in Section 2.4:

X_AUTH_VIOLATION — Broken Object & Function Level Authorization;

X_SEM_CHECK_2_GETS — Immutability in Subsequent GET Requests; X_SEM_MUTATOR — Mutability Between a Mutator Request; X_SEM_NON_INTERFERENCE — Non-Interference Between an Out-of-Context Mutator Request.

In our assessment strategy, for every test scenario, we initiated three successive testing suites without erasing the Hypothesis generated files. This method not only expedited the error detection process during subsequent runs but also strived for a more expansive coverage of the search space. The outcomes procured for the prototype API via our tool are encapsulated in Table 6.2, while Table 6.3 chronicles the results using RESTler. The results tables contain the outcome of the test (*Result*): *Expected* means the test did not provide any unexpected results, i.e., if we were expecting an error type, that error was found, and if no error was expected, then no error was reported. The columns for *Time Till First Error* may contain *N/A*, signaling that no error was found, and therefore there is no value to report. Finally, the columns for *Runtime* relate to the total time a test scenario took.

Test ID	Result	Average TTFE ¹	σ ² TTFE ¹	Average Runtime	σ ² Runtime
1	Expected	N/A	N/A	189.60	88.03
2	Expected	N/A	N/A	386.82	148.21
3	Expected	14.81	18.39	293.06	78.20
4	Expected	N/A	N/A	306.50	6.96
5	Expected	8.13	8.76	370.52	15.95
6	Expected	7.01	7.20	411.41	52.90
7	Expected	8.28	9.06	364.93	16.57

Table 6.2: XSecEngine: Prototype API Test Results (values in seconds)

¹Time Till First Error detected

²Standard Deviation

Test ID	Result	Average Runtime	σ^2 Runtime
1	Expected	178.67	14.12
2	Expected	193.56	4.20
4	Expected	198.75	4.03
6	No Errors	197.12	8.90
7	Expected	185.60	10.94

Table 6.3: RESTler: Prototype API Test Results (values in seconds)

All expectations for our implementation were met as shown in Table 6.2; moreover, we can verify that having the `x-auth` present in the schema (enhanced schemas) contributes to detecting violations, except for the rule *Immutability in Subsequent GET Requests* which does not require any OAS extensions; this conclusion is trivial since the implementation depends on the schema extensions, nevertheless it validates **RQ1**, since it's proven that an OpenAPI Specification extension can help document security properties that require to be enforced by the system. Additionally, the *Average Time Till First Error* helps answering **RQ3**, since the duration from the start of the test execution until the first violation is reported is, in the worst case, $AverageTTFE + \sigma TTFE$, $14.81 + 18.39$ seconds, in conclusion, in the worst case scenario, it took 33.20 seconds to report the first violation, a value that we acknowledge being very small, as some testing tools, e.g., RESTler [4] would be left running for days to detect errors.

When comparing the values in Table 6.2 to the ones in Table 6.3, we can verify that the average test scenario durations are not very different. This is definitely a good indicator, but we would need to implement active checkers [3] for the properties we defined in order to have a more in-depth comparison between our extension to Schemathesis and the enhanced RESTler, namely the *Time Till First Error* values, provided the efficacies in detecting violations are the same.

6.2 FusionAuth

Having just our own prototype tested seemed insufficient to properly validate our tool's capabilities, and in light of that, we also tested FusionAuth [13] after trying to collect multiple production APIs available as Open-Source projects; unfortunately, we had difficulties finding a suitable API to test, for several reasons: lack of OpenAPI schema, lack of authorization/authentication mechanisms, not correctly implementing RESTful practices, etc. (to be better explained in the following sections).

In Table 6.4, we have defined three test cases for FusionAuth, using *XSecEngine*, as well as the expected errors to be detected. The column *Schema Details* entails the description of the OAS schema modifications/enhancements done; since we did not have access to the source code, and thus, not being able to inject vulnerabilities/bugs in the system (as we did for our prototype), we took a different approach: by changing the system's specifications, inserting constraints that were not initially present:

1. **Enhanced GET user** — we have inserted the `x-auth` property in `GET /api/user/:userId` in a way that only a valid token would be accepted for the endpoint, i.e., only the owner for the resource, or an administrator, would be able to fetch the user's data. This would imply that users with different claims would not be able to make this request, something that is allowed by FusionAuth under certain scenarios: FusionAuth allows for user data manipulation by users with `manage` permissions, indicating that they are not necessarily an administrator or the owner of the resource;
2. **Enhanced PATCH user** — similarly, we have inserted the `x-auth` property in `PATCH /api/user/:userId` so that only users with ownership to that resource would be able to edit that user's information, once again, this is not something FusionAuth requires.

These two enhancements further extend FusionAuth's specifications beyond its original configuration. However, by counterexample, it helps us validate the implementation by detecting an error documented, even though the system does not consider it an error.

Test ID	Schema Details	Affected Endpoints	Expected Errors
1	Default	N/A	None
2	Enhanced GET user	GET <code>/api/user/:userId</code>	X_AUTH_VIOLATION
3	Enhanced PATCH user	PATCH <code>/api/user/:userId</code>	X_AUTH_VIOLATION

Table 6.4: FusionAuth API Test Scenarios

FusionAuth [13] defines 279 endpoints in its schema, to reduce the search space into a minimal reproducible example, we have added links between 6 endpoints, as illustrated in Figure 6.2.

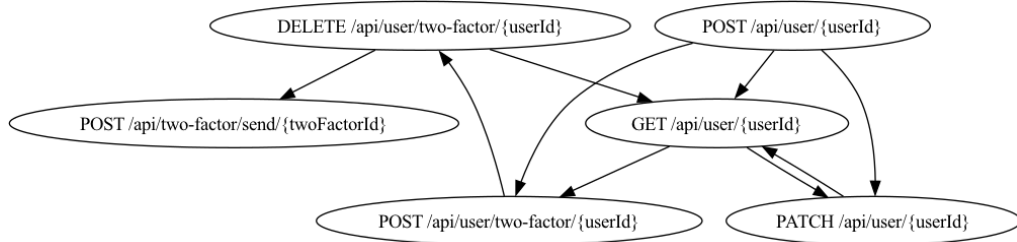


Figure 6.2: FusionAuth Links Graph

For FusionAuth, we did not compare our tool against RESTler [4] since it would not provide any new insights relevant to our study: we had already established that the original RESTler implementation was not able to detect the properties we were targetting, following the experimentation detailed in the previous section. The results can be observed in Table 6.5. On the bright side, our tool met our expectations, as defined in Table 6.4, detecting, with relative ease, the violation in Test Case 2; as we can see, Test Case 3 had the highest *Average Time Till First Error* ever detected: we concluded that 1) it takes more steps to reach the `PATCH` method, than to reach the `GET`, for `/api/user/:userId`, and thus, delaying the capture of errors; 2) the payload for the `PATCH`

method is very complex, which takes Hypothesis longer to generate valid input data in order to be successfully processed.

Test ID	Result	Average TTFE	σ TTFE	Average Runtime	σ Runtime
1	Expected	N/A	N/A	374.11	17.06
2	Expected	5.19	5.34	20.05	3.87
3	Expected	748.82	725.61	10798.93	12089.78

Table 6.5: XSecEngine: FusionAuth API Test Results (values in seconds)

Unfortunately, we were unable to capture violations to other properties, defined in Section 2.4, by testing FusionAuth in an uninterrupted testing environment; however, it helped validate our tool by extending the original set of constraints originally defined so our tool would find violations. In order to synthetically capture violations to the remaining properties, we have paused the checkers we have implemented in crucial points, e.g., when checking if two GET requests return identical responses, we paused the checker after the first request, then proceed to make changes to the target resource, and, finally, unpause the checker: this would throw an error, indicating the response changed from one GET request to the following. We were successful in asserting the efficacy of the remaining three checkers, but since the checkers were being paused and resumed, we felt that time measures would not be appropriate.

To conclude, FusionAuth contributed to validating the answers provided to research questions **RQ1** and **RQ3** in the previous section: the extension to the OAS successfully helped to document and encapsulate security properties, and our tool is efficient in detecting violations to the properties defined.

6.3 Discussion of Results

The evaluations carried out, and their respective outcomes form the backbone of our understanding concerning the research questions delineated in Chapter 1. While individual results and their correlations with specific research questions, such as **RQ1** and **RQ3**, have been examined in isolation, a holistic discussion is vital to glean a broader perspective.

Revisiting Question **RQ1**, *Does the OpenAPI Specification offer avenues for extending its purview to encapsulate security properties, transcending its conventional security schema?*, our answer is bifurcated. Initially, the native OpenAPI Specification permits the inclusion of supplementary fields within a schema. Subsequently, our endeavors yielded results where we defined access control policies using either JWT claims or predetermined values through the `x-auth` property. Not only does this property serve as a direct marker, but it also lays the foundation for information flow security evaluations via the `mutators` object. Conclusively, Question **RQ1** has been affirmatively addressed.

Next, Question **RQ2**, *What's the feasibility and complexity of enhancing an existing testing tool to encompass a broader set of security properties?*, necessitates an analysis of our tool's intricacy. An early observation pinpointed the concise length of the Schemathesis extension, not exceeding a thousand lines, attributable to Python's succinctness and clarity. Python's nature fosters efficient coding, potentially minimizing the code footprint for distinct tasks. Therefore, the essence of Question **RQ2** is that extending a tool for augmented security characteristics is indeed feasible. However, the complexity varies with factors such as the tool's architecture, the language employed, and its operational paradigm. Extending inherently stateless tools might pose challenges, especially when they neither account for nor support request sequences. Delving deeper into Schemathesis, while its documentation enlightens on generic applications, the actual process demanded an intricate understanding of its workflow with its Hypothesis core. Grasping how the rules were formulated and their collaborative function was an arduous journey.

Lastly, Question **RQ3**, *How proficiently do the expanded OpenAPI and testing tool configurations discern security breaches?*, is tackled by evaluating the results derived from our assessment. Our prototype Web API adeptly identified vulnerabilities pertinent to the properties described in Section 2.4. With FusionAuth, set in a practical scenario, detecting transgressions using the standard schema was not realized. Consequently, we integrated supplementary semantic logic to test our tool's competence. Our techniques, combined with interruptible tests, effectively gauged our tool's acumen in identifying property breaches. However, for a comprehensive evaluation, more Web API tests are essential, offering a clearer lens to assess our tool's capabilities.

Subsequent sections delve into the challenges tied to designing test scenarios for diverse Web APIs and spotlight our tool's constraints. This will pave the way for potential advancements in the future.

6.4 Test Scenario Implementation Complexity

When evaluating *XSecEngine*, it's crucial to address the complexities encountered during its application. Contrary to Schemathesis [17], which typically operates seamlessly without additional configurations, *XSecEngine* demands meticulous adjustments in both the OpenAPI schema and the overarching testing workflow.

One of the primary considerations is the identification of the existing authentication and authorization protocols. This understanding shapes the direction of the tests. In instances using OAuth 2.0 (JWT), the key necessities are the secret value, which the system leverages to sign the token for verifying its authenticity, and the algorithm that encrypts it. An alternative solution, especially relevant when testing FusionAuth, is the use of plaintext API tokens serialized in JSON. Such tokens are structured as strings that can be parsed into a JSON value. To illustrate:

```
"{"email": "user@example.com", "scopes": ["read:users"]}"
```

This representation effectively simulates the *payload* of a JSON Web Token, encapsulating the user's claims and scopes.

Beyond authentication, the integrity and richness of the OpenAPI schema play a pivotal role. It is imperative that the schema accurately represents the possible input data, response formats, and, notably, the *links* delineating the operational flow within the application. As an example, if there's a provision to create a resource and subsequently retrieve it using an ID, the schema should manifest a link between these operations. Such a connection implies that data output from the creation can serve as an input for the retrieval process.

Once these foundational requirements are met, *XSecEngine* introduces its proprietary extension, `x-auth`. This extension specifies the security constraints for endpoints. In other words, it clarifies whether specific claims or roles are essential to access them. Moreover, it also offers insights into the *mutator* requests for each endpoint, clarifying the relationships between different methods, like fetching and updating a resource using the same ID.

Concluding the setup, *XSecEngine* demands a customized implementation of the testing workflow specific to the Web API under scrutiny. This workflow, segmented into the *setup*, *initialization*, and *teardown* phases, ensures that the system remains consistent across sequential test runs. Detailed insights into this aspect, as touched upon in Section 5.3, can be explored in Appendix A.

Delving into the process of extending Schemathesis [17], the endeavor proved more intricate than initially forecasted. However, it is worth noting that the complexity we encountered was still below our most pessimistic projections. A prevailing concern had been the potential need to intimately acquaint ourselves with the internal workings of Hypothesis's state machine. This assumption stemmed from the idea that a deep understanding would be vital for effective management and utilization. To our relief, such deep immersion wasn't required. Hypothesis, as we discovered, offers user-friendly interfaces, especially through the `rule` decorator, which alleviates the intricacies of navigating its state machine.

Drawing a comparison between Schemathesis and RESTler brings forth interesting observations. Schemathesis, in its original design, does not inherently validate security properties. In comparison, RESTler, in its first implementation, did not check for security properties, and it was later that the authors implemented *active checkers* for security properties in an independent manner. However, adapting Schemathesis was not an onerous task. We successfully incorporated four custom checkers to validate each of the properties delineated in Section 2.4.

Conversely, RESTler, with its integrated four *active checkers* for properties predefined by its developers, suggests — at least in theory — a potentially smoother adaptation process for additional security properties. However, it is essential to tread with caution here. Theoretical assumptions can be deceptive. Factors like the inherent complexity of the software, its underlying architecture, and the programming language can dramatically influence the intricacy of any extension process.

In conclusion, we assessed the feasibility of our project through the successful extension of Schemathesis. The complexity ranged from moderate to relatively high, a perception magnified by the lack of any formal documentation on extending Schemathesis in the manner we approached.

This forced us into the trenches of its source code, demanding an exploration of its core mechanics and its intricate relationship with Hypothesis.

6.5 Limitations

Our implementation presents a series of limitations that must be thoroughly understood for a comprehensive assessment of its utility and scope. Primarily, our tool’s efficacy is significantly tethered to the specific architecture of the target Web API. For the tool to operate efficiently, the Web API must come equipped with a valid OpenAPI schema that not only exists but also demonstrates valid linkages between operations. This necessitates the Web API to incorporate solid authentication and authorization mechanisms to align seamlessly with our tool’s functionalities.

Another pivotal limitation is the authentication paradigm support. Currently, our tool predominantly supports serialized JSON values and signed JWTs (JSON Web Tokens). This focus considerably narrows the range of Web APIs our tool can effectively engage with, given the prevalent use of diverse API tokens in contemporary systems. The result is a potential restriction in practical, real-world application scenarios for our tool, especially when moving beyond strictly academic or experimental contexts. However, we consider this a technical limitation that can be overcome by extending the supported authentication paradigms and not necessarily a research limitation.

Furthermore, our project is intrinsically tied to Schemathesis [17] and, by extension, Hypothesis [21]. As the core components of our extension, any limitations or constraints associated with these frameworks invariably ripple into our own work. A notable implication of this relationship is the interplay between the efficiency of input data generation and the depth and detail of the OpenAPI schema. A detailed schema can potentially boost the efficiency of our tool, indicating that the effectiveness of our tool is deeply interwoven with the robustness of the OpenAPI schema it interacts with.

Finally, testing RESTful Web APIs in a black-box environment is a true challenge: we needed to account for the statefulness part of the environment, i.e., making sure transitions between states account for state changes in the Web API, as well as making sure that the test scenarios always execute with the same assumptions and initial values.

The cumulative impact of these limitations was a constrained pool of testing candidates, limiting the breadth and depth of our evaluations, especially in real-world, production-ready contexts. Consequently, the general applicability and broad-scale relevance of our findings may be limited, and interpretations should be made with this context in mind.

In essence, while our research and developed tool contribute significantly to Web API interaction and testing, it is crucial to recognize and address these limitations. Future work might pivot towards enhancing adaptability, ensuring compatibility with a broader range of Web API architectures, and expanding the evaluation scope (more on future work in the next Chapter).

Chapter 7

Conclusions

Throughout the course of this thesis, we embarked on a journey into the intricate realm of Web API security, attempting to push the boundaries of contemporary tools and methodologies. The development and evaluation of *XSecEngine* not only stand as a testament to our efforts in enhancing Web API interaction and testing but also paints a vivid picture of the challenges and complexities inherent in such undertakings.

From our initial introduction, where the profound need for robust security measures in the context of Web APIs was emphasized, to our deep dive into the evaluative chapters, a recurrent theme has been the delicate balance between adaptability, effectiveness, and complexity. The digital landscape is ever-evolving, with security challenges mounting as technologies advance. In this dynamic milieu, tools that can adapt to emerging threats while maintaining their efficiency are of paramount importance.

The evaluation of *XSecEngine* provided a nuanced perspective on this balance. While the tool made significant strides in Web API interaction and testing, the associated complexities and limitations also highlighted areas ripe for improvement and future exploration. Particularly notable was the role of documentation and the importance of a tool's foundational architecture in influencing its extensibility.

Moreover, our exploration with Schemathesis and RESTler offered invaluable insights into tool development and adaptation. While both tools possessed their unique strengths and limitations, their juxtaposition underscored the importance of foresight in tool design, emphasizing the need for modularity and adaptability. It became evident that the ability of a tool to adapt to unforeseen challenges, especially in a domain as unpredictable as security, is as crucial as its inherent functionalities.

In summation, this thesis is more than just an exploration of a novel tool or an evaluation of its efficacy. It is a reflection on the broader challenges and opportunities present in the field of Web API security. As digital interactions continue to expand and technologies continue to evolve, the lessons gleaned from this endeavor will remain pertinent. The future of Web API security, while promising, is fraught with challenges, necessitating a continuous commitment to innovation, adaptability, and meticulous evaluation. As we conclude, the hope is that *XSecEngine* and the

insights derived from its journey will serve as a beacon, guiding and inspiring future endeavors in the quest for a more secure and reliable digital realm.

7.1 Future Work

The landscape of Web API security is in constant flux, and the insights derived from our exploration with *XSecEngine* and its evaluation offer several potential avenues for future research and development. Here, we outline some promising directions that might significantly bolster the capabilities of our tool and further our understanding of Web API interaction and testing.

1. **Broader Data Format Support** — As identified in our limitations, the current version of *XSecEngine* predominantly supports serialized JSON values and signed JWTs. There is an immediate need to diversify this support to cater to a wider range of data formats and API tokens, broadening the tool's applicability across different systems and technologies.
2. **Broader Set of Security Properties** — There is a large number of security properties still left to implement; based on the CWE database, one possible improvement point would be to extend the number and diversity of security properties supported. This would likely involve more OAS extensions to guide the testing procedures and, ultimately, the Web API documentation scope.
3. **Evaluate False-Positives** — We also see a need to assess how false-positives impact our tool's effectiveness in detecting violations to the properties we have defined, even more so if we are to extend the set of security properties supported.
4. **Enhanced Authentication Mechanisms** — The integration of more sophisticated authentication mechanisms, such as OAuth 2.1, OpenID Connect, and even biometric-based authentication methods, could not only increase the scope of APIs the tool can interact with but also provide a comprehensive testing ground for next-generation Web APIs.
5. **Automatic Schema Enrichment** — One potential enhancement could be the automatic enrichment or correction of OpenAPI schemas, ensuring they possess the necessary linkages between operations and offer a holistic representation of the application's operational flow.
6. **Tool Integration** — The possibility of integrating *XSecEngine* with other security testing tools could amplify its capabilities manifold. By tapping into the strengths of multiple tools, we can envision a robust ecosystem that offers comprehensive security testing.
7. **Machine Learning-based Anomaly Detection** — Incorporating machine learning algorithms to detect anomalies or unusual patterns during Web API interactions could provide an additional layer of security, flagging potentially malicious or unintentional threats that traditional methods might overlook.

8. **Improved Documentation and User-Friendly Extensions** — Given the complexities faced due to the lack of documentation during our extension of Schemathesis, there is an evident need to ensure that *XSecEngine* is accompanied by thorough documentation. This would facilitate easier extensions, modifications, and contributions by the broader developer community.
9. **Real-world Deployment Scenarios** — Exploring the deployment of *XSecEngine* in real-world production environments would provide invaluable insights into its practical applicability, highlighting areas of improvement that laboratory settings might miss.

In essence, while our journey with *XSecEngine* has offered significant contributions, there remains a vast horizon to explore. Each of these suggested directions not only aims to improve the current iteration of the tool but also underscores the need for continual evolution in the face of ever-evolving security challenges. The future of Web API security, teeming with both opportunities and challenges, awaits our proactive endeavors.

References

- [1] *Allure Framework*. Accessed: January 2023. URL: <https://docs.qameta.io/allure/>.
- [2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. 2nd ed. Cambridge University Press, 2016.
- [3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “Checking Security Properties of Cloud Service REST APIs”. In: *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation, ICST 2020*. Institute of Electrical and Electronics Engineers Inc., Oct. 2020, pp. 387–397. ISBN: 9781728157771. DOI: [10.1109/ICST46399.2020.00046](https://doi.org/10.1109/ICST46399.2020.00046).
- [4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *Proceedings - International Conference on Software Engineering*. Vol. 2019-May. IEEE Computer Society, May 2019, pp. 748–758. ISBN: 9781728108698. DOI: [10.1109/ICSE.2019.00083](https://doi.org/10.1109/ICSE.2019.00083).
- [5] Antonia Bertolino. *Software Testing Research: Achievements, Challenges, Dreams*. Future of Software Engineering (FOSE’07), 2007.
- [6] Nazanin Bayati Chaleshtari et al. *Metamorphic Testing for Web System Security*. 2023. arXiv: [2208.09505](https://arxiv.org/abs/2208.09505) [cs.SE].
- [7] *CWE - About - CWE Overview*. Accessed: January 2023. URL: <https://cwe.mitre.org/about/index.html>.
- [8] *Datadog — Cloud Monitoring as a Service*. Accessed: January 2023. URL: <https://www.datadoghq.com>.
- [9] Dorothy E. Denning. “A Lattice Model of Secure Information Flow”. In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: [10.1145/360051.360056](https://doi.org/10.1145/360051.360056). URL: <https://doi.org/10.1145/360051.360056>.
- [10] *FastAPI*. Accessed: June 2023. URL: <https://fastapi.tiangolo.com>.
- [11] R. Fielding and J. Reschke. *RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 2014.
- [12] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. Irvine: University of California, 2000.

- [13] *FusionAuth: Auth. Built for Devs, by Devs*. Accessed: August 2023. URL: <https://fusionauth.io/>.
- [14] Carlo Ghezzi, Sam Guinea, and Paola Spoletini. “Specification-based testing of Web services”. In: *Test and Analysis of Web Services*. Springer, 2002, pp. 145–166.
- [15] J. A. Goguen and J. Meseguer. “Security policies and security models”. In: *Proceedings of the 1982 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE, 1982.
- [16] Johan Haleby and many other contributors. *REST Assured*. Accessed: January 2023. URL: <https://rest-assured.io>.
- [17] Zac Hatfield-Dodds and Dmitry Dygalo. “Deriving Semantics-Aware Fuzzers from Web API Schemas”. In: (Dec. 2021). DOI: [10.48550/arxiv.2112.10328](https://arxiv.org/abs/2112.10328). URL: <https://arxiv.org/abs/2112.10328>.
- [18] M. Jones and D. Hardt. *RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage*. 2012.
- [19] *JSON Schema | The home of JSON Schema*. Accessed: December 2022. URL: <https://json-schema.org/>.
- [20] Saravanan Kumarasamy. “Distributed Denial of Service (DDOS) Attacks Detection Mechanism”. In: *International Journal of Computer Science, Engineering and Information Technology* 1 (5 Dec. 2011), pp. 39–49. ISSN: 22313605. DOI: [10.5121/ijcseit.2011.1504](https://doi.org/10.5121/ijcseit.2011.1504).
- [21] David R. MacIver. *What is Property Based Testing? - Hypothesis*. Accessed: January 2023. May 2016. URL: <https://hypothesis.works/articles/what-is-property-based-testing/>.
- [22] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (Nov. 2019), p. 1891. ISSN: 2475-9066. DOI: [10.21105/joss.01891](https://doi.org/10.21105/joss.01891). URL: <https://joss.theoj.org/papers/10.21105/joss.01891>.
- [23] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. “Online testing of RESTful APIs: promises and challenges”. In: *ESEC/FSE 2022: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery (ACM), Nov. 2022, pp. 408–420. ISBN: 9781450394130. DOI: [10.1145/3540250.3549144](https://doi.org/10.1145/3540250.3549144). URL: <https://dl.acm.org/doi/10.1145/3540250.3549144>.
- [24] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. “REStest: Automated black-box testing of RESTful web APIs”. In: *ISSTA 2021 - Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, July 2021, pp. 682–685. ISBN: 9781450384599. DOI: [10.1145/3460319.3469082](https://doi.org/10.1145/3460319.3469082). URL: <https://dl.acm.org/doi/10.1145/3460319.3469082>.

- [25] *OpenAPI Initiative*. Accessed: December 2022. URL: <https://www.openapis.org/>.
- [26] *OWASP API Security Project - 2019*. Accessed: January 2023. URL: <https://owasp.org/API-Security/editions/2019/en/0x11-t10/>.
- [27] *OWASP API Security Top Ten - 2023*. Accessed: June 2023. URL: <https://owasp.org/API-Security/editions/2023/en/0x00-header>.
- [28] *OWASP Foundation, the Open Source Foundation for Application Security*. Accessed: January 2023. URL: <https://owasp.org/>.
- [29] *OWASP Top Ten*. Accessed: December 2022. URL: <https://owasp.org/www-project-top-ten/>.
- [30] *Pydantic*. Accessed: July 2023. URL: <https://docs.pydantic.dev/>.
- [31] *RapidAPI Testing*. Accessed: January 2023. URL: <https://rapidapi.com/products/api-testing/>.
- [32] Debra J Richardson and Yashwant K Malaiya. “Specification-based testing with formal methods: A case study”. In: *Proceedings. The Sixth International Conference on Software Engineering and Knowledge Engineering*. IEEE. 1992, pp. 47–54.
- [33] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O’Reilly Media, Inc., 2007.
- [34] *Rule Based Stateful Testing*. Accessed: July 2023. URL: <https://hypothesis.works/articles/rule-based-stateful-testing/>.
- [35] R.S. Sandhu and P. Samarati. “Access control: principle and practice”. In: *IEEE Communications Magazine* 32.9 (Sept. 1994), pp. 40–48. ISSN: 1558-1896. DOI: 10.1109/35.312842.
- [36] *Sauce Labs*. Accessed: January 2023. URL: <https://saucelabs.com>.
- [37] *Schemathesis: Stateful Testing*. Accessed: August 2023. 2023. URL: <https://schemathesis.readthedocs.io/en/stable/stateful.html#stateful-testing>.
- [38] W. Stallings. *Network Security Essentials: Applications and Standards*. Pearson, 2017. ISBN: 9780134527338.
- [39] *Starlette*. Accessed: July 2023. URL: <https://www.starlette.io>.
- [40] Ari Takanen, Jared D. DeMott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.
- [41] Hypothesis Development Team. *Details of Shrinking*. Accessed: August 2023. 2023. URL: <https://hypothesis.readthedocs.io/en/latest/data.html#shrinking>.
- [42] Hypothesis Development Team. *Features that make Hypothesis stand out*. Accessed: August 2023. 2023. URL: <https://hypothesis.works/features/>.
- [43] Hypothesis Development Team. *Stateful Testing*. Accessed: August 2023. 2023. URL: <https://hypothesis.readthedocs.io/en/latest/stateful.html>.

- [44] Schemathesis Development Team. *Using additional Hypothesis strategies*. Accessed: August 2023. 2023. URL: <https://schemathesis.readthedocs.io/en/stable/python.html#using-additional-hypothesis-strategies>.
- [45] Elaine J Weyuker. “Testing finite state machines: A study of methods useful for testing a family of protocols”. In: *SIAM Journal on Computing* 11.3 (1982), pp. 486–496.
- [46] *YAML Revision 1.2.2*. Accessed: January 2023. URL: <https://yaml.org/spec/1.2.2/>.
- [47] Andreas Zeller. “Simplifying and Isolating Failure-Inducing Input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200. URL: <https://www.st.cs.uni-saarland.de/papers/tse2002/tse2002.pdf>.

Appendix A

Test Workflow Implementations

```
1 import requests
2 import schemathesis
3 from hypothesis import HealthCheck, Verbosity, settings
4 from hypothesis.stateful import initialize, multiple
5
6 from app.core.auth import get_token_header_value
7 from app.core.config import config
8 from app.core.evaluation import EvaluationService
9 from app.core.workflow import Workflow
10
11 if __name__ == "__main__":
12     schema = schemathesis.from_path(
13         config.OPENAPI_SCHEMA_PATH, base_url=config.BASE_URL
14     )
15
16     evaluation_service = EvaluationService(results_dir=config.RESULTS_DIR)
17
18     workflow = Workflow(evaluation_service=evaluation_service)
19     BaseAPIWorkflow = workflow.get_api_workflow()
20
21     class APIWorkflow(BaseAPIWorkflow):
22         admin_token = None
23
24         def setup(self):
25             def _get_token(username, email, scopes):
26                 response = requests.post(
27                     f"{config.BASE_URL}{config.TOKEN_ENDPOINT}",
28                     json={
29                         "username": username,
30                         "email": email,
31                         "scopes": scopes,
32                     },
33                 )
```

```

34         return response.json().get("access_token", "invalid")
35
36         self.admin_token = _get_token(
37             config.ADMIN_USERNAME, config.ADMIN_EMAIL, [*config.SCOPEES, "admin"
38                 ]
39         )
40         for user in config.USERS:
41             self.contexts.append(
42                 (user, _get_token(user["username"], user["email"], config.
43                     SCOPEES))
44             )
45
46     @initialize(
47         target=BaseAPIWorkflow.bundles["/api/v1/users"] ["POST"],
48     )
49     def init_users(self):
50         result = []
51         for user in config.USERS:
52             case = schema["/api/v1/users"] ["POST"].make_case(body=user)
53             result.append(self.step(case))
54         return multiple(*result)
55
56     def teardown(self):
57         for user in config.USERS:
58             requests.delete(
59                 f"{config.BASE_URL}/api/v1/users/{user['email']}",
60                 headers={
61                     "Authorization": get_token_header_value(token=self.
62                         admin_token),
63                 },
64             )
65         self.admin_token = None
66         self.contexts.clear()
67
68     workflow.print_links()
69     APIWorkflow.TestCase.settings = settings(
70         verbosity=Verbosity.debug,
71         deadline=4000,
72         suppress_health_check=[HealthCheck.filter_too_much],
73         max_examples=1000,
74         stateful_step_count=30,
75     )
76     try:
77         APIWorkflow.run()
78     except Exception as e:
79         print(e)
80
81     evaluation_service.generate_evidence()

```

Listing A.1: Prototype API Test Workflow Implementation

```
1 from json import dumps as json_dumps
2
3 import requests
4 import schemathesis
5 from hypothesis import HealthCheck, Verbosity, settings
6 from hypothesis.stateful import initialize, multiple
7
8 from app.core.config import config
9 from app.core.evaluation import EvaluationService
10 from app.core.workflow import Workflow
11
12 if __name__ == "__main__":
13     schema = schemathesis.from_path(
14         config.OPENAPI_SCHEMA_PATH,
15         base_url=config.BASE_URL,
16     )
17
18     evaluation_service = EvaluationService(results_dir=config.RESULTS_DIR)
19
20     workflow = Workflow(evaluation_service=evaluation_service)
21     BaseAPIWorkflow = workflow.get_api_workflow()
22
23     class APIWorkflow(BaseAPIWorkflow):
24         def _get_token(self, user):
25             response = requests.post(
26                 f"{config.BASE_URL}/api/api-key/{user['keyId']}",
27                 json={
28                     "apiKey": {
29                         "key": json_dumps(
30                             {
31                                 "id": user["id"],
32                                 "email": user["email"],
33                                 "name": user["name"],
34                             }
35                         )
36                     }
37                 },
38                 headers={
39                     "Authorization": config.ADMIN_API_TOKEN,
40                 },
41             )
42             try:
43                 json = response.json()
44             except Exception:
45                 return "invalid"
46
47             return json.get("apiKey", {}).get("key", "invalid")
48
49     def setup(self):
```

```

50         for user in config.USERS:
51             self.contexts.append((user, self._get_token(user)))
52
53     @initialize(
54         target=BaseAPIWorkflow.bundles["/api/user/{userId}"]["POST"],
55     )
56     def init_users(self):
57         result = []
58         for user in config.USERS:
59             case = schema["/api/user/{userId}"]["POST"].make_case(
60                 body={
61                     "user": {
62                         "email": user["email"],
63                         "password": user["password"],
64                     }
65                 },
66                 path_parameters={"userId": user["id"]},
67                 headers={"Authorization": config.ADMIN_API_TOKEN},
68             )
69             result.append(self.step(case))
70         return multiple(*result)
71
72     def teardown(self):
73         for user in config.USERS:
74             print(f"Deleting user {user['id']}...", end="")
75             delete = requests.delete(
76                 f"{config.BASE_URL}/api/user/{user['id']}?hardDelete=true",
77                 headers={
78                     "Authorization": config.ADMIN_API_TOKEN,
79                 },
80             )
81             print(delete.status_code, end=" token: ")
82             delete = requests.delete(
83                 f"{config.BASE_URL}/api/api-key/{user['keyId']}",
84                 headers={
85                     "Authorization": config.ADMIN_API_TOKEN,
86                 },
87             )
88             print(delete.status_code)
89             self.contexts.clear()
90
91     def validate_response(self, response, case):
92         case.validate_response(response, checks=())
93
94 workflow.print_links()
95 APIWorkflow.TestCase.settings = settings(
96     verbosity=Verbosity.debug,
97     deadline=4000,
98     suppress_health_check=[HealthCheck.filter_too_much],

```

```
99         max_examples=1000,  
100         stateful_step_count=30,  
101     )  
102  
103     try:  
104         APIWorkflow.run()  
105     except Exception as e:  
106         print (e)  
107  
108     evaluation_service.generate_evidence()
```

Listing A.2: FusionAuth Test Workflow Implementation

Appendix B

Links Graph Implementation

```
1 from typing import Dict, List
2
3 import matplotlib.pyplot as plt
4 import networkx as nx
5 from hypothesis.stateful import Bundle
6 from hypothesis.strategies._internal.lazy import LazyStrategy
7 from hypothesis.strategies._internal.misc import JustStrategy
8 from hypothesis.strategies._internal.strategies import (FilteredStrategy,
9                                                         MappedSearchStrategy)
10
11
12 class Sequence:
13     def __init__(self, nodes: List[str]):
14         self.nodes = nodes
15
16     def _nodes_to_str(self):
17         return " -> ".join(self.nodes)
18
19     def __repr__(self):
20         return f"Sequence({self._nodes_to_str()})"
21
22     def __eq__(self, other):
23         for node in self.nodes:
24             if node not in other.nodes:
25                 return False
26         return True
27
28     def __hash__(self):
29         return hash(self._nodes_to_str())
30
31     def size(self):
32         return len(self.nodes)
33
```

```

34     def __str__(self):
35         return " -> ".join([f"{node:32}" for node in self.nodes])
36
37
38 class SequenceSet:
39     def __init__(self, sequences: List[Sequence]):
40         self.sequences = list(set(sequences))
41
42     def __repr__(self):
43         return f"SequenceSet({self.sequences})"
44
45     def __str__(self):
46         self.sequences.sort(key=lambda x: -x.size())
47         return "\n".join([str(sequence) for sequence in self.sequences])
48
49
50 class Graph:
51     @classmethod
52     def _get_strategy_name(cls, strategy):
53         if type(strategy) == MappedSearchStrategy:
54             return cls._get_strategy_name(strategy.mapped_strategy)
55         if type(strategy) == Bundle:
56             return strategy.name
57         if type(strategy) == FilteredStrategy:
58             return cls._get_strategy_name(strategy.filtered_strategy)
59         if type(strategy) == LazyStrategy:
60             return cls._get_strategy_name(strategy.wrapped_strategy)
61         if type(strategy) == JustStrategy:
62             return None
63
64         return "Unknown"
65
66     @classmethod
67     def build_from_rules(cls, rules: Dict[type, List[classmethod]] = {}):
68         graph = nx.DiGraph()
69
70         for rule in rules:
71             graph.add_node(rule.targets[0])
72             previous = rule.arguments["previous"].branches
73
74             for branch in previous:
75                 node_name = cls._get_strategy_name(branch)
76                 if node_name:
77                     graph.add_edge(node_name, rule.targets[0])
78
79         return graph
80
81     @classmethod
82     def draw(cls, graph: nx.DiGraph, filename=None):

```

```

83     nx.draw(graph, with_labels=True)
84     if filename:
85         plt.savefig(f"{filename}.png", format="PNG")
86         nx.drawing.nx_pydot.write_dot(graph, f"{filename}.dot")
87     return graph
88
89     @classmethod
90     def calculate_all_sequences(cls, graph: nx.DiGraph) -> SequenceSet:
91         roots = []
92         leaves = []
93         for node in graph.nodes:
94             if graph.in_degree(node) == 0:
95                 roots.append(node)
96             elif graph.out_degree(node) == 0:
97                 leaves.append(node)
98
99         sequences = [
100             Sequence(path)
101             for root in roots
102             for leaf in leaves
103             for path in nx.all_simple_paths(graph, root, leaf)
104         ]
105         subsequences = []
106
107         # for each sequence, we want all sub-sequences except the last node by
108         itself
109         # example: [A, B, C] -> [A, B], [A], [B, C], [B] but not [C]
110         for sequence in sequences:
111             for i in range(1, len(sequence.nodes)):
112                 subsequences.append(Sequence(sequence.nodes[:i]))
113                 subsequences.append(Sequence(sequence.nodes[i:]))
114
115         subsequences = [
116             subsequence
117             for subsequence in subsequences
118             if subsequence.size() > 1 or subsequence.nodes[0] not in leaves
119         ]
120
121     return SequenceSet(sequences + subsequences)

```

Listing B.1: Links Graph Implementation